CS 455/555: Some Turing-complete formalisms

Stefan D. Bruda

Fall 2020

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 in a sequence of numbered statements
  - In each work cycle 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		$PC = \begin{cases} m & \text{if} R_i > 0 \\ PC + 1 & \text{otherwise} \end{cases}$

- The RAM is also called random-access Turing machine
- Indeed, operation is identical to the original Turing machine except that we do not spend time moving the head!
- RAM = the formal basis of all the “imperative” programming languages (C, Java, etc.)

- Basic concept: function with no name = lambda-expression

peanuts	$\rightarrow$	chocolate-covered peanuts
raisins	$\rightarrow$	chocolate-covered raisins
ants	$\rightarrow$	chocolate-covered ants

- 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:

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

- 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:

$$(\lambda y.\lambda x.y\text{-covered } x) \text{ caramel} \rightarrow \lambda x.\text{caramel-covered } x$$

$$((\lambda y.\lambda x.y\text{-covered } x) \text{ caramel}) \text{ ants} \rightarrow (\lambda x.\text{caramel-covered } x) \text{ ants} \rightarrow \text{caramel-covered ants}$$

- Functions can also be parameters to other functions:

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

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

Lambda Calculus

- The lambda calculus is a formal system designed to investigate function definition, function application and recursion. It was introduced by Alonzo Church and Stephen Kleene in the 1930s
- We start with a countable set of identifiers, e.g., $\{a, b, c, \ldots, x, y, z, x_1, x_2, \ldots\}$ and we build expressions using the following rules:

LExpression	$\rightarrow$	Identifier	
LExpression	$\rightarrow$	λIdentifier.LExpression	(abstraction)
LExpression	$\rightarrow$	(LExpression)LExpression	(combination)
LExpression	$\rightarrow$	(LExpression)	

  - 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; In practice literals are used for clarity

- In lambda calculus, an expression $(\lambda x.E)F$ can be reduced to $E[x/F]$. $E[x/F]$ 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 $\lambda y.E[x/y]$ if y is not free in E (change of variable)
  - β: $(\lambda x.E)F$ reduces to $E[x/F]$ (functional application)
  - γ: $\lambda x.(Fx)$ reduces to F if x is not free in F (extensionality)
- Computation = given some expression, repeatedly apply these reduction rules in order to bring that expression to its "irreducible" form (normal form)

- If-then-else:

$$\begin{aligned} \textit{true} &= \\ \textit{false} &= \end{aligned}$$

- If-then-else:

$$\begin{aligned} \mathit{true} &= \lambda x.\lambda y.x \\ \mathit{false} &= \lambda x.\lambda y.y \end{aligned}$$

Sample computation

- If-then-else:

$$\begin{array}{lcl} \textit{true} & = & \lambda x.\lambda y.x \\ \textit{false} & = & \lambda x.\lambda y.y \\ \text{if-then-else} & = & \lambda a.\lambda b.\lambda c.((a)b)c \end{array}$$

$$\begin{array}{rl} \multicolumn{2}{l}{(((\text{if-then-else})\textit{false})\textit{caramel})\textit{chocolate}} \\ \Rightarrow & (((\lambda a.\lambda b.\lambda c.((a)b)c)\lambda x.\lambda y.y)\textit{caramel})\textit{chocolate} \\ \overset{\beta}{\Rightarrow} & ((\lambda b.\lambda c.((\lambda x.\lambda y.y)b)c)\textit{caramel})\textit{chocolate} \\ \overset{\beta}{\Rightarrow} & (\lambda c.((\lambda x.\lambda y.y)\textit{caramel})c)\textit{chocolate} \\ \overset{\beta}{\Rightarrow} & ((\lambda x.\lambda y.y)\textit{caramel})\textit{chocolate} \\ \overset{\beta}{\Rightarrow} & (\lambda y.y)\textit{chocolate} \\ \overset{\beta}{\Rightarrow} & \textit{chocolate} \end{array}$$

Multiple reductions

- Let $\omega = \omega + 1$

innermost (eager evaluation)

$$
\begin{aligned}
(\lambda x.3)\omega \quad &\Rightarrow \quad (\text{def. } \omega) \\
&\qquad (\lambda x.3)(\omega + 1) \\
&\Rightarrow \quad (\text{def. } \omega) \\
&\qquad (\lambda x.3)(\omega + 1 + 1) \\
&\Rightarrow \quad (\text{def. } \omega) \\
&\qquad (\lambda x.3)(\omega + 1 + 1 + 1) \\
&\vdots
\end{aligned}
$$

outermost (lazy evaluation)

$$
\begin{aligned}
(\lambda x.3)\omega \quad &\Rightarrow \quad (\text{def. } \lambda x.3) \\
&\qquad 3
\end{aligned}
$$

- Two terminating reductions are guaranteed to reach the same normal form
- If any reduction terminates then the outermost reduction is guaranteed to terminate

- Lambda-calculus = formal basis for all functional programming languages (Haskell, ML, etc.)

	Functional programming	Ordinary programming
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Write functions that define the problem	Figure out solution
4.	Coffee break	Program solution
5.	Encode problem instance as data	Encode problem instance as data
6.	Apply function to data	Apply program to data
7.	Mathematical analysis	Debug procedural errors

First-Order Logic (FOL): Syntax

- Basic ingredients are Constants (*KingJohn*, 2, *UB*, ...), predicates (*Brother*, $>$, ...), functions (*Sqrt*, *LeftLegOf*, ...), variables (x, y, a, b, ...), boolean operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$), equality ($=$), quantifiers ($\forall$, $\exists$)
- Atomic sentence: $predicate(term_1, \ldots, term_n)$ or $term_1 = term_2$
  - Term: $function(term_1, \ldots, term_n)$ or *constant* or *variable*
  - Examples:

$$Brother(KingJohn, RichardTheLionheart)$$

$$> (Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))$$

- Complex sentences consist in atomic sentences joined together using logical operators
  - Examples:

$$Sibling(KingJohn, Richard) \Rightarrow Sibling(Richard, KingJohn)$$

$$>(1,2) \vee \leqslant(1,2)$$

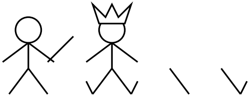
$$>(1,2) \wedge \neg>(1,2)$$

- Sentences are true with respect to a model and an interpretation
  - The model contains objects and relations among them
  - 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 $p : D^n \rightarrow \{True, False\}$ and to each function of k arguments a function $f : D^k \rightarrow D$
  - The interpretation specifies referents for

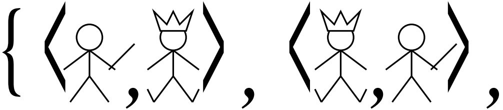
constant symbols	$\rightarrow$	objects (individuals)
predicate symbols	$\rightarrow$	relations
function symbols	$\rightarrow$	functional relations

  - 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 $predicate$

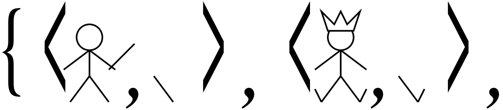
objects



relations: sets of tuples of objects



functional relations: all tuples of objects + "value" object



- $\forall \langle variable \rangle \; \langle sentence \rangle$
  - Everyone at Bishop's is smart: $\forall x \;\; Attends(x, Bishops) \Rightarrow Smart(x)$
  - $\forall P$ is equivalent with the conjunction of instantiations of P

$$
\begin{array}{lll}
 & Attends(KingJohn, Bishops) & \Rightarrow & Smart(KingJohn) \\
\wedge & Attends(Richard, Bishops) & \Rightarrow & Smart(Richard) \\
\wedge & Attends(Bishops, Bishops) & \Rightarrow & Smart(Bishops) \\
\wedge & \ldots & &
\end{array}
$$

- $\exists \langle variable \rangle \; \langle sentence \rangle$
  - Someone at Queen's is smart: $\exists x \;\; Attends(x, Queens) \wedge Smart(x)$
  - $\exists x \;\; P$ is equivalent to the disjunction of instantiations of P

$$
\begin{array}{lll}
 & Attends(KingJohn, Queens) & \wedge & Smart(KingJohn) \\
\vee & Attends(Richard, Queens) & \wedge & Smart(Richard) \\
\vee & Attends(Queens, Queens) & \wedge & Smart(Queens) \\
\vee & \ldots & &
\end{array}
$$

Equality and Substitution

- $=$ is a predicate with the predefined meaning of identity: $term_1 = term_2$ is true under a given interpretation iff $term_1$ and $term_2$ refer to the same object
- Suppose that we have a given set of statements known to be true (knowledge base, KB) and we wonder whether the KB entails

$$\exists a \;\; Action(a)$$

  (i.e. is the sentence true given the KB)
  - Possible answer: *Yes*, $\{a/Shoot\}$ $\leftarrow$ substitution (binding list)
  - Given a sentence S and a substitution σ, S_σ denotes the result of plugging σ into S; e.g.,

$$S = Smarter(x, y)$$
$$\sigma = \{x/Hillary, y/Bill\}$$
$$S_\sigma = Smarter(Hillary, Bill)$$

  - We look for the most general substitution = unification algorithm

Unify:	With:	Substitution:
Dog	Dog	$\emptyset$
x	y	$\{x/y\}$
x	A	$\{x/A\}$
$F(x, G(T))$	$F(M(H), G(m))$	$\{x/M(H), m/T\}$
$F(x, G(T))$	$F(M(H), t(m))$	Failure!
$F(x)$	$F(M(H), T(m))$	Failure!
$F(x, x)$	$F(y, L(y))$	Failure!

- Equality, revised: $=$ is a predicate with the predefined meaning of identity: $term_1 = term_2$ is true under a given interpretation iff $term_1$ and $term_2$ unify with each other

FOL PROOFS

- Inference rules: generalized resolution

$$\frac{\alpha \vee \beta', \qquad \neg\beta'' \vee \gamma, \qquad \exists\,\sigma \;\; \beta = \beta'_\sigma \wedge \beta = \beta''_\sigma}{\alpha_\sigma \vee \gamma_\sigma}$$

and generalized modus ponens

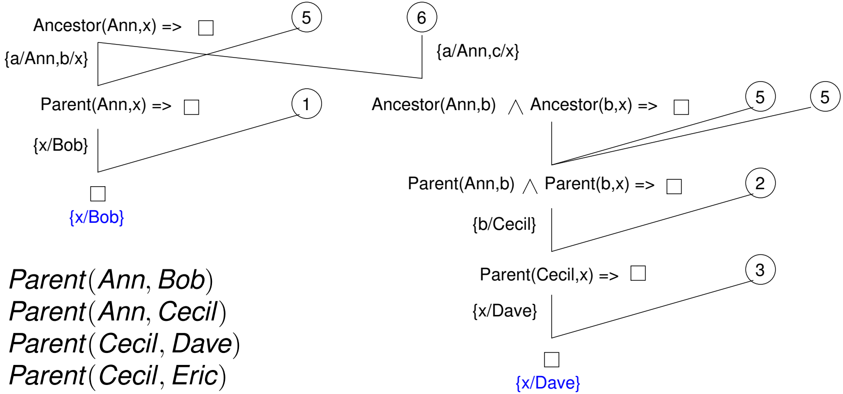
$$\frac{\begin{array}{c}\alpha_1, \ldots, \alpha_n, \qquad \alpha'_1 \wedge \cdots \wedge \alpha'_n \Rightarrow \beta, \\ \exists\,\sigma \;\; (\alpha_1)_\sigma = (\alpha'_1)_\sigma \wedge \cdots \wedge (\alpha_n)_\sigma = (\alpha'_n)_\sigma\end{array}}{\beta_\sigma}$$

- Application of inference rules: sound generation of new sentences from old
    - Proof = a sequence of inference rule applications
    - Can use inference rules as operators in a standard search algorithm

PROOF BY CONTRADICTION

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?		$Faster(u, v)$
Negated query:	4.	$Faster(u, v) \Rightarrow \square$
(1), (2), and (3), $\sigma = \{x/Bob, y/Pat\}$	5.	$Faster(Bob, Pat)$
(4) and (5), $\sigma = \{u/Bob, v/Pat\}$		$\square$

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



(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) $Ancestor(a, b) \wedge Ancestor(b, c) \Rightarrow Ancestor(a, c)$

- FOL = formal basis for all logic programming languages (Prolog, etc.)

	Logic programming	Ordinary programming
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Coffee break	Figure out solution
4.	Encode information in KB	Program solution
5.	Encode problem instance as facts	Encode problem instance as data
6.	Ask queries	Apply program to data
7.	Find false facts	Debug procedural errors