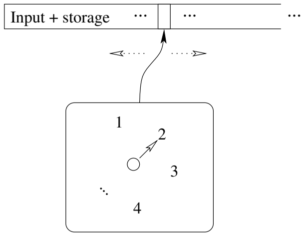
Mathematical models of computation

Stefan D. Bruda

CS 403, Fall 2024

TURING MACHINES



- Finite state control (program) + storage
- An infinite tape used as storage and also input
  - The head scans the tape, can read the current cell, can overwrite the current cell, or can move left or right
- Formally, $M = (K, \Sigma, \delta, s, h)$
- Finite set of states K, tape alphabet Σ
- Special halt state $h \notin K$ and blank symbol $\# \in \Sigma$
- $\delta : K \times \Sigma \to (K \cup \{h\}) \times (\Sigma \cup \{L, R\})$
- Configuration: $K \times \Sigma^* \times (\Sigma^*(\Sigma \backslash \{\#\}) \cup \{\varepsilon\})$, commonly written $(q, w\underline{a}w')$
- Yields in one step:
  - $(q_1, w\underline{a}u) \vdash_M (q_2, w\underline{b}u)$ iff $\delta(q_1, a) = (q_2, b)$, $b \in \Sigma$
  - $(q_1, w\underline{a}bu) \vdash_M (q_2, wa\underline{b}u)$ iff $\delta(q_1, a) = (q_2, R)$
  - $(q_1, wb\underline{a}u) \vdash_M (q_2, w\underline{b}au)$ iff $\delta(q_1, a) = (q_2, L)$
- Yields: $\vdash_M^*$, the reflexive and transitive closure of $\vdash_M$
- M computes $f : \Sigma^* \to \Sigma^*$ iff $(s, \#w\underline{\#}) \vdash_M^* (h, \#f(w)\underline{\#})$
- Computation of a Turing machine = sequence of configurations

---

THE RANDOM ACCESS MACHINE

- The Random Access Machine (RAM) consists of an unbounded set of registers R_i, $i \geqslant 0$, one register PC, and a control unit
  - The size (i.e. the number of bits) of a register is $\log n$ for an input of size n
- The control unit executes a program consisting of a sequence of numbered statements
  - In each computation step the RAM executes one statement of the program; the execution start with the first statement
  - The register PC specifies the number of the statement that is to be executed
  - The program halts when the program counter takes an invalid value (i.e. there is no statement with the specified number in the program)
- To "run" a RAM we need to
  - Specify a program
  - Define an initial values for the registers R_i, $0 \leqslant i < n$ (input)
  - The output is the content of the registers upon halting

RAM STATEMENTS

Statement	Effect on registers	Program counter
$R_i \leftarrow R_j$	$R_i := R_j$	$PC := PC + 1$
$R_i \leftarrow R[R_j]$	$R_i := R_{R_j}$	$PC := PC + 1$
$R[R_j] \leftarrow R_i$	$R_{R_j} := R_i$	$PC := PC + 1$
$R_i \leftarrow k$	$R_i := k$	$PC := PC + 1$
$R_i \leftarrow R_j + R_k$	$R_i := R_j + R_k$	$PC := PC + 1$
$R_i \leftarrow R_j - R_k$	$R_i := \max\{0, R_j - R_k\}$	$PC := PC + 1$
GOTO m		$PC := m$
IF $R_i = 0$ GOTO m		$PC := \begin{cases} m & \text{if } R_i = 0 \\ PC + 1 & \text{otherwise} \end{cases}$
IF $R_i > 0$ GOTO m		$PC := \begin{cases} m & \text{if} R_i > 0 \\ PC + 1 & \text{otherwise} \end{cases}$

Customary extensions:

- Named registers (or variables), even arrays and structures
- All the usual arithmetic operations (multiplication, division, shift, etc.)
- Structured control statements (if-then-else statements, while loops, etc.)

DESCRIBING COMPUTATIONS

- The Turing machine and the RAM are equivalent to each other within polynomial speedup/slowdown
  - These plus a lot of other models of computation (the Church-Turing thesis)
  - So it makes a lot of sense to use the RAM to express and analyze algorithms
- These two models are used for completely different purposes
- Turing machines are used to analyze problems ("what would be the common properties of all the Turing machines that solve this problem") and then to classify problems into classes (solvable, unsolvable, easy, hard, . . .)
- When a philosophical question about mechanical computation is to be answered the most common model used for such an answer is the Turing machine
- The RAM programming language is pseudocode and is the golden standard for describing algorithms
- The Turing machine/RAM constitute the mathematical model of imperative programming

---

THE LAMBDA NOTATION

- Recall that a Haskell function accepts one argument and returns one result

$$\begin{array}{lcl} \text{peanuts} & \rightarrow & \text{chocolate-covered peanuts} \\ \text{raisins} & \rightarrow & \text{chocolate-covered raisins} \\ \text{ants} & \rightarrow & \text{chocolate-covered ants} \end{array}$$

- Using the lambda calculus, a general "chocolate-covering" function (or rather λ-expression) is described as follows:

$$\lambda x.\text{chocolate-covered } x$$

- Then we can get chocolate-covered ants by applying this function:

$$(\lambda x.\text{chocolate-covered } x) \text{ ants} \quad \rightarrow \quad \text{chocolate-covered ants}$$

THE LAMBDA NOTATION (CONT'D)

- A general covering function:

$$\lambda y.\lambda x.y\text{-covered } x$$

The result of the application of such a function is itself a function:

$$\begin{array}{lcl} (\lambda y.\lambda x.y\text{-covered } x) \text{ caramel} & \rightarrow & \lambda x.\text{caramel-covered } x \\ \\ ((\lambda y.\lambda x.y\text{-covered } x) \text{ caramel}) \text{ ants} & \rightarrow & (\lambda x.\text{caramel-covered } x) \text{ ants} \\ & \rightarrow & \text{caramel-covered ants} \end{array}$$

- Functions can also be parameters to other functions:

$$\lambda f.(f) \text{ ants}$$

$$\begin{array}{cl} \multicolumn{2}{l}{((\lambda f.(f) \text{ ants}) \; \lambda x.\text{chocolate-covered}) \; x} \\ \rightarrow & (\lambda x.\text{chocolate-covered } x) \text{ ants} \\ \rightarrow & \text{chocolate-covered ants} \end{array}$$

THE LAMBDA CALCULUS

- The lambda calculus is a formal system designed to investigate function definition, function application and recursion
  - Introduced by Alonzo Church and Stephen Kleene in the 1930s
- We start with a countable set of identifiers, e.g., $\{a, b, c, \ldots, x, y, z, x1, x2, \ldots\}$ and we build expressions using the following rules:

$$\begin{array}{lcll} \textsc{LExpression} & \rightarrow & \textsc{Identifier} & \\ \textsc{LExpression} & \rightarrow & \lambda\textsc{Identifier}.\textsc{LExpression} & \text{(abstraction)} \\ \textsc{LExpression} & \rightarrow & (\textsc{LExpression})\textsc{LExpression} & \text{(combination)} \\ \textsc{LExpression} & \rightarrow & (\textsc{LExpression}) & \end{array}$$

  - In an expression $\lambda x.E$, x is called a bound variable; a variable that is not bound is a free variable
- Syntactical sugar: Normally, no literal constants exist in lambda calculus. We use, however, literals for clarity
  - Further sugar: HASKELL
  - The lambda calculus is the mathematical model of functional programming

REDUCTIONS

- In lambda calculus, an expression $(\lambda x.E)F$ can be reduced to $E[F/x]$
  - $E[F/x]$ stands for the expression E, where F is substituted for all the bound occurrences of x
- In fact, there are three reduction rules:

  α: $\lambda x.E$ reduces to $\lambda y.E[y/x]$ if y is not free in E (change of variable)
  β: $(\lambda x.E)F$ reduces to $E[F/x]$ (functional application)
  γ: $\lambda x.(Fx)$ reduces to F if x is not free in F (extensionality)
- The purpose in life of a Haskell program, given some expression, is to repeatedly apply these reduction rules in order to bring that expression to its “irreducible” form or normal form

HASKELL AND THE LAMBDA CALCULUS

- In a Haskell program, we write functions and then apply them
  - Haskell programs are nothing more than collections of λ-expressions, with added sugar for convenience (and diabetes)
- We write a Haskell program by writing λ-expressions and giving names to them:

```
succ x = x + 1                 | succ = \ x -> x + 1
                               |
length = foldr onepl 0         | length = foldr (\ x -> \ n -> 1+n) 0
    where onepl x n = 1+n      | -- shorthand:   (\ x n -> 1+n)
                               |
Main> succ 10                  | Main> (\ x -> x + 1) 10
11                             | 11
```

- Another example: `map (\ x -> x+1) [1,2,3]` maps (i.e., applies) the λ-expression $\lambda x.x + 1$ to all the elements of the list, thus producing `[2,3,4]`
- In general, for some expression E, $\lambda x.E$ (in Haskell-speak: `\ x -> E`) denotes the function that maps x to the (value of) E

MULTIPLE REDUCTIONS

- More than one order of reduction is usually possible in lambda calculus (and thus in Haskell, at least in theory):

```
square   :: Integer -> Integer
square x = x * x

smaller  :: (Integer, Integer) -> Integer
smaller (x,y) = if x<=y then x else y
```

$\mathit{square}\,(\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. $\mathit{smaller}$) $\mathit{square}\,5$ $\Rightarrow$ (def. square) 5×5 $\Rightarrow$ (def. $\times$) 25	$\mathit{square}\,(\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. square) $(\mathit{smaller}\,(5,78)) \times (\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. $\mathit{smaller}$) $5 \times (\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. $\mathit{smaller}$) 5×5 $\Rightarrow$ (def. $\times$) 25

MULTIPLE REDUCTIONS (CONT’D)

- Sometimes it even matters:

```
three   :: Integer -> Integer
three x = 3

infty :: Integer
infty = infty + 1
```

$\mathit{three}\ \mathit{infty}$ $\Rightarrow$ (def. infty) $\mathit{three}\,(\mathit{infty} + 1)$ $\Rightarrow$ (def. infty) $\mathit{three}\,((\mathit{infty} + 1) + 1)$ $\Rightarrow$ (def. infty) $\mathit{three}\,(((\mathit{infty} + 1) + 1) + 1)$ $\vdots$	$\mathit{three}\ \mathit{infty}$ $\Rightarrow$ (def. three) 3

LAZY HASKELL

- Haskell uses the second variant, called lazy evaluation (normal order, outermost reduction), as opposed to eager evaluation (applicative order, innermost reduction):

  ```
  Main> three infty
  3
  ```

- Why is good to be lazy:
  - Doesn’t hurt: If an irreducible form can be obtained by both kinds of reduction, then the results are guaranteed to be the same
  - More robust: If an irreducible form can be obtained, then lazy evaluation is guaranteed to obtain it
  - Even useful: It is sometimes useful (and, given the lazy evaluation, possible) to work with infinite objects

---

INFINITE OBJECTS

- `[1 .. 100]` produces the list of numbers between 1 and 100, but what is produced by `[1 .. ]`?

  ```
  Prelude> [1 ..] !! 10
  11
  Prelude> [1 .. ] !! 12345
  12346
  Prelude> zip [’a’ .. ’g’] [1 ..]
  [(’a’,1),(’b’,2),(’c’,3),(’d’,4),(’e’,5),(’f’,6),(’g’,7)]
  ```

  - A stream of prime numbers:

  ```
  primes :: [Integer]
  primes = sieve [2 .. ]
    where sieve (x:xs) = x : [n | n <- sieve xs, mod n x /= 0]
       -- alternative definition:
       -- sieve (x:xs) = x : sieve (filter (\ n -> mod n x /= 0) xs)

  Main> take 20 primes
  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
  ```

MEMO FUNCTIONS

- Streams can also be used to improve efficiency (dramatically!)
- Take the Fibonacci numbers:

  ```
  fib :: Integer -> Integer
  fib 0 = 1
  fib 1 = 1
  fib n = fib (n - 1) + fib (n - 2)
  ```

  - Complexity? $O(2^n)$
- Now take them again, using a memo stream:

  ```
  fastfib :: Integer -> Integer
  fastfib n = fibList %% n
      where fibList = 1 : 1 : zipWith (+) fibList (tail fibList)
            (x:xs) %% 0 = x
            (x:xs) %% n = xs %% (n - 1)
  ```

  - Complexity? $O(n)$
- Typical application: dynamic programming

KNOWLEDGE REPRESENTATION

- A proposition is a logical statement that can be either false or true
- To reason about and with propositions one needs a formal system i.e., a symbolic logic
- Predicate calculus or first-order logic is one such a logic
  - A term is a constant, structure, or variable
  - An atomic proposition (or predicate) denotes a relation. It is composed of a functor that names the relation, and an ordered list of terms (parameters):

    $\text{secure}(\mathit{room})$, $\text{likes}(\mathit{bob}, \mathit{steak})$, $\text{black}(\mathit{crow})$, $\text{capital}(\mathit{ontario}, \mathit{toronto})$
  - Variables can appear only as arguments. They are free:

    $$\text{capital}(\mathit{ontario}, X)$$

    unless bounded by one of the quantifiers ∀ and ∃:

    $$\exists\, X : \text{capital}(\mathit{ontario}, X) \qquad \forall\, Y : \text{capital}(Y, \mathit{toronto})$$
  - A compound proposition (formula) is composed of atomic propositions, connected by logical operators: ¬, ∧, ∨, →; all variables are bound using quantifires

    $$\forall X.(\text{crow}(X) \rightarrow \text{black}(X))$$

    $$\exists X.(\text{crow}(X) \wedge \text{white}(X))$$

    $$\forall X.(\text{dog}(\mathit{fido}) \wedge (\text{dog}(X) \rightarrow \text{smelly}(X)) \rightarrow \text{smelly}(\mathit{fido}))$$

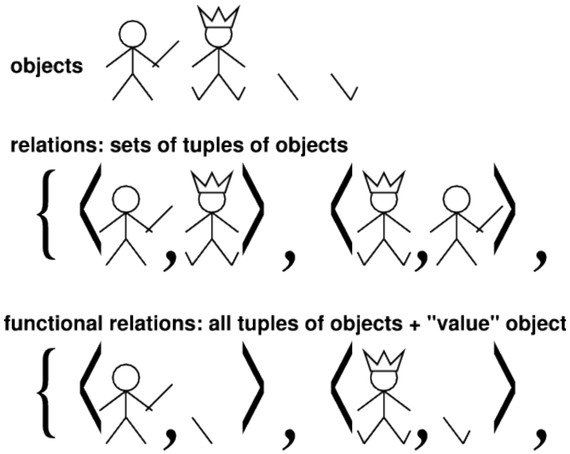
SEMANTICS OF THE PREDICATE CALCULUS

- The meaning is in the eye of the beholder
- Sentences are true with respect to a model and an interpretation
  - The model contains objects and relations among them (your view of the world)
  - An interpretation is a triple $I = (D, \phi, \pi)$, where
    - D (the domain) is a nonempty set; elements of D are individuals
    - ϕ is a mapping that assigns to each constant an element of D
    - π is a mapping that assigns to each predicate with n arguments a function $p : D^n \to \{True, False\}$ and to each function of k arguments a function $f : D^k \to D$
  - The interpretation specifies the following correspondences:

    constant symbols → objects (individuals)
    predicate symbols → relations
    function symbols → functional relations

    - An atomic sentence $predicate(term_1, \ldots, term_n)$ is true iff the objects referred to by $term_1, \ldots, term_n$ are in the relation referred to by $predicate$

SEMANTICS OF THE PREDICATE CALCULUS (CONT)



- Objects (richard, kingJohn, leg1, leg2), predicates or relations (brother), functions (leftLegOf)
- The predicate calculus is the mathematical model of logic programming

PREDICATE CALCULUS PROOFS

- Inference rules → sound generation of new sentences from old
  - Most general inference rule: resolution
  - Most used in practice: generalized modus ponens

$$\frac{\alpha_1, \ldots, \alpha_n \qquad \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta} \quad \text{(modus ponens)}$$

$$\frac{\begin{array}{l} \alpha_1, \ldots, \alpha_n \\ \alpha'_1 \wedge \cdots \wedge \alpha'_n \Rightarrow \beta \\ \exists\, \sigma : (\alpha_1)_\sigma = (\alpha'_1)_\sigma \wedge \cdots \wedge (\alpha_n)_\sigma = (\alpha'_n)_\sigma \end{array}}{\beta_\sigma} \quad \text{(generalized modus ponens)}$$

- Proof → a sequence of applications of inference rules

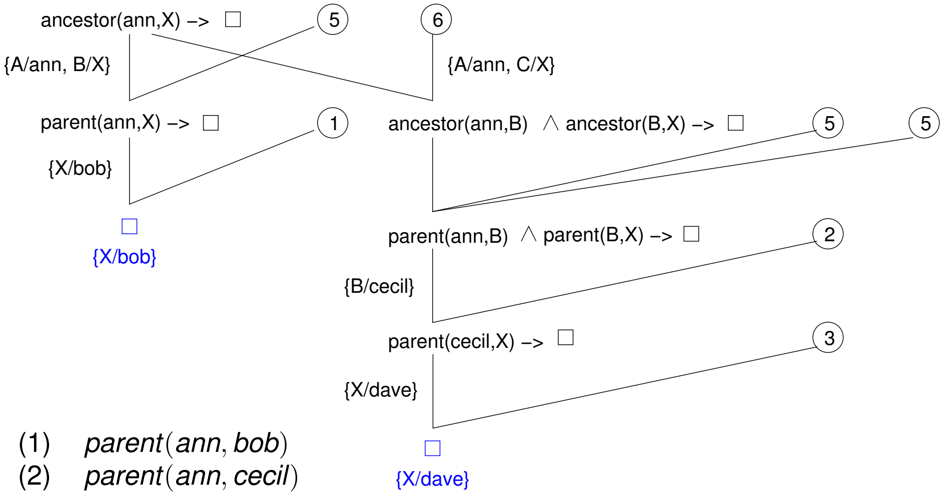
PROOF BY CONTRADICTION

- For convenience (why?) unless otherwise stated all the variables are henceforth universally quantified

KB		
Bob is a buffalo	1.	$buffalo(bob)$
Pat is a pig	2.	$pig(pat)$
Buffaloes outrun pigs	3.	$buffalo(X) \wedge pig(Y) \Rightarrow faster(X, Y)$
Query		
Is something outran by something else?		$\exists\, U : \exists\, V : faster(U, V)$
Negated query:	4.	$faster(U, V) \Rightarrow \square$
(1), (2), and (3) with $\sigma = \{X/bob, Y/pat\}$	5.	$faster(bob, pat)$
(4) and (5) with $\sigma = \{U/bob, V/pat\}$		$\square$

- All the substitutions regarding variables appearing in the query are typically reported (why?)

INFERENCE AND MULTIPLE SOLUTIONS



ancestor(ann,X) –> ☐ ⑤ ⑥

{A/ann, B/X}

{A/ann, C/X}

parent(ann,X) –> ☐ ①

{X/bob}

☐

{X/bob}

ancestor(ann,B) ∧ ancestor(B,X) –> ☐ ⑤ ⑤

parent(ann,B) ∧ parent(B,X) –> ☐ ②

{B/cecil}

parent(cecil,X) –> ☐ ③

{X/dave}

☐

{X/dave}

(1) $parent(ann, bob)$
(2) $parent(ann, cecil)$
(3) $parent(cecil, dave)$
(4) $parent(cecil, eric)$
(5) $parent(A, B) \Rightarrow ancestor(A, B)$
(6) $ancestor(A, B) \land ancestor(B, C) \Rightarrow ancestor(A, C)$