** Context **

In many early programming systems, it was specific compilers (and interpreters) that determined the meaning of programs. Since computer processors were rapidly changing and since compilers map high-level languages to these evolving processors, compilers needed to evolve in order to exploit new processor architectures. Since the new compilers did not commit to preserving the same execution behavior of programs as earlier compilers, the meaning of programs would also change. For the many people writing high-level code, the fact that their code could break when moving it between computer systems or to higher version numbers eventually became a serious problem. This situation became untenable when programs also grew dependent on the services---such as memory management, file systems, and network access---offered by operating systems: now programs could also break whenever there were changes to operating systems.

Early efforts to formalize the meaning of programs employed the computation-as-model paradigm [computing happens in machines and logic is used to describe what happens]. [Illustrate with Hoare triples] While this approach to reasoning about the meaning of programs has had some success and is used in several existing systems today, it has also had some significant failures. In fact, the topic of model checking, in which the search for counterexamples (bugs) replaced the search for formal proofs, arose from frustration that it was too difficult to use pre- and post-condition reasoning in many systems, particularly, those that had elements of distributed and concurrent execution.

Other mathematical frameworks for specifying the meaning of programming languages were given by denotational semantics and operational semantics.

** Premise **

Still another approach to providing a formal semantics for a programming language is to base that language directly on a formal structure that already has a mathematical and well-understood semantics.

** An outline of how this premise can be developed **

* What is a formal structure?

  Generally a mathematically described formal structure.

  Not all formal structures are, however, of a quality useful for designing a programming language.

  - Turing machines are formally defined objects. They have played an important role in computer science for at least two reasons.
    - They obviously capture the notion of computation. (Goedel did not feel that Church's lambda-calculus captured computing but he did feel that Turing's machines did.) Capturing computing in a convincing manner is important for establishing a negative result.
    - TMs have been useful for providing formal definitions of the time and space hierarchies.

  - Turing machines are not a good starting place for programming languages. They are too low-level, admit almost no abstractions, they are difficult to make modular, etc.

* Formal Structures for programming languages

    The formal structures used in designing programming languages
    usually have multiple ways to characterize central notions.

    E.g: First-order logic: model theory (truth) = theorems (proofs).
    These have equivalent characterizations using resolution, natural
    deduction, sequent calculus, etc.

    E.g: Typed Lambda-calculus is a form of computing (a la Church)
    and is a form of proof (natural deduction). Beta-reduction
    corresponds to proof normalization. Curry-Howard isomorphism.

  - Also, formal structures have deep results.

    Boehn separation result for the lambda-calculus. Confluence.

    Classical logic has Herbrand's theorem. Classical,
    intuitionistic, and linear logics have cut-elimination rules.

    pi-calculus has various equivalences defined using observations
    that can be shown to be congruences.

* Example of formal structures
  These have been used within the programming language world.

  - lambda-calculus: functional programming
  - sequent calculus, resolution: logic programming
  - finite state machines, regular languages: tokenizers, search queries
  - context-free languages, push down automata: parsers, attribute grammers,
  - CSP; Occam
  - CCS, pi-calculus, spi-calculus: concurrent processes, security protocols
  - relational calculus; datalog, database programming
  - Petri nets: concurrent processes

* Benefits of using formal structures

  Using such formal structures to organize computation (instead of
  generalizing from Turing machines, assembly language, etc) is that
  they generally introduce a lot of interesting, new ideas.

  - E.g: recursion, pattern matching, polymorphic typing, ...

  - E.g: "logic variables", nondeterministic programming, relational
    programming, ...

  - E.g: asynchronous/synchronous communications, distributed bindings
    (scope extrusions),

  More importantly, however, is that the deep properties of a formal
  structure can help in direct reasoning on the actual artifact that
  is a computer program.

  E.g: programs can be unfolded, partially evaluated,

  E.g: Rich static properties can be described: these static checks
  have proved invaluable to programmers and compilers alike.

* Problems with using formal structures

  There is usually a tension between the needs of programming and
  solutions offered by formal systems.

  E.g: classical and intuitionistic logics do not naturally allow one model change (linear logic can help).

E.g: since formal systems are often conceived separately from modern computer hardware, there might significant problems getting good performance from such high-level languages.

E.g: etc.