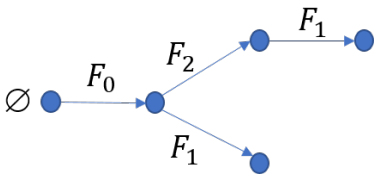
Take Home Final S19

Do your own work. **Do not discuss these problems with anyone except me.** You may ask me questions directly (batory@cs.utexas.edu) but **NOT via Piazza**. Submit a single PDF file with your name and email address in the PDF file itself, so that when I click your email address, an e-mail envelope appears (ex: click my e-address above). Use 1 page per solution, so your PDF file should be 4 pages long. Not following these instructions will cause you to lose points. Not all questions carry the same weight (number of points). I may adjust these weights after grading if some problems are found to be too difficult. Problem 5, as I mentioned in class, has its own purpose and is separate from the final.

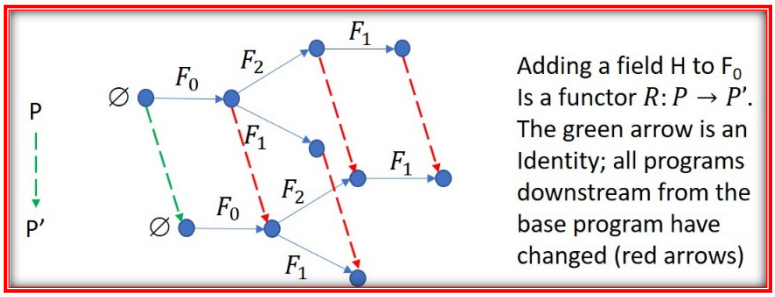
Note: Clarifications on questions are generally free. If you want a hint about an answer, I am willing to provide them. But hints are not free: 4 points per hint.

1. [25pts] Explore the relationship between feature-based product lines and refactorings. Suppose refactoring R renames field H to H'. Let p be a member of product line P. R(p) = p' says R refactors program p into p'. I want you to consider what R(P)=P' means, where P' is a "R-refactored" product line.

   a. Suppose field H is introduced by the base feature of product line P. (The base feature, F_0, is present in all products of P). What is a categorical description of the relationship between product lines P and P'? P can be any product line. Use this product line "graph" in your explanations; circles are programs and arrows are features.



Ans:



   b. Now generalize: Suppose field H is introduced by some non-base feature (say F_2). Generalize your answer to (a) to account for this and justify your answer.

      Nothing really has changed. $R: P \rightarrow P'$ is still a functor as drawn above. Any program downstream of F_2 will change; any program upstream from F_2 will not change. Not much more can be said.

   c. Again, let p be a member of product line P. Suppose p = F_1•F_2•F_0, where • is the feature composition operation. Its refactored counterpart is R(p) = p'. What can you say about R(p) = R(F_1•F_2•F_0)?

      We know that refactorings distribute across feature composition. So R(p) = R(F_1•F_2•F_0) = R(F_1) •R(F_2) • R(F_0)

      What is this relationship that you have deduced in (c) called? There is a general term for it.

      Homomorphism: each expression in P corresponds to an expression in P': $f(x+y) = f(x) + f(y)$

2. [25pts] A market place in component-based software engineering was envisioned years ago where component **binaries** could be purchased as needed by end-users.

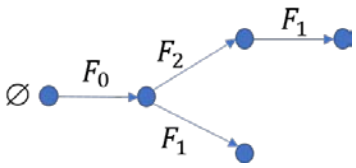
   a. Why were binaries, *not* source code, the key to this vision?

      Ans: don't want to give away trade secrets (or the fact that companies didn't have secrets).

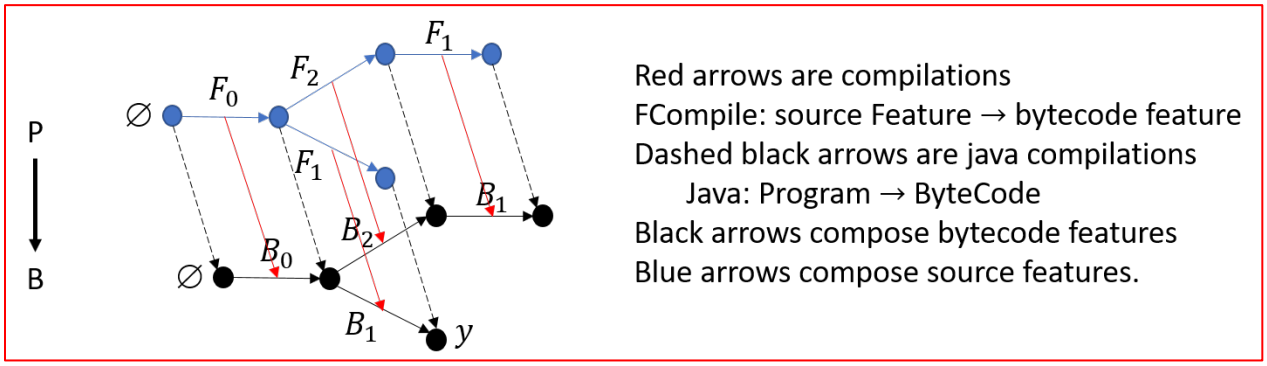
   b. Explain why FeatureHouse (AHEAD, and other SPL tools like it) work as they do, and what technology is ultimately needed to support SPLs in a component-based software engineering market place.

      FH and AHEAD (and other SPL tools) are based on preprocessors that compose code fragments or snippets into larger fragments or snippets. Why? Because doing anything else would be really, really hard. What is this really hard task? compile code fragments and then link binaries together to achieve the same result. There is a name for this in Java – which you might not know, and I don't expect you to know – single class compilation; we need something a bit more than this but it is close.

   c. Express your answer (b) in terms of a 3-dimensional commuting diagram. Hint: use the "diagram" below to illustrate your answer"



      Ans:



   d. Using answer (c) re-explain your answers in (a) and (b) in terms of geodesics.

Choose any program in P' – say it is $y = F_1 \bullet F_0$. To build y in FH or AHEAD you had to compose the source of features F_0 and F_1 (yielding $y_{source} = F_1 \bullet F_0$) and then compile y = javac(ysource). Any other path had infinite cost.

The component-based approach would say: composing source has infinite cost, so traversing blue arrows is verboten. Instead, traverse only dashed black arrows ∅ → ∅, and only black arrows to minimize costs.

3. [35pts] Below is the introduction to "Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting", Conf. Parallel and Distributed Information Systems, Miami, 1991. My (clarifying) comments are in red.

> In this paper we consider the problem of external sorting in a shared-nothing parallel database system. "Shared-nothing" means that the database system is implemented on top of a multi-processor in which each processor has its own local memory and disk, and all communication between processors must take place through an interconnection network. The specific sorting problem we address is "multiple-input multiple-output" sorting. In this sorting problem, initially the data in the file to be sorted is on disk, distributed throughout the multiprocessor, and unsorted. At the termination of the sorting algorithm, the file must again be on disk, but partitioned into approximately equal sized non-overlapping sorted runs, one at each processor.

Hint: The above means that the file (relation) F to sort is horizontally partitioned into unsorted subfiles (relations), F[1]...F[n], where n is the number of processors. The partitioning described above is basically the first k tuples/records of F are in subfile F[1], the next k tuples/records are in subfile F[2], etc, where the total number of records in file F is n*k.

A sequential algorithm for this problem is:

1. Determine a single splitting vector v[i], where $1 \leq i \leq n$ such that in the final sorted order, all records on processor p[1] have a sort key ≤ v[1], all records on processor p[2] have a sort key > v[1] but ≤ v[2], and … all records on processor p[n] have sort key value > v[n-1].

2. Based upon this splitting vector, redistribute the records in each subfile F[i] so that each record of F[i] is sent and stored at the appropriate processor. This is done in parallel for all F[i] where $1 \leq i \leq n$.

3. After redistribution, locally sort the records on each processor to produce the final result.

4. Write the output of each processor to file F'[i] where $1 \leq i \leq n$.

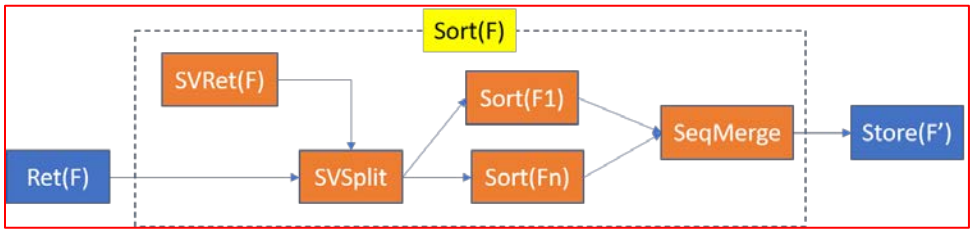
Hint: Not clearly stated above is that a splitting vector sv[i] must be computed for each partition F[i] and that these splitting vectors must be combined into a single splitting vector v[i], described above, which is then used to globally partition F (as described in step 2).

Create a DxT derivation of this algorithm, starting from a sequential description of a sort:

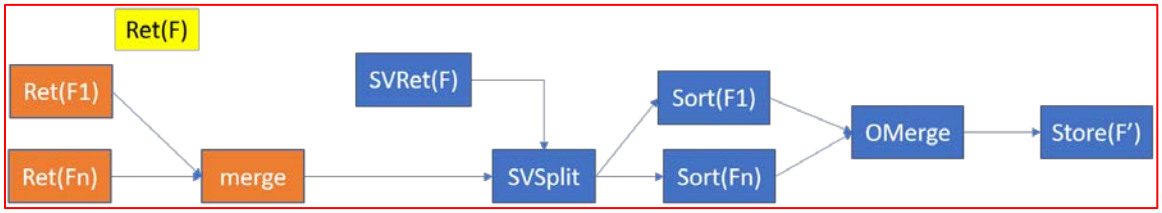
Where F is the file (relational table) to be sorted, F' is its key sorted counterpart. An implicit parameter of the sort is K, an attribute of F. Box ret(F) means retrieve from file F on disk and produce a stream of all of its records. Box store(F') says store its input stream in file F' on disk.

> Remember: A DxT derivation starts with a sequential execution, and using progressive (and easily explainable) dataflow rewrites – in this case, refinements or optimizations; no extensions – to yield a dataflow graph of the final implementation.

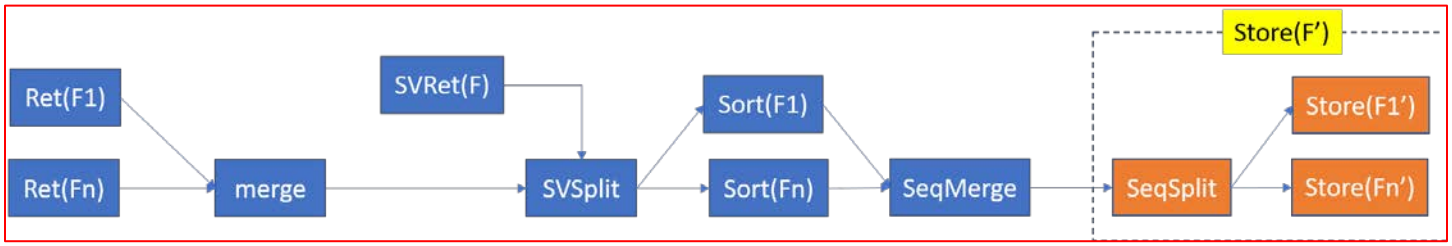
Ans: There is no unique way to express the answer to this question. I will show you one that closely matches the algorithm in the Parallel Sorting text. A first refinement is to expose the core sorting algorithm used. A special operation on F creates a splitting vector, which is then input to a Vector split (SVSplit) box. The stream of F records is partitioned into F1…Fn, which are sorted in parallel, and then a "sequentialMerge" of Sort(F1)|Sort(F2)|…|Sort(Fn) is created, where "|" are markers that indicate the boundaries of F(i) and F(i+1). This merge is then stored in a single file F' (where "|" markers appear to be ignored).



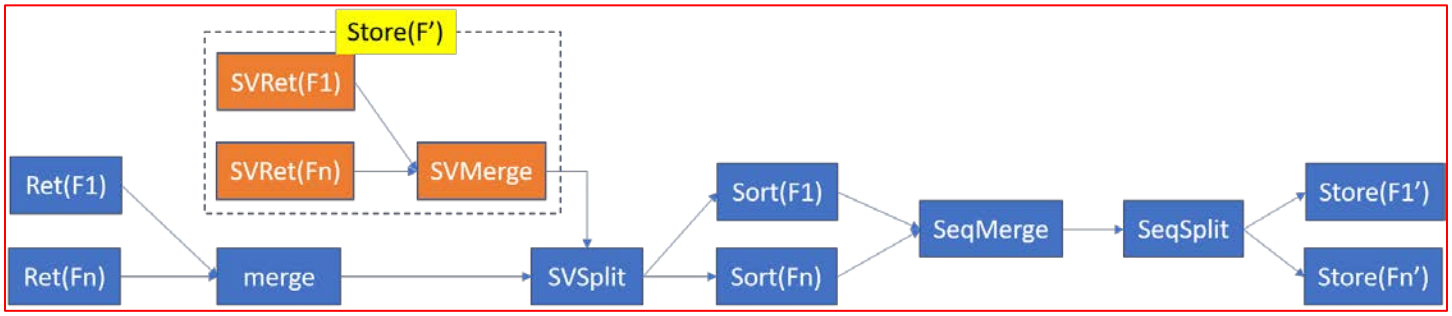
Next, the Ret(F) box is parallelized because F is stored in partitions F1…Fn. To reproduce the retrieval of a single file F requires the merge of their individual retrievals:



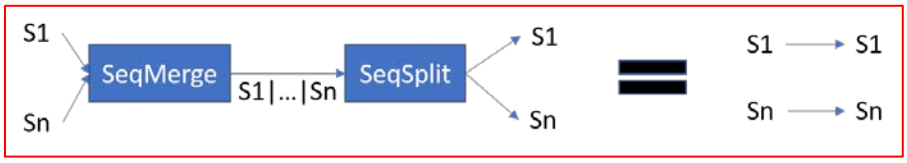
Next, the Store(F') operation is parallelized because each sorted partition of F1'…Fn' is stored separately:



We next have to parallelize the SVRet(F) operation, because there is no file F, but rather subfiles F1…Fn:

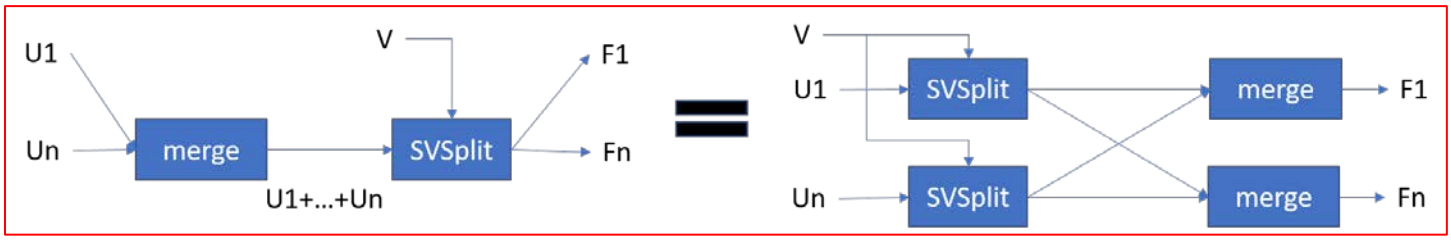


Two optimizations are possible. Here's the simplest: the SeqMerge→ SeqSplit effectively does nothing:



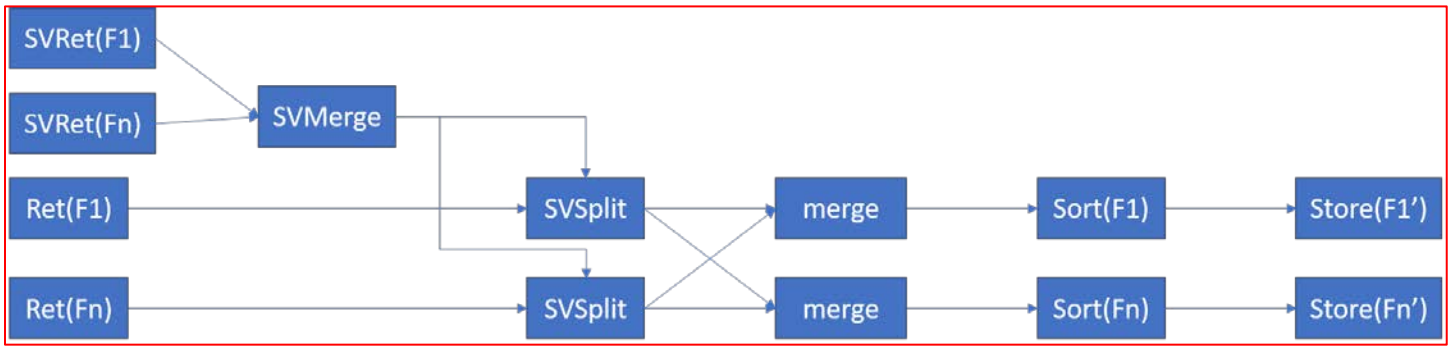
Sorted streams S1…Sn are merged (with markers "|" indicating where S(i) ends and S(i+1) begins. SeqSplit undoes this merging.

The next and last optimization is the most complicated: the merge and SVSPlit are not inverses of each other, but can be rotated for better performance:



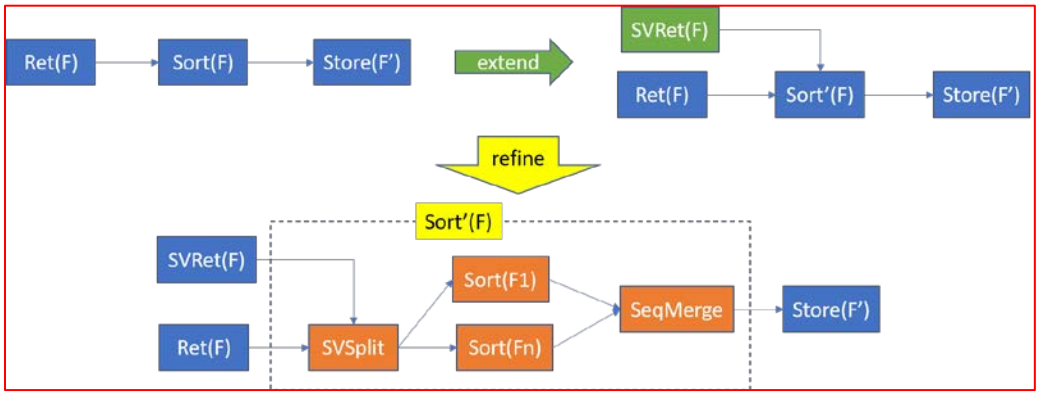
Unsorted streams U1…Un come in to the left, are merged into a single stream U = U1+…+Un and then vector split into streams F1…Fn for later sorting. The same effect is achieved of merge—SVSPlit are rotated, eliminating the sequential bottleneck of U.

Applying both optimizations yields the final design:



The hard part of this design is dealing with two retrieval operations – one to get a splitting vector and another to apply this splitting vector. In the end, it looks simple.

Another approach would be to start with the abstract computation, extend the notion of sort, which introduces the idea of a splitting vector, and you're off to the races:



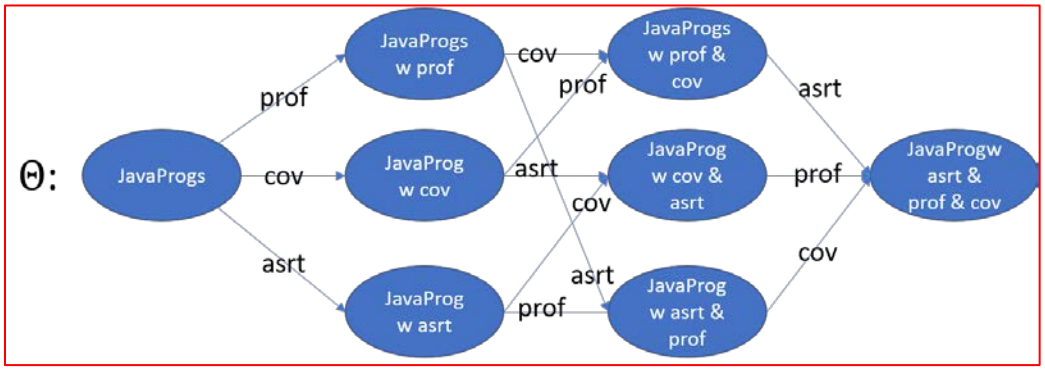
4. [10pts] Today's IDEs offer its users multiple, independent analyses that can be injected into their programs; 0+ analyses are executed during a run of that program. Such analyzes include:
   - **profiling** (a method×time) report on where time is spent in program execution,
   - **coverage** (a report that says for each method, which of its code branches were executed), and
   - **enable assertion** checks (to explicitly check declared preconditions).

   There are other analyses.

   Conceptually this is a small product line θ, but it is different than the (small) product lines you encountered in class and in assignments.

   a) How is it different?

      Typical product lines are expressed as categories whose objects (domains) are individual programs. Product line θ is a category whose nodes represent the domain of Java programs:



   b) How is it related to another topic we discussed in this course?

      arrows between infinite sized domains is model driven engineering

5. Fill out this anonymous survey. Thank you.