Deterministic algorithm

In computer science, a **deterministic algorithm** is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. Deterministic algorithms are by far the most studied and familiar kind of algorithm, as well as one of the most practical, since they can be run on real machines efficiently.

Formally, a deterministic algorithm computes a mathematical function; a function has a unique value for any input in its domain, and the algorithm is a process that produces this particular value as output.

Contents

Formal definition

Deterministic algorithms can be defined in terms of a state machine: a *state* describes what a machine is doing at a particular instant in time. State machines pass in a discrete manner from one state to another. Just after we enter the input, the machine is in its *initial state* or *start state*. If the machine is deterministic, this means that from this point onwards, its current state determines what its next state will be; its course through the set of states is predetermined. Note that a machine can be deterministic and still never stop or finish, and therefore fail to deliver a result.

Examples of particular abstract machines which are deterministic include the deterministic Turing machine and deterministic finite automaton.

What makes algorithms non-deterministic?

A variety of factors can cause an algorithm to behave in a way which is not deterministic, or non-deterministic:

- If it uses external state other than the input, such as user input, a global variable, a hardware timer value, a random value, or stored disk data.
- If it operates in a way that is timing-sensitive, for example if it has multiple processors writing to the same data at the same time. In this case, the precise order in which each processor writes its data will affect the result.

- If a hardware error causes its state to change in an unexpected way.

Although real programs are rarely purely deterministic, it is easier for humans as well as other programs to reason about programs that are. For this reason, most programming languages and especially functional programming languages make an effort to prevent the above events from happening except under controlled conditions.

The prevalence of multi-core processors has resulted in a surge of interest in determinism in parallel programming and challenges of non-determinism have been well documented.[1][2] A number of tools to help deal with the challenges have been proposed[3][4][5][6] to deal with deadlocks and race conditions.

Disadvantages of Determinism

It is advantageous, in some cases, for a program to exhibit nondeterministic behavior. The behavior of a card shuffling program used in a game of blackjack, for example, should not be predictable by players — even if the source code of the program is visible. The use of a pseudorandom number generator is often not sufficient to ensure that players are unable to predict the outcome of a shuffle. A clever gambler might guess precisely the numbers the generator will choose and so determine the entire contents of the deck ahead of time, allowing him to cheat; for example, the Software Security Group at Reliable Software Technologies was able to do this for an implementation of Texas Hold 'em Poker that is distributed by ASF Software, Inc, allowing them to consistently predict the outcome of hands ahead of time.[7] These problems can be avoided, in part, through the use of a cryptographically secure pseudo-random number generator, but it is still necessary for an unpredictable random seed to be used to initialize the generator. For this purpose a source of nondeterminism is required, such as that provided by a hardware random number generator.

Note that a negative answer to the P=NP problem would not imply that programs with nondeterministic output are theoretically more powerful than those with deterministic output. The complexity class NP (complexity) can be defined without any reference to nondeterminism using the verifier-based definition.

Determinism categories in languages

Mercury

This logic-functional programming language establish different determinism categories for predicate modes as explained in the ref.[8][9]

Haskell

Haskell provides several mechanisms:

non-determinism or notion of Fail

- the *Maybe* and *Either* types include the notion of success in the result.
- the *fail* method of the class Monad, may be used to signal *fail* as exception.
- the Maybe monad and MaybeT monad transformer provide for failed computations (stop the computation sequence and return Nothing)[10]

determinism/non-det with multiple solutions

you may retrieve all possible outcomes of a multiple result computation, by wrapping its result type in a MonadPlus monad. (its method *mzero* makes an outcome fail and *mplus* collects the successful results).[11]

ML family and derived languages

As seen in Standard ML, OCaml and Scala

- The *option* type includes the notion of success.

Java

- The *null* reference value may represent an unsuccessful (out-of-domain) result.

References

1. Edward A. Lee. "The Problem with Threads" (http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf) (PDF). Retrieved 2009-05-29.
2. Bocchino Jr., Robert L.; Adve, Vikram S.; Adve, Sarita V.; Snir, Marc (2009). *Parallel Programming Must Be Deterministic by Default* (https://www.usenix.org/legacy/event/hotpar09/tech/full_papers/bocchino/bocchino_html/). USENIX Workshop on Hot Topics in Parallelism.
3. "Intel Parallel Inspector Thread Checker" (http://software.intel.com/en-us/videos/intel-parallel-inspector-thread-checker/). Retrieved 2009-05-29.
4. Yuan Lin. "Data Race and Deadlock Detection with Sun Studio Thread Analyzer" (http://developers.sun.com/sunstudio/documentation/product/sd_west_threadAnalyzer.pdf) (PDF). Retrieved 2009-05-29.
5. Intel. "Intel Parallel Inspector" (http://software.intel.com/en-us/intel-parallel-inspector). Retrieved 2009-05-29.
6. David Worthington. "Intel addresses development life cycle with Parallel Studio" (https://web.archive.org/web/20090528030044/http://www.sdtimes.com/link/33497). Archived from the original (http://sdtimes.com/link/33497) on 2009-05-28. Retrieved 2009-05-26.
7. Gary McGraw and John Viega. Make your software behave: Playing the numbers: How to cheat in online gambling. http://www.ibm.com/developerworks/library/s-playing/#h4
8. "Determinism categories in the Mercury programming language" (https://web.archive.org/web/20120703001434/http://www.mercury.csse.unimelb.edu.au/information/doc-release/mercury_ref/Determinism-categories.html#Determinism-categories). Archived from the original (http://www.mercury.csse.unimelb.edu.au/information/doc-release/mercury_ref/Determinism-categories.html) on 2012-07-03. Retrieved 2013-02-23.
9. "Mercury predicate modes" (https://web.archive.org/web/20120703001411/http://www.mercury.csse.unimelb.edu.au/information/doc-release/mercury_ref/Predicate-and-function-mode-declarations.html#Predicate-and-function-mode-declarations). Archived from the original (http://www.mercury.csse.unimelb.edu.au/information/doc-release/mercury_ref/Predicate-and-function-mode-declarations.html) on 2012-07-03. Retrieved 2013-02-25.
10. Representing failure using the Maybe monad (http://www.haskell.org/haskellwiki/Monad#Common_monads)
11. The class MonadPlus (http://www.haskell.org/haskellwiki/MonadPlus)