Formal Models: from Algorithms to Distributed Computing

Mirko Viroli
DEIS, Università degli Studi di Bologna
mviroli@deis.unibo.it

Presentazione..

- Docenza
  - II Facoltà di Ingegneria, Univ. Bologna, Cesena
  - Fondamenti di Informatica L-A
  - "Programmazione imperativa (in Java)"
- Ricercatore
  - DEIS: Dipartimento di Informatica, Elettronica, e Sistemistica
- Interessi di ricerca
  - sistemi distribuiti: coordinazione ed interazione
  - artificial intelligence: multi-agent systems
  - linguaggi di programmazione: OO mechanisms

The seminar today

- Title:
  - Formal Models in Computer Science: from Algorithms to Distributed Computing
- Elements:
  - 1. Why Formal Models in Computer Science
  - 2. Formal Models for Algorithms
  - 3. From Algorithms to Interaction
  - 4. Formal Models for Distributed Computing
- Connection:
  - "Linguaggi e Modelli Computazionali" @ ICT

Outline

- Why Formal Models in Computer Science
- Formal Models for Algorithms
- From Algorithms to Interaction
- Formal Models for Distributed Computing

Formal framework

- Formal framework, or theory
  - a set of mathematical descriptions
  - defining in a precise, non-ambiguous way mathematical structures (sets, functions, relations,...)
  - ➢ used to represent abstractions of interest
- Examples:
  - mathematics:
    - set-theory, arithmetics, functional analysis,..
  - logics, algebras, geometry, statistics,...

Arithmetics

- Can be derived on top of set-theory
  - given the concepts of sets, functions, cartesian product..
- Natural Numbers N
  - 0 is the void set {}, 1 is {0}, 2 is {0,1}, 3 is {0,1,2},..
  - n is the set {0,1,..,n-1}
- Operations
  - n*m is the number that bijects with set n X m
  - ➢ this is to say that n*m is the number of elements of n X m
  - n+m is the number that bijects with set (n X 0) ∪ (m X 1)
  - ➢ this is to say that n+m is the number of elements of the (disjoint) union of n and m
- Similarly define Q, R, C , operations -, /, ...

On Formal Frameworks

- Arithmetics is used to execute basic calculus
- Other formal frameworks are defined similarly
- Features of a formal framework
  - it provides mathematical structures
  - they can be given useful interpretations in terms of concrete concepts
  - (sometimes, more interpretations are allowed)
- There are always two levels
  - the mathematical structures
  - the concrete concepts

Models

- What is a model?
- Is a partial description of a system of interest
  - only some aspects of this system are of interest
  - in general, not all aspects are
  - (the only complete model to a system, is the system itself)
- A model:
  - focusses on some aspect
  - abstract away from other aspects
- Why models?
  - forgetting about irrelevant things leads to simplicity
  - is the key to harness complexity

Examples of Models

- Example 1: Scale models (e.g. of a house)
  - preserve relative dimensions, colors,..
  - forget about actual dimensions
- Example 2: Probabilistic models (e.g. crap throw)
  - preserve the rate of outcomes of an event
  - forget about how an outcome is reached
  - ➢ 1/6 probability to make a "3"

Formal/Informal Models

- How do you represent a model?
- Informally
  - describing the model with "words", graphics, figures,..
  - building some artifact
  - ➢ useful to give rough impressions
  - e.g. a scale model
- Formally
  - described in terms of a formal framework
  - ➢ useful to give precise, unambiguous descriptions
  - e.g. probability models
- When do I need formal models
  - when I should precisely predict properties and behaviour of a system
  - when the system is critical!!!!

Formal Models and Engineers

- Where are formal models used in engineering? Basically, everywhere..
- Electronics
  - Circuits, Electronic devices
  - "Modello a parametri concentrati"
  - Laws of Phisics, Mathematics, Probability
- Telecommunication, Mechanics, ...
- All these engineers work with formal frameworks!!
- Typically
  - an engineer applies a formal framework to the description of a critical system, to ensure it is well constructed!

Formal Models in CS

- Does this apply to Computer Science Engineers?
  - (formal) frameworks in CS are called "computational models"
- Do you use formal models to build good systems?
- A common answer:
  - No, I simply know technology and use experience
  - E.g.: programming languages + idioms
- Actually:
  - technology sometime enjoys formal frameworks
  - still, you do not use formal models on top..

On Engineering and CS

- Why CS is different from other fields?
  - it is younger
  - it is the science of discrete mathematics
- Still, formal models exist for CS
  - they describe in an abstract way a software system you need to build
- You can use them at each step of the engineering process
  - analysis, design, implementation, testing

Formal Models to Design

- They are particularly useful when designing a system
  - when deciding how the software has to be constructed to satisfy the properties coming from analysis
- Benefits
  - to devise system behaviour precisely
  - to avoid underspecification
  - to formally check that your design is sound
  - to drive towards correct implementations

The "models" viewpoint

- Being formal is not always necessary
  - sometimes (informal) models are precise and complete, even though not mathematical
  - they are good anyway..
- What is the difference from a technician and a (good) engineer?
  - the engineer knows models, and use them to represent, design, and build fruitful systems
  - a technician has only experience, he shortly falls to deal with complexity and with new situations
- Keep the "models" viewpoint!!

The goals of this seminar

- Not to teach you formal models of computer science in details
- Rather, to focus on distributed system:
  - give you examples of core languages
  - make you understand, or at least, intuit, the difference and the key features of algorithms and interaction
  - give you the most relevant attack to the problem of interaction
  - describe techniques to model distributed systems
  - describe some well-known computational models
  - show other ways of programming
- What is left outside
  - a true example of a system specification
  - how formal properties are proven

Outline

- Why Formal Models in Computer Science
- Formal Models for Algorithms
- From Algorithms to Interaction
- Formal Models for Distributed Computing

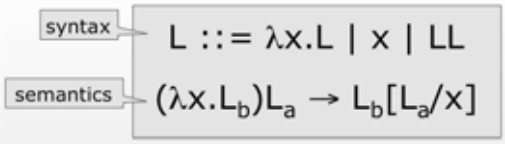
Algorithms

- Computers execute algorithms
- An algorithms is a sequence of atomic steps of a machine (abstract or concrete) whose final goal is to produce an output from a given input
- Hence an algorithm specifies a function from input to output
  - sorting a vector
  - searching an element in a list
  - name a complex algorithm.. it is a function!

Lambda-calculus

- By Alonzo Church ('30)
  - almost concurrently to the Turing Machine
  - long before any programming language and computer
  - developed to study the meaning of "computation" and "algorithm"
- Goals:
  - to develop the smaller formal framework capable of describing any algorithm, that is Turing equivalent
  - it is actually a (programming) language
  - ➢ focuses on just one concept: the function
  - ➢ focuses on just one manipulation paradigm: function application
  - sensibly simpler than e.g. Java
- Result:
  - it was the foundation basis for functional and OO languages in the whole XX century

Everything here



- L can express
  - ANY data structure and ANY algorithm!
- → represents an atomic transformation
  - sequences of → model ANY computation!

Syntax

L,M,N ::= λx.L | x | LL

A lambda-term can be

- λx.L, a function
  - without a name (called a closure)
  - has one parameter, called x (a variable)
  - has a body L, which is again a lambda-term
- x, a variable
- LM, a function application
  - L is supposed to be a function
  - M is a parameter, applied to the function

Relation w.r.t. JavaScript

- JavaScript
  - is a functional language with closures
  - (it has OO and imperative features as well)
  - as all functional languages, has lambda as a foundational language
- JavaScript syntax
  - ➢ function(x){return <exp>;} stands for λx.<exp>
  - given var f=function(x){ return <exp_b>;}
  - ➢ f(<exp_a>) stands for (λx.<exp_b>)(<exp_a>)

Examples in Java

- Method definition
  - λx.x is "Object m(Object x){ return x;}"
  - notice:
    - functions in lambda have no name
    - λx.x is the same as λz.z
    - this is the identity function!
- Method invocation
  - (λx.x)y is "m(y)"

Grammar ambiguities

L ::= λx.L | x | LL

- Consider the lambda λx.xy
  - is it (λx.x)y? applies y to function (λx.x)
  - or λx.(xy)? is a function on x, returning (xy), that is the result of applying y to (function) x
- How to disambiguate?
  - obviously, by parenthesis
- Priorities
  - Application is left-associative
    - yLxM is ((yL)x)M
  - Bodies extend as much as possible
    - λx.λy.xyx = λx.(λy.xyx) = λx.(λy.(xy)x)

Textual Notation

- Sometimes you need it
- Just substitute λ by \l
- Write \lx.x for λx.x

Semantics

(λx.L)M → L[M/x]

- Consider any lambda-term, a computational step is
  - identify inside it a pattern of the kind (λx.L)M
  - ¬ that is, a function λx.L applied to parameter M
  - transform it to the lambda-term obtained by
  - ¬ taking the function body L and substitute each occurrence of x by M
- Each such transformation is called *reduction*

Examples

(λx.L)M → L[M/x]

- x is clearly not reducible
- λx.x again is not reducible
- (λx.x)y reduces to y, write (λx.x)y → y
  - L is x, M is y, L[M/x] is x[y/x], that is, y
- (λx.xx)(λy.(λz.yz))→(λy.(λz.yz))(λy.(λz.yz))
  - L is xx, M is λy.(λz.yz)
  - do not be scared, this is just syntactic stuff!

Meaning

(λx.L)M → L[M/x]

- Invoking a function means executing its body after applying the formal parameters!
- (λx.x)y → y
  - Object m(Object x){ return x;}
  - m(y) returns y
- Another example
  - given int m2(int x){ return x*2*x;}
  - m2(4) means executing 4*2*4, that is,x*2*x[4/x]

On substitutions...

(λx.L)M → L[M/x]

- you can always substitute, but x should not appear in M, to avoid confusion
- if you need, alpha-rename functions!
- e.g. write (λz.z) instead of (λx.x)
- example:
  - (λx.xx)xyx = (λz.zz)xyx → (xyx)(xyx)

More arguments

- How to define a function with two arguments?
- Define a function of function..
  - λx.λy.xy, which is λx.(λy.xy)
  - is a function on x, whose body is a function on y
  - (λx.λy.xy)(λz.zz)(w) reduces to
  - (λy.(λz.zz)y)(w) reduces to
  - (λz.zz)w, which is ww
- In general
  - λx.λy.λz.L is a function with three args x,y,z
  - (λx.λy.λz.L)MNO applies the function to M,N,O
  - obtains: L[M/x,N/y,O/z]

Computing with lambda

- Algorithms describe functions
  - they take a data structure in input
  - after a finite computation
  - they provide a data structure in output
- In lambda-calculus
  - let L be lambda-term describing the algorithm
  - let D be the lambda-term for the input data
  - LD is "apply L to D"
  - reduce LD as much as you can
  - obtain the new lambda-term O, which is the output
- Which example?

Boolean values

- Boolean values are "true" and "false"
- Operations (algorithms) on such values are
  - OR, AND, NOT, XOR..
- What does it mean to realise booleans?
- A good characterisation is the bool ADT
  - values are true and false
  - in whatever way you implement "true", "false", "NOT", "AND".. be sure that
    - NOT(true)=false, NOT(false)=true
    - AND(true,true)=true, AND(true,false)=false, ...
    - or, more easily, AND(true,X)=X, AND(false,X)=false

The Bool ADT

```
adt Bool
   constructors:
              true : Bool
              false : Bool
   functions:
              and,or : Bool x Bool → Bool
              not : Bool → Bool
   axioms:
        [B1]  and(true,x)=  x
        [B2]  and(false,x)= false
        [B3]  not(true)=    false
        [B4]  not(false)=   true
        [B5]  or(x,y)=      not(and(not(x),not(y)))
   end
```


Example

- True: λx.λy.x
- False: λx.λy.y
- NOT: λx.xFT
- AND: λx.λy.xyF
- Call them T,F,Not,And for short
- Do axioms hold?
  - Not T = (λx.xFT)T → TFT → (λx.λy.x)FT → → F
  - Not F = (λx.xFT)F → FFT → (λx.λy.y)FT → → T
  - And T X = (λx.λy.xyF)TX → → T X F → X
  - And F X = (λx.λy.xyF)FX → → F X F → F

Which meaning?

"Everything is a function" philosophy:

- True: λx.λy.x
  - Object T(Object x,Object y){ return x;}
- False: λx.λy.y
  - Object F(Object x,Object y){ return y;}
- NOT: λx.xFT

  bool Not(bool x){ return x(F,T);}, or:

  "boolean Not(boolean x){ return x?false:true;}"
- AND: λx.λy.xyF

boolean And(boolean x,boolean y){ return x?y:false;}

Meaning/Representation

- Simple and intuitive values such as True and False are represented by the strange syntax "λx.λy.x" e "λx.λy.y"
  - this should not be surprising or scaring at all!
- Never confuse our perception of a value with its internal representation into a computing machine
- As in all formal frameworks:
  - the same as defining 3 as {0,1,2}
- Isn't it the same when you say that
  - 00100 + 00011 = 00111 means 4+3=7
- Internal representation is a convention with the goal of easing the work of computers!

Lambda is Turing-complete!

- What does it need to be complete?
  - any partial recursive function should be realised!
- Which ingredients?
  - representing natural numbers: 0,1,2,3..
  - representing the successor function, succ(n)=n+1
  - representing projection, f(x,y)=y, etc...
  - representing composition, f(g(x))
  - representing recursion

Numerals

Example

- 0 = λz.z
- 1 = λz.λs.sz
- 2 = λz.λs.s(sz)
- 3 = λz.λs.s(s(sz)), ...
- Succ = λn.λz.λs.s (n z s)
  - Succ 1 → λz.λs.s(1 z s) → → λz.λs.s(sz) = 2
- Sum = λm.λn.λz.λs.n (m z s) s
  - Sum 1 1 → λz.λs.1(1zs)s → λz.λs.1(sz)s → λz.λs.s(sz) = 2
- Analogously: isZero, Pred, Times

Recursion

- By a strange lambda
  - Y= λf. (λx.f(x x)) (λx.f(x x))
  - property: whatever F, we have Y F→→F(Y F)→→F(F(Y F))...
- Factorial
  - informally, fact(x)= (x==0?1:x*fact(x-1))
  - Fact = λfact. λn. ((isZero n) 1 (times n (fact (pred n))))
- Computation:
  - Y Fact 2 → Fact (Y Fact) 2 →* (times 2 ((Y Fact) 1)) →* (times 2 1) → 2
- Strange, a bit hard to understand, but it works!
- Any computable function can be represented

Normalizing a lambda

- A lambda is in normal-form if no reductions are appliable
  - such a lambda is the result of the computation
- Notice that at a given time, more than one reduction is appliable
  - (add (add 0 1) (add 1 2))..
  - the same as in 1+3+4*5 (=1+3+20, or =4+4*5)
- Lambda strongly-normalizable:
  - whichever sequence of reductions, we will end with a normal-form
- Lambda not normalizable:
  - a normal-form is never reached
- Lambda (weakly-)normalizable:
  - at least one sequence of reductions leads to a normal-form

Reduction Graph

- Consider a graph with lambda in nodes and arcs for reductions, if a node is:
  - strongly normalizable (SN): acyclic graph
  - weakly normalizable (WN): has at least one leaf
  - non normalizable (NN): no leafs
  - leafs are lambda in normal form
- Examples:
  - Ω = (λx.xx) (λx.xx) → Ω → Ω → Ω

Meaning

- SN: terminating computations
  - given, int f(int x,int y){ return x+y+1;}
  - the computation f(2,4)
- NN: non-terminating computations
  - given, int f(int x){ return f(x);}
  - the computation f(2)
- WN: possibly non-terminating computations
  - depending of the evaluation order
  - given, int f(int x){ return x==0?0:f(x);}
  - the computation f(0)

Church-Rosser Theorem

- TEO: a lambda has at most one normal formal
  - this theorem justifies its interpretation as the "result"
  - and it means that lambda-calculus is deterministic!
- Question:
  - If we have a weakly-normalizable lambda (both loops and results), choosing the proper reduction is relevant...
  - which choices guarantee to converge?
- Reduction choice amounts to the problem of passing arguments
  - call-by-value guarantees convergence

Call-by-name

- Normal order
  - applying the left-most reduction... second lambda below.
  - λx. (λz.z) ((λw.w) (x(λy.y)))
- In extensions of this calculus dealing with side-effects
  - this strategy amounts to the call-by-name of parameters (or call-by-reference)
  - for instance, objects and arrays in Java

Call-by-value

- Applicative order
  - applying the left-most reduction for which the body is of the kind: E ::= x | x E | (λx.L)... third lambda below.
  - λx. (λz.z) ((λw.w) (x(λy.y)))
- This strategy amounts to the call-by-value of parameters
  - only "evaluated" parameters are passed
  - for instance, primitive values in Java
  - ML and Scheme functional languages

Call-by-need

- Lazy order
  - applying the left-most reduction that is not inside another function... none of the lambda below.
  - λx. (λz.z) ((λw.w) (x(λy.y)))
- This strategy amounts to the call-by-need of parameters
  - stops when reaching a "weak-head normal form"
  - Haskell functional language
- Which properties?
  - On normalizable lambdas, -by-value and -by-need terminates

Which applications

- Defining properties of programming languages (parameters..)
- Inspiring functional languages (ML, Scheme, Lisp, Haskell, ..)
- Type systems on lambda-calculus are used to introduce the main typing concepts
  - e.g. generic lamba-calculus (2nd order lambda-calculus)
  - object oriented typing
  - ...
- Also inspired modern foundational languages

How to use these models?

- They are not used by end-users
  - noone defines algorithms by lambda-calculus
- They are used by people building technologies
  - most notably, languages
  - the formal framework has been used to guarantee important properties
  - if compilation succeeds, your program won't cause errors! (called type-soundness)
- You enjoy formal frameworks, but don't know that!

Models for Engineer?

- Not a lot for algorithms, other than programming languages themselves
- Some exist:
  - Finite-State Automata
  - Petri-Nets
- We do not describe them, our focus will go to distributed systems, which are more challenging

Outline

- Why Formal Models in Computer Science
- Formal Models for Algorithms
- From Algorithms to Interaction
- Formal Models for Distributed Computing

From Algorithm to Interaction

- A unified characterization of algorithmic computation
- Why is it not enough?
- Characterizing interaction

Formal models for algorithms

- There are many:
  - Lambda Calculus
  - Automata
    - FSA
    - PDA
    - Turing Machine
  - Register Machine
  - ...
- What do they have in common?
  - they have an input
  - they elaborate on that input
  - they (possibly) terminates yielding an output

Reduction systems (RS)

- Formally $<S, \rightarrow>$, with $\rightarrow \subseteq S \times S$
  - S, information to elaborate, (a numerable set)
  - $\rightarrow$, elaboration (a computable relation)
- Notation and meaning:
  - $(s,s') \in \rightarrow$, is written $s \rightarrow s'$
  - read " s reduces to s' "
  - information s is transformed to s'
  - $s \rightarrow s' \rightarrow s'' \ldots \rightarrow s^n$, is written $s \rightarrow^* s^n$
  - $s \rightarrow \perp$ means that s is no more reducible

Computing schema

- Computing:
  - start from an $s_i \in S$ representing "f(input)"
  - applies $\rightarrow$ many times, until reaching a final value $s_o \in S$, which is the "output"
- Formally
  - $s_i \rightarrow^* s_o \rightarrow \perp$

Properties

- In general, RS can deal with non-determinism
  - two different ouputs are admissible
  - $s_i \to^* s_o \to \bot$, $s_i \to^* s_o' \to \bot$, with $s_o \neq s_o'$
  - it's not the case for the lambda-calculus!
- Loops can occurs
  - in particular, in any Turing-complete formalims
  - $s \to s \to s \to s \to \ldots$
  - $s_1 \to s_2 \to s_1 \to s_2 \to \ldots$
- Can be used to deal with any formalism for algorithms!

Other models..

E.g. FSA, finite-state automata

- S is a triple
  - the current state
  - the sequence of inputs still to be consumed
  - the sequence of outputs already produced
- $\to$ is basically the state-transition function
  - applying a reduction means
    - take an input
    - apply the state-transition function
    - obtain the new state and an output
  - until reaching a final state!

Defining our own RS

- First, provide a numerable set S
  - mathematically
  - or by a BNF grammar (always numerable)
- Then, give rules defining the behaviour of "$\to$"

$$\frac{condition}{\sigma \to \sigma'}$$

- Rules define schemata of reductions using variables, similarly to logic programming (Prolog)
- This way, a RS resembles an operational semantics

Example

- Want to define algorithms over sequences of integers...
  - $s ::= \varepsilon \mid n \mid s;s$
  - e.g.: 1;2;3
- As algorithm, just consider one rule:

$$\frac{i > j}{s; i; j; s' \to \ s; j; i; s'}$$

i,j variables over numbers
s,s' variables over set S

Which algorithm?

$$\frac{i > j}{s; i; j; s' \to \ s; j; i; s'}$$

- Bubble-Sort (non-deterministic)
  - 3;2;1 → 2;3;1 → 2;1;3 → 1;2;3
  - 3;2;1 → 3;1;2 → 1;3;2 → 1;2;3
- Which rule(s) for the determinstic one?
- Which rules for other algorithms?

The framework of RS

- Can be used to:
  - give an abstract and compact representation of algorithms
  - allow for a mathematical treatment
- Which applicability?
  - the execution is "blind"
  - during execution
    - cannot provide an input value
  - to a certain extent
    - cannot observe partial results (which meaning?)
    - cannot drive non-determinism
- For algorithms this is OK, but what about real systems?

The interaction issue

- In order to describe modern systems
  - distributed, interactive, open
  - web application, pervasive computing, component-based systems
- ...we need a different framework, capturing:
  - interaction during computation
  - communicating information
  - synchronisation
  - composability of subsystems
  - the notion of environment
  - observation of a system behaviour

The paradigm shift

Some outstanding papers:

- Peter Wegner, Why Interaction Is More Powerful Than Algorithms, Communications of the ACM., May 1997

  http://www.cs.brown.edu/people/pw/papers/ficacm.ps
- Robin Milner, Elements of Interaction - Turing Award Lecture, Communications of the ACM., Jan 1993

  download from www.acm.org, or ask to me

Transition systems

- The key idea is captured by extending RS
  - using a different formal framework!
- A transition system:
  - $<s_0, S, \rightarrow, A>$ with $\rightarrow \subseteq S \times A \times S$
  - S, state of the system of interest, numerable
  - $s_0 \in S$, an initial state for the system
  - A, actions visible from outside, numerable
  - $\rightarrow$, represents change state + executing action
- Notation:
  - read as "from state s to state s' by action a"

$\langle s, a, s' \rangle \in \rightarrow$ $\quad$ $s \xrightarrow{a} s'$

Interactive computing

- Computation:
  - from an initial state
  - as actions occur, the state changes
- What are actions? Can be interpreted as:
  - internal changes (algorithmic computations).. silent actions
  - receiving an input (a stimulus, a message)
  - producing an output (sending message)
  - observing a partial result
- What is interactive computing..
  - ..given that algorithmic computations are functions from input to output?

Observable behaviour

- An interactive computing machine
  - does not realise a functional transformation
  - rather, it has an observable behaviour (OB)
- OB
  - is what an external entity, living in the environment is allowed to perceive about the evolution of the interactive machine
  - OB defined in terms of the set of all the possible sequences of actions, called *interaction histories*
  - the set S is completely useless from the OB viewpoint
  - ➢ (this is to tackle information hiding!!)
- Formally:
  - $(a_0, a_1, a_2, .., a_n) \in OB$ if $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_{n+1} \not\rightarrow$

Example: the Clock

- A hardware component issuing an electrical stimulus, called "tic", each microsecond
- Set of observable actions
  - A={tic}
- Which OB, which interaction histories?
  - H={(tic,tic,tic,tic,tic,tic,...)}
- Notice:
  - we do not really care about time here, only about the sequence of what happens

Example: Summing Integers

- A machine computing the sum of two integer numbers
- Actions
  - receiving integers m and n in input
  - producing output r
  - A={inp(m,n):m,n∈N}∪{out(r):r∈N}
- OB
  - H={(inp(1,1),out(2)), (inp(3,6),out(9)), ...}
- Notice:
  - an algorithm executes only once!
  - our description applies to H/W and S/W machines

Example: Generating Primes

- A machine that generates prime numbers
- Actions
  - p(n), produces the prime number n
  - stop, the machine is stopped
  - A = {p(n):n∈N}∪{stop}
- OB:
  - H ={ (p(2),stop),(p(2),p(3),stop), (p(2),p(3),p(5),stop),...}
- Notice:
  - inputs and outputs are handled symmetrically

On observational semantics

- The OB we defined is called "completed trace semantics"
  - we observe sequences (traces) of interactions
  - we are aware of when the computation is over
- Depending on the application
  - there are many many kinds of observation semantics (trace, simulation, bisimulation,..)
  - they are 23!
  - trace semantics is the coarsest
  - bisimulation is the finest
- In the following we stick to completed trace

Equivalence & Refinement

- Two processes are equivalent if they have the same OB
  - that is, the same set of observations
- A process P is an "implementation" of a process Q
  - if P has a smaller set of observations
  - also say that P is a refinement of Q
- Both depend on which OB we choose

Paradigm shift

- The ability of interacting emerged as computers (and only later, networks) evolved
  - Computers interact with users
  - Programs run concurrently and interact
  - Computers interact with each others in nets
- This change of perspective influenced formal models (basically from 70s)
  - automata (e.g. Persistence Turing Machine,..)
  - fundational languages

Outline

- Why Formal Models in Computer Science
- Formal Models for Algorithms
- From Algorithms to Interaction
- Formal Models for Distributed Computing

There are many

- Petri Nets
- Process Algebras
- Interactive Automata
- Persistent Turing Machine
- Transition Systems - Based

Process Algebras

- Idea, similar to lambda-calculus
  - capturing the few concepts of interaction
  - having a simple BNF syntax
  - providing a TS-based semantics (instead of a RS-based)
- Supporting composability
  - defining a process, composition as an operator
  - representing communications
- S as an algebra of processes
  - operators of composition, choice, action execution,...
  - "parallel" of two processes as a process
  - process moves to another (state) by executing an action

Usages

- Many algebras
  - CCS, communication & synchronisation
  - pi-calculus, mobile processes
  - various variations dealing with
    - time (for real-time descriptions)
    - probability
    - locality
    - mobility
- In engineering
  - differently from lambda-calculus, which is used only to define language properties
  - process algebras are very often used to describe the behaviour of interacting systems
- I give an incremental description of CCS

Op 1: action execution

- Give a set of actions a∈A (i/o, silent... whatever)
- Define
  - P ::= 0 | a.P
  - e.g.: Ps=a.a'.a".a.0 (more simply: a.a'.a".a)
  - Ps executes a, then a', then a", then again a, then terminates
- Which (operational) semantics?

$$a.P \xrightarrow{a} P$$

- Which evolution of Ps?

$$a.a'.a''.a \xrightarrow{a} a'.a''.a \xrightarrow{a'} a''.a \xrightarrow{a''} a \xrightarrow{a} 0$$

- Only one admissible interaction history [a,a',a",a]

Towards multiple histories

- Is it sensible to have systems allowing for just one interaction history?
- For instance:
  - we may want to model an unpredictable source of events
  - a system may receive at a time different kinds of stimulus
- In general:
  - we want to characterize a system in terms of all its admissible interaction histories...
  - we need an operator for non-determinism

Op 2: Nondeterministic choice

- Extending our algebra of processes:
  - P ::= 0 | a.P | P + P
  - e.g.: Ps=a.a'+a".a
  - Ps executes (a, then a') or (a" then a)
- Semantics

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$$

- With the hypothesis:
  - (P+0)=P, P+Q=Q+P, (P+Q)+R=P+(Q+R)

Choices..

- Example: P = a.a'+a".a

$$a.a' + a''.a \xrightarrow{a''} a \xrightarrow{a} 0$$

$$a.a' + a''.a \xrightarrow{a} a' \xrightarrow{a'} 0$$

- Two possible interaction histories, [a,a'] and [a",a]
- Notice:
  - a transition does not mean that the action has been executed
  - rather, that it can execute!
  - but this is, as usual, a matter of interpretation

Example

- A machine with one button and two leds
  - as the button is pressed either one of two leds is turned on
- Actions:
  - {press, led1 , led2}
- Observable behaviour
  - OB = {[press,led1],[press,led2]}
- Which specification?
  - P_1= press.(led1+led2)
  - P_2= press.led1+press.led2
- Are them the same?
  - they are in trace semantics, not in bisimulation!

Op 3: Parallel composition

- Another extension to our algebra
  - P ::= ... | P|P
  - e.g.: Pr= Ps|a.a" = (a.a'+a".a)|a.a"
  - Pr is the composition of Ps and a.a"
- Semantics

$$\frac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q} \qquad \frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'}$$

- With the hypothesis:
  - (P|0)=P, P|Q=Q|P, (P|Q)|R=P|(Q|R)
- This schema is called interleaved concurrency

Interleaved concurrency

- Consider the process Pr = (a.a'+a".a)|a.a" = Q|R
- There are three main available evolutions:

$$(a.a' + a''.a)|a.a'' \xrightarrow{a} (a.a' + a''.a)|a'' \xrightarrow{a} a'||a''$$

$$(a.a' + a''.a)|a.a'' \xrightarrow{a} a'|a.a'' \xrightarrow{a'} a.a''$$

$$(a.a' + a''.a)|a.a'' \xrightarrow{a''} a|a.a'' \xrightarrow{a} a.a''$$

- Q and R separately proceed on their evolution
  - OB(Q)={[a,a'],[a",a]}, OB(R)={[a,a"]}
  - OB(Q|R)={[a,a',a,a"],[a,a,a',a"],[a,a,a",a'],[a,a",a,a'],[a",a,a,a"],[a,a",a,a"],[a,a,a",a"]}

On distribution

- Suppose Q and R represent processes located at different sites
- In distributed systems there is no notion of global time and global state! So,
  - what is the meaning of Q|R ?
  - what does it mean that an action happens before another?

The distributed state

- Q|R is called the distributed state (is not the global state)
  - is a state observable by an entity navigating the distributed system
  - when transiting over the two sites, it sees states Q and R
- Transitions
  - if S moves to S' by action a,
  - the observer first sees S
  - on a subsequent trip sees S'
  - the second time it has transited over the system's subpart responsible for "a", after "a" occurred.

The distributed time

- Sequence of transitions
  - if S moves to S' by action a and then by a'
  - it does not mean that a happened before a' somewhere in S
  - we just know that there is no reason for a to wait before a' occurs...
  - ➢ there is no cause-effect relationship between a' and a
- Real-time, location, etc..., require extensions to this framework
- This notion does not impact very much!

Which concept of synchrony

- Until now, processes execute concurrently but do not influence each other
- What is synchrony?
  - when two processes are said to synchronize?
  - when they wait each other in a given state before carrying on
- Process R reaches a certain state R' and waits for the event to occur
- Process S reaches a certain state S' and waits for the event to occur
- When both processes reached their point, they synchronize, so they can both proceed

Synchrony

- Synchrony can be modelled by introducing a binary relation over actions: e.g. for some action "a" there is an action "$\underline{a}$" related to it.
  - in P|Q, P executes "a" only if Q executes "$\underline{a}$"
  - P|Q globally and atomically moves to P'|Q' in a silent way

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\underline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

- A useful interpretation
  - P wants to receive a stimulus by someone
  - Q sends that stimulus
  - actions τ means that P|Q evolves internally

Synchrony and distribution

- Is this notion of sincrony plausible in distributed scenarios?
- How do we realise the fact that P and Q evolves to P' and Q' in an atomic way?
- The only possibility is by using transactions..
  - a transaction manager (TM) ensures that they both evolve
  - any observer of P and Q perceives P and Q only through the TM, which ensures atomicity
  - as for distributed databases
- Transactions are costly in distributed systems (DS)
  - DS are generally asynchronous!

Finiteness

- So far, processes execute and eventually terminate
  - they are finite.
- We may want to describe processes that indefinitely have some interactive behaviour
- For instance:
  - a web server reply to requests many many times (we don't know how many)
  - possibly it can stop because of a particular interaction, but we do not know when
- We may need to deal with infinite histories....

Op 4: Replication

- Another extension to our algebra
  - P ::= ... | !P
  - e.g.: Pr= !a.a'
  - Pr executes a, then a', then again a and a', and so on...
- Semantics by the equivalence

$$!P \equiv P|!P$$

- Each time it is useful, !P can be rewritten as !P|P
- Example:

$$(!a)|\underline{a}|\underline{a} \equiv (!a)|a|\underline{a}|\underline{a} \xrightarrow{\tau} (!a)|\underline{a} \equiv (!a)|a|\underline{a} \xrightarrow{\tau} (!a)$$

Op 4b: Agent Definition

- An alternative definition of replication
  - more expressive from a programmer's viewpoint
- The definition of a process is to be associated to a number of agent definitions of the kind:
  - A(v1,..,vn) := P
  - (for some process P, possibly using v1,..,vn)
- Then the syntax of processes is extended by:
  - P ::= ... | A(a1,..,an)

$$A(a_1,..,a_n) \equiv P\{a_1/v_1,..,a_n/v_n\} \quad if \quad A(v_1,..,v_n) := P$$

Example

- Example:

$$Recur(v) := v.Recur(v)$$

$$Recur(a)|\underline{a}|\underline{a} \equiv a.Recur(a)|\underline{a}|\underline{a} \xrightarrow{\tau} Recur(a)|\underline{a} \equiv a.Recur(a)|\underline{a} \xrightarrow{\tau} Recur(a)$$

Compositionality

- Obtained by parallel composition and corresponding actions
- Example: consider processes
  - P = a.a'.a".a"', Q = $\underline{a}$.$\underline{a'}$.a"', R = $\underline{a''}$
- An evolution of P:

$$a.a'.a''.\underline{a'''} \xrightarrow{a} a'.a''.\underline{a'''} \xrightarrow{\underline{a'}} a''.\underline{a'''} \xrightarrow{\underline{a''}} \underline{a'''} \xrightarrow{\underline{a'''}} 0$$

- An evolution of P|Q

$$a.a'.a''.\underline{a'''}|\underline{a}.\underline{a'}.a''' \xrightarrow{\tau} a'.a''.\underline{a'''}|\underline{a'}.a''' \xrightarrow{\tau} a''.\underline{a'''}|.a''' \xrightarrow{\underline{a''}} \underline{a'''}|.a''' \xrightarrow{\tau} 0$$

- P|Q|R can terminate only through silent actions

Environment

- Process P by itself only proceeds by actions "a" and "$\underline{a}$"...
  - can be interpreted as the environment of P provides and gets the requested actions
    - p executes action a, the environment provides "$\underline{a}$"
    - p executes action $\underline{a}$, the environment provides "a"
- By composing to Q, some of this environment is now specified:
  - P proceeds a bit thanks to Q, but after a while it blocks again requiring the environment
- P|Q|R does not need the environment to proceed
- A new problem, what if the environment still provides a or a' to P|Q|R while it evolves?
  - P|Q|R does no longer reach termination

Operator 5: Name restriction

- By name restriction you can define an action that is private to a given process... technically, the environment cannot interact by "a"
- Extending the algebra:
  - P ::= .... | (a)P
  - e.g.: Ps= (a)a.a', in P a is private (∉FV), a' is public (∈FV)
  - FV, free variables of a process, those visible from outside
  - which semantics? has to do with renaming as in lambda....
- Two ideas:
  - You can always rename a private action of a process
  - If a process does not use an action, it is like that action is private for it...

Semantics restrictions

$$\begin{array}{ll} (a)0 \equiv 0 & \\ (a)(b)P \equiv (b)(a)P & \\ ((a)P) + Q) \equiv (a)(P + Q) & if \quad a \notin FV(Q) \\ ((a)P)|Q) \equiv (a)(P|Q) & if \quad a \notin FV(Q) \end{array}$$

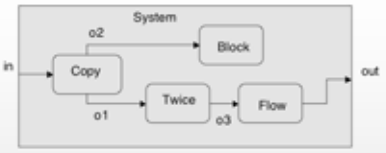
$$P|Q \equiv P\{b/a\}|Q \quad \text{if} \quad b \notin FV(P|Q) \wedge a \notin FV(P)$$

- Allows you to forget about the order of restrictions and about zero
- Says you can extend restriction if no conflicts occur
- In the case of conflicts you can rename a private action

$$((a)(a.a'|\underline{a}.a''))|\underline{a'}.\underline{a''} \xrightarrow{\tau} ((a)(a'|a''))|\underline{a'}.\underline{a''} \equiv (a)(a'|a''|\underline{a'}.\underline{a''}) \xrightarrow{\tau} \xrightarrow{\tau} (a)0 \equiv 0$$

$$((a)(a'|\underline{a}|a''))|a|\underline{a'} \equiv ((b)(b|\underline{b}|a''))|a|\underline{a'} \equiv (b)(b|\underline{b}|a'|a|\underline{a'})$$

A complete example



$$
\begin{array}{ll}
Twice(in,out) := & !in.\underline{out}.\underline{out} \\
Copy(in,out_1,out_2) := & !in.\underline{out_1}.\underline{out_2} \\
Block(in) := & !in \\
Flow(in,out) := & !in.\underline{out} \\
System(in,out) := & (o1)(o2)(o3)(\\
 & \quad Copy(in,o1,o2) \| Twice(o1,o3) \| Flow(o3,out) \| Block(o2))
\end{array}
$$

$$System(in,out) \mid \underline{in} \xrightarrow{\tau}{}^{*} System(in,out) \mid \underline{out} \mid \underline{out}$$

Value passing

- We also want to deal with communication of values, not just stimulus
- To do that, we first consider actions as being of three kinds
  - x(v), receiving a value from channel named "x", which is stored in variable "v"
    - (similar to an argument of a method...)
  - $\underline{x}v$, sending value "v" from channel named "x"
    - (similar to a method invocation)
  - τ as usual

$$\frac{P \xrightarrow{x w} P' \qquad Q \xrightarrow{x(v)} Q'}{P|Q \xrightarrow{\tau} P'|(Q'\{w/v\})}$$

Example:

$$x(v).\underline{y}v|\underline{x}5 \xrightarrow{\tau} \underline{y}5$$

CCS (Calculus for Communicating Systems)

- Turing Award for Robin Milner (1982...)

$$P ::= 0 \mid \tau.P \mid a(v).P \mid \overline{a}v.P \mid (P|P) \mid (P+P) \mid P/a \mid !P$$

- Features:
  - synchronous communications between processes
  - compositionality, non-determinism by choices, replication
- Fully exploited to verify properties of interactive systems, protocols..

Limits of CCS

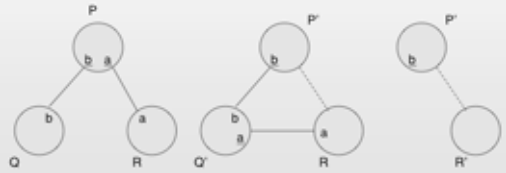
- Is not Turing-complete.... cannot represent recursion....
- What do we still lack?
  - interconnection of processes are fixed
  - how do we deal with mobility (physical or virtual)?
  - how do we deal with dynamic structures?

Pi-calculus

- Is a relatively simple modification to CCS
  - structures of processes may change
  - allows to represent any algorithm (Turing-complete)
  - it's the candidate for being the standard foundation of interactive languages
- Basic idea:
  - We do not have values (as in lambda, actually)
  - What processes exchanges are just channel names...
    - who receives it, has a new channel to communicate to
    - when the name is no more used.. is as the link disappears
- Syntax and semantics?
  - basically similar to CCS (only technical differences)

Process Mobility

$$
\begin{array}{l}
P := \overline{b}a.\overline{b}5.P' \\
Q := b(x).b(y).\overline{x}y.0 \\
R := a(v).R' \\
System := P \mid Q \mid R \\
System \xrightarrow{\tau}\xrightarrow{\tau} P' \mid \overline{a}5.0 \mid R \xrightarrow{\tau} P' \mid R'
\end{array}
$$

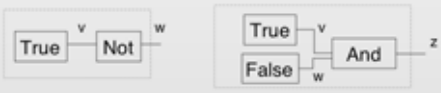


You can play with it...

- In the lambda calculus:
  - "values" are lambda-expressions accepting some argument and returning some other
    - you evaluate by applying values and observing results
    - T= λt.λf.t, F= λt.λf.f
  - "operations" are functions, accepting values and behaving as values
    - Not = λx.x F T
- In the pi-calculus:
  - "values" are processes "localized" at some channels...
  - a process accepts messages from that channels
  - replies by sending some message to the received inputs
  - operations: process to be composed to arguments

Logics in Pi

- Boolean values
  - True(x):=!x(t).x(f).t (t is the same as tt)
  - False(x):=!x(t).x(f).f
- Operations:
  - Not(x,y):= !x(t).x(f).yf.yt
  - (v)(True(v)|Not(v,w)) ≈ False(w)
  - And(x,y,z):=(c)(d)z(a).z(b).xc.xd.(d.b+c.(ya.yb))
  - (v)(w)(True(v)|False(w)|And(v,w,z)) ≈ False(z)



Other encodings?

- Mathematics
  - numbers as processes with channels (s,z)
  - Succ, Sum, Prod, Factorial, as processes
- Encoding of Lambda Calculus
  - any lambda can be encoded into a pi-process
  - (variables translated to channels)
  - application translated to
    - sending values
    - receiving a result
- Hence... pi is Turing-complete
  - but it represents interactions and composition

Isn't too big?

$$P ::= 0 \mid \tau.P \mid a(v).P \mid \overline{a}v.P \mid (P|P) \mid (P + P) \mid P/a \mid !P$$

...at least, with respect to lambda-calculus...

There are studies that make it simpler

- only asynchronous communications
- only sending new names
- no name restriction, no choice

- ai-pi-calculus

$$P ::= \overline{x}(y) \mid x(y).P \mid (P|P) \mid !P$$

Which applications?

- Similar to lambda-calculus for languages
  - providing a foundation to languages with interactive abilities
  - reasoning about interactive programs
  - ...
- Really useful on systems
  - describing the behaviour of interactive systems
  - proving properties of interest on protocols
  - ...
- Typically:
  - not encoding things in Pi (useful if you use verifiers...)
  - but writing our own language using similar features...

Domande all'esame

- Domande generali su quanto detto
- Esercizietti
  - Se da una lambda si può giungere ad un altra
  - Quale algoritmo è realizzato da una o più regole
  - Quali sono le histories di un processo CCS
  - Cosa succede applicando ad un dato processo una azione?
  - ...

Formal Models: from Algorithms to Distributed Computing

Mirko Viroli
DEIS, Università degli Studi di Bologna
mviroli@deis.unibo.it