Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

SVP - A MODEL CAPTURING SETS, STREAMS, AND PARALLELISM

D. S. Parker
E. Simon

April 1992
CSD-920020

SVP — a Model Capturing Sets, Streams, and Parallelism

D. Stott Parker*
Computer Science Department
University of California
Los Angeles, CA 90024-1596 USA

Eric Simon
Patrick Valduriez
I.N.R.I.A.
Domaine de Voluceau–Rocquencourt
78153 Le Chesnay FRANCE

April 27, 1992

Abstract

We describe the SVP data model. The goal of SVP is to model both set and stream data, and to model parallelism in bulk data processing.

SVP models *collections*, which include sets and streams as special cases. Collections are represented as ordered tree structures, and divide-and-conquer mappings are easily defined on these structures. We show that many useful database mappings (queries) have a divide-and-conquer format when specified using collections, and that this specification exposes parallelism.

We formalize a class of divide-and-conquer mappings on collections called SVP-*transducers*. SVP-transducers generalize aggregates, set mappings, stream transductions, and scan computations. At the same time, they have a rigorous semantics based on continuity with respect to collection orderings, and permit implicit specification of both independent and pipeline parallelism. We achieve these semantics by extending Kahn's networks of parallel processes to operate on collections instead of sequences, and in so doing extract both independent and pipeline parallelism.

*This research supported by NSF grant IRI-8917907.

Contents

1 Introduction

Achieving parallelism in bulk data processing is a relatively old problem, which has recently enjoyed a resurgence of interest. This paper proposes a new approach to addressing the problem. Since many of the issues involved are complex, we begin with first principles.

1.1 Parallel Programming

Parallel programming aims at exploiting high-performance multiprocessor systems. An important objective is to be able to express the parallelism available in an application. There are essentially three ways to accomplish this:

- automatically detect parallelism in programs written with a sequential language (e.g., Fortran, OPS5);
- augment an existing language with explicit parallel constructs that exploit the computational capabilities of a parallel architecture (e.g., C* [32], Fortran90 [38]);
- create a new language in which parallelism can be expressed in an architecture-independent manner.

The first approach can be practical in the short-term, but is faced by many difficult problems. Among these, development of a parallelizing compiler is a major challenge. Methods for automatic program restructuring, and the parallelization of serial programs can produce good results for some programs (e.g., certain scientific programs), but most of the time the resulting speed-up is quite limited. For instance, experiments conducted with the OPS5 rule-based language revealed that in practice, the true speed-up achievable from parallelism was less than tenfold [13]. A related serious problem with this approach is that, in the final analysis, the serial programming paradigm does not encourage the use of parallel algorithms.

The second approach enables the programmer to express parallel constructs such as task creation and inter-task synchronization, thereby providing leverage over parallelism. Although this approach can lead to high-performance, it is generally too low-level and difficult for the programmer. Furthermore, the large variety of parallel architectures result in distinct, architecture-specific extensions to the original language.[1] In order to achieve efficient program execution, the programmer must first become acquainted with the programming paradigm dictated by the architecture of the target machine.

The third approach can combine the advantages of the other two. It can ease the task of programming while allowing the programmer to express non-sequential computation in a high-level way [28]. Once the programmer has specified the algorithmic aspects of his program using high-level programming constructs, automatic or semi-automatic methods can be used to derive a mapping from the computational requirements of the program to parallel hardware. The basis for this mapping is data partitioning (also called data-parallelism), whereby program data can be divided into fragments on which either the same instructions can be executed in parallel (with the SIMD computation model) or different instructions are executed in parallel (with the MIMD computation model). The regularity of the data structures available in the language permits exploitation of different forms of parallelism, such as independent and pipeline parallelism [15].

In this paper, we follow the third approach, and propose a model for parallel database programming where the primary sources for parallelism are parallel set and stream expressions. Parallel programming environments that follow this approach have recently been proposed. For example, in Paragon [9], the primary source for parallelism is parallel array expressions. Paragon is targeted to scientific programming applications and offers the essential features of parallel Fortran languages. Our model is targeted at database applications, and bulk data processing.

1.2 Parallelism for Bulk Data Processing

There are various forms of parallelism. Figure 1 shows four simple kinds of parallelism graphically. A few key ideas can be derived from studying this figure, and applying the parallelism structures there to problems in bulk data processing:

[1] Linda [8] is a notable exception of 'coordination language' with simple, language-independent parallel constructs, which can mate easily with many non-parallel languages.

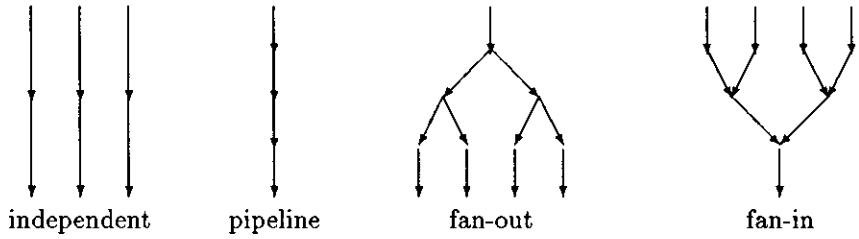

Figure 1: Types of Parallelism

- Division of problems is the essence of parallelism. Dividing into independent subproblems gives independent parallelism, while dividing into incremental computations gives pipeline parallelism. Set mappings naturally expose to independent parallelism (a given instruction is independently applied to each element of a set) while stream mappings expose to pipeline parallelism (some instructions are successively applied to each element of a stream). Thus, sets and streams suggest a divide-and-conquer format for specifying mappings which is implicitly also a format for specifying parallelism.

- Divide-and-conquer computations can be represented with series-parallel graphs. Series-parallel graphs [24] are defined recursively as graphs having one input and one output that can be constructed using two combination rules: series or parallel composition of the inputs and outputs. A typical series-parallel graph is shown in Figure 2. It models a situation where 1 and 2 are performed in parallel before 3, and 3 is performed before the parallel execution of 4, 5, and (6 followed by 7).

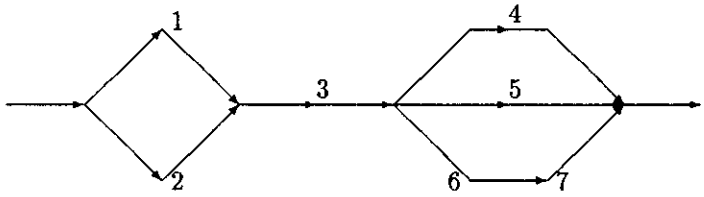

Figure 2: A Series-Parallel Graph

  These graphs use only the constructs in Figure 1. Dividing a problem is represented by fan-out nodes in the graph, while conquering gathers results into a set (with independent parallelism), a stream (with pipeline parallelism), and/or an aggregate (with fan-in parallelism). Thus, divide-and-conquer solutions of problems often directly correspond to these four kinds of parallelism.

- Database applications provide excellent opportunities for parallel processing. The set-oriented nature of the relational model makes exploitation of independent parallelism natural [34]. In fact, set operators such as the relational algebra operators can often be naturally expressed as divide-and-conquer computations, as we will show in sections 2 and 4.

These ideas raise hope for a parallel bulk data processing system that rests upon divide-and-conquer techniques. However, such a system must deal with several important technical issues to be viable.

A first problem is that the relational model offers no way to talk about order among data (e.g., sorted relations, or ordered tuples). Relational languages are therefore inadequate for specifying 'stream processing', in which ordered sequences of data are processed sequentially [23]. Pipeline parallelism is generally used, transparently to the user, in lower-level languages implementing relational algebra (e.g., PLERA [5], or PFAD [14]). However, higher-level relational interfaces do not permit streams to be exploited, preventing specification of stream computations and also pipeline parallelism.

A second problem is that parallel data processing requires effective data restructuring capabilities. Typically, a relational query (select-project-join expression) is translated into a low-level form of relational algebra with explicit (low-level) parallel constructs [5]. Data restructuring includes techniques like partitioning used to spread the computation of relational algebra operators among parallel processors [4]. Partitioning is

typically defined during the physical database design and then exploited by a compiler. Most of the time, a partitioned computation requires that processors exchange intermediate results in order to compute the final result.

In our view, data restructuring must be expressible by the programmer within a parallel database language. Specifying parallel computations over relations often requires specifying how data restructuring (fan-out parallelism) will be done and how distributed results will be collected (fan-in parallelism). This view is supported by recent results on data reduction for Datalog programs [37], in which rules are replaced by their per-processor specializations. These specialized rules include appropriate hash functions that capture partitioning information. This approach is very interesting in that it incurs no communication costs between processors. However, determining the appropriate hash functions to perform data reduction is still an open problem, known to be undecidable in some cases. It seems unlikely that database systems will be able to completely automate restructuring decisions.

Database models have been developed before that permit expression of both ordering among tuples and data restructuring. For example, the FAD language has operators that express various forms of fan-out and fan-in parallelism [10]. FAD is a strongly-typed set-oriented database language based on functional programming and relational algebra. It provides a fixed set of higher-order functions to aggregate functions, like the pump parametrized aggregate operator and the grouping operator. The pump operator applies a unary function to each element of a set, producing an intermediate set which is then 'reduced' to a single datum using a binary function that combines the intermediate set elements. Indeed, pump naturally expresses a special case of fan-out and fan-in parallelism. At the same time, the group operator permits set restructuring.

1.3 Goals of the Paper

Based on the observations above, our main goal is to develop a data model, called SVP, that supports both:

- ordered and unordered (stream and set) data representations;
- a formal semantics for divide-and-conquer computations on sets and streams to express independent (set) and pipeline (stream) parallelism.

This model is intended to serve as a formal foundation for defining parallel database languages in which parallelism is specified at a high-level.

The SVP data model has the following features:

- SVP values either are *collections* (a generalization of sets and streams), or are tuples of SVP values. Collections are represented as ordered binary tree structures. Intuitively, lists can represent streams, balanced trees can represent sets, and ordered binary trees can represent either.
- SVP allows restricted divide-and-conquer mappings on SVP values. In this paper these mappings are specified with recursive functional equations. They generalize other specification techniques, including restricted higher-order mappings like the reduction operator in APL [16] and the pump operator in FAD [10], automata [29, 30], and series-parallel computation graphs [24].
- Parallelism in the dividing and conquering is specified using both the structure of the data, and the structure of the divide-and-conquer mapping: dividing-parallelism is specified by the data, and conquering-parallelism is specified by the mapping. Partitioning can always be used to modify data structure, and thus affect dividing-parallelism.

Objectives of the model include elegance, expressiveness, and efficiency (computational efficiency in practice is captured by the model). We have been inspired by the work of Kahn [17], who stresses the importance of:

> ... a principle that has been so often fruitful in Computer Science and that is central to Scott's theory of computation: a *good* concept is one that is closed
>
> 1. under arbitrary composition
> 2. under recursion.

This principle is certainly in effect here.

The paper is organized as follows. Section 2 investigates the relationships between set and stream processing, and demonstrates with examples how divide-and-conquer mappings are important for data processing. Section 3 presents the SVP model and defines SVP values, types, and mappings. Section 4 then gives examples of SVP-mappings for expressing relational algebra operators, as well as restructuring, partitioning and grouping operators. In Section 5, we continue with a longer case study that illustrates the potential of SVP for real applications that are difficult to manage with today's RDBMS. Section 6 investigates formal properties of the SVP model. Section 7 then shows how SVP can be used as a foundation for defining a parallel database programming language. We make the case with a language that uses an imperative style. Finally, Section 8 summarizes the contributions of the SVP model, and Section 9 points out several promising directions for future work.

2 Set and Stream Processing

Let us clarify first what set processing and stream processing are, and then study how they might be integrated in a parallel processing model.

2.1 Sets and Streams

For the purposes of this paper, we will rely on similar formulations of sets and streams. Given a finite or countably infinite set of values D, we will write 2^D to denote the sets on D, and write $D^\bullet$ to denote the streams on D. These sets and streams technically can be either finite or countably infinite.

Sets use the following notation:

1. $\{\}$ is a set (the empty set);
2. $\{x\}$ is a set, for any value x;
3. Finite sets are written with set braces, as with: $\{1, 2, 3\}$.
4. The union $S_1 \cup S_2$ is a set, if S_1 and S_2 are sets. (In this paper, the symbol '$\cup$' always denotes a *disjoint* set union when describing the contents of a set. However, the $\cup$ operator for forming a set requires only that its operands be sets.)
5. The *cardinality* $\|S\|$ of any set S is the number of values in the set.

Streams analogously use the following notation:

1. $[\,]$ is a stream (the empty stream);
2. $[x]$ is a stream, for any value x;
3. Finite streams are written with square braces, as with: $[1, 2, 3]$.
4. The concatenation $S_1 \bullet S_2$ is a stream, if S_1 and S_2 are streams. (We use the symbol '$\bullet$' for stream concatenation ('*append*') in this paper.)
5. The *length* $|S|$ of any stream S is the number of values in the stream.

As usual, set union is associative and commutative, where stream concatenation is only associative.

Although streams are formalized here like strings, with a concatenation operator, they are accessible like *lists*. Specifically, every nonempty stream S satisfies

$$S \quad = \quad (h \cdot T)$$

where h is the *head* of S, and T is the *tail* of S. Here h will be a value, and T will be a stream. The constructor symbol '$\cdot$' ('cons') can be viewed as an operator that combines a value and a stream into a stream. The single-element stream $[x]$ is actually a shorthand for $(x \cdot [\,])$, and $[1, 2, 3]$ is a shorthand for $(1 \cdot 2 \cdot 3 \cdot [\,])$. All finite streams are terminated explicitly with $[\,]$.

One more bit of notation will be useful. We use parentheses to set off *tuples* (fixed-length sequences, vectors). Thus

$$(a, 1, b)$$

denotes a 3-tuple (tuple with 3 elements).

2.2 Set and Stream Mappings

Consider the following mappings, using the formalization of sets and streams given above. We would like to be able to formalize these mappings in our model.

The equations

$$\begin{array}{lcl} \mathit{count}(\{\}) & = & 0 \\ \mathit{count}(\{x\}) & = & 1 \\ \mathit{count}(S_1 \cup S_2) & = & \mathit{count}(S_1) + \mathit{count}(S_2) \end{array}$$

define a set mapping (in this case an aggregate) recursively. This definition reflects parallelism that can be obtained by computing cardinalities of subsets independently. For example, in the computation

$$\begin{array}{lcl} \mathit{count}(\{a,b,c\}) & = & \mathit{count}(\{a,b\}) + \mathit{count}(\{c\}) \\ & = & \mathit{count}(\{a\}) + \mathit{count}(\{b\}) + \mathit{count}(\{c\}) \\ & = & 1 + 1 + 1 \\ & = & 3 \end{array}$$

we have ultimately three independent parallel threads that are 'fanned-in' to an aggregate.

Consider now the stream mapping

$$\begin{array}{lcl} \mathit{diffs}([]) & = & [] \\ \mathit{diffs}(x \cdot []) & = & [] \\ \mathit{diffs}(x \cdot y \cdot S) & = & (y - x) \cdot \mathit{diffs}(y \cdot S) \end{array}$$

This yields a stream of the differences between adjacent elements in the input stream. For example:

$$\begin{array}{l} \mathit{diffs}(98 \cdot 99 \cdot 97 \cdot 97 \cdot 99 \cdot 96 \cdot []) \\ \quad = \; +1 \cdot \mathit{diffs}(99 \cdot 97 \cdot 97 \cdot 99 \cdot 96 \cdot []) \\ \quad = \; +1 \cdot -2 \cdot \mathit{diffs}(97 \cdot 97 \cdot 99 \cdot 96 \cdot []) \\ \quad = \; +1 \cdot -2 \cdot \; 0 \cdot \mathit{diffs}(97 \cdot 99 \cdot 96 \cdot []) \\ \quad = \; +1 \cdot -2 \cdot \; 0 \cdot +2 \cdot \mathit{diffs}(99 \cdot 96 \cdot []) \\ \quad = \; +1 \cdot -2 \cdot \; 0 \cdot +2 \cdot -3 \cdot \mathit{diffs}(96 \cdot []) \\ \quad = \; +1 \cdot -2 \cdot \; 0 \cdot +2 \cdot -3 \cdot []. \end{array}$$

This mapping implements a kind of 'automaton', or 'transducer', that scans the stream of values and translates it to a stream of pairwise differences. These transducer mappings are important in analyzing streams, but are (at best) quite challenging to implement with a set-oriented model.

2.3 Composition of Set and Stream Mappings

Functional mappings can be composed naturally. We consider a simple example that illustrates how composition of set and stream mappings allows us to answer arbitrary queries by composing a few elementary mappings.

Example: Areas of Convex Polygons

We are given a convex polygon as a stream of points in (x, y)-coordinate form that trace out the boundary of the polygon, and the problem is to compute the total area of the polygon.

This problem can be solved by triangulating the polygon, i.e., cutting the polygon into triangles, and computing the total area of the triangles. Specifically we can transform the stream of points of the polygon

$$[(x_1, y_1),\ (x_2, y_2),\ \cdots\ (x_n, y_n)],$$

into a set of triangles (triples of points)

$$\{\ ((x_1,y_1),(x_2,y_2),(x_3,y_3)),\ ((x_1,y_1),(x_3,y_3),(x_4,y_4)),\ \cdots\ ((x_1,y_1),(x_{n-1},y_{n-1}),(x_n,y_n))\ \},$$

and then compute the sum of the areas of the triangles.

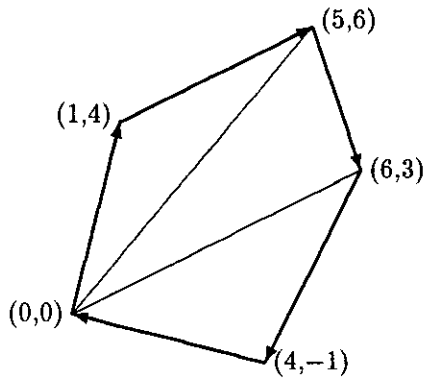

Figure 3: Triangulation of a Convex Polygon

For example the polygon given by the stream of points

$$[(0,0),(1,4),(5,6),(6,3),(4,-1)]$$

corresponds to the set of triangles

$$\{\ ((0,0),(1,4),(5,6)),\ \ ((0,0),(5,6),(6,3)),\ \ ((0,0),(6,3),(4,-1))\ \}$$

having respective areas[2]

$$\{\ 7.0,\ 10.5,\ 9.0\ \}$$

and a total area of 26.5. See Figure 3. This is expressible as

$$\begin{array}{lcl} \mathit{polygon} & = & [(0,0),(1,4),(5,6),(6,3),(4,-1)] \\ \mathit{total\_area} & = & \mathit{sum}(\ \mathit{areas}(\ \mathit{triangles}(\ \mathit{polygon}\ )\ )\ ) \end{array}$$

where we define *triangles* with

$$\begin{array}{lcl} \mathit{triangles}([\,]) & = & \{\} \\ \mathit{triangles}(p_0 \cdot [\,]) & = & \{\} \\ \mathit{triangles}(p_0 \cdot p_1 \cdot [\,]) & = & \{\} \\ \mathit{triangles}(p_0 \cdot p_1 \cdot p_2 \cdot S) & = & \{(p_0,p_1,p_2)\} \cup \mathit{triangles}(p_0 \cdot p_2 \cdot S). \end{array}$$

and the aggregate functions needed are:

$$\begin{array}{lcl} \mathit{sum}(\{\}) & = & 0 \\ \mathit{sum}(\{x\}) & = & x \\ \mathit{sum}(S_1 \cup S_2) & = & \mathit{sum}(S_1) + \mathit{sum}(S_2). \end{array}$$

$$\begin{array}{lcl} \mathit{areas}(\{\}) & = & \{\} \\ \mathit{areas}(\{(p_0,p_1,p_2)\}) & = & \{\mathit{area}(p_0,p_1,p_2)\} \\ \mathit{areas}(S_1 \cup S_2) & = & \mathit{areas}(S_1) \cup \mathit{areas}(S_2). \end{array}$$

[2]The Heron formula for the area of a triangle whose sides are known to have respective lengths a, b, c, is given by $\sqrt{s(s-a)(s-b)(s-c)}$ where we define $s = (a+b+c)/2$.

$$
\begin{aligned}
area(p_0, p_1, p_2) &= \sqrt{s(s-a)(s-b)(s-c)} \\
\text{where:} \quad & a = distance(p_0, p_1) \\
& b = distance(p_1, p_2) \\
& c = distance(p_2, p_0) \\
& s = (a+b+c)/2
\end{aligned}
$$

$$
distance((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.
$$

To show our approach here is flexible, let us change the problem now so that we are given a *set* of polygons, instead of a single polygon, and we wish to find the total area of all polygons. The program above will still solve this extended problem, if we simply modify it with the following assertions:

$$
\begin{aligned}
total\_area &= sum(\ areas(\ triangles^*(\ polygons\ )\ )\ ) \\
\\
triangles^*(\{\}) &= \{\} \\
triangles^*(\{P\}) &= triangles(P) \\
triangles^*(S_1 \cup S_2) &= triangles^*(S_1) \cup triangles^*(S_2).
\end{aligned}
$$

In terms of execution, this simply allows the computation of triangles to be carried out independently in parallel across the members of a set.

Finally, assume we are given a set of *surfaces*, where surfaces are (identifier, set of polygons)-pairs, and we wish to find the corresponding set of (identifier, total area)-pairs. We can accomplish this by defining

$$
\begin{aligned}
surface\_area(\{\}) &= \{\} \\
surface\_area(\{(Id,\ S)\}) &= \{(\ Id,\ sum(areas(triangles^*(S)))\ )\} \\
surface\_area(S_1 \cup S_2) &= surface\_area(S_1) \cup surface\_area(S_2).
\end{aligned}
$$

Parallelism is reflected directly in terms of data dependence. To find all (identifier,area)-pairs, we can:

1. convert all surfaces to sets of triangles in parallel;
2. compute areas of all triangles in parallel;
3. sum all the areas belonging to each surface in parallel.

The examples here hopefully make two points: First, a model based on composing mappings on sets and streams is sufficient to develop expressive database systems — significantly more expressive than standard DBMS. Although the example problems above are not easy to solve with standard DBMS, the structures involved (sets of streams, etc.) are easy to understand, and the queries are easy to state, and easy to state mathematically.

Second, the structure of the data (sets and streams) directly reflects parallelism in the data processing required. Both pipeline and independent parallelism are crucial in data processing, and these kinds of parallelism can be made evident by the stream or set structure of the data.

2.4 Perspective: Divide-and-Conquer Mappings

Our goal is to develop a formal data model that will support all of the mappings shown earlier. The challenge comes in developing a model that encourages optimization and extraction of parallelism and supports at least the set and stream mappings shown earlier.

The mappings above are all 'divide-and-conquer' mappings, of three kinds:

1. *Aggregates*
   Aggregates can be described as functions of sets with the format

$$
\begin{aligned}
f(\{\}) &= id \\
f(\{x\}) &= h(x) \\
f(S_1 \cup S_2) &= f(S_1)\ \theta\ f(S_2)
\end{aligned}
$$

where θ is an associative, commutative operator whose identity is id, and h is a function that yields values of the type taken by θ.

2. *Set Mappings*
   Set mappings have the divide-and-conquer form

$$\begin{array}{lcl} f(\{\}) & = & \{\} \\ f(\{x\}) & = & h(x) \\ f(S_1 \cup S_2) & = & f(S_1) \cup f(S_2) \end{array}$$

   where h is a set-valued function.

3. *Stream Transducers*
   Stream mappings like *diffs* and *triangles* are naturally characterized as 'automata' that incrementally translate their input. We will call this kind of mapping a *transducer*.

   In general form, we define a stream transducer f in terms of two function parameters, δ and h, and an iterative control structure F:

$$\begin{array}{lcl} f(S) & = & F(q_0, S) \\ F(q, []) & = & h(q, []) \\ F(q, x \cdot S) & = & h(q, x) \bullet F(\delta(q, x), S). \end{array}$$

   Here intuitively there is a set of 'states', q_0 is the 'initial state', δ is a 'state transition function' that maps a (state,input)-pair to a new state, and $h(q, x)$ is the output stream produced in state q with input x. So, in particular, $h(q, [])$ is the output stream produced in state q when no input remains. Thus F maps a (state,stream)-pair into a stream.

   We call f a stream transducer because its definition directly mirrors the definition of a finite state transducer — a finite automaton that produces output given its current input symbol and current state.

An obvious question facing us now is:

> *What is a useful generalization of aggregates, set mappings, and stream transducers, that can be applied successfully in parallel data processing?*

The SVP model described next offers one answer to this question.

3 The SVP Model

The goals of SVP require a model in which collections (both stream collections and (multi-)set collections) can be expressed, and mappings on these collections can be defined. For simplicity, and without loss of generality, we limit ourselves to a value-based model — i.e., objects are not handled by the model currently.

3.1 SVP Values

SVP models two kinds of values: *atomic values*, and *constructed values*. Constructed values represent complex structures, or nested values, and can be either *tuples* or *collections*. Tuples are typically heterogeneous structures with a small number of elements, while collections are typically homogeneous structures with a large number of elements.

Values are recursively defined as follows:

- Any atom is a SVP value.
- Any finite tuple $(v_1, \ldots, v_n)$ of SVP values $v_1, \ldots, v_n$ is a SVP value. A tuple with one atom is called a 1-tuple, a tuple with two atoms is called a 2-tuple, etc.
- Any collection is a SVP value.

In SVP, *collections* are recursively defined as follows:

- $\langle\rangle$ is the empty collection.
- $\langle v \rangle$ is a unit collection if v is a SVP value.
- $S_1 \diamond S_2$ is a collection if S_1 and S_2 are *nonempty* SVP collections. Collections are forbidden to properly contain the empty collection.

This definition allows SVP collections to model many structures of interest, including:

- *sets*
  The SVP-collection $(\langle 1\rangle \diamond \langle 2\rangle) \diamond (\langle 3\rangle \diamond \langle 4\rangle)$ represents the set {1,2,3,4} as a balanced binary tree.
- *streams*
  The SVP-collection $\langle 1\rangle \diamond (\langle 2\rangle \diamond (\langle 3\rangle \diamond (\langle 4\rangle \diamond [])))$ represents the stream [1,2,3,4] as a linear list-like structure.
- *sequences*
  A sequence is a right-linear tree, such as the SVP-collection $\langle 1\rangle \diamond (\langle 2\rangle \diamond (\langle 3\rangle \diamond \langle 4\rangle))$. A sequence is a non-[]-terminated stream. In many situations the stream terminator [] is not significant, and can be omitted.
- *groups*
  The SVP-collection $\langle (0, \langle a\rangle \diamond \langle e\rangle)\rangle \diamond \langle (2, \langle c\rangle \diamond \langle g\rangle)\rangle \diamond \langle (3, \langle d\rangle)\rangle$ is a collection of three tuples, where each tuple represents a group — a pair possessing an atomic value and a set.

3.2 SVP Types

Database systems concentrate on homogeneous collections of data. SVP does also, resting on a simple polymorphic type system that defines the following *value types*:

- **atom**
- **tuple**(T_1, ..., T_n) is a constructed value type, if each T_i is a value type.
- **collection**(T) is a homogeneous collection type, if T is a value type.

Thus the following are homogeneous collection types: **collection(atom)**, **collection(collection(atom))**, **collection(tuple(atom,collection(atom)))**, etc.

3.3 SVP Mappings

Data models typically specify how all permissible mappings can be constructed. We take a different approach. SVP imposes few restrictions on atomic value mappings — essentially any properly typed, semantically well-defined mapping on atomic values is permitted. However, SVP requires *all* collection mappings to be SVP-*transducers*. This class of mappings is powerful, and suited to bulk data processing on homogeneous collections. At the same time SVP-transducers are restrictive enough to permit optimization and extraction of parallelism.

3.3.1 Basic SVP Mappings

SVP explicitly provides the following basic mappings:

- *Constructors (and constructor-like operators)*
  - *tupling* $((\cdots))$
    If $x_1, \ldots, x_n$ are values of types $T_1, \ldots, T_n$, then $(x_1, \ldots, x_n)$ is of type **tuple**$(T_1, \ldots, T_n)$.

- *collection* ($\diamond$)
  If S_1 and S_2 are of type **collection**(T), $S_1 \diamond S_2$ is also. In the normal situation where both S_1 and S_2 are proper nonempty collections, it is natural to think of $\diamond$ as a constructor. However, when either of S_1 or S_2 is empty, the requirement stated above that no subcollection be the empty collection is enforced by making $\diamond$ be an *operator* that yields a collection.[3] Specifically,

$$\langle\rangle \diamond S \;=\; S \diamond \langle\rangle \;=\; S.$$

- *Deconstructors*
  SVP provides the following type-membership predicates for SVP values v:
  - atom(v) — whether v is an atomic value.
  - tuple(v) — whether v is a tuple.
  - collection(v) — whether v is a collection.
  - emptycollection(v) — whether v is $\langle\rangle$.
  - unitcollection(v) — whether v is $\langle x\rangle$ for some x.

  Furthermore, the following functions are provided:
  - unitcollectionvalue(S) — value x of a unit collection $S = \langle x\rangle$.
  - arity(t) — number n of elements in a tuple t.
  - $t[i]$ — tuple subscripting. If t is a tuple $(x_1, \ldots, x_n)$ of type **tuple**$(T_1, \ldots, T_n)$, and i is an integer between 1 and n, then $t[i]$ yields x_i, of type T_i.

  Note only tuple deconstructors are allowed to appear in user-defined mappings. General deconstructors are not provided for collections. *The only construct for iterating over collections is the SVP-transducer.*

SVP collections can be regarded as an abstract data type, whose only defined operations are the collection constructor, the collection primitives listed just above, and SVP-transducers introduced next.

3.3.2 SVP-Transducers

SVP-transducers are capable of implementing all the example mappings shown earlier. SVP-transducers specify mappings of collections as:

1. the restructuring of the input collection;
2. the mapping of the elements in the restructured collection;
3. the collecting of the resulting mapped input elements into an output.

Definition A mapping f on SVP collections is an *SVP-transducer* if it can be written in the following divide-and-conquer form:

$$\begin{aligned} f(S) &= F(Q_0, \rho(S)) \\ F(Q, \langle\rangle) &= id_\theta \\ F(Q, \langle x\rangle) &= h(Q, x) \\ F(Q, S_1 \diamond S_2) &= F(Q, \rho(S_1)) \;\; \theta \;\; F(\delta(Q, S_1), \rho(S_2)). \end{aligned}$$

[3] Why does SVP make this dual treatment of $\diamond$ as both a tree constructor and a tree-yielding operator? Briefly, SVP requires $\langle\rangle$ to behave like the empty set {} or like the empty string Λ — and unlike the stream terminator []. In reasoning about collections, $\langle\rangle$ defines a special case, but not the base case of an induction; unit collections $\langle x\rangle$ define the base case.

In the usual case of nonempty arguments, the tree constructor and tree-yielding operator are identical, so there is no confusion: $S_1 \diamond S_2$ yields the ordered binary tree with left subtree S_1 and right subtree S_2. In the other case, one argument is empty. Avoiding occurrences of the empty collection within a collection simplifies programming enormously: large collections cannot turn out to be empty, finding the first nonempty element in a collection becomes trivial, subtleties about the semantics of occurrences of $\langle\rangle$'s within a collection are neatly avoided, etc. Consequently, SVP makes it impossible to construct such a collection.

Here Q_0 is an arbitrary fixed value, ρ is either the identity mapping or an SVP-transducer, and h, θ, and δ are arbitrary SVP-mappings of two arguments. We have written θ as an infix operator.

We also permit f, F, and h to take additional arguments not shown explicitly here; in particular f can be a function of other parameters besides the collection S (including other collection parameters). Also, the Q argument can be omitted if it is not used by h or δ.

The mapping $\rho(S)$ typically performs *data restructuring* on the collection S. Common values for $\rho(S)$ include just S (the identity mapping, with no restructuring), and the operator partition(P,S), in which P is a predicate defining a splitting of S into two parts $S_1 \diamond S_2$, the first for which P yields the value **true**, and the latter the value **false** (assuming both are nonempty). Restructuring operators will be investigated later.

The operator θ must be of type $T \times T \rightarrow T$, for some SVP type T which must be declared. For example, the $\diamond$ collector here is restricted to work on operands of type **collection**, and produce a **collection**. This type also restricts the values produced by the h function. For example, when θ is '$\diamond$', h must produce a **collection**.

When θ is a complete binary operator with identity id_θ, we call θ a *collector*. The table below gives examples of collectors. Properties of collectors (*A*ssociativity, *C*ommutativity, *I*dempotency) can be exploited to obtain greater parallelism; parallel evaluation of associative collector expressions is sometimes called *parallel prefix computation* [19].

θ	*Result Type*	id_θ	*Properties*
$\diamond$	**collection**	$\langle\rangle$	—
$\diamond^R$	**collection**	$\langle\rangle$	—
$\star$	**collection**	$\langle\rangle$	A
+	**atom**	0	A,C
*	**atom**	1	A,C
max	**atom**	$-\infty$	A,C,I
min	**atom**	$+\infty$	A,C,I

Here $\diamond$ is the collection-forming operator, $\diamond^R$ is its reversal (so $x \diamond^R y = y \diamond x$), and $\star$ is the append operator for collections. Thus $\langle\rangle$ is the identity for $\star$, and when S and T are nonempty $S \star T$ is the collection consisting of S but with the rightmost leaf $\langle x \rangle$ of S replaced by $\langle x \rangle \diamond T$.

In the general SVP-transducer, θ is permitted to be an arbitrary operator, and id_θ an arbitrary value. For example, two standard combinators

$$x \text{ fst } y = x$$
$$x \text{ snd } y = y$$

are useful and will be employed later in this paper. However, we shall mainly deal in the rest of the paper with transducers in which θ is a collector.

The important restriction this form imposes is that the computation over the collection be performed by a divide-and-conquer traversal. Basically, SVP-transducers provide a ('large') control structure that directs a function on values ('small' data manipulation) to be applied as needed. This control structure can be viewed as a generalized scan over a collection, together with gathering of scan results, where both the scan and the final gathering may be performed using parallel techniques.

3.3.3 Definition of SVP Mappings

Further mappings can be built using the basic mappings and SVP-transducers defined above. However, SVP-mappings are restricted in the following ways:

- All mappings arguments are typed, and all mappings must be well-typed. In particular, constructors and deconstructors can be applied only to operands of the appropriate type.
- The only operators that can be applied directly to collections are constructors, the primitives collection, emptycollection, unitcollection, unitcollectionvalue, and SVP-transducers.
- An SVP-mapping can invoke at most a bounded number of SVP-transducers.

A concern here is that SVP-mappings can appear as parameters of SVP-transducers, so in particular SVP-transducers can be parameters to SVP-transducers. If an SVP-mapping used as a parameter performs SVP-transductions within a loop, recursion, or other iteration, transducers can be used (for example) to implement arbitrary iterative deconstructors, or even Turing machines. General query optimization of this sort of program is not feasible, and the semantics of transducers becomes considerably more complex. The third restriction limits SVP-mappings to produce only bounded networks of transductions on collections. Later we will see that a rigorous semantics can be given to networks of 'continuous' transductions.

3.4 Some Simple Examples

Most of the examples given earlier require only trivial changes of notation to be written as SVP transducers.

1. The set mapping f defined by

$$\begin{array}{lcl} \mathit{areas}(\{\}) & = & \{\} \\ \mathit{areas}(\{t\}) & = & \mathit{area}(t) \\ \mathit{areas}(S_1 \cup S_2) & = & \mathit{areas}(S_1) \cup \mathit{areas}(S_2) \end{array}$$

   immediately becomes the SVP transducer

$$\begin{array}{lcl} \mathsf{areas}(\langle\rangle) & = & \langle\rangle \\ \mathsf{areas}(\langle t\rangle) & = & \mathsf{area}(t) \\ \mathsf{areas}(S_1 \diamond S_2) & = & \mathsf{areas}(S_1) \diamond \mathsf{areas}(S_2). \end{array}$$

2. The *diffs* transducer shown earlier

$$\begin{array}{lcl} \mathit{diffs}([\,]) & = & [\,] \\ \mathit{diffs}(x \cdot [\,]) & = & [\,] \\ \mathit{diffs}(x \cdot y \cdot S) & = & ((y - x) \cdot \mathit{diffs}(y \cdot S)) \end{array}$$

   can be implemented as an SVP transducer as follows, assuming that the input collection is a sequence (a right-linear tree):

$$\begin{array}{lcl} \mathsf{diffs}(S) & = & \mathsf{diffs1}(\langle\rangle, S) \\ \\ \mathsf{diffs1}(Q, \langle\rangle) & = & \langle\rangle \\ \mathsf{diffs1}(Q, \langle x\rangle) & = & \textbf{if } Q = \langle\rangle \textbf{ then } \langle\rangle \textbf{ else } \langle x - \mathsf{unitcollectionvalue}(Q)\rangle \\ \mathsf{diffs1}(Q, S_1 \diamond S_2) & = & \mathsf{diffs1}(Q, S_1) \diamond \mathsf{diffs1}(S_1, S_2). \end{array}$$

   This transducer does not handle streams terminated with [ ]. Such streams can be accomodated merely by modifying the **if** expression to yield $\langle\rangle$ or $\langle[\,]\rangle$ when $x = [\,]$.

3. The *diffs* transducer requires its input to be a sequence. We can define **sequence** transducer that transforms an arbitrary collection to a sequence.

$$\begin{array}{lcl} \mathsf{sequence}(S) & = & \mathsf{sequence1}(\mathsf{first\_rest}(S)) \\ \mathsf{sequence1}(\langle\rangle) & = & \langle\rangle \\ \mathsf{sequence1}(\langle x\rangle) & = & \langle x\rangle \\ \mathsf{sequence1}(S_1 \diamond S_2) & = & \mathsf{sequence1}(\mathsf{first\_rest}(S_1)) \diamond \mathsf{sequence1}(\mathsf{first\_rest}(S_2)) \end{array}$$

   This works by repeatedly partitioning a collection S into a collection $H \diamond T$, where H is the unit collection giving the first of S, and T is the rest of S. The first-rest partitioning can be defined as follows:

$$\begin{array}{lcl} \mathsf{first\_rest}(S) & = & \mathsf{first}(S) \diamond \mathsf{rest}(S) \\ \\ \mathsf{first}(\langle\rangle) & = & \langle\rangle \\ \mathsf{first}(\langle x\rangle) & = & \langle x\rangle \\ \mathsf{first}(S_1 \diamond S_2) & = & \mathsf{first}(S_1). \\ \\ \mathsf{rest}(S) & = & \mathsf{rest1}(\textbf{true}, S) \\ \mathsf{rest1}(\mathsf{initial}, \langle\rangle) & = & \langle\rangle \\ \mathsf{rest1}(\mathsf{initial}, \langle x\rangle) & = & \textbf{if } \mathsf{initial} \textbf{ then } \langle\rangle \textbf{ else } \langle x\rangle \\ \mathsf{rest1}(\mathsf{initial}, S_1 \diamond S_2) & = & \mathsf{rest1}(\mathsf{initial}, S_1) \diamond \mathsf{rest1}(\textbf{false}, S_2). \end{array}$$

This is not the most efficient way to flatten a tree, but it illustrates various features of SVP. Note that first uses as θ the operator fst that yields its left argument (i.e., x fst $y = x$), and we have simply omitted it and the right argument in the third equation for first. This operator is not a collector. The definition of rest uses a boolean state variable initial, indicating that the first element has been already extracted. The δ function here is constant, always taking the value **true**.

A more compact definition of sequence can be obtained by recursive appending of sequences:

$$\begin{aligned} \mathsf{sequence}(\langle\rangle) &= \langle\rangle \\ \mathsf{sequence}(\langle x\rangle) &= \langle x\rangle \\ \mathsf{sequence}(S_1 \diamond S_2) &= \mathsf{sequence}(S_1) \star \mathsf{sequence}(S_2). \end{aligned}$$

Here '$\star$' is the collection 'append' operator.

3.5 Properties of the SVP Model

The SVP model was designed to address the goals given at the outset. This section has provided a variety of examples using SVP that will help motivate its being the way it is. To help clarify, however, below are some perceptions about the model that helped shape its current form.

1. The definition of SVP transducers is a generalization of earlier definitions of set mappings, aggregates, and stream transducers. Furthermore, it is a modest generalization, covering essentials only. On the other hand, it is a relatively complete generalization with a formal parallel semantics.

2. SVP collections are ordered binary trees because this is sufficient to let them represent sets and streams, physical data organization (such as sort order and grouping), how recursive problem division should work, and also lets them be used as an index (search) structure if that is desired.

   Supporting partitioning and sorting is essential for real operation. Many DB transductions work only on sorted inputs. Join algorithms rely heavily on partitioning. Restructuring is a key member of any ensemble of operators on collections.

   The actual topology of the tree is important in that it determines parallel problem division (and conquering). We can use the tree structure to guide how processes are spawned. Spawning stops on subtrees of a certain granularity. Linear trees can spawn many small processes; also they take more time to spawn processes. Balanced trees spawn parallel processes more effectively.

   SVP collections can be searched. When the leaves of an collection tree are in sort order and the tree is balanced, we can search it in $O(\log n)$ time in much the same way that we can search B-trees, although the constant factor is larger. (Adding internal node keys to SVP collections would make them indistinguishable from indices, but that is an extension we will not pursue in this paper.)

3. When θ is an associative operator, the expression

   $$x_1 \ \theta \ x_2 \ \theta \ \cdots \ \theta \ x_n$$

   gives the same result regardless of the way it is parenthesized, i.e., regardless of the topology of the expression tree. The result is affected only by the ordering of the x_i. Thus *associative operators are naturally stream mappings.* Furthermore, when θ is associative and commutative, the result is the same regardless of the ordering of the x_i. Thus *associative, commutative operators are naturally set mappings.*

4. The definition of SVP transducers leads to a very nice theory, based on the idea of *structure preservation.* Divide-and-conquer techniques work only when the data can be divided in a way that represents some underlying 'structure'. SVP collections allow us to model various kinds of structure important in data processing, including sort ordering and physical data partitioning. It is possible to generalize the classic work of Kahn for continuous functions on sequences [17] to work for continuous functions on collections. The basic idea is that prefix-continuous functions on sequences[4] are exactly those functions

[4] Prefix-continuous functions on sequences are functions that are monotone with respect to the sequence prefix ordering, so giving the function more sequence input cannot result in the function's producing less sequence output, and also cannot delay indefinitely before producing an output. Fixed-point results for continuous functions lead to a rigorous semantics for networks of SVP-transducers, even for cyclic networks.

that yield pipeline parallelism. Thus 'stream-continuity' gives pipeline parallelism, and 'set-continuity' gives independent parallelism.

5. SVP is a *model.* It is not intended as a full database system and query language, but rather the sketch of a larger, full-featured system. It permits many practical extensions, including *n-ary trees* (not just binary) for representing collections that permit n-ary problem division, *trees with labeled internal nodes* that permit direct representation of indices, and *if-then-else constructs* that permit early termination of a scan over a collection. Earlier versions of the SVP model permitted these extensions explicitly, but the result was a more complicated model. Also, these extensions tend to encourage transducers that are more 'automata-like', with less parallelism. The current model is simple, and encourages a transducer style with more parallelism.

4 The Algebra of SVP-Transducers

SVP transducers can be formulated as an algebra, somewhat like the relational algebra, but in which the operators are parameterized by functions. To demonstrate the power of this algebra and of SVP, we show how it can express important data processing primitives.

4.1 Three Basic Transducers

SVP offers essentially three functionals:

- collect
  Aggregates over collections are implementable with collect:

$$\begin{array}{lcl} \mathsf{collect}(\theta,\ id,\ \langle\rangle) & = & id \\ \mathsf{collect}(\theta,\ id,\ \langle x\rangle) & = & x \\ \mathsf{collect}(\theta,\ id,\ S_1 \diamond S_2) & = & \mathsf{collect}(\theta,\ id,\ S_1)\ \theta\ \mathsf{collect}(\theta,\ id,\ S_2). \end{array}$$

- transduce
  General transductions on collections can be performed with transduce:

$$\begin{array}{lcl} \mathsf{transduce}(h,\ \delta,\ Q,\ \langle\rangle) & = & \langle\rangle \\ \mathsf{transduce}(h,\ \delta,\ Q,\ \langle x\rangle) & = & h(Q, x) \\ \mathsf{transduce}(h,\ \delta,\ Q,\ S_1 \diamond S_2) & = & \mathsf{transduce}(h,\ \delta,\ Q,\ S_1) \diamond \mathsf{transduce}(h,\ \delta,\ \delta(Q, S_1)\ ,\ S_2). \end{array}$$

- restructure
  Tree restructuring repeatedly applies a function to all levels of a tree. It is defined by:

$$\begin{array}{lcl} \mathsf{restructure}(\rho,\ S) & = & \mathsf{restructure1}(\rho,\ \rho(S)) \\ \\ \mathsf{restructure1}(\rho,\ \langle\rangle) & = & \langle\rangle \\ \mathsf{restructure1}(\rho,\ \langle x\rangle) & = & \langle x\rangle \\ \mathsf{restructure1}(\rho,\ S_1 \diamond S_2) & = & \mathsf{restructure1}(\rho,\ \rho(S_1)) \diamond \mathsf{restructure1}(\rho,\ \rho(S_2)) \end{array}$$

Most SVP transducers are definable in terms of collect, transduce, and restructure. Note that whenever $\delta(Q, S) = \delta'(Q, \mathsf{restructure}(\rho, S))$ for all Q and S, the SVP transducer f defined by the recursion

$$\begin{array}{lcl} f(S) & = & F(\ Q_0, \rho(S)) \\ F(\ Q,\ \langle\rangle\ ) & = & id_\theta \\ F(\ Q,\ \langle x\rangle\ ) & = & h(Q, x) \\ F(\ Q,\ S_1 \diamond S_2\ ) & = & F(\ Q,\ \rho(S_1)\ )\ \ \theta\ \ F(\ \delta(Q, S_1),\ \rho(S_2)\ ). \end{array}$$

is equivalent to

$$f(S) \quad = \quad \mathsf{collect}(\theta,\ id,\ \ \mathsf{transduce}(h,\ \delta',\ Q_0,\ \ \mathsf{restructure}(\rho,\ S))).$$

In the following sections we show that many useful operators, including all relational algebra operators can be implemented as SVP transducers, and in fact they are implementable with these three operators.

4.2 Restructuring, Partitioning and Grouping

Often it is useful to transform of one structure to another. Reorganization can be done both with restructuring and collecting in a variety of surprising ways. Good examples were given earlier when we defined sequence, a transducer that converts its argument to a sequence. Notice that

$$\mathsf{sequence}(S) \quad = \quad \mathsf{restructure}(\,\mathsf{first\_rest},\, S\,)$$

where first_rest is the transducer that partitions its collection into its first and remaining elements. Also we essentially showed that

$$\mathsf{sequence}(S) \quad = \quad \mathsf{collect}(\,\star,\, \langle\rangle,\, S\,)$$

— i.e., collections can be flattened by recursive appending.

The restructure functional is useful for *top-down* reorganization of collections. For example, if ρ splits a tree into two trees of equal cardinality, then $\mathsf{restructure}(\rho, S)$ produces a balanced version of S. Also, if ρ partitions a tree into two subtrees by comparing with a median-estimate key value, then $\mathsf{restructure}(\rho, S)$ sorts a tree S by that key.

We can also restructure any collection into a balanced collection (balanced tree), by repeated splitting into halves of equal size:

$$\begin{array}{lcl}
\mathsf{balance}(S) & = & \mathsf{restructure}(\mathsf{split}, S) \\
\mathsf{split}(S) & = & \mathsf{partition_tree}(\,\mathsf{halves}(1, \mathsf{count}(S), \mathsf{sequence}(S))\,) \\
\mathsf{halves}(i, n, \langle\rangle) & = & (\,\langle\rangle,\, \langle\rangle\,) \\
\mathsf{halves}(i, n, \langle x\rangle) & = & \textbf{if } i \le n/2 \textbf{ then } (\,\langle x\rangle,\, \langle\rangle\,) \textbf{ else } (\,\langle\rangle,\, \langle x\rangle\,) \\
\mathsf{halves}(i, n, S_1 \diamond S_2) & = & \mathsf{halves}(i, n, S_1)\ \ \mathsf{combine_partitions}\ \ \mathsf{halves}(i+1, n, S_2).
\end{array}$$

Here we need several operators on partitions:

$$\begin{array}{rcl}
\mathsf{partition_tree}(\,(S_1, S_2)\,) & = & S_1 \diamond S_2 \\
(P_1, P_2)\ \ \mathsf{combine_partitions}\ \ (Q_1, Q_2) & = & (P_1 \diamond Q_1,\ P_2 \diamond Q_2).
\end{array}$$

Note combine_partitions is a binary operator with identity $(\langle\rangle, \langle\rangle)$, and is thus a collector.

Collecting is useful for *bottom-up* restructuring of collections. Partitioning mappings can also be developed with collecting. The mapping

$$\mathsf{partition}(P, S) \quad = \quad \mathsf{partition\_tree}(\,\mathsf{collect}(\,\mathsf{combine\_partitions},\ (\langle\rangle, \langle\rangle),\ \mathsf{partitionify}(P, S))\,)$$

performs partitioning by using collect to split a collection S into two subcollections (S_1, S_2) according to a predicate, and using partition_tree to recombine these into a collection. Here we need the definition:

$$\begin{array}{lcl}
\mathsf{partitionify}(P, \langle\rangle) & = & \langle\rangle \\
\mathsf{partitionify}(P, \langle x\rangle) & = & \textbf{if } P(x) \textbf{ then } (\,\langle x\rangle,\, \langle\rangle\,) \textbf{ else } (\,\langle\rangle,\, \langle x\rangle\,). \\
\mathsf{partitionify}(P, S_1 \diamond S_2) & = & \mathsf{partitionify}(P, S_1) \diamond \mathsf{partitionify}(P, S_2).
\end{array}$$

As another useful example, the 'mirror image' of a collection can be obtained by collecting with the reflection operator $\diamond^R$:

$$\mathsf{mirror}(S) \quad = \quad \mathsf{collect}(\,\diamond^R,\, \langle\rangle,\, S)$$

and for example $\mathsf{mirror}(\,\langle 1\rangle \diamond (\langle 2\rangle \diamond \langle 3\rangle)\,) \;=\; (\langle 3\rangle \diamond \langle 2\rangle) \diamond \langle 1\rangle$.

Grouping can also be expressed as collecting. The grouping operation takes a set S and a characteristic function h (say a hash function or a key function) as input, and produces as output a set of 2-tuples (k, S_i), $1 \le i \le p$, where k is the value obtained by applying h to any member of the set S_i, and such that S is partitioned into the S_i subsets.

Indeed, there are various possible algorithms for grouping that range from sequential ones to truly parallel ones. The interesting point is that SVP-mappings capture the specification of parallelism in these algorithms.

The algorithm proceeds in two main steps. First, for each member x in the input collection, a tuple $(h(x), \langle x\rangle)$ is built. This is done in parallel for all members x in the input set:

$$
\begin{array}{lcl}
\mathsf{group}(h, S) & = & \mathsf{gather}(\mathsf{hashify}(h, S)) \\
\\
\mathsf{hashify}(h, \langle\rangle) & = & \langle\rangle \\
\mathsf{hashify}(h, \langle x \rangle) & = & \langle (h(x), \langle x \rangle) \rangle \\
\mathsf{hashify}(h, S_1 \diamond S_2) & = & \mathsf{hashify}(h, S_1) \diamond \mathsf{hashify}(h, S_2).
\end{array}
$$

Note that the input collection S is converted into a collection of pairs by hashify. The second step proceeds successively with each element in the collection and inserts it into the current output. Initially, the output is empty.

$$
\begin{array}{lcl}
\mathsf{gather}(S) & = & \mathsf{gather1}(\langle\rangle, \mathsf{sequence}(S)) \\
\\
\mathsf{gather1}(Q, \langle\rangle) & = & \langle\rangle \\
\mathsf{gather1}(Q, \langle x \rangle) & = & \mathsf{insert}(Q, \langle x \rangle) \\
\mathsf{gather1}(Q, S_1 \diamond S_2) & = & \mathsf{gather1}(\ \mathsf{insert}(Q, S_1),\ S_2).
\end{array}
$$

This transducer uses as θ the operator **snd** that yields its second argument; that is, $x \,\mathbf{snd}\, y \;=\; y$. This operator is not a collector, since it has no right identity.

We now provide a parallel insert: x is considered for insertion into each current 'bucket' in parallel. At most one insertion can be successful. In the other cases, an empty set is returned. We can implement this with parallel fan-in by defining an associative, commutative operator

$$
(b_1, S_1)\ \mathsf{combine\_groups}\ (b_2, S_2) \;=\; (b_1 \vee b_2,\ S_1 \diamond S_2)
$$

where $\vee$ is Boolean **or**, with identity element $(\mathbf{false}, \langle\rangle)$. We use the boolean value to represent success of insertion within a subset:

$$
\begin{array}{lcl}
\mathsf{insert}(S_0, \langle\rangle) & = & \langle\rangle \\
\mathsf{insert}(S_0, \langle b \rangle) & = & \begin{cases} S_0 \diamond \langle b \rangle & \text{if } \mathsf{insert1}(b, S_0) = (\mathbf{false}, S_0) \\ S_1 & \text{if } \mathsf{insert1}(b, S_0) = (\mathbf{true}, S_1) \end{cases} \\
\mathsf{insert}(S_0, S_1 \diamond S_2) & = & \mathsf{insert}(\ \mathsf{insert}(S_0, S_1),\ S_2\ ) \\
\\
\mathsf{insert1}(Q, \langle\rangle) & = & (\mathbf{false}, \langle\rangle) \\
\mathsf{insert1}((k, \langle x \rangle), \langle (k', S) \rangle) & = & \begin{cases} (\mathbf{true}, \langle (k, S \diamond \langle x \rangle) \rangle) & \text{if } k = k' \\ (\mathbf{false}, \langle (k', S) \rangle) & \text{otherwise} \end{cases} \\
\mathsf{insert1}(Q, S_1 \diamond S_2) & = & \mathsf{insert1}(Q, S_1)\ \ \mathsf{combine\_groups}\ \ \mathsf{insert1}(Q, S_2).
\end{array}
$$

An example will illustrate the operation of insert1:

$$
\begin{array}{l}
\mathsf{insert1}(\ \ (2, \langle c \rangle),\ \ \langle\ (0, \langle a \rangle \diamond \langle e \rangle) \rangle \diamond \langle\ (2, \langle g \rangle)\ \rangle\ \ ) \\
\quad = \ \mathsf{insert1}(\ (2, \langle c \rangle),\ \langle (0, \langle a \rangle \diamond \langle e \rangle) \rangle\ )\ \ \mathsf{combine\_groups}\ \ \mathsf{insert1}(\ (2, \langle c \rangle),\ \langle (2, \langle g \rangle) \rangle\ ) \\
\quad = \ (\mathbf{false},\ \langle (0, \langle a \rangle \diamond \langle e \rangle) \rangle)\ \ \mathsf{combine\_groups}\ \ (\mathbf{true},\ \langle (2, \langle g \rangle \diamond \langle c \rangle) \rangle) \\
\quad = \ (\mathbf{true},\ \ \langle\ (0, \langle a \rangle \diamond \langle e \rangle)\ \rangle\ \diamond\ \langle\ (2, \langle g \rangle \diamond \langle c \rangle)\ \rangle).
\end{array}
$$

Here **true** indicates that the insertion of $(2, \langle c \rangle)$ resulted in the update of a group element.

The group transducer could have been expressed as a composition like

$$
\mathsf{group}(S) \;=\; \mathsf{collect}(\mathsf{snd}, \langle\rangle,\ \mathsf{transduce}(\mathsf{insert}, \mathsf{insert}, \langle\rangle,\ \mathsf{transduce}(\mathsf{hash}, \_, \_,\ S)))),
$$

but this is incomprehensible. The great compactness of notation provided by functionals is often also dangerously cryptic, so SVP encourages the use of the divide-and-conquer notation, which is comparatively much easier to understand.

4.3 Functional Query Languages

We can relate SVP transducers to the functionals commonly used in functional query languages. There has been a good deal of work on functional languages in database systems — see for example the references of [35].

The standard foldleft and map operators work on lists as follows:

$$\begin{array}{lcl} \mathsf{map}(f,\,[]) & = & [] \\ \mathsf{map}(f,\,[x_1,\,\cdots,\,x_n]) & = & [f(x_1),\,\cdots,\,f(x_n)] \\ \mathsf{foldleft}(\theta,\,id,\,[]) & = & id \\ \mathsf{foldleft}(\theta,\,id,\,[x_1,\,\cdots,\,x_n]) & = & (\,\cdots\,((id\;\theta\;x_1)\;\theta\;x_2)\;\cdots\;\theta\;x_n). \end{array}$$

These are special cases of SVP transducers. For example, if S is a non-[]-terminated sequence:

$$\mathsf{foldleft}(\theta,\,id,\,S) \quad = \quad \textbf{if}\ \mathsf{emptycollection}(S)\ \textbf{then}\ id\ \textbf{else}\ id\;\theta\;\mathsf{collect}(\theta,\,id,\,S).$$

When S is not such a sequence, we would first restructure it so that it is.

Recently Wadler has shown how a popular stream/list query construct (known as 'list comprehensions') is equivalent to a slight extension of a monad system containing the operators unit, map, and join, where join flattens a list of lists into a corresponding list [35]. The beautiful monad operators are also special cases of SVP operators, but restricted to lists instead of trees. For example, $\mathsf{unit}(x) = \langle x \rangle$, and join is the specialization of collect in which θ is list concatenation. Where functional programming work is normally concerned with lists, and database programming languages have been concerned with sets, SVP is concerned with both.

4.4 Aggregation

The SVP-transducer apparently accomplishes most of what one could want from an aggregate. In FAD [10], the parameterized aggregate operator $\mathsf{pump}(h,\theta,id_\theta,S)$ is defined by

$$\mathsf{pump}(h,\theta,id_\theta,S) \quad = \quad \begin{cases} id_\theta & \text{if } S = \{\} \\ h(x_1)\;\theta\;\ldots\;\theta\;h(x_n) & \text{if } S = \{x_1,\ldots,x_n\} \end{cases}$$

where θ is an associative, commutative binary operator, with identity id_θ. It is definable as an SVP transducer:

$$\begin{array}{lcl} \mathsf{pump}(h,\theta,id_\theta,\langle\rangle) & = & id_\theta \\ \mathsf{pump}(h,\theta,id_\theta,\langle x\rangle) & = & h(x) \\ \mathsf{pump}(h,\theta,id_\theta,S_1 \diamond S_2) & = & \mathsf{pump}(h,\theta,id_\theta,S_1)\;\;\theta\;\;\mathsf{pump}(h,\theta,id_\theta,S_2). \end{array}$$

The list1 operator in [7] is similar.

The APL reduction operator [16] allows non-associative, non-commutative operators. In particular, if θ is a binary operator and $S = (x_1, x_2, \ldots, x_n)$ is a vector (tuple), the APL reduction of S by θ is

$$\theta/S \quad = \quad ((\cdots\;(x_1\;\;\theta\;\;x_2)\;\;\theta\;\;\cdots)\;\;\theta\;\;x_n).$$

This is an aggregation that reflects the ordering of the input. It can also be written as a SVP-transducer on collections, provided the collections are in in left-linear form: $((\cdots\;(x_1 \diamond x_2) \diamond \cdots) \diamond x_n)$. In this case the transducer is obvious:

$$\begin{array}{lcl} \mathsf{APLreduction}(\theta,\langle\rangle) & = & \langle\rangle \\ \mathsf{APLreduction}(\theta,\langle x\rangle) & = & x \\ \mathsf{APLreduction}(\theta,S_1 \diamond S_2) & = & \mathsf{APLreduction}(\theta,S_1)\;\;\theta\;\;\mathsf{APLreduction}(\theta,S_2). \end{array}$$

We can furthermore restructure any collection to left-linear form with the SVP-mapping

$$\mathsf{left\_linear\_sequence}(S) \quad = \quad \mathsf{mirror}(\mathsf{sequence}(\mathsf{mirror}(S))).$$

Note APLreduction is actually more expressive than fold, in that the operand θ need not be an associative operator.

4.5 Joins

Join algorithms involve complex combinations of fan-out, combination, and fan-in operations on sets of tuples. It may appear difficult to come up with a set of primitives that express different join algorithms effectively and efficiently. Surprisingly, important n-ary operations like joins can be implemented with transducers! In fact, interesting join algorithms can be developed.

Let us define a general join algorithm. One general specification for joins would be something like:

$$\mathsf{combine}(R,S) \;=\; \{\; \mathsf{RESULT}(r,s) \mid \mathsf{TEST}(r,s) \wedge r \in R \wedge s \in S \;\}.$$

Most join algorithms use a simple definition for RESULT (e.g., tuple concatenation) and TEST (e.g., testing equality of key values). The join partitions the cross product $R \times S$ into equivalence classes. The kind of equivalence classes used are determined by the join algorithm, and can be used to introduce 'groups' over which the join is to be done — for example grouping the tuples with equal key values.

With this in mind, we can produce a generalized join mapping, in which R has 'groups' R_i, S has 'corresponding groups' S_i, and the join is made over groups:

$$\begin{aligned}\mathsf{combine}(R,S) \;=\; \{\; \mathsf{RESULT}(r,s) \mid\quad & \mathsf{TEST}(r,s) \wedge r \textit{ in } R_i \wedge s \textit{ in } S_i \\ & \wedge R_i = \mathsf{MAP1}(P) \wedge P \textit{ in } R \\ & \wedge S_i = \mathsf{MAP2}(P,S) \;\}.\end{aligned}$$

Here P is a 'part' of R (such as a (key value, group)-pair), and R_i and S_i are the actual groups the join is to be done over. MAP1 and MAP2 are arbitrary functions that convert groups to a suitable representation. Groups give what is needed for joins to deal with multiple occurrences of join keys (or even of tuples); they capture join equivalence classes.

This generalized join mapping could be implemented in pseudocode as follows:

```
combine(R,S) = T where
{
T = ∅;
for P in R                                   [a part of R, typically a group]
      {
      R_i = MAP1(P);                         [a suitable mapping of a group in R]
      S_i = MAP2(P,S);                       [mapping of the corresponding group in S]
      for r in R_i                           [a member of the R_i group]
            for s in S_i                     [a member of the S_i group]
                  if (TEST(r,s))
                    then T = T ∪ RESULT(r,s)
      }
}
```

This provides a 'macro'-like control structure for joins with several function parameters: MAP1, MAP2, TEST, RESULT.

This definition implements various join algorithms according to the structure chosen for R and S, and the choice of parameters.

- If R and S are collections of tuples, MAP1 maps a tuple P in R to the collection $R_i = \langle P \rangle$, and MAP2 simply takes S_i to be the entire collection S, then we obtain the *nested loops* algorithm.
- If R and S are groups (collections of collections) of elements with the same join key value, R_i is the group of R with join key i, S_i is the group of S with join key i, we obtain a general indexed join algorithm. Specifically, if the groups are collections of elements with the same hash key value, then the groups represent hash buckets, and we have the *parallel hash join* algorithm. The algorithm is parallel in that all groups can be joined in parallel.
- The parameterized set map operator filter$(h,S_1,\ldots,S_m)$, in FAD [10] yields the value of h applied to each tuple in the cross product of the sets $S_1,\ldots,S_m$ (for $m > 0$):

$$\mathsf{filter}(h,S_1,\ldots,S_m) \;=\; \{\; h(x_1,\ldots,x_m) \mid x_1 \in S_1,\; \ldots,\; x_m \in S_m \;\}.$$

We can implement filter as a cascade of $m-1$ combines implementing nested loops joins, where the final combine in the cascade applies h as its RESULT mapping.

The generalized join operator described above can be implemented with cascaded transductions:

$$
\begin{array}{lcl}
\mathsf{combine}(R,S) & = & \mathsf{combine1}(S,R) \\
\\
\mathsf{combine1}(S,\langle\rangle) & = & \langle\rangle \\
\mathsf{combine1}(S,\langle P\rangle) & = & \mathsf{combine2}(\mathsf{MAP2}(P,S),\mathsf{MAP1}(P)) \\
\mathsf{combine1}(S,P_1 \diamond P_2) & = & \mathsf{combine1}(S,P_1) \diamond \mathsf{combine1}(S,P_2) \\
\\
\mathsf{combine2}(S_i,\langle\rangle) & = & \langle\rangle \\
\mathsf{combine2}(S_i,\langle r\rangle) & = & \mathsf{combine3}(\langle r\rangle,S_i) \\
\mathsf{combine2}(S_i,R_{i1} \diamond R_{i2}) & = & \mathsf{combine2}(S_i,R_{i1}) \diamond \mathsf{combine2}(S_i,R_{i2}) \\
\\
\mathsf{combine3}(\langle r\rangle,\langle\rangle) & = & \langle\rangle \\
\mathsf{combine3}(\langle r\rangle,\langle s\rangle) & = & \textbf{if } \mathsf{TEST}(r,s) \textbf{ then } \mathsf{RESULT}(r,s) \textbf{ else } \langle\rangle \\
\mathsf{combine3}(\langle r\rangle,S_{i1} \diamond S_{i2}) & = & \mathsf{combine3}(\langle r\rangle,S_{i1}) \diamond \mathsf{combine3}(\langle r\rangle,S_{i2}).
\end{array}
$$

4.6 Merge Scans

Merge scans, and general n-way merges of multiple streams, naturally seem to require simultaneous recursion on multiple arguments. Surprisingly, perhaps, they can be implemented as a single SVP recursion. Merging of two streams with a single SVP-transducer is accomplished by using one of the streams as the initial state, and incrementally consuming this state while simultaneously consuming the other stream.

For example, let us define an transducer for producing the *set union* of two sorted, []-terminated streams (sequences). Since every sequence $S_1 \diamond S_2$ is actually $\langle y\rangle \diamond S_2$ for some y, we use a shorthand notation to define transducers for set union as follows:

$$
\begin{array}{lcl}
\mathsf{sorted\_union}(R,\langle[]\rangle) & = & R \\
\mathsf{sorted\_union}(R,\langle y\rangle \diamond S) & = & \mathsf{union\_output}(y,R) \diamond \mathsf{sorted\_union}(\mathsf{union\_state}(y,R),S). \\
\\
\mathsf{union\_output}(y,\langle[]\rangle) & = & \langle y\rangle \\
\mathsf{union\_output}(y,\langle x\rangle \diamond R) & = & \begin{cases} \langle y\rangle & \text{if } x = [] \\ \langle y\rangle & \text{if } x > y \\ \langle\rangle & \text{if } x = y \\ \langle x\rangle \diamond \mathsf{union\_output}(y,R) & \text{if } x < y. \end{cases} \\
\mathsf{union\_state}(y,\langle[]\rangle) & = & \langle[]\rangle \\
\mathsf{union\_state}(y,\langle x\rangle \diamond R) & = & \begin{cases} \langle[]\rangle & \text{if } x = [] \\ \langle x\rangle \diamond R & \text{if } x > y \\ \langle x\rangle \diamond R & \text{if } x = y \\ \mathsf{union\_state}(y,R) & \text{if } x < y. \end{cases}
\end{array}
$$

Although they do what is needed, union_state and union_output technically violate our restrictions on SVP-mappings. Such mappings must be properly implemented as transducers that scan through the state stream R, outputting the prefix of R of elements less than y, omitting the elements in R equal to y, and retaining the suffix of R of elements greater than y. This is easily accomplished. For example, if we define

$$
\begin{array}{lcl}
\mathsf{union\_state\_increment}(y,x) & = & \textbf{if } x = [] \textbf{ then } \langle[]\rangle \\
 & & \textbf{else if } x > y \textbf{ then } \langle x\rangle \\
 & & \textbf{else if } x = y \textbf{ then } \langle x\rangle \\
 & & \textbf{else } /^*\ x < y\ ^*/\ \langle\rangle
\end{array}
$$

then we can write:

$$
\begin{array}{lcl}
\mathsf{union\_state}(y,\langle[]\rangle) & = & \langle[]\rangle \\
\mathsf{union\_state}(y,\langle x\rangle) & = & \mathsf{union\_state\_increment}(y,x) \\
\mathsf{union\_state}(y,Q_1 \diamond Q_2) & = & \mathsf{union\_state}(y,Q_1) \diamond \mathsf{union\_state}(y,Q_2)
\end{array}
$$

in proper SVP form.

5 Case Study: Bond Investment Analysis

In this section, we describe a realistic application that shows potential for a model like SVP.

In [25], Rozen and Shasha describe BondDB, a decision support system developed to support investment banks in the buying and selling of bonds. The system was built using a relational DBMS (Oracle), storing basic information on about 10,000 different bonds, daily bond quotes, and the status of bond portfolios. BondDB was developed to help investors improve profitability and reduce risk of their portfolios, as well as perform various kinds of forecasting about expected values of investments over a variety of possible future scenarios. Investment strategies for bonds have become very complex, and decisions often require a great deal of data analysis. A very readable and comprehensive introduction to the subject can be found in [11].

5.1 Bond Attributes

As an example of what one finds when considering various investments in the United States, the information for a bond abstracted from Moody's *Public Utility Manual* could look as follows:[5]

Issuer	Southern California Gas
Type of bond	Open Mortgage–Outstg.
Coupon	$8\frac{1}{4}\%$
Face value	$100.00
Moody's Rating	A1
Maturity Date	November 1, 1996
Payment frequency	semiannual
Payment schedule	every May 1 and November 1:

91/05/01	$4.12
91/11/01	$4.13
92/05/01	$4.12
92/11/01	$4.13
93/05/01	$4.12
93/11/01	$4.13
94/05/01	$4.12
94/11/01	$4.13
95/05/01	$4.12
95/11/01	$4.13
96/05/01	$4.12
96/11/01	$104.13

Call schedule — with thirty days' notice:

90/11/01 to 91/10/31	$104.43
91/11/01 to 92/10/31	$103.32
92/11/01 to 93/10/31	$102.21
93/11/01 to 94/10/31	$101.11
94/11/01 to 95/10/31	$100.00
95/11/01 to 96/10/31	$100.00

Although there are various kinds of bonds (open mortgages, municipal general obligation bonds, treasury notes, etc.), most share the following attributes:

- *Issuer* — a corporation, local government, or (a department or agency within) the federal government;
- *Face value* — the value to be repaid investors when the bond matures;
- *Coupon* — the interest payment (in percent) that is issued to holders of the bond at fixed dates, typically semiannually;

[5] The information about this bond differs slightly from the real values in order to simplify the presentation here.

- *Maturity date* — the date on which the bond is 'retired' (terminated), and holders are paid the bond's face value;
- *Ratings* — risk classifications, such as offered by Moody's or Standard & Poor's;
- *Payment frequency* — number of times per year an interest payment is made, typically 2 (semiannual payments);
- *Payment schedule* — the dates and amounts upon which interest payments are made over the lifetime of the bond;
- *Call schedule* — the dates upon which the bond issues can *call* the bond, i.e., can redeem the bond prior to its maturity date. The call schedule allows the issuer to cancel the bond if its coupon gets out of line with current interest rates, or if the issuer wishes to retire the debt for some other reason.

Note that bonds are traded on stock exchanges, and their prices can vary from day to day. The price need not match the face value.

Representing this information about bonds in a relational database is challenging because attributes like the call schedule would most naturally be represented by a non-first-normal-form structure. Furthermore, the call schedule records are ordered by time, requiring some kind of ordered representation. BondDB was obliged to store the call schedule in a relation by itself, rather than replicate all the basic bond information. However this solution is imperfect, as it forces queries about bonds to be split into several subqueries, and does not support querying of the call schedule as a stream. These problems were viewed as a truly 'ugly' consequence of using relational systems [25].

These problems do not occur with SVP. The basic bond information can be represented as a collection of SVP tuples, whose entries are mostly atomic values, with the exception of the payment schedule and call schedule, which can be represented as streams. Being able to represent both sets and streams is important in this application.

5.2 Bond Yield Analysis

In BondDB, some queries were too complex to express using SQL. Let us consider a simple example involving bond yields. Bond yields give a measure of the 'interest rate' offered by a bond. A bond's yield can differ from its coupon because the bond price can vary.

The *yield_to_maturity* of a bond, given a purchase price and purchase date, is the interest rate for which compound interest on the purchase amount (at maturity), plus the purchase price itself, equals the future value of the bond's payments (at maturity and using the same interest rate). Let $yield(P, C, R, m, n)$ denote the yield of a bond with:

- purchase price P,
- annual coupon interest C (the bond's coupon multiplied by its face value),
- final redemption value (maturity face value, or call price) R,
- yearly payment frequency m,

that is held for n years.[6] Then *yield_to_maturity* is the special case in which we hold the bond until its maturity date.

For example, suppose we buy Southern California Gas on 90/11/01 for P = \$92.25 and hold the bond for 6 years (to maturity). The yield obtained here ultimately turns out to be

$$yield(92.25, 8.25, 100.0, 2, 6) \;=\; 10.00\%.$$

[6] This yield $y = yield(P, C, R, m, n)$ is defined by the equation

$$P \;=\; \sum_{i=1}^{mn} \frac{C}{m}\,(1+\frac{y}{m})^{-i} \;+\; R\,(1+\frac{y}{m})^{-mn}$$

— the interest rate for which price matches return. Typically values for y are found iteratively, or with tables; see [11]. We have used a Newton's method solver here.

If for example the price were P = \$95, our yield would reduce to 9.36%; if P = \$100, the yield drops to 8.25% (the coupon); and if P = \$105, the yield falls below the coupon to 7.21%.

Unfortunately, we are not guaranteed the bond will last to maturity: the issuer may call the bond. In this case our yield will be determined by the date of the call and the call price, rather than the maturity date. With the Southern California Gas bond, for example, if we buy at P = \$105.00, the possibilities are:

call occurs after payment on 91/05/01:	*yield*(105.00, 8.25, 104.43, 2, 0.5) = 6.77%
call occurs after payment on 91/11/01:	*yield*(105.00, 8.25, 103.32, 2, 1.0) = 6.28%
call occurs after payment on 92/05/01:	*yield*(105.00, 8.25, 103.32, 2, 1.5) = 6.83%
call occurs after payment on 92/11/01:	*yield*(105.00, 8.25, 102.21, 2, 2.0) = 6.59%
call occurs after payment on 93/05/01:	*yield*(105.00, 8.25, 102.21, 2, 2.5) = 6.86%
call occurs after payment on 93/11/01:	*yield*(105.00, 8.25, 101.11, 2, 3.0) = 6.72%
call occurs after payment on 94/05/01:	*yield*(105.00, 8.25, 101.11, 2, 3.5) = 6.90%
call occurs after payment on 94/11/01:	*yield*(105.00, 8.25, 100.00, 2, 4.0) = 6.80%
call occurs after payment on 95/05/01:	*yield*(105.00, 8.25, 100.00, 2, 4.5) = 6.94%
call occurs after payment on 95/11/01:	*yield*(105.00, 8.25, 100.00, 2, 5.0) = 7.04%
call occurs after payment on 96/05/01:	*yield*(105.00, 8.25, 100.00, 2, 5.5) = 7.13%
maturity after payment on 91/11/01:	*yield*(105.00, 8.25, 100.00, 2, 6.0) = 7.21%

and the worst case yield turns out to be 6.28%, if the call occurs right after we receive our second payment on 91/11/01. Recall that the *yield_to_maturity* in this case is 7.21%. In this example, the call would be made on 91/11/01 with a call price of \$103.32. (In fact the call need not always occur on a coupon payment date, but we will ignore this complication for the moment.)

On the other hand, if we buy on 90/11/01 at P = \$92.25, the following yields are possible:

call occurs after payment on 91/05/01:	*yield*(92.25, 8.25, 104.43, 2, 0.5) = 35.35%
call occurs after payment on 91/11/01:	*yield*(92.25, 8.25, 103.32, 2, 1.0) = 20.36%
call occurs after payment on 92/05/01:	*yield*(92.25, 8.25, 103.32, 2, 1.5) = 16.32%
call occurs after payment on 92/11/01:	*yield*(92.25, 8.25, 102.21, 2, 2.0) = 13.81%
call occurs after payment on 93/05/01:	*yield*(92.25, 8.25, 102.21, 2, 2.5) = 12.75%
call occurs after payment on 93/11/01:	*yield*(92.25, 8.25, 101.11, 2, 3.0) = 11.71%
call occurs after payment on 94/05/01:	*yield*(92.25, 8.25, 101.11, 2, 3.5) = 11.26%
call occurs after payment on 94/11/01:	*yield*(92.25, 8.25, 100.00, 2, 4.0) = 10.68%
call occurs after payment on 95/05/01:	*yield*(92.25, 8.25, 100.00, 2, 4.5) = 10.45%
call occurs after payment on 95/11/01:	*yield*(92.25, 8.25, 100.00, 2, 5.0) = 10.27%
call occurs after payment on 96/05/01:	*yield*(92.25, 8.25, 100.00, 2, 5.5) = 10.12%
maturity after payment on 91/11/01:	*yield*(92.25, 8.25, 100.00, 2, 6.0) = 10.00%

The worst yield will be the minimum of the collection, which is 10.00%.

With this example in mind, let us define the *yield_to_call* of the bond precisely like *yield_to_maturity*, but use the call price instead of the maturity face value, and the call date instead of the maturity date. We need to consider all call dates after every interest payment. In the worst case, we will get the minimum of the *yield_to_maturity* and the least *yield_to_call*.

We can write definitions for *yield_to_worst* as follows then:

$$\begin{aligned} &yield\_to\_worst(bond, price, date) \\ &\quad = \; min(\langle\; yield\_to\_maturity(bond, price, date),\;\; yield\_to\_call(bond, price, date)\;\rangle) \end{aligned}$$

$$\begin{aligned} &yield\_to\_maturity(bond, price, date) \\ &\quad = \; yield(price, coupon\_interest(bond), face\_value(bond), ipf(bond), years(maturity\_date(bond) - date)) \end{aligned}$$

$$\begin{aligned} &yield\_to\_call(bond, price, date) \\ &\quad = \; \min_{pdate > date} yield(price, coupon\_interest(bond), cprice(pdate, bond), ipf(bond), years(pdate - date)) \end{aligned}$$

Here *pdate* ranges over the payment dates in *payment_schedule*(*bond*), *cprice*(*pdate*, *bond*) is the call price after the payment on *pdate* (determined by the call schedule), *ipf*(*bond*) is the interest payment frequency of the bond, $years(date_1 - date_0)$ is the (possibly fractional) number of years between $date_0$ and $date_1$, and *ipf*(*bond*) is the interest payment frequency (number per year) for *bond*.

5.3 Realized Compound Yield and Beyond

Our definition for *yield_to_worst* computes minima of yields to the ends of different time intervals, ignoring reinvestment of the coupon payments and the redemption value of the bond after it is redeemed. Basically, both *yield_to_maturity* and *yield_to_call* assume that the reinvestment rate (the rate of interest during the lifetime of the bond) is always equal to the yield. This is a not necessarily a realistic assumption, but it is very commonly made.

A more accurate estimate of yield can be obtained by considering reinvestment under various scenarios about future interest rates. Let us define the 'future value'

$$FV(P, r, m, n) \quad = \quad P\ (1 + \frac{r}{m})^{mn}$$

of an amount P invested today at annual interest rate r and compounded m times per year for n years. Now define

$$horizon\_value(bond, horizon\_date)$$
$$= \sum_{(date, amount)\ \in\ payment\_schedule(bond)} FV(amount, reinvestment\_rate(date), ipf(bond), years(horizon\_date - date)).$$

Finally, the *realized compound yield* of the bond is the interest rate y such that

$$horizon\_value(bond, horizon\_date) \quad = \quad FV(price, y, ipf(bond), years(horizon\_date - date)).$$

It is a more realistic measure of yield than *yield_to_worst*.

Note *horizon_value* is an aggregate transducer that takes a *reinvestment_rate*, which could be either a constant or a scenario (time series) of possible future rates. In the latter case, the query is not just a simple sum, but requires 'merging' of streams: the interest rate scenario must be merged with the payment schedule, to obtain a sequence of payments and rate changes ordered by dates.

Things can get much more complex. Investment managers may wish to select different *horizon_dates*, and consider the various possible call scenarios for this date. Some very complex transducers can be produced here, which compute horizon values over various simulated histories. They would be quite hard to express in SQL.

Furthermore, several simplifying assumptions made throughout the discussion above may not apply. For example, one can buy bonds in the *middle* of coupon periods, and not just on payment dates; and calls can occur *at any time*, and not just on call dates. These modifications require the use of involved 'interpolation' formulas for accrued interest that follow industry conventions for the kind of bond in question. This complication seems to defy expression with SQL, as it requires fairly complex numeric computations and 'merging' of streams of events again. However, it would be tractable with SVP.

5.4 Conclusions on Case Study

BondDB is a cutting-edge application of database technology. Current relational database systems were found to be not quite up to the task, and produced 'ugly' partial solutions for some problems. SVP seems to avoid some of the weaknesses of the relational model, as a result of the SVP model's ability to represent constructed values within tuples, and to express interesting mappings on these collections.

Beyond the examples shown here, BondDB performs many other functions, including temporal queries such as computing moving averages and asking whether a bond is within 10% of its best price over the last 30 days. These again seem well-suited for SVP. For example, moving averages are naturally computed as transducers. In general, the m-th moving average of a stream S is a stream whose i-th element is the average of elements i, $(i+1)$, ..., $(i+m-1)$ in S. The following definition finds the m-th moving average of a stream

by computing averages of 'windows' (substreams) of size m:

$$
\begin{array}{lcl}
\mathsf{moving\_average}(m, S) & = & \mathsf{windowAvgs}(m, \langle\rangle, \mathsf{sequence}(S)) \\
 & & \\
\mathsf{windowAvgs}(m, W, \langle\rangle) & = & \langle\rangle \\
\mathsf{windowAvgs}(m, W, \langle x \rangle) & = & \textbf{if } \mathit{count}(W) < m - 1 \textbf{ then } \langle\rangle \textbf{ else } \langle \mathsf{average}(\mathsf{window}(m, W, \langle x \rangle)) \rangle \\
\mathsf{windowAvgs}(m, W, S_1 \diamond S_2) & = & \mathsf{windowAvgs}(m, W, S_1) \diamond \mathsf{windowAvgs}(m, \mathsf{window}(m, W, S_1), S_2). \\
 & & \\
\mathsf{window}(m, W, \langle x \rangle) & = & \mathsf{all\_but\_first}((\mathsf{count}(W) + 1 - m), W \diamond \langle x \rangle) \\
\mathsf{all\_but\_first}(n, \langle\rangle) & = & \langle\rangle \\
\mathsf{all\_but\_first}(n, \langle x \rangle) & = & \textbf{if } n > 0 \textbf{ then } \langle\rangle \textbf{ else } \langle x \rangle \\
\mathsf{all\_but\_first}(n, S_1 \diamond S_2) & = & \mathsf{all\_but\_first}(n, S_1) \diamond \mathsf{all\_but\_first}(n - \mathsf{count}(S_1), S_2)
\end{array}
$$

Generalizing, it seems that once database systems are required to answer queries about 'scenarios', we are led to the situation where the database must perform limited kinds of simulation. Event-based simulation seems to require stream processing capabilities.

Rozen and Shasha conclude that a database would have been closer to 'ideal' for supporting the BondDB application if it had supported data abstraction and abstract data types, allowed programming constructs to be stored in the database, and provided better locking and performance tuning capabilities. This experience suggests important future extensions for SVP.

6 Theory of SVP-Transducers

Recall that SVP is intended as a data model that supports parallel processing of 'collections', which generalize sets and streams. Our main goals in developing a theory for the model are that:

- the model should support *independent parallelism* on sets
- the model should support *pipeline parallelism* on streams
- the model should be fairly *'complete'* in the sense that most (reasonable) collection functions can be expressed
- the model should provide a rigorous semantics, to permit reasoning (both formal and informal) about parallel computations.

In this section we study SVP-transducers and show how they address these goals.

Our approach is to develop a theory of structure-preserving mappings, which include order-preserving (monotone) mappings as a special case. This theory clarifies the kinds of divide-and-conquer mappings that SVP-transducers implement, and characterizes their properties. We then show how the classic results of Kahn for networks of sequence transducers can be generalized for networks of SVP transducers, while demonstrating that certain transducers offer either independent or pipeline parallelism.

6.1 Structure-Preserving Mappings

The divide-and-conquer mappings given earlier have regular, recursive definitions that directly reflect the structure of their inputs. These mappings all produce a new output 'view' of their inputs, which presents the input in a new way.

Basically, the mappings preserve some aspect of the original data, and are in this precise sense 'abstractions' of the data. Since data processing amounts to abstraction of principle from the details of data, these kinds of mappings are naturally important to us. Several general classes of these mappings are easily identified:

- collection maps (collection structure-isomorphisms)
- homomorphisms (collection structure-preserving mappings)

- continuous functions (collection order- and limit-preserving functions)
- monotone functions (collection order-preserving functions)

These classes are of increasing generality: so every collection map is a homomorphism, every homomorphism is continuous with respect to the subcollection ordering, etc. The interesting thing is that all these classes, which we define now, can be characterized structurally.

6.1.1 Homomorphisms and Collection Maps

Homomorphisms and collection maps are the simplest kind of structure-preserving mappings. Each element of the input collection is mapped to one or more elements of the output collection.

A *collection homomorphism* f is a mapping in $D^\diamond \rightarrow D^\diamond$ such that there is a function h in $D \rightarrow D^\diamond$ for which

$$\begin{array}{lcl} f(\langle\rangle) & = & \langle\rangle \\ f(\langle x \rangle) & = & h(x) \\ f(S_1 \diamond S_2) & = & f(S_1) \diamond f(S_2). \end{array}$$

f is called a *collection map* if f is a collection homomorphism with the property that $h(x)$ is a unit collection for all x in D. Collection maps preserve the topological structure of collections. We make this (initially possibly confusing) use of the word 'map' here following common practice in functional programming systems, which define functions like map, mapcar, etc.

6.1.2 Monotone and Continuous Functions

Monotone functions do not necessarily preserve the topological structure of their input, but always preserve the 'order' of their input.

In what follows we let $D^\diamond$ be the set of all finite or countably infinite collections over the finite or countably infinite set D. If we define $\langle D \rangle = \{ \langle x \rangle \mid x \in D \}$ then $D^\diamond$ satisfies the recursive domain specification [27]

$$D^\diamond = \langle D \rangle + D^\diamond \times D^\diamond.$$

Also, we need to introduce $\perp$, the undetermined collection. Extending the definition above for the lifting $D_\perp = D \cup \{\perp\}$, let $\langle D \rangle_\perp = \langle D \rangle \cup \{\perp\}$, and $D_\perp^\diamond$ be the set of collections containing $\perp$ as a subcollection zero or more times. In other words,

$$D_\perp^\diamond = \langle D \rangle_\perp + D_\perp^\diamond \times D_\perp^\diamond.$$

Henceforth, we will also extend any function defined on $D^\diamond$ to a strict function on $D_\perp^\diamond$ by asserting that $f(\perp) = \perp$. We make the simplifying assumption that all functions are strict.

A *collection monotone function* f is a mapping in $D^\diamond \rightarrow D^\diamond$ such that there is a partial ordering $\sqsubseteq$ on $D^\diamond$ for which

$$S_1 \sqsubseteq S_2 \quad \text{implies} \quad f(S_1) \sqsubseteq f(S_2).$$

The following are partial orderings on collections that are important for us in deriving semantics for SVP-transducers.

- *collection prefix ordering*
  $S \preceq_\diamond T$ if $S = T$, or $S = \perp$, or $S = (S_1 \diamond S_2)$ and $T = (T_1 \diamond T_2)$ and recursively $S_1 \preceq_\diamond T_1$ and $S_2 \preceq_\diamond T_2$.
- *collection sequence prefix ordering*
  $S \preceq_{\diamond seq} T$ if either $S = T$ or $S \prec_{\diamond seq} T$.
  $S \prec_{\diamond seq} T$ if $S = \perp$ and $T \neq \perp$, or $S = (S_1 \diamond S_2)$ and $T = (T_1 \diamond T_2)$ where either $S_1 \prec_{\diamond seq} T_1$ and $S_2 \preceq_\diamond T_2$, or $S_1 = T_1$, neither S_1 nor T_1 contain $\perp$, and $S_2 \prec_{\diamond seq} T_2$.

Collection prefix ordering $S \preceq_\diamond T$ insists that S be a prefix of T, growing outward from the root. Collection sequence prefix ordering $S \preceq_{\diamond seq} T$ further insists that either $S = T$ or the leftmost $\perp$ in S be non-$\perp$ in T.

With each of the orderings above, $D_\perp^\diamond$ is a complete partial order (cpo). This is shown by demonstrating that ascending chains have limits. An ascending chain with respect to $\sqsubseteq$ is a finite or countably infinite sequence

$$C = [S_1, S_2, S_3, \ldots]$$

of collections $S_1,\ S_2,\ S_3,\ \ldots$ in $D^\diamond$ such that

$$S_1 \sqsubseteq S_2 \sqsubseteq S_3 \sqsubseteq \cdots$$

For each ordering above, every ascending chain C over $D^\diamond_\perp$ has a limit (least upper bound)

$$\lim C \ = \ \sqcup_{i\in\omega} S_i.$$

Probably the main subtlety here is to order limits in a standard way such that

$$\sqcup S_i \ \sqsubseteq \ \sqcup T_j \quad \textbf{iff} \quad \forall m\ \exists n\ \ S_m \ \sqsubseteq \ T_n.$$

Since the ordering $\sqsubseteq$ used is the key determiner of monotonicity, we will henceforth write $\sqsubseteq$-*monotonicity* when we need to make the ordering explicit. More generally, for two domains D_1 and D_2 with orderings $\sqsubseteq_1$ and $\sqsubseteq_2$, a function $f:\ D_1 \ \rightarrow\ D_2$ is monotone if

$$X \ \sqsubseteq_1 \ Y \quad \text{implies} \quad f(X) \ \sqsubseteq_2 \ f(Y).$$

In this case, we say f is $(\sqsubseteq_1, \sqsubseteq_2)$-*monotone*. For example, the function collection_size: $D^\diamond \rightarrow \mathbf{N}$ from collections to natural numbers is monotone with respect to the collection prefix ordering $\preceq_\diamond$ on its input and the numeric ordering $\leq$ on its output. So, collection_size is $(\preceq_\diamond, \leq)$-monotone.

Continuous functions are monotone functions with one additional property: they preserve limits. Specifically, f is called $\sqsubseteq$-*continuous* if f is $\sqsubseteq$-monotone, and for every chain $C = S_1,\ S_2,\ \ldots$ that is ascending with respect to $\sqsubseteq$, we have

$$f(\lim C) \quad = \quad \lim f(C)$$

where $\lim f(C) \ = \ \sqcup_{i\in\omega} f(S_i)$. Also f is called $(\sqsubseteq_1, \sqsubseteq_2)$-*continuous* if f is $(\sqsubseteq_1, \sqsubseteq_2)$-monotone and f preserves limits accordingly. For example, collection_size is $(\preceq_\diamond, \leq)$-continuous.

Functional composition preserves monotonicity and continuity. When f is $(\sqsubseteq_1, \sqsubseteq_2)$-continuous and g is $(\sqsubseteq_2, \sqsubseteq_3)$-continuous, then the composition of f with g is $(\sqsubseteq_1, \sqsubseteq_3)$-continuous.

The relational algebra operators are monotone on their arguments. When expressed as SVP transducers, selection, projection, cross product, set union, set intersection, and set difference are all collection homomorphisms, monotone functions, and continuous functions! Moreover, they are monotone in multiple senses:

- Operators of the relational algebra are monotone with respect to (multi)set inclusion on each of their arguments. An operator f is monotone with respect to set inclusion in an argument if whenever $S_1 \subseteq S_2$, then $f(S_1) \subseteq f(S_2)$. (There is one exception: the set difference operator is monotone in its first argument and *antimonotone* in its second argument, i.e., $S_1 \subseteq S_2$ implies $R - S_1 \supseteq R - S_2$.)

- Relational algebra operators are monotone with respect to ordering of tuples in significant ways. Define the value partition ordering $\leq^\diamond$ on collections by

  $$S_1 \ \leq^\diamond \ S_2 \ \text{ if } x \leq y \text{ for all } x \in S_1, \text{ and all } y \in S_2,$$

  where $\leq$ is any total ordering on values. Then many relational operators are monotone with respect to $\leq^\diamond$. If $f(S)$ is a binary relational operator with argument S (for example, $f(S) = S \cap T$, for a given relation T), then $S_1 \leq^\diamond S_2$ implies $f(S_1) \leq^\diamond f(S_2)$.

These monotonicity properties of the relational algebra operators are very important, being exploited by virtually every database query optimizer.

6.1.3 Structure-Preserving Mappings

The structure of trees that implement collections can capture important kinds of information: individual sort ordering, subcollection lexicographic sort ordering, segmenting (partitioning by predicate), balance (partitioning by size; parallelism), sequence (linear organization; sequentiality). This physical structure can be crucial both for efficient and for correct operation. We have shown also that collections can be coerced to take these particular structures with the SVP restructure transducer. More generally, now, what do we mean by 'structure' of a collection?

Let us define a *structure* of collections to be a predicate on collections. For example, the predicate sorted($\leq$,S) is satisfied by collections S whose leaf sequences are in increasing sorted order with respect to $\leq$. Also, the predicates partitioned(h,S), balanced(S), sequenced(S), etc. identify useful structures.

We say that a function f on collections is *structure preserving* if there is a structure predicate P such that whenever S satisfies P, then $f(S)$ satisfies P. More specifically we say f is *P-structure preserving*. A function f is *(P_1,P_2)-structure preserving* if whenever S satisfies P_1, then $f(S)$ satisfies P_2. When f is a function with multiple arguments, then it may be structure preserving in a particular argument, etc.

This formalism allows us to capture the properties of the nice classes of mappings introduced in the previous sections, particularly the monotone mappings. The preceding sections showed that collection homomorphisms preserve structure — for example, relational algebra operators preserve sort order, set inclusion, and value partitioning with respect to the orderings . Value partitioning is extremely important in bulk data processing. To see this, write $S_1 \diamond_{\leq} S_2$ for a collection that is partitioned into two nonempty subcollections S_1 and S_2 such that $S_1 \leq^{\circ} S_2$. Then the following equations hold:

$$\begin{array}{lcl} \mathit{select}_B(S_1 \diamond_{\leq} S_2) & = & \mathit{select}_B(S_1) \diamond_{\leq} \mathit{select}_B(S_2) \\ \mathit{project}_A(S_1 \diamond_{\leq} S_2) & = & \mathit{project}_A(S_1) \diamond_{\leq} \mathit{project}_A(S_2) \\ R \cap (S_1 \diamond_{\leq} S_2) & = & (R \cap S_1) \diamond_{\leq} (R \cap S_2). \end{array}$$

In other words, the operators *select*, *project*, and $\cap$ preserve partitioning by $\leq$. The union and difference operators preserve partitioning when both their arguments are partitioned identically:

$$\begin{array}{lcl} (R_1 \diamond_{\leq} R_2) \cup (S_1 \diamond_{\leq} S_2) & = & (R_1 \cup S_1) \diamond_{\leq} (R_2 \cup S_2) \\ (R_1 \diamond_{\leq} R_2) - (S_1 \diamond_{\leq} S_2) & = & (R_1 - S_1) \diamond_{\leq} (R_2 - S_2). \end{array}$$

If a mapping on individual items is monotone, then the corresponding collection homomorphism is structure preserving, and its computation permits division (and conquering).

Necessary and sufficient conditions for structure preservation are straightforward for SVP transducers that perform no restructuring or collecting. These transducers preserve sort ordering when the h function is appropriately monotone, preserve balance or sequence when the h function maps unit collections to unit collections, preserve partitions with respect to the predicate p when the h function preserves p, and so forth. Often conditions can be derived when restructuring and collecting are present. SVP transducers encourage this kind of reasoning about the structure of collections and its preservation under mappings.

6.2 Sequence Transducers and Pipeline Parallelism

The material above describes how to analyze single SVP transducers. Complex mappings on multiple collections will in general require a network of these transducers. Assigning a semantics to networks of processes can be very complex in general, and the fact that processes work here on tree-structured data (the transducers here read and write trees) complicates the picture further.

Fortunately, we can generalize on one key result in parallel processing: Kahn's result for networks of processes [17]. Kahn showed that networks of continuous functions permit pipeline parallelism, and provide an elegant semantics of stream-oriented computation. We first summarize these results, and then show how to generalize upon them for SVP.

6.2.1 Kahn's Networks of Functions

We provide a brief summary of the classic paper [17].

D^{ω} is the set of all finite or countably infinite *sequences* of items drawn from a set D, also including the empty sequence Λ. We order members of D^{ω} by prefix inclusion $\preceq_{seq}$,[7] so that

$$S \preceq_{seq} T \quad \text{iff } S \text{ is a prefix of } T, \text{ for all } S,T \in D^{\omega}.$$

The least member of D^{ω} is therefore Λ. Note Λ is *not* an end-marker, unlike [] for lists. The set D^{ω} with the ordering $\preceq_{seq}$ is a complete partial order, since every ascending chain has a limit.

We are particularly interested in functions f in $D^{\omega} \rightarrow D^{\omega}$ that are *$\preceq_{seq}$-continuous*. Such a function is monotone with respect to the sequence prefix ordering, so giving the function more input cannot result

[7] In [17], the symbol $\subseteq$ is used instead of $\preceq_{seq}$, with the definition: '$X \subseteq Y$ iff X is an initial segment of Y'.

in the function's producing less output. Also, such a function cannot wait indefinitely before producing an output. Kahn shows several useful functions on D^ω are $\preceq_{seq}$-continuous, notably F (first), R (remainder), and A (a cons-like approximation of append), where $A(X, Y)$ yields the leftmost element of X followed by Y.

A *process* is represented as a collection of $\preceq_{seq}$-continuous functions on sequences. More precisely, a process is a program that reads items from the sequence on a specified input port (*wait*s on that port), and writes sequences to a specified output port (*send*s on that port), Reads on a port always block until something is written there. Each output of the process can be defined by a $\preceq_{seq}$-continuous function of the inputs. Finally, a *network of processes* is a collection of (continuous) processes corresponding to the nodes of a static communication graph, whose edges specify their inputs and outputs.

Kahn points out that $\preceq_{seq}$-continuity has the following important consequences:

1. A finite network of $\preceq_{seq}$-continuous processes implements a $\preceq_{seq}$-continuous function; it follows that a least fixed point semantics for the network are then straightforward to obtain, and amount to the ensemble of histories corresponding to the edges in the communication graph.

2. Parallel programs consisting of a network of $\preceq_{seq}$-continuous processes can be proven to yield the same result regardless of the control scheme, as long as the control scheme is fair. (Furthermore, if the control scheme is unfair the worst that can happen is that the network will produce less output than it would with a fair scheduler.)

Thus, continuity provides an elegant formal semantics for stream computations, and *also* provides a limited guarantee of pipeline parallelism. It is remarkable that Kahn's concept of continuity goes far towards formalizing what we want in a function that provides pipeline parallelism, as well as matching the properties we want from database mappings.

6.2.2 Networks of SVP Transducers

Our results follow on the foundation set by Kahn. We replace the cpo of sequences D^ω by the cpo of collections $D^\diamond$, ordered by either collection prefix $\preceq_\diamond$ or collection sequence prefix $\preceq_{\diamond seq}$. Thus networks of SVP collection homomorphisms are continuous, and have a continuous semantics.

Note that $\preceq_{\diamond seq}$-continuity corresponds to pipeline parallelism (as it did for Kahn), while $\preceq_\diamond$-continuity corresponds to independent parallelism. Let us say that a collection S is *less determined* than another T if wherever S differs from T, S is undetermined ($\perp$). A transducer is $\preceq_\diamond$-continuous if further determining its input cannot cause it to yield a less determined output, and the transducer will not consume input indefinitely before producing output. Similarly, transducers are $\preceq_{\diamond seq}$-continuous if determining their input collections left-to-right cannot cause them to yield output that is less determined left-to-right, and they cannot delay indefinitely before producing output.

A straightforward approach gives us a parallel implementation that corresponds to these semantics. Where Kahn's processes read and write sequences, the processes here read and write collections. The syntax of SVP transducers can be seen as 'reading' the collection structures in the left hand side of the equations, and 'writing' the right hand side. Again, all reads are blocking. We use the symbol $\perp$ to denote an indeterminate collection — a collection that, when read, causes suspension of the reading process. We digress from Kahn's static network of processes, however, in allowing recursive invocations of SVP transducers to spawn new processes.

Evaluation of a collection homomorphism statement such as

$$f(S_1 \diamond S_2) \quad = \quad f(S_1) \diamond f(S_2)$$

therefore suspends until a collection $(S_1 \diamond S_2)$ is obtained as input, and then forks the evaluations of $X_1 = f(S_1)$ and $X_2 = f(S_2)$. The X_1 and X_2 evaluations have independent parallelism. Tree prefix continuity requires that recursive expansion of the collection homomorphism definition be treated properly. Expanding this recursive definition will require spawning of new processes (for parallel subtrees), and thus the network of processes we arrive at is not static – it grows and shrinks as expressions are expanded and evaluated. Kahn actually described proper treatments of recursion in section 4 of his paper; one approach is to evaluate subexpressions of the expansion 'on demand'. The network of processes then can grow arbitrarily large without growing in a way that avoids useful computation.

Collection transductions are similar to collection homomorphisms, but have reduced parallelism. The statement

$$f(Q, S_1 \diamond S_2) \quad = \quad f(Q,\ S_1) \diamond f(\delta(Q, S_1),\ S_2)$$

suspends until $(S_1 \diamond S_2)$ is read, and then forks the evaluations of $X_1 = f(Q, S_1)$, $Q' = \delta(Q, S_1)$, and $X_2 = f(Q', S_2)$. Again, the X_1 and X_2 evaluations have independent parallelism, but this time X_2 is evaluated in pipeline with Q'. Both X_1 and Q' will be suspended while S_1 is $\perp$. We can obtain continuous semantics again, but this time with regard to the the collection sequence prefix ordering $\preceq_{\diamond seq}$, i.e., with regard to left-to-right evaluation of collection trees. This mirrors the Kahn's requirements for sequence transducers [17]. Furthermore, since $\preceq_\diamond$ implies $\preceq_{\diamond seq}$, every $\preceq_\diamond$-continuous function is also $\preceq_{\diamond seq}$-continuous, and a network of transduce transducers and collection homomorphisms is $\preceq_{\diamond seq}$-continuous.

More generally, SVP transducers will involve restructuring and collecting. Unfortunately, in general the restructuring statement

$$f(S_1 \diamond S_2) \quad = \quad f(\rho(S_1)) \; \theta \; f(\rho(S_2))$$

destroys $\preceq_\diamond$-continuity of a network of transducers, as either ρ or θ can require *all* of its input to be determined before it can produce any output, or may be discontinuous in some other way. (For example, when ρ sorts its input it is not continuous, and the operators $\theta = \diamond^R$ is not $\preceq_{\diamond seq}$-continuous, although it is $\preceq_\diamond$-continuous.) In this situation, we cannot give a continuous semantics for a network of transducers. Nevertheless, we can often show that the network is *structure preserving*. Restructuring and collecting often changes the organization of collections to obey a new structure; so restructuring or collecting from a collection preserving P_1 to a collection preserving P_2 will be (P_1,P_2)-structure preserving. Structure preservation of a network of SVP-transducers gives us good handle on the semantics of the network, even though the loss of continuity permits indefinite delays for output to become impossible.

7 An SVP-based Parallel Programming Language

The notation used to specify SVP-mappings is probably too cumbersome to serve as a programming language. Various styles of languages like functional, rule-based, imperative or graphical could be adopted to design a programming language in which SVP collections can be represented and mappings on these collections can be defined. To illustrate this point, we informally present a simple toy parallel programming language with imperative style, henceforth called PL.

7.1 The PL Language

The variables of the PL language take atomic and constructed values as values, and terms can be built up using operators (basic SVP-mappings, SVP-transducers) and functions (arbitrary) that map non-collection values to non-collection values. Programs in PL consist of *iterators* and *definitions*, composed by *sequencing*, *conditional* and a *while-do* looping structure.

Formally, the PL language is defined to contain the following:

Variables

x_1, ..., S_1, ... We use small letters to denote atomic values and tuple values, and capital letters to denote collections.

Terms

The variables x_1, ..., S_1, ... will denote terms, and if $x_1, \ldots, x_n$ are terms, then so is $f(x_1, \cdots, x_n)$, where f is the name of an operator.

Programs

The following are programs:

definitions

if x and S are variable names and t is a term then $x = t$ and $S = t$ are programs.

iterators

if x is a variable name, P is a program, θ is a collector, $f(x, x_1, \ldots, x_n)$ is a term, and S is a term of type collection, then:

$$\textbf{for } x \textbf{ in } S \textbf{ do } P(x, x_1, \ldots, x_n); \textbf{ od}$$

is a program. The program in an iterator may contain special statements:

new $y = f(y, x, x_1, \ldots, x_n)$
output $y\ \theta{=}\ f(x, x_1, \ldots, x_n)$

The first statement updates the y variable for the duration of the program within the iterator; it may (and typically does) refer to the previous value of y. The second statement permits accumulation of output with a collector θ. The notation $y\ \theta{=}\ z$ is shorthand for $y = (y\ \theta\ z)$.

sequencing, looping

if P_1 and P_2 are programs and b is a boolean expression, then: $(P_1;P_2)$, (**if** b **then** P_1 **else** P_2), and (**while** b **do** P_1 **endwhile**), are programs.

We now informally describe the semantics of PL.

Variables can range over SVP values. *Definitions* enable us to name expressions in a way similar to *let* in functional languages, and similar to Kahn's networks of equations.

Iterators enable us to express mapping of collections while 'ranging over' the collection. Let x be a variable and S be a SVP collection. In the statement **for** x **in** S **do** $Prog(x)$ **od**, x *ranges* over the collection S. $Prog(x)$ is executed for each value in S with parallelism determined by $Prog(x)$: the **new** statement forces sequentialization of an iterator; independent parallelism results when $Prog(x)$ contains no **new** declarations. Notice that an iterator exactly corresponds to an SVP-transducer without restructuring, and the **new** statement corresponds to state update.

Finally, sequencing, conditional and looping are defined with the usual semantics.

7.2 PL Program Examples

We start with a program that computes the area of convex polygons as shown earlier. Let triangle(Q) be a predicate that yields **true** if its argument is a 3-tuple of points $Q = (p_0, p_1, p_2)$ such that none of the points are undefined (_). Similarly let next_triangle(Q,p) be a function that yields the 3-tuple (p_0, p_2, p) when Q is a triangle, and otherwise replaces the leftmost _ in Q with p.

```
define total_area (Polygon) as
        T = {};
        Q = (_,_,_) ;
                    for p in Polygon do
                        new Q = next_triangle(Q, p);
                        if triangle(Q) then R = {Q} else R = {};
                        output T ∪= R;
                    od;
        total_area = 0;
                    for x in T do
                        output total_area += area(x);
                    od;
end.
```

We now give a program example that defines a transitive closure operator (henceforth, TCPO) on a binary relation (a set of tuples), using an algorithm originally described in [33]. The use of parallelism is made explicit in the program.

The algorithm TCPO consists of two phases. First, the input relation $Input(A, B)$ is partitioned on the second attribute B, into a tuple of size n called R. Below, partition_to_tuple is a restructuring transducer that accomplishes this. $TCPO$ (the eventual transitive closure) is initialized to {}. Second, the transitive closure is applied to R as a loop of the following operations. The transitive closure increment *Delta* is partitioned on the join attribute (A) to give the D vector. Ranging i over the collection of values $1 \ldots n$, D_i is joined with R_i using the predicate $D_i.A = R_i.B$ (which can be done with SVP-transducers described earlier). The result of the join is then collected with a parallel union into $TCPO$. The while loop terminates when no new tuples are generated (*Delta* becomes empty).

```
define TCPO (Input, h_A, h_B, n) as
        R = partition_to_tuple(h_B, n, Input);
        TCPO = {};
        Delta = Input;
        while Delta ≠ {} do
                D = partition_to_tuple(h_A, n, Delta);
                new Delta = {};
                        for i in 1..n do
                            output Delta ∪= Join(R[i], D[i]);
                        od;
                output TCPO ∪= Delta;
        endwhile;
end.
```

8 Summary

To review what is new about SVP:

- SVP models information with *collections*. Collections include many interesting special cases, including streams, multisets, and groups, and combine sets and streams neatly in a single model.
- Collections are represented as *trees*. Where there have been many attempts to develop data processing models using functional operators on sets or on lists, SVP uses trees. This not only permits us to handle sets and streams in the same model, but also gives an explicit way to represent divide-and-conquer processing and parallel processing.
- SVP operates on collections with SVP transducers, which can be summarized as providing three basic operations: restructure (collection reshaping, including sorting and rebalancing), transduce (tree transduction), and collect (aggregation over collections with pump or APL-reduction).
- SVP is *simple*. It does not rely on sophisticated algebraic concepts, or a powerful higher-order function framework, but on divide-and-conquer and functional composition. Earlier versions of the model experimented with greater sophistication (in fact a sizeable running prototype was written that treated transducers as higher-order functions), but ultimately this was discarded in favor of the current simplicity.
- SVP permits a natural characterization of *structure preserving mappings* on collections, and these mappings have important properties that yield parallelism and performance in data processing.

In SVP, database mappings (queries) are formalized as transducers. These mappings have important properties:

1. SVP-transducers implement many useful bulk data operations: scan computations, relational algebra operators, arbitrary aggregate operators, including FAD's pump operator, arbitrary set mappings, including FAD's filter operator, and many stream mappings (specifically, stream transductions). More generally, SVP-transducers implement divide-and-conquer mappings that appear useful in bulk data processing.
2. SVP-transducers provide a natural means of specifying both independent and pipeline parallelism. At the same time, they have a rigorous semantics based on continuity with respect to collection orderings, that supports both independent and pipeline parallelism. Rigorous fixed point semantics can be derived for networks of SVP-transducers, even for cyclic networks.

The objective of a database model is to find a class of structures and mappings on those structures that: permit conceptualization of complex problems; permit adaptation and extensibility for new situations; permit efficient implementation; are rigorously defined; are generally useful. We feel the SVP model meets these essential criteria, and in addition offers insights on parallel data processing.

9 Future Digressions

SVP is still a model and will benefit greatly from further experience. At UCLA we have developed a simple SVP implementation in Bop, a rewrite rule language, and the examples in this paper all operate as stated. Nevertheless there is no substitute for having 'been there', particularly when it comes to parallel systems.

This section records some topics that have occurred to us as promising avenues for further work.

9.1 A General SVP-based Parallel Programming Language

SVP was designed largely with parallel data processing in mind. The ideas here can also be applied in parallel programming languages, particularly scientific languages aimed at supporting both pipeline and independent parallelism. Recently a variety of parallel programming models have taken a larger view of how parallelism should be expressed. For example, [6] combines both a set orientation with parallel lambda calculus (rewriting) execution. It is interesting to consider a programming language based on the SVP transducer, or equivalents like divide-and-conquer mappings or a generalized pump operator.

We have not discussed it in this paper, but transducers can implement their own *parallel control scheme*. That is, we can use transducers to control networks of transducers, just as control units are used to control CPUs. This aspect of SVP gives it still more promise as a parallel language.

In order to concentrate on modeling issues, we have left the entire issue of 'a nice query language interface for SVP' open. The SVP-transducer formalism is verbose, and not really appropriate for a user-level interface, just as relational algebra is verbose. One natural approach here might be a visual query interface such as the one used for interactive graphic specification of transducers in Stardent's AVS visualization system.

Another natural approach would be a query language with a 'set-former' flavor to it (SQL has such a flavor), perhaps adapting the list-comprehensions used with increasing popularity in functional programming [35]. Note that in [35] Wadler argues list-comprehensions can be generalized to *monad-comprehensions*, with many applications in functional programming. The monad is an interesting algebraic structure that, among other things, can serve as a theoretical setting for the pump and APL reduction operators.

9.2 Array Processing and SVP

One direction to pursue is the use of SVP for array processing problems. There are several possibilities here. First, we might try to modify SVP to avoid the distinction between collections and tuples, perhaps by embedding both in a more general structure like arrays. This is something that Iverson himself attempted to some degree in APL [16], encoding ordered trees in arrays.

Second, we can adopt sequence processing as a basis for scientific computations. This has been proposed by a variety of authors recently. In [1], for example, four kinds of transducers are shown to provide a very useful kernel for parallel array operations. A more abstract recent proposal for basing parallel computation on functional operators over lists appears in [28].

Third, we can modify SVP to include some features for arrays, and some kinds of nested arrays. More has developed an extensive theory of nested arrays in [20, 21], and has shown how it generalizes and improves the (non-nested) model of arrays used in APL. Others have gotten very interesting results with recursive arrays. The classic paper [36] explores the results of storing square arrays as recursive quadtrees (which Wise calls 2^d-ary trees, or quaternary trees), and developing matrix operators as divide-and-conquer operations over these trees. For example, the matrix product can be defined recursively as:

$$
\begin{aligned}
&\mathit{times}(x,0) &&= 0\\
&\mathit{times}(0,y) &&= 0\\
&\mathit{times}(x,1) &&= x\\
&\mathit{times}(1,y) &&= y\\
&\mathit{times}(x,y) &&= xy \quad \text{when } x, y \text{ are numbers}\\
&\mathit{times}\left(\begin{pmatrix} A & B\\ C & D\end{pmatrix},\begin{pmatrix} E & F\\ G & H\end{pmatrix}\right) &&= \begin{pmatrix} \mathit{plus}(\mathit{times}(A,E),\mathit{times}(B,G)) & \mathit{plus}(\mathit{times}(C,E),\mathit{times}(D,G))\\ \mathit{plus}(\mathit{times}(A,F),\mathit{times}(B,H)) & \mathit{plus}(\mathit{times}(C,F),\mathit{times}(D,H))\end{pmatrix}.
\end{aligned}
$$

It is striking how many simple matrix operators can be specified this way — sum, product, inverse, determinant, transpose, etc. Wise shows also that nontrivial operators (full-total-complete-stable pivoting, and the FFT) are easily programmed with this representation. Still another interesting property of this representation is that it not only works, it works for sparse arrays! Sparse arrays are quaternary trees some of whose quadrants are only the constant 0. The definition for *times* above efficiently stores many

sparse arrays and identity matrices (which can be represented as the constant 1). Note that Wise's array representation is easily implementable within SVP by treating quaternary trees as 2-level binary trees. However, it appears the restrictions on recursions imposed by SVP would have to be extended to support general matrix operators.

In array processing, it seems that the restrictions on SVP recursions over collections are not flexible enough for expressing some computations. A good example comes from *sequence comparison* problems [26]. These problems usually have a natural divide-and-conquer structure. For example, if A and B are sequences, and $\langle\rangle$ is the empty sequence, their *minimal string-edit distance* $d(A, B)$ is defined by

$$\begin{array}{lcl}
d(\langle\rangle, \langle\rangle) & = & 0 \\
d(\langle\rangle, y \cdot B) & = & d(\langle\rangle, B) \ + \ w(\langle\rangle, y) \\
d(x \cdot A, \langle\rangle) & = & d(A, \langle\rangle) \ + \ w(x, \langle\rangle) \\
d(x \cdot A, y \cdot B) & = & \min \left\{ \begin{array}{llll} d(A, B) & + & w(x, y) & (x \text{ is replaced by } y) \\ d(A, y \cdot B) & + & w(x, \langle\rangle) & (x \text{ is deleted}) \\ d(x \cdot A, B) & + & w(\langle\rangle, y) & (y \text{ is deleted}) \end{array} \right.
\end{array}$$

where $w(x, y)$ is a function giving the cost of replacing item x by item y. Matrix-oriented dynamic programming techniques are much better for solving this problem than naive recursion. So, even if SVP were extended to handle this kind of dividing-and-conquering, it would be open how to compile it efficiently.

9.3 Query Optimization and Performance in General

We have not discussed it here, but there is a simple theory for composing SVP-transducers, only slightly more complex than that for composing simple functions. In particular, if f_1 and f_2 are collection homomorphisms, there is a single collection homomorphism f_{12} such that for all S

$$f_{12}(S) \ = \ f_2(f_1(S))$$

and f_{12} is definable using the definitions of f_1 and f_2. This is significant for optimization, since it underscores that *multipass mappings can be transformed into single-pass mappings.* Furthermore, many nontrivial algebraic properties of specific transducers (particularly APL reduction) are derivable, and can be used as the basis of a transformation-based query optimizer [3, 35]. It appears that SVP may be a very good model for describing query optimization, since it can formally model scans, access paths, algebraic operators on collections, and parallelism.

We have avoided discussing performance of the transducers in this paper. In some cases extensions of SVP will permit more efficient execution of particular mappings. For example, we might modify the left-to-right transduce transducer to give us a right-to-left variant:

$$\begin{array}{lcl}
\mathsf{ecudsnart}(h,\ \delta,\ Q,\ \langle\rangle) & = & \langle\rangle \\
\mathsf{ecudsnart}(h,\ \delta,\ Q,\ \langle x\rangle) & = & h(Q, x) \\
\mathsf{ecudsnart}(h,\ \delta,\ Q,\ S_1 \diamond S_2) & = & \mathsf{ecudsnart}(h,\ \delta,\ \delta(Q, S_2)\,,\ S_1) \diamond \mathsf{ecudsnart}(h,\ \delta,\ Q,\ S_2).
\end{array}$$

Such a thing is very useful. For example, we can write a relatively fast transducer for '$\star$' (append) with this kind of recursion:

$$\begin{array}{lcl}
S \star T & = & \textbf{if } \mathsf{emptycollection}(S) \textbf{ then } T \textbf{ else } \mathsf{append1}(S, T) \\
\mathsf{append1}(\langle\rangle, T) & = & \langle\rangle \\
\mathsf{append1}(\langle x\rangle, T) & = & \langle x\rangle \diamond T \\
\mathsf{append1}(S_1 \diamond S_2, T) & = & \mathsf{append1}(\,S_1,\ \mathsf{append1}(S_2, T)\,)
\end{array}$$

(Here fst is used as a collector, and append1 is its own δ function.) This is a 'conservative extension' of SVP, since right-to-left recursions are already expressible in SVP:

$$\mathsf{ecudsnart}(h,\ \delta,\ Q,\ S) \ = \ \mathsf{mirror}(\ \mathsf{transduce}(h,\ \delta,\ Q,\ \mathsf{mirror}(\,S\,)))\,.$$

Nevertheless, the overhead of the mirror transducers is avoided if we directly offer right-to-left recursion. It seems inevitable that SVP will grow to incorporate extensions like this over time.

9.4 Networks of Finite Automata, Tree Automata, and Büchi Automata

Networks of finite automata have been studied since the dawn of computer science. A recent survey of the area appeared in [29]. Our work shows that stream processing computations can often be expressed with such networks, and exploring the connection would be interesting.

It is also interesting to consider limitations of transducers that are preserved under composition. A classic limitation restricts attention to 'finite state transducers', also referred to as 'Mealy Machines'. These are finite state automata augmented with an output alphabet and an output function that maps a state and current input symbol to a current output symbol. Each transition of the automaton produces some output.

Tree automata generalize finite state automata to accept as input rooted, ordered, labeled trees, rather than sequences. (The generalization comes from viewing sequences as degenerate trees in which each node has at most one descendant.) A clear and refreshingly informal early survey appears in [30], and a more recent and complete treatment in [12].

In the root-to-frontier tree automaton (RFA), the input tree is traversed from the root to the leaves. Each node in the tree, including the root, is labeled with an input, and usually the tree is taken as having a fixed branching factor (such as 2). The state of the automaton at the root is given, and thereafter the state at any node of the tree is determined by the state at its ancestor node, the input at its ancestor node, and its index among the (possibly multiple) descendants of the ancestor node.

At first it seems that tree automata and SVP-transducers may be very similar. However, the definition of tree automata is not directly comparable with that of SVP-transducers for several reasons:

1. SVP collections are rooted, ordered, trees, but their internal nodes do not carry labels, as is the case with the trees processed by tree automata.

2. The state of a tree automaton at a node in the tree is determined by the state and input at its ancestor node in the tree. On the other hand, the 'state' of an SVP-transducer at a node in a collection is determined by the 'state' at its ancestor node, as well as upon the entire left sibling subtree, if such a sibling exists.

3. Tree automata with output (tree transducers) are typically defined as 'Mealy machines', producing a tree incrementally such that the output at each node of the input tree is a function of the current state and input symbol. SVP-transducers permit considerably more general transductions.

Thus the relationship between SVP and tree automata is not a direct one. Nevertheless, the literature concerning tree automata is considerable, and studying it will perhaps provide some insights or perspective about SVP.

Over the past decade there has been increasing interest in formalizing the behavior of nonterminating processes (such as operating systems), and proving properties of programs over time. One approach for formalizing these has been as automata on infinite sequences; another has been to use temporal logic.

One class of automata, known as *Büchi automata*, have been shown to be equivalent in expressive power to temporal logic. Büchi automata are defined almost exactly like ordinary (nondeterministic) finite automata, and accept a finite word if it drives them into an accepting state, but also accept an infinite word if it drives them into an accepting state an infinite number of times. An excellent review of the work in this area is provided in Chapter 4 of [31].

It would be interesting to relate these results for infinite streams to what we have developed with SVP. In particular, we can perhaps derive a *Temporal Relational Calculus*, a query logic for streams with temporal operators.

History

The bulk of this paper was completed during Parker's sabbatical visit to INRIA in Spring 1991, with the exception of the section on bond analysis, which was added during August 1991. However, we felt something was missing. During Simon's visit to UCLA in November 1991 we realized that restructuring was needed to complete SVP. The present version of this paper thus includes restructuring, chooses names to skirt confusion with similar functional programming concepts, and also reflects experience gained from running the various programs developed in this paper. A demonstration system written in Bop, a rewriting language developed at UCLA, is attached at the end of this paper. All programs in this paper (and more) are presented there.

Earlier this month we were informed by Shamim Naqvi that very similar work was independently developed by Breazu-Tanen, Buneman, and Naqvi [22] during the early part of 1991. This work takes a somewhat different tack, emphasizing not parallelism or 'divide-and-conquer', but the elegant properties of structural recursion for database programming languages. We think this is a strong confirmation of the concepts that have motivated us and suggests that database operators based on structural recursion or SVP-style transduction have a great future.

Acknowledgement

We are indebted to Dennis Shasha for incisive remarks in his review of the paper. Steve Rozen made many significant contributions, with patient and lucid explanations of bond investment concepts, that led ultimately to the case study section in this paper. We are also grateful to Serge Abiteboul for important critical comments and suggestions, to Cliff Walinsky for discussions on parallel data processing, and to Ed Stabler for directing us to the literature on tree automata. Members of the POP Laboratory at UCLA provided helpful comments, particularly Shen-Tzay Huang and Xin Wang. Dave Martin clarified technical issues about cpo's. In addition, Paolo Atzeni provided great support through several drafts of this paper. Finally, the whole SABRE group at INRIA contributed ideas generously during Parker's sabbatical at INRIA in Spring 1991.

References

[1] A. Agrawal, G.E. Blelloch, R.L. Krawitz, C.A. Phillips, Four Vector-Matrix Primitives, *Proc. ACM SPAA*, 292–302, 1989.

[2] Bird, R.S., An Introduction to the Theory of Lists, in M. Broy (ed.), *Logics of Programming and Calculi of Discrete Design*, NY: Springer-Verlag, 3–42, 1987.

[3] Bird, R.S., Algebraic Identities for Program Calculation, *The Computer Journal* **32**:2, 122–126, 1989.

[4] D. Bitton, H. Boral, D. DeWitt, W. Wilkinson, Parallel Algorithms for the Execution of Relational Database Operations, *ACM Transactions on Database Systems*, **8**:3, 1983.

[5] P. Borla-Salamet, C. Chachaty, B. Dageville, Compiling Control into Database Queries for Parallel Execution, *Proc. Int. Conf. on Parallel and Distributed Information Systems*, Miami, Florida, Dec. 1991.

[6] G. Boudol, G. Berry, The Chemical Abstract Machine, *Proc. Seventeenth ACM Symposium on Principles of Programming*, San Francisco, 81–94, January 1990.

[7] W.H. Burge, *Recursive Programming Techniques*, Reading, MA: Addison-Wesley, 1975. Chapter 3, 'Data Structures'

[8] N. Carriero, D. Gelertner, Linda in Context, *Communications of the ACM* **32**: 4, April 1989.

[9] C. Chase et al., Paragon: a Parallel Programming Environment for Scientific Applications Using Communications Structures, *Proc. Int. Conf. on Parallel Programming*, St. Charles Illinois, Aug. 1991.

[10] S. Danforth, P. Valduriez, A FAD for Data-Intensive Applications, *IEEE Trans. Knowledge and Data Engineering*, to appear, 1991.

[11] F.J. Fabozzi, I.M. Pollack (eds.), *The Handbook of Fixed Income Securities* (3rd ed.), Homewood, Illinois: Business One Irwin, 1991.

[12] Gécseg, F. and Steinby, M. *Tree Automata*, Budapest: Akadémiai Kiadó, 1984.

[13] A. Gupta, C. Forgy, A. Newell, High Speed Implementations of Rule-Based Systems, *ACM Transactions on Computer Systems*, **7**:2, May 1989.

[14] B. Hart, S. Danforth, P. Valduriez, Parallelizing FAD, a Database Programming Language, *Int. Symp. on Databases in Distributed and Parallel Systems*, Austin, Texas, Dec. 1988.

[15] W.D. Hillis, G.L. Steele, Data Parallel Algorithms, *Communications of the ACM*, **29**:12, Dec. 1986.

[16] K.E. Iverson, *A Programming Language*, NY: J. Wiley, 1962.

[17] G. Kahn, The Semantics of a Simple Language for Parallel Programming, *Proc. IFIP 74*, North-Holland, 471–475, August 1974.

[18] G. Kahn, D.B. MacQueen, Coroutines and Networks of Parallel Processes, *Proc. IFIP 77*, North-Holland, 993–998, 1977.

[19] R.E. Ladner, M.J. Fischer, Parallel Prefix Computation, *J. ACM* **27**:4, 831–838, 1980.

[20] T. More, Axioms and Theorems for A Theory of Arrays, *IBM J. Res. Devel.* **17**:2, 195–175, 1973.

[21] T. More, The Nested Rectangular Array as a Model of Data, *Proc. APL 1979*, 55–73, May 1979.

[22] V. Breazu-Tanen, P. Buneman, S. Naqvi, "Structural Recursion as a Query Language", *Proc. 3rd International Workshop on Database Programming Languages*, Greece, September 1991.

[23] D.S. Parker, Stream Data Analysis in Prolog, in L. Sterling, ed., *The Practice of Prolog*, MIT Press, 1990.

[24] I. Rival, ed., *Graphs and Order: the role of graphs in the theory of ordered sets and its applications*, Hingham, MA: D. Reidel Publishing Co., distributed by Kluwer Academic Publishers, 1985.

[25] S. Rozen, D. Shasha, Using a Relational System on Wall Street: The Good, The Bad, The Ugly, and the Ideal, *Communications of the ACM* **32**:8, 988–994, August 1989.

[26] D. Sankoff, J.B. Kruskal (eds.), *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, Reading, MA: Addison-Wesley, Advanced Book Program, 1983.

[27] D.A. Schmidt, *Denotational Semantics*, Newton, MA: Allyn & Bacon, Inc., 1986.

[28] D.B. Skillikorn, Architecture-Independent Parallel Computation, *IEEE Computer* **23**:12, 38–50 (December 1990).

[29] F.F. Soulié, Y. Robert, M. Tchuente, *Automata networks in computer science: Theory and applications*, Biddles Ltd. and Princeton University Press, 1987.

[30] J.W. Thatcher, Tree Automata: an Informal Survey, in A.V. Aho, ed., *Currents in the Theory of Computing*, Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

[31] A. Thayse, ed., *From Modal Logic to Deductive Databases*, J. Wiley and Sons, 1989.

[32] Thinking Machines Corporation, *Programming in C**, Version 5.0, Cambridge, Ma, 1989.

[33] P. Valduriez, S. Khoshafian, Parallel Evaluation of the Transitive Closure of a Database Relation, *International Journal of Parallel Programming*, Vol. 17, No. 1, 19–42 (1988).

[34] P. Valduriez, ed., *Data Management and Parallel Processing*, London: Chapman and Hall, 1991.

[35] P. Wadler, Comprehending Monads, *Proc. 1990 ACM Conf. on LISP and Functional Programming*, Nice, France, 61–78, June 1990.

[36] David S. Wise, Matrix Algebra and Applicative Programming, in Gilles Kahn (ed.), *Functional Programming Languages and Computer Architecture* (Proceedings), Portland, Oregon, USA, Springer LNCS 274, pp. 134-153. September 1987.

[37] O. Wolfson, A. Ozeri, A New Paradigm for Parallel and Distributed Rule Processing, Proc. ACM SIGMOD Int. Conf., Atlantic City, May 1990.

[38] ANSI X3J3 Committee, *Draft International Standard Fortran*. Technical Report, ANSI, June 1990.

```
%-------------------------------------------------------------------------
%       simple SVP implementation
%-------------------------------------------------------------------------
%
%  The implementation is done here in Bop, a conditional rewrite rule language
%  and extension of Prolog, developed at UCLA.
%
%  The examples here are drawn directly from the paper:
%
%      D.S. Parker, E. Simon, P. Valduriez,
%      SVP -- a Model Capturing Sets, Streams, and Parallelism
%      1992.
%
%  In some cases we have changed the argument or statement order slightly
%  as a consequence of Bop's implementation.  Prolog indexing is typically
%  restricted to the first argument of predicates, and Bop does not change
%  this.  As a result, the first argument is used for the recursion argument,
%  whereas in the paper the last argument is typically used.
%
%  Bop can be viewed as a lazy functional language.  The '=>>' primitive
%  can be used to force evaluation explicitly when necessary.
%
%-------------------------------------------------------------------------
%       SVP model primitives
%-------------------------------------------------------------------------
%  Here the infix operator '<>' is used as the collection constructor.
%  '{}' represents the empty collection.
%  {X} represents the collection containing one value X.
%  Thus   ({1}<>{2})<>({3}<>{4})  is a balanced collection.
%
%  By default, the '<>' operator is right associative, so the unparenthesized
%  collection   {1}<>{2}<>{3}<>{4}  is a (right-linear, list-like) sequence.
%-------------------------------------------------------------------------

:- op(550,xfy,<>).

({}) <> S => S :- !.
S <> ({}) => S :- !.
S <> T => S <> T.

({}) => ({}).
({X}) => ({X}).

emptycollection({})  =>  true  :-  !.
emptycollection(_)   =>  false.

unitcollection({_})  =>  true  :-  !.
unitcollection(_)    =>  false.

unitcollectionvalue({X})  =>  X.

collection({})    =>  true  :-  !.
collection({_})   =>  true  :-  !.
collection(_<>_)  =>  true  :-  !.
collection(_)     =>  false.

atom(X)  =>  true  :-  atomic(X), !.
atom(_)  =>  false.

tuple(X)  =>  true  :-  functor(X,t,_), !.
tuple(_)  =>  false.    % tuples are represented as terms of form  t(_,_,...,_)

t(X1)          =>  t(X1).
t(X1,X2)       =>  t(X1,X2).
t(X1,X2,X3)    =>  t(X1,X2,X3).
t(X1,X2,X3,X4) =>  t(X1,X2,X3,X4).

%-------------------------------------------------------------------------
%       Collectors and other useful binary operators
%-------------------------------------------------------------------------
:- op(400,xfy,(::)).    %  collection append
:- op(400,xfy,(<>~)).   %  S <>~ T  =  T <> S
:- op(400,xfy,(min)).   %  value min
:- op(400,xfy,(max)).   %  value max
:- op(400,xfx,(or)).    %  boolean or
:- op(400,xfx,(and)).   %  boolean and
:- op(400,xfx,(fst)).   %  X fst Y = X  [not a collector]
:- op(400,xfx,(snd)).   %  X snd Y = Y  [not a collector]


(S <>~ T)  =>  T <> S.  % reverse of <>

(S :: T)   =>  if  emptycollection(S)  then  T  else  append1(S,T).

append1({},_)       =>  {}.
append1({X},T)      =>  {X} <> T.
append1(S1<>S2,T)   =>  append1(S1,append1(S2,T)).

(A + B)    =>  C  :- A =>> X, B =>> Y, C is X+Y.
(A * B)    =>  C  :- A =>> X, B =>> Y, C is X*Y.
(A min B)  =>  C  :- A =>> X, B =>> Y, minValue(X,Y,C).
(A max B)  =>  C  :- A =>> X, B =>> Y, maxValue(X,Y,C).

minValue(X,Y,X) :- X =< Y, !.
minValue(X,Y,Y) :- X > Y.

maxValue(X,Y,X) :- X >= Y, !.
maxValue(X,Y,Y) :- X < Y.

true  or true   =>  true.
true  or false  =>  true.
false or true   =>  true.
false or false  =>  false.

true  and true   =>  true.
true  and false  =>  false.
false and true   =>  false.
false and false  =>  false.


(A - B)   =>  C  :- A =>> X, B =>> Y, C is X-Y.
(A / B)   =>  C  :- A =>> X, B =>> Y, C is X/Y.

(X fst _) => X.
(_ snd X) => X.

%-------------------------------------------------------------------------
%       Diffs of a sequence or stream ([]-terminated sequence)
%-------------------------------------------------------------------------

diffs(S)  =>  diffs1(S,{}).

diffs1({},_)       =>  {}.
diffs1({_},{})     =>  {}    :-  !.
diffs1({[]},_)     =>  {[]}  :-  !.
diffs1({X},{Y})    =>  {D}   :-  D is X-Y.
diffs1(S1<>S2,Q)   =>  diffs1(S1,Q) <> diffs1(S2,S1).
```

```
diffs_demo  => diffs( {98}<>{99}<>{97}<>{97}<>{99}<>{96}<>{[]} ).

%-----------------------------------------------------------------------------
%       Polygon example
%-----------------------------------------------------------------------------
:- [library(math_pl)].   %  include Prolog sqrt() function

total_area(Polygon)  =>  sum(areas(triangles(Polygon))).


triangles(S) =>  triangles1(S,t(_,_,_)).

triangles1({},_)       =>  {}.
triangles1({P},Q)      =>  output_triangle(NQ)   :- next_triangle({P},Q,NQ).
triangles1(S1<>S2,Q)   =>  triangles1(S1,Q) <> triangles1(S2,NQ)
                                          :- next_triangle(S1,Q,NQ).

output_triangle(Q)  =>  {Q} :- complete_triangle(Q), !.
output_triangle(_)  =>  {}.

complete_triangle(t(P0,P1,P2)) :- nonvar(P0), nonvar(P1), nonvar(P2).

next_triangle({P},t(P0,_ ,P2),t(P0,P2,P)) :- nonvar(P2), !.
next_triangle({P},t(P0,P1,_ ),t(P0,P1,P)) :- nonvar(P1), !.
next_triangle({P},t(P0,_ ,_ ),t(P0,P,_))  :- nonvar(P0), !.
next_triangle({P},t(_ ,_ ,_ ),t(P,_,_)).


areas({})             =>  {0}.
areas({t(P0,P1,P2)})  =>  {A}  :-  heron_area(P0,P1,P2,A).
areas(S1<>S2)         =>  areas(S1) <> areas(S2).

heron_area(P0,P1,P2,Area) :-
        distance(P0,P1,A),
        distance(P1,P2,B),
        distance(P2,P0,C),
        S is (A+B+C)/2,
        AreaSq is S*(S-A)*(S-B)*(S-C),
        sqrt(AreaSq,Area).

distance((X1,Y1),(X2,Y2),Distance) :-
        DistanceSq is (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2),
        sqrt(DistanceSq,Distance).



sample_polygon  =>  {(0,0)} <> {(1,4)} <> {(5,6)} <> {(6,3)} <> {(4,-1)}.

polygon_demo  => total_area(sample_polygon).


%-----------------------------------------------------------------------------
%       Restructuring
%-----------------------------------------------------------------------------

mirror({})      =>  {}.
mirror({X})     =>  {X}.
mirror(S1<>S2)  =>  mirror(S1)  <>~  mirror(S2).

mirror_demo  =>  mirror( ({1} <> {2}) <> ({3} <> ({4} <> {5})) ).

%-----------------------------------------------------------------------------
%       Sequencing (flattening) a collection
%-----------------------------------------------------------------------------

naive_sequence(S)  =>  sequence1(first_rest(S)).

sequence1({})       =>  {}.
sequence1({X})      =>  {X}.
sequence1(S1<>S2)   =>  sequence1(first_rest(S1)) <> sequence1(first_rest(S2)).

first_rest(S)  =>  first(S) <> rest(S).

first({})       =>  {}.
first({X})      =>  {X}.
first(S1<>S2)   =>  first(S1) fst first(S2).

rest(S)  =>  rest1(S,true).

rest1({},_)             =>  {}.
rest1({X},Initial)      =>  if Initial then {} else {X}.
rest1(S1<>S2,Initial)   =>  rest1(S1,Initial) <> rest1(S2,false).

naive_sequence_demo  =>
        sequence( ({8}<>(({3}<>{1})<>(({7}<>{5})<>{9})))<>({4}<>{2}) ).


sequence({})       =>  {}.
sequence({X})      =>  {X}.
sequence(S1<>S2)   =>  sequence(S1) :: sequence(S2).

sequence_demo  =>
        sequence( ({8}<>(({3}<>{1})<>(({7}<>{5})<>{9})))<>({4}<>{2}) ).


streamify(S)  =>  if nil_terminated(S) then sequence(S) else sequence(S<>{[]}).

nil_terminated(S) => if last(S) = {[]} then true else false.

last({})       =>  {}.
last({X})      =>  {X}.
last(S1<>S2)   =>  last(S1)  snd  last(S2).

%-----------------------------------------------------------------------------
%       Partitioning
%-----------------------------------------------------------------------------

:- op(650,xfx,(combine_partitions)).

t(R1,R2)  combine_partitions  t(S1,S2)  =>  t(R1S1,R2S2)  :-
        R1<>S1 =>> R1S1,
        R2<>S2 =>> R2S2.

partition_tree(t(P1,P2)) => P1 <> P2.


partition(S,P) => partition_tree(partitionify(S,P)).

partitionify({},_)       =>  t({},{}).
partitionify({X},P)      =>  if P@X then t({X},{}) else t({},{X}).
partitionify(S1<>S2,P)   =>  partitionify(S1,P)
                                 combine_partitions  partitionify(S2,P).


partition_demo  => partition( {a}<>{1}<>{2}<>{b}<>{3}, numeric).

numeric => X \ (Num  :-  X =>> Xval, number(Xval) -> Num=true ; Num=false).
```

```
%-----------------------------------------------------------------------------
%       Balancing
%-----------------------------------------------------------------------------

balance(S)  =>  balance1(split(sequence(S))).

balance1({})     =>  {}.
balance1({X})    =>  {X}.
balance1(S<>T)   =>  balance1(split(S)) <> balance1(split(T)).

split(S)  =>  partition_tree(halves(sequence(S),1,N)) :- count(S) =>> N.

halves({},_,_)       =>  t({},{}).
halves({X},I,N)      =>  t({X},{})  :-  I =< N/2, !.
halves({X},_,_)      =>  t({},{X}).
halves(S1<>S2,I,N)   =>  halves(S1,I,N)  combine_partitions  halves(S2,I1,N)
                                :-  I1 is I+1.


balance_demo  => balance( {a}<>{3}<>{1}<>{b}<>{5}<>{d}<>{4}<>{2} ).

halves_demo  => halves( {a}<>{3}<>{1}<>{b}<>{5}<>{d}<>{4}<>{2} ,1, 8).

%-----------------------------------------------------------------------------
%       Grouping
%-----------------------------------------------------------------------------
:-  op(650,xfy,(combine_groups)).

group(H,S)  =>  gather(hashify(S,H)).

hashify({},_)       =>  {}.
hashify({X},H)      =>  {t(HX,{X})}  :-  H@X =>> HX.
hashify(S1<>S2,H)   =>  hashify(S1,H) <> hashify(S2,H).

gather(S)  =>  gather1(sequence(S),{}).

gather1({},_)       =>  {}.
gather1({X},Q)      =>  insert(Q,{X}).
gather1(S1<>S2,Q)   =>  gather1(S1,Q)  snd  gather1(S2,insert(S1,Q)).

insert({},_)        =>  {}.
insert({B},S0)      =>  S1          :-  insert1(S0,B) =>> t(true,S1), !.
insert({B},S0)      =>  S0 <> {B}.
insert(S1<>S2,S0)   =>  insert(S1,S0)  snd  insert(S2,insert(S1,S0)).

insert1({},_)                 =>  t(false,{}).
insert1({t(K1,S)},t(K,{X}))   =>  t(true, {t(K,S<>{X})})  :-  K=K1, !.
insert1({t(K1,S)},_)          =>  t(false,{t(K1,S)}    ).
insert1(S1<>S2,Q)             =>  insert1(S1,Q)  combine_groups  insert1(S2,Q).

t(B1,S1)  combine_groups  t(B2,S2)  =>  t(B1orB2, S1S2)  :-
                                        (B1 or B2) =>> B1orB2,
                                        S1<>S2 =>> S1S2.

group_demo  =>  group( crude_hash, {a}<>{b}<>{c}<>{d}<>{e}<>{f}<>{g} ).

crude_hash => X \ (Hashvalue :- name(X,[C|_]), Hashvalue is C/\3).


%-----------------------------------------------------------------------------
%       Simple Statistics
%-----------------------------------------------------------------------------

avg => avg.
avg(S)  =>  A :- sum(S) =>> T, count(S) =>> N, A is T/N.

sum({})       =>  0.
sum({X})      =>  X.
sum(S1<>S2)   =>  sum(S1) + sum(S2).

count({})     =>  0.
count({_})    =>  1.
count(S1<>S2) =>  count(S1) + count(S2).


avg_demo  => avg( {8}<>{3}<>{1}<>{7}<>{5}<>{9}<>{4}<>{2}<>{6} ).

%-----------------------------------------------------------------------------
%       Sorting
%-----------------------------------------------------------------------------

sort(S) =>  sort(S, lessEq, first).

sort(S,Ordering,MedianEstimate) =>
        sort1(
                partition(S, (X\(Ordering@X@(MedianEstimate@S)))),
                Ordering,
                MedianEstimate
        ).

sort1({},_,_) => {}.
sort1({X},_,_) => {X}.
sort1(S<>T,Ordering,MedianEstimate) =>
        sort1(
                partition(S, (X\(Ordering@X@(MedianEstimate@S)))),
                Ordering,
                MedianEstimate)
        <>
        sort1(
                partition(T, (Y\(Ordering@Y@(MedianEstimate@T)))),
                Ordering,
                MedianEstimate).

lessEq => (X \ Y \ (T :- X=>>A, Y=>>B, (compare(>,A,B) -> T=false ; T=true))).

sort_demo  => sort( {8}<>{3}<>{1}<>{7}<>{5}<>{9}<>{4}<>{2}, lessEq, avg).

%-----------------------------------------------------------------------------
%       Generalized joins and Combine
%-----------------------------------------------------------------------------

combine(Map1,Map2,Test,Rslt,R,S)        => combine1(R,S,Map1,Map2,Test,Rslt).

combine1({},                    _,_,_,_,_) => {}.
combine1({P},   S,Map1,Map2,Test,Rslt) => combine2(Map1P,Map2PS,Test,Rslt) :-
                                             Map1@P =>> Map1P, Map2@P@S =>> Map2PS.
combine1(P1<>P2,S,Map1,Map2,Test,Rslt) => combine1(P1,S,Map1,Map2,Test,Rslt)
                                             <> combine1(P2,S,Map1,Map2,Test,Rslt).

combine2({},            _,_,_)  =>  {}.
combine2({X},     Si,Test,Rslt)  =>  combine3(Si,{X},Test,Rslt).
combine2(Ri1<>Ri2,Si,Test,Rslt)  =>  combine2(Ri1,Si,Test,Rslt)
                                         <> combine2(Ri2,Si,Test,Rslt).

combine3({},            _,_,_)  =>  {}.
combine3({Y},     {X},Test,Rslt)  =>  if  Test@X@Y  then  Rslt@X@Y  else  {}.
combine3(Si1<>Si2,  Q,Test,Rslt)  =>  combine3(Si1,Q,Test,Rslt)
```

```
                                <> combine3(Si2,Q,Test,Rslt).


nested_loops_join(RJoinField,SJoinField,R,S) =>
        combine(
                unit,                                           % MAP1
                second,                                         % MAP2
                matching_tuples(RJoinField,SJoinField),         % TEST
                join_output(RJoinField,SJoinField),             % RESULT
        R,S).

unit  =>  (X \ {X}).
second  => (_ \ Y \ Y).
matching_tuples(RJoinField,SJoinField) =>
                (Rtuple \ Stuple \ (Match :-
                        tuple_field(RJoinField,Rtuple) =>> RField,
                        tuple_field(SJoinField,Stuple) =>> SField,
                        compare(C,RField,SField),
                        (C == (=) -> Match = true ; Match = false))
                ).
join_output(RJoinField,SJoinField) =>
        (Rtuple\ Stuple\ join_result(RJoinField,SJoinField,Rtuple,Stuple)).

tuple_field(FieldNumber) => (Tuple \ (Field :- arg(FieldNumber,Tuple,Field))).
tuple_field(FieldNumber,Tuple) => (Field :- arg(FieldNumber,Tuple,Field)).

join_result(FieldNumber1,FieldNumber2,Tuple1,Tuple2) => {Tuple} :-
        tuple_args_except(Tuple1,FieldNumber1,List1),
        tuple_args_except(Tuple2,FieldNumber2,List2),
        appendList(List1,List2,OtherFields),
        arg(FieldNumber1,Tuple1,JoinField),
        Tuple =.. [t,JoinField|OtherFields].

tuple_args_except(Tuple,FieldNumber,List) :-
        functor(Tuple,t,Arity),
        tuple_args_except(0,Arity,Tuple,FieldNumber,List).
tuple_args_except(N,N,_,_,[]) :- !.
tuple_args_except(I0,Arity,Tuple,FieldNumber,List) :-
        I is I0+1,
        I == FieldNumber, !,
        tuple_args_except(I,Arity,Tuple,FieldNumber,List).
tuple_args_except(I0,Arity,Tuple,FieldNumber,[Field|List]) :-
        I is I0+1,
        arg(I,Tuple,Field),
        tuple_args_except(I,Arity,Tuple,FieldNumber,List).

appendList([],L,L).
appendList([X|L1],L2,[X|L3]) :- appendList(L1,L2,L3).



hash_join(RJoinField,SJoinField,R,S) =>
        combine(
                group_bucket,                                   % MAP1
                matching_bucket,                                % MAP2
                matching_tuples(RJoinField,SJoinField),         % TEST
                join_output(RJoinField,SJoinField),             % RESULT
                RGs,
                SGs
        ) :-
                group(tuple_field_hash(RJoinField),R) =>> RGs,
                group(tuple_field_hash(SJoinField),S) =>> SGs.

group_bucket  => (Group \ (Bucket  :-  Group = t(_Key,Bucket))).

matching_bucket => (Group \ GroupCollection \ (MatchingBucket :-
                        Group = t(Key,_Bucket),
                        keyBucket(GroupCollection,Key) =>> MatchingBucket)).

:- op(550,xfy,(select_bucket)).
{} select_bucket G  =>  G  :-  !.
G  select_bucket _  =>  G.

keyBucket({},                   _)  =>  {}.
keyBucket({t(Key1,Bucket)},Key)  =>  Bucket  :-  Key1 == Key, !.
keyBucket({t(_,_)},             _)  =>  {}.
keyBucket(S1<>S2,           Key)  =>  keyBucket(S1,Key) select_bucket
                                        keyBucket(S2,Key).


tuple_field_hash(FieldNumber) => (Tuple \ (HashValue :-
                                        arg(FieldNumber,Tuple,Field),
                                        name(Field,[C|_]),
                                        name(HashValue,[C]))).
tuple_field_hash(FieldNumber,Tuple) => (HashValue :-
                                        arg(FieldNumber,Tuple,Field),
                                        name(Field,[C|_]),
                                        name(HashValue,[C])).



nested_loops_demo  =>  nested_loops_join(1,1,emp,project).

hash_join_demo  =>  hash_join(1,1,emp,project).


emp =>
        {t( amiel,           39-63-59-65 )} <>
        {t( amsaleg,         39-63-52-55 )} <>
        {t( baque,           39-63-52-50 )} <>
        {t( bellosta,        39-63-59-65 )} <>
        {t( cheiney,         39-63-52-52 )} <>
        {t( cirio,           39-63-52-50 )} <>
        {t( darrieumerlou,   39-63-55-72 )} <>
        {t( daynes,          39-63-52-55 )} <>
        {t( de_maindreville, 39-63-56-19 )} <>
        {t( gardarin,        39-63-56-86 )} <>
        {t( gruber,          39-63-56-35 )} <>
        {t( kiernan,         39-63-56-19 )} <>
        {t( lanzelotte,      39-63-56-32 )} <>
        {t( parker,          39-63-59-64 )} <>
        {t( pucheral,        39-63-55-79 )} <>
        {t( simon,           39-63-52-52 )} <>
        {t( thevenin,        39-63-55-79 )} <>
        {t( valduriez,       39-63-52-51 )} <>
        {t( viallet,         39-63-55-79 )} <>
        {t( zait,            39-63-56-18 )} <>
        {t( ziane,           39-63-56-32 )}.


project =>
        {t( amiel,           omnis          )} <>
        {t( amsaleg,         eos            )} <>
        {t( baque,           sabre          )} <>
        {t( bellosta,        omnis          )} <>
        {t( cheiney,         rdl            )} <>
        {t( cirio,           sabre          )} <>
        {t( darrieumerlou,   geode          )} <>
```

```
        {t( daynes,          geode          )} <>
        {t( de_maindreville, rdl            )} <>
        {t( gardarin,        optimisation   )} <>
        {t( gardarin,        sabre          )} <>
        {t( gruber,          eos            )} <>
        {t( kiernan,         rdl            )} <>
        {t( lanzelotte,      optimisation   )} <>
        {t( parker,          svp            )} <>
        {t( pucheral,        geode          )} <>
        {t( simon,           rdl            )} <>
        {t( simon,           svp            )} <>
        {t( simon,           sabre          )} <>
        {t( thevenin,        geode          )} <>
        {t( valduriez,       eos            )} <>
        {t( valduriez,       geode          )} <>
        {t( valduriez,       omnis          )} <>
        {t( valduriez,       optimisation   )} <>
        {t( valduriez,       sabre          )} <>
        {t( valduriez,       svp            )} <>
        {t( viallet,         omnis          )} <>
        {t( zait,            optimisation   )} <>
        {t( ziane,           optimisation   )}.


%-----------------------------------------------------------------------------
%       Merge Scans
%-----------------------------------------------------------------------------

merge_scan(R,S)    =>  merge1(sequence(S),sequence(R)).

merge1({},     R)  =>  R.
merge1({Y},    R)  =>  prefixy(R,{Y}).
merge1(S1<>S2,R)   =>  merge1(S1,R) :: merge1(suffix(R,S1),S2).

prefixy(R,{Y})     =>  prefix(R,{Y}) :: {Y}.

prefix({},     _)  =>  {}.
prefix({X}, {Y})   =>  {X}  :-  X=<Y, !.
prefix({_},    _)  =>  {}.
prefix(S1<>S2,V)   =>  prefix(S1,V) :: prefix(S2,V).

suffix({},     _)  =>  {}.
suffix({X}, {Y})   =>  {X}  :-  X>Y, !.
suffix({_},    _)  =>  {}.
suffix(S1<>S2,V)   =>  suffix(S1,V) :: suffix(S2,V).

merge_demo  =>  merge_scan( {1}<>{4}<>{8}<>{9}, {2}<>{3}<>{5}<>{11}).

%-----------------------------------------------------------------------------
%       Set operations
%-----------------------------------------------------------------------------

union(R,S)  =>
                sorted_union(streamify(sort(R)),streamify(sort(S))).

sorted_union(R,S)  =>  sorted_union1(S,R).

sorted_union1({[]},  R) => R  :-  !.
sorted_union1({Y}<>S,R) =>
        union_output(R,Y) <> sorted_union1(S,union_state(R,Y)).

union_output(R,Y)  =>  union_output1(R,Y,false).

union_output1({},     _,_)  =>  {}.
union_output1({X},    Y,C)  =>  union_output_increment(Y,X,C).
union_output1(Q1<>Q2,Y,C)   =>
                union_output1(Q1,Y,C)
                <> union_output1(Q2,Y,output_complete(Q1,Y,C)).

union_output_increment(Y,X,Complete)  =>
                if Complete then {}
                else (if X = [] then {Y}
                else (if X > Y then {Y}
                else (if X = Y then {}
                else /*  X < Y  */  {X}))).

union_state({},     _)   =>  {}.
union_state({X},    Y)   =>  union_state_increment(Y,X).
union_state(Q1<>Q2,Y)    =>  union_state(Q1,Y) <> union_state(Q2,Y).

union_state_increment(Y,X)  =>
                if X = [] then {[]}
                else (if X > Y then {X}
                else (if X = Y then {X}
                else /*  X < Y  */  {})).


intersection(R,S)  =>
                sorted_intersection(streamify(sort(R)),streamify(sort(S))).

sorted_intersection(R,S)  =>  sorted_intersection1(S,R).

sorted_intersection1({[]},  _) => {[]}  :-  !.
sorted_intersection1({Y}<>S,R) =>
                intersection_output(R,Y)
                <> sorted_intersection1(S,intersection_state(R,Y)).

intersection_output(R,Y)  =>  intersection_output1(R,Y,false).

intersection_output1({},     _,_)  =>  {}.
intersection_output1({X},    Y,C)  =>  intersection_output_increment(Y,X,C).
intersection_output1(Q1<>Q2,Y,C)   =>
                intersection_output1(Q1,Y,C)
                <> intersection_output1(Q2,Y,output_complete(Q1,Y,C)).

intersection_output_increment(Y,X,Complete)  =>
                if Complete then {}
                else (if X = [] then {}
                else (if X > Y then {}
                else (if X = Y then {Y}
                else /*  X < Y  */  {}))).

intersection_state({},     _)   =>  {}.
intersection_state({X},    Y)   =>  intersection_state_increment(Y,X).
intersection_state(Q1<>Q2,Y)    =>  intersection_state(Q1,Y)
                                    <> intersection_state(Q2,Y).

intersection_state_increment(Y,X)  =>
                if X = [] then {[]}
                else (if X > Y then {X}
                else (if X = Y then {X}
                else /*  X < Y  */  {})).


difference(R,S)  =>
                sorted_difference(streamify(sort(R)),streamify(sort(S))).

sorted_difference(R,S)  =>  sorted_difference1(S,R).
```

```
sorted_differencel({[]},  R} => R  :-  !.
sorted_differencel({Y}<>S,R} => difference_output(R,Y)
                              <> sorted_differencel(S,difference_state(R,Y)).

difference_output(R,Y)  =>  difference_output1(R,Y,false).

difference_output1({},     _,_)  =>  {}.
difference_output1({X},    Y,C)  =>  difference_output_increment(Y,X,C).
difference_output1{Q1<>Q2,Y,C)  =>
               difference_output1(Q1,Y,C)
               <> difference_output1(Q2,Y,output_complete(Q1,Y,C)).

difference_output_increment(Y,X,Complete)  =>
               if Complete then {}
               else (if X = [] then {}
               else (if X > Y then {}
               else (if X = Y then {}
               else /*  X < Y  */  {X}))).

difference_state({},     _)  =>  {}.
difference_state{X},     Y)  =>  difference_state_increment(Y,X).
difference_state(Q1<>Q2,Y)   =>  difference_state(Q1,Y)
                                   <> difference_state(Q2,Y).

difference_state_increment(Y,X)  =>
               if X = [] then {[]}
               else (if X > Y then {X}
               else (if X = Y then {}
               else /*  X < Y */   {})).


output_complete(_,_,true) => true  :-  !.
output_complete({[]},_,_) => true  :-  !.
output_complete({X},Y,_)  => if X < Y then false else true.



(X < Y)  =>  true  :-  compare(<,X,Y), !.
(_ < _)  =>  false.

(X = Y)  =>  true  :-  compare(=,X,Y), !.
(_ = _)  =>  false.

(X > Y)  =>  true  :-  compare(>,X,Y), !.
(_ > _)  =>  false.


r => {t(a,1)} <> {t(b,2)} <> {t(c,3)} <> {t(e,5)} <> {[]}.

s => {t(a,1)} <> {t(c,3)} <> {t(d,4)} <> {[]}.

union_demo  =>  sorted_union(R,S)  :-  r =>> R, s =>> S.
difference_demo  =>  sorted_difference(R,S)  :-  r =>> R, s =>> S.
intersection_demo  =>  sorted_intersection(R,S)  :-  r =>> R, s =>> S.

%-------------------------------------------------------------------------
%       Bond yield analysis
%-------------------------------------------------------------------------

bond(
        scg,
        'Southern California Gas',
        'Open Mortgage--Outstg.',
        0.0825,
        100.00,
        'A1',
        96/11/15,
        2,
                {payment(91/05/01, 4.12)} <>
                {payment(91/11/01, 4.13)} <>
                {payment(92/05/01, 4.12)} <>
                {payment(92/11/01, 4.13)} <>
                {payment(93/05/01, 4.12)} <>
                {payment(93/11/01, 4.13)} <>
                {payment(94/05/01, 4.12)} <>
                {payment(94/11/01, 4.13)} <>
                {payment(95/05/01, 4.12)} <>
                {payment(95/11/01, 4.13)} <>
                {payment(96/05/01, 4.12)} <>
                {payment(96/11/01, 4.13)} <>
                {payment(96/11/15, 100.00)} <>
                {[]}
        ,
                {call(90/11/15, 91/11/14, 104.43)} <>
                {call(91/11/15, 92/11/14, 103.32)} <>
                {call(92/11/15, 93/11/14, 102.21)} <>
                {call(93/11/15, 94/11/14, 101.11)} <>
                {call(94/11/15, 95/11/14, 100.00)} <>
                {call(95/11/15, 96/11/14, 100.00)} <>
                {[]}
).

yield_to_worst(BondId,Price,Date) =>
        min(
                yield_to_maturity(BondId,Price,Date),
                yield_to_call(BondId,Price,Date)
        ).

yield_to_maturity(BondId,Price,Date) => Yield :-
        Date = (CurrentYear/_/_),
        bond(BondId,_,_,Coupon,Face,_,(MaturityYear/_/_),M,_,_),
        N is MaturityYear-CurrentYear+1,
        C is Coupon*Face,
        yield(Price,C,Face,M,N,Yield).

yield_to_call(BondId,Price,Date) =>
        collection_min( yields(Payments,Calls,Date,Price,C,M,0) )
        :- bond(BondId,_,_,Coupon,Face,_,_,M,Payments,Calls), C is Coupon*Face.

yields({[]},_,_,_,_,_) => {[]}.
yields({_}<>{[]},_,_,_,_,_,_) => {[]} :- !.
yields({payment(PD,_)}<>Ps,Cs,Date,P,C,M,N) => yields(Ps,Cs,Date,P,C,M,N) :-
        beforeDate(PD,Date), !.
yields({payment(PD,_)}<>Ps,Cs,D,P,C,M,N0)    => {Yield}<>yields(Ps,Cs,D,P,C,M,N)
        :- N is N0 + 1/M,
        call_price(Cs,Ps,PD) =>> R,
        yield(P,C,R,M,N, Yield).


call_price({call(_,CallEnd,_CP)}<>Calls,Ps,PD) => call_price(Calls,Ps,PD) :-
        beforeDate(CallEnd,PD),
        CallEnd \== PD,
        !.
call_price({call(CallStart,CallEnd,CP)}<>Cs,Ps,PD) => CallPrice :-
        beforeDate(CallStart,PD),
        beforeDate(PD,CallEnd),
        !,
```

```
        Cs =>> Calls,
        Ps =>> Payments,
        callPrice(Calls,Payments,CP,CallPrice).

callPrice({[]},_,CP,CP) :- !.
callPrice(_,{[]},CP,CP) :- !.
callPrice({call(NextCallStart,_,NextCP)}<>_,{payment(NextPD,_)}<>_,CP,CallPrice)
        :- (beforeDate(NextCallStart,NextPD) ->
                CallPrice is NextCP
        ;
                CallPrice is CP
        ),
        !.


beforeDate(Date1,Date2) :-
        Date1 = (Y1/M1/D1),
        Date2 = (Y2/M2/D2),
        (Y1<Y2 ; (Y1=Y2, (M1<M2 ; M1=M2, D1=<D2))),
        !.

collection_min({})      =>  X  :-  plusInfinity(X).
collection_min({[]})    =>  X  :-  plusInfinity(X), !.
collection_min({X})     =>  X.
collection_min(S1<>S2)  =>  collection_min(S1)  min  collection_min(S2).

plusInfinity(1.79e+308).  % double precision IEEE

%-------------------------------------------------------------------------------
%
%  Let $yield(P,C,R,m,n)$ denote the yield of a bond with:
%  \begin{itemize}
%   \item purchase price $P$,
%   \item annual coupon interest $C$,
%   \item final redemption value (maturity face value) $R$,
%   \item yearly payment frequency $m$,
%   \item which is held over $n$ years.
%  \end{itemize}
%
%  The yield $y~=~yield(P,C,R,m,n)$ is defined by the equation
%  \[
%  P
%  ~~~=~~~
%  \frac{C}{m} ~ \sum_{i=1}^{mn} ~ (1 + \frac{y}{m})^{-i}
%  ~~+~~
%  R ~ (1 + \frac{y}{m})^{-mn} ,
%  \]
%  using the definition for compound interest.
%  For bonds, $m$ is typically 2.
%  Values for $y$ are typically found iteratively, or with tables.
%
%  Using Newton's method, the error $error(y)$ in this sum is
%  \[
%  error(y)
%  ~~~=~~~
%  \frac{C}{m} ~ \sum_{i=1}^{mn} ~ (1 + \frac{y}{m})^{-i}
%  ~~+~~
%  R ~ (1 + \frac{y}{m})^{-mn} ,
%  ~~~~~
%  P
%  \]
%  and we use an iteration $y_{n+1} ~=~ y_n - error(y_n)/error'(y_n)$.
%
%  We also define the 'future value'
%  \[
%  FV( PV, r, m, n )  ~~ = ~~ PV ~ (1 + \frac{r}{m} )^{mn}
%  \]
%  of an amount $PV$ invested today and compounded at annual interest rate $r$
%  $m$ times per year for $n$ years.  Similary we have the 'present value'
%  \[
%  PV( FV, r, m, n )  ~~ = ~~ FV ~ (1 + \frac{r}{m} )^{-mn}.
%  \]
%
%
%  See:
%  F.J. Fabozzi, I.M. Pollack (eds.),
%  {\em The Handbook of Fixed Income Securities} (3rd ed.),
%  Homewood, Illinois: Business One Irwin, 1991.
%
%-------------------------------------------------------------------------------

fv(PV,R,M,N, FV) :-
        powerValue((1+(R/M)),(M*N),Compound),
        FV is PV * Compound.

pv(FV,R,M,N, PV) :-
        powerValue((1+(R/M)),-(M*N),Compound),
        PV is FV * Compound.

yield_from_horizon_value(HV,P,M,N, Y) :-
        %  HV = fv(P,Y,M,N) = P*(1+(Y/M))^(-M*N)
        %  so Y = ((HV/P)^(-1/(M*N)) - 1) * M
        powerValue(HV/P, (-1/(M*N)), HVPP),
        Y is (HVPP-1)*M.


yield(P,C,R,M,N, Y) :- newtonYield(0.0,P,C,R,M,N,Y).


newtonYield(Y0,P,C,R,M,N,Y) :-
        yieldError(Y0,P,C,R,M,N,Error),
        newtonIterate(Y0,P,C,R,M,N,Error,Y).

newtonIterate(Y0,_,_,_,_,_,Error,Y0) :- negligibleError(Error), !.
newtonIterate(Y0,P,C,R,M,N,Error,Y)  :-
        yieldErrorDerivative(Y0,P,C,R,M,N,ErrorDerivative),
        NewY0 is Y0 - Error/ErrorDerivative,
        newtonYield(NewY0,P,C,R,M,N,Y).

yieldError(Y,P,C,R,M,N, Error) :-
        X is (1/(1+Y/M)),
        powerSum(X,(M*N), CPS),
        CouponCashFlow is (C/M)*CPS,
        powerValue(X,(M*N), RP),
        RedemptionCashFlow is R * RP,
        Error is (CouponCashFlow + RedemptionCashFlow) - P.

yieldErrorDerivative(Y,_,C,R,M,N, ErrorDerivative) :-
        X is (1/(1+Y/M)),
        DXDY is (-1/(X*X))*(1/M),  % chain rule
        powerSumDerivative(X,(M*N), PSD),
        CouponCashFlowDerivative is (C/M) * PSD * DXDY,
        powerValue(X,(M*N-1), PX1),
        XPD is (M*N) * PX1,
        RedemptionCashFlowDerivative is R * XPD * DXDY,
        ErrorDerivative is (CouponCashFlowDerivative
                                + RedemptionCashFlowDerivative).
```

```
negligibleError(0.0) :- !.
negligibleError(X) :- X>0.0, !, X < 1.0e-5.
negligibleError(X) :- X > -1.0e-5.


% powerSum/3 computes PS = (x + x^2 + ... + x^N) = x(1 - x^N)/(1 - x)
powerSum(0.0,_,1.0) :- !.
powerSum(1.0,N,N) :- !.
powerSum(X,N,PS) :-
        powerValue(X,N,XN),
        PS is X*(1-XN)/(1-X).

% powerSumDerivative/3 computes
%     d/dx (x + x^2 + ... + x^N)
%   = d/dx (1 + x + x^2 + ... + x^N)
%   = d/dx [(1 - x^(N+1))/(1 - x)]
%   = [(1 - x) (- (N+1) x^N)  +  (1 - x^(N+1))] / (1 - x)^2
powerSumDerivative(0.0,_,1.0) :- !.
powerSumDerivative(1.0,N,NN1) :- !, NN1 is N*(N+1)/2.
powerSumDerivative(X,N,PSD) :-
        powerValue(X,N,XN),
        XN1 is X*XN,
        PSD is (( (1-X)*(-(N+1)*XN) + (1 - XN1) ))/((1-X)*(1-X)).

powerValue(0.0,_,0.0) :- !.
powerValue(X,Y,Power) :-
        XValue is X,
        YValue is Y,
        log(XValue,LogX),
        YLogX is YValue*LogX,
        exp(YLogX,Power).

% yield_to_maturity(scg, 105.00, 91/04/04) =>> 0.07208791346601839

yield_to_maturity_demo  =>  yield_to_maturity(scg, 105.00, 91/04/04).

% Example: yield_to_worst(scg, 105.00, 91/04/04) =>> 0.06281876021279004
% This (6.28 percent) is the yield at the call on 91/11/15.
yield_to_worst_demo  =>  yield_to_worst(scg, 105.00, 91/04/04).

%-------------------------------------------------------------------------
%       Moving averages
%-------------------------------------------------------------------------

moving_average(M,S)  =>  windowAvgs(sequence(S),M,{}).

windowAvgs({},     _,_) => {}.
windowAvgs({X},    M,W) => windowAvg({X},M,W).
windowAvgs(S1<>S2,M,W) => windowAvgs(S1,M,W) <> windowAvgs(S2,M,window(S1,M,W)).

windowAvg({_},M,W)  => {}  :-  count(W) =>> N, N < M-1, !.
windowAvg({X},M,W)  => {A} :-  avg(window({X},M,W)) =>> A.

window({X},M,W)  =>  all_but_first(W<>{X},M1)  :-  count(W) =>> N, M1 is N+1-M.

all_but_first({},     _)  =>  {}.
all_but_first({X},    N)  =>  if (N > 0) then {} else {X}.
all_but_first(S1<>S2,N)  =>  all_but_first(S1,N) <> all_but_first(S2,N1)  :-
                count(S1) =>> C, N1 is N-C.

moving_average_demo  =>
        moving_average(3, {1}<>{4}<>{9}<>{16}<>{25}<>{36}<>{49}<>{64} ).

%-------------------------------------------------------------------------
%       Conservative extensions...
%-------------------------------------------------------------------------

mirrored_append(S,T)  =>  if emptycollection(S)
                                then T
                                else mirror(prepend(mirror(S),mirror(T))).

prepend({},_)       =>  {}.
prepend({X},T)      =>  T <> {X}.
prepend(S1<>S2,T)   =>  prepend(S1,T) <> prepend(S2,{}).

append_demo  =>  mirrored_append( ({1}<>{2})<>({3}<>{4}),  ({5}<>{6})<>{7} ).

%-------------------------------------------------------------------------

X => X :- atomic(X), !.  % atoms and numbers are values, terminating rewriting
X => X :- functor(X,t,_), !.  % all tuples are values.

%-------------------------------------------------------------------------

demos :-
        run_demo(diffs_demo),
        run_demo(polygon_demo),
        run_demo(naive_sequence_demo),
        run_demo(sequence_demo),
        run_demo(mirror_demo),
        run_demo(balance_demo),
        run_demo(partition_demo),
        run_demo(group_demo),
        run_demo(halves_demo),
        run_demo(avg_demo),
        run_demo(sort_demo),
        run_demo(nested_loops_demo),
        run_demo(hash_join_demo),
        run_demo(merge_demo),
        run_demo(union_demo),
        run_demo(difference_demo),
        run_demo(intersection_demo),
        run_demo(yield_to_maturity_demo),
        run_demo(yield_to_worst_demo),
        run_demo(moving_average_demo),
        run_demo(append_demo).

run_demo(D) :-
        nl, write(D), nl,
        (D => Expression),
        Expression =>> Value,
        tab(4), write(Expression), nl,
        tab(4), write(' =>> '), write(Value), nl.

%-------------------------------------------------------------------------
%       Demo output
%
%       Six long lines in the output below were edited into (more readable)
%       multiple lines, but otherwise the output is exactly as produced.
%-------------------------------------------------------------------------
%
%  ?-  demos.
%
%  diffs_demo
%      diffs({98}<>{99}<>{97}<>{97}<>{99}<>{96}<>{[]})
%       =>> {1}<>{-2}<>{0}<>{2}<>{-3}<>{[]}
%
%  polygon_demo
```

```
%     total_area(sample_polygon)
%      => 26.50000000000002
%
%  naive_sequence_demo
%     sequence({{8}<>{{3}<>{1}}<>{{7}<>{5}}<>{9}}<>{4}<>{2})
%      => {8}<>{3}<>{1}<>{7}<>{5}<>{9}<>{4}<>{2}
%
%  sequence_demo
%     sequence({{8}<>{{3}<>{1}}<>{{7}<>{5}}<>{9}}<>{4}<>{2})
%      => {8}<>{3}<>{1}<>{7}<>{5}<>{9}<>{4}<>{2}
%
%  mirror_demo
%     mirror({{1}<>{2}}<>{3}<>{4}<>{5})
%      => {{5}<>{4}}<>{3}}<>{2}<>{1}
%
%  balance_demo
%     balance({a}<>{3}<>{1}<>{b}<>{5}<>{d}<>{4}<>{2})
%      => {{{a}<>{3}}<>{1}<>{b}}<>{{5}<>{d}}<>{4}<>{2}
%
%  partition_demo
%     partition({a}<>{1}<>{2}<>{b}<>{3},numeric)
%      => {{1}<>{2}<>{3}}<>{a}<>{b}
%
%  group_demo
%     group(crude_hash,{a}<>{b}<>{c}<>{d}<>{e}<>{f}<>{g})
%      => {{t(3,{g}<>{c})}<>{t(1,{a}<>{e})}}<>{t(2,{b}<>{f})}}<>{t(0,{d})}
%
%  halves_demo
%     halves({a}<>{3}<>{1}<>{b}<>{5}<>{d}<>{4}<>{2},1,8)
%      => t({a}<>{3}<>{1}<>{b},{5}<>{d}<>{4}<>{2})
%
%  avg_demo
%     avg({8}<>{3}<>{1}<>{7}<>{5}<>{9}<>{4}<>{2}<>{6})
%      => 5.0
%
%  sort_demo
%     sort({8}<>{3}<>{1}<>{7}<>{5}<>{9}<>{4}<>{2},lessEq,avg)
%      => {8}<>{3}<>{1}<>{7}<>{5}<>{9}<>{4}<>{2}
%
%  nested_loops_demo
%     nested_loops_join(1,1,emp,project)
%      => {t(amiel,39-63-59-65,omnis)} <>
%      {t(amsaleg,39-63-52-55,eos)} <>
%      {t(baque,39-63-52-50,sabre)} <>
%      {t(bellosta,39-63-59-65,omnis)} <>
%      {t(cheiney,39-63-52-52,rdl)} <>
%      {t(cirio,39-63-52-50,sabre)} <>
%      {t(darrieumerlou,39-63-55-72,geode)} <>
%      {t(daynes,39-63-52-55,geode)} <>
%      {t(de_maindreville,39-63-56-19,rdl)} <>
%      {{t(gardarin,39-63-56-86,optimisation)} <>
%      {t(gardarin,39-63-56-86,sabre)}} <>
%      {t(gruber,39-63-56-35,eos)} <>
%      {t(kiernan,39-63-56-19,rdl)} <>
%      {t(lanzelotte,39-63-56-32,optimisation)} <>
%      {t(parker,39-63-59-64,svp)} <>
%      {t(pucheral,39-63-55-79,geode)} <>
%      {{t(simon,39-63-52-52,rdl)} <>
%      {t(simon,39-63-52-52,svp)} <>
%      {t(simon,39-63-52-52,sabre)}) <>
%      {t(thevenin,39-63-55-79,geode)} <>
%      {{t(valduriez,39-63-52-51,eos)} <>
%      {t(valduriez,39-63-52-51,geode)} <>
%      {t(valduriez,39-63-52-51,omnis)} <>
%      {t(valduriez,39-63-52-51,optimisation)} <>
%      {t(valduriez,39-63-52-51,sabre)} <>
%      {t(valduriez,39-63-52-51,svp)}) <>
%      {t(viallet,39-63-55-79,omnis)} <>
%      {t(zait,39-63-56-18,optimisation)} <>
%      {t(ziane,39-63-56-32,optimisation)}
%
%  hash_join_demo
%     hash_join(1,1,emp,project)
%      => {{{{{{{{{{{t(ziane,39-63-56-32,optimisation)} <>
%      {t(zait,39-63-56-18,optimisation)}) <>
%      {t(amiel,39-63-59-65,omnis)} <>
%      {t(amsaleg,39-63-52-55,eos)}) <>
%      {t(baque,39-63-52-50,sabre)} <>
%      {t(bellosta,39-63-59-65,omnis)}) <>
%      {t(cheiney,39-63-52-52,rdl)} <>
%      {t(cirio,39-63-52-50,sabre)}) <>
%      {{t(darrieumerlou,39-63-55-72,geode)} <>
%      {t(daynes,39-63-52-55,geode)}) <>
%      {t(de_maindreville,39-63-56-19,rdl)}) <>
%      {{t(gardarin,39-63-56-86,optimisation)} <>
%      {t(gardarin,39-63-56-86,sabre)}) <>
%      {t(gruber,39-63-56-35,eos)}) <>
%      {t(kiernan,39-63-56-19,rdl)}) <>
%      {t(lanzelotte,39-63-56-32,optimisation)}) <>
%      {t(parker,39-63-59-64,svp)} <>
%      {t(pucheral,39-63-55-79,geode)}) <>
%      {{t(simon,39-63-52-52,rdl)} <>
%      {t(simon,39-63-52-52,svp)}) <>
%      {t(simon,39-63-52-52,sabre)}) <>
%      {t(thevenin,39-63-55-79,geode)}) <>
%      {{{{{{t(valduriez,39-63-52-51,eos)} <>
%      {t(valduriez,39-63-52-51,geode)}) <>
%      {t(valduriez,39-63-52-51,omnis)}) <>
%      {t(valduriez,39-63-52-51,optimisation)}) <>
%      {t(valduriez,39-63-52-51,sabre)}) <>
%      {t(valduriez,39-63-52-51,svp)}) <>
%      {t(viallet,39-63-55-79,omnis)}
%
%  merge_demo
%     merge_scan({1}<>{4}<>{8}<>{9},{2}<>{3}<>{5}<>{11})
%      => {1}<>{2}<>{3}<>{4}<>{5}<>{8}<>{9}<>{11}
%
%  union_demo
%     sorted_union({t(a,1)}<>{t(b,2)}<>{t(c,3)}<>{t(e,5)}<>{[]},
%             {t(a,1)}<>{t(c,3)}<>{t(d,4)}<>{[]})
%      => {{t(a,1)}<>{t(b,2)}}<>{{t(c,3)}<>{t(d,4)}}<>{t(e,5)}<>{[]}
%
%  difference_demo
%     sorted_difference({t(a,1)}<>{t(b,2)}<>{t(c,3)}<>{t(e,5)}<>{[]},
%             {t(a,1)}<>{t(c,3)}<>{t(d,4)}<>{[]})
%      => {t(b,2)}<>{t(e,5)}<>{[]}
%
%  intersection_demo
%     sorted_intersection({t(a,1)}<>{t(b,2)}<>{t(c,3)}<>{t(e,5)}<>{[]},
%             {t(a,1)}<>{t(c,3)}<>{t(d,4)}<>{[]})
%      => {t(a,1)}<>{t(c,3)}<>{[]}
%
%  yield_to_maturity_demo
%     yield_to_maturity(scg,105.0,91/4/4)
%      => 0.07208791346601839
%
%  yield_to_worst_demo
%     yield_to_worst(scg,105.0,91/4/4)
```

```
%       =>> 0.06281876021279004
%
%  moving_average_demo
%      moving_average(3,{1}<>{4}<>{9}<>{16}<>{25}<>{36}<>{49}<>{64})
%       =>> {4.666666666666668}<>{9.666666666666664}<>{16.66666666666667}<>
%       {25.66666666666667}<>{36.66666666666666}<>{49.66666666666666}
%
%  append_demo
%      mirrored_append(({1}<>{2})<>{3}<>{4},({5}<>{6})<>{7})
%       =>> ({1}<>{2})<>{3}<>{4}<>{5}<>{6})<>{7}
%
```