FINITE HIGHER-ORDERED BINARY RELATIONS

Robbert van Dalen
robbert.van.dalen@gmail.com

Abstract

The relational data model - first introduced by Codd in the 1970s - can be regarded as the most widespread data model to date. In this short paper we extend Codd's model by considering also higher-ordered relations: relations between relations. We will recursively built higher-ordered structures that ultimately leads to a concurrent relational data model - and more.
A concrete implementation based on annotated treaps is also discussed.

Definitions

We only consider finite ordered binary relations because n-ary relations can be easily represented with binary relations. We use the product `DOMAIN*RANGE` to denote the **range** and **domain** of a relation. We start out with two simple relations `R1` and `R2` and from there on recursively built higher and higher ordered relations.

In our first example, relation `R1` has **type** `Nat*Nat` and `R2` type `Char*Nat`. Because `Char` and `Nat` both have a total ordering, their pairs are also totally ordered.
A concrete instance of a relation - an ordered subset of pairs - is called a **version** of a typed relation. The following example shows two versions of `R1` and `R2`.

R1.1	
1	1
2	2
3	1

R1.2	
1	1
2	2
3	1
3	2

R2.1	
a	1
c	2
d	1

R2.2	
a	1
a	2
c	2
d	1

Names

Each version is given an **unique** name. We use these names to embed relations into other relations. This shortens the notation, but also hints towards data sharing: when the same version is always mapped to the same name (and vice versa) we can achieve optimal sharing and data compression.

A **cryptographic** hash of all sorted pairs will serve that purpose. Alternatively, **collision-free** names can be sourced from multiple incremental global counters.
However, cryptographic hashes have the advantage that they can be derived **locally** without any coordination, while collision-free counters require some kind of **global** coordination.

The relational database

A typed relational **database** `D` can be modeled as an ordered set of typed relations, called **tables**. A database version is determined by an unique set of its versioned tables. We can track tables if we give versioned tables user-defined names. This introduces a second-order binary relation:
`Database = Name*((Nat*Nat)|(Char*Nat))`

The following example shows three versions of such database:

D.1	
R1	R1.1
R2	R2.1

D.2	
R1	R1.2
R2	R2.1

D.3	
R1	R1.1
R2	R2.2

Database evolution

A typed **branch** B is an ordered set of databases that versions a single stream of **evolutions**. We will track evolutions with an incremental counter, introducing a third-order binary relation: `Branch = Nat*Database`

The following example shows three possible branches:

B.1	
1	D.1

B.2	
1	D.1
2	D.2

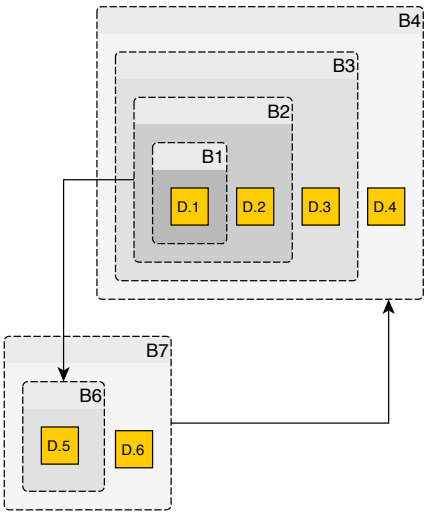
B.3	
1	D.1
2	D.2
3	D.4

Concurrently branching universe

The creation and merging of branches can be modeled with a Direct Acyclic Graph (DAG) of typed branches, which we call the database **universe**. This introduces a fourth-order binary relation `Universe = Branch*Branch`

It is an open question if cyclic universe graphs have strictly more expressive power. We believe cyclic graphs can be allowed if they are interpreted as two-way dependencies.

U.1	
B.2	B.6
B.7	B.4



The previous example shows the 'simple' case of an universe `U.1` without any cycles. The above visualization shows the creation, merging and evolution of branches that led to universe `U.1`.

Untyped relations

Nothing stops us from mixing second-order relations (databases) with fourth-order relations (universes). But we have to be careful that such relations can still be typed. So typed relations somewhat **restrict** our expressive power. But when we allow relations to be untyped, we are free to mix any relations we want.

Memoization of relational operations

We only discuss the union operation here as we deem it the canonical relational operation. Of course we can imagine more advanced operations, for instance an operation that can automatically remove conflicts between versions.

The union of two relations holds all pairs of both its arguments. We can model a union operation with an untyped higher-order memoization relation `MUNION`. The domain of this relation is the pair of (argument) relations and a single relation as its range (result). The following example shows 2 versions of `MUNION=(R,R)*R`

MUNION.1	
R1.1,R2.1	R3.1
R3.1,R4.1	R5.1

MUNION.2	
R1.1,R2.1	R3.1
R3.1,R4.1	R5.1
R4.1,R6.1	R7.1

Implementation

To make higher-ordered finite binary relations practically feasible, any implementation must at least meet two important requirements:

1. Confluently persistent
2. Uniquely represented

We need unique representations so that we can map them to unique names and vice versa. But foremost, it gives us `O(1)` equality checks if we memoize the constructor operations (also known as **hash consing**).

We also want confluence because we want an optimally efficient union operation, especially when dealing with two nearly identical versions.

This combination of requirements rules out many possible functional data structures, including red-black trees, finger trees and more. Fortunately, there is one particular data structure that does meet all requirements. Strangely enough, its power comes from randomization.

Treaps

A treap (Aragon and Seidel, 1989) is a kind of two-dimensional binary tree, also known as a cartesian tree (Vuillemin, 1980). Each tree node holds a two-dimensional pair: `(priority,value)`

A treap is balanced with high probability when its priorities and values are maximally uncorrelated. Because a treap is balanced, the insertion and deletion of nodes can be done in `O(log n)` running time.

m>n	insert	delete	union*
standard	O(log n)	O(log n)	O(m*log n)
hash-consing	O(log n)	O(log n)	O(log m)

(Blelloch and Reid-Miller, 1998) have shown that treaps are confluently persistent and that set operations on them can be made efficient. When treaps are also hash-consed, the running time of the union* operation can be further improved, especially if two treaps differ only by one element*.

To see why the union operation can be improved, let us consider two treaps that have exactly the same content. Because of hash-consing, they must have the same name

(or memory pointer) so the union operation only has to take O(1) due to a fast equality check.

(Golovin, 2009) has shown that - when values are consistently hashed to create their associated priorities - treaps collapse into uniquely represented trees.
(Golovin, 2009) also shows that B-treaps can be stored and retrieved from memory in blocks for optimal IO.

Treaps are typically used as sorted set containers but they can be easily extended to represent multi-sets or even multi-maps (and thus binary relations).

We represent a binary relation by pairing each domain value with a set of range values at each treap node.

```
(domain value, sorted-set(range value))
```

But in doing so, we only order the domain values and **must** ignore the range values. Also, only the domain values are used for consistent priority hashing. That way, treaps that share exactly the same domain will have exactly the same unique tree shape.

Annotating Treaps

Because treaps can recursively hold other treaps as both domain and range values, we need to be able to quickly determine the hash of a treap (used for the priority in the parent treap) without excessive recalculation. Next to its hash, we may decide to annotate a treap with other annotations such as size or its depth.

To achieve quick hashing, we lazily and recursively annotate each treap with its hash.
We do this by combining the hashes of the left- and right nodes, together with the hash of the contents of the treap node, i.e. the combined hash of the domain and range values. After that we store the hash in the treap node itself.

Using this annotation scheme, treaps become a kind of **incremental** Merkle trees (Merkle, 1979) with the insertion of a node only taking `log(n)` re-hashes. By design, we also get **incrementally** unique names when we annotate treaps with cryptographic hashes.

To differentiate this special kind of treap from ordinary treaps we feel we should give it a special name:

The **spread** data structure.

Related work

The hypernode model (Levene and Poulovassilis, 1990) appears to have the same recursive modeling power as higher-order relations. As an operational model however, it lacks confluence, unique representations and a consistent naming scheme. Also, the relational model is much more established and has a well defined relational algebra.

Discussion

Finite higher-ordered binary relations - or spreads - are very powerful objects. So powerful in fact that the author believes it will revolutionize the way we deal with information - and much more. This paper has only shown some of its merits.

Acknowledgments

Nicolas Oury for telling me that two-way multi-maps are 'just' binary relations!

References

R. Seidel and C. Aragon, 1996. *Randomized Search Trees*

J. Vuillemin, 1980. *A unifying look at data structures*

E. Blelloch and M. Reid-Miller, 1998. *Fast Set Operations Using Treaps*

D. Golovin, 2009. PhD thesis: *Uniquely Represented Data Structures with Applications to Privacy*

R.C. Merkle, 1979. *A digital signature based on a conventional encryption function*