

Modelling Shape Languages with Type Constraint Systems

Dietrich Bollmann

Department of Architecture, The University of Tokyo, 4-6-1 Komaba, Meguro-ku, Tokyo 153-8505, Japan
E-mail address: dietrich@formgames.org

(Received August 9, 2011; Accepted November 14, 2011)

Shape grammars are computational production systems used in various fields such as painting, sculpture and architecture for generating geometric shapes from a set of abstract rules. While similar to formal grammars as used in linguistics and computer science, they differ in using shapes instead of discrete symbols as representation. This makes them more intuitive and richer in possible interpretations than their symbolic counterparts but also more difficult to implement as computer programs.

Using an example, this paper shows how a shape language can be modelled with a Type Constraint System (TCS), a formalism similar to the grammar formalism underlying the Head-Driven Phrase Structure Grammar (HPSG)*¹, widely used in computational linguistics for the modelling of natural languages. The result is a two-level approach to the generation of shapes: an initial abstract symbolic representation is generated, from which the actual shapes are subsequently derived.

While shape grammars and type constraint systems are not directly translatable into each other, the approach described in this paper can be implemented efficiently, making it easy to develop new shape languages and allowing for a wide range of interesting approaches to the generation of shapes.

Key words: Generative Design, Formal Grammars, Shape Grammars, Type Constraint Systems

1. Introduction

The most straightforward way to demonstrate the effectiveness of a problem solving approach is to apply it to a relevant example. In this paper we show, that *Type Constraint Systems* (TCS; Carpenter, 1992) are a valuable framework for the generative description of shape, by applying them to the shape language generated by the ‘Urform’ shape grammar, formulated by Stiny and Gips in 1971. We demonstrate, how the same language can be generated, by first deducing an abstract description of a shape using a TCS, and then interpreting this description to obtain the actual two-dimensional shape. The differences between the shape grammar approach and the type constraint system approach are discussed, and some new ideas for the generation of shapes are introduced.

This paper is written in a tutorial style and technical details, not necessary for the understanding of the general idea, are omitted whenever possible.

2. George Stiny and James Gips’ Urform Grammar

2.1 The Urform language

Shape grammars were introduced about forty years ago by George Stiny and James Gips in their seminal paper “*Shape Grammars and the Generative Specification of Painting and Sculpture*” (Stiny and Gips, 1971) and have since been very influential in fields concerned with generative approaches to design. To illustrate how shape grammars work, Stiny and Gips introduced a simple example,

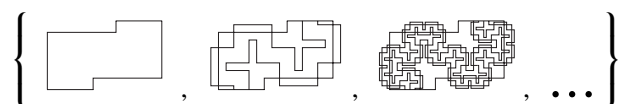
called *Urform grammar*, which generates shapes such as the following:

- (1) *Urform I, II, and III* (Stiny, 1970. *Acrylic on canvas 30 ins. x 57 ins.*)*²



The *generative specification* of the Urform grammar consists of two components: a *shape specification* or *shape grammar* for the generation of the shape geometry, and a *material specification* for the selection of materials and colours in the final representation. The images in (1) show the first three elements of the language generated by the system after applying the material specification. The shape geometry of these images as defined by the shape grammar alone, i.e. the shapes before the application of the material specification, looks as follows:

- (2) *The Language defined by the Urform grammar*



The set notation is used to indicate that only the first three elements out of a countably infinite series of shapes generated by the grammar are shown.

*¹See Pollard and Sag, 1987, 1994.

*²The three images of the paintings Urform I, II, and III are from Stiny and Gips’ original paper (Stiny and Gips, 1971).

2.2 The Urform grammar

The *Urform shape grammar*, responsible for the generation of the shapes shown in the last paragraph, is defined as follows:

(3) The Urform Grammar $SG_1 = \langle V_T, V_M, R, I \rangle$

$$V_T = \{ \text{—} \}$$

$$V_M = \{ \text{◡} \}$$

$$R = \left\{ \begin{array}{l} \text{◡} \rightarrow \text{—} \quad (r_1), \\ \text{◡} \rightarrow \text{—} \quad (r_2), \\ \text{◡} \rightarrow \text{—} \quad (r_3) \end{array} \right\}$$

$$I = \text{—} \text{◡}$$

- V_T describes the set of basic graphic elements from which all shapes are assembled. In the case of the present grammar, all shapes are generated from one simple element: a line segment.
 - V_M is a set of graphic elements called *markers* which are only used during the generation of the shapes and are then deleted from the final images. The Urform grammar makes use of only one marker, represented by the ginkgo leaf ◡.
 - R is the set of rules used for generating the shapes. The shapes on the left and right sides of a rule are made of elements chosen from the sets V_T and V_M . During the generation of a shape, the left side of the rule is matched against the current state of the shape and then the matched part of the pattern is substituted with the right side of the rule. Geometric transformations like scaling, or rotations necessary to match the left side against the shape, have to be applied to both sides of the rule in the same way.
- The Urform grammar has three rules: Starting from the initial shape I , the first two rules generate intermediate states by transforming the basic graphic elements and the position and state of the marker; the last rule is responsible for deleting the marker from an intermediate shape and thus generating the markerless final shape.
- I is the initial shape from which all other shapes of the language have to be generated.

The language defined by a shape grammar is defined as the set of all shapes without markers which can be generated by successive application of the rules to the initial shape.

For a more detailed definition of shape grammars refer to the original paper by Stiny and Gips (1971).

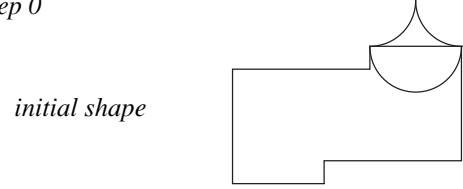
2.3 The generation of shapes

The easiest way to understand, how the different parts of a shape grammar interact to generate a shape, is to look at the generation of some examples. This paragraph therefore explains how the first three elements of the Urform lan-

guage are generated.

Urform I The shape generation starts from the initial shape I , composed of line elements from V_T and the V_M marker ◡:

(4) Step 0

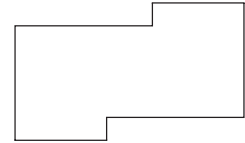
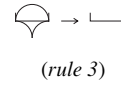


By iteratively matching rules to the current state of the shape and substituting the left side of the matching rule with its right side, a succession of shapes can be generated. When the marker has been eliminated due to the application of the last rule, a final shape is obtained, which is an element of the language generated by the shape grammar.

Starting from the initial shape we can immediately apply r_3 and obtain the first element of the shape language defined by SG_1 :

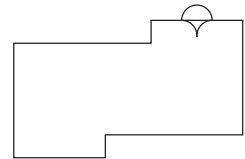
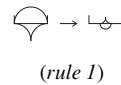
(5) The simplest shape generated by SG_1

Step 1



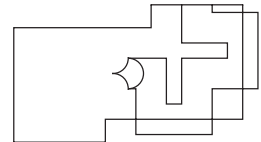
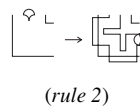
Urform II If we apply r_1 instead of r_3 to the initial shape, we obtain a different succession of shapes. The second element of this series is similar to the initial shape but with the marker scaled down and pointing to the bottom:

(6) Step 1



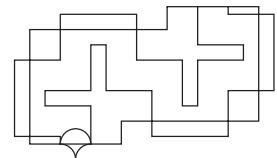
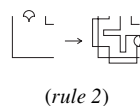
The only rule which can be applied to this shape is r_2 which has to be scaled first and mirrored in order to match:

(7) Step 2



The same rule can be applied again in step 3,

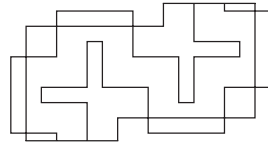
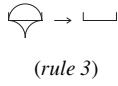
(8) Step 3



which finally allows us to generate the second shape, defined by SG_1 , by applying r_3 once again:

(9) *The second-simplest shape generated by SG_1*

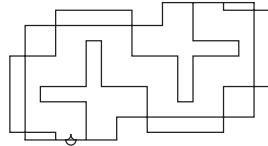
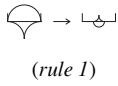
Step 4



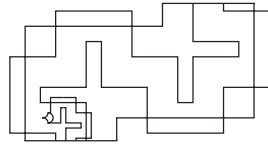
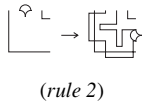
Urform III Following the same logic and applying r_2 instead of r_3 once again in step 4, we obtain the third shape of the Urform language:

(10) *The third-simplest shape generated by SG_1*

Step 4

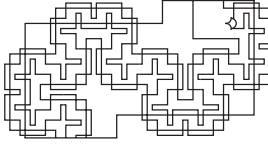
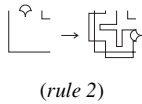


Step 5

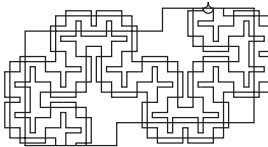
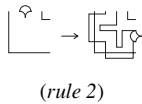


• • •

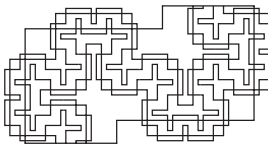
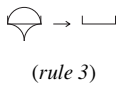
Step 17



Step 18



Step 19



Urform IV, ... Continuing this way all shapes defined by the Urform language can be enumerated.

Finally, by applying the material specification (explained in full in Stiny and Gips (1971)^{*3}), the final coloured shapes shown in (1) can be generated.

3. Modelling the Urform Grammar as Type Constraint System

3.1 Type constraint systems

The implementation of *Type Constraint Systems* (TCS) as used in this paper is characterised by four components:

- A *Type Hierarchy* (TH) over a set of *Types*
- *Type Constraints* (TCs) formulated as *Typed Feature Structures*
- *Value Constraints*
- *Resolution*

Value constraints are an extension of TCSs added for the current research. They allow the calculation of numerical shape attributes like size and location.

These four components will be discussed further during the explanation of the implementation of the Urform grammar as a TCS.

A formal definition of TCSs can be found, together with a more detailed explanation and overview of their theory and application, in Carpenter (1992).

3.2 From shape grammars to type constraint systems

Continuous shapes versus discrete symbolic representations In the case of shape grammars, the initial shape, the patterns used on the left and right side of the rules, the intermediate states produced by the successive application of rules and the final results of the generation process are all of the same nature: they are continuous geometric shapes, images or three-dimensional objects. The shape grammar formalism therefore is based on *shape embedding*, a process similar to geometric pattern matching; because of the continuous nature of the representations, the shape grammar formalism allows for complex ways of matching.

On the other hand, the approach introduced in this paper (the implementation of shape languages using TCSs) uses recursive combinatorial structures over a set of discrete symbols as representation. The matching procedure is based on unification which, because it is a discrete symbolic procedure, is in some respects more restricted than its continuous counterpart used in shape grammars.^{*4} Nevertheless, its abstract symbolic nature allows for arbitrary geometric interpretations as well as for unrestricted symbolic processing.

Due to the different nature of representations and matching procedures used, shape grammars and the approach introduced in this section are fundamentally different. However, seen from a more abstract perspective, both are based on the idea of using production systems for the generation of shape. As a result, in many cases there are more similarities than differences between them.

Even if the result might differ in elegance, every grammar that can be written in one of the two formalisms can also be written in the other. At least from a theoretical point of view, the two approaches are equivalent: shape grammars as well as TCSs are Turing complete and every calculation that can be done by a computer could also be implemented, at least in theory, as a shape grammar or as a TCS.

^{*3}See the original paper by Stiny and Gips (1971) for a detailed explanation of the material specification and its application to the output of the shape grammar.

^{*4}For a detailed discussion of the differences between shapes and symbolic representations and the consequences regarding the expressiveness of the grammars see Stiny (2006).

A two-level approach based on an abstract representation Different representations of shapes make it necessary to rely on different modelling mechanisms. The use of a discrete abstract representation for the formulation of rules and the intermediate states used during the generation process results in a two-level approach to the generation of shape:

- 1) *Level 1*: Generation of the symbolic structure representing the shapes in an abstract way^{*5};
- 2) *Level 2*: Translation of the symbolic structure into the actual geometric shape.

Due to the different representations and organisation of shape grammars and TCSs, it is difficult to specify a generic method for the translation from one into another. However, in many cases it is not difficult to formalise a shape language with a TCS when combined with an adequate visual interpretation of the generated symbolic structures.

The following example, demonstrating how the Urform grammar can be implemented as a TCS, will give a better idea of the differences and similarities of both approaches and how they can be translated into each other.

3.2.1 Describing the Urform images as sequence of types

The vocabulary of the Urform language The images of the Urform language (1) show that all forms can be assembled from one basic element, an L-shape:

(11) *L-shape*



The graphic elements are named with symbols called *types* in the context of TCSs. Using the type *right* to represent this shape and the type *left* for its mirror image, all primitive graphic elements needed to assemble the urform images have been defined.

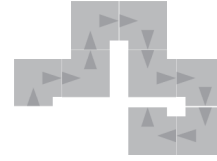
Shape composition Only two methods are necessary to combine the elements of the Urform vocabulary into complex Urform images:

- 1) Shapes of the same size are aligned one after the other similar to dominos. The sequence
right, left, right, right, left, right, right, right,
for example, looks as follows:^{*6}

^{*5}The abstract symbolic structure brings to mind the *deep structure* introduced by Chomsky for the representation of the underlying structure of sentences in natural language (Chomsky, 1957). There have been different motivations for introducing the deep structure in his model of grammar which mostly have no equivalent in the approach introduced in this paper. The most important similarity between both structures is a purely technical one: in both cases the use of an abstract representation makes it easier to apply the formalism to the modelled domain.

^{*6}Similar to the dots stamped on dominos, small grey arrows are used to indicate the location and direction in which the shapes are arranged.

(12)



- 2) The direction is inverted while the shapes are scaled to one third of their previous size. This operation is represented by the type *turn*. The sequence
right, turn, right,
for example, looks as shown in the following image:

(13)



Both methods of shape combination can be used together, as the following random sequence over the types *right*, *left* and *turn* shows:

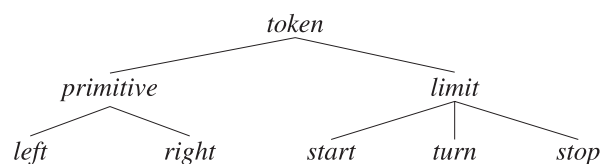
(14)



By adding the types *start* and *stop* to mark the beginning and end of the sequence, all Urform images can be described as sequences over the set of types {*start*, *left*, *right*, *turn*, *stop*}.

Organising the types as a type hierarchy A set of types can be understood as a terminology for the description of a certain domain. Similar to the concepts of natural language, types can be ordered depending on the generality of their meaning via an *is-a* relation: the primitive shapes *left* and *right*, for example, can be grouped together under a more general type called *primitive*. On the other hand, the types *start*, *turn* and *stop*, which are used to limit levels of aligned primitive forms of the same size, can be made subtypes of a type called *limit*. Finally, *primitive* and *limit* can be grouped under the type *token*, thus subsuming all types used for sequences representing Urform images. Graphically this hierarchy of types can be represented in the following diagram:

(15) *The token type hierarchy*



Often a textual representation in the form of a list of types with their immediate subtypes, as shown in the following figure, is more convenient:

- (16)
- | | | |
|------------------|---|--|
| <i>token</i> | → | <i>primitive</i> <i>limit</i> |
| <i>primitive</i> | → | <i>left</i> <i>right</i> |
| <i>limit</i> | → | <i>start</i> <i>turn</i> <i>stop</i> |

3.2.2 Unification of types Types can be *consistent* or *inconsistent*. The types *left* and *right*, for example, as given in the *token* hierarchy (15), can never be valid for the same graphic element and therefore are *inconsistent*; however, the types *primitive* and *left* are both valid for the L-shape labelled as *left* and therefore are *consistent* relative to the *token* hierarchy.

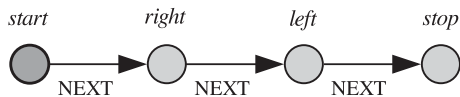
In the current paper, type hierarchies are defined as trees^{*7}. Two types therefore can be consistent only if they are identical or if one of them is a subtype of the other.

In an extensional semantic, the meaning of a type is defined by the set of entities described by it: its *extension*. Therefore, an entity which is part of the extension of a certain type is also part of the extensions of all its supertypes. Thus the extension of the conjunction of two consistent types is the same as the extension of the more specific of the two. In the case above, for example, stating that some entity is described by the type *left*, as well as by the type *primitive*, provides no more information than simply stating that it is described by the more specific type *left* alone.

The partial function that selects the more specific of two consistent types is called *unification*.

3.2.3 Typed feature structures and the representation of sequences of types We have already seen how Urform images can be described as sequences of types of the *token* hierarchy. These sequences can be represented as directed, rooted and annotated graphs: see (17).

- (17) *Representing an Urform image by a sequence of types*



The vertex labelled with the type *start* is the root of the graph. NEXT is used to label edges connecting the elements of the token sequence.

Graphs like (17) can also be notated in matrix form:

- (18) *Representing an Urform image by a sequence of types (matrix notation)*

$$\begin{bmatrix} \text{start} \\ \text{NEXT} \begin{bmatrix} \text{right} \\ \text{NEXT} \begin{bmatrix} \text{left} \\ \text{NEXT stop} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Both notations can be used to represent the same kind of structures, referred to as *typed feature structures (TFS)* from

this point on. As the matrix notation is generally easier to read, it will be used from here onwards.

Typed feature structures as representations of partial information Similar to the way types can be seen as *atomic representations* of information, TFS can be seen as *complex representations* of information. The information can be *partial* as in the following example:

(19)

$$\begin{bmatrix} \text{start} \\ \text{NEXT} \begin{bmatrix} \text{right} \\ \text{NEXT token} \end{bmatrix} \end{bmatrix}$$

The value *token* at the path^{*8} NEXT|NEXT can be instantiated by any token sequence. Therefore structure (19) can be seen as a partial description of a token sequence that consists of at least three elements and starts with the types *start* and *right*.

The following sequence similarly describes a four-token sequence of which only the last two—*left* and *stop*—are known:

(20)

$$\begin{bmatrix} \text{token} \\ \text{NEXT} \begin{bmatrix} \text{token} \\ \text{NEXT} \begin{bmatrix} \text{left} \\ \text{NEXT stop} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Consistent and inconsistent typed feature structures

As types, TFS can be *consistent* or *inconsistent*. The structures (19) and (20), for example, describe one sequence starting with the types *start* and *right*, and another of four types ending with *left* and *stop*. Since a sequence exists which is described by them both (*start, right, left, stop*), the two structures are consistent.

On the other hand, a structure with type *left* as the second element and one with type *right* as the second element contradict each other and therefore are *inconsistent*:

(21)

$$\begin{bmatrix} \text{token} \\ \text{NEXT left} \end{bmatrix} \quad \begin{bmatrix} \text{token} \\ \text{NEXT right} \end{bmatrix}$$

3.2.4 Unification of typed feature structures Similar to consistent types, consistent typed feature structures can be unified. The structures (19) and (20), for example, describing a sequence starting with *start* and *right* and a sequence of four types ending on *left* and *stop* can be unified to form a structure describing the sequence (*start, right, left, stop*):

- (22) *Unification of typed feature structures*

$$\begin{aligned} & \begin{bmatrix} \text{start} \\ \text{NEXT} \begin{bmatrix} \text{right} \\ \text{NEXT token} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{token} \\ \text{NEXT} \begin{bmatrix} \text{token} \\ \text{NEXT} \begin{bmatrix} \text{left} \\ \text{NEXT stop} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} \text{start} \\ \text{NEXT} \begin{bmatrix} \text{right} \\ \text{NEXT} \begin{bmatrix} \text{left} \\ \text{NEXT stop} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{aligned}$$



^{*7}A more powerful way to define type hierarchies allowing for multiple inheritance is to define them as lattices. See Chapter 2 “Types and Inheritance” in Carpenter (1992).


^{*8}Paths are used when talking about substructures in a typed feature structure: The value of the *empty path* ϵ is the structure itself; the value of path A is the same as the value of the feature A; the value of path A|B is the value of feature B in the structure which is the value of A and so on.

The symbol \sqcup in figure (22) represents the unification function.

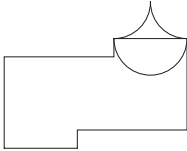
3.2.5 The fractal nature of the Urform language

If we analyse the first three images of the Urform language into the corresponding token sequences, we get the following result:

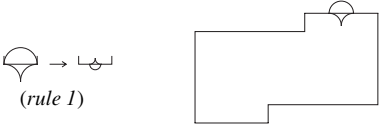
- (23) a. 
right, left.
- b. 
*right, left, turn,
left, right, right, left, right, right, left,
right, left, left, right, left, left, right.*

- c. 
Using *l* for *left*, *r* for *right* and
t for *turn*:
- rl t*
lrrllrl rllrllr t
rllrllr lrrllrl lrrllrl rllrllr lrrllrl lrrllrl rllrllr
lrrllrl rllrllr rllrllr lrrllrl rllrllr rllrllr lrrllrl.

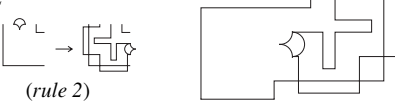
How is this regular sequence of left and right curves enforced by the shape grammar? The initial sequence of a *right* and *left* shape underlying the first and simplest shape of the Urform language, is determined by the initial shape *I* as given in the definition of the Urform shape grammar SG_1 in (3):

- (24) $I =$ 

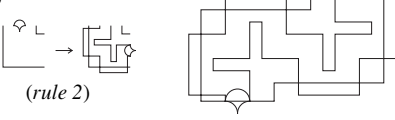
First the marker is turned and downscaled by applying the first shape rule (r_1) to the initial form. This is represented in the token sequence by the type *turn* and starts a new 'level' of smaller overlaid L-shapes.

- (25) *Step 1*
- 
(rule 1)

After step 1, the second rule (r_2) can be applied twice. These applications correspond to the matching of the already existing sequence $\langle \textit{right}, \textit{left} \rangle$, representing the last level of shapes, in reverse order. First the form we labelled as *left* is matched, resulting in the original form being overlaid with downscaled shapes in the order *left, right, right, left, right, right, left*:

- (26) *Step 2*
- 
(rule 2)

Then the form labelled as *right* is matched, resulting in the shape labelled as *right* being overlaid by a sequence of shapes in the order *right, left, left, right, left, left, right*:

- (27) *Step 3*
- 
(rule 2)

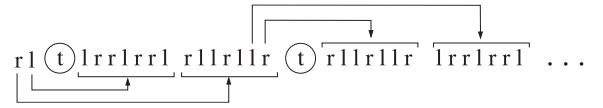
All elements of the Urform language are generated by recursively repeating this process of traversing the shapes of the last level in inverse order and overlaying them with a sequence of downscaled shapes. The result are the fractal shapes of the Urform language.

We can summarise the mechanism constraining the sequence of primitive shapes using the following two rules:

- (28) a. *left* \rightarrow *left, right, right, left, right, right, left.*
b. *right* \rightarrow *right, left, left, right, left, left, right.*

The application of these rules to the sequence already existing in inverse order is shown in the following diagram:

- (29) *Order of the basic left and right shapes:*

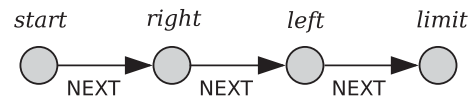


These rules will be referred to as *token substitution rules*.

3.2.6 Generating the Urform sequence with a type constraint system After the mechanism responsible for generating the Urform images as a sequence of L-shapes has been described, the mechanism needs to be formalised as a constraint system.

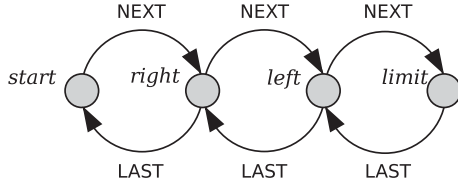
Representing the initial sequence Starting with the *start* token, representing the primitive shapes by their corresponding types and adding the type *limit* as a marker for the end of the first level, the sequence of primitive shapes contained in the initial shape *I* of the original shape grammar SG_1 (3) can be represented by the following graph:

- (30) *Sequence of primitive shapes contained in the initial shape*

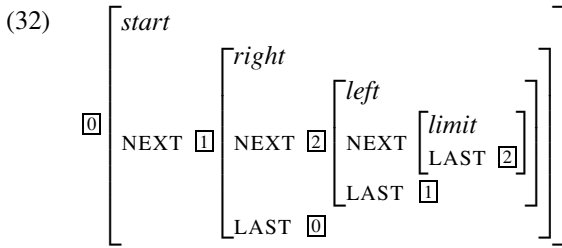


As we need to be able to traverse the sequence backwards as well as forwards, we extend the graph with edges pointing backwards and add the label, *LAST*:

(31) *Sequence of primitive shapes contained in the initial shape*

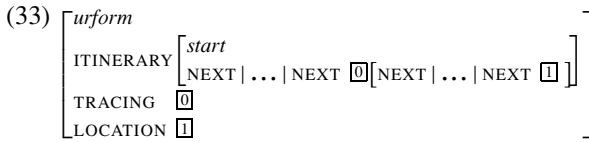


Written in matrix notation, the same structure looks like this:



A *tag*—the small number in the rectangle—is used when two different features share the same value. The path NEXT|LAST, for example, points back to the original structure via the tag [0].

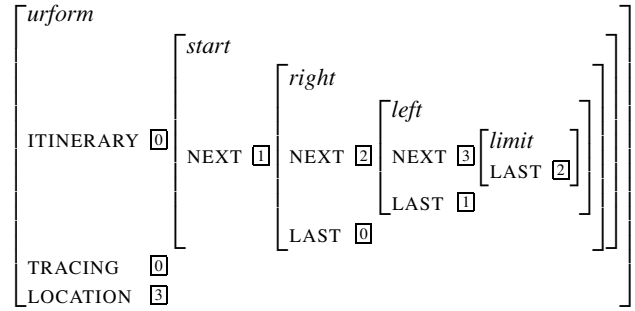
The initial sequence has to be extended into the sequences corresponding to the Urform language images. In order to formalise the rules necessary to generate the sequences using *right*, *left*, *turn*, characteristic of Urform images, two more features are added: TRACING and LOCATION.



- 1) TRACING — The sequence corresponding to the previous level of primitive shapes is traversed in inverse order to generate the sequence of primitive shapes at the new level. The feature TRACING points to the token of the previous level, which has to be read next to generate the corresponding token sequences at the new level.
- 2) LOCATION — This feature indicates the end of the sequence generated up to this point. The shape can be extended by instantiating the value of this feature with a sequence of *left*, *right*, *turn* and *stop* tokens.

Putting these structures together, the initial shape is represented by the following typed feature structure which we call *query* as it can be understood as a query to the constraint system, producing the structures of the Urform images as answers:

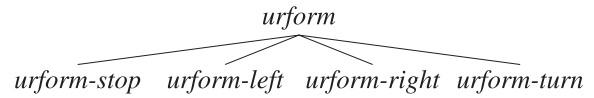
(34) *Query structure:*



Encoding the token substitution rules as type constraints Every substructure in a typed feature structure has a type. The query structure (34), for example, has the type *urform*, the substructure at path ITINERARY has the type *start* and so on. Using this type, certain constraints concerning the features and values of the corresponding structure can be formulated. Together with the type hierarchy (TH), which allows the specification of different subtypes for a given type, these constraints can be used to implement complex algorithms.

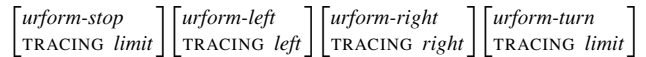
Let us suppose the following TH for the root type *urform* of the query given in (34):

(35) *The urform type hierarchy*



For every subtype a constraint is given, which, for the time being, only constrains the value of the feature TRACING:

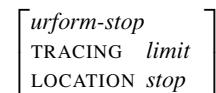
(36) *Constraints for subtypes of type urform*



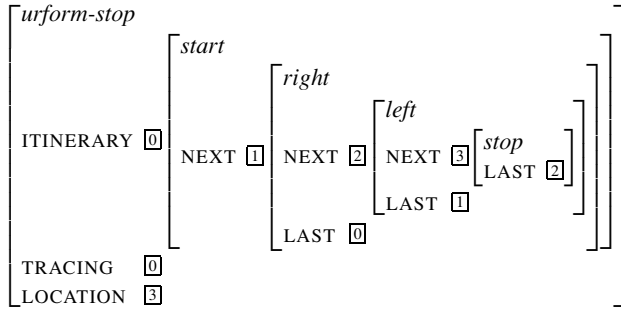
Comparing these constraints with the query (34), we see that only two of them, the constraints for the subtypes *urform-stop* and *urform-turn* can be unified with the query. The reason is as follows: the query has the value *start* as value of the feature TRACING; the TCs for the subtypes *urform-stop* and *urform-turn* have the type *limit* as the value for the feature TRACING; the type *limit* is the supertype of the type *start* and therefore *limit* and *start* can be unified. All other constraints have incompatible values for feature TRACING and therefore are not consistent with the query structure.

Generating the first solution When extending the constraint for *urform-stop* in the following way

(37) *Constraint for urform-stop*



its unification with the query results in the following structure:

(38) *The first solution*

This structure differs from the original query structure in only two points:

- 1) The root type has been replaced by the type *urform-stop*, which results from the unification of the root types of the query *urform* and the *urform-stop* constraint.
- 2) The type at path *LOCATION*, which is the same as the one at path *ITINERARY|NEXT|NEXT|NEXT*, has been replaced with the type *stop*, which results from the unification of *limit* and *stop*, the first being the original value of the query structure at path *LOCATION*, the second the value of the same path of the *urform-stop* constraint.

The sequence encoded in this structure is $\langle \text{start}, \text{right}, \text{left}, \text{stop} \rangle$, which corresponds to the first image of the Urform language:

(39) *Urform I*

Having demonstrated how the token sequence corresponding to the first solution can be generated, the paper will now explain in more detail the process responsible for generating solutions (*resolution*).

3.3 Resolution

The TH and TCs of a constraint system can be understood as a kind of knowledge base: Questions to this knowledge base can be posed in the form of *queries*, which are answered by applying the knowledge stored in the knowledge base. The mechanism responsible for this procedure is called *resolution*.

The resolution algorithm of the constraint system used for this paper is simple: The type subsumption relations and the TCs are applied to the query structure until

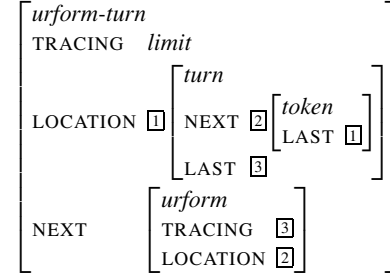
- 1) all types in the query structure are minimal in the sense, that they have no subtypes;
- 2) all TCs matching types in the solution structure have been applied.

In the TCS discussed here, the type subsumption relations and TCs are applied in a depth-first order and backtracking is used whenever the unification of a TC fails. The order in which subtypes of a type are tried, and features inside a

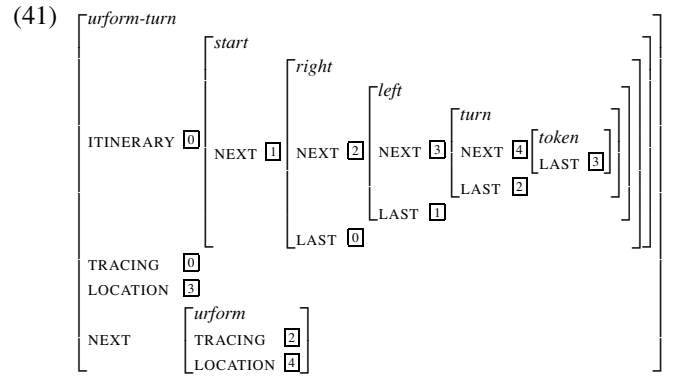
structure are resolved, follows their definition order unless otherwise stated.

The following section, which explains the generation of the second solution, demonstrates the resolution process in more detail.

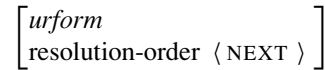
Generating the second solution Besides *urform-stop*, the second subtype of *urform* with a type constraint matching the query structure (34) is *urform-turn*. The complete TC for *urform-turn* is:

(40) *Constraint for urform-turn*

Unifying it with the query results in the following structure:



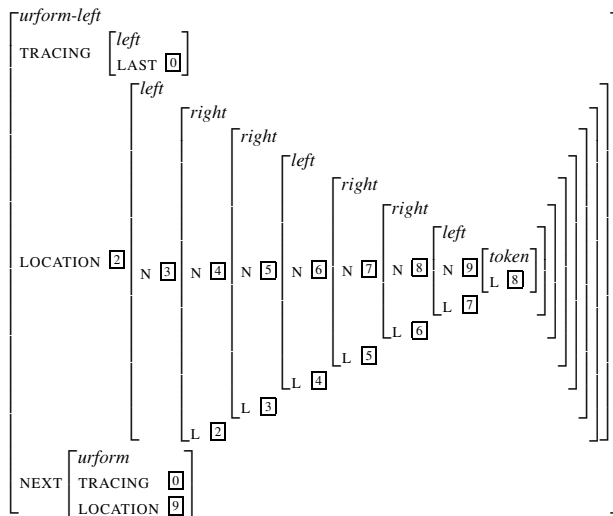
The root type *urform-turn* is a minimal type and therefore already resolved. The resolution procedure continues with its feature values. In the case of structures with the root type *urform* or one of its subtypes, the resolution order for its features has been given explicitly:

(42) *Resolution order of urform structures*

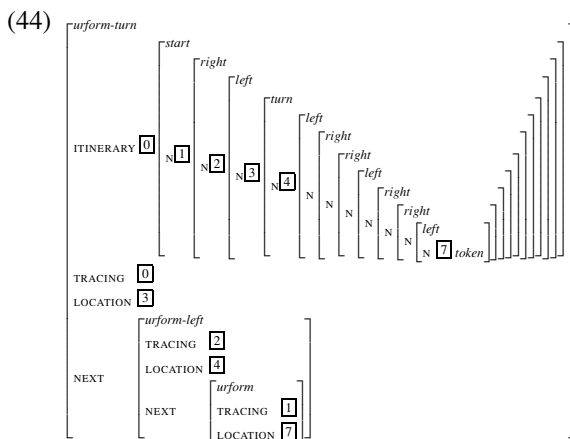
Resolution-order is a meta-attribute which can be used to specify the resolution order of the features: the listed features are resolved first; features not listed are resolved later, using the order in which they have been defined.

Looking at the value of feature *NEXT*, it can be seen to have the same type *urform* and features *TRACING* and *LOCATION* as the original query. The feature *TRACING*, however, now points to the last token of the first level, and the *LOCATION* feature points to the new end of the token sequence, extended by the type *turn*. This time only the type constraint for *urform-left* matches, which in its complete form looks as follows (the features *NEXT* and *LAST* have been abbreviated in most cases as *N* and *L* respectively):

(45) *Urform II*



(46) *Urform III*



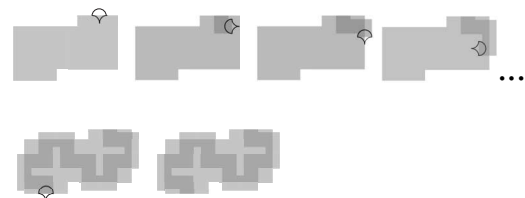
Continuing the same way, the token sequences for all images of the Urform language are enumerated.

Calculating the position of the L-shapes In order to instantiate the L-shapes corresponding to the elements of the token sequences three pieces of information are necessary for every single L-shape instance: its location, direction and size. Adapting the ginkgo leaf marker \curvearrowright used in the original shape grammar to represent these factors, their role during the generation of the actual shapes can be visualised by the following diagrams:

(47) a. *Urform I*



b. *Urform II*



The far left image of (47a) shows the initial state of the marker. The next image shows a *right* shape positioned at its place and the marker advanced to the next position. The image to its right shows a *left* shape instantiated corresponding to the new marker state and the marker advanced to the next position again. This final marker position is ignored as in the fourth image, which corresponds to the first Urform image.

As before the resulting structure contains a new inner feature NEXT with an *urform* structure as the value which has to be resolved first. The feature TRACING contained in this new query-like structure points to the *start* token at path ITINERARY again. Again the two constraints *urform-stop* and *urform-turn* match. Exactly in the same way as the *Urform I* image was generated, the first of these constraints delimits the sequence with the token *stop*, and by doing so generates the token sequence corresponding to the second Urform image:

The sequence of images in (47b) shows the continuation of the process when the marker is scaled down and turned to the opposite direction to initiate the next level of overlaid L-shapes resulting in the second Urform image.

Modelling the marker with type constraints In the TCS, the marker is modelled by a *marker* structure containing the features LOCATION, DIRECTION, ROTATE and SCALE.

$$(48) \quad \left[\begin{array}{l} \text{marker} \\ \text{LOCATION } location \\ \text{DIRECTION } direction \\ \text{ROTATE } rotate \\ \text{SCALE } scale \end{array} \right]$$

The calculation of the marker states corresponding to the elements of the token sequence is enforced by a number of constraints.

The initial state of the marker is added to the *start* token of the query structure:

$$(49) \quad \left[\begin{array}{l} \text{urform} \\ \text{ITINERARY} \left[\begin{array}{l} \text{start} \\ \text{MARKER} \left[\begin{array}{l} \text{marker-start} \\ \text{LOCATION } \langle 100, 190 \rangle^{*9} \\ \text{DIRECTION } north \\ \text{ROTATE } 0 \\ \text{SCALE } 1 \end{array} \right] \end{array} \right] \end{array} \right]$$

The feature ROTATE encodes the same information as the feature DIRECTION represented in the form of the corresponding rotation angle.

Every structure with the root type *token* or one of its subtypes contains a feature MARKER with its corresponding *marker* structure as the value:

(50) *The token constraint*

$$\left[\begin{array}{l} \text{token} \\ \text{MARKER} \left[\begin{array}{l} \text{marker} \\ \text{TOKEN } \boxed{0} \end{array} \right] \end{array} \right]$$

The feature TOKEN in the marker structure refers back to the token structure and allows access to all features which are accessible from both the token structure and the marker structure.

The marker state depends on the direction of the previous marker and the type of the current token as shown in the following table:

(51) *Calculation of the marker features*

LAST DIRECTION	TOKEN	NEW DIRECTION	ROTATE	TRANSLATE	SCALE
north	left	west	-90	$\langle -90, -90 \rangle$	1
	right	east	90	$\langle 90, -90 \rangle$	1
	turn	south	180	$\langle 0, 0 \rangle$	1/3
east	left	north	0	$\langle 90, -90 \rangle$	1
	right	south	180	$\langle 90, 90 \rangle$	1
	turn	west	-90	$\langle 0, 0 \rangle$	1/3
south	left	east	90	$\langle 90, 90 \rangle$	1
	right	west	-90	$\langle -90, 90 \rangle$	1
	turn	north	0	$\langle 0, 0 \rangle$	1/3
west	left	south	180	$\langle -90, 90 \rangle$	1
	right	north	0	$\langle -90, -90 \rangle$	1
	turn	east	90	$\langle 0, 0 \rangle$	1/3

If, for example, the last marker pointed to the *north* and the current token has type *left*, the next marker points to the *west*, which corresponds to a -90° rotation of the L-shape. The next marker location results from the translation of the previous marker location by the vector $\langle -90, -90 \rangle$ multiplied by the current scale factor 1 and the next marker size corresponds to the size of the previous marker multiplied by the same scale factor 1. This calculation of the marker feature values can be formalised by the following TC:

(52) *Marker constraint*

$$\left[\begin{array}{l} \text{marker-north-left} \\ \text{TOKEN } \left[\begin{array}{l} \text{left} \\ \text{LAST | MARKER} \left[\begin{array}{l} \text{DIRECTION } north \\ \text{LOCATION } \langle \boxed{0}, \boxed{1} \rangle \\ \text{SCALE } \boxed{2} \end{array} \right] \end{array} \right] \\ \text{DIRECTION } west \\ \text{TRANSLATION } \langle \boxed{3} - 90, \boxed{4} - 90 \rangle \\ \text{LOCATION } \langle (\boxed{0} + \boxed{2} * \boxed{3}), (\boxed{1} + \boxed{2} * \boxed{4}) \rangle \\ \text{ROTATE } -90 \\ \text{SCALE } \boxed{2} \end{array} \right]$$

The type of the current token, as well as the feature values of the marker belonging to the previous token, are accessed via the TOKEN feature. The path TOKEN|LAST|MARKER|DIRECTION, for example, refers back to the direction value of the previous marker.

The new values are calculated with the aid of *value constraints*, arithmetic expressions which allow the calculation of numerical feature values depending on the values of other features. The *x* value of the marker LOCATION, for example, is calculated by adding the *x*-value of the location of the previous marker to the product of the scale factor and the *x* translation value of the current marker.

The calculation of the marker features differs depending on the current token and the direction of the previous marker as shown in Table (51). Each row of Table (51) therefore has to be translated into a constraint similar to the one shown in (52). The application of the matching constraint is then enforced by subtyping the marker type into the subtypes corresponding to the single constraints:

^{*9} $\langle x, y \rangle$ is an abbreviation for the structure $\left[\begin{array}{l} \text{vector} \\ \text{X } x \\ \text{Y } y \end{array} \right]$

(53) *Marker subtypes*

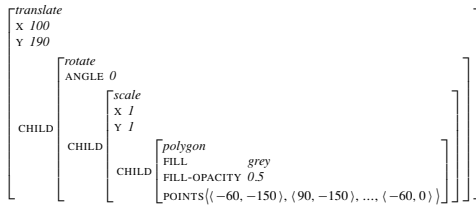
<i>marker</i> → <i>marker-start</i>	<i>marker-stop</i>
<i>marker-north-left</i>	<i>marker-north-right</i> <i>marker-north-turn</i>
<i>marker-east-left</i>	<i>marker-east-right</i> <i>marker-east-turn</i>
<i>marker-south-left</i>	<i>marker-south-right</i> <i>marker-south-turn</i>
<i>marker-west-left</i>	<i>marker-west-right</i> <i>marker-west-turn</i>

The value of the current token at path *TOKEN* and the direction of the previous marker at *TOKEN|LAST|MARKER|DIRECTION* differ for every constraint. Therefore only the constraint with matching values can be applied, while the unification with all other constraints fails. For every possible combination of token and marker direction, precisely one matching constraint exists. The implicit disjunction encoded in the marker subtype list, together with the backtracking built into the resolution procedure, always enforce the application of the right constraint.

4.1 *Generation of the code for the final images*

After the token sequence and the values for the location, scale and rotation of the corresponding L-shapes have been calculated, this information has to be brought into a format which can be interpreted by the computer to generate the actual images. In the present approach, the Scalable Vector Graphics (SVG)^{*10} format was selected for this purpose.

The L-shapes represented by the token *left* and *right* are represented as polygons, which are translated, rotated and scaled corresponding to the values given in their marker structures. The polygon itself is described by its colour, opacity and the list of its vertices:

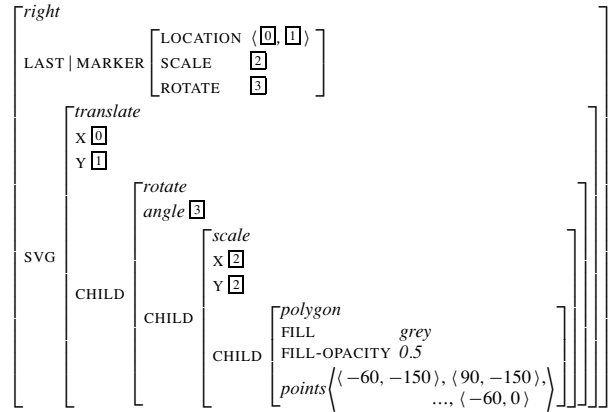
(54) *L-shape (TFS)*

These structures can be translated directly into the corresponding XML format used for scalable vector graphics:

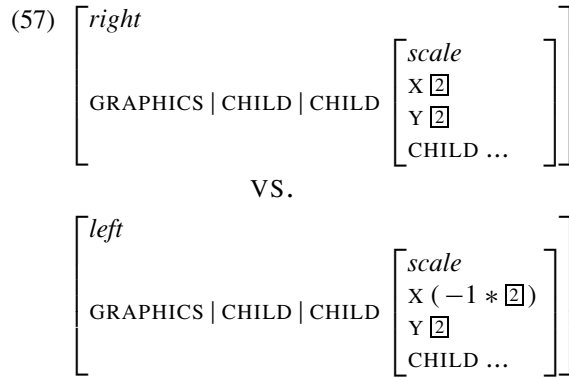
(55) *L-shape (SVG)*

```
<g transform="translate(100,190)">
  <g transform="rotate(0)">
    <g transform="scale(1,1)">
      <polygon
        fill="grey"
        fill-opacity="0.5"
        points="-60,-150 90,-150 ... -60,0"/>
    </g>
  </g>
</g>
```

The SVG structures for the *right* token are generated by the following constraint:

(56) *Constraint for type right*

An equivalent constraint is defined for the *left* token. These two constraints differ only in the calculation of the *SCALE|X* value, which is multiplied by -1 in the case of the constraint for *left*, resulting in the mirror image of the one described by the constraint for *right*:



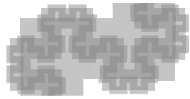
The code for the Urform image is generated by collecting the values of the SVG features, translating them into the appropriate XML format and adding a header with information about the image size:

```
(58) <?xml version="1.0" encoding="UTF-8"?>
      <svg height="200"
          width="380"
          xmlns="http://www.w3.org/2000/svg">
        list of L-shapes
      </svg>
```

The following images show the first three instances generated by the resulting system:

(59) *Urform I, II, and III*

^{*10}See the *Scalable Vector Graphics (SVG) Full 1.2 Specification*, W3C Working Draft 13 April 2005 at <http://www.w3.org/TR/SVG12/>.

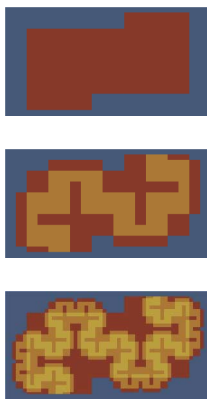


The different grey shadings are caused by the overlapping of the opaque L-shapes.

Rendering of the final colour images To obtain coloured images identical to the original Urform images the different grey shadings have to be translated into the colours given in the material specification of the original Urform grammar described by Stiny and Gips (1971).

The following figure shows the first three of the series of final images:

(60) *Urform I, II, and III, regenerated*



Summary

Using Stiny and Gips' *Urform* grammar as an example, this paper has shown how shape languages can be modelled as TCSs. When seen from an abstract point of view, shape grammars and TCSs look like two instances of the same idea—the generation of shapes with formal grammars—and therefore seem to have more similarities than differences. However, when examining the formalisation of the same shape language in both systems in more detail, it can be seen that the different nature of the representations and resolution procedures on which the formalisms are based also makes it necessary to rely on different mechanisms to encode the shape languages. In the case of the original Urform shape grammar, the continuous graphic representations and pattern matching procedures allow for intuitive graphic rules resulting in an easy to understand and simple grammar; the discrete symbolic structures and unification algorithm used by TCSs, however, make it necessary to introduce an intermediate symbolic layer and result in a more abstract and indirect formalisation, which is less intuitive than the original shape grammar. The abstractionism of the TCS approach, however, also has advantages: its discrete nature makes it easy to integrate symbolic calculations into it and allows complex graphic interpretations, which are more difficult to realise when using shape grammars.

Depending on the shape language encoded, one or other of the formalisms might be more appropriate: while in the case of the described Urform language the shape grammar approach is more straightforward, for other more abstract image generation procedures, constraint systems might be easier to use.

Finally, seen from a theoretical perspective, both systems are equally powerful: shape grammars as well as TCSs are Turing complete and any possible algorithm can be implemented in either system.

Future prospects

Looking at the remodelling of the Urform language as a TCS and the similarities and differences of the resulting formalisation compared with the original shape grammar, two directions of further research present themselves. The first would be to concentrate on the similarities of the two formalisms and the reimplementing of other shape grammars as TCSs, in order to learn more about the consequences resulting from the selection of each approach concerning the generation of shapes. The second direction for further research could be to focus on the differences between the two approaches, experimenting with grammars and making use of the existence and symbolic character of the intermediate representation in order to describe languages of shape, which are more difficult to realise when experimenting with shape grammars. The integration of transformations and distortions, which transform simple shapes such as cubes or spheres into complex forms, as used, for example, by architects like Peter Eisenman and Frank O. Gehry, seems an interesting perspective for further experimentation. This might result in interesting, new approaches to the generation of shape, directly applicable to disciplines such as product design or architecture.

Acknowledgments. The ideas presented in this paper have been published thanks to the support and encouragement of Professor Akira Fujii, Kenichiro Hashimoto, Yukie Ogoma and Dr. Alvaro Bonfiglio, Wanwen Huang, Stefan A. Gerstmeier, Brandon Yeup Hur, Rainer Sandrock and Jonquil Melrose-Woodman to all of whom I am very grateful.

I would also like to express my gratitude to Makiko Tanaka, who invited me to work in her beautiful old house in Kamakura during the hot summer months of 2011.

References

- Carpenter, B. (1992) *The Logic of Typed Feature Structures with Applications to Unification-based Grammars, Logic Programming and Constraint Resolution*, Volume 32 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, New York.
- Chomsky, N. (1957) *Syntactic Structures*, Mouton & Co., La Haye.
- Pollard, C. and Sag, I. A. (1987) *Information-based Syntax and Semantics, Vol. 1*, Number 13 in Lecture Notes. CSLI Publications, Stanford University, University of Chicago Press, Chicago.
- Pollard, C. and Sag, I. A. (1994) *Head-Driven Phrase Structure Grammar*, University of Chicago Press, Chicago.
- Stiny, G. (2006) *Shape: Talking about Seeing and Doing*, MIT Press, Cambridge, Massachusetts.
- Stiny, G. and Gips, J. (1971) "Shape Grammars and the Generative Specification of Painting and Sculpture", *IFIP Congress* (2), 1460–1465.