Structured program theorem

The **structured program theorem**, also called **Böhm-Jacopini theorem**,[1][2] is a result in programming language theory. It states that a class of control flow graphs (historically called charts in this context) can compute any computable function if it combines subprograms in only three specific ways (control structures). These are

1. Executing one subprogram, and then another subprogram (sequence)
2. Executing one of two subprograms according to the value of a boolean expression (selection)
3. Executing a subprogram as long as a boolean expression is true (iteration)

The structured chart subject to these constraints may however use additional variables in the form of bits (stored in an extra integer variable in the original proof) in order to keep track of information that the original program represents by the program location. The construction was based on Böhm's programming language P″.

Contents

Origin and variants

The theorem is typically credited[3]:381 to a 1966 paper by Corrado Böhm and Giuseppe Jacopini.[4] David Harel wrote in 1980 that the Böhm–Jacopini paper enjoyed "universal popularity",[3]:381 particularly with proponents of structured programming. Harel also noted that "due to its rather technical style [the 1966 Böhm–Jacopini paper] is apparently more often cited than read in detail"[3]:381 and, after reviewing a large number of papers published up to 1980, Harel argued that the contents of the Böhm–Jacopini proof was usually misrepresented as a folk theorem that essentially contains a simpler result, a result which itself can be traced to the inception of modern computing theory in the papers of von Neumann and Kleene.[3]:383

Harel also writes that the more generic name was proposed by H.D. Mills as "The Structure Theorem" in the early 1970s.[3]:381

Single-while-loop, folk version of the theorem

This version of the theorem replaces all the original program's control flow with a single global `while` loop that simulates a program counter going over all possible labels (flowchart boxes) in the original non-structured program. Harel traced the origin of this folk theorem to two papers marking the beginning of computing. One is the 1946 description of the von Neumann architecture, which explains how a program counter operates in terms of a while loop. Harel notes that the single loop used by the folk version of the structured programming theorem basically just provides operational semantics for the execution of a flowchart on a von Neumann computer.[3]:383 Another, even older source that Harel traced the folk version of the theorem is Stephen Kleene's normal form theorem from 1936.[3]:383

Donald Knuth criticized this form of the proof, which results in pseudocode like the one below, by pointing out that the structure of the original program is completely lost in this transformation.[5]:274 Similarly, Bruce Ian Mills wrote about this approach that "The spirit of block structure is a style, not a language. By simulating a Von Neumann machine, we can produce the behavior of any spaghetti code within the confines of a block-structured language. This does not prevent it from being spaghetti."[6]

```
p := 1;
while p > 0 do begin
  if p = 1 then begin
    perform step 1 from the flowchart;
    p := resulting successor step number of step 1 from the flowchart (0 if no successor);
  end;
  if p = 2 then begin
    perform step 2 from the flowchart;
    p := resulting successor step of step 2 from the flowchart (0 if no successor);
  end;
  ...
  if p = n then begin
    perform step n from the flowchart;
    p := resulting successor step of step n from the flowchart (0 if no successor);
  end;
end.
```

Böhm and Jacopini's proof

The proof in Böhm and Jacopini's paper proceeds by induction on the structure of the flow chart.[3]:381 Because it employed pattern matching in graphs, the proof of Böhm and Jacopini's was not really practical as a program transformation algorithm, and thus opened the door for additional research in this direction.[7]

Implications and refinements

The Böhm-Jacopini proof did not settle the question of whether to adopt structured programming for software development, partly because the construction was more likely to obscure a program than to improve it. On the contrary, it signalled the beginning of the debate. Edsger Dijkstra's famous letter, "Go To Statement Considered Harmful," followed in 1968.[8]

Some academics took a purist approach to the Böhm-Jacopini result and argued that even instructions like `break` and `return` from the middle of loops are bad practice as they are not needed in the Böhm-Jacopini proof, and thus they advocated that all loops should have a single exit point. This purist approach is embodied in the Pascal programming language (designed in 1968–1969), which up to the mid-1990s was the preferred tool for teaching introductory programming classes in academia.[9]

Edward Yourdon notes that in the 1970s there was even philosophical opposition to transforming unstructured programs into structured ones by automated means, based on the argument that one needed to think in structured programming fashion from the get go. The pragmatic counterpoint was that such transformations benefited a large body of existing programs.[10] Among the first proposals for an automated transformation was a 1971 paper by Edward Ashcroft and Zohar Manna.[11]

The direct application of the Böhm-Jacopini theorem may result in additional local variables being introduced in the structured chart, and may also result in some code duplication.[12] The latter issue is called the loop and a half problem in this context.[13] Pascal is affected by both of these problems and according to empirical studies cited by Eric S. Roberts, student programmers had difficulty formulating correct solutions in Pascal for several simple problems, including writing a function for searching an element in an array. A 1980 study by Henry Shapiro cited by Roberts found that using only the Pascal-provided control structures, the correct solution was given by only 20% of the subjects, while no subject wrote incorrect code for this problem if allowed to write a return from the middle of a loop.[9]

In 1973, S. Rao Kosaraju proved that it's possible to avoid adding additional variables in structured programming, as long as arbitrary-depth, multi-level breaks from loops are allowed.[1][14] Furthermore, Kosaraju proved that a strict hierarchy of programs exists, nowadays called the *Kosaraju hierarchy*, in that for every integer n, there exists a program containing a multi-level break of depth n that cannot be rewritten as program with multi-level breaks of depth less than n (without introducing additional variables).[1]

Kosaraju cites the multi-level break construct to the BLISS programming language. The multi-level breaks, in the form a `leave label` keyword were actually introduced in the BLISS-11 version of that language; the original BLISS only had single-level breaks. The BLISS family of languages didn't provide an unrestricted goto. The Java programming language would later follow this approach as well.[15]:960–965

A simpler result from Kosaraju's paper is that a program is reducible to a structured program (without adding variables) if and only if it does not contain a loop with two distinct exits. Reducibility was defined by Kosaraju, loosely speaking, as computing the same function and using the same "primitive actions" and predicates as the original program, but possibly using different control flow structures. (This is a narrower notion of reducibility than what Böhm-Jacopini used.) Inspired by this result, in section VI of his highly-cited paper that introduced the notion of cyclomatic complexity, Thomas J. McCabe described an analogue of Kuratowski's theorem for the control flow graphs (CFG) of non-structured programs, which is to say, the minimal subgraphs that make the CFG of a program non-structured. These subgraphs have a very good description in natural language. They are:

1. branching out of a loop (other than from the loop cycle test)
2. branching into a loop
3. branching into a decision (i.e. into an if "branch")
4. branching out of a decision

McCabe actually found that these four graphs are not independent when appearing as subgraphs, meaning that a necessary and sufficient condition for a program to be non-structured is for its CFG to have as subgraph one of any subset of three of these four graphs. He also found that if a non-structured program contains one of these four sub-graphs, it must contain another distinct one from the set of four. This latter result helps explain how the control flow of non-structured program becomes entangled in what is popularly called "spaghetti code". McCabe also devised a numerical measure that, given an arbitrary program, quantifies how far off it is from the ideal of being a structured program; McCabe called his measure essential complexity.[16]

McCabe's characterization of the forbidden graphs for structured programming can be considered incomplete, at least if the Dijkstra's D structures are considered the building blocks.[17]:274–275

Up to 1990 there were quite a few proposed methods for eliminating gotos from existing program, while preserving most of their structure. The various approaches to this problem also proposed several notions of equivalence, which are stricter than simply Turing equivalence, in order to avoid output like the folk theorem discussed above. The strictness of the chosen notion of equivalence dictates the minimal set of control flow structures needed. The 1988 JACM paper by Lyle Ramshaw surveys the field up to that point, as well proposing its own method.[18] Ramshaw's algorithm was used for example in some Java decompilers because the Java virtual machine code has branch instructions with targets expressed as offsets, but the high-level Java language only has multi-level `break` and `continue` statements.[19][20][21] Ammarguellat (1992) proposed a transformation method that goes back to enforcing single-exit.[7]

Application to Cobol

In the 1980s IBM researcher Harlan Mills oversaw the development of the COBOL Structuring Facility, which applied a structuring algorithm to COBOL code. Mills's transformation involved the following steps for each procedure.

1. Identify the basic blocks in the procedure.
2. Assign a unique label to each block's entry path, and label each block's exit paths with the labels of the entry paths they connect to. Use 0 for return from the procedure and 1 for the procedure's entry path.
3. Break the procedure into its basic blocks.
4. For each block that is the destination of only one exit path, reconnect that block to that exit path.
5. Declare a new variable in the procedure (called L for reference).
6. On each remaining unconnected exit path, add a statement that sets L to the label value on that path.
7. Combine the resulting programs into a selection statement that executes the program with the entry path label indicated by L
8. Construct a loop that executes this selection statement as long as L is not 0.
9. Construct a sequence that initializes L to 1 and executes the loop.

Note that this construction can be improved by converting some cases of the selection statement into subprocedures.

See also

- Structured programming
- Turing completeness

References

1. Dexter Kozen and Wei-Lung Dustin Tseng (2008). "The Böhm–Jacopini Theorem Is False, Propositionally" (http://www.cs.cornell.edu/~kozen/papers/bohmjacopini.pdf) (PDF). *Mpc 2008*. Lecture Notes in Computer Science. **5133**: 177–192. ISBN 978-3-540-70593-2. doi:10.1007/978-3-540-70594-9_11 (https://doi.org/10.1007%2F978-3-540-70594-9_11).
2. "CSE 111, Fall 2004, BOEHM-JACOPINI THEOREM" (http://www.cse.buffalo.edu/~rapaport/111F04/greatidea3.html). Cse.buffalo.edu. 2004-11-22. Retrieved 2013-08-24.
3. Harel, David (1980). "On Folk Theorems" (http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/OnFolkTheorems.pdf) (PDF). *Communications of the ACM*. **23** (7): 379–389. doi:10.1145/358886.358892 (https://doi.org/10.1145%2F358886.358892).
4. Bohm, Corrado; Giuseppe Jacopini (May 1966). "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". *Communications of the ACM*. **9** (5): 366–371. doi:10.1145/355592.365646 (https://doi.org/10.1145%2F355592.365646).
5. Donald Knuth (1974). "Structured Programming with go to Statements". *Computing Surveys*. **6** (4): 261–301. doi:10.1145/356635.356640 (https://doi.org/10.1145%2F356635.356640).
6. Bruce Ian Mills (2005). *Theoretical Introduction to Programming*. Springer. p. 279. ISBN 978-1-84628-263-8.
7. Ammarguellat, Z. (1992). "A control-flow normalization algorithm and its complexity". *IEEE Transactions on Software Engineering*. **18** (3): 237–251. doi:10.1109/32.126773 (https://doi.org/10.1109%2F32.126773).
8. Dijkstra, Edsger (1968). "Go To Statement Considered Harmful" (http://www.acm.org/classics/oct95/). *Communications of the ACM*. **11** (3): 147–148. doi:10.1145/362929.362947 (https://doi.org/10.1145%2F362929.362947).
9. Roberts, E. [1995] "Loop Exits and Structured Programming: Reopening the Debate (http://cs.stanford.edu/people/eroberts/papers/SIGCSE-1995/LoopExits.pdf)," ACM SIGCSE Bulletin, (27)1: 268–272.
10. E. N. Yourdon (1979). *Classics in Software Engineering*. Yourdon Press. pp. 49–50. ISBN 978-0-917072-14-7.
11. Ashcroft, Edward; Zohar Manna (1971). "The translation of go to programs to 'while' programs". *Proceedings of IFIP Congress*. The paper, which is difficult to obtain in the original conference proceedings due to their limited distribution, was republished in Yourdon's 1979 book pp. 51-65
12. David Anthony Watt; William Findlay (2004). *Programming language design concepts*. John Wiley & Sons. p. 228. ISBN 978-0-470-85320-7.
13. Kenneth C. Louden; Kenneth A. Lambert (2011). *Programming Languages: Principles and Practices* (3 ed.). Cengage Learning. pp. 422–423. ISBN 1-111-52941-8.
14. KOSARAJU, S. RAO. "Analysis of structured programs," Proc. Fifth Annual ACM Syrup. Theory of Computing, (May 1973), 240-252; also in J. Computer and System Sciences, 9, 3 (December 1974), doi: 10.1016/S0022-0000(74)80043-7 (https://doi.org/10.1016%2FS0022-0000%2874%2980043-7) cited by Donald Knuth (1974). "Structured Programming with go to Statements". *Computing Surveys*. **6** (4): 261–301. doi:10.1145/356635.356640 (https://doi.org/10.1145%2F356635.356640).
15. Brender, Ronald F. (2002). "The BLISS programming language: a history" (http://www.cs.tufts.edu/~nr/cs257/archive/ronald-brender/bliss.pdf) (PDF). *Software: Practice and Experience*. **32** (10): 955–981. doi:10.1002/spe.470 (https://doi.org/10.1002%2Fspe.470).
16. The original paper is Thomas J. McCabe (December 1976). "A Complexity Measure" (https://books.google.com/books?id=vtNWAAAAMAAJ&pg=PA3). *IEEE Transactions on Software Engineering* (4): 315–318. doi:10.1109/tse.1976.233837 (https://doi.org/10.1109%2Ftse.1976.233837). For a secondary exposition see Paul C. Jorgensen (2002). *Software Testing: A Craftsman's Approach, Second Edition* (https://books.google.com/books?id=Yph_AwAAQBAJ&pg=PA150) (2nd ed.). CRC Press. pp. 150–153. ISBN 978-0-8493-0809-3.
17. Williams, M. H. (1983). "Flowchart Schemata and the Problem of Nomenclature". *The Computer Journal*. **26** (3): 270–276. doi:10.1093/comjnl/26.3.270 (https://doi.org/10.1093%2Fcomjnl%2F26.3.270).

18. Ramshaw, L. (1988). "Eliminating go to's while preserving program structure". *Journal of the ACM*. **35** (4): 893–920. doi:10.1145/48014.48021 (https://doi.org/10.1145%2F48014.48021).
19. Godfrey Nolan (2004). *Decompiling Java*. Apress. p. 142. ISBN 978-1-4302-0739-9.
20. https://www.usenix.org/legacy/publications/library/proceedings/coots97/full_papers/proebsting2/proebsting2.pdf
21. http://www.openjit.org/publications/pro1999-06/decompiler-pro-199906.pdf

Further reading

Material not yet covered above:

- DeMillo, Richard A. (1980). "Space-Time Trade-Offs in Structured Programming: An Improved Combinatorial Embedding Theorem". *Journal of the ACM*. **27** (1): 123–127. doi:10.1145/322169.322180 (https://doi.org/10.1145%2F322169.322180).
- Devienne, Philippe (1994). "One binary horn clause is enough". *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. **775**: 19–32. ISBN 978-3-540-57785-0. doi:10.1007/3-540-57785-8_128 (https://doi.org/10.1007%2F3-540-57785-8_128).

External links

- http://www.cs.uwlax.edu/~riley/CS421/lect8_boehm.ppt a slightly more detailed explanation of the construction used in the folk theorem's proof, with a concrete example of transformed program

---