Fundamentele limbajelor de programare

CURS 0

Traian Florin Șerbănuță și Andrei Sipoș

Facultatea de Matematică și Informatică, DL Info, Anul II

Semestrul II, 2024/2025

Motto

"In physics, math, and computer science, the state of a system is an encapsulation of all the information you'd ever need to predict what it will do, or at least its probabilities to do one thing versus another, in response to any possible prodding of it. In a sense, then, the state is a system's 'hidden reality', which determines its behaviour beneath surface appearances. But, in another sense, there's **nothing** *hidden about a state–for any part of the state that never mattered for observations could be sliced off with Occam's Razor, to yield a simpler and better description."*

– Scott Aaronson

Aspecte organizatorice

Desfășurare

Informații actualizate despre curs, incluzând acest suport, se pot găsi la pagina:

https://cs.unibuc.ro/~asipos/flp/

Activitățile didactice vor fi ținute de către următorii oameni:

- curs:
  - seriile 23, 24: Traian Florin Șerbănuță
  - seria 25: Andrei Sipoș
- laborator și seminar:
  - grupa 231: Virgil Nicolae Șerbănuță
  - grupele 232, 233: Alexandru Oltean
  - grupele 234, 244: George Radu
  - grupele 241, 242, 243: Horațiu Cheval
  - grupele 251, 252: Andrei Sipoș

Notare

Examenul va consta într-o lucrare scrisă ce va cuprinde probleme totalizând 12 puncte. Puteți avea la dispoziție orice materiale **fizice** (nu digitale). Problemele vor fi inspirate de subiectele tratate la curs și seminar (în particular, ele pot fi dintre enunțurile lăsate ca exercițiu la curs). În redactarea răspunsurilor puteți folosi orice rezultat din curs sau laborator care fie este **deja demonstrat**, fie este un exercițiu **ușor**.

Pentru promovarea examenului sunt necesare 5 puncte obținute în lucrarea scrisă. La acest punctaj se poate adăuga maxim 1 punct dobândit în urma activității de la curs și laborator.

Introducere istorică

Motivația matematică

Acest curs se va ocupa, în genere, cu studiul programelor ca **obiecte matematice** despre care se pot face raționamente.

Încă dinaintea apariției calculatoarelor electronice, motivarea noțiunii formale de program a fost dată de problema (ridicată de David Hilbert și Wilhelm Ackermann) numită *Entscheidungsproblem* („problema de decizie") – anume, problema existenței unei **proceduri de decizie** pentru logica de ordinul întâi[1], adică a unui algoritm care să spună dacă un enunț dat în acea logică este sau nu universal adevărat.

Pentru aceasta, trebuia spus mai întâi ce înseamnă (ca obiect matematic, așadar) un algoritm, o procedură de decizie: în fond, a se spune când o funcție este calculabilă.

[1] Pentru o istorie chiar mai îndepărtată a acestor concepte, a se vedea introducerea cursului „Logică matematică", disponibil la adresa https://cs.unibuc.ro/~asipos/lm/, precum și referințele indicate acolo.

Funcții primitiv-recursive

Ideea de la care s-a pornit a fost cea de recursivitate, metodă prin care multe funcții (de pildă șirul lui Fibonacci) erau deja în mod uzual definite, și care are schema generală

$$f(0) := m, \quad f(n+1) := g\left(f_{|\{0,\ldots,n\}}\right)$$

Pornind de la această schemă, Thoralf Skolem a introdus în 1923 clasa funcțiilor **primitiv-recursive**. Definiția precisă a fost dată de Kurt Gödel în cursul demonstrării teoremei sale de incompletitudine (1931), funcțiile primitiv-recursive fiind un ingredient esențial al acelei demonstrații.

Funcții recursive

Gabriel Sudan (1927) și Wilhelm Ackermann (1928) au găsit, însă, exemple de funcții evident calculabile, dar care nu erau primitiv-recursive.

Exemplul oferit de obicei în cărțile actuale este o variantă a funcției lui Ackermann datorată lui Rózsa Péter, unul dintre părinții teoriei recursiei. Este vorba de acea unică funcție $A : \mathbb{N}^2 \to \mathbb{N}$ cu proprietatea că, pentru orice n, $m \in \mathbb{N}$, avem:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)) \end{aligned}$$

Ca urmare, Gödel a definit în 1934 clasa funcțiilor **general-recursive**, care includea și aceste exemple.

În 1936, Alonzo Church și Alan Turing introduc noi moduri de a defini funcțiile calculabile – modele de calcul – **calculul lambda**, respectiv **mașina Turing**. Fiecare dintre ei demonstrează că în modelul său nu poate fi decisă *Entscheidungsproblem*, folosindu-se în demonstrațiile lor de codificări asemănătoare cu cele din demonstrația teoremei de incompletitudine. Problema pusă de Hilbert și Ackermann are, așadar, un răspuns negativ.

Doar mașinile Turing reușesc să îl convingă pe Gödel că reprezintă o definiție adecvată, Turing oferind argumente că ele cuprind întreaga sferă a ceea ce se poate calcula din punct de vedere informal (afirmație cunoscută acum ca **teza Church-Turing**); el arată, însă, că ele sunt echivalente cu λ-calculul lui Church (în plus, sunt echivalente și cu funcțiile general-recursive ale lui Gödel).

Despre studiul acestora ca modele de calcul se va vorbi mai mult, însă, la cursul „Calculabilitate și complexitate".

Primele calculatoare electronice

După nu mult timp, au început să apară primele calculatoare electronice.

În vara lui 1944, Herman Goldstine, unul dintre cei care dezvoltaseră ENIAC, s-a întâlnit din întâmplare cu John von Neumann (care demonstrase independent a doua teoremă de incompletitudine a lui Gödel). Colaborarea lor a condus la un calculator succesor, numit EDVAC, a cărui arhitectură „von Neumann" este folosită în majoritatea calculatoarelor de azi. Ea este descrisă lucid în documentele redactate de ei în 1947, *First Draft of a Report on the EDVAC* și *Planning and coding of problems for an electronic computing instrument*.

În ultimul document, se simțea influența masivă a logicii asupra modului cum priveau ei folosirea acestor mașini: *"coding [...] has to be viewed as a logical problem and one that represents a new branch of formal logics."*

Din nou Turing

Și Turing, în același an 1947, este încrezător în potențialul logicii de a ajuta activitatea programării: *“there will be much more practical scope for logical systems than there has been in the past.”*

Doi ani mai târziu, el schițează cum s-ar putea desfășura verificarea formală a programelor: *“the programmer should make assertions about the various states that the machine can reach [...] the checker [i.e., the one doing the proof] has to verify that [these assertions] agree with the claims that are made for the routine as a whole [...] finally the checker has to verify that the process comes to an end.”*

Deși logica matematică, după cum am văzut, stătuse la baza dezvoltării informaticii teoretice și, în măsura în care ele pot fi considerate domenii diferite, informatica putem zice că avusese deja aplicații în logică, aplicarea logicii în informatică s-a lăsat așteptată.

Declinul

În anii '50, nu a existat aproape niciun efort semnificativ de a continua aceste idei.

Donald Knuth, 2003: *"People would write code and make test runs, then find bugs and make patches, then find more bugs and make more patches, and so on. We never realized that there might be a way to construct a rigorous proof of validity. [...] The early treatises of Goldstine and von Neumann, which provided a glimpse of mathematical program development, had long been forgotten."*

Cliff B. Jones, 2003: *"There is no compelling explanation of why more than a decade elapsed before the next landmark. Possible reasons include hardware developments, and a period of optimism that the development of programming languages would make the expression of programs so clear as to eliminate errors. In fact, as the hardware became less restrictive, the programming task became much more complex."*

McCarthy

La începutul anilor ’60, John McCarthy, care tocmai concepuse limbajul de programare Lisp, lansează, în lucrarea sa *A Basis for a Mathematical Theory of Computation*, un program ambițios de a studia matematic procesul de calcul în sine (nu doar modelul de calcul sau calculabilitatea):
“It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.”

El nu reușește să dezvolte o teorie funcțională, dar ideile sale vor rămâne:
“We hope that the reader will not be angry about the contrast between the great expectations of a mathematical theory of computation and the meager results presented in this paper.”

Floyd

Cel care reușește (în 1967) să obțină primul avans semnificativ este Robert Floyd. El reduce verificarea programelor imperative la aserțiuni ca „dacă valorile inițiale ale variabilelor satisfac proprietatea A, atunci valorile finale vor satisface proprietatea B”, obținute și demonstrate adnotând schemele programelor.

Floyd: *“The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever the connection is taken. [...] The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigour for proofs about programs in the language, appear to be novel, although McCarthy has done similar work for programming languages based on evaluation of recursive functions.”*

Metoda lui Floyd

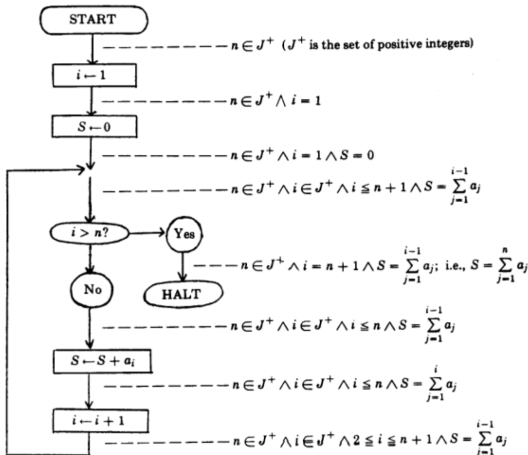

FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geq 0)$

Metoda lui Floyd de a adnota schemele programelor.

Cel care aduce la maturitate ideile lui Floyd este Tony Hoare. În 1969, el publică lucrarea *An axiomatic basis for computer programming*, în care formalizează regulile de deducție asupra condițiilor scrise ca

$$\{A\}c\{B\},$$

însemnând faptul (pomenit mai devreme) că un program c, care satisface la începutul rulării proprietatea A, va satisface la sfârșit proprietatea B. De exemplu, regula pentru instrucțiunea **while** apare ca:

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}(\textbf{while } b \textbf{ do } c)\{A \wedge \neg b\}}$$

Aceasta va fi considerată mai târziu **semantică axiomatică**, primul exemplu istoric de semantică a limbajelor de programare.

Strachey

Între timp, probleme similare apăreau în proiectarea noilor limbaje de programare de nivel înalt care erau dezvoltate la acea vreme.

Christopher Strachey a lucrat la limbajul CPL (un succesor al ALGOL 60 și un strămoș al C-ului).

Strachey: *"The development of basic semantic ideas has taken place alongside of the development of the programming language CPL and a compiler for it."*

În lucrările sale *Towards a formal semantics* (1966) și *Fundamental Concepts in Programming Languages* (1967), el introduce noțiunile acum consacrate de *l-value/r-value*, *environment*, *store*.

Semantica matematică

Ideea care îl motiva pe Strachey era de a da înțeles (semantică) unui program ca obiect matematic – el numea aceasta semantică matematică.

El întrevede faptul că semantica unei instrucțiuni va fi o funcție de la stări la stări.

Strachey: *"We are chiefly interested in the values of the expressions and not in the steps by which they are obtained."*

Pentru a trata cazul buclelor, el a folosit operatori de punct fix, despre care nu era așa clar dacă se puteau implementa sau chiar dacă existau ca obiecte matematice bine definite.

Cel care a oferit o soluție a fost Dana Scott (student al lui Church și Tarski). El a început să lucreze cu Strachey în 1969 și a ajuns repede la problema de a formaliza riguros semantica lui, în particular când era vorba de bucle.

O buclă reprezintă (intuitiv) un comportament repetat până în momentul în care rezultatul nu se mai schimbă, de aici ideea de a folosi operatori de punct fix. Contribuția lui Scott a fost de a arăta că acești operatori formali pot fi modelați ca operatori pe anumite mulțimi ordonate, așa-numitele **domenii**. Existența lor este garantată de teoreme de punct fix.

Formalizarea care rezultă este denumită astăzi **semantică denotațională**, a cărei formă specifică apare în lucrarea din 1971 a lui Scott și Strachey, *Toward a mathematical semantics for computer languages*.

Burstall

Ultima abordare principală despre care vom vorbi acum este **semantica operațională**. Ea se baza la început pe modelarea unor mașini care să execute programe pas cu pas. Aici am putea pomeni mașina SECD a lui Peter J. Landin (1964).

Contribuții esențiale aduce Rod Burstall, care dă (în anii '60) o nouă semantică a instrucțiunii de atribuire și care introduce ideea de a folosi relații, și nu funcții, pentru a modela legăturile dintre părțile programelor.

Burstall: *"[Semantics] tries to connect the very empirical subject of programming with the main body of mathematics, particularly with mathematical logic."*

Plotkin

Gordon D. Plotkin, studentul lui Burstall, după ce oferă pe parcursul anilor '70 noi contribuții la semantica denotațională, introduce (în 1981) și o nouă abordare operațională, **semantica operațională structurală**.

Ideea principală a fost să reducă ideea de mașină (matematică/virtuală) la un minim necesar pentru a exprima aspectele semantice ale limbajului. Pentru a înlocui complexitatea inițială, sunt folosite constructe în genul unor reguli de deducție.

Abordarea s-a dovedit a fi foarte populară, existând numeroase variațiuni, de exemplu **semantica naturală** (sau **big-step**) a lui Gilles Kahn, numită așa datorită asemănării cu regulile deducției naturale.

Planul cursului

În acest curs:

- vom studia câteva paradigme principale de programare – imperativă, logică, funcțională – din perspectiva semanticilor care li se pot da;
- vom explora instrumentele matematice necesare pentru exprimarea riguroasă a acestor semantici;
- vom implementa aceste semantici pe cât de fidel posibil în limbajul Haskell.