HAL open science

Bernstein's Conditions

Paul Feautrier

To cite this version:

Paul Feautrier. Bernstein's Conditions. [Research Report] ENS Lyon. 2011, pp.1-9. hal-01930890

HAL Id: hal-01930890

https://inria.hal.science/hal-01930890v1

Submitted on 22 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Bernstein's Conditions

Paul Feautrier

November 21, 2018

1 Definition

Bersntein's conditions [4] are a simple test for deciding if statements or operations can be interchanged without modifying the program results. The test applies to operations which read and write memory at well defined addresses. If u is an operation, let $\mathcal{M}(u)$ be the set of (adresses of) the memory cells it modifies, and $\mathcal{R}(u)$ the set of cells it reads. Operations u and v can be reordered if:

$$\mathcal{M}(u) \cap \mathcal{M}(v) = \mathcal{M}(u) \cap \mathcal{R}(v) = \mathcal{R}(u) \cap \mathcal{M}(v) = \emptyset \tag{1}$$

If these conditions are met, one says that u and v commute or are independent.

Note that in most languages, each operation writes at most one memory cell: $W(u)$ is a singleton. However, there are exceptions: multiple and parallel assignments, vector assignments among others.

The importance of this result stems from the fact that most program optimizations consist – or at least, involve – moving operations around. For instance, to improve cache performance, one must move all uses of a datum as near as possible to its definition. In parallel programming, if u and v are assigned to different threads or processors, their order of execution may be unpredictable, due to arbitrary decisions of a scheduler or to the presence of competing processes. In this case, if Bernstein's conditions are not met, u and v must be kept in the same thread.

Checking Bernstein's conditions is easy for operations accessing scalars – but beware of aliases –, is more difficult for array accesses, and is almost impossible for pointer dereferencing. See the **Dependences** entry for an in-depth discussion of this question.

2 Discussion

2.1 Notations and Conventions

Is this essay, a program is represented as a sequence of operations, i.e. of instances of high level statements or machine instructions. Such a sequence is called a *trace*. Each operation has a unique name, u, and a text $\mathcal{T}(u)$, usually specified as a (high-level language) statement. There are many schemes for naming operations: for polyhedral programs, one may use integer vectors, and operations are executed in lexicographic order. For flowcharts programs, one may use words of a regular language to name operations, and if the program has function calls, words of a context-free language [2]. In the last two cases, u is executed before v iff u is a prefix of v. In what follows, $u \prec v$ is a shorthand for "u is executed before v". For sequential programs, $\prec$ is a well-founded total order: there is no infinite chain $x_0, x_1, \ldots, x_i, \ldots$ such that $x_{i+1} \prec x_i$. This is equivalent to stipulating that a program execution has a begining, but may not have an end.

All operations will be assumed deterministic: the state of memory after execution of u depends only on $\mathcal{T}(u)$ and on the previous state of memory.

For *static control programs*, one can enumerate the unique trace – or at least describe it – once and for all. One can also consider static control program *families*, where the trace depends on a few parameters which are know at program start time. Lastly, one can consider static control parts of programs or SCoPs. Most of this essay will consider only static control programs.

When applying Bernstein's conditions, one usually considers a reference trace, which comes from the original program, and a candidate trace, which is the result of some optimization or parallelization. The problem is to decide whether the two traces are equivalent, in a sense to be discussed later. Since program equivalence is in general undecidable, one has to restrict the set of admissible transformations. Bernstein's conditions are specially usefull for dealing with operation reordering.

2.2 Commutativity

To prove that Berstein's conditions are sufficient for commutativity, one needs the following facts:

- When an operation u is executed, the only memory cells which may be modified are those whose adresses are in $\mathcal{M}(u)$

```
 x = 0 |          |  x = 0 |          |  x = 0 |
 r1 = x |   --    |  r1 = x |   --    |  r1 = x |   --
   --   | r2 = x  | r1 += 1 |   --    |   --    | r2 = x
r1 += 1 |   --    | x = r1  |   --    | r1 += 1 |   --
   --   | r2 += 2 |   --    | r2 = x  |   --    | r2 += 2
 x = r1 |   --    |   --    | r2 += 2 |   --    | x = r2
   --   | x = r2  |   --    | x = r2  | x = r1  |   --
   --   |   --    |   --    |   --    |   --    |   --
        | x = 2   |         | x = 3   |         | x = 1
  P #1  |  P #2   |  P #1   |  P #2   |  P #1   |  P #2
```

Figure 1: Several possible interleaves of `x=x+1` and `x=x+2`

- The values stored in $\mathcal{M}(u)$ depend only on u and on the values read from $\mathcal{R}(u)$.

  Consider two operations u and v which satisfy (1). Assume that u is executed first. When v is executed later, it finds in $\mathcal{R}(v)$ the same values as if it were executed first, since $\mathcal{M}(u)$ and $\mathcal{R}(v)$ are disjoint. Hence, the values stored in $\mathcal{M}(v)$ are the same, and they do not overwrite the values stored by u, since $\mathcal{M}(u)$ and $\mathcal{M}(v)$ are disjoint. The same reasoning applies if v is executed first.

The fact that u and v do not meet Bernstein's conditions is written $u \perp v$ to indicate that u and v cannot be executed in parallel.

2.3 Atomicity

When dealing with parallel programs, commutativity is not enough for correctness. Consider for instance two operations u and v with $\mathcal{T}(u) =$ `[x = x+1]` and $\mathcal{T}(v) =$ `[x = x+2]`. These two operations commute, since their sequential execution in whatever order is equivalent to a unique operation w such that $\mathcal{T}(w) =$ `[x = x+3]`. However, each one is compiled into a sequence of more elementary machine instructions, which when executed in parallel, may result in x being increased by 1 or 2 or 3 (see Fig. 1, where `r1` and `r2` are processor registers).

Observe that these two operations *do not* satisfy Bernstein's conditions. In contrast, operations that satisfy Bernstein's conditions do not need to be protected by critical sections when run in parallel. The reason is that neither operation modifies the input of the other, and that they write in distinct memory cells. Hence, the stored values do not depend on the order in which the writes are interleaved.

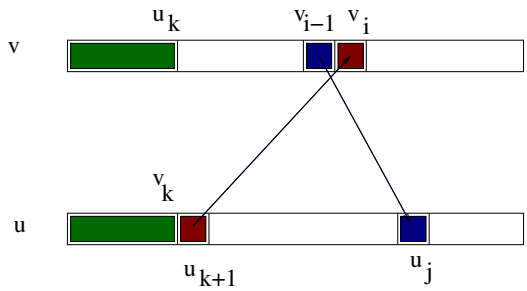

Figure 2: The Commutation Lemma

2.4 Legality

Here, the question is to decide whether a candidate trace is equivalent to a reference trace, where the two traces contains exactly the same operations. There are two possibilities for deciding equivalence. Firstly, if the traces are finite, one may examine the state of memory after their termination. There is equivalence if these two states are identical. Another possibility is to construct the *history* of each memory cell. This is a list of values ordered in time. A new value is appended to the history of x each time an operation u such that $x \in \mathcal{M}(u)$ is executed. Two traces are equivalent if all cells have the same history. This is clearly a stronger criterion than equality of the final memory; it has the advantage of being applicable both to terminating programs and to non-terminating systems. The histories are especially simple when a trace has the *single assignment property*: there is only one operation that writes into x. In that case, each history has only one element.

2.4.1 Terminating Programs

A terminating program is specified by a finite list of operations, $[u_1, \ldots, u_n]$, in order of sequential execution. There is a dependence relation $u_i \to u_j$ iff $i < j$ and $u_i \perp u_j$.

All reorderings of the u: $[v_1, \ldots, v_n]$ such that the execution order of dependent operations is not modified:

$$u_i \to u_j, u_i = v_{i'}, u_j = v_{j'} \Rightarrow i' < j'$$

are legal.

> The proof is by a double induction. Let k be the length of the common prefix of the two programs:
>
> $$u_i = v_i, i = 1, k.$$

Note that k may be null. The element u_{k+1} occurs somewhere among the v, at position $i > k$. The element v_{i-1} occurs among the u at position $j > k+1$ (see Fig. 2). It follows that $u_{k+1} = v_i$ and $u_j = v_{i-1}$ are ordered differently in the two programs, and hence must satisfy Bernstein's condition. v_{i-1} and v_i can therefore be exchanged without modifying the result of the reordered program. Continuing in this way, v_i can be brought in position $k + 1$, which means that the common prefix has been extended one position to the right. This process can be continued until the length of the prefix is n. The two programs are now identical, and the final result of the candidate trace has not been modified.

The property which has just been proved is crucial for program optimization, since it gives a simple test for the legality of statement motion, but what is its import for parallel programming?

The point is that when parallelizing a program, its operations are distributed among several processors or among several threads. Most parallel architectures do not try to combine simultaneous writes to the same memory cell, which are arbitrarily ordered by the bus arbiter or a similar device. It follows that if one is only interested in the final result, each parallel execution is equivalent to some interleave of the several threads of the program. Taking care that operations which do not satisfy Bernstein's condition are excuted in the order specified by the original sequentail program guarantees deterministic execution and equivalence to the sequential program.

2.4.2 Single Assignment Programs

A trace is in single assignment form if, for each memory cell x, there is one and only one operation u such that $x \in \mathcal{M}(u)$. Any trace can be converted to (dynamic) single assignment form – at least in principle – by the following method.

Let A be an (associative) array indexed by the operation names. Assuming that all $\mathcal{M}(u)$ are singletons, operation u now writes into $A[u]$ instead of $\mathcal{M}(u)$. The *source* of cell x at u, noted $\sigma(x, u)$, is defined as:

- $x \in \mathcal{M}(\sigma(x, u))$,
- $\sigma(x, u) \prec u$,
- there is no v such that $\sigma(x, u) \prec v \prec u$ and $x \in \mathcal{M}(v)$.

In words, $\sigma(x, u)$ is the last write to x that precedes u. Now, in the text of u, replace all occurences of $y \in \mathcal{R}(u)$ by $A[\sigma(y, u)]$. That the new trace

has the single assignment property is clear. It is equivalent to the reference trace in the following sense: for each cell x, construct a history by appending the value of $A[u]$ each time an operation u such that $x \in \mathcal{M}(u)$ is executed. Then the histories of a cell in the reference trace and in the single assigment trace are identical.

> Let us say that an operation u has a discrepancy for x if the value assigned to x by u in the reference trace is different from the value of $A[u]$ in the single assignment trace. Let u_0 be the earliest such operation. Since all operations are assumed deterministic, this means that there is a cell $y \in \mathcal{R}(u_0)$ whose value is different from $A[\sigma(y, u_0)]$. Hence $\sigma(y, u_0) \prec u_0$ also has a discrepancy, a contradiction.

Single assignment programs (SAP) where first proposed by Tesler and Enea [9] as a tool for parallel programming. In a SAP, the sets $\mathcal{M}(u) \cap \mathcal{M}(v)$ are always empty, and if there is a non-empty $\mathcal{R}(u) \cap \mathcal{M}(v)$ where $u \prec v$, it means that some variable is read before being assigned, a programming error. Some authors [3] then noticed that a single assignment program is a collection of algebraic equations, which simplifies the construction of correctness proofs.

2.4.3 Non-Terminating Systems

.

The reader may have noticed that the above legality proof depends on the finiteness of the program trace. What happens when one wants to build a non-terminating parallel system, as found for instance in signal processing applications or operating systems? For assessing the correctness of a transformation, one cannot observe the final result, which does not exists. Beside, one clearly needs some fairness hypothesis: it would not do to execute all even numbered operations, *ad infinitum*, and *then* to execute all odd numbered operations, even if Bernstein's conditions would allow it. The needed property is that for all operations u in the reference trace, there is a finite integer n such that u is the n-th operation in the candidate trace.

Consider first the case of two single assignment traces, one of which is the reference trace, the other having been reordered while respecting dependences. Let u be an operation. By the fairness hypothesis, u is present in both traces. Assume that the values written in $A[u]$ by the the two traces are distinct. As above, one can find an operation v such that $A[v]$ is read by u, $A[v]$ has different values in the two traces, and $v \prec u$ in the two traces. One can iterate this process indefinitely, which contradicts the well-foundedness of $\prec$.

Consider now two ordinary traces. After conversion to single assignment, one obtain the same values for the $A[u]$. If one extract an history for each cell x as above, one obtain two identical sequence of values, since operations that write to x are in dependence and hence are ordered in the same direction in the two traces.

Observe that this proof applies also to terminating traces. If the cells of two terminating traces have identical histories, it obviously follows that the final memory states are identical. On the other hand, the proof for terminating traces applies also, in a sequential context, to operations which commutes without satisfying Bernstein's conditions.

3 Dynamic Control Programs

The presence of tests whose outcome cannot be predicted at compile time greatly complicates program analysis. The simplest case is that of well structured programs, which uses only the **if then else** construct. For such programs, a simple syntactical analysis allows the compiler to identify all tests which have an influence on the execution of each operation. One has to take into account three new phenomena:

- A test is an operation in itself, which has a set of read cells, and perhaps a set of modified cells if the source language allows side effects;
- An operation cannot be executed before the outcomes of all controlling tests are known;
- No dependence exists for two operations which belong to opposite branches of a test.

A simple solution, known as *if-conversion* [1], can be used to solve all three problems at once. Each test: **if(e) then ... else ...** is replaced by a new operation **b = e;** where b is a fresh boolean variable. Each operation in the range of the test is guarded by b or $\neg b$, depending on whether the operation is on the **then** or **else** branch of the test. In the case of nested tests, this transformation is applied recursively; the result is that each operation is guarded by a conjunction of the b's or their complements. Bernstein's conditions are then applied to the resulting trace, the b variables being included in the read and modified sets as necessary. This insures that the test is not executed too early, and since there is a dependence from a test to each enclosed operation, that no operation is executed before the tests results are known. One must take care not to compute dependences

between operations u and v which have incompatible guards g_u and g_v such that $g_u \wedge g_v = \textbf{false}$.

The case of **while** loops is more complex. Firstly, the construction **while(true)** is the simplest way of writing a non terminating program, whose analysis has been discussed above. Anything that follows an infinite loop is dead code, and no analysis is needed for it. Consider now a terminating loop:

```
while(p) do S;
```

The several executions of the continuation predicate, `p`, must be considered as operations. Strictly speaking, one cannot execute an instance of `S` before the corresponding instance of `p`, since if the result of `p` is false, `S` is not executed. On the other hand, there must be a dependence from `S` to `p`, since otherwise the loop would not terminate. Hence, a **while** loop must be executed sequentially. The only way out is to run the loop *speculatively*, i.e. to execute instances of the loop body before knowing the outcome of the continuation predicate, but this method is beyond the scope of this essay.

4 Bibliographic Notes and Further Readings

See Allen and Kennedy's book [7] for many uses of the concept of dependence in program optimization.

For more information on the transformation to Single Assignment form, see [5] or [6]. For the use of Single Assignment Programs for hardware synthesis, see [8] or [10].

References

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 177–189. ACM, 1983.

[2] P. Amiranoff, A. Cohen, and P. Feautrier. Beyond iteration vectors: Instancewise relational abstract domains. In *Static Analysis Symposium (SAS'06)*, Seoul, Corea, August 2006.

[3] Jacques Arsac. *La construction de programmes structurés*. Dunod, Paris, 1977.

[4] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15:757–762, 1966.

[5] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[6] Paul Feautrier. Array dataflow analysis. In Pande S. and Agrawal D., editors, *Compiler Optimizations for Scalable Parallel Systems*, LNCS 1808, chapter 6, pages 173–216. Springer, 2001.

[7] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufman, 2001.

[8] Hervé Leverge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[9] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *SJCC*, pages 403–408, 1968.

[10] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. Improved derivation of process networks. In *4th Workshop on Optimization for DSP and Embedded Systems*, pages 1–10, March 2006.