



◀ ▶

# Limits of Computation

---

I2 - Measuring Time Usage

Bernhard Reus



◀ ▶

## The story so far

- We have discussed “computability”,
- encountered computable and non-computable problems,
- discussed Church-Turing Thesis.



◀ ▶

THIS TIME

# Time Complexity

- From now on restrict interest to *computable, i.e. decidable, problems*
- *measuring the running time of programs*
- *compare language (simulation up to certain factor in runtime)*



image: mindhacks.com/blog



◀ ▶

## Unit-cost time measure

**Definition (unit-cost measure).** For an imperative language  $L$ , the function  $\text{time}^L : L\text{-program} \rightarrow L\text{-data} \rightarrow \mathbb{N}_\perp$  is defined as follows: For any  $p \in L\text{-program}$ , and  $d \in L\text{-data}$  we define

$$\text{time}_p^L(d) = \begin{cases} t + 1 & \text{if } p \vdash s_1 \rightarrow s_2 \dots \rightarrow s_t \text{ where } s_1 = \text{Readin}(d) \text{ and } s_t \text{ terminal} \\ \perp & \text{otherwise} \end{cases}$$

operational semantics  
(need something different for WHILE)

With any completed program execution associate the number of its transition steps (according to its semantic description);  
“read input” is implicitly one step;



◀ ▶

# Time measures are functions

- Note that for each program its running time measure is a *function*: running time depends on data input.
- Time measure function is a *partial function*: the program may not terminate on some input in which case there is nothing to measure.



◀ ▶

## Is this measure really “fair”?

- What counter arguments are there?
  - what about complexity of expressions (like equality = or cons) in WHILE?
  - what about unbounded size of numbers in RAM registers?
- thus *transition step (unit cost) measure* in use for TM, GOTO and CM but *not for language WHILE* with tree expressions (GOTO does not use nested expressions just variables, so it's fine).
- SRAM/RAM/CM needs some more discussion:



is unit measure ok for those models?

# Measuring RAM & CM

- length of addresses and byte representation of numbers, shouldn't they be considered?
- in most machines fixed length (32bit, 64bit) so maybe not. But this is ok only for small numbers (no register overflow).
- problematic if larger numbers are computed, so for RAM a *logarithmic cost model* is needed that takes length of addresses and content into account. Interested students are referred to Neil Jones's book.
- But SRAM does only allow +1 in one step, so unit measure is fine.
- CM model fine but measure does not give much insight as costs always extremely high.



# Time cost for WHILE

**Definition** (time measure for WHILE-expressions). The function  $\mathcal{T}$  maps WHILE-expressions into the time it takes to evaluate them and thus has type:

$$\mathcal{T} : \text{WHILE-Expressions} \rightarrow \mathbb{N}$$

It is defined inductively according to the shape of expressions as follows:

$$\begin{aligned}\mathcal{T}_{\text{nil}} &= 1 && \text{special atom} \\ \mathcal{T}_X &= 1 && \text{where } X \text{ is variable} \\ \mathcal{T}_{\text{hd } E} &= 1 + \mathcal{T}_E \\ \mathcal{T}_{\text{tl } E} &= 1 + \mathcal{T}_E \\ \mathcal{T}_{\text{cons } E \ F} &= 1 + \mathcal{T}_E + \mathcal{T}_F\end{aligned}$$

- Slogan: count operations and constants in expressions.
- Example:  $\mathcal{T}(\text{cons } \text{hd } x \ \text{tl } \text{nil}) = 1 + (1+1) + (1+1) = 5$



# Time cost for WHILE (cont'd)

**Definition** (time measure for WHILE-commands statement lists). For a store  $\sigma$  the relation  $S \vdash^{time} \sigma \Rightarrow t$  states that executing WHILE-statement list  $S$  in store  $\sigma$  takes  $t$  units of time. Deviating minimally from the grammar in Lecture 3 we allow statement lists to be empty here to avoid unnecessary distinction between empty and non-empty blocks. The relation  $\bullet \vdash^{time} \bullet \Rightarrow \bullet \subseteq \text{StatementList} \times \text{Store} \times \mathbb{N}$  is defined as the smallest relation satisfying the rules below.

$$X := E \quad \vdash^{time} \sigma \Rightarrow t + 1 \quad \text{if } \mathcal{T}_E = t$$



# Time cost for WHILE (cont'd)

$$\text{if } E \{ S_T \} \text{ else } \{ S_E \} \vdash^{time} \sigma \Rightarrow t + 1 + t' \text{ if } \begin{array}{l} \mathcal{E}[E]\sigma = \text{nil}, \quad \mathcal{T}_E = t \text{ and} \\ S_E \vdash^{time} \sigma \Rightarrow t' \end{array}$$

$$\text{if } E \{ S_T \} \text{ else } \{ S_E \} \vdash^{time} \sigma \Rightarrow t + 1 + t' \text{ if } \begin{array}{l} \mathcal{E}[E]\sigma \neq \text{nil}, \quad \mathcal{T}_E = t \text{ and} \\ S_T \vdash^{time} \sigma \Rightarrow t' \end{array}$$

$$\text{while } E \{ S \} \quad \vdash^{time} \sigma \Rightarrow t + 1 \quad \text{if } \mathcal{E}[E]\sigma = \text{nil}, \quad \mathcal{T}_E = t$$

$$\text{while } E \{ S \} \quad \vdash^{time} \sigma \Rightarrow t + 1 + t' \text{ if } \begin{array}{l} \mathcal{E}[E]\sigma \neq \text{nil}, \quad \mathcal{T}_E = t \text{ and} \\ S; \text{while } E \{ S \} \vdash^{time} \sigma \Rightarrow t' \end{array}$$



# Time cost for WHILE (cont'd)

statement lists for blocks

$$C; S \quad \vdash^{time} \sigma \Rightarrow t + t' \quad \text{if } \begin{array}{l} C \vdash^{time} \sigma \Rightarrow t, C \vdash \sigma \rightarrow \sigma' \\ \text{and } S \vdash^{time} \sigma' \Rightarrow t' \end{array}$$

for empty blocks

$$\vdash^{time} \sigma \Rightarrow 0$$



# Time cost for WHILE (cont'd)

**Definition (time measure for WHILE-programs).** For a WHILE-program  $p = \text{read } X \{S\} \text{ write } Y$  we define the time measure

$$time_{\bullet}^{\text{WHILE}} : \text{WHILE-program} \rightarrow \mathbb{D} \rightarrow \mathbb{N}_{\perp}$$

i.e. the time it takes  $p$  to run with input  $d$  as follows:

$$time_p^{\text{WHILE}}(d) = \begin{cases} t + 2 & \text{iff } S \vdash^{time} \sigma_0^p(d) \Rightarrow t \\ \perp & \text{otherwise} \end{cases}$$

In other words, the runtime of a program  $p$  with input  $d$  is the time it takes to execute the body of the program in the corresponding initial state plus two for reading the input and writing the output.



# Timed Prog. Language

**Definition** (timed programming language). A timed programming language consists of

syntax of programs and data type

1. two sets, namely L-programs and L-data
2. a function  $\llbracket \_ \rrbracket^L : L\text{-programs} \rightarrow (L\text{-data} \rightarrow L\text{-data}_\perp)$  that describes the semantics of L and
3. a function  $time_\bullet^L : L\text{-programs} \rightarrow (L\text{-data} \rightarrow \mathbb{N}_\perp)$ , the time measure for L, such that for every  $p \in L\text{-programs}$  and  $d \in L\text{-data}$  we have that  $\llbracket p \rrbracket^L(d) \uparrow$  if, and only if,  $time_p^L(d) \uparrow$ .

program semantics

time measure



# Comparing Languages

**Definition** (simulation relation). Suppose we are given two *timed* programming languages L and M with L-data = M-data. We define

1.  $L \preceq^{ptime} M$  if for every L-program  $p$  there is an M-program  $q$  such that  $\llbracket p \rrbracket^L = \llbracket q \rrbracket^M$  and a polynomial  $f(n)$  such that for all  $d \in L\text{-data}$

$$time_q^M(d) \leq f(time_p^L(d))$$

M can simulate L up to polynomial difference in time



# Comparing Languages II

2.  $L \preceq^{lintime} M$  if for every L-program  $p$  there is a constant  $a_p \geq 0$  and an M-program  $q$  such that  $\llbracket p \rrbracket^L = \llbracket q \rrbracket^M$  such that for all  $d \in L\text{-data}$

$$time_q^M(d) \leq a_p \times time_p^L(d)$$

M can simulate L up to linear difference in time

one constant for all programs

$L \preceq^{lintime-pg-ind} M$  if there is a constant  $a \geq 0$  such that for every L-program  $p$  there is an M-program  $q$  such that  $\llbracket p \rrbracket^L = \llbracket q \rrbracket^M$  for all  $d \in L\text{-data}$

$$time_q^M(d) \leq a \times time_p^L(d)$$

M can simulate L up to a program-independent linear time difference



# Comparing Languages III

We can now define "equivalent up to ... simulation"

**Definition (simulation equivalence).** Suppose we are given two *timed* programming languages L and M with L-data = M-data. We define

1.  $L \equiv^{lintime-pg-ind} M$  if, and only if,  $L \preceq^{lintime-pg-ind} M$  and  $M \preceq^{lintime-pg-ind} L$ . In words: L and M are *strongly linearly equivalent*.
2.  $L \equiv^{lintime} M$  if, and only if,  $L \preceq^{lintime} M$  and  $M \preceq^{lintime} L$ . In words: L and M are *linearly equivalent*.
3.  $L \equiv^{ptime} M$  if, and only if,  $L \preceq^{ptime} M$  and  $M \preceq^{ptime} L$ . In words: L and M are *polynomially equivalent*.



◀ ▶

# Comparing Languages III

- The simulation relation (of languages) is transitive (Exercises).
- The equivalence relation (of languages) is transitive (follows from the definition and above).
- Simulation is shown by compiling and comparing runtime of compiled vs original program.



◀ ▶

# END

© 2008-19. Bernhard Reus, University of Sussex

Next time:  
Defining complexity classes