The Sheaf Data Model

Part 1: Objects

David M. Butler

Limit Point Systems, Inc.

Abstract

This document establishes the mathematical definition and interpretation of the objects of the sheaf data model.

1 Introduction

This document describes the mathematical definition and interpretation of the objects of the sheaf data model, emphasizing the definition and interpretation of schema. A primarily non-mathematical discussion of the context and motivation for the model are given in (1).

The document is organized as follows. In sections 2 through 6 we develop the notion of lattice-ordered schema for relational tables. In sections 7 and 8 we introduce restriction and the notion of a sheaf of lattice ordered relations, which we use in section 9 to define the sheaf data model. The remaining sections discuss further features of schema, in particular the schema of map types.

We assume the reader is familiar with the basic ideas of partially ordered sets and lattices, see for instance (2).

2 Object-relational mapping

We begin by establishing a relational (table) interpretation for the notion of type as present in typical object-oriented languages. For concrete examples, we will use the C++ programming language. This language is of major practical importance and, for our purposes at least, its notion of type is close enough to many other languages that our results should apply generally.

In C++, the types we are interested in are primitive types and class types. There is a given set of primitive types (int, float, etc). Each primitive type implies a finite set of values, the implementation of which is not specified.

Class types are programmer-defined types. Each class specifies a collection of data members and a collection of function members. Since we are focusing on a data model,

we will be mostly concerned with the data members. In C++, an object can contain other objects, which are referred to as subobjects (3 pp. 7-8). A subobject can be a data member subobject or a base class subobject. We'll refer to the set of immediate subobjects associated with a class as the subobject schema, with each subobject specified by a name and a type.

We would like to associate a relational table with each class. The obvious approach is for each row in the table to represent an object of the class and each column of the table to represent a subobject. That is, we'd like the relation schema to be the same as the subobject schema. The difficulty in this approach is that a subobject can be of any type, another class type in particular, but an attribute of a relation must be a primitive type.

3 Some notation

Before we resolve this object-relational schema mismatch, there's an important notational issue to deal with. Associated with a type T are two important sets: the set of instances of the type and the set of subobjects of the type, that is, the subobject schema of the type. Informally, these two sets correspond, respectively, to the set of rows and the set of columns in the table representing the type.

We need a notation that clearly distinguishes these two sets while at the same time maintaining the close connection between them. We will denote a type with an upper case Roman letter or a name with an initial upper case letter, for instance "T" or "A_type_name". We will refer to the set of instances of the type (the rows in the table) by underlining: $\underline{T}$ is the row set of T. We will refer to the set of subobjects (the columns of the table) with a vertical bar: $T|$ is the column set of T.

4 From subobject schema to lattice schema

We can resolve the object-relational schema mismatch issue by generalizing the subobject schema from a set of subobjects to a lattice of subobjects. We start with a simple example, then describe the general procedure.

4.1 An example

We have two types: R2, a two dimensional vector, and Phase_space, the combination of a vector r and a vector p, both of type R2.

```
class R2
{
 public:
  double x;
  double y;
  R2(double tuple[2]);
  R2& operator=(double tuple[2]);
}

class Phase_space
{
 public:
```

```
  R2 r;
  R2 p;
  Phase_space(double tuple[4]);
  Phase_space& operator=(double tuple[4]);
}
```

The subobject schema for these two are:

schema(R2) = R2| = {(x: double|), (y: double|)}

schema(Phase_space) = Phase_space| = {(r: R2|), (p: R2|)}

It's useful to cast these schema in the form of a relation "has_subobject". The schema for R2| becomes:

R2| has_subobject (x: double|)

R2| has_subobject (y: double|)

and the schema for Phase_space becomes:

Phase_space| has_subobject (r: R2|)

Phase_space| has_subobject (p: R2|)

Now form the sum (disjoint union) of the Phase_space schema with the schema for each of its subobjects. Although r and p are both of type R2| and hence have the same schema, the sum treats these identical copies as distinct, so we get:

Phase_space| has_subobject (r: R2|)

Phase_space| has_subobject (p: R2|)

r: R2| has_subobject (r.x: double|)

r: R2| has_subobject (r.y: double|)

p: R2| has_subobject (p.x: double|)

p: R2| has_subobject (p.y: double|)

We now interpret the has_subobject relation as the cover relation for a partially ordered set, the schema poset. We can visualize the schema poset using a Hasse diagram, which is a directed acyclic graph in which the nodes represent the poset members and the links represent the cover relation. The graph is drawn so that if member $a \leq b$, then a is below b on the page. The Hasse diagrams for the schema posets of R2 and Phase_space are shown in Figure 1(a) and Figure 2(a), respectively.

Associated with every finite poset P is a finite distributive lattice $L = \mathcal{O}(P)$, the set of down sets of P ordered by inclusion. Conversely, P = J(L), the set of join irreducible members of L. We define the schema of Phase_space to be the lattice associated with the schema poset. The Hasse diagrams for the schema lattices for R2 and Phase_space are shown in Figure 1(b) and Figure 2(b), respectively. In Appendix A we explicitly construct the schema lattice for Phase_space from the schema poset.

4.2 General procedure

The procedure we used in the example above is general. Given the subobject schema for a type and the subobject schema for the types of its subobjects, form the sum of the has_subobject relations of the class and its subobjects. Do this recursively for subobjects of subobjects; the recursion terminates on subobjects of primitive type. Interpret the resulting relation as the cover relation of the schema poset and define the schema lattice to be the finite distributive lattice generated by the schema poset. So in general we have schema lattice = $\mathcal{O}$(schema poset) and schema poset = J(schema lattice).

4.3 Some more notation

We need some more notation to denote the ordered structures introduced in the preceding sections. We will denote a cover relation by :< and an order relation by ≤. A partially ordered set will be denoted using the same symbol as its unordered base set and an italic font: $T = (\mathrm{T}, \leq)$. The lattice associated with a poset will be denoted with the same symbol as the poset and a script font: $\mathcal{T} = \mathcal{O}(T)$.

So for instance, the column set of a type T is T|, the schema poset is $T|$, and the schema lattice is $\mathcal{T}|$.

4.4 Interpretation of the lattice schema

The lattice schema has the following features:

1. The lattice schema represents a collection of types related by subobject inclusion.

2. Each join-irreducible member ("jim") of the lattice corresponds to a type explicitly specified in the has_subobject relation. This captures the notion that an explicitly specified type is not just the collection of subobjects specified by the schema, it may have additional features, such as member functions, that are outside the scope of the schema.

3. Each join reducible member ("jrm") of the lattice corresponds to a type implicitly generated by the lattice construction. An implicit type is precisely the collection of subobjects represented by the schema, a struct in C++.

4. Each atom corresponds to a primitive type. This captures the notion that representation of a primitive type is outside the scope of the schema.

5. The subobject schema for a given member is specified by the set of maximal jims in its strict down set. Equivalently, we can define the subobject schema of a given member to be the join of the maximal jims, which is a unique member of the lattice. So we can think of the subobject schema as either a set of subobjects or as a member of the lattice. If the given member is a jim, the subobject schema is the only member in its lower cover. If the given member is a jrm, the subobject schema member is the same as the given member.

6. The relation schema for a given member is specified by the set of minimal jims (that is, atoms) in its down set. Equivalently, the relation schema is the join of the minimal jims. So we can think of the relation schema as either a set of primitive subobjects (attributes) or a member of the lattice.

7. The relation schema for a member t| specifies a Cartesian product space which we refer to as the attribute space for t| and denote by $A_{t|}$. Each instance t of the type specified by t| has a corresponding tuple in $A_{t|}$. We assume that every non-primitive type has both a constructor and an assignment operator that take as input a tuple of the type specified by the associated relation schema. We also assume that each type has a conversion operator that produces as output the relation tuple associated with an instance. We can thus convert back and forth between objects and tuples.

5 The table metaphor for lattice schema

The table metaphor for the relational data model visualizes a relation as a table with column headings describing the schema. We can extend this metaphor to lattice schema. We have already visualized a schema poset or lattice using a Hasse diagram. The table metaphor for lattice schema replaces the column headings with the Hasse diagram for the schema poset or lattice. A simple example is shown in Figure 1(c).

6 From lattice schema to subobject schema

In the previous section, we showed how we can generate a lattice schema given a collection of subobject schemas. We also showed how class types are associated with the members of the lattice and how primitive types are associated with the atoms of the lattice. We can reverse the procedure: given any lattice and an assignment of a primitive type to each atom of the lattice, we can interpret the lattice as a schema. We can interpret each member of the lattice as a type with a subobject schema specified by the maximal jims in its down set and a relation schema specified by the minimal jims in its down set.

7 Restriction and sheaves of relations

If t| and t|' are schema members with $t|' \leq t|$ we say that t|' is a subtype of t|. If o is an object of type t|, the subobject of o specified by t|' is defined as the object o' of type t|' with tuple equal to the projection of the o tuple onto the relation schema of t|'. We also refer to object o' as the restriction of o to t|'.

Let $\top|$ denote the top member of the schema lattice $\mathfrak{J}|$ for a type T. $\top|$ defines a Cartesian product space $A_{\top|}$, as described in section 4.4 item 7. Let $R_{\top|}$ denote a subset of $A_{\top|}$, that is, a relation. Given any member t| of the schema with $t| \leq \top|$, we can restrict each tuple in $R_{\top|}$ to t|. This process creates another relation $R_{t|}$ and a restriction map:

$\rho_{t|,\top|}: R_{\top|} \rightarrow R_{t|}$ (note the order of the subscripts)

that associates each tuple in $R_{T|}$ with a tuple in $R_{t|}$. For any $t|'' \leq t|' \leq t|$, we can create $R_{t|''}$, $R_{t|'}$, and $R_{t|}$ and the restriction process also defines restriction maps between them, satisfying the conditions

$$\rho_{t|,t|} = \text{identity map}$$

$$\rho_{t|'',t|} = \rho_{t|'',t|'} \circ \rho_{t|',t|}$$

If we create the restricted relation $R_{t|}$ for every member $t|$ of the schema lattice, we get a family of relations:

$$R = \{R_{t|} : t| \in \mathcal{J}|\}$$

We define the restriction relation $\preccurlyeq$ on R with $R_{t|} \preccurlyeq R_{t|'}$ if and only if there exists a restriction map $\rho_{t|',t|}$. Given the conditions satisfied by the restriction map, $\preccurlyeq$ is reflexive and transitive, so it is a preordering and $R = (R, \preccurlyeq)$ is a preorder.

The above construction associates with each $t| \in \mathcal{J}|$ a relation $R_{t|} \in R$ with $R_{t|} \preccurlyeq R_{t|'}$ precisely when $t|' \leq t|$ (note the order reversal). Such an order reversing map from a finite distributive lattice to a preorder of relations is called a sheaf of relations. The schema lattice is referred to as the source of the sheaf and the preorder of relations is referred to as the target of the sheaf. We will call the largest member of the target, the table which is restricted to produce all the other members of the target, the top table. The sheaf of relations associated with the table of Figure 1(c) is shown in Figure 3(a).

We can take the sheaf construction one step further. Noting that the construction so far specifies a lattice order only for the columns of a relational table and that the row order is entirely independent of the column order, we can impose an order on the rows as well. We assume that the top table has an externally specified ordering relation, so the rows form a partially ordered set, which defines an associated finite distributive lattice. The notation introduced in section 4.3 for columns can be applied to the rows as well. With this extension, a sheaf of relations thus becomes a sheaf of lattice-ordered relations.

8 The table metaphor for lattice ordered relations.

We can extend the table metaphor to lattice ordered relations by providing a row graph as well as a column graph. An example is shown in Figure 4. Using this metaphor, we can refer to a sheaf of lattice ordered relations as a "sheaf table".

9 The sheaf data model

The objects of the sheaf data model are sheaves of lattice-ordered relations. A data base is a collection of sheaf tables. Each table has an associated schema lattice and the schema for the table is specified as a member of the schema lattice. The conventional relation schema for the table, the columns of the table, are defined by the minimal jims in the down set of the schema member, as described in section 4.4.

The schema lattice of a given table is given by the row lattice of some other table. Every table thus has another table as its schema. This recursion is terminated in a special table called the primitives schema table. The primitives schema table is its own schema; its rows specify its columns. Figure 5 shows a simple example. The primitives_schema is the schema for all schema tables. The tuple associated with any schema member thus provides the type in the host language (e.g. C++) and the memory allocation requirements of an object described by the schema member.

Schema in the sheaf data model are first class objects and the collection of types in a database forms a dependent type system. As shown below, this is an essential feature for representing map types.

10 Subtypes and inheritance

The remainder of this document discusses additional features of schema in the sheaf data model. We begin by clarifying the relationship between the notion of subtype in the sheaf model and the usual notion of inheritance. The lattice definition of subobject given in section 7 is semantically similar to the corresponding definition in C++, although the syntax is different. Consider the following modification of the Phase_space class defined above:

```
class Phase_space2 : public R2
{
 public:
  R2 p;
  Phase_space(double tuple[4]);
  Phase_space& operator=(double tuple[4]);
}

Phase_space2 ps, *psp = &ps;
```

The schema poset for Phase_space2 is shown in Figure 6.

Phase_space2 inherits R2, so object ps has an R2 subobject. The Phase_space2 object ps can be restricted to its R2 subobject by an up-cast:

```
R2 &r2 = ps
```

The members of the inherited subobject can be also referenced directly as members of ps, ps.x and ps.y, or more explicitly using the scope operator, psp->R2::x and psp->R2::y.

Since member p is declared by value (as opposed to pointer or reference), ps has a second R2 subobject named p. The member access operator ps.p restricts the ps object to its R2 subobject p.

Thus, there are two main differences in C++ between an inherited subobject and a data member subobject:

1. an inherited subobject doesn't have an explicit name and is referred to by type, while a data member subobject is referred to by name, and

2. an inherited subobject can be accessed via up-casting, as long as the inherited type is unique (no repeated inheritance), while a data member subobject is accessed via the member access operator.

The lattice schema notion of subtype, as described so far, does not distinguish between inheritance and aggregation (data membership), both mechanisms are represented by subtype inclusion. The lattice schema notion of subobject is the same as the C++ notion of subobject, but the syntax for referencing a subobject is different. Every subobject can be accessed via restriction to a named subtype.

We can however distinguish between inheritance and aggregation with a simple naming convention, not formally part of the model. Namely, a subtype is directly inherited if its local name (the part of the name after the last dot) is the same as its type. A subtype is indirectly inherited (or just inherited for short) by a type if there is a chain of direct inheritance between the type and the subtype. This convention is adequate for many practical cases.

11 Schema of map types

So far we've dealt with explicit class schema and the lattice of subtypes generated from them. Now we turn our attention to the schema for maps between types. We want to specify a type S (for "section") such that an instance s of S represents a map from a domain type B (for "base") to a range type F (for "fiber"). In other words, the row set of S is a set of maps between the row set of B and the row set of F:

$$\underline{S} = \{\underline{s}: \underline{B} \rightarrow \underline{F}\}$$

Note that we are concerned here with ordinary maps on the row set B, not with order-preserving maps on the row poset *B*; we are ignoring the row order for now.

What is $\mathbb{S}|$, the schema lattice for S? We'll approach this question in two steps. First we'll develop the schema poset, then derive the schema lattice.

11.1 Schema poset

The notion of a map or function is usually introduced in elementary mathematics as a procedure or rule that assigns a member of the range to each member of the domain. In more advanced treatments, a map is defined as a special kind of relation, as a set of pairs

$$\{(b \in \underline{B}, f \in \underline{F})\}$$

containing exactly one such pair for each member of B. We'll start with the map as procedure approach and return to the map as relation later.

11.1.1 Local map objects and schema

Recall that, as described in section 4, a general object has a schema poset, data members, a constructor and an assignment operator that take a relational tuple as input, a

conversion operator that produces a tuple as output. A local map object is a general object with an additional member function, the evaluation method, which provides the procedure for the map. The evaluation method takes as input a member b of the domain and, using the data members of the object, it produces as output the member f of the range that is the value of the map at b. The schema for a local map implicitly specifies the existence of the evaluation method, but is otherwise just a general object schema.

The class specified by such a local map schema can be viewed as a parameterized family of procedures. The associated relation schema specifies a Cartesian product space, A, the attribute space of the local map. The evaluation method is then formally a map:

$$\text{evaluation: } \underline{B} \times A \rightarrow F$$

We can view this map on $\underline{B} \times A$ as a family of maps on $\underline{B}$, with one member of the family for each tuple in A. Since the data members of a specific instance of the class correspond to a tuple from A, there is a one to one correspondence between members of the family of maps and instances of the class. That is, each local map object s can indeed be viewed as a map

$$\underline{s}.\text{evaluation: } \underline{B} \rightarrow \underline{F}$$

with the values of the attributes relegated to the background.

It will be convenient in the following to cast the evaluation method into a slightly different form. Since we assume every type F has a constructor that takes a relational tuple as input, we can define the evaluation method by

$$\text{evaluation: } \underline{B} \times A \rightarrow A_F$$

where A_F is the attribute space for the type F. The previous definition is then equivalent to composition of the revised redefinition with the constructor of F. We will use this revised definition in the remainder of the document.

11.1.2 Global map objects and schema

We refer to the map objects above as local because for a given map s, there may not be a single procedure that is valid for the entire domain. In general, the domain may be decomposed into multiple, possibly overlapping subsets, with a different procedure defined for each subset. A given procedure applies only to the members of B contained in the subset the procedure is associated with.

More formally, let $\mathcal{P}(\underline{B})$ denote the power set of B, that is, the set of all subsets of B. $\mathcal{P}(\underline{B})$ is a poset and a lattice under the inclusion ordering. We assume that we have specified an evaluation subset, $\underline{E} \subset \mathcal{P}(\underline{B})$, that contains precisely the subsets of B on which the procedures are defined. Since $\mathcal{P}(\underline{B})$ is a poset, so is $\underline{E}$.

If the map is to be defined everywhere on B, then $\underline{E}$ must cover B, that is:

$\underline{E} = \{e_0, e_1, e_2 ...\} \subseteq \mathcal{P}(\underline{B})$ and $\bigcup_i e_i = \underline{B}$

where the union is computed in $\underline{B}$.

If the members of $\underline{E}$ considered as subsets of B overlap, multiple procedures may apply to a given member of $\underline{B}$. An evaluation member for $b \in \underline{B}$ is any member of $\underline{E}$ that contains b. The evaluation subset for $\underline{b}$, $\underline{E}_b$, is set of all evaluation members for b and is given by:

$\underline{E}_b = \{b\}\uparrow \cap \underline{E}$

where $\{b\} \in \mathcal{P}(\underline{B})$ is the singleton subset containing b and both the up-set and the intersection are computed in $\mathcal{P}(\underline{B})$.

A global map schema s| is a schema that contains a local map schema $s|_e$ as subtype for each member e of $\underline{E}$. We assume there is a map, the evaluation-schema map:

$\varepsilon: \underline{E} \rightarrow S| : e \mapsto s|_e$

that defines the association between evaluation members and local schema members.

A global map object is an object instantiated on a global map schema. The attributes of the global map object are a tuple in the attribute space defined by the relation schema associated with the global map schema.

The restriction $\underline{s}_e$ of a global map object $\underline{s}$ to an evaluation member e is the subobject defined by restricting $\underline{s}$ to the subtype $s|_e = \varepsilon(e)$. The attribute tuple of the restriction is the projection of the global attribute tuple onto the relation schema associated with the subtype.

The evaluation method of a global map object is defined by restriction to an evaluation member:

$\underline{s}$.evaluation(b) = $\underline{s}_e$.evaluation(b) with $e \in \underline{E}_b$

11.1.3 Compatibility

If the evaluation subset $\underline{E}_b$ for a given domain member b contains more than one member, say $\underline{E}_b = \{e, e'\}$, then we can evaluate the map $\underline{s}$ in each member: $\underline{s}_e$.evaluation(b) and $\underline{s}_{e'}$.evaluation(b). If $\underline{s}$ is to define a map from $\underline{B}$ to A_F, then it must be single-valued at each point of $\underline{B}$. This defines a compatibility condition:

$\underline{s}_e$.evaluation(b) = $\underline{s}_{e'}$.evaluation(b) for all $e, e' \in \underline{E}_b$

This compatibility condition is a constraint on both the global map schema and the map attribute tuple. The evaluation methods for the local map subtypes must be chosen so that

compatibility can be satisfied for some subset of the global attribute space and then the attribute tuple for the given map must lie in the compatible subspace.

11.1.4 Constructing global map schema posets

A global schema poset and the associated evaluation schema map can be constructed in many different ways. Any poset and map combination that satisfies the compatibility condition is a valid schema. While there is no exhaustive enumeration or taxonomy for global schema, it is useful to describe a few constructions. We will construct our examples using class R2, which we introduced previously, as the range type and the very simple domain shown in Figure 7. This combination of domain and range is complex enough to exhibit the constructions we want to discuss, but simple enough we can still conveniently draw diagrams of the constructions.

11.1.4.1 Coarsest schema

The simplest schema is constructed by choosing the coarsest evaluation subset, a single member, as shown in Figure 8(a). This corresponds to a single evaluation method that covers the entire domain and the schema has a single member, as shown in Figure 8(b). The schema as shown assumes four attributes, r0, θ0, r1, θ1, which are the polar components for the result at v0 and v1. The evaluation method is:

```
void evaluation(B& b, A_R2& result)
{
  if(b == v0)
  {
    result.x = r0*cos(θ0);
    result.y = r0*sin(θ0);
  }
  else if(b == v1)
  {
    result.x = r1*cos(θ1);
    result.y = r1*sin(θ1);
  }
  else  // b == seg
  {
    result.x = 0.5*(r0*cos(θ0) + r1*cos(θ1));
    result.y = 0.5*(r0*sin(θ0) + r1*sin(θ1));
  }
}
```

Of course, this particular schema and evaluation method are just for illustration. Any evaluation method and corresponding set of attributes is possible.

11.1.4.2 Finest schema

The opposite extreme is to choose the finest evaluation subset, as shown in Figure 9(a). Every member of the domain gets its own evaluation method, as shown Figure 9(b). This example assumes the local map type for each evaluation member is the same, each has attributes which are polar components and each uses the same evaluation method:

```
void evaluation(B& b, A_R2& result)
{
  result.x = r*cos(θ);
  result.y = r*sin(θ);
}
```

As with the previous example, this particular choice of evaluation method and attributes is just for illustration.

11.1.4.3 Intermediate schema

An intermediate construction is shown in Figure 10. The evaluation subset contains two members, Figure 10(a), and the corresponding schema contains two local members, Figure 10(b). Once again, the attributes are polar components and the evaluation methods are the obvious variations on the previous examples.

11.1.4.4 Domain order embedding

A particularly useful construction is to require the evaluation subset to be a down-set in $\mathcal{P}(\underline{B})$ and require the evaluation-schema map ε to be an order embedding. In Figure 11, we show the example of Figure 10 with $\underline{E}$ extended to a down-set and embedded in the schema. As before, the attributes are polar components and the evaluation methods are the obvious variations on the previous examples

This construction has the desirable property that since $\underline{E}$ is a down-set, the schema explicitly supports all meaningful restrictions. For instance, the schema in Figure 10(b) specifies an evaluation method on the subset {v0, v1}, which means restriction to the subset {v0} is meaningful, but the schema does not have sufficient resolution to specify the restriction. The schema in Figure 11 does.

The schema in Figure 11 also exhibits another desirable property: the attribute set of the restriction to {v0} or {v1} is a a strict subset of the attribute set of {v0, v1}. This is typical of order embedding schema, although not strictly necessary, as the next example will show.

11.1.4.5 Product order embedding

An even more useful construction is to require the evaluation subset to be a down-set in $\mathcal{P}(\underline{B})$ and require the evaluation-schema map ε to be an order embedding of $\underline{E} \times F|$, where $F|$ is the schema poset for the range type. In Figure 12, we show a product order embedding for the example of Figure 10.

A schema of this form provides explicit support for restriction to any combination of domain subset and range component.

This schema serves as a further example of the interaction between restriction and attribute sets mentioned in the previous example. For instance, the restriction from ({v0,

v1}, R2|) to ({v0}, R2|), reduces the attribute set, but the restriction from ({v0}, R2|) to (v0, x) does not.

11.1.4.6 Product order isomorphism.

We can make the even more stringent requirement that ε be an order isomorphism, that is, an onto order embedding, as shown in Figure 13.

11.1.4.7 Map as relation

As promised, we return to the map as relation approach we mentioned at the beginning of section 12. If we adopt the product order isomorphism schema and choose $\underline{E}$ to be the set of atoms of $\mathcal{P}(\underline{B})$, that is:

$\underline{E} = \{ \{b\}, \text{for all } b \in \underline{B} \}$

we get a schema with a copy of the range schema for each member of the domain, as shown in Figure 14. A map instantiated on this schema contains an instance of the range type for each member of the domain and hence is equivalent to the map as relation approach.

11.1.4.8 Piece-wise order embedding procedure

The product order embedding property is an extremely useful property and hence it is interesting to ask: given a schema which is not a product order embedding, we can always construct an equivalent schema which is an embedding? The answer is yes, almost. Since any map is determined by just the schema of the maximal members of the evaluation subset, we can always construct a schema that is a *piecewise* embedding of the down-set of each maximal member. We demonstrate the construction in Figure 15 using the schema from Figure 10.

Let $\underline{E}$, ε, and $S|$ denote the input evaluation subset, evaluation-schema map, and map schema, respectively. Let $\underline{E}^+$, ε^+, and $S|^+$ denote the output evaluation subset, evaluation-schema map and schema, respectively. Let $F|$ and $f|^t$ denote the range schema poset and its top member, respectively, which are the same for both input and output.

We start by defining the output evaluation subset. Let $\max(\underline{E})$ denote the set of maximal members of $\underline{E}$. We define $\underline{E}^+$ to be the sum of the down-sets of the maximal members of the input:

$\underline{E}^+ = \Sigma_{e \text{ in } \max(\underline{E})} \; e{\downarrow}$

Note that $\max(\underline{E}^+) = \max(\underline{E})$ and for each $e \in \underline{E}^+$ there is a unique member $\max(e) \in \max(\underline{E}^+)$ such that $e \leq \max(e)$.

The second step is to create a piecewise embedding ε^+:

$\varepsilon^+: \underline{E}^+ \times F| \rightarrow S|^+$

with

$\varepsilon^+(e, f|) \leq \varepsilon^+(e', f|')$ if and only if $e \leq e'$ and $f| \leq f|'$

and with

$\varepsilon^+(e, f|).\text{evaluation} = \pi_{f|} \circ \varepsilon(\max(e))$ for all $e \in \underline{E}^+$ and $f| \in F|$

In words, the evaluation method for $\varepsilon^+(e, f)$ is the composition of the input evaluation method for max(e) with projection onto f|. The embedding creates a disjoint piece $S|^+_e$ of schema for each $e \in \max(\underline{E}^+)$, specifically:

$S|^+_e = \varepsilon^+(e, f|^t)\downarrow$

Figure 15(a) and Figure 15(b) show $S|^+_{\{seg\}}$ and $S|^+_{\{v0, v1\}}$, respectively.

The third step is to extract the attribute subsets from the input schema. Define the total attribute subset A

$A = S| - \varepsilon(\underline{E})$

The total attribute set for the schema of Figure 10 is shown in Figure 15(c).

For each e in $\max(\underline{E})$, define the member attribute subset A_e:

$A_e = \varepsilon(e)\downarrow \cap A$

The member attribute subsets for the schema of Figure 10 are shown in Figure 15(d) and (e).

Finally, we can define the output schema. We define the base set for $S|^+$ to be the sum of the total attribute set A and the piecewise embeddings $S|^+_e$:

$S|^+ = A + \Sigma_{e \text{ in } \max(\underline{E})} S|^+_e$

We define the order relation to be: $s \leq s'$ if $s \leq s'$ in $S|^+_e$ or A or s in A_e and s' in $S|^+_e$. The output schema is shown in Figure 15(f).

The piecewise embedding construction does not in general produce a global embedding of the input $\underline{E} \times F|$, nor does it necessarily produce a schema that it optimal is some sense. For instance, the schema in Figure 12 specifies that the restriction to {v0} requires only a subset of the attributes of {v0, v1}, while in the schema in Figure 15 the restriction to {v0} requires the full set of attributes of {v0, v1}. Nevertheless, the existence of the procedure means we can require schema to be in at least piece-wise embedded form

without limiting the types of maps we can support. This in turn means we need to support only a single form of ε^+, as opposed to introducing some data-driven mechanism for representing a wider class of evaluation-schema maps.

11.2 Global section schema lattice

However the global section schema poset is constructed, the schema lattice is defined as the finite distributive lattice generated by the schema poset:

$\mathcal{S}| = \mathcal{O}(S|)$

If $S|$ is a product order isomorphism, $S| \sim \underline{E} \times F|$, then $\mathcal{S}| \sim \mathcal{O}(\underline{E} \times F|)$.

The tensor product $\mathcal{X} \otimes \mathcal{Y}$ of finite distributive lattices $\mathcal{X}$ and $\mathcal{Y}$ (4), is defined by

$\mathcal{X} \otimes \mathcal{Y} = \mathcal{O}(J(\mathcal{X}) \times J(\mathcal{Y}))$.

If $\mathcal{F}| = \mathcal{O}(F|)$ is the range schema lattice and we define $\underline{\mathcal{E}} = \mathcal{O}(\underline{E})$, then the schema lattice for a product schema poset is the tensor product:

$\mathcal{S}| \sim \mathcal{O}(\underline{E} \times F|) = \underline{\mathcal{E}} \otimes \mathcal{F}|$

In particular, the schema lattice for the map as relation schema is a tensor product:

$\mathcal{S}| \sim \mathcal{P}(\underline{B}) \otimes \mathcal{F}|$

The above definition of tensor product states the product is a down set lattice with the jims being pairs of jims of the factors. The join operation in a down set lattice is given by union of down sets. Since the down sets are sets of pairs, it can be shown the join satisfies the conditions:

$(x_1, y_1) \sim (x_1, y_2) = (x_1, y_1 \sim y_2)$

$(x_1, y_1) \sim (x_2, y1) = (x_1 \sim x_2, y_1)$

The members of the tensor product lattice are thus either pairs ($x \in \mathcal{X}$, $y \in \mathcal{Y}$) or are joins of pairs with no components in common, that is, they have the form:

$(x_1, y_1) \sim (x_2, y_2) \sim ... \sim (x_n, y_n)$ with $x_i \neq x_j$ and $y_i \neq y_j$ for $i \neq j$.

This means the schema can describe a piecewise map consisting of any combination of pairs (evaluation part, range schema part). This supports the specification of piecewise heterogeneous map representations, where the evaluation varies with position. It also supports restriction to any combination of domain and range parts.

11.3 Internal and external domains

We've shown above that to represent a map with a domain $\underline{B}$ requires an evaluation poset $\underline{E} \subseteq \mathcal{P}(\underline{B})$. We can think $\underline{E}$ of as the row poset of a type E, so every map type requires an auxiliary type as its evaluation poset. But notice that the map schema refers to the original domain type $\underline{B}$ only in the signature of the evaluation methods of the local map objects. In other words, there is no need for the domain itself to be represented as a type (i.e. a table) within the system. We will refer to a domain that is represented in the system as an internal domain, otherwise we will refer to it as an external domain. If a domain B is external, then so is the representation of $\mathcal{P}$(B) and the evaluation subsets, $\underline{E}_b$. A map type is internal or external if its domain is internal or external, respectively.

Note that an external domain need not be finite. The only requirement is that there be some way of uniquely identifying each member.

Turning this argument around, given a type B, we can either consider it the domain for an internal map type or the evaluation poset for an external map type.

The most important application for external domains is the representation of spatial objects, where the external domain is the (infinite) set of points in the object and the evaluation poset is the set of cells in a spatial decomposition (a mesh), ordered by point set inclusion. The points within each cell are referred to by local coordinates. Maps defined on such a domain associate an instance of the range type with each point in each cell of the domain.

Figure 16 shows an example based on the domain and range of Figure 7, but the domain is now considered external. The segment and vertex objects, which were members of the domain in the previous examples are now interpreted as cells and members of the evaluation poset $\underline{E}$, shown in Figure 16(a). Figure 16(b) shows the range schema $F|$, which is the same as previous examples. Figure 16(c) shows $\underline{E} \times F|$ and Figure 16(d) shows an order embedding schema. These diagrams all look similar to the previous examples with internal domains.

What is different are the evaluation methods. The domain is the infinite point set of the segment and the map can be evaluated at any point along the segment. Since the domain is one-dimensional, we can refer to any point using a real-valued coordinate. For example, a suitable evaluation method for the segment is:

```
void evaluation(double& b, A_R2& result)
{
  result.x = (1-b)*r0*cos(θ0) + b*r1*cos(θ1));
  result.y = (1-b)*r0*sin(θ0) + b*r1*sin(θ1));
}
```

12 Conclusion

The sheaf data model provides a unified framework for describing traditional tabular (relational) data, object-oriented types with inheritance, and maps on both finite and infinite domains.

13 References

1. **Butler, D. M.** Sheaf Data Model: Context and Overview. *Limit Point Systems, Inc.* [Online] 2012. http://www.sheafsystem.com/images/Publications/sdm_context_and_overview.pdf.

2. **Priestley, B.A. Davey and H.A.** *Introduction to Lattices and Order.* Second Edition. Cambridge : Cambridge University Press, 2002. ISBN 0-521-78451-4.

3. **INCITS/ISO/IEC.** *Information technology — Programming - C++*. New York : American National Standards Institute, 2012. INCITS/ISO/IEC 14882-2011[2012].

4. *The tensor product of distributive lattices.* **Shmuely, Zahava.** 1979, Algebra Universalis, Vol. 9, pp. 281-296.

14 Appendices

14.1 Construction of Phase_space schema lattice

The finite distributive lattice generated from a poset P is isomorphic to the set $\mathcal{O}(P)$ of all down sets of P ordered by inclusion. As shown in (2 pp. 20-21), every member of $\mathcal{O}(P)$ is a union of the down sets of antichains of members of P. We can exhaustively enumerate $\mathcal{O}(P)$ as long as P is small enough.

$$rx\downarrow = \{rx\} \quad (A.1)$$
$$ry\downarrow = \{ry\} \quad (A.2)$$
$$vx\downarrow = \{vx\} \quad (A.3)$$
$$vy\downarrow = \{vy\} \quad (A.4)$$
$$rx\downarrow \cup ry\downarrow = \{rx, ry\} \quad (A.5)$$
$$rx\downarrow \cup vx\downarrow = \{rx, vx\} \quad (A.6)$$
$$rx\downarrow \cup vy\downarrow = \{rx, vy\} \quad (A.7)$$
$$ry\downarrow \cup vx\downarrow = \{ry, vx\} \quad (A.8)$$
$$ry\downarrow \cup vy\downarrow = \{ry, vy\} \quad (A.9)$$
$$vx\downarrow \cup vy\downarrow = \{vx, vy\} \quad (A.10)$$
$$rx\downarrow \cup ry\downarrow \cup vx\downarrow = \{rx, ry, vx\} \quad (A.11)$$
$$rx\downarrow \cup ry\downarrow \cup vy\downarrow = \{rx, ry, vy\} \quad (A.12)$$
$$rx\downarrow \cup vx\downarrow \cup vy\downarrow = \{rx, vx, vy\} \quad (A.13)$$
$$ry\downarrow \cup vx\downarrow \cup vy\downarrow = \{ry, vx, vy\} \quad (A.14)$$
$$rx\downarrow \cup ry\downarrow \cup vx\downarrow \cup vy\downarrow = \{rx, ry, vx, vy\} \quad (A.15)$$
$$r\downarrow = \{r, rx, ry\} \quad (A.16)$$
$$r\downarrow \cup vx\downarrow = \{r, rx, ry, vx\} \quad (A.17)$$
$$r\downarrow \cup vy\downarrow = \{r, rx, ry, vy\} \quad (A.18)$$
$$r\downarrow \cup vx\downarrow \cup vy\downarrow = \{r, rx, ry, vx, vy\} \quad (A.19)$$
$$v\downarrow = \{v, vx, vy\} \quad (A.20)$$
$$v\downarrow \cup rx\downarrow = \{v, vx, vy, rx\} \quad (A.21)$$
$$v\downarrow \cup ry\downarrow = \{v, vx, vy, ry\} \quad (A.22)$$
$$v\downarrow \cup rx\downarrow \cup ry\downarrow = \{v, vx, vy, rx, ry\} \quad (A.23)$$
$$r\downarrow \cup v\downarrow = \{r, rx, ry, v, vx, vy\} \quad (A.24)$$
$$Phase\_space\downarrow = \{Phase\_space, r, rx, ry, v, vx, vy\} \quad (A.25)$$

The Hasse diagram for the above lattice is shown in Figure 2(b).

15 Figures

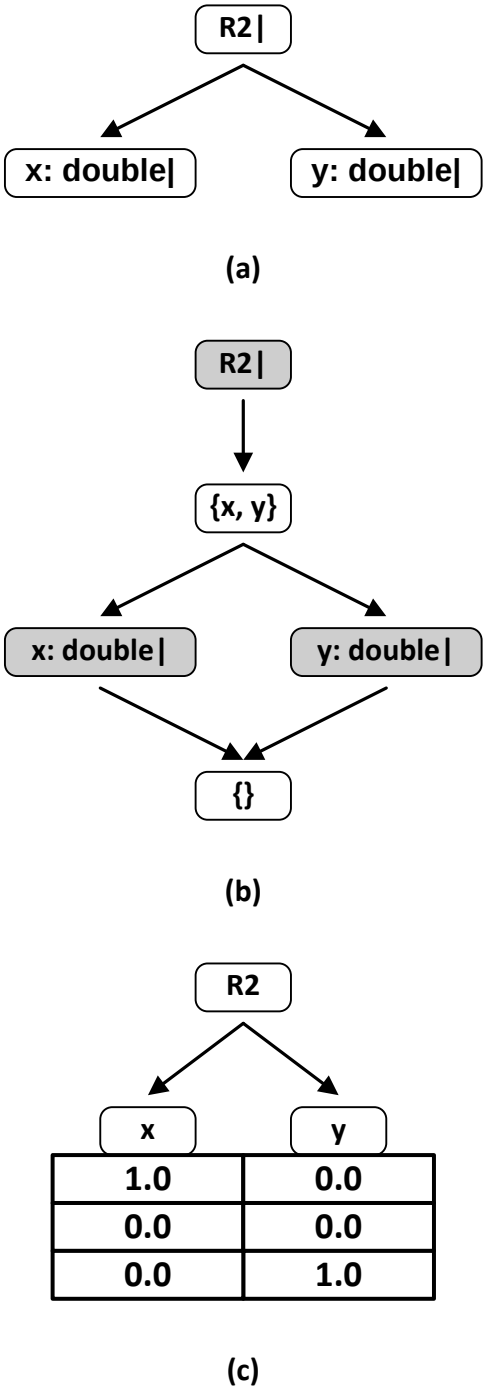
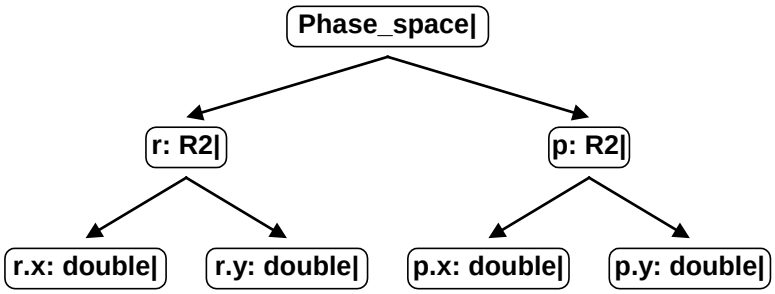
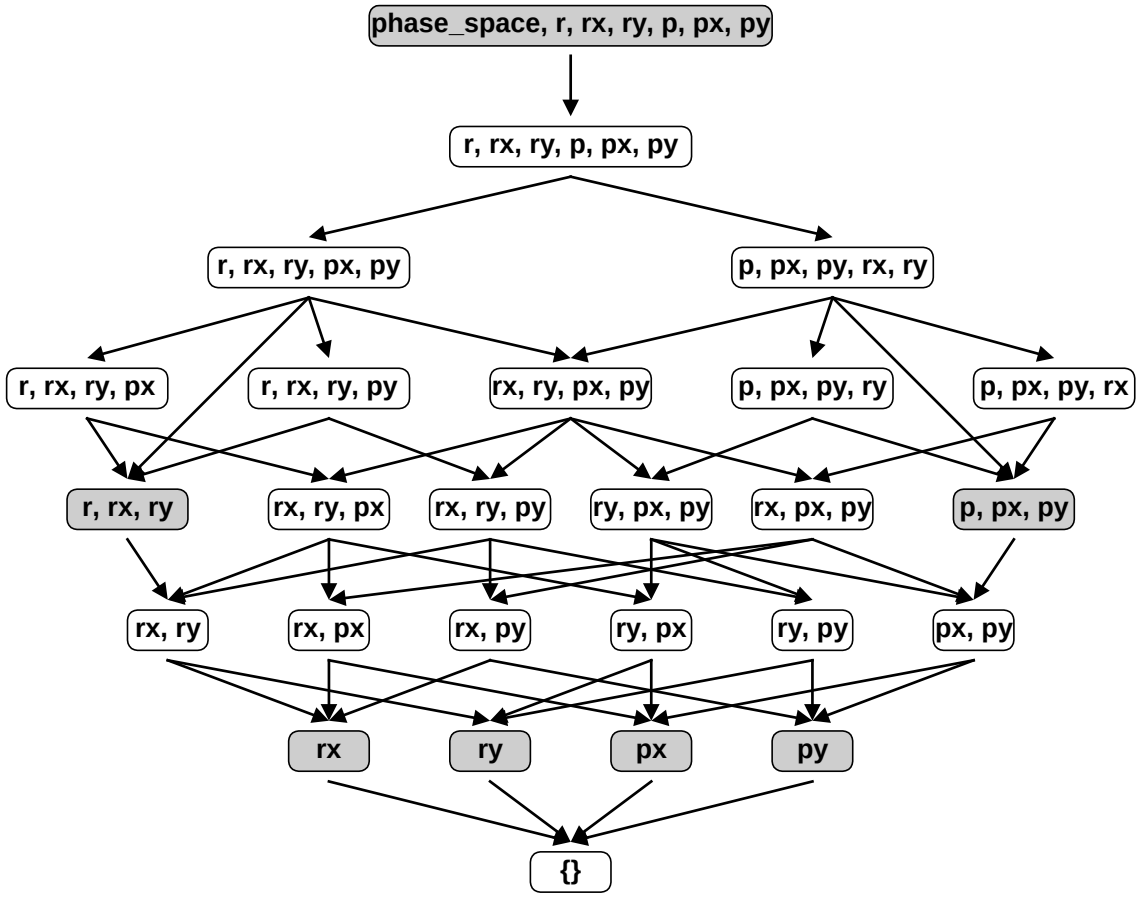


Figure 1: Class R2 (a) schema poset , (b) schema lattice, with join-irreducible members shaded, and (c) table metaphor for lattice schema.



(a)



(b)

Figure 2: Class Phase_space (a) schema poset and (b) schema lattice, with join-irreducible members shaded.

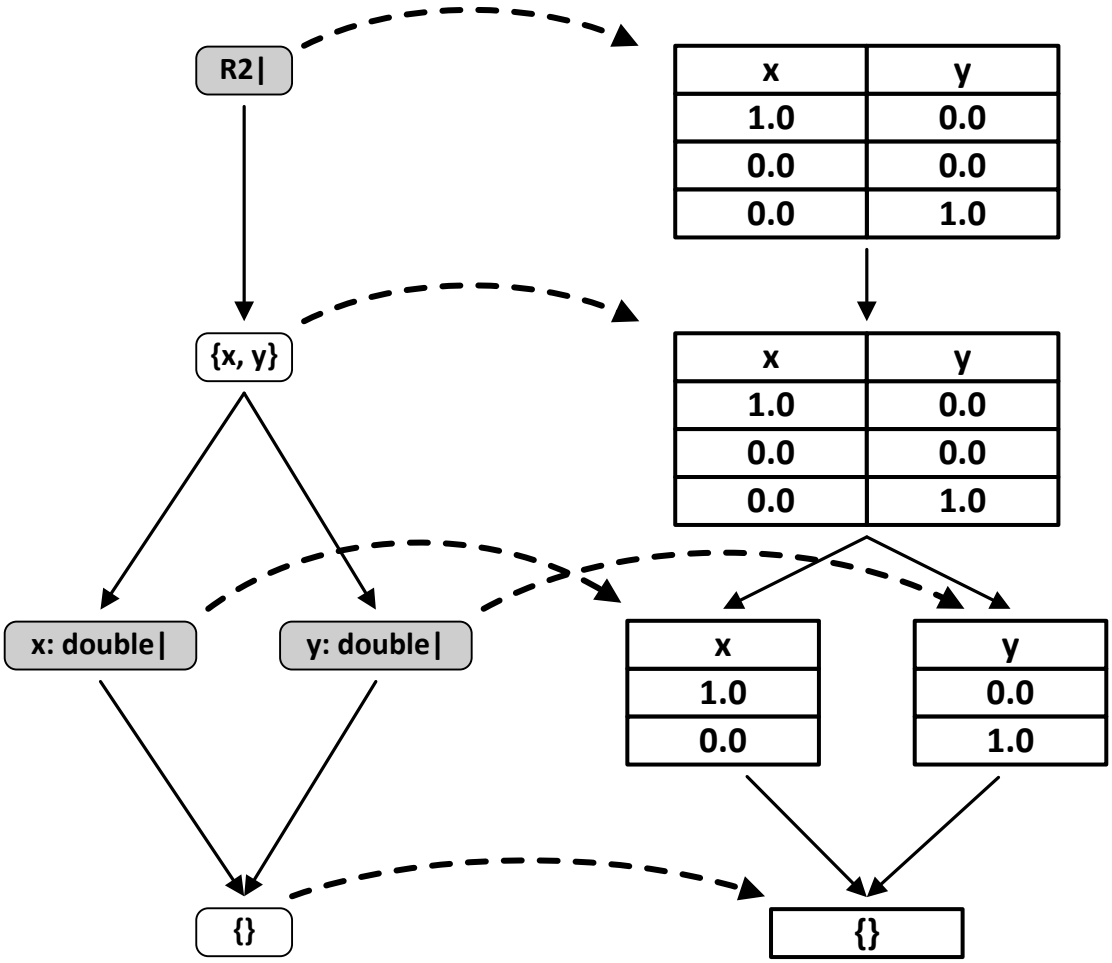


Figure 3: Sheaf of relations for the table of Figure 1(c).

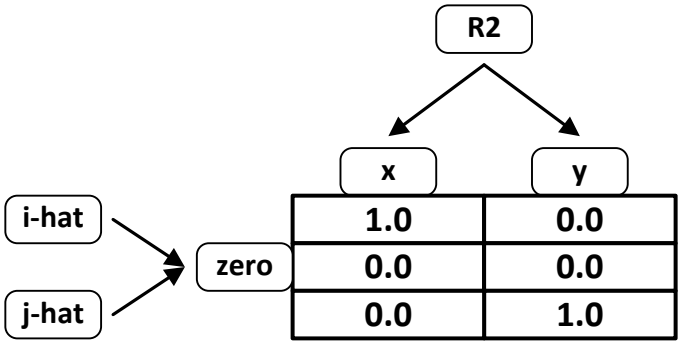


Figure 4: Table metaphor for lattice ordered relation.

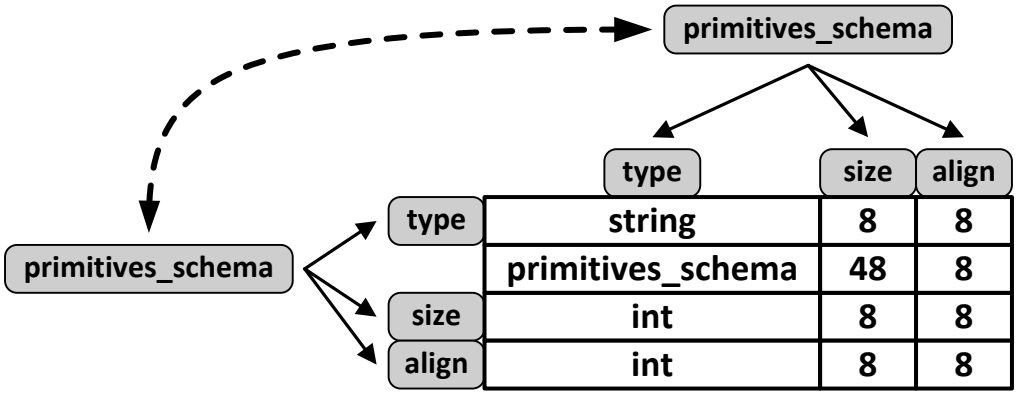


Figure 5: A simple primitives schema table.

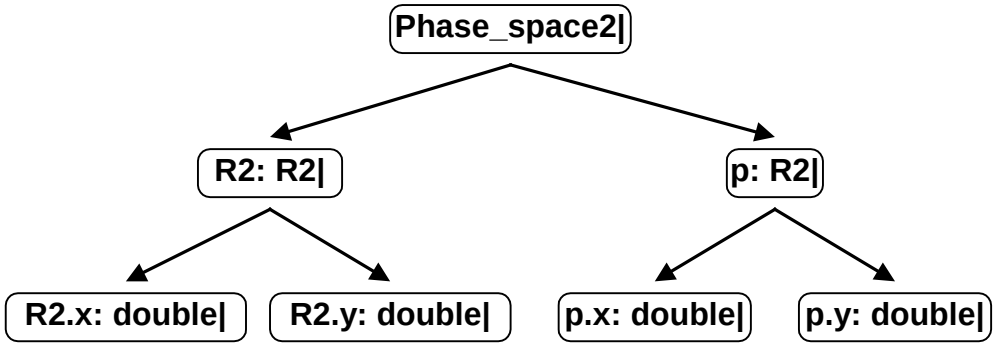


Figure 6: Phase_space schema poset with inheritance

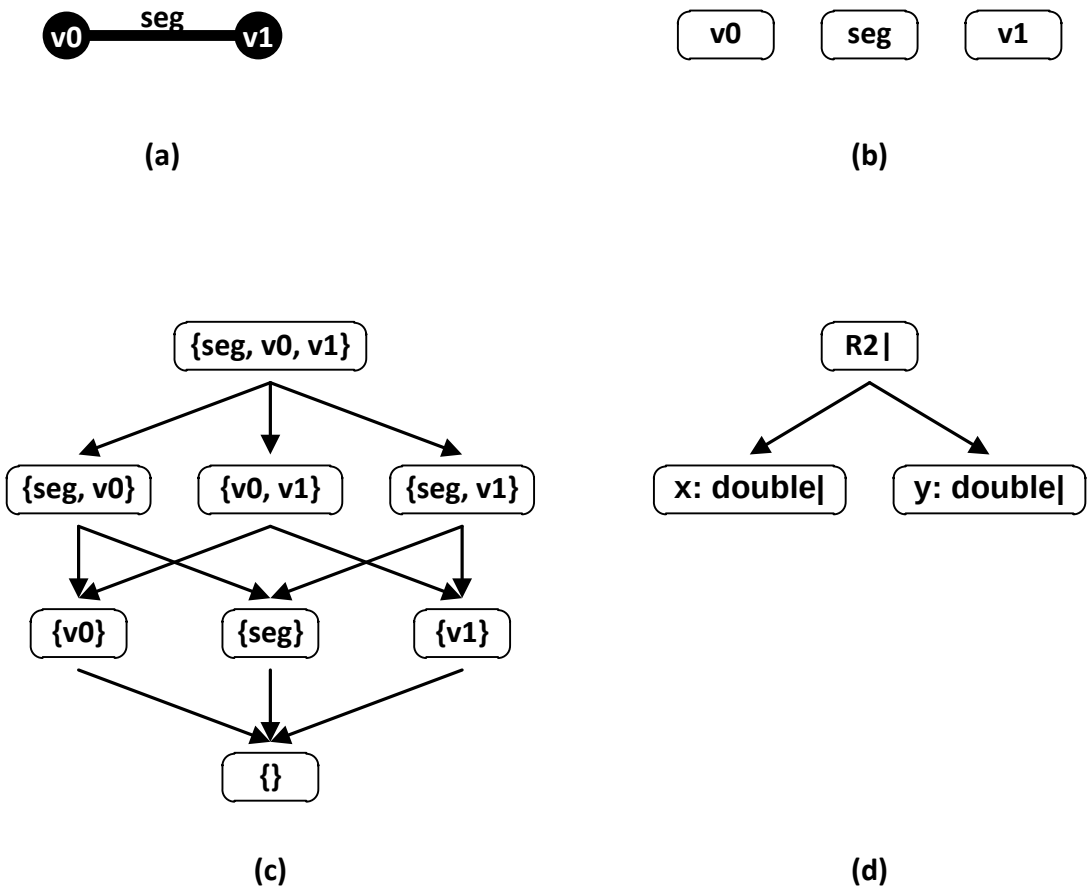
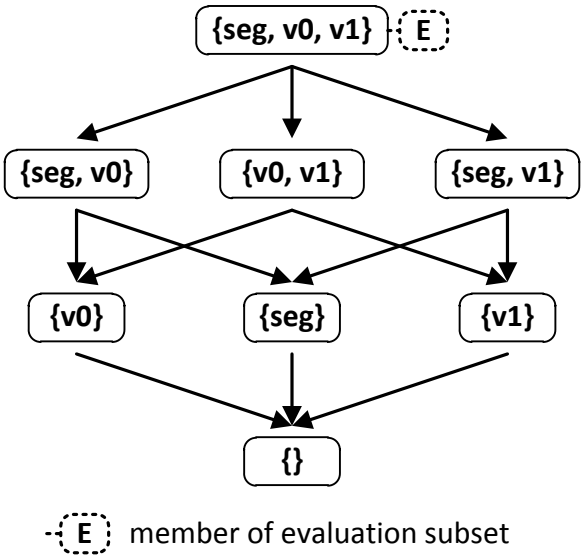
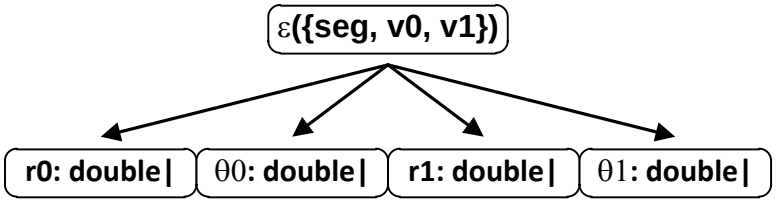


Figure 7: Domain and range for map schema examples: (a) domain geometry, (b) domain poset, (c) domain powerset, (d) range schema poset.

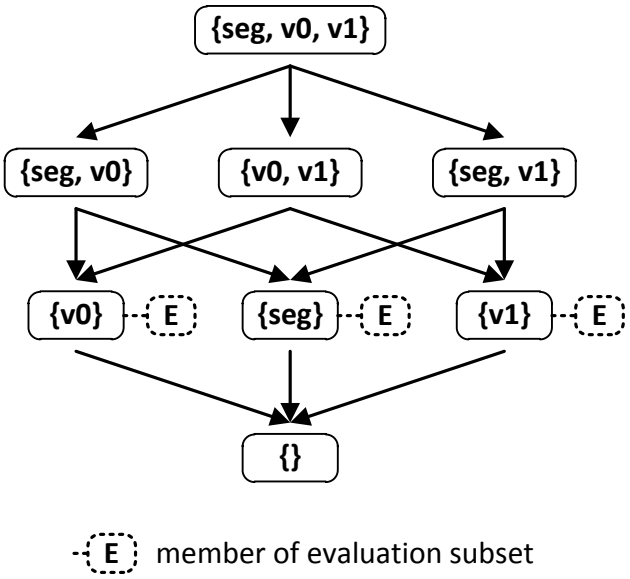


(a)

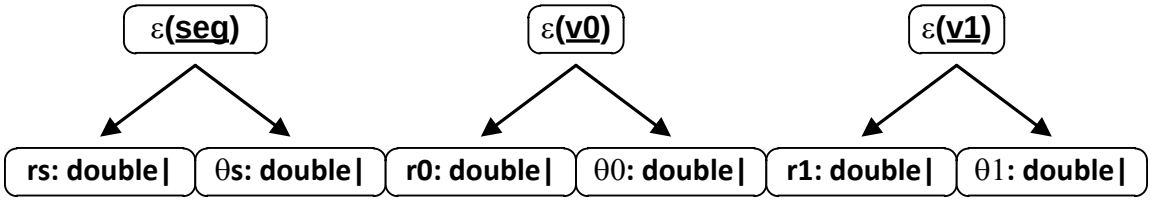


(b)

Figure 8: Coarsest schema: (a) evaluation subset, (b) a global map schema

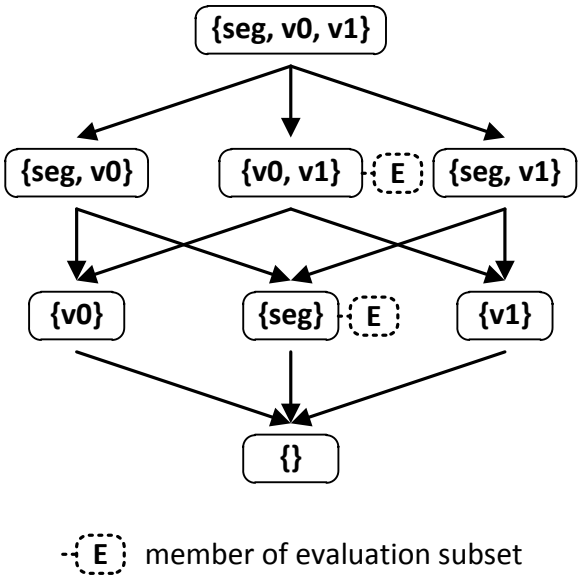


(a)

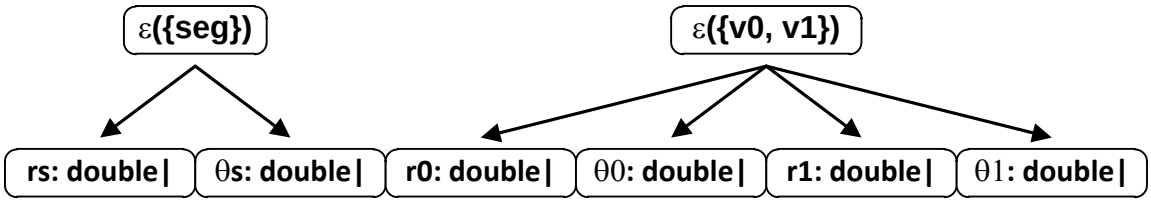


(b)

Figure 9: Finest schema: (a) evaluation subset, (b) a global map schema.

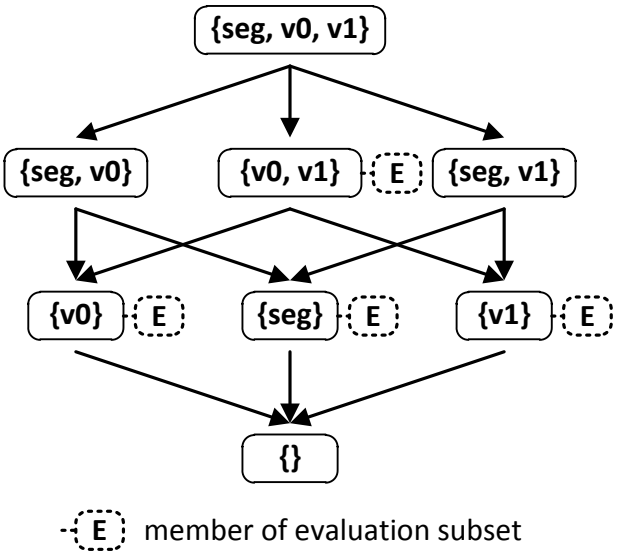


(a)

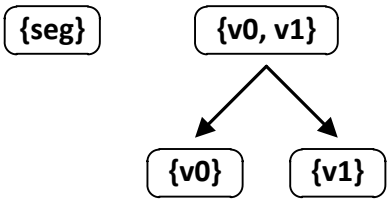


(b)

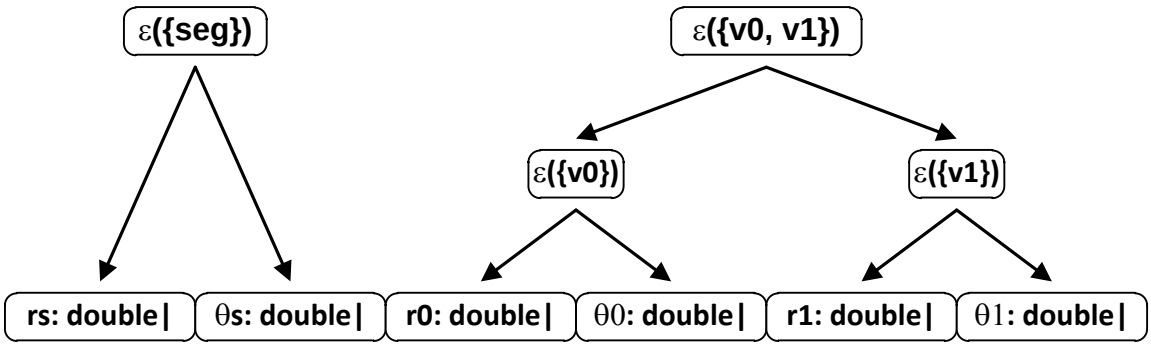
Figure 10: Intermediate schema: (a) evaluation subset, (b) a global map schema.



(a)



(b)



(c)

Figure 11: Domain order embedding for example of Figure 10: (a) $\underline{\underline{E}}$ extended to a down-set in $\mathcal{P}(\underline{B})$, (b) $\underline{\underline{E}}$ as a poset, (c) a global map schema.

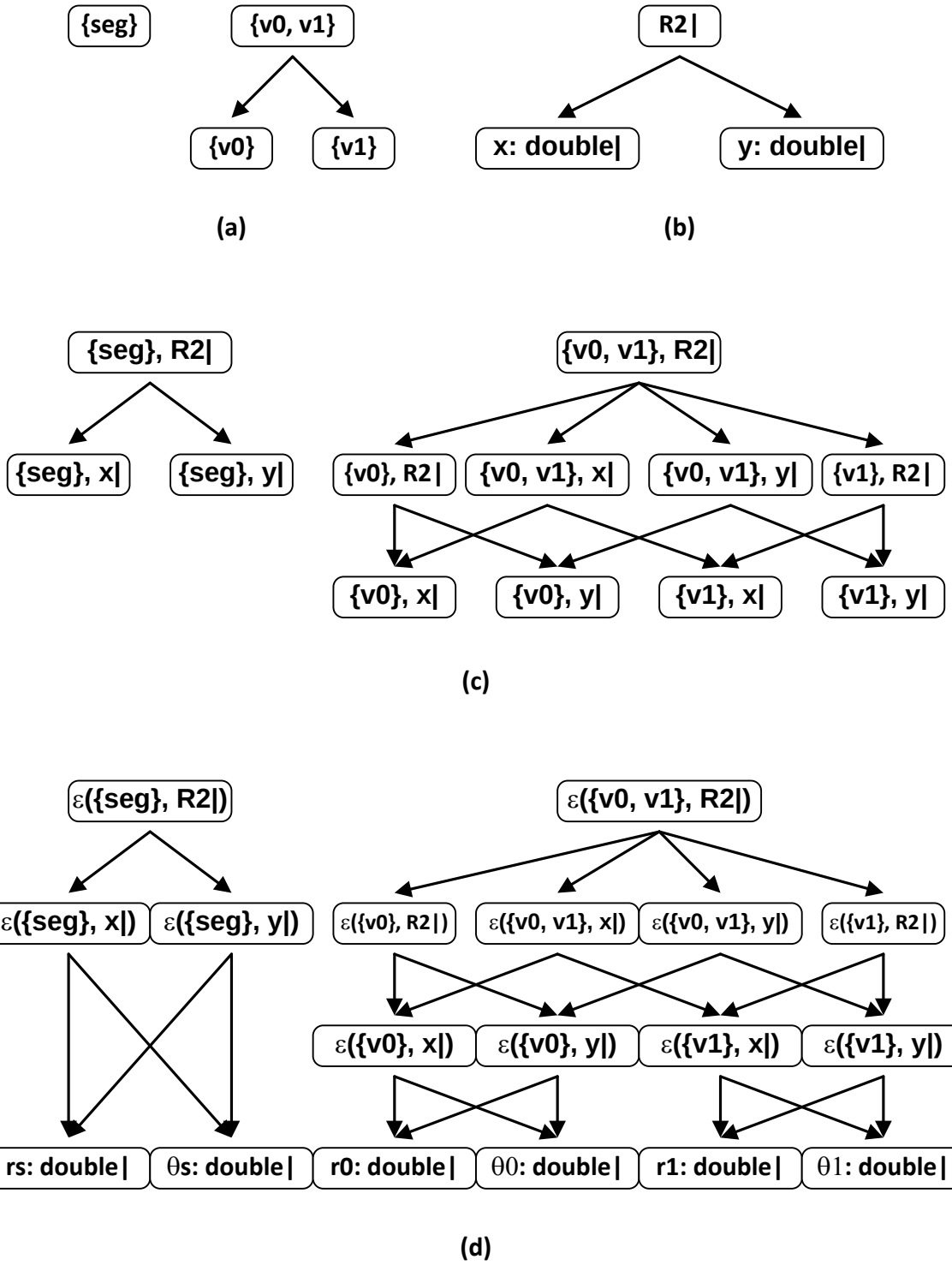


Figure 12: Product order embedding for example of Figure 10: (a) evaluation subset $\underline{E}$, (b) range schema $F|$, (c) Cartesian product poset $\underline{E} \times F|$, (d) global map schema.

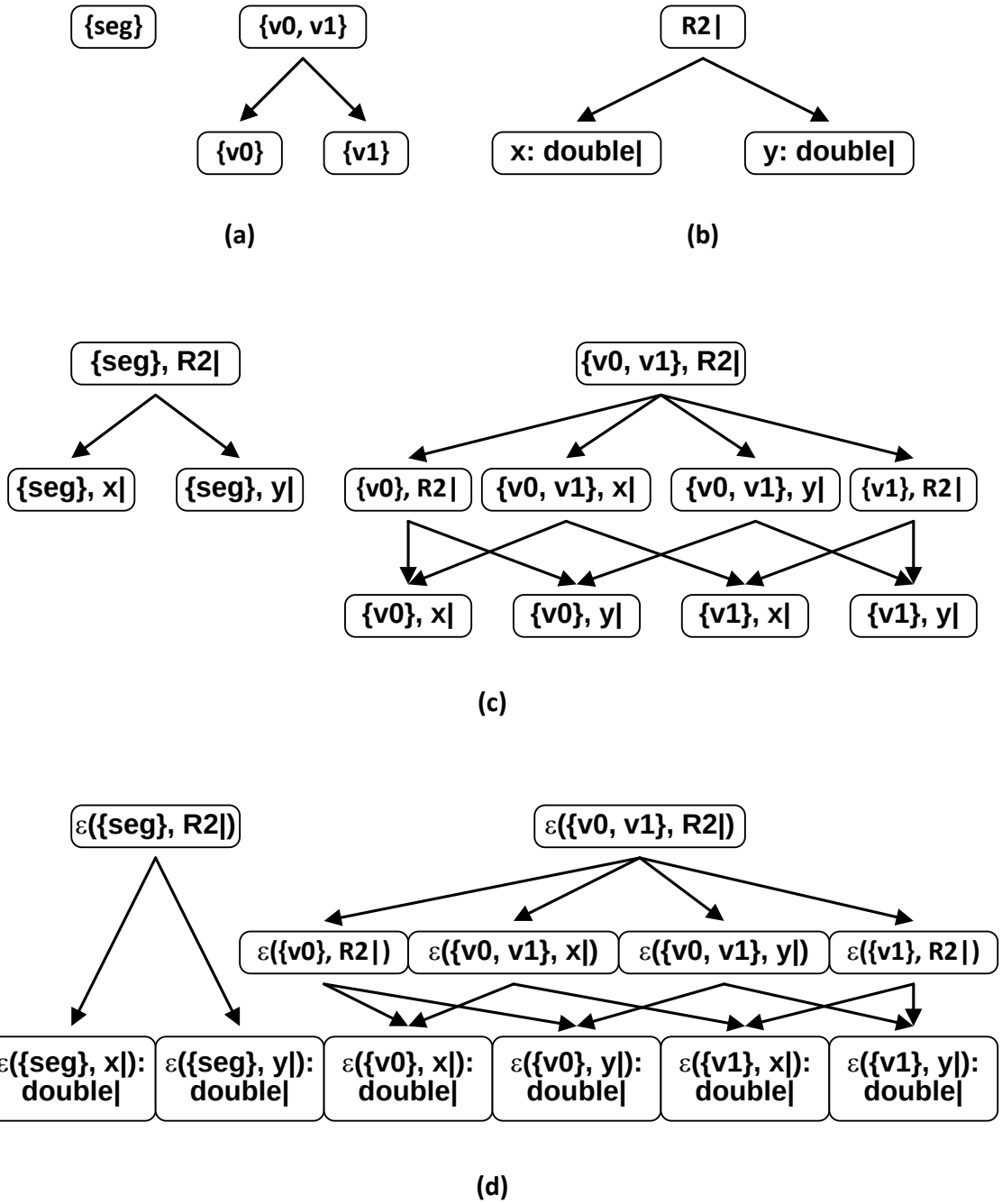
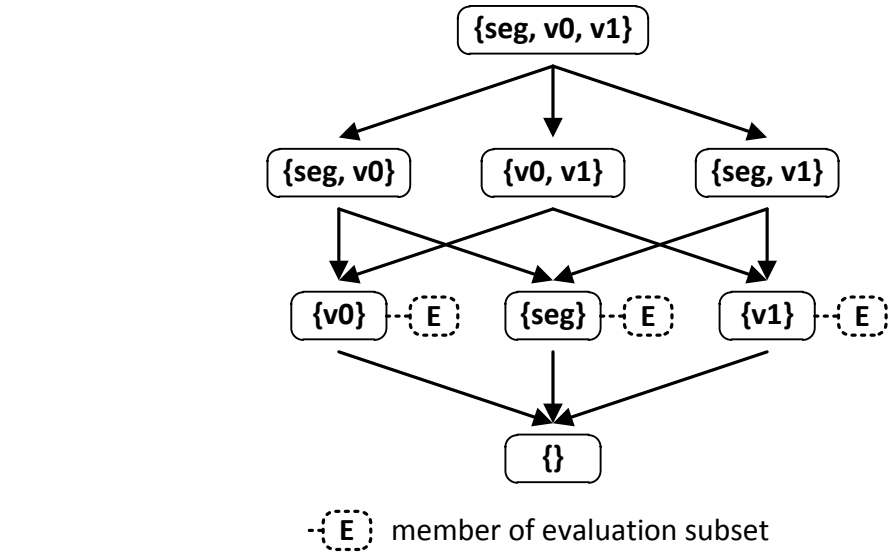
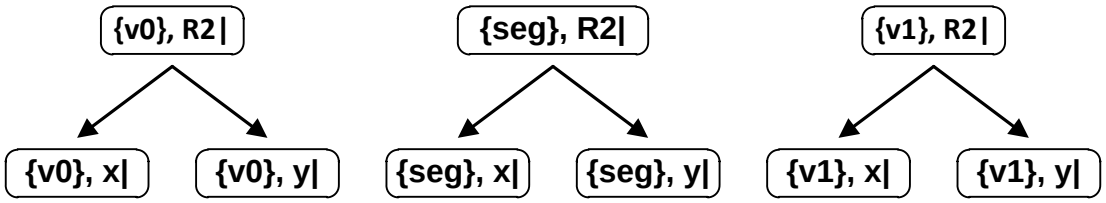


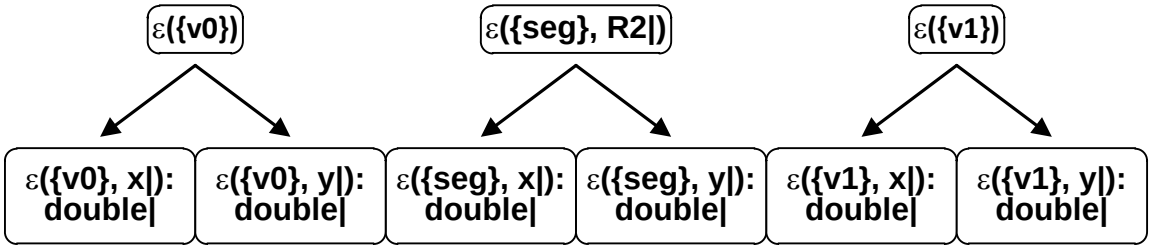
Figure 13: Product order isomorphism for example of Figure 10: (a) evaluation subset $\underline{E}$, (b) range schema $F|$, (c) Cartesian product poset $\underline{E} \times F|$, (d) global map schema



(a)



(b)



(c)

Figure 14: Map as relation: (a) evaluation subset, (b) Cartesian product poset $\underline{E} \times F|$, (c) global map schema

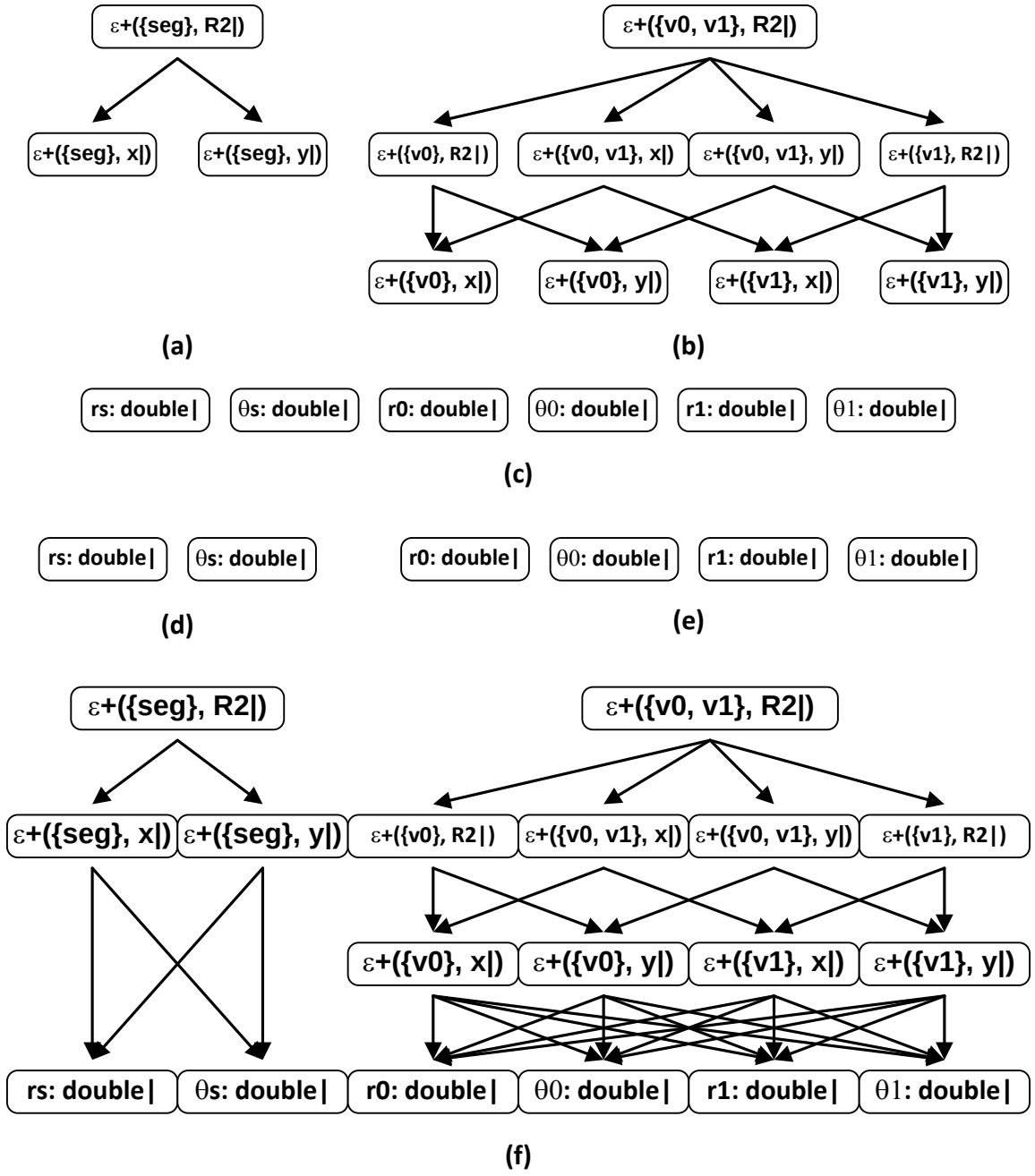


Figure 15: Piecewise embedding procedure for example in Figure 10: (a) $S|^{+}_{\{seg\}}$, (b) $S|^{+}_{\{v0, v1\}}$, (c) total attribute subset, (d) attribute subset for ε({seg}), (e) attribute subset for ε({v0, v1}), (f) global map schema.

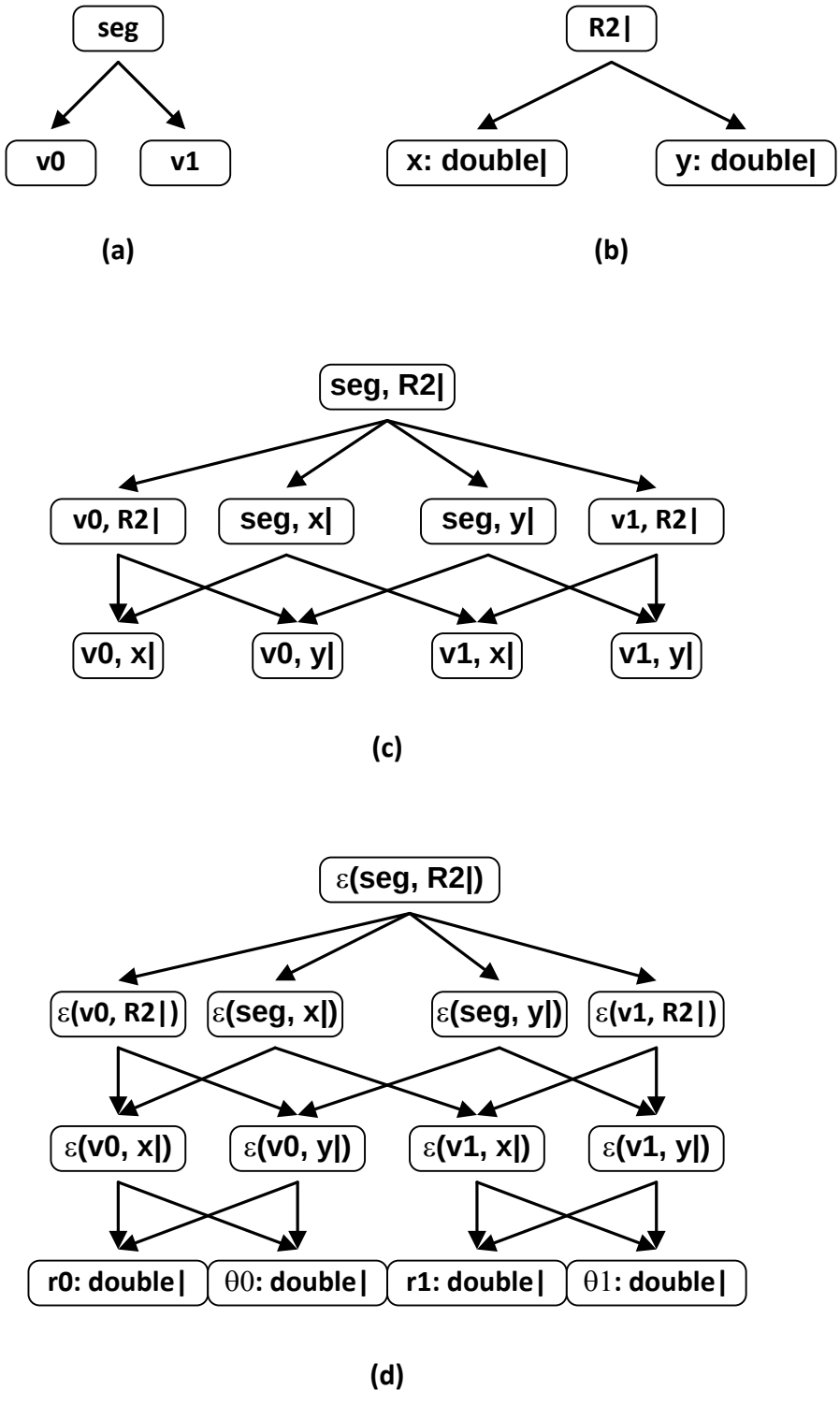


Figure 16: External domain product embedding: (a) evaluation poset $\underline{E}$, (b) range schema $F|$, (c) $\underline{E} \times F|$, (d) global map schema

16 Release history

Release 1.0: 10/1/2012. Initial release.