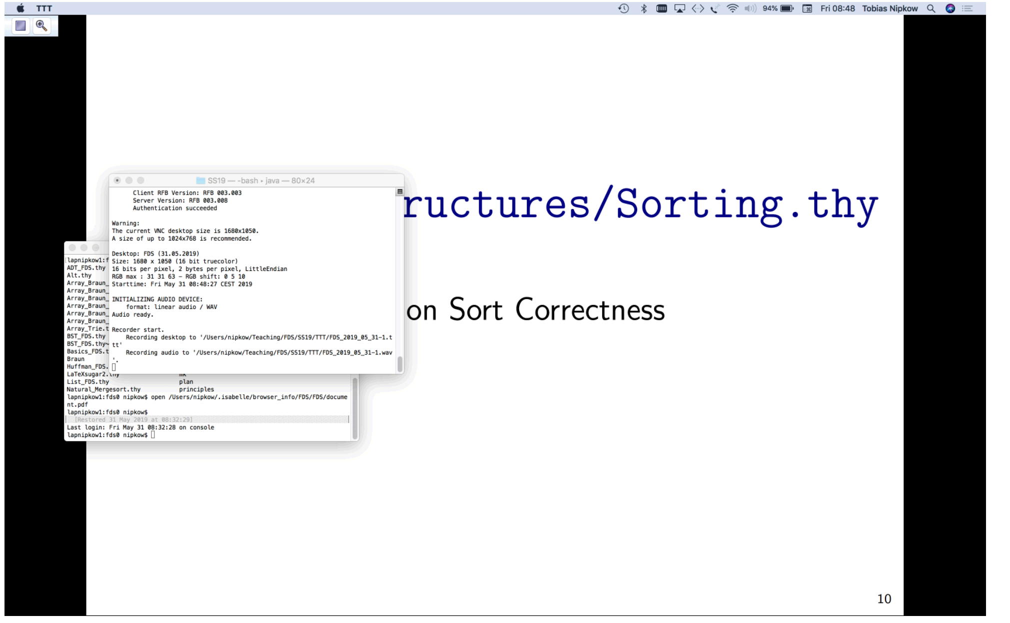
Script **generated by TTT**

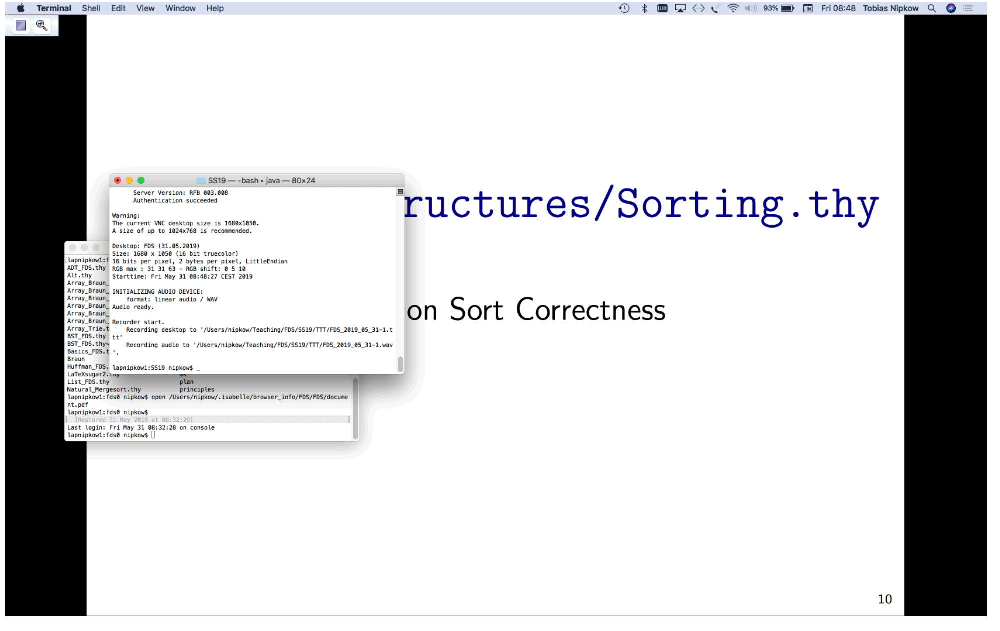
Title: FDS (31.05.2019)

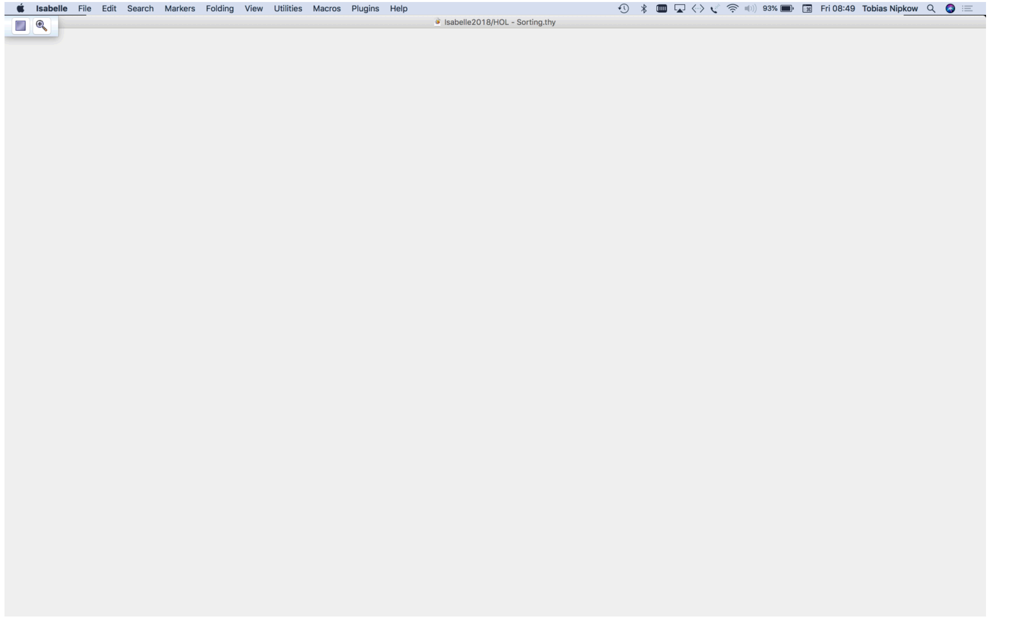
Date: Fri May 31 08:48:36 CEST 2019

Duration: 79:47 min

Pages: 49







Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:
Translation of defining equations:

$$\frac{e \leadsto e'}{f\ p_1 \ldots p_n = e \ \leadsto \ t\_f\ p_1 \ldots p_n = e' + 1}$$

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:
Translation of defining equations:

$$\frac{e \leadsto e'}{f\ p_1 \ldots p_n = e \ \leadsto \ t\_f\ p_1 \ldots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \leadsto t_1 \quad \ldots \quad s_k \leadsto t_k}{g\ s_1 \ldots s_k \leadsto t_1 + \cdots + t_k + t\_g\ s_1 \ldots s_k}$$

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:
Translation of defining equations:

$$\frac{e \rightsquigarrow e'}{f\, p_1 \ldots p_n = e \;\rightsquigarrow\; t\_f\, p_1 \ldots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \rightsquigarrow t_1 \quad \ldots \quad s_k \rightsquigarrow t_k}{g\, s_1 \ldots s_k \rightsquigarrow t_1 + \cdots + t_k + t\_g\, s_1 \ldots s_k}$$

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:
Translation of defining equations:

$$\frac{e \rightsquigarrow e'}{f\, p_1 \ldots p_n = e \;\rightsquigarrow\; t\_f\, p_1 \ldots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \rightsquigarrow t_1 \quad \ldots \quad s_k \rightsquigarrow t_k}{g\, s_1 \ldots s_k \rightsquigarrow t_1 + \cdots + t_k + t\_g\, s_1 \ldots s_k}$$

- Variable $\rightsquigarrow$ 0, Constant $\rightsquigarrow$ 0

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$
define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:
Translation of defining equations:

$$\frac{e \rightsquigarrow e'}{f\, p_1 \ldots p_n = e \;\rightsquigarrow\; t\_f\, p_1 \ldots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \rightsquigarrow t_1 \quad \ldots \quad s_k \rightsquigarrow t_k}{g\, s_1 \ldots s_k \rightsquigarrow t_1 + \cdots + t_k + t\_g\, s_1 \ldots s_k}$$

- Variable $\rightsquigarrow$ 0, Constant $\rightsquigarrow$ 0
- Constructor calls and primitive operations on $bool$ and numbers cost 1

Example

$app\ []\ ys = ys$

Example

$app\ []\ ys = ys$

$\rightsquigarrow$

$t\_app\ []\ ys = 0 + 1$

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$

define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:

Translation of defining equations:

$$\frac{e \rightsquigarrow e'}{f\ p_1 \dots p_n = e \ \rightsquigarrow\ t\_f\ p_1 \dots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \rightsquigarrow t_1 \quad \dots \quad s_k \rightsquigarrow t_k}{g\ s_1 \dots s_k \rightsquigarrow t_1 + \dots + t_k + t\_g\ s_1 \dots s_k}$$

- Variable $\rightsquigarrow$ 0, Constant $\rightsquigarrow$ 0
- Constructor calls and primitive operations on *bool* and numbers cost 1

Example

$app\ []\ ys = ys$

$\rightsquigarrow$

$t\_app\ []\ ys = 0 + 1$

13

Principle: Count function calls

For every function $f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau$

define a *timing function* $t\_f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow nat$:

Translation of defining equations:

$$\frac{e \rightsquigarrow e'}{f\ p_1 \dots p_n = e \ \rightsquigarrow\ t\_f\ p_1 \dots p_n = e' + 1}$$

Translation of expressions:

$$\frac{s_1 \rightsquigarrow t_1 \quad \dots \quad s_k \rightsquigarrow t_k}{g\ s_1 \dots s_k \rightsquigarrow t_1 + \dots + t_k + t\_g\ s_1 \dots s_k}$$

- Variable $\rightsquigarrow$ 0, Constant $\rightsquigarrow$ 0
- Constructor calls and primitive operations on *bool* and numbers cost 1

12

Example

$app\ []\ ys = ys$

$\leadsto$

$t\_app\ []\ ys = 0 + 1$

$app\ (x\#xs)\ ys = x\ \#\ app\ xs\ ys$

Example

$app\ []\ ys = ys$

A compact formulation of $e \leadsto t$

A compact formulation of $e \leadsto t$

t is the sum of all $t\_g\ s_1\ \ldots\ s_k$
such that $g\ s_1\ \ldots\ s_k$ is a subterm of e

A compact formulation of $e \leadsto t$

t is the sum of all $t\_g\ s_1\ \ldots\ s_k$
such that $g\ s_1\ \ldots\ s_k$ is a subterm of e

If g is

- a constructor or
- a predefined function on $bool$ or numbers

then $t\_g\ \ldots = 1$.

A compact formulation of $e \leadsto t$

Example

$app\ []\ ys = ys$

if and *case*

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated lazily.
Translation:

$$\frac{b \leadsto t \quad s_1 \leadsto t_1 \quad s_2 \leadsto t_2}{\textit{if } b \textit{ then } s_1 \textit{ else } s_2 \leadsto t + (\textit{if } b \textit{ then } t_1 \textit{ else } t_2)}$$

if and *case*

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated lazily.
Translation:

$$\frac{b \rightsquigarrow t \quad s_1 \rightsquigarrow t_1 \quad s_2 \rightsquigarrow t_2}{\textit{if } b \textit{ then } s_1 \textit{ else } s_2 \rightsquigarrow t + (\textit{if } b \textit{ then } t_1 \textit{ else } t_2)}$$

Similarly for *case*

$O(.)$ is enough

$O(.)$ is enough

$\Longrightarrow$ Reduce all additive constants to 1

A compact formulation of $e \rightsquigarrow t$

$O(.)$ is enough

$\Longrightarrow$ Reduce all additive constants to 1

Example

$t\_app\ (x\#xs)\ ys = t\_app\ xs\ ys + 1$

Discussion

- The definition of t_f from f can be automated.
- The correctness of t_f could be proved w.r.t. a semantics that counts computation steps.

Discussion

- The definition of t_f from f can be automated.
- The correctness of t_f could be proved w.r.t. a semantics that counts computation steps.
- Precise complexity bounds (as opposed to $O(.)$) would require a formal model of (at least) the compiler and the hardware.

`HOL/Data_Structures/Sorting.thy`

Insertion sort complexity

HOL/Data_Structures/Sorting.thy

merge :: 'a *list* ⇒ 'a *list* ⇒ 'a *list*

merge :: 'a *list* ⇒ 'a *list* ⇒ 'a *list*

merge [] *ys* = *ys*
merge xs [] = *xs*
merge (*x* # *xs*) (*y* # *ys*) =
(if $x \le y$ then *x* # *merge xs* (*y* # *ys*)
 else *y* # *merge* (*x* # *xs*) *ys*)

merge :: 'a *list* ⇒ 'a *list* ⇒ 'a *list*

merge [] *ys* = *ys*
merge xs [] = *xs*
merge (*x* # *xs*) (*y* # *ys*) =
(if $x \le y$ then *x* # *merge xs* (*y* # *ys*)
 else *y* # *merge* (*x* # *xs*) *ys*)

msort :: 'a *list* ⇒ 'a *list*

msort xs =
(let *n* = *length xs*
 in if $n \le 1$ then *xs*
 else *merge* (*msort* (*take* (*n div* 2) *xs*))
 (*msort* (*drop* (*n div* 2) *xs*)))

Number of comparisons

$c\_merge :: 'a\ list \Rightarrow 'a\ list \Rightarrow nat$
$c\_msort :: 'a\ list \Rightarrow nat$

Number of comparisons

$c\_merge :: 'a\ list \Rightarrow 'a\ list \Rightarrow nat$
$c\_msort :: 'a\ list \Rightarrow nat$

Lemma

$c\_merge\ xs\ ys$

Number of comparisons

$c\_merge :: 'a\ list \Rightarrow 'a\ list \Rightarrow nat$
$c\_msort :: 'a\ list \Rightarrow nat$

Lemma

$c\_merge\ xs\ ys \le length\ xs + length\ ys$

Number of comparisons

$c\_merge :: 'a\ list \Rightarrow 'a\ list \Rightarrow nat$
$c\_msort :: 'a\ list \Rightarrow nat$

Lemma

$c\_merge\ xs\ ys \le length\ xs + length\ ys$

Theorem

$length\ xs = 2^k \Longrightarrow c\_msort\ xs \le k * 2^k$

$msort\_bu :: 'a\ list \Rightarrow 'a\ list$

$msort\_bu\ xs =$
$(\text{if } xs = [] \text{ then } [] \text{ else } merge\_all\ (map\ (\lambda x.\ [x])\ xs))$

$msort\_bu :: 'a\ list \Rightarrow 'a\ list$

$msort\_bu\ xs =$
$(\text{if } xs = [] \text{ then } [] \text{ else } merge\_all\ (map\ (\lambda x.\ [x])\ xs))$

$merge\_all :: 'a\ list\ list \Rightarrow 'a\ list$

$merge\_all\ [] = undefined$
$merge\_all\ [xs] = xs$

$msort\_bu :: 'a\ list \Rightarrow 'a\ list$

$msort\_bu\ xs =$
$(\text{if } xs = [] \text{ then } [] \text{ else } merge\_all\ (map\ (\lambda x.\ [x])\ xs))$

$merge\_all :: 'a\ list\ list \Rightarrow 'a\ list$

$merge\_all\ [] = undefined$
$merge\_all\ [xs] = xs$
$merge\_all\ xss = merge\_all\ (merge\_adj\ xss)$

$msort\_bu :: 'a\ list \Rightarrow 'a\ list$

$msort\_bu\ xs =$
$(\text{if } xs = [] \text{ then } [] \text{ else } merge\_all\ (map\ (\lambda x.\ [x])\ xs))$

$merge\_all :: 'a\ list\ list \Rightarrow 'a\ list$

$merge\_all\ [] = undefined$
$merge\_all\ [xs] = xs$
$merge\_all\ xss = merge\_all\ (merge\_adj\ xss)$

$merge\_adj :: 'a\ list\ list \Rightarrow 'a\ list\ list$

$merge\_adj\ [] = []$
$merge\_adj\ [xs] = [xs]$
$merge\_adj\ (xs\ \#\ ys\ \#\ zss) =$
$merge\ xs\ ys\ \#\ merge\_adj\ zss$

Number of comparisons

$c\_merge\_adj :: 'a\ list\ list \Rightarrow nat$
$c\_merge\_all :: 'a\ list\ list \Rightarrow nat$
$c\_msort\_bu :: 'a\ list \Rightarrow nat$

Number of comparisons

$c\_merge\_adj :: 'a\ list\ list \Rightarrow nat$
$c\_merge\_all :: 'a\ list\ list \Rightarrow nat$
$c\_msort\_bu :: 'a\ list \Rightarrow nat$

Theorem
$length\ xs = 2^k \Longrightarrow c\_msort\_bu\ xs \le k * 2^k$

Number of comparisons

$c\_merge\_adj :: 'a\ list\ list \Rightarrow nat$
$c\_merge\_all :: 'a\ list\ list \Rightarrow nat$
$c\_msort\_bu :: 'a\ list \Rightarrow nat$

Theorem
$length\ xs = 2^k \Longrightarrow c\_msort\_bu\ xs \le k * 2^k$

Number of comparisons

$c\_merge\_adj :: 'a\ list\ list \Rightarrow nat$
$c\_merge\_all :: 'a\ list\ list \Rightarrow nat$
$c\_msort\_bu :: 'a\ list \Rightarrow nat$

Even better

Even better

Make use of already sorted subsequences

Example

Sorting	$[7, 3, 1, 2, 5]$:
do not start with	$[[7], [3], [1], [2], [5]]$
but with	$[[1, 3, 7], [2, 5]]$