An introduction to Representational System Theory: Part 2

Daniel Raggi

October 28, 2021

In Part 1 I introduced the formal notion of representational system (three construction spaces), and that of a construction, which is a specific structure formed within a construction space. Here I introduce the notion of *pattern* and *correspondence*, and then give an overview of structure transfer, and rep2rep's input language *oruga*.

1 Patterns

Definition 1. Let (g, t) and (γ, v) be constructions in constructions spaces over the same type system T. We say that (g, t) *matches* (γ, v) if there exists an isomorphism $f : g \to \gamma$ such that

1. t maps to v: $f(t) = v$,
2. the subtype relation is respected by f: for all $t' \in To$, the vertex-labelling functions, *type* and *type*$'$, ensure that $type(t') \leq type'(f(t'))$,
3. the constructors assigned to configurators match: for all $u \in Cr$, $co(u) = co'(f(u))$, and
4. the arrow indices are identical: for all $a \in A$, the arrow-labelling functions, *index* and *index*$'$, ensure that $index(a) = index'(f(a))$.

Such an isomorphism is called an *embedding*.

So there really is no difference between constructions and patterns other than in relation to each other. Generally when we talk about a construction that matches a pattern, the construction is taken from some specific representational system but the pattern is not.

In general, when we have a construction space $\mathcal{C}$, we can talk about *a pattern for* $\mathcal{C}$. When we do so we don't mean that the pattern lives in $\mathcal{C}$, but that there may be constructions in the construction space which match the pattern. Basically, the 'tokens' of a pattern for $\mathcal{C}$ are not necessarily tokens in $\mathcal{C}$.

The next definition simply connects the notion of decomposition and the notion of pattern-matching.

Definition 2. Let (g, t) be a construction that matches a pattern (γ, v). Let D be a decomposition of (γ, v). Then we say that D *describes* (g, t).

For structure transfer, there's an implicit decomposition of the construction for which we are seeking a transformation. But more importantly, the result of the structure transfer algorithm is a pattern decomposition of the sought-after construction. This means that the construction we are looking for might not be apparent, or even exist; but if it exists then the decomposition we got is a description of it.

2 Correspondences

Correspondences are the basic links between representational systems. They allow us to transform constructions.

A correspondence will be defined as a tuple, $((\gamma, v), (\gamma', v'), L, L', R, S)$, where (γ, v) and (γ', v') patterns, L and L' are sequences of tokens taken from the patterns, and R and S are relations. The idea is that two patterns are in such a correspondence if whenever they are instantiated (with concrete tokens) and R holds (for tokens which 'embed' into L and L') then S must also hold.

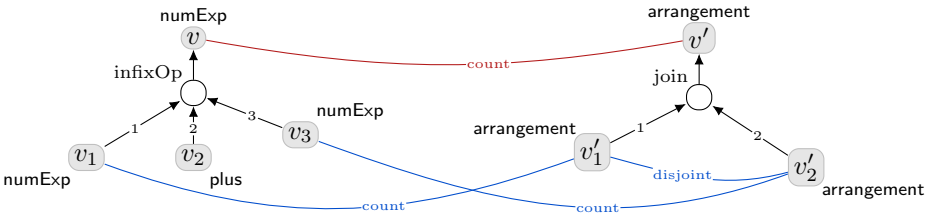
Definition 3. Let $\mathcal{C}$ and $\mathcal{C}'$ be construction spaces, and let (γ, v) and (γ', v') be patterns. Let $L = [v_1, \ldots, v_n]$ and $L' = [v'_1, \ldots, v'_m]$ be sequences of tokens taken from γ and γ', respectively. Let R be a relation of arity $n + m$ and let S be a binary relation. Then, the tuple $((\gamma, v), (\gamma', v'), L, L', R, S)$ is a *correspondence* for $\mathcal{C}$ and $\mathcal{C}'$ if for any constructions (g, t) and (g', t') in $\mathcal{C}$ and $\mathcal{C}'$, whenever

1. (g, t) matches (γ, v) with embedding f, and
2. (g', t') matches (γ', v') with embedding f', and
3. $(f^{-1}(v_1), \ldots, f^{-1}(v_n), f'^{-1}(v'_1), \ldots, f'^{-1}(v'_m)) \in R$ holds

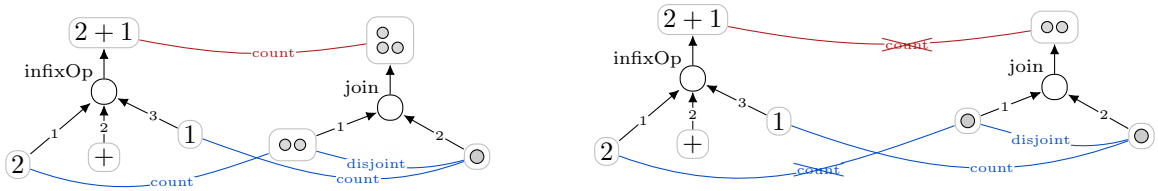
then $(f^{-1}(v), f'^{-1}(v')) \in S$ (or equivalently $(t, t') \in S$) also holds.

Given such a correspondence, (γ, v) is called the *source pattern*, (γ', v') is called the *target pattern*, R is called the *token relation*, and S is called the *construct relation*.

Take, for example, a correspondence $((\gamma, v), (\gamma', v'), L, L', R, S)$, where (γ, v) is the pattern below (left), (γ', v') is the pattern bellow (right), $L = [v_1, v_3]$ and $L' = [v'_1, v'_2]$, and R is the relation captured by the conjunction of the blue edges, and S is the relation captured by the red edge.



Now, below see instantiations of the patterns, one in which the token relation holds, and another in which the token relation doesn't hold. The above is indeed a correspondence because whenever the token relation holds the construct relation will necessarily hold (as demonstrated on the left).



In practice, a correspondence is best declared as a conjunction of relations. For example, the above R is easily expressed as the conjunction:

$$(t_1, t_3, t'_1, t'_2) \in R \quad \equiv \quad (t_1, t'_1) \in \text{count} \ \wedge \ (t_3, t'_2) \in \text{count} \ \wedge \ (t'_1, t'_2) \in \text{disjoint}.$$

See below the type of correspondences, as implemented[1]:

```
type corr = {name : string,
             sourcePattern : Pattern.construction,
             targetPattern : Pattern.construction,
             tokenRels : Relation.relationship list,
             constructRel : Relation.relationship};
```

A relationship is, essentially, just an encoding of statements such as $(x, y) \in R$, with explicit names referring to the tokens in the source and target patterns. The relationship list should be interpreted as a conjunction of various relationships.

Given the implicational nature of correspondences they can be used as inference rules. This is exploited in the structure transfer algorithm.

3 Structure transfer

Structure transfer is the algorithm by which we transform one construction in a representational system into a composition in another representational system. It is essentially an inference mechanism (with correspondences as inference rules) which tries to find a way to satisfy some given relation while using information from the correspondences to build a composition which witnesses the relation being satisfied.

Next I will list a few auxiliary concepts to understand structure transfer.

[1] Note this is a *record* type.

Relations and Relationships If R is a *relation*, the triple (x, y, R) is called a *relationship*, meant to represent the statement $(x, y) \in R$. In the implementation, all relationships are of the form

$$([t_1, \ldots, t_n], [t'_1, \ldots, t'_m], R),$$

where every t_i is a token from *source* construction space, and every t'_i is a token from the *target* construction space. By source I mean the construction space from where we take our original construction, and by target I meant he construction space to which we want to transform.

Goal A *goal* is a relationship that we want to satisfy. For structure transfer, we start with a goal, $([t], [v], R)$, where t refers to a token in a given construction and v represents an unknown token we want to find. This goal will be further broken down into subgoals, and these will be broken down into subgoals (or satisfied) and so on. We refer to the goals that have not been satisfied as *open goals*.

Knowledge base A knowledge base consists of a set of correspondences, and a *sub-relation function*, which tells you whether some relations are included in others (e.g., the relation $=$ is included in the relation $\leq$). See below the type of a knowledge base, as implemented:

```
type base = {subRelation : Relation.T * Relation.T -> bool,
             correspondences : Correspondence.corr Seq.seq};
```

`subRelation`, where `subRelation`(R_1, R_2) = `true` if $R_1 \subseteq R_2$. The default is that `subRelation`(R_1, R_2) will output true only if R_1 equals R_2, but in principle this knowledge can be augmented.

State Given that structure transfer relies on graph search, we need to define the possible states from and into which the algorithm moves. The type of a state is implemented as follows:

```
type T = {sourceTypeSystem : Type.typeSystem,
          targetTypeSystem : Type.typeSystem,
          transferProof : TransferProof.tproof,
          construction : Construction.construction,
          originalGoal : Relation.relationship,
          goals : Relation.relationship list,
          composition : Composition.composition,
          knowledge : Knowledge.base};
```

Most of this is self-explanatory. In every step of structure transfer the `composition`, the `goals` and the `transferProof`[2] will be updated. The rest remains the same.

Search and heuristic A *heuristic* is any function that, given two states, tells you which one is more desirable. The current implementation uses a depth-first search, keeping all the intermediate steps, and in the end orders the results according to the heuristic.

3.1 The structure transfer algorithm

The main function for understanding what structure transfer does is `applyCorrespondenceForGoal`. This function is the single-step state change (walking exactly one edge in graph search).

Let us analyse what the function application `applyCorrespondenceForGoal st corr goal` would do. If the following holds:

1. `goal` is of the form $([t_1], [t'_1], R_1)$, and
2. `corr` is a correspondence whose construct relationship[3] is $([t_2], [t'_2], R_2)$ where $R_2 \subseteq R_1$ and $type(t_1) \leq type(t_2)$ and $t' = \min(t'_1, t'_2)$ (assuming the minimum type exists), and
3. the source pattern of `corr` matches the relevant part of the original construction,

then `applyCorrespondenceForGoal st corr goal` will return a state that modifies the following from `st`:

[2] The `transferProof` structure keeps every step of the 'proof' of the goal from the open goals.

[3] By *construct relationship* of a correspondence we mean the construct relation applied to the constructs.

1. replace the goal $([t_1], [t'_1], R_1)$ with the list of token relationships of the correspondence,
2. attach the target pattern of the correspondence to t'_1 in the composition.

Structure transfer uses `applyCorrespondenceForGoal`, trying all combinations of the given correspondences (from the knowledge base) and the current open goals, to traverse the search graph until either it cannot apply more correspondences to the given goals, or some given limit has been exceeded.

4 The oruga language

The input language for the rep2rep implementation is called *oruga*, after the Spanish word for *caterpillar* (the input of a transformational process!).

The purpose of the language is to declare all the relevant rep2rep structures and to call the structure transfer algorithm.

The language is very simple, and the parser is as simple as can be. It relies on a set of global keywords, which are: `typeSystem`, `conSpec` (constructor specification), `construction`, `correspondence`, `import`, and `transfer`. Whatever follows one keyword (and before the next keyword is found) is parsed as a local environment with its own local keywords.

When a *document* string is parsed it is converted to a structure of the following type:

```
type documentContent = {typeSystems : Type.typeSystem list,
                        conSpecs : CSpace.conSpec list,
                        knowledge : Knowledge.base,
                        constructions : {name : string,
                                         conSpec : string,
                                         construction : Construction.construction} list,
                        transferRequests : (string list) list,
                        strengths : string -> real option}
```

As we will see, most of the declarations have the shape ⟨`globalKeyword`⟩ ⟨`name`⟩ = ⟨`content`⟩.

4.1 Type systems

The local keywords for the `typeSystem` environment are `types` and `order`. A type system declaration looks as follows (for a simple arithmetic algebra):

```
typeSystem arith =
    types _:numeral, _:var, _:numExp, _:formula, plus, minus,
          times, div, binOp, equal, binRel, oB, cB, par
    order var < numExp,
          numeral < numExp,
          plus < binOp,
          minus < binOp,
          times < binOp,
          equal < binRel,
          oB < par,
          cB < par
```

The content after the keyword `types` should be a list separated by commas. Most will just stand for the names of types, but the ones with shape $\_ : \langle\tau\rangle$ indicate that type $\langle\tau\rangle$ has a bunch of subtypes that don't need to be declared individually. In practice, it means that, later, we can declare a token such as $1 + 2 : \texttt{1plus2} : \texttt{numExp}$. And this should be interpreted as saying that $1 + 2$ is a token of type `1plus2`, which is of subtype `numExp` (and this can be known without declaring type `1plus2` explicitly).

The parser for type systems computes the reflexive and transitive closure. This ensures it will be an order.

4.2 Constructor specifications

A construction specification is declared with the following form:

`conSpec` ⟨`name`⟩ : ⟨`typeSystem name`⟩ = ⟨`content`⟩.

The following example declares a the constructor specification for the grammatical space of the type system `arith`:

```
conSpec arithG:arith =
    infixOp : [numExp,binOp,numExp] -> numExp,
    infixRel : [numExp,binRel,numExp] -> formula,
    frac : [numExp,div,numExp] -> numExp,
    implicitMult : [numExp,numExp] -> numExp,
    addParentheses : [oB,numExp,cB] -> numExp
```

Each constructor is declared along with its signature. It should be self-evident how the above is interpreted as constructors with their signatures.

4.3 Constructions

A construction is declared with the following form: `construction` ⟨`name`⟩ : ⟨`conSpec name`⟩ = ⟨`content`⟩. The recursive form of a construction is either a trivial construction (with no constructors):

$$\langle \texttt{token} \rangle : \langle \texttt{type} \rangle$$

or

$$\langle \texttt{token} \rangle : \langle \texttt{type} \rangle \texttt{ <- } \langle \texttt{constructor} \rangle [\langle \texttt{construction}_1 \rangle, \ldots, \langle \texttt{construction}_n \rangle].$$

A simple construction of $1 + 2$ can be declared as:

```
construction 1plus2:arithG =
    t:1plus2:numExp <- infixOp[t1:1:numeral, t2:plus, t3:2:numeral]
```

Note that the names we give to the tokens is irrelevant, as long as they only get the same name if they are the same token.

A construction of $1 + 2 + 3 + 4 = \frac{4(4+1)}{2}$ can be declared as follows:

```
construction 1plus2plus3plus4equalsStuff:arithG =
    t:1plus2plus3plus4equal4oB4plus1cBdiv2:formula
        <- infixRel[t1:1plus2plus3plus4:numExp
                    <- infixOp[t11:1plus2plus3:numExp
                                <- infixOp[t111:1plus2:numExp
                                            <- infixOp[t1111:1:numeral,
                                                       t1112:plus,
                                                       t1113:2:numeral],
                                           t112:plus,
                                           t113:3:numeral],
                               t12:plus,
                               t13:4:numeral],
                   t2:equal,
                   t3:4oB4plus1cBdiv2:numExp
                       <- frac[t31:4oB4plus1cB:numExp
                                <- implicitMult[t311:4:numeral,
                                                t312:oB3plus1cB:numExp
                                                    <- addParentheses[t3121:oB,
                                                                      t3122:4plus1:numExp
                                                                            <- infixOp[t31221:4:numeral,
                                                                                       t31222:plus,
                                                                                       t31223:1:numeral],
                                                                      t3123:cB]],
                               t32:div,
                               t33:2:numeral]]
```

4.4 Correspondences

The `correspondence` environment has 5 local keywords: `source`, `target`, `tokenRels`, `constructRel`, `strength`. The general form of a correspondence declaration is

$$\begin{aligned}&\texttt{correspondence } \langle \texttt{name} \rangle : \big(\langle \texttt{source conSpec name} \rangle, \langle \texttt{target conSpec name} \rangle\big) = \\ &\quad \texttt{source } \langle \texttt{source pattern} \rangle \\ &\quad \texttt{target } \langle \texttt{target pattern} \rangle \\ &\quad \texttt{tokenRels } \langle \texttt{token relationships} \rangle \\ &\quad \texttt{constructRel } \langle \texttt{construct relationship} \rangle \\ &\quad \texttt{strength } \langle \texttt{real number} \rangle\end{aligned}$$

Naturally, source and target patterns are written simply as constructions. Strength is meant to capture fuzzy implication or uncertainty. Use value 1.0 for when the implication of the construct relationship by the token relationship is universally true and known.

See the following correspondence declaration, for the analogy between plus (in arithmetic) and join (in dot diagrams).

```
correspondence plusJoin:(arithG,dotDiagramsG) =
    source t:numExp <- infixOp[n:numExp,p:plus,m:numExp]
    target t':arr <- join[a:arr,b:arr]
    tokenRels ([n:numExp],[a:arr]) :: count,
              ([m:numExp],[b:arr]) :: count,
              ([],[a:arr,b:arr]) :: disjoint
    constructRel ([t:numExp],[t':arr]) :: count
    strength 1.0
```

Note that, in a relationship declaration `::` should be interpreted as $\in$.

4.5 Import

Import simply takes a file name in the rep2rep folder input/$\cdots$ and appends the contents of that document to the current one:

```
import correspondences/arithDots
```

4.6 Transfer requests

Anything under the global keyword `transfer` is a *request* to apply structure transfer to some construction.

The local keywords of the `transfer` environment are `sourceConstruction`, `targetTypeSystem`, `goal`, `output`, and `limit`. The general form of a transfer request is:

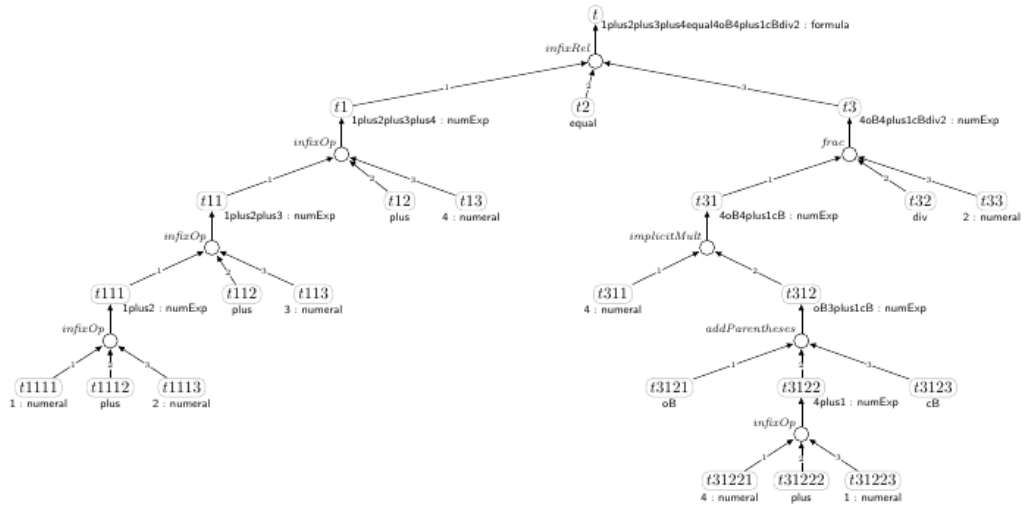
```
transfer
    sourceConstruction ⟨construction name⟩
    targetTypeSystem ⟨typeSystem name⟩
    goal ⟨relationship⟩
    output ⟨filename⟩
    limit ⟨positive integer⟩
```

When the document reader parses transfer requests, it asks structure transfer to transform the construction inputted to `sourceConstruction` to something in the type system inputted to `targetTypeSystem` that satisfies the goal inputted to `goal`. The results (limited to the number inputted to `limit`) will be written into a LaTeX document ⟨`filename`⟩`.tex` in the folder `output/latex/`. A transfer request looks as follows:

```
transfer
    sourceConstruction 1plus2plus3plus4equalsStuff
    targetTypeSystem dotDiagrams
    goal ([t:1plus2plus3plus4equal4oB4plus1cBdiv2:formula],[t':arr]) :: formulaIsValid
    output test
    limit 10
```

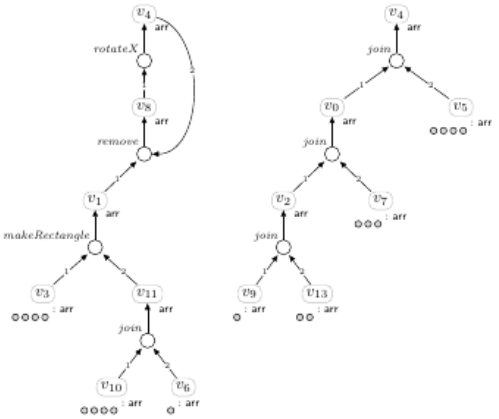
See below a sample of a pdf file, compiled from the automatically generated tex file from such a transfer request:

Original construction



Structure transfer results

Result 1



Original goal

$([t : \text{1plus2plus3plus4equal4oB4plus1cBdiv2 : formula}], [v_4 : \text{arr}]) \in \text{formulaIsValid}$

Open goals

$([], [v_9 : \circ : \text{arr}, v_{13} : \circ\circ : \text{arr}]) \in \text{disjoint}$

$([], [v_2 : \text{arr}, v_7 : \circ\circ\circ : \text{arr}]) \in \text{disjoint}$

$([], [v_0 : \text{arr}, v_5 : \circ\circ\circ\circ : \text{arr}]) \in \text{disjoint}$

$([], [v_{10} : \circ\circ\circ\circ : \text{arr}, v_6 : \circ : \text{arr}]) \in \text{disjoint}$

$([], [v_3 : \circ\circ\circ\circ : \text{arr}, v_{11} : \text{arr}]) \in \text{canRectangulate}$

IS score

0.03125

Result 2

Original goal

$([t : \text{1plus2plus3plus4equal4oB4plus1cBdiv2 : formula}], [v_4 : \text{arr}]) \in \text{formulaIsValid}$

5 Summary

In Part 1 I introduced representational systems, constructions, and associated concepts. In this note I introduced patterns, correspondences, structure transfer, and the syntax oruga, the input language for the implementation. To get started using it, visit: `https://github.com/danielraggi/rep2rep`.