Limits of Computation

12 - Measuring Time Usage
Bernhard Reus

The story so far

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

Time Complexity

THIS TIME

- 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 12.1 (unit-cost measure). For an imperative language $\mathtt{L}$, the function $time^{\mathtt{L}} : \mathtt{L}\text{−program} \to \mathtt{L}\text{−data} \to \mathbb{N}_{\bot}$ is defined a follows: For any $p \in \mathtt{L}$-program, and $d \in \mathtt{L}$-data we define

IN means natural numbers

$$time^{\mathtt{L}}_{p}(d) = \begin{cases} t+1 \text{ if } p \vdash s_1 \to s_2 \ldots \to s_t \text{ where } s_1 = Readin(d) \text{ and } s_t \text{ terminal} \\ \bot \quad \text{otherwise} \end{cases}$$

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" & "write output" is implicitly one step each;

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

$$\mathscr{T} : \texttt{WHILE}\text{-Expressions} \to \mathbb{N}$$

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

$$\begin{array}{lll}
\mathscr{T}\texttt{nil} & = 1 & \text{special atom} \\
\mathscr{T}\texttt{X} & = 1 & \text{where } \texttt{X} \text{ is variable} \\
\mathscr{T}\texttt{hd E} & = 1 + \mathscr{T}\texttt{E} & \\
\mathscr{T}\texttt{tl E} & = 1 + \mathscr{T}\texttt{E} & \\
\mathscr{T}\texttt{cons E F} & = 1 + \mathscr{T}\texttt{E} + \mathscr{T}\texttt{F} &
\end{array}$$

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

Time cost for WHILE (cont'd)

Definition 12.3 (time measure for `WHILE`-commands statement lists). For a store σ the relation $\mathtt{S} \vdash^{time} \sigma \Rightarrow t$ states that executing `WHILE`-statement list `S` in store σ takes t units of time. Deviating minimally from the grammar in Figure 3.23 we allow statement lists to be empty here to avoid unnecessary distinction between empty and non-empty blocks. The relation $\bullet \vdash^{time} \bullet \Rightarrow \bullet \subseteq \text{StatementList} \times \mathit{Store} \times \mathbb{N}$ is defined as the smallest relation satisfying the rules in Fig. 12.1.

$$\mathtt{X := E} \qquad \vdash^{time} \sigma \Rightarrow t+1 \qquad \text{if} \quad \mathscr{T}\,\mathtt{E} = t$$

Time cost for WHILE (cont'd)

$$\mathtt{if\ E\ \{S_T\}\ else\ \{S_E\}} \vdash^{time} \sigma \Rightarrow t+1+t' \ \text{ if } \ \mathscr{E}[\![\mathtt{E}]\!]\sigma = \text{nil}, \quad \mathscr{T}\,\mathtt{E} = t \ \text{ and } \ \mathtt{S_E} \vdash^{time} \sigma \Rightarrow t'$$

$$\mathtt{if\ E\ \{S_T\}\ else\ \{S_E\}} \vdash^{time} \sigma \Rightarrow t+1+t' \ \text{ if } \ \mathscr{E}[\![\mathtt{E}]\!]\sigma \neq \text{nil}, \quad \mathscr{T}\,\mathtt{E} = t \ \text{ and } \ \mathtt{S_T} \vdash^{time} \sigma \Rightarrow t'$$

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

$$\mathtt{while\ E\ \{S\}} \qquad \vdash^{time} \sigma \Rightarrow t+1+t' \ \text{ if } \ \mathscr{E}[\![\mathtt{E}]\!]\sigma \neq \text{nil}, \quad \mathscr{T}\,\mathtt{E} = t \ \text{ and } \ \mathtt{S;while\ E\ \{S\}} \vdash^{time} \sigma \Rightarrow t'$$

Time cost for WHILE (cont'd)

statement lists for blocks

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

for empty blocks

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

Time cost for WHILE (cont'd)

Definition 12.4 (time measure for WHILE-programs). For a WHILE-program $p =$ `read X {S} write Y` we define the time measure

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

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

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

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 12.5 (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 $[\![\_]\!]^{\mathtt{L}}$: L-programs $\to$ (L-data $\to$ L-data$_\perp$) that describes the semantics of L and

   program semantics

3. a function $time^{\mathtt{L}}_{\bullet}$: L-programs $\to$ (L-data $\to \mathbb{N}_\perp$), the time measure for L, such that for every $p \in$ L-programs and $d \in$ L-data we have that $[\![p]\!]^{\mathtt{L}}(d)\uparrow$ if, and only if, $time^{\mathtt{L}}_{p}(d)\uparrow$.

   time measure

Comparing Languages

Definition 12.6 (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 $[\![p]\!]^{\mathtt{L}} = [\![q]\!]^{\mathtt{M}}$ and a polynomial $f(n)$ such that for all $d \in$ L-data

$$time^{\mathtt{M}}_{q}(d) \le f(time^{\mathtt{L}}_{p}(d))$$

M can simulate L up to polynomial difference in time

Comparing Languages II

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

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

M can simulate L up to linear difference in time

one constant for all programs

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

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

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

Comparing Languages III

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

Definition 12.7 (simulation equivalence). Suppose we are given two *timed* programming languages $\mathtt{L}$ and $\mathtt{M}$ with $\mathtt{L}$-data = $\mathtt{M}$-data. We define

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

Next time:
Defining complexity classes