GEOMETRIC TYPED FEATURE STRUCTURES: CARRYING GEOMETRIC INFORMATION USING TYPED FEATURE STRUCTURES

<u>Teng-Wen Chang</u>[1], Robert F. Woodbury[2]

1 Assist. Prof., Department of Architecture, Ming Chuan University
2 Associate Prof., School of Architecture, Landscape Architecture and Urban Design, The University of Adelaide

ABSTRACT: This paper explores Geometric Typed Feature Structures as a concept for carrying geometric information based on the theory of Typed Feature Structures[Carpenter, 1992]. Geometric Typed Feature Structures cover an important aspect of design space explorers in which the symbol level representation must carry 3D geometric information. Order Types are the devices in Geometric Typed Feature Structures that carry the continuous infinite domain information, that is, geometry. In this paper, theories and algorithms are applied to two kinds of Order type examples for carrying numerical values and geometric information. We describe the requirements as well as the conditions in which an Order Type can be specified and synchronized with other domain knowledge.

We show two examples of Order Types: lifted reals and IGOSet intervals based on the theory of Geometric Typed Feature Structures. In each example, we outline the mathematics linking it to the theory of Order Types.

KEYWORDS: Design Space Explorers, 3D Geometry, Typed Feature Structures

1. BACKGROUND

Within the domain of computational design, *exploration* is one of the most widely respected and useful models. It is commonly referred to as the *exploration model*[Woodbury, 1987, Smithers, 1992]. The exploration model is used as a prototype for exploring possible alternative designs arrayed in design spaces. The device for exploring large design spaces is called a *design space explorer*. One of issues that have not been resolved in design space explorers is the representation of 3D design instances. Geometric Typed Feature Structures[Chang, 1999] propose a solution to this research problem.

In computational linguistics, a record-like data structure known as *feature structures* has been used by many researchers[1], see [Kaplan and Bresnan, 1982 and Ait-kaci, 1984]. One of the feature structure formalisms, *Typed Feature Structures* developed by Bob Carpenter [Carpenter, 1992], is particular interesting in the context of design space explorers. It provides natural notions of information specificity and partiality essential to creating structured design spaces.

Typed Feature Structures comprise four elements: a set of *types* organized in an inheritance hierarchy, a frame-like representation (feature structure*s*), a description language and the algorithms above those.

[1]. For a more general introduction about this field see[Shieber, 1986].

Construction Informatics Digital Library http://itc.scix.net/
paper w78-2000-166.content

1.1. The Architecture of Typed Feature Structures

The three structured components of Typed Feature Structures are types, *feature structures*, and *descriptions*. Each component is in relation to others. The formal definition of each term borrowed from [Carpenter, 1992] is also given in each section.

1.1.1. Types

Types are a means of efficiently encoding information. They are organized in an unambiguous multiple inheritance hierarchy. In such an inheritance hierarchy, information associated with a type is extended in inheriting *types*, that is, an informational ordering. We say that type α subsumes type γ, if and only if type γ contains more information than type α (the actual algorithm is described later in this chapter). A *bounded complete partial order* (BCPO)[2] condition is imposed in this information ordering described above. Reasoning operations over types are available as the lattice operations such as ♈(join, or most general common specialization) and ♉(meet, or the most specific common generalization). A universal type is described as Bottom (written as □), at which all the *types* meet.

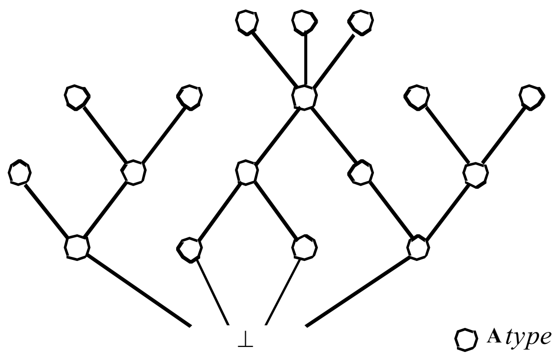


Figure 1. A Graph Notation of an inheritance Hierarchy

All *types* are organized in so-called *inheritance hierarchies* based on the information inheritance of any two *types*, that is a subtype carries more information than its *supertypes*. With a graph-notation (as shown in Figure 1), inheritance hierarchies can be described as a directed acyclic graph. Each vertex of this graph represents a type, and each edge between any two vertices represents the relationship of information inheritance between any two *types*. Cycles are not allowed in inheritance hierarchies. Without cycles in inheritance hierarchies, a subtype of a type can not become the super-type of that same type.

1.1.2. Feature Structure

Feature structure*s* are frame-like representations based on a set of existing type*s*. A minimal feature structure simply contains a single node labelled by type information. An informal description of feature structure*s* is that they are a representation for a labelled, rooted, directed, finite node and edge graph, which can be visualized in graph notation (as shown in Figure 2) or in AVM (attribute-value matrix) notation. Cycles are allowed in *feature structures*. With these characteristics, *feature structures* can easily organize domain information in an intricate structure.

[2]. For the mathematic explanation of the BCPO conditions, see [Davey and Priestley, 1994].

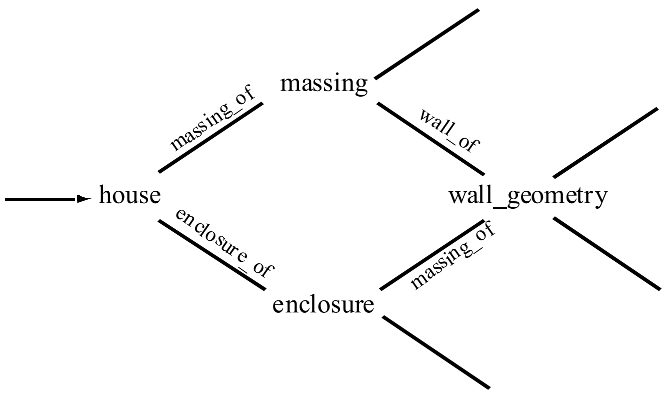


Figure 2. An example Feature Structure in Graph Notation

Naturally, traversing *feature structures* is important. Thus the transition function has to be able to be composed. The path is needed for retrieving the value at the end of a path. "*A path is a sequence of features and we let Path = Feat* be the collection of paths*" [Carpenter, 1992, page 37]. In terms of graph-notation, a path comprises all the edges from its root node to the final destination node. Using the graph notation in Figure 42, a partial path of *wall_geometry* from house is either *massing_of → wall_of* or *enclosure_of→massing_of*.

1.1.3. Description

Descriptions provide a means to express a set of *feature structures* with respect to a given inheritance hierarchy. A type, a description at the end of a path, a path equation, a path in-equation, disjunction of descriptions and conjunction of descriptions are all description*s*. A description is satisfied by a set of *feature structures*. This set of *feature structures*, however, might be empty if a description is unsatisfiable.

Every feature structure can be called out by some description that includes no disjunction. Conversely, every description might have an empty feature structure or more than one satisfied feature structure. The members of the set of satisfying feature structures are pairwise incomparable.

1.2. The Algorithms

1.2.1. Subsumption

A feature structure represents partial information. The ability to specify that one structure is more specific than the other is important. The informational ordering over types is extended to *feature structures* by the *subsumption ordering. An intuitive explanation of subsumption over two feature structures* is "*a feature structure F subsumes another feature structure F′ if and only if every two paths which are shared in F are also shared in F′, and every type assigned by F to a path subsumes the type assigned to the same path in F′ in the type ordering*"[Carpenter, 1992, page 40].

1.2.2. Unification

The unification of two *feature structures* generates the minimal feature structure carrying more information than both argument objects together. Unification is a partial function, and where undefined, the two typed feature structures are said to be inconsistent. Linear time subsumption and unification algorithms exist, though typically a quasi-linear time algorithm it used.

1.2.3. π-resolution

Constraint resolution is a process by which a query, expressed as a description, is either satisfied or found to be unsatisfiable. The resolution mechanism proceeds, substructure (of a given feature structure) by substructure, until all the substructures satisfy (or fail) the constraint on their *types*. During the constraint resolution stages, the mechanism adds as just much information as is needed to the substructure in order to satisfy its constraints. This constraint resolution mechanism is called *π-resolution*. It captures a relationship between *descriptions* and *feature structures* and acts as the main exploration mechanism in the design space explorer.

2. CARRYING GEOMETRIC INFORMATION USING TYPED FEATURE STRUCTURES

We presume Typed Feature Structures are the only carriers of information in our design space explorer, then we must regard feature structures and their types as the lexicon within which we must work to devise appropriate representations. We choose feature structures representing geometric information to be instances of atomic types, that is, types carrying geometric information have no features. The type hierarchy itself thus must do almost all the duty of defining our geometric representation.

In conventional treatments of feature structures types hierarchies are formed from finite sets of types, though this is not a formal restriction. Our type hierarchies are infinite and thus implicit. In essence, a type hierarchy carrying a set of geometric objects is a graph with the node set being the set of geometric objects and the arc set capturing a "natural" relation of information specificity over the geometric objects.

There are lots of such "natural" relations, for example point set inclusion. Some of these relations have the BCPO properties required for inheritance hierarchies[Carpenter, 1992, page 13]. For example (Figure 3), in the domain of real intervals, the join of [-12.2, 17.74] and [5.023, 20.47] is consistent as [5.023, 17.74] under the point-set relation •, and a most general type (carrying an infinite interval □) is trivial to describe.

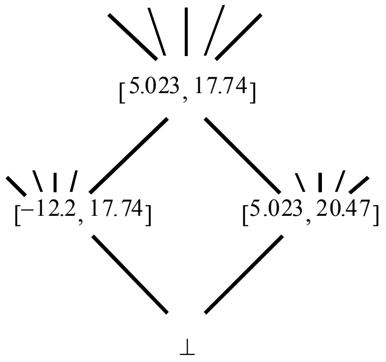


Figure 3. Example of a Complete Partial Order of Three Real Intervals and a Bottom

In summary, the problems of using types as carriers for domain relevant information are 1)continuous domains are infinite—thus we need a finitely describable way of representing infinite inheritance hierarchies and 2) the relations between domain objects must conform to an inheritance relation between the types carrying these objects.

Before describing of our solution to the problems described above, we need to address the terminology for the types we are using. The terms used here follow the literature [Carpenter, 1992, Burrow and Woodbury, 1998] very closely. First of all, the top of an inheritance hierarchy is called an absurd type, and the bottom is called a universal type. The standard types described in [Carpenter, 1992] are called Succession Types, as they succeed one another in an inheritance hierarchy specification. We describe a different kind of types called Order Types in which the ordering of types can be computed by direct comparison. The name for "Order Type" is given because of its ordering characteristic.

The elements of a set of types comprising both Succession Types and Order Types are called regular types, to distinguish them from absurd types. Consequently, regular types and the absurd type are from the universe of types. A detailed definition of the set of Order Types is given in the following section. The theory following that of Typed Feature Structures with the necessary extensions to include regular types is called Geometric Typed Feature Structures.

2.1. The characteristics of Order Types

Several differences between Order Types and Succession Types are identified. Those differences form the characteristics of Order Types below, which includes 1) Order Types are un-evaluated, 2) Order Types have no feats, 3) descriptions of Order Types have no paths, 4) Order Types carry no constraints and 5) some Order Types are intentional.

2.1.1. Order Types are un-evaluated

Regular types, that is, both Succession Types and Order Types, are defined by inheritance. In Succession Type finite inheritance hierarchies are explicitly constructed, either directly or through closure operations over ISA networks [Carpenter, 1992]. In the Order Type we do not explicitly store a type hierarchy. Instead we determine information consistency by direct examination of type tokens. Therefore, if the value has not been evaluated, the Order Type carrying this value will not exist in the type system. This reflects Order Type hierarchies are un-evaluated until users need to describe certain domain information that is implicit in them.

As a result, Order Types need not be evaluated until a description specifies a particular value, *i.e.*, Order Type hierarchies are presumed to exist but remain un-evaluated until needed in the universe of types.

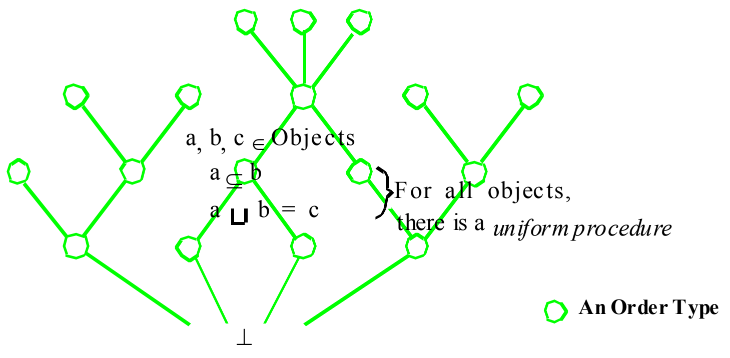
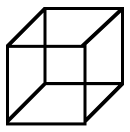


Figure 4. Example of an Order Type Hierarchy

2.1.2. Order Types have no feats

The main motivation for employing Order Types is to represent continuous domain objects, such as point-sets, numbers, and intervals. These objects have no sub-structures. If we want to represent the structure of domain information, we need to utilise higher-level structures. For example, in Figure 5, the topological relation between a solid and its faces can be organized in a feature structure-fashion, such that each feat of this feature structure carries a face as its value. Thus, there is no need to provide a sub-structure for Order Types and consequently, Order Types need no feats.



A cube

$$\begin{bmatrix}\text{Cube} & \\ \quad\text{FACE\_1:} & [\text{face}] \\ \quad\text{FACE\_2:} & [\text{face}] \\ \quad\text{FACE\_3:} & [\text{face}] \\ \quad\text{FACE\_4:} & [\text{face}] \\ \quad\text{FACE\_5:} & [\text{face}] \\ \quad\text{FACE\_6:} & [\text{face}] \\ \quad\text{GEOM:} & [\text{solid}] \end{bmatrix}$$

face and solid are order types

A *feature structure* representing the cube

Figure 5. An Example Represents the Properties of a Cube with a Higher Level Structure

2.1.3. Descriptions of Order Types have no paths

Descriptions contain four key components: types, values at the end of paths, path equations and and/or expressions of descriptions. Without feats of their own, Order Types have no path values and no path equations. Thus a description of Order Types comprises only a set of Order Types.

Let us recall that the definition of a path value $F@\pi$ is the value of the feature structure F at a path (which must be defined). If F is typed by an Order Type in Geometric Typed Feature Structures, there will be no path for F as F will have no feats at all. Thus, there will be no path values for F. Consequently, there are no path equations and path description for Order Types either, since Order Types have no path values. This means that the descriptions over Order Types are merely descriptions of those types, along with the disjunctions and conjunctions over them.

Typed Feature Structures Description	***Geometric Typed Feature Structures Description***
$\sigma \in$ Desc if $\sigma \in$ Type	$\sigma \in$ Desc if $\sigma \in$ Type
$\top \in$ Desc	$\top \in$ Desc
$\pi : \phi \in$ Desc if $\pi \in$ Path, $\phi \in$ Desc	
$\pi_1 = \pi_2 \in$ Desc if $\pi_1, \pi_2 \in$ Path	
$\phi \wedge \psi, \phi \vee \psi \in$ Desc if $\phi, \psi \in$ Desc	$\phi \wedge \psi, \phi \vee \psi \in$ Desc if $\phi, \psi \in$ Desc

Figure 6. The Definitions of Order Types have no Path Values.

In Figure 6, we compare Carpenter's definition of description[Carpenter, 1992, page 52] with Geometric Typed Feature Structures. Without path values, the definition of description becomes only the type description and the conjunction/disjunction of types, which are also types (or sets of types). A definition of a Geometric feature structure is given in Figure 7.

A Geometric feature structure over Order Types is a tuple $F = \langle Q, \bar{q}, \theta, \delta \rangle$ where

- Q: *a singleton, rooted at* $\bar{q}$
- $\bar{q} \in Q$: *the root node, and the sole member of Q*
- $\theta : Q \rightarrow$ Type : *a total node typing function*

Figure 7. Definition of a Geometric Typed Feature Structures Over Order Types.

The satisfaction of Order Types is also simplified by such a characteristic. As shown in Figure 8, the definition of satisfaction of Geometric Typed Feature Structures covers the absurd type, type satisfaction and conjunction/disjunction of satisfaction of two types.

Typed Feature Structures Satisfaction	***Geometric Typed Feature Structures Satisfaction***
$F \models \sigma$ if $\sigma \in$ Type and σ ❖ $\theta(\bar{q})$	$F \models \sigma$ if $\sigma \in$ Type and σ ❖ $\theta(\bar{q})$
$F \not\models \top$ for any $F \in$ ☞	$F \not\models \top$ for any $F \in$ ☞
$F \models \pi : \phi$ if F ✍$\pi \models \phi$	
$F \models \pi_1 = \pi_2$ if $\delta(\pi_1, \bar{q}) = \delta(\pi_2, \bar{q})$	
$F \models \phi \wedge \psi$ if $F \models \phi$ and $F \models \psi$	$F \models \phi \wedge \psi$ if $F \models \phi$ and $F \models \psi$
$F \models \phi \wedge \psi$ if $F \models \phi$ or $F \models \psi$	$F \models \phi \wedge \psi$ if $F \models \phi$ or $F \models \psi$

Figure 8. Comparison of Satisfaction of Typed Feature Structures with Satisfaction of Geometric Typed Feature Structures.

2.1.4. Order Types carry no constraints

As the domain described comprises a set of objects under a natural relation such as point-set containing, there is no need for further structure. What needs to be represented is nothing more or less than a chosen relation over a chosen set of domain objects. The only restriction is that the relation has to have the BCPO properties to form a well-behaved inheritance hierarchy, so that it can stand in an isomorphism to that hierarchy. Introducing *cons(t)*$\neq\phi$ will change that desired state. For example, in Figure 9, three solids are under a point-set superset relation. If the constraints over these Order Types are allowed then there will be no way of representing the cuboids marked *c*.

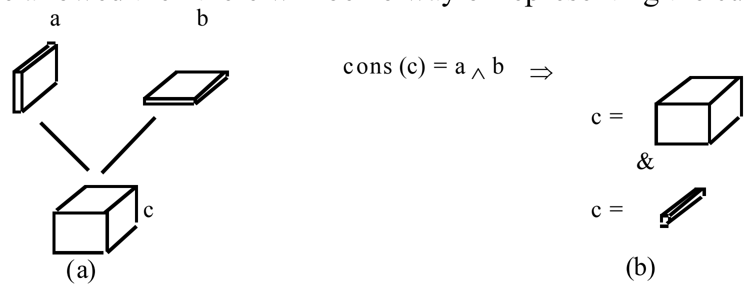


Figure 9. Order Types Need No Constraints.

2.1.5. Some Order Types are intentional

We shall now discuss some other properties of Order Types. Although it seems natural to think of an integer interval as an extensional type, it can be a nuisance since it generates structure sharing which prevents us from specializing a value by unification. For example, using Burrow and Woodbury's feature structure example[3][Burrow and Woodbury, 1998]:

```
human
  FOODSTUFF: animal
                 LEG_COUNT: interval_0_4
  PET: animal
         LEG_COUNT: interval_0_4
```

If intervals were extensional the above is the alphabetic variant of the following:

```
  human
  FOODSTUFF: animal
                 LEG_COUNT: [1] interval_0_4
  PET: animal
         LEG_COUNT: [1]
```

Then if a constraint identifies that we only eat four legged animals, we get

```
  human
  FOODSTUFF: animal
                 LEG_COUNT: [1] interval_4_4
```

[3]. The syntax of this example follows that used in the description of Kryos: an implementation of Typed Feature Structures.

```
PET: animal
      LEG_COUNT: [1]
```

Such structure is preventing us from owning snakes! If instead, we treat these types as being intentional, we avoid this problem.

2.2. The Model Space of Order Types

There are three criteria regarding of Order Types: domain relevance, the chosen relation has to be a BCPO, and efficiency of the computations corresponding to subsumption and unification. The values carried by the instances of Order Types must be domain relevant as the relation over these values that must be the inheritance hierarchy. One example of such objects regarding of geometry is two-dimensional rectangles. This 2D rectangle representation is used for several domain representation. Efficient constraint resolution algorithms computed over those rectangles are well developed. Another example is the 3D point-set, which is also of great interest in this dissertation. Relevant relations over those objects are also essential for Order Types. The example of the 3D point-sets, point-set containing is one relevant relation, that devoting all the points of *PsetA* are also points of *PsetB* if *PsetA is contained by PsetB*.

A BCPO contains these conditions: partial order condition (reflexivity, anti-symmetry, and transitivity), type consistency and a unique bottom. In addition, the existence of consistent joins is a defining characteristic of inheritance hierarchies for Typed Feature Structures. Informally, if a set of types is consistent, there is a single most general satisfied type that is subsumed by them all. This type subsumes all other types subsumed by all members of the set. A unique bottom is ensured by enforcing a universal type as described before. These two conditions associated with the partial order condition are the key of describing Order Types. In other words, Order Types and their associated relations have to confront the BCPO conditions for being included in inheritance hierarchies. A simplest BCPO is a tree.

Considering a mapping from a continuous domain, such as real intervals, an efficient domain representation is required. When one specifies an Order Type to carry certain domain information, such information has to be efficiently represented for a straightforward reason: computability.

Order Types are specified under domain relations with given mechanisms, so that the subsumption/unification relations are defined in each Order Type. Consequently, each Order Type is responsible to the algorithms over that specified type. Thus, when one defines a useful Order Type, extra care has to be put on the efficiency of subsumption/unification algorithms defined over that Order Type simultaneously. In addition, without path values, unification of two Order Types is simply based on the subsumption relations that are defined within the domain of interest.

3. TWO EXAMPLES

3.1. The lifted reals

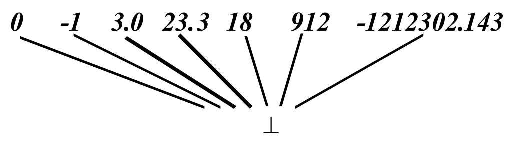


Figure 10. A part of Lifted Reals as an inheritant hierarchy

Given the ordered set $\overline{\Re}$ □ □ℜ, •□, where *a* • *b iff a*□*b*, we form $\overline{\Re_{□}}$, which we call the "lifted" reals by adding an element □, ℜ□ :□ℜ□{□}. We define an order relation '•' (ℜ is an an$\overline{\text{tic}}$hain[4]) on ℜ□ as ℜ□ □ □$\overline{\Re_{□}, •}$□. The definition is

a* • *b **if** ***a* • *b*** **in** $\overline{\Re}$ **(EQ 1)**

a* • *b **if** ***a*□□ and in b**🗎**ℜ□** **(EQ 2)**

Usually we refer to $\overline{\Re_{□}}$ simply as ℜ□. A partial inheritance hierarchy formed by □ℜ□, •□ is shown in Figure 10. With this definition, the four conditions described above apply as follows:

1. □ℜ□, •□ is a partial order.

Let *a, b, c*🗎ℜ□,

reflexivity: *a* • *b* is true by definition (see EQ 1)

anti-symmetry: *a* • *b* and *b* • *a* imply *a*□*b*

by exhaustive enumeration, if *a* • *b*, either *a* = *b* or *a* =□, *b≠a*,

in the former, *a* • *b* and *b* • *a* ⇒ *a* • a and *a* • *a* ⇒ *a* • *a* ⇒*a* = *b*

in the latter, *a* • *b* but 💻*b* • *a* □(*a* • *b*) and (*b* • *a*) and *b≠a* are false.

□ (*a* • *b* and *b* • *a* imply *a*□*b*) is true. ▯

transitivity: *a* • *b* and *b* • *c* imply *a* • *c*

by exhaustive enumeration.

If a • *c, either a=b or a=*□, *a≠b,* **(EQ 3)**

in the former, *a* • *b* and *b* • *c* ⇒ *a* • a and *a* • *c* ⇒ *a* • *c*

in the latter, *a* • *b* and *b* • *c* ⇒ □• *b* and *b* • *c* ⇒

Note that *b*=*c*, as *b≠a* from [EQ3].

□• *b* and *b* • *c* ⇒□• *c* and *c* • *c* ⇒□• *c*

but *a*=□, so *a* • *c*.

□ (*a* • *b* and *b* • *c* imply *a* • *c)* is true. ▯

2. □ℜ□, •□is a BCPO.

As mentioned before, a tree is a BCPO. A "lifted" anti-chain is a tree, thus, □ℜ□, •□is a BCPO. ▯

3. There must be an efficient means of computing subsumption and unification. We take □ℜ□, •□ as an inheritance hierarchy such that *a* ❖ *b* = *a* • *b*.

Subsumption

□ ❖ *a, a* 🗎ℜ, and *a* ❖ *a, a*🗎ℜ□

The subsumption algorithm is trivial and computed based on its simple symbol comparison.

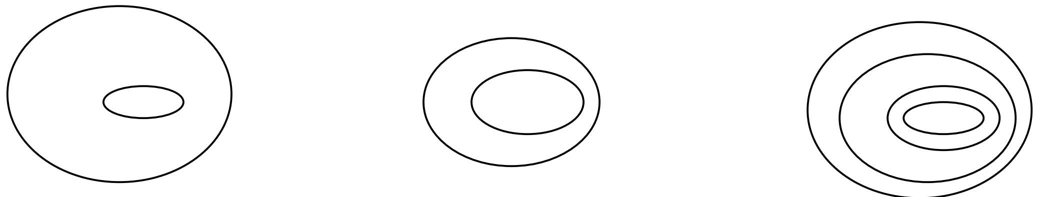
Unification

□ ⧫ *a* = *a, a* 🗎ℜ□, and *a* ⧫ *a* = *a, a*🗎ℜ

The unification algorithm is trivial and computed based on its simple symbol comparison.

4. There must be a way to specify members of the set. Elements of ℜ□ stand as names for themselves. Explicit expressions over ℜ with the usual operators of from { □, □, ×, ÷}, can also stand as names for elements of ℜ□.

3.2. The IGOSet intervals

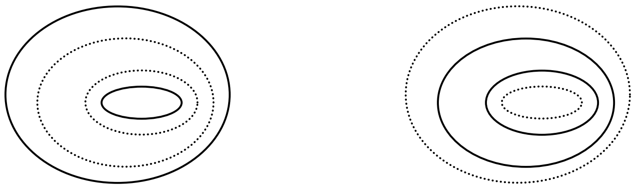


IOPSet a *IOPSet b* *A cell complex comprises both a and b*

4. "The ordered set P is an antichain if x<y in P only if x□y." [Davey and Priestley, 1994].

Figure 11. Example of Two IOPsets a and b, Such that a ❖ b

The ordered set *PSet*□ = ♒[*PSet*$_{\subseteq}$, *PSet*$_{\supseteq}$]❖, □♓where *PSet* is the set of all point-set and □ denotes inner growing outer shrinking relation. We call the elements of *PSet*□ the *IGOSET intervals*. First we define *pset*(*n*) as a polymorphic function giving the point-set corresponding to an element in



PSet$_{\subseteq}$ or *PSet*$_{\supseteq}$. *Pset*(*a*) and *pset*(*b*) are comparable by ⊆ and ⊇ irrespective of the set membership of a and b in *PSet*$_{\subseteq}$ and *PSet*$_{\supseteq}$. We distinguish the set ∞∈ *PSet* as the set of all points and the set φ∈ *PSet* as the null set.

We refer to *PSet*□ as *IOPSet* (inner-outer point-sets). We write an IGOSet interval a as $a = [a_i, a_o]$ and use a_i and a_o to denote the inner and outer bound of the interval respectively, such that *pset*(a_i)⊆*pset*(a_o). A partial inheritance hierarchy formed by *PSet*□ is shown in Figure 12. The definition of *PSet*□ is

Let a ∈ *PSet*$_{\subseteq}$, b ∈ *PSet*$_{\supseteq}$,[*a*,*b*] ∈ *PSet*□, if *pset*(*a*) ⊆*pset*(*b*). **(EQ 4)**
a □ b if $a_i \subseteq b_i$ and $a_o \supseteq b_o$ **(EQ 5)**

1. *PSet*□ is a partial order.
   Let a, b, c ∈ *PSet*□
   reflexivity: a □ a
   As a special case shown in equation: 5, a □ a is true by definition.
   anti-symmetry: *a* □ *b* and *b* □ *a* imply *a*□ *b*
   The definition of *a* □ *b* relies directly on the relations ⊆ and ⊇ over the sets *PSet*$_{\subseteq}$ and *PSet*$_{\supseteq}$ respectively. Both *PSet*$_{\subseteq}$ and *PSet*$_{\supseteq}$ are partial orders and anti-symmetric. Therefore, *PSet*□ is anti-symmetric as its definition (see EQ5) is simply the conjunction of the above two conditions. □
   transitivity: *a* □ *b* and *b* □ *c* imply *a* □ *c*
   By the argument used for anti-symmetry, *PSet*□ also satisfies the transitivity condition. □

2. *PSet*□ is a BCPO.

Bottom. [□$_{PSet_{\subseteq}}$, □$_{PSet_{\supseteq}}$] □ a, a ∈ *PSet*□.
A least upper bound.
Following the definition of upper bound[5] from [Davey and Priestley, 1994],
   Let S∈ ℘(*PSet*□), and let S be consistent, that is, there is some x∈*PSet*□ such that s□x, s∈S. Informally, two point-set intervals are consistent if the union of their inner sets is inside the intersection of their outer sets. The least upper bound of S is
      [□s_i, ∩s_o], s∈S, if (□s_i ⊆ ∩s_o), *undefined otherwise.* □

3. There must be an efficient means of computing subsumption and unification. We take *PSet*□ as an inheritance hierarchy such that *a* ❖ *b* = *a* □ *b*. An example is shown in Figure 11.

[5]. "Let P be an ordered set and let *S*⊆*P*. An element x∈*P* is an upper bound of S if s⌛x for all s ∈S. ... The set of all upper bounds of S is denoted by S^{u}"[Davey and Priestley, 1994]

Subsumption

The subsumption algorithm is computed based on its lower bound ($PSet_{\subseteq}$) and upper bound ($PSet_{\supseteq}$) subsumption algorithms, which is linear in the non-manifold representation.

Unification

The unification algorithm is computed based on its lower bound ($PSet_{\subseteq}$) and upper bound ($PSet_{\supseteq}$) unification algorithms, which is linear in the non-manifold representation.

4. There must be a way to specify members of the set $PSet_{\square}$. Primitive parametric objects, transformations of these and combination of both under the un-regularized or regularised Boolean operation can stand as names of inner/outer bounds of elements of $PSet_{\square}$.

Let's define two operations: $\square$ and **o** as:

$$a \square b = [(a_i \square b_i), (a_o \cap b_o)],\ a, b \in PSet_{\square}, (a_i \square b_i) \subseteq (a_o \cap b_o)$$

$$a \circ b = [(a_i \cap b_i), (a_o \square b_o)],\ a, b \in PSet_{\circ}, (a_i \cap b_i) \subseteq (a_o \square b_o)$$

Thus, the set of such expressions using these two operations be called Exp. Exp is not close. Other algebras, for example, sweeps are also possible. An advantage of using these two operations is that it is possible to write expressions that name subsumed elements of $a \in$ *IPPSet*. These comprise any expression that can be reduced to $a \square b$, $b \in$ Exp.

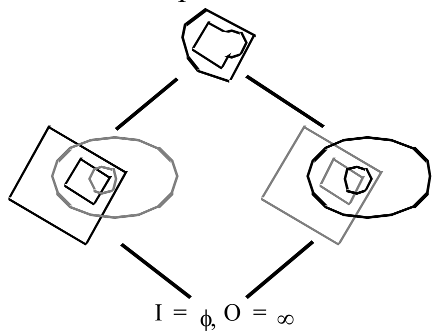


Figure 12. A Part of IOPSet as an Inheritance Hierarchy

4. CONCLUSION

Following the examples above, two kinds of Order Types are revised. Each provides a certain structured continuous information and a meaningful relation. This feature demonstrates the key approach of Geometric Typed Feature Structures and their usability. Following this approach, a new group of geometric objects are discovered and their characteristics remain unidentified. The benefits from the design space explores are clear—structured geometric information. With specifying in a certain relation such as point-set inclusion, the geometric information can be organized in a structured way—Order Types. It's the same in the continuous domain such as reals and real intervals.

In addition, as based on the same objects (Order Types), geometric information and continuous domain are treated as the same data structure. Therefore, the unification and subsumption algorithms can be applied to both uniformly. This provides a different approach for carrying non-geometric information within the geometric objects and simplifies the unifcation process of design space explorers notably. With these significances, Geometric Typed Feature Structures resolve an important issue in the field of construction information—carrying geometric information.

5. REFERENCES

Ait-kaci, H. (1984). *A lattice-theoretic approach to computation based on a calculus of partially ordered type*s. PhD thesis, University of Pennsylvania.

Burrow, A. and Woodbury, R. (1999). Pi-resolution in design space exploration. CAADFutures 1999.

Carpenter, B. (1992). *The Logic of Typed Feature Structures with applications to unification grammars, logic programs and constraint resolution*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Chang, T.-W. (1999). *Geometric Typed Feature Structures:Toward Design Space Exploration*. PhD thesis, The University of Adelaide, Adelaide, SA, Australia.

Davey, B. and Priestley, H. (1994). *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks. Cambridge University Press, Cambridge, U.K.

Kaplan, R. and Bresnan, J. (1982). Lexical-functional grammar: A formal system for grammatical representation. The Mental Representation of Grammatical Relations, pages 173-281.

Sedgewick, R. (1990). Algorithms in C. Addison-Wesley Publishing, New York.

Shieber, S. (1986). An introduction to unification-based approaches to grammar. In CSLI Lecture Notes, Volume 4. Center for the Study of Language and Information.

Smithers, T. (1992). Design as exploration: Puzzle-making and puzzle-solving. In AID92, Workshop Notes, Exploration-based models of design and search-based models of design. Carnegie-Mellon University, Pittsburgh, PA.

Woodbury, R.F. (1987). Strategies for interactive design systems. In Kalay, Y., editor, The Computability of Design, volume2 of Principles of Computer-Aided Design, pages 11-36. John Wiley and Sons.