The C^3 Constraint Object-Oriented Database System

Alexander Brodsky Victor E. Segal Pavel A. Exarkhopoulo
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030
brodsky@isse.gmu.edu

August 1996

Abstract

Constraints provide a flexible and uniform way to conceptually represent diverse data capturing spatio-temporal behavior, complex modeling requirements, partial and incomplete information etc, and have been used in a wide variety of application domains. Constraint databases have recently emerged to deeply integrate data captured by constraints in databases. This paper reports on the development of the *first constraint object-oriented database system*, C^3, and describes its specification, design and implementation. The C^3 system is designed to be used for both implementation and optimization of high-level constraint object-oriented query languages such as $\mathcal{L}yri\mathcal{C}$ or constraint extensions of OQL, and for directly building software systems requiring extensible use of constraint database features. The C^3 data manipulation language, *Constraint Comprehension Calculus*, is an integration *constraint calculus* for extensible constraint domains within *monoid comprehensions*, which serve as an optimization-level language for object-oriented queries. The data model for constraint calculus is based on constraint spatio-temporal (CST) objects that may hold spatial, temporal or constraint data, conceptually represented by constraints. New CST objects are constructed, manipulated and queried by means of constraint calculus. The model for monoid comprehensions, in turn, is based on the notion of monoids, which is a generalization of collection and aggregation types to structures over which one can iterate and apply merge operator; this includes disjunctions and conjunctions of constraints. The focal point of our work is achieving the right balance between expressiveness, complexity and representation usefulness, without which the practical use of the system would not be possible. To that end, C^3 constraint calculus guarantees polynomial time data complexity, and, furthermore, is tightly integrated with monoid comprehensions to allow deep global optimization.

1 Introduction

Constraints provide a flexible and uniform way to conceptually represent diverse data capturing spatio-temporal behavior, complex modeling requirements, partial and incomplete information etc, and have been used in a wide variety of application domains. Constraint databases have recently emerged to deeply integrate data captured by constraints in databases. Although a relatively new realm of research, constraint databases have drown much attention and increasing interest, more in aspects of expressibility and complexity, but also in algorithms and optimization.

While many fundamental research questions are yet to be answered, we believe that the area of constraint databases became mature for a reliable research prototype that could serve as a stable platform for experimentation with algorithms and optimization as well as for real-life case studies of a number of promising application domains. Building such a system is a necessary and important step toward proving the validity of constraint databases as a technology with significant practical impact.

The contribution of the work reported in this paper is the development, i.e. the specification, design and implementation, of the *first constraint object-oriented database system*, C^3. The C^3 data manipulation language, *Constraint Comprehension Calculus*[1], is an integration *constraint calculus* for extensible constraint

[1] which C^3 stands for

domains within *monoid comprehensions*, which were suggested as an optimization-level language for object-oriented queries [FM].

The data model for constraint calculus is adapted from *constraint spatio-temporal* (CST) objects [BK95], that may hold spatial, temporal or constraint data, conceptually represented by constraints (i.e. symbolic expressions). In the current version, linear arithmetic constraints (i.e. inequalities and equations) over reals [2] are implemented. New CST objects are constructed using logical connectors, existential quantifiers and variable renaming, within constraint calculus. Constraints module also provides predicates such as for testing satisfiability, implication etc, that are used as selecting conditions in the hosting monoid comprehension query.

In general, CST objects possess great modeling power and as such can serve as a *uniform data type* for conceptual representation of heterogeneous data, including spatial and temporal behavior, complex design requirements and partial and incomplete information. Moreover, constraint calculus operating on CST is highly *expressive* and *compact* language. For example, just linear arithmetic CSTs and its calculus currently implemented in the system, allow the description and powerful manipulation of a wide variety of data, including 2- or 3-D geographic maps; geometric modeling objects for CAD/CAM; fields of vision of sensors; 4-D (3 + 1 for time) trajectories of objects moving in 3-D space, based on the movements equations; translations of different system of coordinates; operations research type models such as manufacturing patterns describing interconnections among quantities of manufactured products and resource materials. It is important to note that while constraint objects are *conceptually* represented by constraints, their physical structure may very much differ for the purpose of efficient storage and manipulation.

The general framework of the C^3 language is, so-called, monoid comprehensions query language, in which CST objects serve as a special data type, and are implemented as a library of interrelated C++ classes. In turn, many operations of constraint algebra working on CST objects are expressed through nested monoid comprehensions, which allows potentially deep global optimization. The data model for monoid comprehensions is based on the notion of *monoid*, which is a conceptual data type capturing uniformly collections, aggregations, and other types over which one can "iterate", including (long) disjunctions and conjunctions of constraints. The C^3 system is designed to support:

1. Besides CST objects, any (complex) data structures expressible in C++.
2. Extensible family of parameterized and possibly nested collection monoids currently including sets, bags, lists, as well as (long) disjunction and conjunctions of CST objects.
3. Extensible family of aggregation monoids such as *sum*, *count*, *some* and *all*.
4. Extensible family of search structures implemented as parameterized monoids and currently including B-trees, hashing and kD-trees (for multidimensional) rectangles, serving as approximations of constraints.
5. Extensible family of special-purpose algorithms, such as constraint joins, implemented as parameterized monoids.
6. Approximation-based filtering, indexing and regrouping based on internal components of nested collection monoids.

Moreover, because C^3 system is implemented using the commercial OODB ObjectStore, it inherits features including persistence, transaction management, data integrity and crash recovery, version management, and multi-client/multi-server architecture.

The functionality of the C^3 system, depicted in Figure 1, is the combination of the new C^3 layer, and ObjectStore. Note that C^3 as a virtual system (1) inherits "lower" features of ObjectStore, (2) replaces "middle" ObjectStore's features with those of the C^3 layer, and (3) adds "upper" features of the C^3 layer. The implementation of the C^3 layer, in turn, uses ObjectStore and the linear programming package CPLEX. We feel important to note that while C^3 is a research prototype, it is a reliable system designed to carry out implementations of serious, massive data applications. That is due to the use of commercially available components (i.e. ObjectStore and CPLEX).

[2] using finite precision arithmetic

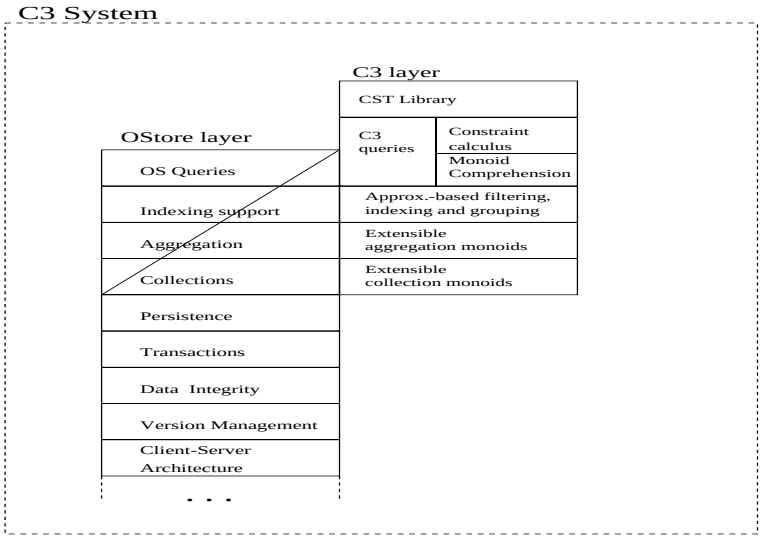

Figure 1: C^3 Functional Components

C^3, similar to ObjectStore, can be better viewed as a powerful extension of C++ with (constraint) database features, rather than a full-scale DBMS, and is currently to be used from within a hosting C++ program. As a C++ extension, C^3 uses the native C++ data structures and the type system. In fact, in the current implementation only monoid comprehensions are pre-compiled, and all the other C^3 features, including constraint calculus, are implemented as C++ libraries; hence the native C++ syntax is preserved.

The use of the "dirty" C++ data model, as opposed to "clean" and formally defined models such as of ODMG OQL [ABD+96] or XSQL [KKS92] was our pragmatic choice due to the intended purpose of C^3: intermediate optimization-level language, i.e. one in which an optimizer or a programmer can (manually) write deeply optimized queries, using appropriate order, nesting and built-in optimization primitives. Because of the intended use as an intermediate language, we prefer to regain the flexibility of and uniformity with the underlying programming language, C++. We designed C^3 to be used both for implementation and optimization of high-level constraint object-oriented query languages such as $\mathcal{L}yri\mathcal{C}$ or constraint extensions of OQL, and for directly building software systems (by application or system programmers) requiring extensible use of constraint database features.

The potential applications of C^3 include engineering design; manufacturing and warehouse support; command and control (such as spatio-temporal data fusion and sensor management [ABK95] and maneuver planning [BVCS93]); distribution logistics; and market analysis. The focal point of our work is achieving the right balance between expressiveness, complexity and representation usefulness [**?**] without which the practical use of the system would not be possible. To that end, C^3 constraint calculus guarantees polynomial data complexity, and, furthermore, tightly integrated with monoid comprehensions to allow deep global optimization.

The paper is organized as follows. Following the introduction, Section 2 informally discusses C^3 queries, including CST objects, constraint calculus and monoid comprehensions by examples. In Section 3, we review formal definitions of monoid comprehensions, explain implementation of monoids in C^3 and syntax and semantics of C^3 queries. Section 4 describes the design and implementation of CST families and their operations, on which constraint calculus is based. C^3 primitives for optimization using approximation-based filtering, regrouping and indexing are discussed in Section 5. Section 6 briefly surveys the related work. Finally, in Section 7 we conclude and mention topics of future work.

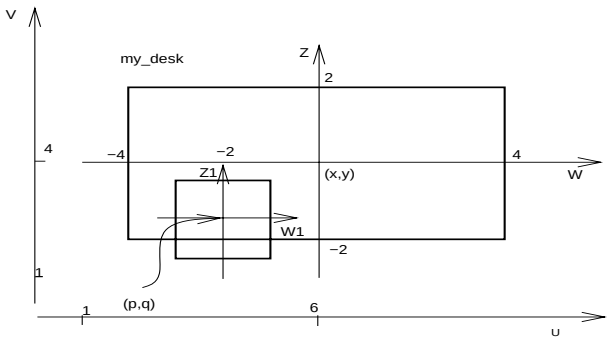

Figure 2: An instance of a desk with drawer in the room

2 CST Objects and C^3 Queries by Example

In this section we informally discuss C^3 queries, including CST objects, constraint calculus and monoid comprehensions using an architectural design example similar to the one in [BK95]. We assume, briefly, that the database stores a collection of office objects such as desks, file cabinets etc, which have extents (or shapes) and moving parts, such as drawers, as well as other attributes. A designer then may ask queries such as: Given a room and location of a number of objects in it, can we put an additional desk such that its drawer will not touch any other object in the room, and still have an unoccupied 4×4 feet space? Can we put in a room two desks, two file cabinets and two chairs such that (1) no two objects or their opened drawers will touch each other or the walls, and (2) there will be at least 4 feet between the front of each desk and the opposite wall? Can the system give constraints describing possible interconnections of centers of objects such that the above goals are achieved? What would be the location of the above mentioned objects if we want to maximize the size of a square of available empty space? Given a collection of objects in the room, show a projection of their cut at the height of $1/2$ feet. Those queries can be efficiently answered in C^3without using user implemented predicates or functions.

2.1 Constraints, CST objects and Schema by Example

Consider a two-dimensional desk ‘‘`my-desk`’’ depicted in Figure 2 as a larger rectangle. The smaller square is the desk's drawer, which may open, i.e. move relatively to the desk. Similarly, the desk may be moved in the room. There are three systems of coordinates used in Figure 2: U, V, the room's (or global) system; W, Z, the desk's coordinate system used to describe the desk's shape independently of the location of the desk in the room; and $W1, Z1$, the drawer's coordinate system. The pair of variables (x, y) describes the center of the desk in the room's coordinates; similarly, (p, q) describe the center of the drawer in the desk's coordinates.

The basic idea in constraint databases is the introduction of constraint formulae as a basic data type in databases. For example, a constraint (formula)

$$(-4 \leq w \leq 4) \wedge (-2 \leq z \leq 2)$$

with the variables ranging over reals can be viewed as a set of points

$$\{(w, z) | (-4 \leq w \leq 4) \wedge (-2 \leq z \leq 2)\}$$

in two-dimensional space and describes, say, the extent of `my-desk` (Figure 2) given in the desk's coordinates. More accurately, the constraint formula $(-4 \leq w \leq 4) \wedge (-1 \leq z \leq 2)$ will be interpreted as an infinite relation over the schema W, Z, that contains all tuples (w, z) satisfying the constraint.

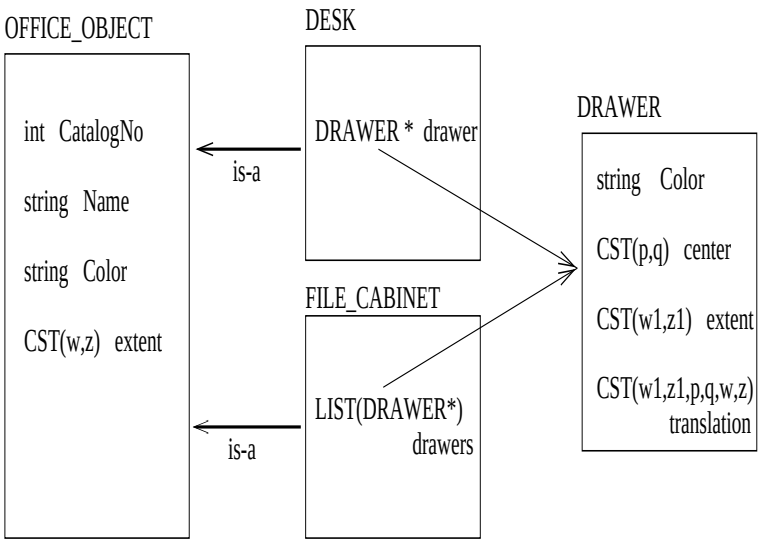

Figure 3: An Object-Oriented Database Schema

Similarly, the extent of the desk's drawer in the drawer's coordinates can be described by the constraint $(-1 \leq w1 \leq 1) \wedge (-1 \leq z1 \leq 1)$. The possible locations (p, q) of the drawer's center in the desk's coordinates can be described by $p = -2 \wedge -3 \leq q \leq -1$; note that the horizontal component of the center, p, equals to a constant since the drawer in the example cannot move left or right; note also that the vertical component, q, is between -3, when the drawer is fully open, and -1 when it is closed.

The translation between the desk's W, Z and the room's U, V systems of coordinates can be captured by the constraint $u = x + w \wedge v = y + z$, meaning that if the desk's center is at (x, y), then a point (w, z) in desk's coordinates is (u, v) in the global (room's) coordinates. Suppose that the desk's center has room's coordinates $(6, 4)$, i.e. $x = 6 \wedge y = 4$. Then, for example, we can find the extent of the desk in the room's coordinates by simplifying the following formula:

$$(\exists x, y, w, z)[[(-4 \leq w \leq 4) \wedge (-2 \leq z \leq 2)] \wedge [u = x + w \wedge v = y + z] \wedge [x = 6 \wedge y = 4]]$$

where the first []-component is the desk's extent in local coordinates, the second is the translation of coordinate systems and the third is the position of the desk's center. Finally, since we only interested in the free variables u, v, representing 2D-points in the room's coordinates, we existentially quantify all other variables. If we substitute the constants into x, y we get $u = 6 + w \wedge v = 4 + z$, and then, by using it in the first inequality we finally get $(2 \leq u \leq 10) \wedge (2 \leq v \leq 6)$. ¿From Figure 2 we can easily see that this is exactly the extent of `my-desk` described in the room's coordinate system.

In the C^3 syntax the above formula will look as follows:

```
(u,v) |   ((-4 <= w <= 4) && (-2 <= z <= 2) &&
          (u == x + w && v == y + z) &&
          (x == 6 && y == 4))
```

The `(u,v) | ...` notation is a projection, where we indicate all free variables in the result, rather rather than variables to be existentially quantified. We also use && and == instead of $\wedge$ and $=$, correspondingly, to preserve the C++ style. Interestingly, the above constraint syntax is native in C++, which is achieved by exploiting C++ operators' overloading mechanism. Users can intuitively think of a constraint with d free variables as a (possibly) infinite relation of d-tuples, as an object in d-dimensional space (i.e. set of points), or in or as a symbolic expression, interchangeably, depending on the application and the context of its use. Thus, we will be referring to a constraint by a generic name CST (i.e. *constraint spatio-temporal*) object.

Consider now an architectural design example schema depicted in Figure 3. We assume that the database keeps a bag collection (i.e. multiset) `all-office-objects`, i.e. it is declared as `Bag<OFFICE_OBJECT*>`, where "`*`" denotes reference (i.e. pointer in C++) to. Similarly, `all-desks` and `all-file-cabinets` are kept,

which are declared as `Bag<DESK*>` and `Bag<FILE_CABINET*>` correspondingly. Note that the same object may appear more than once in a bag collection.

Objects of class `OFFICE_OBJECT` have standard attributes such as `catalog#`, `name` and `color`, and also the `extent` attribute, describing the object's shape. The `extent` attribute is declared as `CST(w,z)`, to indicate that it must be represented as a constraint with free variables `w` and `z`. Note that, unless otherwise stated, we use the term "class" in this paper in the C++ sense. For instance, there is no class extent automatically associated with each class, as usually assumed in OO database models.

The classes `DESK` and `FILE_CABINET` are subclasses of `OFFICE_OBJECT`, and, therefore, inherit all its attributes. This is indicated by double arrows in Figure 2. In addition, the class `DESK` has the attribute `drawer`, which is declared as reference to an object of the class `DRAWER`. Similarly, an additional attribute of the class `FILE_CABINET` is `drawers` that declared as a `list_of` references to objects of class `DRAWER`. Note, that the single arrows in Figure 2 indicate the composition hierarchy.

Each drawer, in turn, is characterized by possible locations of its center, declared as `CST(p,q)*`; its `extent` declared as `CST(w1,z1)`; and `translation`, declared as `CST(w1,z1,p,q,w,z)`, that would hold an equation describing the interconnection between $(w1, z1)$, a point in the drawer's coordinates and (w, z), the same point in the desk's coordinates, provided the center of the drawer is at (p, q) in the desk's coordinates.

Below is an example of the C^3 (in fact, C++) definition `my_desk` of the class `DESK`:

```
DESK my_dsk = DESK(
  22354,                                  // catalog#
  ``one-drawer-desk'',                    // name
  ``red'',                                // color
  (-4 <= w <= 4 && -2 <= z <= 2),         // extent
  *new  DRAWER(                           // drawer
        ``blue'',                                 // color
        (p == -2 && -3 <= q <= -1),               // center
        (-1 <= w1 <= 1 && -1 <= z1 <= 1),         // extent
        (w == w1 + p && z == z1 + q)              // translation
  )
);
```

2.2 C^3 Queries by Example

Consider the following C^3 query, yet without CST objects, which finds a bag of all red `file_cabinets`, that have at least one blue drawer:

```
SELECT fc                                 // for file cabinet
INTO {Bag<FILE_CABINET*>} result          // result is a bag-collection
FROM all_file_cabinets
         AS {FILE_CABINET*} fc            // iterator: fc iterates over BAG
WHERE fc->color == ``red''                // predicate, i.e. condition
FROM  fc->drawers AS {DRAWER*} dr         // iterator: dr iterates over LIST
WHERE dr->color == ``blue''               // predicate
```

The `WHERE` clause consists of an possibly interleaved list of `FROM`-clause iterators, bounding variables to the write of `AS`, and predicates, i.e. conditions, in the WHERE clauses. Any order of iterators and predicates, in which variables are only used after they are bound in iterators is allowed. However, in general, different order may lead, as we shall see, to different resulting collections. In the `SELECT` clause we may have any C++ expression, possibly using the variables bounded in the iterators, or invoking another monoid comprehension.

The semantics of the query is best understood, intuitively, through the following nested loop program, which is a conceptual skeleton of the actual algorithm evaluating monoid comprehensions.[3]

```
result = empty_bag;
```

[3] The real algorithm also deals with many other issues such as persistency, dynamic buffer management, type management and interface with C++ etc.

```
FOREACH fc IN {BAG} all_file_cabinets DO
   IF fc->color == ``red'' THEN
       FOREACH dr IN {LIST} fc->drawers DO
          IF dr->color == ``blue'' THEN
              INSERT fc INTO result
```

If the bag collection `all_file_cabinets` contains a file cabinet `fc` which has 3 blue drawers, it will be inserted in the `result` 3 times, creating 3 copies of `fc` in the `result` bag collection. However, if the collection type of the `result` were SET, then the insertion of `fc` into the `result` more than once would be equivalent to a single insertion. If, on the other hand, the `result` collection were of type LIST, the order in which the insertions are made would also impact the `result` collection.[4] Also important to note is that a query can be always written with just one FROM clause, with all the iterators, followed by just one WHERE clause, with all the conditions. However, we allow any order of interleaved iterators and conditions, in order to control the evaluation of the query, as is necessary for optimization-level languages such as C^3.

The next query demonstrates the construction of CST objects in the SELECT clause; it finds, for each desk in `all_desks`, its extent in the rooms coordinates, assuming the center of the desk is located at the point $(6, 4)$ and the desk orientation is aligned with the rooms walls, i.e. translation equation is $u = w+x \wedge v = z+y$.

```
SELECT new CST( (u,v) | (dsk->extent &&       // recall: in var's w and z
                 u == w + x && v == z + y &&  // translation of coord.
                 x == 6 && y == 4) )          // location of the center
INTO  {Bag<CST*>} result
FROM  all_desks AS {DESK*} dsk                 // iterates over BAG
```

Note that `Bag<CST*>` in the INTO clause indicates the type of the `result`. The notion `dsk->extent`, in C^3 as well as in C++, stands for `(*dsk).extent` i.e. the attribute `extent` of the object referenced by (i.e. pointed to) variable `dsk`. The next query finds all pairs `(dsk, dsk->extent)`, for all desks that, if centered at $(6, 4)$, would intersect the area `(3 <= u <= 4 && 8 <= v <= 10)` in the room.

```
SELECT new pair(dsk,dsk->extent)
INTO   {Bag<pair*>} result
FROM   all_desks AS {DESK*} dsk
DEFINE area   AS {CST} (3 <= u <= 4 && 8 <= v <= 10)
DEFINE transl AS {CST} (u == x + w && v == y + z)
WHERE  SAT(area && dsk->extent && transl && x == 6 && y == 4)
```

Here, `pair(dsk, dsk->extent)` is a constructor of the class `pair`; expressions `DEFINE expr1 AS expr2` cause the replacement `expr1` by `expr2` in the remainder of the comprehension; they are used simply as shortcuts. `SAT` stands for satisfiability test of the constraint expression inside the parentheses, to check whether the `area` intersects the desk's extent.

The following query exemplifies the use of IMPLY predicate in the where clause and some geometrical manipulation of CST objects in the SELECT clause. For all desks that, if located at $(6, 4)$, contain the `area` `3 <= u <= 4 && 8 <= v <= 10`, it finds the desk's extent above the diaganal (45 degree) through its center. .

```
SELECT new CST(dsk->extent && w <= z)      // Note: w <= z for above
INTO   {Bag<CST*>} result                  //       45 degree diagonal
FROM   all_desks AS {DESK*} dsk
DEFINE area   AS {CST} (3 <= u <= 4 && 8 <= v <= 10),
DEFINE transl AS {CST} (u == x + w && v == y + z),
DEFINE dsk_ext_in_room AS
            {CST} (u,v) | (dsk->extent && transl && (x == 6 && y == 4))
WHERE  IMPLY(area, dsk_ext_in_room)         // To test containment
                                            // of area in dsk_ext_in_room
```

[4] In fact, as discussed in further sessions, the formal definition of monoid comprehension disallows to create LIST collection in the `result` in our example, since it has more "structure" than BAG, on of the collection types used inside; we, however, do allow such situation.

Note that `dsk_ext_in_room` stands for the desk's extent in the room's coordinates. Finally, the last query finds all desks whose drawer may intersect, if closed or partly or fully open, the desk area `(-1 <= w <= 1 && -1 <= z <= 1)`

```
SELECT dsk
INTO  {Bag<DESK*>} result
FROM   all_desks AS {DESK*} dsk
DEFINE dr_ext    AS {CST} dsk->drawer->extent,
DEFINE dr_loc    AS {CST} dsk->drawer->center,
DEFINE dr_transl AS {CST} dsk->drawer->translation,
DEFINE dr_ext_in_dsk AS
             {CST} (w,z) | (dr_ext && dr_loc && dr_transl),
WHERE  SAT(dr_ext_in_dsk && -1 <= w <= 1 && -1 <= z <= 1)
```

3 C^3 Monoids and Monoid Comprehensions

In this section we describe the syntax, semantics and implementation of C^3 monoids and monoid comprehension. The formal counterpart of C^3 monoid comprehensions is *monoid comprehensions* of [FM], which is a restricted version of *monoid homomorphizms* [BTBN, BTS, BTBW] written using the syntax of *monad comprehensions* [Wad], as is done by [BLS+94]. . We first review the formal definition of monoids and monoid comprehensions borrowing heavily from [FM] and [BW95].

3.1 Review of Monoid Comprehensions

```
BAG { fc | fc ← all_file_cabinets,
           fc->color == ``red'',
           dr ← fc->drawers,
           dr->color == ``blue'' }
```

This is the original monoid comprehension syntax for the first C^3 query in Subsection 2.2. Here, `BAG` indicates the type of the resulting collection (monoid); `fc` to the left of | is what we SELECT; ← is used to denote an *iterator*, i.e. all statements in the FROM clauses; and the rest are *predicates*, i.e. logical conditions appearing anywhere in the WHERE clauses. The intuitive meaning is given by the nested loop program in Subsection 2.2.

In addition to collections, we can also compute aggregation functions. For example,

```
SUM { 1 | fc ← all_file_cabinets,
          fc->color == ``red'',
          dr ← fc->drawers,
          dr->color == ``blue'' }
```

will count the number of `file_cabinets` in the result.

More formally, a set of basic data types given, e.g., `int`, `real` and `char`, and a set of type constructors, e.g., `set`, `list`, `bag`. A *data type* is defined recursively as a basic data type or a constructed type $T(\alpha)$ determined by the type parameter α.

A *monoid* is a triple $(T, \texttt{zero}, \texttt{merge})$, where T is a data type and `merge` is an associative function, of type $T \times T \to T$, with left and right identity `zero`. For example, $sum = (int, 0, +)$ is a monoid. A *collection monoid* is a quadruple $(T(\alpha), \texttt{zero}, \texttt{unit}, \texttt{merge})$, where (1) $T(\alpha)$ is a constructed type determined by the type parameter α, (2) $(T(\alpha), \texttt{zero}, \texttt{merge})$ is a monoid, and (3) `unit` is a function of type $\alpha \to T(\alpha)$. As an example, $(list(int), [], f, ++)$, where [] is the empty list, $f(i) = [i]$ for each i and $++$ is the concatenation operation on lists.[5] Finally, a *primitive monoid* is a quadruple $(T, \texttt{zero}, \texttt{unit}, \texttt{merge})$, where $(T, \texttt{zero}, \texttt{merge})$ is a monoid and `unit` is the identity function of type $T \to T$. Examples of primitive monoids include $prod = (int, 1, id, *)$, where $id(i) = i$ for each integer.

[5] We use $[a_1, \ldots, a_n]$ to denote a list and $\{\!\{a_1, \ldots, a_k\}\!\}$ to denote a bag.

Intuitively, a monoid $\mathcal{M} = (T, \texttt{zero}, \texttt{merge})$ is an abstract definition of a data type. Collection monoids capture the bulk types, and primitive monoids capture the basic types. Each instance of the collection type $\mathcal{M} = (T(\alpha), \texttt{zero}, \texttt{unit}, \texttt{merger})$ is expressed as compositions of functions `zero`, `unit` and `merge` on instances of type α. As an example, the monoid $(list(int), [], f, ++)$ given earlier defines a data type of the integer lists. An instance of the type is intuitively a list of integers and the list is expressed as a composition of functions $[]$, u and $++$ applying on integers. For example, the list $\{1, 2, 3, 1\}$ can be expressed as $++(u(1), ++(u(2), ++(u(3), ++(u(1), [])))$.

A monoid $(T, \texttt{zero}, \texttt{merge})$ is called *commutative* (*idempotent*, resp.) if function `merge` is commutative (idempotent, resp.). For monoids $\mathcal{M}$ and $\mathcal{N}$, we say $\mathcal{N} \preceq \mathcal{M}$ if that $\mathcal{N}$ is commutative (idempotent, resp.) implies that $\mathcal{M}$ is commutative (idempotent, resp.). For example, the monoid $set^{\alpha} = (set(\alpha), \{\}, f', \cup\}$, where $f'(i) = \{i\}$ for each instance i of type α, is a commutative and idempotent monoid, and $bag^{\beta} = (bag(\beta), \{\!\{\ \}\!\}, f'', \hat{\cup})$, where $f''(i) = \{\!\{i\}\!\}$ for each instance i of type β and $\hat{\cup}$ is the additive bag union, is a commutative monoid. It is easily seen that $bag^{\beta} \preceq set^{\alpha}$. If $\mathcal{N} \preceq \mathcal{M}$, then an instance of type $\mathcal{N}$ can be "translated" deterministically, by using the merge function of the monoid $\mathcal{M}$, into an instance of the type $\mathcal{M}$, but not necessarily vice versa.

Queries on monoids are expressed as monoid comprehensions. A *monoid comprehension* over the monoid $\mathcal{M}$ takes the form

$$\mathcal{M}\{e \mid r_1, \ldots, r_n\}$$

where e is an expression called the *head* of the comprehension, and $r_1, \ldots, r_n$ is a list of *qualifiers*, each of which is either

- a *iterator* of the form $v \leftarrow e'$, where v is a variable, and e' is an expression that evaluates to an instance of a collection monoid of type which is $\preceq \mathcal{M}$ or
- a *selection-predicate*, which is an expression that evaluates to `true` or `false`.

An expression in turn can include monoid comprehensions. An important condition for the monoid comprehension is that for each $1 < i \leq n$, each free variables (i.e., free variables in the expressions and predicates) appearing in $r_i, \ldots, r_n$ must appear as the variable of a iterator among $r_1, \ldots, r_{i-1}$, and each free variables in the e must appear as the variable of a iterator among $r_1, \ldots, r_n$.

It is assumed that each instance of a monoid appearing as argument in a monoid comprehension is represented as an expression involving `merge`, `unit` and `zero` functions. For example $\texttt{BAG}\{1, 2, 1, 3\}$ can be represented as

```
merge( merge(unit(1),unit(2)),
       merge(unit(1),unit(3)))
```

or, since `merge` is associative and `zero` is a left (and right) identity, as

```
merge( merge( merge( merge( zero,
                            unit(1)),
                    unit(2)),
              unit(1)),
       unit(3))
```

We will assume that every monoid instance is (conceptually) represented this way, and, thus, the notation $\mathcal{N}\{a_1, a_2, \ldots, a_n\}$ will denote the expression

`merge(... merge(merge(zero,unit(`a_1`)),unit(`a_2`)), ...,unit(`a_n`))`

Furthermore, $\mathcal{N}\{\}$, and $\mathcal{N}\{a_1, a_2, \ldots, a_n\}$ where $n = 0$ will both denote $\texttt{zero}^{\mathcal{N}}$, i.e. the empty monoid instance. .

A monoid comprehension `M` over a (collection or primitive) monoid $\mathcal{M} = (T, \texttt{zero}, \texttt{unit}, \texttt{merge})$ defines an instance of type T [6] by first initializing `result` with $\texttt{zero}^{\mathcal{M}}$, and then invoking the procedure `insert_MC(result,M)` defined recursively by the following reduction rules:

[6] Our definition here is different from, but equivalent to the original one; ours is closer to the implementation.

(r1) $\texttt{insert\_MC(result,}\mathcal{M}\{e\,|\,\}\texttt{)} \rightarrow \texttt{result := merge}^{\mathcal{M}}\texttt{(result,unit}^{\mathcal{M}}\texttt{(}e\texttt{))}$

(r2) $\texttt{insert\_MC(result,}\mathcal{M}\{e\,|\,\text{false},\vec{r}\}\texttt{)} \rightarrow \texttt{nil}$ (i.e. do nothing)

(r3) $\texttt{insert\_MC(result,}\mathcal{M}\{e\,|\,\text{true},\vec{r}\}\texttt{)} \rightarrow \texttt{insert\_MC(result,}\mathcal{M}\{e\,|\,\vec{r}\}\texttt{)}$

(r4) $\texttt{insert\_MC(result,}\mathcal{M}\{e\,|\,x \leftarrow \mathcal{N}\{a_1,\ldots,a_n\},\vec{r}\}\texttt{)} \rightarrow$ **for** $i = 1$ **to** n **do** $\texttt{insert\_MC(result,}\mathcal{M}\{e\,|\,\vec{r}\}[x/a_i]\texttt{)}$

where $\mathcal{N}$ is a collection monoid $(S, \texttt{zero}^{\mathcal{N}}, \texttt{unit}^{\mathcal{N}}, \texttt{merge}^{\mathcal{N}})$ with the condition $\mathcal{N} \preceq \mathcal{M}$. Note that $\mathcal{M}\{e \mid \vec{r}\}[x/a_i]$ denote the replacement of x with a_i in $\mathcal{M}\{e \mid \vec{r}\}$.

3.2 Monoids in C^3

To understand the minimum requirements from primitive and collection monoids, consider the recursive rules defining the result of a monoid comprehension. For a monoid $\mathcal{M}$ to appear in the result of a monoid comprehension, we only need (1) to use $\texttt{zero}^{\mathcal{M}}$ and (2) to know to perform

$$\texttt{result = merge}^{\mathcal{M}}\texttt{(result,unit}^{\mathcal{M}}\texttt{(}e\texttt{))}$$

which is, in fact the $\texttt{insert}^{\mathcal{M}}\texttt{(result,}e\texttt{)}$ operation (i.e. we define $\texttt{insert}^{\mathcal{M}}$ this way). [7] In order for a collection monoid $\mathcal{N}$ to appear inside a comprehension, we only need to be able to *iterate* over $\mathcal{N}\{a_1,\ldots,a_n\}$, i.e. to perform the **for** loop.

The representation (and implementation) of collection monoids in C^3 is based on two C++ template classes, parameterized with the type `A` of collection elements: `CollectionMonoid` and `Iterator`:

```
template < class A > class CollectionMonoid
{
friend class Iterator<A>;
public:
    CollectionMonoid();            //  C++ constructor used as zero
    virtual void Insert( A& ) = 0;
    virtual Iterator<A>* CreateIterator() = 0;
private:                           //  specific subclasses contain
                                   //  actual implementation
};
```

The class `CollectionMonoid` reflects the minimum requirements: it has `zero`, implemented as a class constructor, `Insert` and `CreateIterator`. An `Iterator` object, created by `CreateIterator`, has member functions `First`, `More` and `Next` allowing the use of the C++ **for**-loop. Specific collection monoids implemented in C^3, depicted in Figure 3.2, are implemented each with two classes derived from classes `CollectionMonoid` and `Iterator`, correspondingly, by overloading the public member functions. The collection monoids `list`, `set`, and `bag` are currently implemented using ObjectStore collections. It is important to note that most monoids have other member functions as well, such as `merge`, `unit` (i.e. constructor with an element as argument), `delete` etc., for convenience; we have only explained here the required minimum.

As opposed to collection monoids, primitive ones only require `zero` and `insert` member functions, since they are not used in iterators.

```
template< class T > class PrimitiveMonoid
{
public:
  PrimitiveMonoid( );                // Note: creates zero of monoid
  virtual void Insert( T& ) = 0;
  operator T ();
  static T zero;
protected:
```

[7] For a primitive monoid the name `insert` is probably strange; we really mean by `insert` exactly `result := merge(result,unit`$^{\mathcal{M}}$`(`e`))`. For example, for primitive monoid `sum` $(int, +, 0, \texttt{identity})$, $\texttt{insert}^{\mathcal{M}}\texttt{(result,5)}$ is `result:=result+` 5.

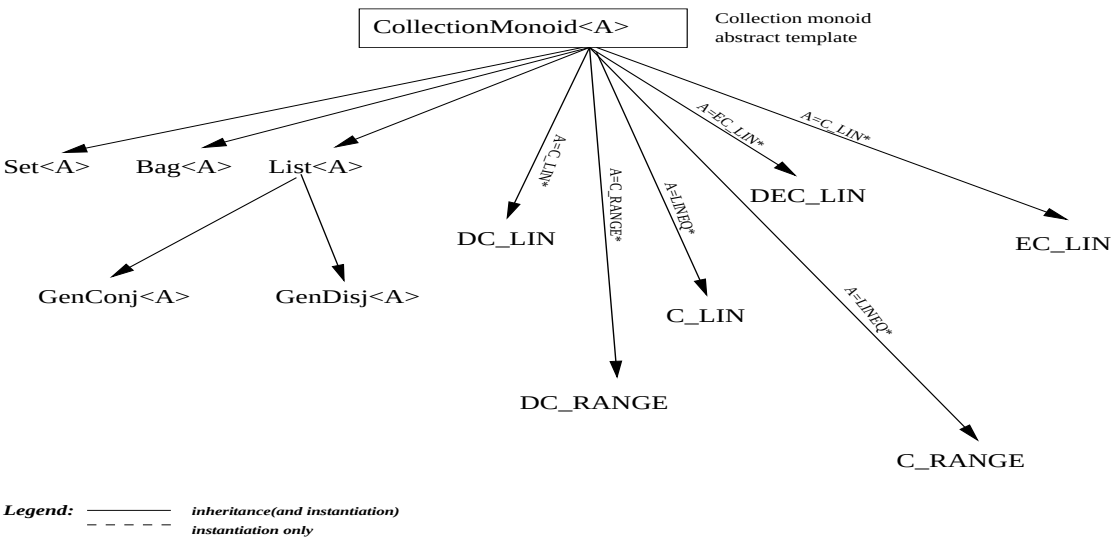

Figure 4: Collection Monoids in C^3

```
  T value;
};
```

Primitive monoids, as opposed to collection ones, are parameterized with the type T. Extensible family of primitive monoids and monoid templates in C^3, depicted in Figure 3.2, include `Prod<T>`, `Max<T>`, `Sum<T>`, `Some` and `All`. Note, that the type used in `Some` and `All` is `Boolean` (encoded as integer in C++) and work as disjunction and conjunction of conditions respectively.

3.3 Syntax and Semantics of C^3 queries

The syntax of C^3 comprehension has been explained by examples. More accurately, it is of the form:

```
SELECT C++expr
INTO [{monoid_type}] [result]
[from-where-define-list]
```

The `C++expr` in the SELECT clause is an arbitrary C++ expression that evaluates to the type `T` of `result's` elements. Note that `C++expr` may involve variables instantiated in the FROM clauses and also may contain nested monoid comprehensions (since they evaluate to types defined in C++). The first (optional) parameter in the INTO clause specifies the type of `result` i.e. the constructed monoid instance and the type of its parameter. If this argument is omitted, the system assumes that `result` is defined elsewhere in the C++ program. When monoid comprehension is nested `result` argument may be omitted. The `from-where-define-list]` is a sequence of FROM, DEFINE and WHERE clauses (explained earlier by examples) in any order. Note that any number of iterators, separated by commas, may appear in each FROM clause; further, any number of predicates (conditions) may appear in each WHERE clause. Also important is that nesting is recursively allowed anywhere in the monoid comprehension, provided that the nested monoid comprehension returns an appropriate type. For instance, collection monoid comprehensions may stand anywhere a collection monoid can; or, monoid comprehensions returning TRUE or FALSE may stand in place of any predicate.

The semantics of C^3 monoid comprehension queries is defined by the corresponding formal monoid comprehension. Furthermore, the basic evaluation is by the nested loop algorithm, with dynamic buffer management. Important, however, is that nested monoid comprehension in the FROM clause do not create physical intermediate results, but rather supporting the overall pipe-lining during query evaluation.

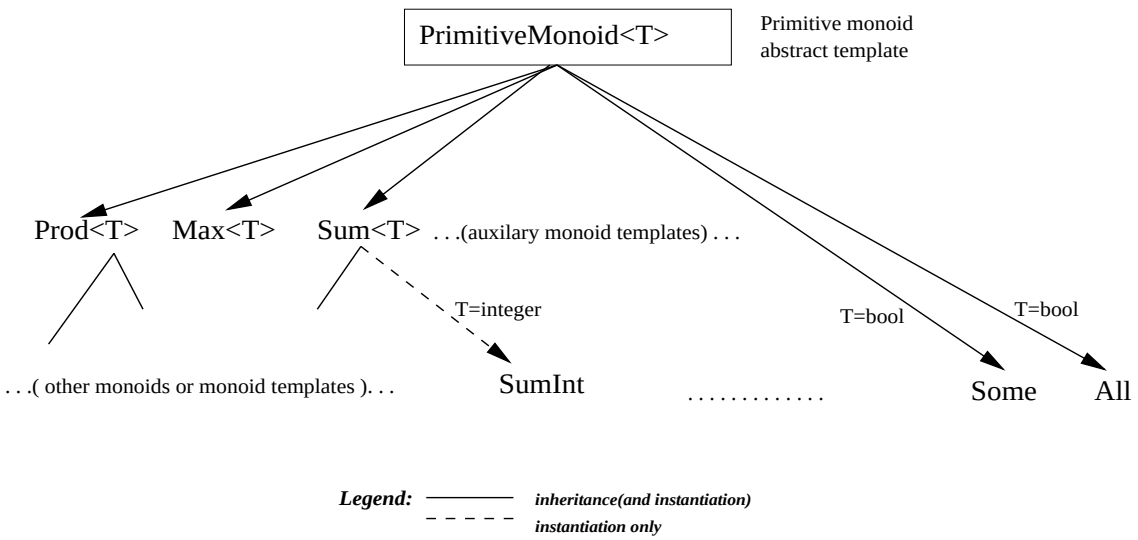

Figure 5: Primitive Monoids in C^3

4 CST Objects and Constraint Calculus

4.1 Framework for Constraint Algebra and Calculus

C^3 uses the framework of [Bro], which we review here. As seen in examples, the notion of CST data relies on a simple and fundamental duality: a constraint (formula) ϕ in free variables $x_1, \ldots, x_n$ is interpreted as a set of tuples $(a_1, \ldots, a_n)$ over the schema $x_1, \ldots, x_n$ that satisfy ϕ; and, conversely, a finitely representable object in $(x_1, \ldots, x_n)$ space can be viewed as a constraint. That is, the syntax is constraints, i.e. symbolic expressions; the semantics are the corresponding, possibly infinite, relations.

CST objects are represented by a sub-family of first order logic, (i.e. with logical connectors $\wedge$, $\vee$, $\neg$ and $\exists$) and a family of atomic constraints, such as linear arithmetic over reals, as in C^3, polynomial or dense order. CST objects are manipulated by means of a *constraint calculus/algebra* that we explain in this section: a sub-family first-order logic, renaming of variables, and atomic (e.g. arithmetic) constraints. For example, if P and Q are CST objects in $x_1, \ldots, x_n$, their intersection can be represented by $P \wedge Q$; union by $P \vee Q$; a test of containment of P in Q by $\forall x_1, \ldots, \forall x_n (P \longrightarrow Q)$ (this is, in fact, the implication test, IMPLY); emptiness of P by $\forall x_1, \ldots, \forall x_n P$ (this is, in fact satisfiability test, SAT) disjointness of P and Q by $\neg(\exists x_1, \ldots, \exists x_n (P \wedge Q)$; projection of P on axes $x_1, \ldots, x_i$, $1 \leq i < n$, by $\exists x_{i+1} \ldots \exists x_n P$ etc. If we only use linear constraint over reals, as implemented in C^3, within first-order logic we can express any linear transformation such as rotation, translation and stretch; check convexity, discreteness and boundness [VGG95]; compute convex hull, augment objects, change coordinate systems; etc.

Thus, constraint objects can be manipulated by a very *expressive language*. Moreover, since this language uses only a small number of operators (i.e. logical connectors and quantifiers), it is also very *compact*, as compared to using a separate operator for each specific type of transformation, which is typically done in extensible or spatial database systems. It is also claimed, that for linear constraints, query languages manipulating constraint objects are deeply *optimizable*, in terms of indexing and filtering (e.g. [BLLM95, KRVV93, Sri92], and constraint algebra algorithms and global optimization (e.g. [BJM93, GK]).

More specifically, constraint algebras operate on a family $\mathcal{F}$ of canonical representations of constraint expressions (objects). For constraint objects $C_1, \ldots, C_n$ a first-order logic formula $\phi(C_1, \ldots, C_n)$ such as $\exists y (C_1[u_1/y, v_1/z] \wedge \ldots \wedge C_n[u_n/y, v_n/z])$, where $[u_i/y, v_i/z]$ denotes variable replacement, defines the following constraint algebra operator *op*: (1) replace each C_i by the corresponding constraint expression, (2) do all variable replacements and (3) transform the resulting constraint expression into the required (equivalent) canonical representation in $\mathcal{F}$. Thus, *op* can be seen as a function from $\mathcal{F} \times \ldots \times \mathcal{F}$ to $\mathcal{F}$. On the other

hand, the operator *op* can be interpreted as a calculus query

$$\mathcal{I}(op) = \{(x_1, ..., x_m) \mid \phi(\mathcal{I}(C_1), \ldots, \mathcal{I}(C_n))\}$$

where $\mathcal{I}(C_i)$, $1 \leq i \leq n$, is the relational interpretation of C_i and $x_1, \ldots, x_m$ are all free variables; thus $\mathcal{I}(op)$ is a function that maps n relations to one. Clearly, the duality between constraints and point sets carries over to the constraint algebra/calculus, that is, the following commutative property holds:

$$\mathcal{I}(op(C_1, \ldots, C_n)) = \mathcal{I}(op)(\mathcal{I}(C_1), \ldots, \mathcal{I}(C_n))$$

A constraint family $\mathcal{F}$ is defined by choosing (1) an atomic constraint domain, (e.g. polynomial over reals or linear over integers), (2) the structure of the logical formula allowed (e.g. disjunction of conjunctions or existentially quantified disjunction) and (3) the required canonical form (e.g. whether eliminate existential quantifiers, eliminate each redundant disjunct, extract all implicit equalities in conjunction, or eliminate redundancy in conjunctions). The definition of constraint algebra amounts to choosing the structure of first-order formulae and the atomic constraints allowed in the query.

The challenge here (and a major area of research) is the development of constraint families and algebras, that strike, for each application realm, a careful balance between (1) *expressiveness*, (2) *computational* complexity and, very importantly, (3) *representation usefulness*.

As one extreme, if the entire first-order logic (as studied in [ACGK94, VGG95]), and the same atomic constraints are allowed in both constraint family $\mathcal{F}$ and algebra, we get a very expressive algebra with low data complexity, since no actual manipulation of constraints would be required. However, the representation of the result might consist of a very large unsimplified constraint expression that might be not useful to the user. For instance, the answer to a query "is constraint object C empty" would be $\exists x_1 \ldots \exists x_n C$, where $x_1, \ldots, x_n$ are all free variables, whereas the user expects a `true` or `false` answer.

An example of a very expressive, but having high (exponential) time data complexity is the DISCO (Datalog with Integer and Set order COnstrains) query language [BR]. Constraint representation is DISCO is useful in many, but not all applications. For example, to express satisfiability of a simple propositional formula, the user needs to encode the formula by a datalog (with constraints) program, in a fairly unnatural way.

Close to the other end, the framework [KKR90] requires a fairly restricted sub-family of first-order logic in constraint objects: disjunction of (unquantified) conjunctions of atomic constraints (the algebra, however, allows more, including quantifier elimination). This representation is useful for many, but not all applications: for example a constraint representation of a triangle given by vertices $(a_1, b_1), (a_2, b_2), (a_3, b_3)$,

$$\exists t_1 \exists t_2 \exists t_3 (x = a_1 t_1 + a_2 t_2 + a_3 t_3 \wedge y = b_1 t_1 + b_2 t_2 + b_3 t_3 \wedge t_1, t_2, t_3 \geq 0 \wedge t_1 + t_2 + t_3 \leq 1)$$

is not directly representable in that framework. Still, for some atomic constraint families, such as linear inequalities over reals, this framework may be computationally unmanageable: the quantifier elimination may result in a constraint exponential in the size of the original conjunction, although for many sub-families more efficient algorithms were developed (e.g. [GK, JMSY92, HLL90, LL91]). A more flexible first-order logic structure that allows the entire linear constraints over reals while controlling computational complexity was described in [BJM93, BK95].

4.2 C^3 Constraint Families and Canonical Forms

In C^3 we concentrate on linear constraint over reals, which are expressive and useful in a variety of application domains. However, in order to control computational complexity, we design a more flexible first-order logic structure by constructing a number of interrelated constraint families. This continues the line work in [BJM93, BK95].

The six interrelated constraint families in C^3 are depicted in Figure4.2. Four main families are for unrestricted linear constraints over reals: `C_LIN`, for conjunctive linear, stands for constraints represented in the form $\wedge_{i=1}^{n} C_i$, where C_i is a linear inequality; `EC_LIN`, for Existential Conjunctive, corresponds to the form $\exists \vec{x} \wedge_{i=1}^{n} C_i$; `DC_LIN`, for Disjunctions of Conjunctions, corresponds to the form $\vee_{i=1}^{m} \wedge_{j=1}^{n} C_{ij}$; and `DEC_LIN`, for Disjunctions of Existential Conjunctive, corresponds to the form $\exists \vec{x} \vee_{i=1}^{m} \wedge_{j=1}^{n} C_{ij}$. The other two families

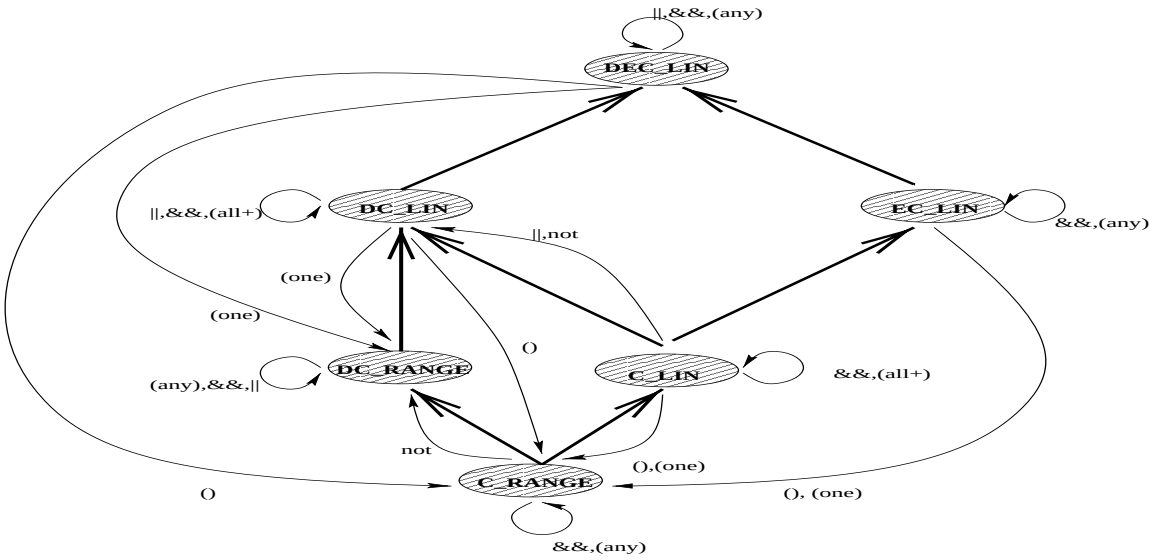

Figure 6: Families of CST Objects

are for range constraints, i.e. of the form a`op` x `op`b, where `op` is either $<$ or $\leq$ and a and b are either real numbers of $-\infty$ or ∞. Namely, `C_RANGE`, for Conjunctive Range, stands for constraints represented in the form $\wedge_{i=1}^{n} C_i$, where C_i is a range constraint; and `DC_RANGE` corresponds to the form $\vee_{i=1}^{m} \wedge_{j=1}^{n} C_{ij}$.

We use the C^3 notation for operations: `not`, `&&`, `||`, and `(...)` for projection. We distinguish between projections on one variable, denoted `(one)`; on zero attributes, denoted `()`, i.e. all free variables are existentially quantified; on all variables, denoted `(all+)`, i.e. no variables are quantified; and, on any number of attributes, denoted `(any)`, for arbitrary projection. The user is recommended to use the most specific projection operator in order to achieve the strongest (i.e. lowest) resulting types.

Not only projection in C^3 can eliminate existing free variables, but they can also add new ones. For example, a CST `(1 <= x >= 5)` can be transformed by the "projection" on `(x,y)` into `(x,y) | (1 <= x <= 5)`, thus adding new free variable `y`, and getting new interpretation as a relation over `x,y` of all tuples with `x` as required and arbitrary real number `y`. However, in the classification of the projection cases discussed earlier, we only consider free variables physically appearing in the constraint expressions.

Thick arrows indicate type hierarchy. For example, `C_LIN` is a sub-type of `DC_LIN`, `EC_LIN` and, transitively, of `DEC_LIN`, meaning that a CST object of type `C_LIN` may be used as argument wherever its supertypes are allowed.

Thin arrows indicate, for each constraint family, the allowed operations and the type of the result, which may belong to a different constraint family. For example, `&&` is allowed in `C_LIN`, returning the result in the same family, while `()`, and `(one)` return the result in `C_RANGE` (which is also in `C_LIN` as a supertype of `C_RANGE`). Note, that the result of `||` on arguments from `C_LIN` will be in `DC_LIN`, not in `C_LIN`. Some of the operations are implicit: for instance, while `||` does not explicitly appear in `C_LIN`, it can be applied since it is allowed for a supertype `DC_LIN`.

Operators may be overloaded: for example `&&` in `C_LIN` is different from `&&` in `DC_LIN`; they are implemented differently and return results of different types (with different representations). The actual

operator applied depends on the types of its arguments. As an example, `&&(C_LIN,C_LIN` will use the `C_LIN` operator; whereas, `&&(DC_LIN,DC_LIN`, as well as `&&(C_LIN,DC_LIN` will use the operator from `DC_LIN`. In general, for the application of `op(arg1,arg2)` we use the lattice structure of type hierarchy, where $\texttt{sub-type} \preceq \texttt{super-type}$ (i.e. the higher the bigger). The actual `op` chosen is the one of the CST type that is the `least upper bound` of `type(arg1)` and `type(arg2)` in which `op` is defined. Note, that the CST families are constructed in such a way that, for every `op` used, such least upper bound, if exists, is unique; hence, there is no ambiguity. If no such bound exists, `op` is not allowed on `arg1` and `arg2` and would result in a compile-time error.

For users of the C^3 CST library it is easy to remember what's allowed and what's not. On arguments of any CST family `&&`, `||`, and `()` can be freely applied. Only `not` is restricted: it can only be applied to arguments of the type `C_LIN` (and thus its sub-type `C_RANGE`). The system will always produce the strongest (i.e. least) type possible for the resulting constraint.

In addition to logical algebraic operators, all families have the following operators:

1. `RENAME(CST-obj,[x1/e1,...,xN/eN])` where `x1,...,xN` are variables to be replaced with variables or constants `e1,...,eN`.

2. `SAT(CST-obj)` to check satisfiability of a CST object `O`, i.e. whether there exists assignment of real constants into its free variables that make `O` true. `MUT_SAT(CST-obj-1, CST-obj-2)` which is equivalent to `SAT(CST-obj-1 && CST-obj-2)`.

3. `TRUTH_VALUE(TV-ASSIGN,CST-obj)` returns the truth value of CST under an assignment of `VAR-ASSIGN` of constants into CST-obj free variables.

4. `MIN-POINT(lin-func,CST-obj)` and `MAX-POINT(lin-func,CST-obj)` where `lin-func` is a linear function with real coefficients. Returned is the assignment of constants into variables of `lin-func` that maximizes it subject to constraints in CST.

5. `MIN(var-name,CST)` and `MAX(var-name,CST)` that return MIN and MAX of the first argument subject to constraints in the second.

Note that the MIN and MAX operators correspond to the problem of linear programming. In addition, `IMPLY(DC_LIN,C_LIN)` operator is allowed[8].

Finally, since all disjunctive CST objects can be considered as collections of disjuncts and conjunctive objects as collection of conjuncts, we make these CST families collection monoids C^3 by implementing the required iterators and member functions.

The six CST families are carefully constructed with the complexity consideration in mind as follows. First, all operations allowed on the families have polynomial data complexity. This is the reason, for example, that `C_LIN` is not closed under general projection: transforming the result into `C_LIN` will require quantifier elimination and thus the size of the result (and, of course, time complexity) may be exponential in the number of variables eliminated. Whereas, `EC_LIN` is closed under general projection since general projection in `EC_LIN` is *lazy*: `EC_LIN` allows quantifiers in the internal representation and hence no physical quantifier elimination is performed. Similar, `not` is allowed on conjunctive CST families, `C_RANGE` and `C_LIN`, but not on, say, `DC_LIN`. The reason is that transforming an expression of the form $\neg \vee \wedge C$ or, of the form $\wedge \vee \neg C$, into `DC_LIN` may result in expression of exponential size, which we would like to avoid. We discuss what operators involve computationally in more detail in the next subsection.

The CST families use canonical forms, i.e. useful standard forms, of constraints, that we adopt in C^3 from [LHM89, BJM93] and review here from [BJM93]. For CST objects in disjunctive families, some disjuncts might be redundant in the sense that omitting them results in an equivalent constraint. Clearly, a canonical form that eliminates such disjuncts would be desirable. However, the problem of detecting such tuples is co-NP-complete [Sri92], and so we will perform only one simplifications of disjunctions: the deletion of inconsistent disjuncts.

For CST objects in conjunctive families, there are a number of simplification that can be requested by the user. One choice is to write all equations in the form $\{x_i = t_i \mid i = 1, \ldots, n\}$ where the x_i's are distinct and appear nowhere else in the constraint. A second choice is whether all equations which are implicit in the

[8] and, of course, for all subtypes of `C_LIN` and `DC_LIN`

inequality constraints should be represented explicitly. (As a simple example of this, consider the constraints $x + y \leq 2, x + y \geq 2$.) A third is the extent to which redundancy within the inequalities should be removed. [?] presents a classification of redundancy that suggests simple forms of redundancy removal. A fourth choice is whether to keep the inequalities in a different form, such as simplex tableau form. In the current C^3 implementation, the only simplification is the removal of inconsistent disjuncts in disjunctive families; however, a range of simplifications on conjunctions is presently being implemented.

4.3 Implementation of CST families

On conjunctive families, `&&` operator simply combines two conjunctions and is constant time; `(one)| C`, which is a projection on a single variable, involves applying linear program (using simplex algorithm of CPLEX) twice for finding minimum and maximum of the variable subject to `C`; `()|C`, which is eliminating all variables in `C` works as a satisfiability test, using the first phase of simplex, as does the `SAT` predicate.

On disjunctive families, `||` operator is constant-time, while `D1 && D2`, where $D1 = \vee_{i=1\ldots n} C1_i$ and $D2 = \vee_{j=1\ldots m} C2_j$ is more involved:

$$D1 \wedge D2 = \bigvee_{i=1,\ldots,n} C1_i \wedge \bigvee_{i=1\ldots m} C2_j = \bigvee_{i=1\ldots n, j=1\ldots m} (C1_i \wedge C2_j)$$

that is, the result consists of all combinations of $C1_i$ and $C2_j$ that are mutually consistent (i.e. their conjunction is satisfiable). Since the CST families have properties of monoids (i.e. they are monoids) `D1 && D2` are represented as the following C^3 query:

```
SELECT *c1 && *c2
INTO  {DEC_LIN} conj_D1_and_D2
FROM  D1 AS {EC_LIN*} c1,
      D2 AS {EC_LIN*} c2
WHERE SAT(*c1,*c2)
```

We show how such queries are optimized using approximation-based filtering, indexing and re-groupings in the next section. Similar, `SAT(D)`, where `D` is of type `DEC_LIN` (as well as other disjunctive families) is represented as

```
SELECT SAT(*c)
INTO  {Some} satisf_flag
FROM  D AS {EC_LIN*} c
```

Note, that `c` is of type `EC_LIN` and so SAT in the WHERE clause works on conjunctions. Further recall that `Some` is a primitive monoid whose `merge` operator is a logical or; thus, the `satisf-flag` will be true if and only if at least one component is true. Finally, `IMPLY(D,C)`, where `D` is `DC_LIN` and `C` is `C_LIN`, is represented as

```
SELECT IMPLY(*c,C)
INTO  {Some} imply_flag
FROM  D AS {C_LIN*} c
```

Beyond algorithms for constraint operations discusses, there are two subtle design problems that we address in C^3: compile-time maintenance of the type lattice and lazy evaluation. Support for lazy evaluation of constraint (i.e. involving CST) expression is necessary for efficiency. For example, if we are interested in SAT test of an expression involving logical connectors, it is typically wasteful to perform simplifications of subexpressions.

To exemplify the problem arising from the type lattice maintenance, consider the `&&` operator. In fact, while `&&` has one conceptual meaning, it works differently in every CST family. Moreover, since `&&` is defined on `DEC_LIN`, the arguments may be any subtypes of `DEC_LIN`. Thus, every ordered pair of (sub) types for arguments of `&&` works uniquely: we need to find the least upper bound type, to perform corresponding type conversions, and then to apply a physical algorithm of the resulting CST family. One possibility is implementing a separate function for each pair of subtypes, but this would result in a quadratic number

of functions for each logical operator: 30 for six families, and unrealistically many for future extensions of C^3 with new CST families. On the other hand, implementing a subtype relationship with C++ derived classes does not work, since each family has a different implementation, data-structures etc, and should not be inherited as what would have happened with C++ classes. Of course, there is also a possibility of maintaining just one global CST type, and to distinguish individual subfamilies only at run time. This, however, would eliminate the capability of compile-time type checking, an important feature of C^3.

To solve the type lattice problem we designed a two-layer architecture for CST families: the lower layer, called `basic_CST`, supports physical representation and manipulation of CST families; and the upper layer, called `lazy_typed_CST`, which is responsible for type lattice management and lazy evaluation, while actual evaluation is passed to the lower, `basic_CST` layer.

The `basic_CST` layer is composed of the six classes `basic_C_LIN`, `basic_DC_LIN` etc., each maintaining its own datastructures to represent the underlying constraints; and one super (base) class, `basic_CST`. All the discussed operations are implemented at this layer as C++ friend functions, that do no automatic sub-typing is supported. However, each `basic` family has member functions for explicit type conversion into `basic` types of CST families that are higher in the type hierarchy. For example, transforming `basic_C_LIN` into `basic_DC_LIN` creates a `basic_DC_LIN` object (disjunction) that have a single disjunct in it.

The `lazy_typed_CST` layer, on the other hand, does support automatic sub-typing and the ability to determine least upper bounds of operators' arguments at compile-time. This is achieved, by six families `lazy_typed_C_LIN`, `lazy_typed_DC_LIN` etc., implemented as six classes with class hierarchy that exactly matches the type hierarchy of CST families, and one super (base) class `lazy_typed_CST`. However, all `lazy_typed` classes have exactly the same type of internal representation, which is inherited from `lazy_typed_CST`. It is basically an expression tree (hence "lazy"), with internal nodes storing operators (such as `&&` or `||`) and encoding the strongest type to which the subtree can be converted; the leaves are objects of the lower layer, `basic_CST`. It is important to emphasize that CST type checking we do in C^3 heavily uses capabilities of C++ and would be impossible (at compile-time without precompiling) in languages such as C.

Finally, C^3 also supports two generic parameterized CST families: `Gen_Conj<T>` for generic conjunction and `Gen_Disj<T>` for generic disjunctions, where `T` is an arbitrary, possibly complex, CST type. Both are collection monoids and support `TRUTH_VALUE` function; further, `SAT` is supported on `Gen_Disj<T>` provided it is supported for `T`, and `IMPLY(Gen_Disj<T>,T` provided it is defined on `T`. These operations are represented again with monoid comprehension queries. For example, `SAT(D)`, where `D` is of type `Gen_Disj<T>`, is represented as

```
SELECT SAT(c)
INTO  {Some} satisf_flag
FROM  D AS {T} c
```

5 Optimization by Approximation-based Filtering and Indexing

General optimization of object-oriented queries (e.g. ENCORE [Zdo], O2 [ea90], POSTGRESS [SRH90]) and monoid comprehensions in particular, (e.g. [FM]), as well as optimization in presence of expensive predicates [CS, HS] is outside the scope of this paper; We concentrate here on approximation-based filtering, regrouping and indexing [BW95], that C^3 is designed to support. More specifically, we describe, mostly by examples, the C^3 primitives for *approximation* and *inverse groupings* [BW95] and *indices* and special purpose algorithms.

To understand the idea, we use a modification of an example from [BW95] of the query: "find all trajectories passing over Fairfax county". It will be assumed here that a set of 4D aircraft trajectories as well as a map is stored in the database. A trajectory is assumed to have a piece-wise linear representation, i.e. it is represented as `DC_LIN` CST object

$$\bigvee_{i=1}^{n} (t_{i-1} \leq t < t_i \wedge x = a_{i,1}t + b_{i,1} \wedge y = a_{i,2}t + b_{i,2} \wedge z = a_{i,3}t + b_{i,3})$$

Where x, y, z are variables for a location, t is a time variable, and t_{i-1}, $a_{i,1}, b_{i,1}, a_{i,2}, b_{i,2}, a_{i,3}, b_{i,3}$, $1 \leq i \leq n$, are constants. Note that for each i, $(t_{i-1} \leq t < t_i \wedge x = a_{i,1}t + b_{i,1} \wedge y = a_{i,2}t + b_{i,2} \wedge z = a_{i,3}t + b_{i,3})\}$

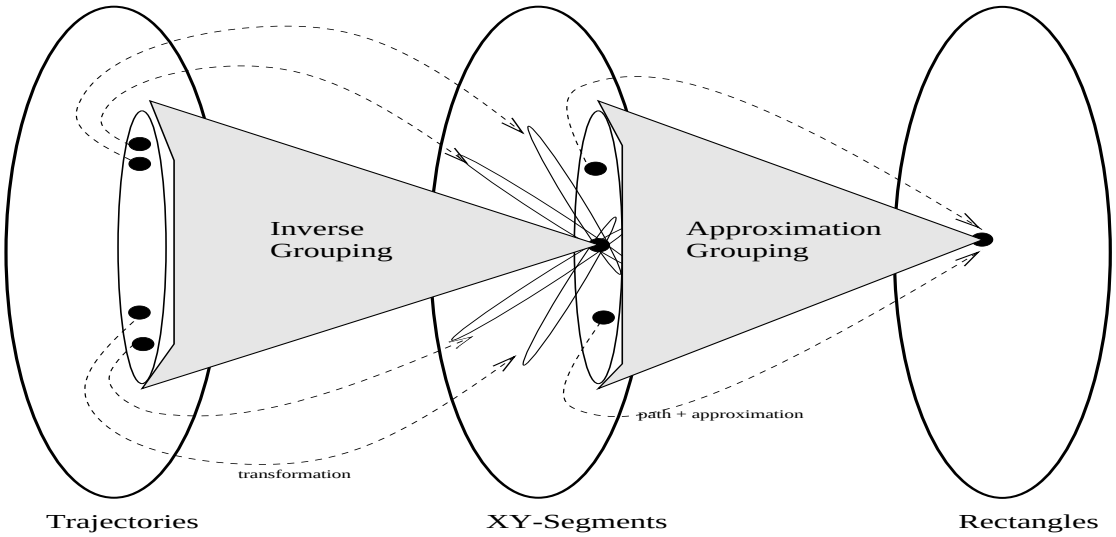

Figure 7: Inverse and Approximation Grouping

describe movement equations for the time interval $[t_{i-1}, t_i)$, for constant 3-D velocity vector $(a_{i,1}, a_{i,2}, a_{i,3})$, starting from the point $(b_{i,1}, b_{i,2}, b_{i,3})$. `All_trajectories` is a variable of type `SET<DC_LIN*>`, i.e. a set of pointers to trajectories. The Fairfax county is assumed to be represented as a polygon,[9] i.e. as a `C_LIN` CST object `Fairfax-area` in variables x and y. The query can be directly expressed in C^3 as

```
SELECT traj
INTO {Set<DC_LIN*>} result
FROM All_Trajectories AS {DC_LIN*} traj
WHERE  MUT_SAT(*traj, Fairfax_area )              // Note: MUT_SAT on DC_LIN
```

or, if `MUT_SAT` is expressed, in turn, as monoid comprehension:

```
SELECT traj
INTO {Set<DEC_LIN*>} result
FROM All_Trajectories AS {DC_LIN*} traj
WHERE SELECT MUT_SAT(*segment,Fairfax_area)  // Note: MUT_SAT on C_LIN
      INTO {Some}                            // SELECT returns True of False
      FROM *traj AS {C_LIN*} segment
```

which is an expensive query if evaluated directly. To optimize, we can first use the *inverse grouping*, described graphically in Figure 5. Intuitively, each trajectory can be viewed as composed of 4D-segments, and each segment has a projection, say `xy-segment` on the horizontal plane `x,y`. Thus, each trajectory has a corresponding sets of `xy-segments` (see dash lines in Figure 5). The inverse grouping, denoted `IG`, is a structure that remembers, for each `xy-segment` the set of all trajectories having `xy-segment` as one of their segment's projections. More accurately, All-Traj-IG here is a structure of the type `Set<IG_Pair*>`, where the class `IG_Pair` represents a pair of arguments: the second argument, of the type `C_LIN*`, stands for an `xy-segment`; and the first, of the type Set¡DC_LIN*¿*, stands for the set of all (pointers to) corresponding trajectories. If All-Traj-IG is (dynamically) maintained, the query can be re-written:

```
SELECT traj
INTO {Set<DC_LIN*>} result
FROM  All_Traj_IG AS {IG_Pair*} IG_pair
```

[9] in fact, it is not convex, but we'll assume that to simplify the example

```
//
DEFINE xy_segment AS {C_LIN*} IG_pair->second
WHERE MUT_SAT( *xy_segment, Fairfax_area )       // Note: MUT_SAT on C_LIN
//
DEFINE trajectories_of_xy_segment
       AS {Set<DC_LIN*>*} IG_pair->first
FROM  *trajectories_of_xy_segment AS {DC_LIN*} traj
```

Next, since `MUT-SAT` for `C_LIN` is typically expensive, we can approximate each `xy-segment` with a minimum bounded box (MBOX), which is of type `C_RANGE`. Then, before testing `MUT_SAT` on `C_LIN` it can be tested first on MBOXes. To do that we use `approximation grouping`, which stores, for each created MBOX, the set of `xy-segments` (in fact, its IG pairs) approximated with that MBOX. More accurately, `All-Traj-IG-AG` here is a structure of the type `Set<AG_Pair*>`, where the class `AG-Pair` represents a pair of arguments: the second argument, of the type `C_RANGE*`, stands for the minimum bound box; the first, for the corresponding IG-pairs, has the type `Set<IG_Pair*>*`. If approximation grouping is dynamically maintained the query can be re-written as follows:

```
SELECT traj
INTO {Set<DC_LIN*>} result
//
FROM All_Traj_IG_AG AS {AG_Pair*}   AG_pair
DEFINE min_box_of_xy_segment AS {C_RANGE*} AG_pair->second  // Note: C_RANGE
WHERE  MUT_SAT(*min_box_of_xy_segment,             // On C_RANGE;
            min_box_of_Fairfax)         // precomputed outside of query
DEFINE Candidate_Traj_IG AS {Set<IG_Pair*>*} AG_pair->first
//
FROM  *Candidate_Traj_IG  AS {IG_PAIR*} IG_pair
DEFINE xy_segment  AS {C_LIN*} IG_pair->second
WHERE Mut_Sat( *xy_segment, Fairfax_area )          // Note: MUT_SAT on C_LIN
//
DEFINE trajectories_of_xy_segment
     AS {Set<DC_LIN*>*} IG_pair->first
FROM  *trajectories_of_xy_segment AS traj
```

Finally, we can use index structures based, for instance, on R-trees with rectangles as values. This is done using *indexed approximation grouping* (IAG). Specifically, we create here `All_Traj_IG_IAG`, which has the same structure as `All_Traj_AG`, and, in addition, an R-tree index imposed on the second argument. To support search queries, IAG has member functions for search. In our case we use `MUT_SAT(C_RANGE)`, that returns, for each MBOX (of type `C_RANGE`) the set of all corresponding `IG-PAIRS`, i.e. of type `Set<IG-PAIR*>*`. Note, that the returned set is not a physical set collection, but rather a structure allowing to iterate over its elements (and thus no intermediate evaluation is necessary when used within monoid comprehension). If IAG is dynamically maintained (instead of AG), the (first part of) the query can be re-written as follows:

```
SELECT traj
INTO {Set<DC_LIN*>} result
//
FROM All_Traj_IG_IAG.MUT_SAT(min_box_of_Fairfax)
                    AS {Set<IG_Pair*>*} Candidate_Traj_IG
//
FROM  *Candidate_Traj_IG  AS {IG_Pair*} IG_pair
DEFINE xy_segment  AS {C_LIN*} IG_pair->second
WHERE Mut_Sat( *xy_segment, Fairfax )          // Note: MUT_SAT on C_LIN
//
DEFINE trajectories_of_xy_segment AS {Set<DC_LIN*>*} IG_pair->first
FROM  *trajectories_of_xy_segment AS traj
```

It is important to note, that, while we intuitively explained the use IG, AG and IAG by examples, these primitives can be applied to any `CollectionMonoid<A>`. For IG grouping the user needs to provide a transformation `f` producing, for each element of type `A` an instance of (another) commutative and idempotent monoid (see [BW95] for details). For AG and IAG grouping, an approximation of elements of type `A` must be provided by the user. The IG, AG and IAG groupings are used to facilitate the query transformation rules supporting approximation-based filtering (by using less expensive predicates first) and indexing (see [BW95]).

6 Related Work

No technology for declarative and efficient querying of databases involving constraint objects exists today. Applications of the kind discussed are typically implemented by special purpose programs; while these programs may use database and constraint programming tools, they typically require considerable programming effort and are not flexible to changes. In addition, they do not perform overall optimization that interleaves database, mathematical programming and computational geometry manipulation techniques. Existing DBMS do not manage constraints as persistent stored data [10]. Constraint Logic Programming [JL87, DHS+88, Col90], on the other hand, was not designed to deal with large amounts of persistent data. Extensions of DBMS with spatio-temporal operators [OM88, Gut89, Wol89, HC91] typically (1) are limited to low (two- or, at most three-) dimensional space, (2) have query languages restricted to predefined spatio-temporal operators, and (3) lack global economical filtering and deep optimization.

There has been work on the use of constraints in databases, earlier of which include [Klu88, HHLvEB, CI89, Ram91, BS89]. The pioneering work [KKR90] proposed a framework for integrating abstract constraints into database query languages by providing a number of design principles, and studied, mostly in terms of expressiveness and complexity, a number of specific instances. The work [HHLvEB] considered polynomial equality constraints, adopting local propagation steps for reasoning on constraints. A restricted form of linear constraints, called *linear repeating points*, was used to model infinite sequences of time points [KSW90, BNW91, NS92]. More recent works on deductive databases [MFPR90, SR92, KS92, LS92] considered manipulation and repositioning of constraints for optimizing recursion. Algorithms for constraint algebra operators such as constraint joins, and generic global optimization were studied in [BJM93], and constraint approximation-based optimization in [BW95]. The work [KRVV93] proposed an efficient data structure for secondary storage suitable for indexing constraints, that achieves not only the optimal space and time complexity as priority search trees [McC85], but also full clustering. The work [BLLM95] proposed an approach to achieve the optimal quality of constraint and spatial filtering. A number of works consider special constraint domains: integer order constraints [Rev93]; set constraints [Rev95]; dense-order constraints [GS95]. Linear constraints over reals drew special attention [ABK95, ACGK94, BJM93, BK95, BLLM95, SG95, VGG95]. The use of constraints in spatial database queries was addressed in [PdBG94]. The work [SRR94] used constraints to describe incomplete information. Constraint aggregation was studied in [Kup93].

DISCO (Datalog with Integer and Set order COnstraints) is a constraint database system being developed at the university of Nebraska [BR]. DISCO incorporates a highly expressive family of constraints. However, its query language has time complexity exponential in the size of a database; hence DISCO's applicability to real-size database problems is not clear. Further, DISCO does not supports standard database features such as persistent storage, transaction management and data integrity.

7 Conclusions and Future Work

We have described the work on the development of the first constraint object-oriented database system. Our work aims at the developing a practical and useful technology for a wide variety of important application realms, for which no existing technology is applicable. For example, C^3 can be directly used to implement the real-life data fusion and sensor management system for air-space command and control [ABK95], and has the ability to achieve performance comparable to special-purpose techniques. C^3 is a deeply optimizable and extensible system, striking the balance between expressiveness and computational complexity.

[10] Note, integrity constraints used in conventional databases are not data, but rather something the data must satisfy.

Many research questions remain open (see [Bro] for an overview): in constraint modeling and canonical forms, data models and query languages, indexing and approximation-based filtering, and, most importantly, special constraint algebra algorithms for specific domains and global optimization.

References

[ABD+96] T. Atwood, D. Barry, J. Dubl, J. Eastman, G Ferran, D. Jordan, M. Loomis, and D. Wade. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1996.

[ABK95] T. Aschenbrenner, A. Brodsky, and Y. Kornatzky. Constraint database approach to spatio-temporal data fusion and sensor management. In *Proc. ILPS95 Workshop on Constraints, Databases and Logic Programming*, Portland, OR, December 1995.

[ACGK94] F. Afrati, S. Cosmadakis, S. Grumbach, and G. Kuper. Linear versus polynomial constraints in database query languages. In A. Borning, editor, *Proc. 2nd International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Conmputer Science*, pages 181–192, Rosario, WA, 1994. Springer Verlag.

[BJM93] A. Brodsky, J. Jaffar, and M.J. Maher. Toward practical constraint databases. In *Proc. 19th International Conference on Very Large Data Bases*, Dublin, 1993.

[BK95] A. Brodsky and Y. Kornatzky. The lyric language: Quering constraint objects. In Carey and Schneider, editors, *Proc. ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.

[BLLM95] A. Brodsky, C. Lassez, J.-L. Lassez, and M. J. Maher. Separability of polyhedra for optimal filtering of spatial and constraint data. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 1995.

[BLS+94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 1994.

[BNW91] M. Baudinet, M. Niezette, and P. Wolper. On the representation of infinite temporal data and queries. In *Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems*, 1991.

[BR] J. Byon and P.Z. Revesz. Disco: A constraint database system with sets. In *Proc. Workshop on Constraint Databases and Applications*.

[Bro] A. Brodsky. Constraint databases: Promising technology or just intellectual exercise? In *ACM workshop on strategic directions in computer science, to appear. Also, to be part of constraint programming survery in ACM Computing Survery, to appear.*

[BS89] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in datalog programs. In *Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems*, pages 190–199, Philadelphia, 1989.

[BTBN] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. Third International Workshop on Database Programming Languages*.

[BTBW] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *4th International Conference on Database Theory*.

[BTS] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *In 18th International Colloquium on Automata, Languages and Programming*.

[BVCS93] M. Benjamin, T. Viana, K. Corbett, and A. Silva. Satisfying multiple rated-constraints in a knowledge based decision aid. In *Proc. IEEE Conf. on Artificial Intelligence Applications*, Orlando, 1993.

[BW95] A. Brodsky and X. S. Wang. On approximation-based query evaluation, expensive predicates and constraint objects. In *Proc. ILPS95 Workshop on Constraints, Databases and Logic Programming*, Portland, OR, 1995.

[CI89] J. Chomicki and T. Imielinski. Relational specifications of infinite query answers. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 174–183, 1989.

[Col90] A. Colmerauer. An introduction to prolog 3. *Communications of the ACM*, 33(7):69–90, 1990.

[CS] S. Chauduri and K. Shim. Query optimization in the presense of foreign functions. In *Proc. 19th Intl. Conf. on Very Large Data Bases.*

[DHS+88] M. Dincbas, P. Van Hentenryck, H. Simnis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proc. Fifth Generation Computer Systems*, Tokyo, Japan, 1988.

[ea90] O.Deux et. al. The story of o2. *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[FM] L. Fegaras and D. Maier. Toward an effective calculus for object query processing. In *Proc. ACM SIGMOD Conf. on Management of Data.*

[GK] D. Q. Goldin and P.C. Kanellakis. Constraint query algebras. *Constraints Journal, to appear.*

[GS95] S. Grumbach and J. Su. Dense-order constraint databases. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1995.

[Gut89] R.H. Guting. Gral: An extensible relational database system for geometric applications. In *Proc. 19th Symp. on Very Large Databases*, 1989.

[HC91] L.M. Haas and W.F. Cody. Exploiting extensible dbms in integrated geographic information systems. In *Proc. Advances in Spatial Databases, 2nd Symposium*, volume 525 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[HHLvEB] M.R. Hansen, B.S. Hansen, P. Lucas, and P. van Emde Boaz. Integrating relational databases and constraint languages.

[HLL90] T. Huynh, C. Lassez, and J-L. Lassez. Practical issues on the projection of polyhedral sets. *Annals of Mathematics and Artificial Intelligence, to appear; also IBM Research Report RC 15872, IBM T.J. Watson RC*, 1990.

[HS] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. ACM SIGMOD Conf. on Management of Data.*

[JL87] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proc. Conf. on Principles of Programming Languages*, pages 111–119, 1987.

[JMSY92] J. Jaffar, M.J. Maher, P.J. Stuckey, and R.H.C. Yap. Output in clp(∇). In *Proc. Int. Conf. on Fifth Generation Computer Systems*, volume 2, pages 987–995, Tokyo, Japan, 1992.

[KKR90] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *J. Computer and System Sciences, to appear. (A preliminary version appeared in* Proc. 9th PODS, pages 299–313, 1990.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 393–402, 1992.

[Klu88] A. Klug. On conjunctive queries containing inequalities. *Journal of ACM*, 35(1):146–160, 1988.

[KRVV93] P. Kanellakis, S. Ramaswamy, D.E. Vengroff, and J.S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1993.

[KS92] D. Kemp and P. Stuckey. Bottom up constraint logic programming without constraint solving. Technical report, Dept. of Computer Science, University of Melbourne, 1992.

[KSW90] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling infinite temporal data. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1990.

[Kup93] G. M. Kuper. Aggregation in constraint databases. In *Proc. Workshop on Principles and Practice of Constraint Programming*, 1993.

[LHM89] J-L. Lassez, T. Huynh, and K. McAloon. Simplification and elimination of redundant linear arithmetic constraints. In *Proc. North American Conference on Logic Programming*, pages 35–51, Cleveland, 1989.

[LL91] C. Lassez and J-L. Lassez. Quantifier elimination for conjunctions of linear constraints via a convex hull algorithm. Technical Report RC16779, IBM T.J. Watson Research Center, 1991.

[LS92] A. Levy and Y. Sagiv. Constraints and redundancy in datalog. In *Proc. 11-th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1992.

[McC85] E.M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, May 1985.

[MFPR90] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 314–330, 1990.

[NS92] M. Niezette and J.-M. Stevenne. An efficient symbolic representation of periodic time. In *Proc. of First International Conference on Information and Knowledge management*, 1992.

[OM88] J.A. Orenstein and F.A. Manola. Probe spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Engineering*, 14(5):611–629, 1988.

[PdBG94] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1994.

[Ram91] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *J. Logic Programming*, (11):189–216, 1991.

[Rev93] P. Z. Revesz. A closed form for datalog queries with integer order. *Theoretical Computer Science*, 116(1), August 1993.

[Rev95] P. Z. Revesz. Datalog queries of set constraint databases. In *Proc. International Conference on Database Theory*, 1995.

[SG95] C. Tollu S. Grumbach, J. Su. Linear constraint databases. In *Proc. LCC; To appear in LNCS Springer-Verlag volume*, 1995.

[SR92] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. In *Proc. 11th ACM SIGACT-SIGMOD-SIGART Simposium on Principles of Database Systems*, pages 301–315, 1992.

[SRH90] M. Stonebraker, M. Rowe, and L. Hirohama. The implementation of postgress. *IEEE Trans. on Knowledge and Data Engineering*, 1990.

[Sri92] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence, to appear*, 1992.

[SRR94] D. Srivastava, R. Ramakrishnan, and P. Revesz. Constraint objects. In *Proc. 2nd Workshop on the Principles and Practice of Constraint Programming*, Orcas Island, WA, May 1994.

[VGG95] L. Vandeurzen, M. Gyssens, and D. Van Gucht. On the desirability and limitations of linear spatial query languages. In M. J. Egenhofer and J. R. Herring, editors, *Proc. 4th Symposium on Advances in Spatial Databases*, volume 951 of *Lecture Notes in Computer Science*, pages 14–28. Springer Verlag, 1995.

[Wad] P. Wadler. Comprehending monads. In *Proc. ACM Symposium on Lisp and Functional Programming.*

[Wol89] A. Wolf. The dasdba geo-kernel, concepts, experiences, and the second step. In *Design and Implementation of Large Spatial Databases, Proc. 1st Symp. on Spatial Databases.* Springer Verlag, 1989.

[Zdo] S. Zdonik. Query optimization in object-oriented databases. In *Proc. 23rd annual Hawaii Intl. Conf. on System Sciences.*