Limits of Computation

11 - Church-Turing Thesis

Bernhard Reus

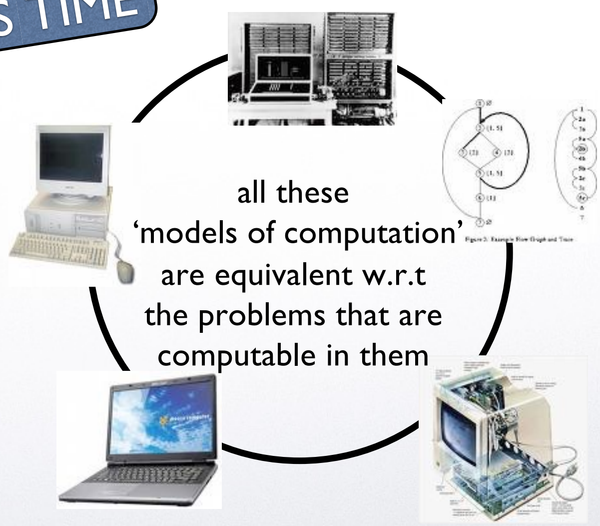
The story so far

- We have found problems that cannot be solved by a `WHILE` program (*Halting*, *Busy-Beaver*, *Tiling*, *Ambiguity of CFGs*, etc).
- But this does not mean yet they could not be solved by a different machine model (different definition of "effective procedure").

More ‘notions of computation’

THIS TIME

- Evidence for the
  - Church-Turing Thesis
- by looking at other notions of computation:
  - Register machines
  - Counter machines
  - Turing machines
  - Goto language
  - Cellular Automata
- and showing (or stating) that they are all equivalent.



Church-Turing Thesis

reasonable formalisations of the intuitive notion of effective computability

All *reasonable* computation models are *equivalent.*

no restrictions on memory size and execution time are assumed

So it does not matter what model we use.
It was alright to use `WHILE`.

We cannot prove this as it refers to informal entities (reasonable computation models). It is thus only a “thesis”. But we provide some *evidence* for it.

Models of computation

- Random Access Machines (register machines): RAM
- Turing machines: `TM`
- Counter machines: CM
- Flowchart language: GOTO
- Cellular Automata: CA
- While language: `WHILE` ✓
- Church's λ-Calculus (see course *Comparative Prog. Languages*)

Machine instructions

- `TM`, `GOTO`, `CM`, `RAM` have the following in common:
- A *program* is a sequence of instructions with labels

$$\boxed{\ell_1{:}\mathrm{I}_1;\ \ell_2{:}\mathrm{I}_2;\ \ldots\ \ell_n{:}\mathrm{I}_n}$$

- a *state* or *configuration* during execution of the program is of the form *(ℓ,σ)* where *ℓ* is the current instruction's label and σ is the "store" which varies from machine to machine.

`WHILE` does not have 'machine flavour', it's more abstract

CA quite different

Semantic Framework for Machines

- definition of store
- function *Readin* producing initial store
- function *Readout* producing output
- description of semantics of the instructions,

$$\text{p} \vdash s \to s'$$

program p transits form configuration s into s' in one step

Semantic Framework for machines

Definition (General framework for machine model semantics). Let p be a machine program with m instructions.

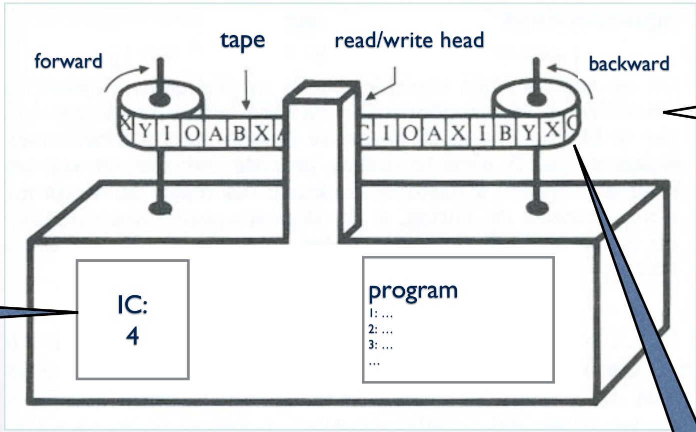
$$\begin{aligned} [\![\text{p}]\!](d) = e \ \text{ iff } \ & \sigma_0 = Readin(d) && \text{and} \\ & \text{p} \vdash (1, \sigma_0) \to^* (m+1, \sigma) && \text{and} \\ & e = Readout(\sigma) \end{aligned}$$

$$\text{p} \vdash s \to^* s'$$

"many-step" operational semantics for machine language, zero, one, or several steps of the "one-step" semantics

We will now show how semantics of other languages/machines are an instance of this framework.

Turing Machine



in this example: we have *1* tape

current instruction counter

tape symbols from a given finite alphabet; special symbol B (blank)

https://www.youtube.com/watch?v=cYw2ewoO6c4

Turing Machine

presented in a way to fit our machine model

j is number of tape

instruction	explanation
$\texttt{right}_j$	move right
$\texttt{left}_j$	move left
$\texttt{write}_j\ \texttt{S}$	write S
$\texttt{if}_j\ \texttt{S}\ \texttt{goto}\ \ell_1$ else ℓ_2	conditional jump (read)

we have k tapes

Store (= Tapes) $(L_1\underline{S_1}R_1, L_2\underline{S_2}R_2, \ldots L_k\underline{S_k}R_k)$

L_i, R_i are strings of symbols

_ denotes position of read/write head

Turing Machine

Readin (input) $\quad Readin(\mathtt{x}) = (\underline{\mathtt{B}}\mathtt{x}, \underline{\mathtt{B}}, \underline{\mathtt{B}}, \ldots, \underline{\mathtt{B}})$

Readout (output) $\quad Readout(\mathtt{L}_1\underline{\mathtt{S}_1}\mathtt{R}_1, \mathtt{L}_2\underline{\mathtt{S}_2}\mathtt{R}_2, \ldots \mathtt{L}_k\underline{\mathtt{S}_k}\mathtt{R}_k) = Prefix(\mathtt{R}_1)$

in both cases: non-blank string right of head on tape 1

where $Prefix(\mathtt{R}_1\mathtt{B}\mathtt{R}_2) = \mathtt{R}_1$ provided that $\mathtt{R}_1$ does not contain any blank symbols

One-step operational semantics for **1-tape** machine (no subscripts)

ℓ-th instruction

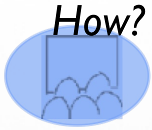
$$
\begin{array}{llll}
\mathtt{p} \vdash (\ell, \mathtt{L}\underline{\mathtt{S}}\mathtt{S'R}) & \to (\ell+1, \mathtt{LS}\underline{\mathtt{S'}}\mathtt{R}) & \text{if } \mathtt{p}(\ell) = \mathtt{right} \\
\mathtt{p} \vdash (\ell, \mathtt{L}\underline{\mathtt{S}}) & \to (\ell+1, \mathtt{LS}\underline{\mathtt{B}}) & \text{if } \mathtt{p}(\ell) = \mathtt{right} \\
\mathtt{p} \vdash (\ell, \mathtt{LS'}\underline{\mathtt{S}}\mathtt{R}) & \to (\ell+1, \mathtt{L}\underline{\mathtt{S'}}\mathtt{SR}) & \text{if } \mathtt{p}(\ell) = \mathtt{left} \\
\mathtt{p} \vdash (\ell, \underline{\mathtt{S}}\mathtt{R}) & \to (\ell+1, \underline{\mathtt{B}}\mathtt{SR}) & \text{if } \mathtt{p}(\ell) = \mathtt{left} \\
\mathtt{p} \vdash (\ell, \mathtt{L}\underline{\mathtt{S}}\mathtt{R}) & \to (\ell+1, \mathtt{L}\underline{\mathtt{S'}}\mathtt{R}) & \text{if } \mathtt{p}(\ell) = \mathtt{write}\ \mathtt{S'} \\
\mathtt{p} \vdash (\ell, \mathtt{L}\underline{\mathtt{S}}\mathtt{R}) & \to (\ell_1, \mathtt{L}\underline{\mathtt{S}}\mathtt{R}) & \text{if } \mathtt{p}(\ell) = \mathtt{if}\ \mathtt{S}\ \mathtt{goto}\ \ell_1\ \mathtt{else}\ \ell_2 \\
\mathtt{p} \vdash (\ell, \mathtt{L}\underline{\mathtt{S'}}\mathtt{R}) & \to (\ell_2, \mathtt{L}\underline{\mathtt{S'}}\mathtt{R}) & \text{if } \mathtt{p}(\ell) = \mathtt{if}\ \mathtt{S}\ \mathtt{goto}\ \ell_1\ \mathtt{else}\ \ell_2 \text{ and } \mathtt{S} \neq \mathtt{S'}
\end{array}
$$

GOTO

instruction	explanation
`X := nil`	assign nil
`X := Y`	assign variable
`X := hd Y`	assign `hd Y`
`X := tl Y`	assign `tl Y`
`X := cons Y Z`	assign `cons Y Z`
`if X goto` ℓ_1 `else` ℓ_2	conditional jump

Unconditional jump can be expressed as well

How?



- uses the tree data type we know from `WHILE`
- does only permit operations on variables
- `if X` tests whether `X` is not nil and then jumps according to a label
- store as for `WHILE`, *ReadIn* and *Readout* like in `WHILE` semantics of programs.

Sample Program

```
1: if X goto 2 else 6;
2: Y := hd X;
3: R := cons Y R;
4: X := tl X;
5: if R goto 1 else 1;
6: X := R;
```

GOTO programs not as readable as WHILE programs

What does it compute?

Semantics of GOTO

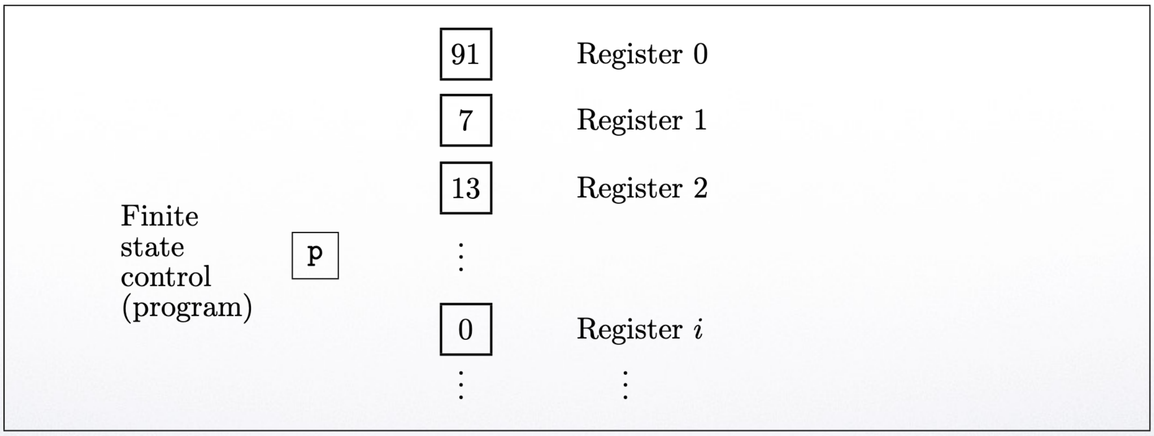
(cont'd)

store σ (sigma) where variable X is assigned value nil

ℓ-th instruction

$$
\begin{array}{lll}
\mathtt{p} \vdash (\ell, \sigma) \to (\ell+1, \sigma[\mathtt{X} := \mathrm{nil}]) & \text{if} & \mathtt{p}(\ell) = \mathtt{X\!:=\!nil} \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell+1, \sigma[\mathtt{X} := \sigma(\mathtt{Y})]) & \text{if} & \mathtt{p}(\ell) = \mathtt{X\!:=\!Y} \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell+1, \sigma[\mathtt{X} := d]) & \text{if} & \mathtt{p}(\ell) = \mathtt{X\!:=\!hd\ Y} \text{ and } \sigma(\mathtt{Y}) = \langle d.e \rangle \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell+1, \sigma[\mathtt{X} := \mathrm{nil}]) & \text{if} & \mathtt{p}(\ell) = \mathtt{X\!:=\!hd\ Y} \text{ and } \sigma(\mathtt{Y}) = \mathrm{nil} \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell+1, \sigma[\mathtt{X} := e]) & \text{if} & \mathtt{p}(\ell) = \mathtt{X\!:=\!tl\ Y} \text{ and } \sigma(\mathtt{Y}) = \langle d.e \rangle \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell+1, \sigma[\mathtt{X} := \mathtt{nil}]) & \text{if} & \mathtt{p}(\ell) = \mathtt{X\!:=\!tl\ Y} \text{ and } \sigma(\mathtt{Y}) = \mathrm{nil} \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell+1, \sigma[\mathtt{X} := \langle d.e \rangle]) & \text{if} & \mathtt{p}(\ell) = \mathtt{X\!:=\!cons\ Y\ Z} \text{ and } \sigma(\mathtt{Y}) = d \text{ and } \sigma(\mathtt{Z}) = e \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell_1, \sigma) & \text{if} & \mathtt{p}(\ell) = \mathtt{if\ X\ goto}\ \ell_1\ \mathtt{else}\ \ell_2 \text{ and } \sigma(\mathtt{X}) \neq \mathrm{nil} \\
\mathtt{p} \vdash (\ell, \sigma) \to (\ell_2, \sigma) & \text{if} & \mathtt{p}(\ell) = \mathtt{if\ X\ goto}\ \ell_1\ \mathtt{else}\ \ell_2 \text{ and } \sigma(\mathtt{X}) = \mathrm{nil}
\end{array}
$$

RAM model



- uses *arbitrarily many* registers containing *arbitrarily big* numbers.

(S)RAM instruction set

Successor RAM is like RAM but without binary operations

instruction		explanation
`Xi := 0`		reset register value
`Xi := Xi+1`		increment register value
`Xi := Xi∸1`		decrement register value
`Xi := Xj`		move register value
`Xi := <Xj>`	indirect addressing	move content of register addressed by `Xj`
`<Xi> := Xj`		move into register addressed by `Xi`
`if Xi=0 goto` ℓ_1 `else` ℓ_2		conditional jump
available only in RAM		
`Xi := Xj + Xk`		addition of register values
`Xi := Xj * Xk`		multiplication of register values

- data type of *natural numbers*
- angle brackets < > *indirect* addressing
- if-goto-else tests whether X is 0 and then *jumps* accordingly.

Semantics SRAM

$\texttt{SRAM-}\textit{store} = \{ \sigma \mid \sigma : I\!N \to I\!N \}$

$\textit{Readin}(\texttt{x})$	$=$	$\{0{:}x, 1{:}0, 2{:}0, \ldots\}$	Input in register X0
$\textit{Readout}(\sigma)$	$=$	$\sigma(0)$	From register X0

$$
\begin{array}{lll}
\mathtt{p} \vdash (\ell,\sigma) \to (\ell+1, \sigma[i := \sigma(i)+1]) & \text{if} & \mathtt{p}(\ell) = \mathtt{Xi:=Xi+1} \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell+1, \sigma[i := \sigma(i)-1]) & \text{if} & \mathtt{p}(\ell) = \mathtt{Xi:=Xi\dot{-}1} \text{ and } \sigma(i) > 0 \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell+1, \sigma[i := 0]) & \text{if} & \mathtt{p}(\ell) = \mathtt{Xi := Xi\dot{-}1} \text{ and } \sigma(i) = 0 \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell+1, \sigma[i := \sigma(j)]) & \text{if} & \mathtt{p}(\ell) = \mathtt{Xi:=Xj} \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell+1, \sigma[i := 0]) & \text{if} & \mathtt{p}(\ell) = \mathtt{Xi:=0} \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell_1, \sigma) & \text{if} & \mathtt{p}(\ell) = \mathtt{if\ Xi} = 0 \mathtt{\ goto\ } \ell_1 \mathtt{\ else\ } \ell_2 \text{ and } \sigma(\mathtt{Xi}) = 0 \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell_2, \sigma) & \text{if} & \mathtt{p}(\ell) = \mathtt{if\ Xi} = 0 \mathtt{\ goto\ } \ell_1 \mathtt{\ else\ } \ell_2 \text{ and } \sigma(\mathtt{Xi}) \neq 0 \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell+1, \sigma[i := \sigma(\sigma(j))]) & \text{if} & \mathtt{p}(\ell) = \mathtt{Xi:=<Xj>} \\
\mathtt{p} \vdash (\ell,\sigma) \to (\ell+1, \sigma[\sigma(i) := \sigma(j)]) & \text{if} & \mathtt{p}(\ell) = \mathtt{<Xi>:=Xj}
\end{array}
$$

Counter Machine CM

$\mathtt{I} \quad ::= \quad \mathtt{Xi\ :=\ Xi\ +\ 1} \mid \mathtt{Xi\ :=\ Xi\ \dot{-}\ 1} \mid \mathtt{if\ Xi=0\ goto}\ \ell\ \mathtt{else}\ \ell'$

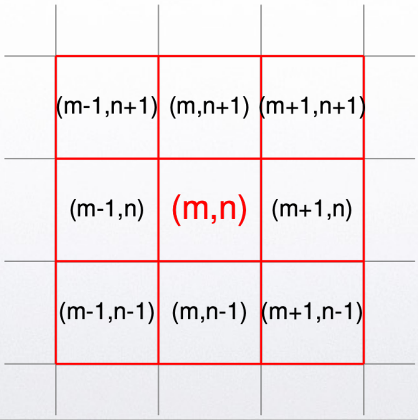
- Counter machines are much simpler than register machines.
- They contain several registers, called *counters* as they can only be incremented or decremented and tested for zero.
- 2CM is like CM but with 2 counters only.
- Semantics as for register machines.

Cellular Automata CA

we just focus one specific version the famous 2-dimensional CA called (Conway's) ***Game of Life***

John Conway (Cambridge Mathematician)

neighbourhood of (m,n)
= 8 cells



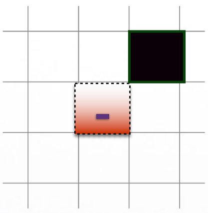
cell lattice (grid)

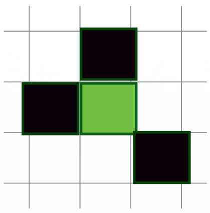
each cell contains 0 or 1

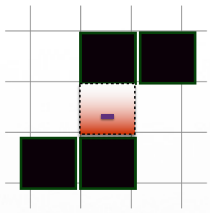
the value of a cell changes every "time tick" and the new value is determined only by the values of the neighbourhood cells

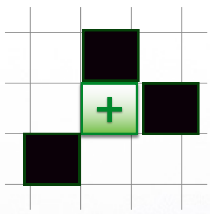
Rules of Game of Life

alive = 1 (solid line/filled)
dead = 0 (no colour)









Underpopulation

die if fewer than two neighbours are alive

Survival

survive (stay alive) if two or three neighbours are alive

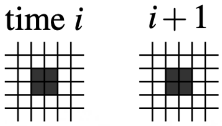
Overcrowding

die if more than three neighbours are alive

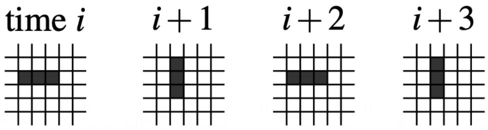
Reproduction

become alive if exactly three neighbours are alive

Patterns of Game of Life

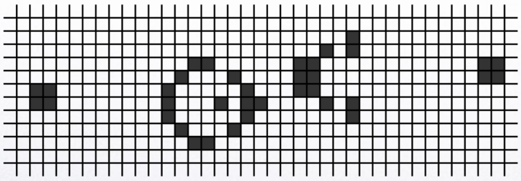


static



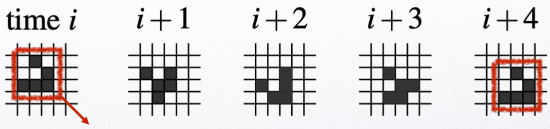
https://www.youtube.com/watch?v=OrCTmfQWCmQ

oscillating (dynamic)



https://www.youtube.com/watch?v=GrlO5RJ76D0

productive:
Gosling's Glider Gun



one-cell diagonal movement
of glider (dynamic)

Cellular Automata

- Fun, but can they compute in our sense?
- Yes, one has to:
  - encode input as starting grid
  - result if a predefined cell changes its state to a predefined "accepting state"
  - "accepting state" must be left alone in further grid changes.

Robustness of Computability

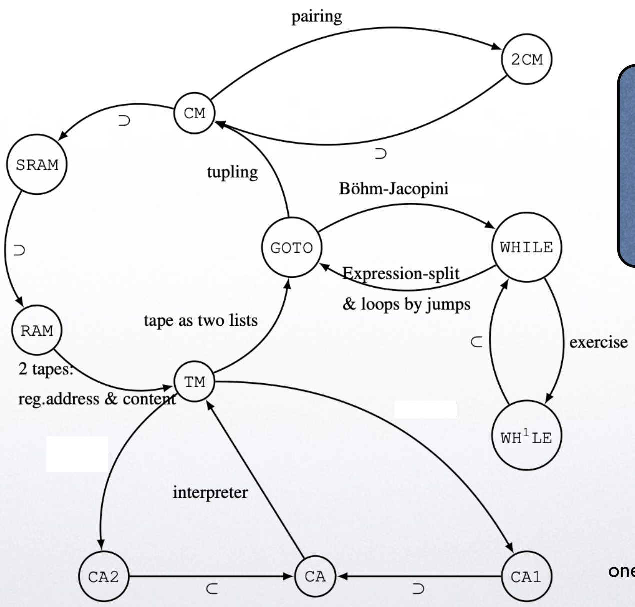
Church-Turing Thesis

Robustness of Computability

- We justify the Church-Turing thesis ...
- ... by showing that all the above notions of computation are equivalent to `WHILE`.
- Proof technique:
  - compile one program of language *X* into an equivalent one of language *Y* ...
  - ... if *X* is not a sub-language of *Y* anyway.
  - compose compilers appropriately to avoid having to write n^2 compilers:

Diagram of Equivalences

The label of each arrow is either ⊂ if one language is included in the other or the name/idea of the required **compilation.**



to compile from one language to another just compose the compilers on the arrows you have to travel along

Details of compilations in N. Jones' book, Chapter 8, exercises.

two-dimensional CA

one-dimensional CA

https://www.youtube.com/watch?v=My8AsV7bA94 https://www.youtube.com/watch?v=xP5-iIeKXE8

END

Next time:
Moving on to Complexity.
How do we measure time usage?