

「(プログラム実行状況についての)論理的表明」

アントニー・ホーア卿

「表明」とは、プログラムテキスト中に記述された論理式で、プログラムのその部分が実行される時点では常に「真」になるとプログラマーが主張するものです。プログラム内での表明とその前後の部分との内部インターフェースの仕様は、表明によって記述されています。今日のソフトウェア産業では、プログラムをテストランする際に表明を条件付きでコンパイルして、誤りの検出や診断に役立てています。表明を最初に提案したのは Alan Turing です。彼は、大きなルーチンを検証する方法としてこれを提案しましたが、その後、Naur が一般化スナップショットとして、Floyd がプログラムに意味づけをする方法として、それぞれ独自に再発見を行いました。Floyd は、内部表明が十分に強力であれば、プログラム全体の正当性の形式的証明になると示唆しました。今回の講演では、このアイデアのその後の進展に関する概要を述べ、また、実用面を与えた影響について言及します。

70 年代前半に、私はプログラムを検証する公理的手法を開発しました。この手法は、反復、局所変数、手続きとパラメータ、再帰、規則的飛び越しまで、高級プログラミング言語の主要な構成要素すべてに対応していました。Dijkstra の考え方に習って、私は常にソフトウェア構築の作業をトップダウン的視点で捉え、表明はプログラム仕様の一部として定式化するもの、検証はプログラム設計の一部として処理するものと考えていました。私はこの研究が、コスト高の要因となるプログラミングミスや、極めて重要な応用分野でコンピュータを利用する際の高いリスクを低減するのに役立つものと期待していました。しかし、それにも増して魅力的だったのは、公理に基づいたプログラム検証を採用すれば、プログラミング言語の設計品質を客観的、科学的に判定できるということでした。少数の明白な規則で記述され、容易に適用できる言語の方が、多数の規則と複雑な副次的条件を必要とする言語より優れていると言えます。私は Wirth と協力して、このアイデアを Pascal で試し、それが後に Xerox 社の PARC(パロアルト研究センター)にいたチームが Euclid を設計する際の着想の元となりました。

検証の手法を小規模の順序アルゴリズムに対応したのから大規模なソフトウェアシステムに合わせたものに拡大するには、表明言語の能力を拡張する必要がありました。Zermelo の集合論を基に Abrial が開発した仕様言語の Z は、数学で知られているすべての概念を表現するのに本質的に妥当であることが Frankel によって示されました。従って、コンピュータ処理に役立つ

つ全抽象概念に対し妥当な表現ができ、その表現の正当性についても妥当な証明ができるはずで
す。Dijkstra は、悪魔が意地悪く選択を行使するという比喻を使って、非決定性を扱いました。
Jones は、他の VDM 設計者たちと共に、プログラム変数の初期値と終値を考慮に入れました。
こうしたアイデアはすべて、IBM が大規模システムである CICS の内部インターフェースの仕
様を決定する際に、その有効性が実証されました。

次の課題は、この技術を並行プログラムへ拡張することでした。Milner は、プログラムの意
味は、合格した検査のリストから決定できると考えました。Popper の誤り立証可能性判定基準
を基に、Roscoe と Brookes は、ある検査についてその欠陥に注目し、並行性に対応した非決定
モデルを構築しました。これは、相互通信逐次プロセス系(CSP)のパラダイムに習ったものでし
た。このモデルは、創業間もないイギリスの半導体メーカーの Inmos 社が産業用に応用し、プ
ログラミング言語 occam の設計と、この言語を実行する同社のトランスピュータのアーキテク
チャに使われました。最後に、Hehner は、Roscoe の成果を、表明を使用した言語で直接コー
ディングする方法を示し、並行、順序を問わずどんな形式のプログラムでも、可能な動作すべて
を記述する最も強い表明に変換できることを示しました。

この洞察が、後に私が行ったすべての研究のヒントとなりました。He Jifeng の助力を得て、
ハードウェアとソフトウェア、並列と順序、宣言形と手続き形などさらに広範囲のプログラミン
グパラダイムやプログラミング言語にこうした考え方を応用してきました。文法上の根本的な違
いを無視し、運用上の技術から抽象化すると、非常に似通った定義や数学的法則を多くの異なっ
たパラダイムに適用することができます。電子計算学はすでにあるレベルの成熟期に達していて、
他の多くの科学的分野の発達をも促すであろうプログラミング理論の統一という課題に取り組
むべき段階にあるのかもしれない。

“Assertions”

Professor Sir Antony Hoare

An assertion is a Boolean formula written in the text of a program, which the programmer asserts will always be true when that part of the program is executed. It specifies an internal interface between all of the program that comes before it and all that follows it. In the software industry today, assertions are conditionally compiled in test runs of a program, and help in the detection and diagnosis of errors. Alan Turing first proposed assertions as a means of checking a large routine. They were rediscovered independently by Naur as generalised snapshots, and by Floyd, who used them to assign meanings to programs. Floyd suggested that if the internal assertions were strong enough, they would constitute a formal proof of the correctness of a complete program. In this lecture, I will summarise the subsequent development of the idea, and describe some of its practical impact.

In the early seventies, I developed an axiomatic approach for proofs of programs, covering all the main constructions of a high-level programming language-iterations, local variables, procedures and parameters, recursion and even jumps. Following Dijkstra, I always took a top-down view of the task of software construction, with assertions formulated as part of program specification, and with proofs conducted as part of program design. I hoped that this research would help to reduce the high costs of programming error, and the high risks of using computers in critical applications. But the real attraction for me was that the axioms underlying program proofs would provide an objective and scientific test of the quality of programming language design: a language described by a small collection of obvious rules, easily applied, would be better than one that required many rules with complex side-conditions. In collaboration with Wirth, we tried out the idea on the Pascal language; and later it inspired the design of Euclid by a team in Xerox PARC.

In scaling proof methods from small sequential algorithms to large software systems, it was necessary to extend the power of the assertion language. The Z specification language was developed by Abrial on the basis of Zermelo's set theory, which Frankel showed to be essentially adequate for expression of all concepts known to mathematics. It should therefore be adequate to express all the abstractions useful to computing, and prove the correctness of their representations. Dijkstra dealt with non-determinism, by imagining the choice to be exercised maliciously by a

demon. Jones and his fellow designers of VDM included initial as well as final values of program variables. All these ideas were successfully tested by IBM in specifying the internal interfaces of a large system, CICS.

The next challenge was to extend the technology to concurrent programs. Milner suggested that their meaning could be specified by the collection of tests which they passed. Following Popper's criterion of falsifiability, Roscoe and Brookes concentrated on failures of a test, and constructed a non-deterministic model of concurrency, following the paradigm of Communicating Sequential Processes. This was applied industrially by the British start-up microchip Company Inmos in the design of the occam programming language, and the architecture of the transputer which implemented it. Finally, Hehner showed how Roscoe's results could be coded directly in the language of assertions, so that any kind of program, concurrent as well as sequential, could be interpreted as the strongest assertion that describes all its possible behaviours.

This insight has inspired all my subsequent research. With the aid of He Jifeng, it has been applied to wider varieties of programming paradigm and language, including hardware and software, parallel and sequential, declarative and procedural. Ignoring radical differences in syntax, and abstracting from implementation technology, very similar definitions and mathematical laws apply in many different paradigms; perhaps Computing Science has achieved a level of maturity to undertake the challenge that drives the progress of many other sciences, namely unification of theories of programming.