Principle: Count function calls

For every function \( f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \tau \)
define a *timing function* \( t.f :: \tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow \text{nat} : \)

Translation of defining equations:

\[
\begin{align*}
  e & \leadsto e' \\
  f \, p_1 \ldots p_n = e & \leadsto t.f \, p_1 \ldots p_n = e' + 1
\end{align*}
\]
**Principle: Count function calls**

For every function \( f : \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau \)

define a **timing function** \( t.f : \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \text{nat} \):

Translation of defining equations:

\[
\begin{align*}
e & \rightsquigarrow e' \\
fp_1 \ldots p_n = e & \rightsquigarrow t.fp_1 \ldots p_n = e' + 1
\end{align*}
\]

Translation of expressions:

\[
\begin{align*}
s_1 & \rightsquigarrow t_1 \ldots s_k \rightsquigarrow t_k \\
gs_1 \ldots s_k & \rightsquigarrow t_1 + \cdots + t_k + t.gs_1 \ldots s_k
\end{align*}
\]

- Variable \( \rightsquigarrow 0 \), Constant \( \rightsquigarrow 0 \)
- Constructor calls and primitive operations on \( \text{bool} \) and numbers cost 1

---

**Example**

\[
\begin{align*}
\text{app} \ [ \ ] \ ys & = ys
\end{align*}
\]
Example

\[ \text{app} [] \ ys = ys \]
\[ \leadsto \]
\[ t_{\text{app}} [] \ ys = 0 + 1 \]

Example

\[ \text{app} [] \ ys = ys \]
\[ \leadsto \]
\[ t_{\text{app}} [] \ ys = 0 + 1 \]
\[ \text{app} (x\#xs) \ ys = x \# \ \text{app} \ xs \ ys \]

Example

\[ \text{app} [] \ ys = ys \]
\[ \leadsto \]
\[ t_{\text{app}} [] \ ys = 0 + 1 \]
\[ \text{app} (x\#xs) \ ys = x \# \ \text{app} \ xs \ ys \]
\[ \leadsto \]
\[ t_{\text{app}} (x\#xs) \ ys = 0 + (0 + 0 + t_{\text{app}} \ xs \ ys) + 1 + 1 \]

A compact formulation of

\[ e \leadsto t \]

\[ t \] is the sum of all \( t_{g_{s_1}} \ldots \ s_k \)
such that \( g_{s_1} \ldots s_n \) is a subterm of \( e \)
A compact formulation of
\[ e \leadsto t \]

\( t \) is the sum of all \( t \cdot g \; s_1 \ldots s_k \)
such that \( g \; s_1 \ldots s_n \) is a subterm of \( e \)

If \( g \) is
  - a constructor or
  - a predefined function on \( \text{bool} \) or numbers
then \( t \cdot g \; s_1 \ldots s_n = 1 \).

if and case

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated \textit{lazily}.

if and case

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated \textit{lazily}.

Translation:

\[
\frac{b \leadsto t \quad s_1 \leadsto t_1 \quad s_2 \leadsto t_2}{\text{if } b \text{ then } s_1 \text{ else } s_2 \leadsto t + (\text{if } b \text{ then } t_1 \text{ else } t_2)}
\]
A compact formulation of

e \rightsquigarrow t

t is the sum of all \( t \cdot g \ s_1 \ldots \ s_k \)
such that \( g \ s_1 \ldots \ s_n \) is a subterm of \( e \)

If \( g \) is
- a constructor or
- a predefined function on \( \text{bool} \) or numbers

then \( t \cdot g \ldots = 1 \).

if and case

So far we model a call-by-value semantics

Conditionals and case expressions are evaluated **lazily**.

Translation:

\[
\frac{\begin{array}{l}
    b \rightsquigarrow t \\
    s_1 \rightsquigarrow t_1 \\
    s_2 \rightsquigarrow t_2
\end{array}}{\text{if } b \text{ then } s_1 \text{ else } s_2 \rightsquigarrow t + (\text{if } b \text{ then } t_1 \text{ else } t_2)}
\]

\( O(\cdot) \) is enough

\( O(\cdot) \) is enough

\( \implies \) Reduce all additive constants to 1

**Example**

\( t \text{ app } (x\#xs) \ ys = t \text{ app } xs \ ys + 1 \)
• 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.

• 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.

Thys/Sorting.thy

Insertion sort complexity
merge :: 'a list ⇒ 'a list ⇒ 'a list

merge [] ys = ys
merge xs [] = xs
merge (x # xs) (y # ys) = 
    (if x ≤ 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 ≤ 1 then xs 
    else merge (msort (take (n div 2) xs)) 
        (msort (drop (n div 2) xs)))

Thys/Sorting.thy

Merge sort
Chapter 7
Binary Trees

HOL/Library/Tree.thy
Thys/Tree_Additions.thy

Binary trees

datatype 'a tree = Leaf | Node ('a tree) 'a ('a tree)

Tree traversal

inorder :: 'a tree ⇒ 'a list
inorder () = []
inorder (l, x, r) = inorder l @ [x] @ inorder r

preorder :: 'a tree ⇒ 'a list
preorder () = []
preorder (l, x, r) = x # preorder l @ preorder r

postorder :: 'a tree ⇒ 'a list
postorder () = []
postorder (l, x, r) = postorder l @ postorder r @ [x]
size :: 'a tree ⇒ nat

|∅| = 0
|⟨⟨l, _, r⟩⟩| = |l| + |r| + 1

size1 :: 'a tree ⇒ nat

|t|₁ = |t| + 1

⇒

|∅|₁ = 1
|⟨⟨l, x, r⟩⟩|₁ = |t|₁ + |r|₁
\[ \text{height} :: \text{'a tree} \Rightarrow \text{nat} \]
\[ h(\langle \rangle) = 0 \]
\[ h(\langle l, _, r \rangle) = \max(h(l)) \cdot (h(r)) + 1 \]

**Warning:** \( h(.) \) only on slides

---

\[ \text{height} :: \text{'a tree} \Rightarrow \text{nat} \]
\[ h(\langle \rangle) = 0 \]
\[ h(\langle l, _, r \rangle) = \max(h(l)) \cdot (h(r)) + 1 \]

---

\[ \text{height} :: \text{'a tree} \Rightarrow \text{nat} \]
\[ h(\langle \rangle) = 0 \]
\[ h(\langle l, _, r \rangle) = \max(h(l)) \cdot (h(r)) + 1 \]

**Warning:** \( h(.) \) only on slides

---

**Lemma** \( h(t) \leq |t| \)

**Lemma** \( |t|_1 \leq 2^{h(t)} \)
Minimal height

\[ \text{min\_height} :: 'a\ tree \Rightarrow \text{nat} \]
\[ \text{mh}(\emptyset) = 0 \]
\[ \text{mh}(\langle l, -, r \rangle) = \min (\text{mh}(l)) (\text{mh}(r)) + 1 \]

\[ \text{min\_height} :: 'a\ tree \Rightarrow \text{nat} \]
\[ \text{mh}(\emptyset) = 0 \]
\[ \text{mh}(\langle l, -, r \rangle) = \min (\text{mh}(l)) (\text{mh}(r)) + 1 \]

Warning: \text{mh}(.) only on slides

Internal path length

\[ \text{ipl} :: 'a\ tree \Rightarrow \text{nat} \]
\[ \text{ipl}\ \emptyset = 0 \]
\[ \text{ipl}\ \langle l, -, r \rangle = \text{ipl}\ l + |l| + \text{ipl}\ r + |r| \]

\[ \text{ipl} :: 'a\ tree \Rightarrow \text{nat} \]
\[ \text{ipl}\ \emptyset = 0 \]
\[ \text{ipl}\ \langle l, -, r \rangle = \text{ipl}\ l + |l| + \text{ipl}\ r + |r| \]

Why relevant?
Complete tree

\[ \text{complete} :: 'a \text{ tree} \Rightarrow \text{ bool} \]

\[ \text{complete} \langle \rangle = \text{True} \]
\[ \text{complete} \langle l, \_, r \rangle = \]
\[ (\text{complete} \ l \land \text{complete} \ r \land h(l) = h(r)) \]

\textbf{Lemma} \text{complete} \ t = (mh(t) = h(t))
Complete tree

\[ \text{complete :: } 'a \text{ tree } \Rightarrow \text{ bool} \]
\[ \text{complete } \langle \rangle = \text{True} \]
\[ \text{complete } \langle l, \_, r \rangle = \]
\[ (\text{complete } l \wedge \text{complete } r \wedge h(l) = h(r)) \]

**Lemma** \( \text{complete } t = (mh(t) = h(t)) \)

**Lemma** \( \text{complete } t \implies |t|_1 = 2^{h(t)} \)

Complete tree

\[ \text{complete :: } 'a \text{ tree } \Rightarrow \text{ bool} \]
\[ \text{complete } \langle \rangle = \text{True} \]
\[ \text{complete } \langle l, \_, r \rangle = \]
\[ (\text{complete } l \wedge \text{complete } r \wedge h(l) = h(r)) \]

**Lemma** \( \text{complete } t = (mh(t) = h(t)) \)

**Lemma** \( \text{complete } t \implies |t|_1 = 2^{h(t)} \)

**Lemma** \( |t|_1 = 2^{h(t)} \implies \text{complete } t \)

Corollary \( \neg \text{complete } t \implies |t|_1 < 2^{h(t)} \)
Complete tree

\[ \text{complete} :: \forall t. \text{tree} \Rightarrow \text{bool} \]
\[ \text{complete} \; \langle \rangle = \text{True} \]
\[ \text{complete} \; \langle l, \_ , r \rangle = \]
\[ (\text{complete} \; l \land \text{complete} \; r \land h(l) = h(r)) \]

**Lemma** \( \text{complete} \; t = (\text{mh} \; t = h(t)) \)

**Lemma** \( \text{complete} \; t \implies |t|_1 = 2^{h(t)} \)

**Lemma** \( |t|_1 = 2^{h(t)} \implies \text{complete} \; t \)

**Lemma** \( |t|_1 = 2^{\text{mh}(t)} \implies \text{complete} \; t \)

**Corollary** \( \neg \text{complete} \; t \implies |t|_1 < 2^{h(t)} \)

**Corollary** \( \neg \text{complete} \; t \implies 2^{\text{mh}(t)} < |t|_1 \)

Complete tree: \( ipl \)

**Lemma** A complete tree of height \( h \) has internal path length \( (h - 2) \times 2^h + 2 \).

In a search tree, finding the node labelled \( x \) takes as many steps as the path from the root to \( x \) is long. Thus the average time to find an element that is in the tree is \( ipl \; t / |t| \).

Complete tree: \( ipl \)

**Lemma** A complete tree of height \( h \) has internal path length \( (h - 2) \times 2^h + 2 \).

In a search tree, finding the node labelled \( x \) takes as many steps as the path from the root to \( x \) is long. Thus the average time to find an element that is in the tree is \( ipl \; t / |t| \).

**Lemma** Let \( t \) be a complete search tree of height \( h \).
Complete tree: \( ipl \)

**Lemma** A complete tree of height \( h \) has internal path length \( (h - 2) * 2^h + 2 \).

In a search tree, finding the node labelled \( x \) takes as many steps as the path from the root to \( x \) is long. Thus the average time to find an element that is in the tree is \( ipl \ t / |t| \).

**Lemma** Let \( t \) be a complete search tree of height \( h \). The average time to find a random element that is in the tree is asymptotically \( h - 2 \) (as \( h \) approaches \( \infty \)):

\[
\frac{ipl \ t}{|t|} \sim h - 2
\]

A problem: \( (h - 2) * 2^h + 2 \) is only correct if interpreted over type \( int \), not \( nat \).

Correct version:

**Lemma** \( complete \ t \Rightarrow \)

\[
int (ipl \ t) = (int (h(t)) - 2) * 2^{h(t)} + 2
\]
Balanced tree

\[ \text{balanced} :: 'a \text{ tree} \Rightarrow \text{bool} \]

\[ \text{balanced } t = (h(t) - mh(t) \leq 1) \]

Balanced trees have optimal height:

**Lemma** If \( \text{balanced } t \wedge |t| \leq |t'| \) then \( h(t) \leq h(t') \).

Balanced tree

Balanced tree

Warning

- The terms *complete* and *balanced* are not defined uniquely in the literature.
Chapter 8

Search Trees

Most of the material focuses on

**BSTs** = binary search trees

---

**BSTs represent sets**

Any tree represents a set:

\[
set\_tree :: \text{'a tree} \Rightarrow \text{'a set} \\
set\_tree \emptyset = \emptyset \\
set\_tree \langle l, x, r \rangle = set\_tree l \cup \{x\} \cup set\_tree r
\]

---

A BST represents a set that can be searched in time \(O(h(t))\)
**BSTs represent sets**

Any tree represents a set:

\[ \text{set\_tree} :: 'a \text{ \textit{tree}} \Rightarrow 'a \text{ \textit{set}} \]

\[ \text{set\_tree} \; \emptyset = \emptyset \]

\[ \text{set\_tree} \; \langle l, \; x, \; r \rangle = \text{set\_tree} \; l \cup \{x\} \cup \text{set\_tree} \; r \]

A BST represents a set that can be searched in time \( O(h(t)) \)

Function \( \text{set\_tree} \) is called an \textit{abstraction function}

because it maps the implementation
to the abstract mathematical object

\[ \begin{align*}
\text{bst} & :: 'a \text{ \textit{tree}} \Rightarrow \text{bool} \\
\text{bst} \; \emptyset & = \text{True} \\
\text{bst} \; \langle l, \; a, \; r \rangle & = \\
& (\text{bst} \; l \land \text{bst} \; r \land \\
& (\forall x \in \text{set\_tree} \; l. \; x < a) \land \\
& (\forall x \in \text{set\_tree} \; r. \; a < x))
\end{align*} \]

Type \( 'a \) must be in class \textit{linorder} \( ('a :: \text{linorder}) \) where

\textit{linorder} are \textit{linear orders} (also called \textit{total orders}).

Note: \textit{nat}, \textit{int} and \textit{real} are in class \textit{linorder}
Set interface

An implementation of sets of elements of type 'a must provide
- An implementation type 's
- empty :: 's
- insert :: 'a => 's => 's
- delete :: 'a => 's => 's
- isin :: 's => 'a => bool

Map interface

Instead of a set, a search tree can also implement a map from 'a to 'b:
Map interface

Instead of a set, a search tree can also implement a map from \( a \) to \( b \):

- An implementation type \( m \)
- \( \text{empty} :: m \)
- \( \text{update} :: a \Rightarrow b \Rightarrow m \Rightarrow m \)

Map interface

Instead of a set, a search tree can also implement a map from \( a \) to \( b \):

- An implementation type \( m \)
- \( \text{empty} :: m \)
- \( \text{update} :: a \Rightarrow b \Rightarrow m \Rightarrow m \)
- \( \text{delete} :: a \Rightarrow m \Rightarrow m \)
- \( \text{lookup} :: m \Rightarrow a \Rightarrow b \text{ option} \)

Map interface

Instead of a set, a search tree can also implement a map from \( a \) to \( b \):

- An implementation type \( m \)
- \( \text{empty} :: m \)
- \( \text{update} :: a \Rightarrow b \Rightarrow m \Rightarrow m \)
- \( \text{delete} :: a \Rightarrow m \Rightarrow m \)
- \( \text{lookup} :: m \Rightarrow a \Rightarrow b \text{ option} \)

Sets are a special case of maps

Comparison of elements

We assume that the element type \( a \) is a linear order.

Instead of using \( < \) and \( \leq \) directly:

**datatype** *cmp_val* = \( LT \mid EQ \mid GT \)

\( cmp \ x \ y = \)

(if \( x < y \) then \( LT \) else if \( x = y \) then \( EQ \) else \( GT \))
Implementation

Implementation type: 'a tree

empty = Leaf

insert x '() = ('(), x, '())
insert x 'l, a, r' = (case cmp x a of
  LT ⇒ (insert x l, a, r)
  | EQ ⇒ ('l, a, r)
  | GT ⇒ ('l, a, insert x r))

Implementation

delete x '() = '()
delete x 'l, a, r' =
  (case cmp x a of
   LT ⇒ (delete x l, a, r)
   | EQ ⇒ if r = '() then 'l
     else let (a', r') = del_min r in ('l, a', r')
   | GT ⇒ ('l, a, delete x r'))

del_min 'l, a, r' =
  (if l = '() then (a, r)
   else let (x, l') = del_min l in (x, (l', a, r)))
Why is this implementation correct?

Because  empty  insert  delete  isin
simulate  \{\}  \cup \{.\}  - \{.\}  \in

\text{set\_tree empty} = \{\}

Why is this implementation correct?

Because  empty  insert  delete  isin
simulate  \{\}  \cup \{.\}  - \{.\}  \in

\text{set\_tree empty} = \{\}
\text{set\_tree (insert x t)} = \text{set\_tree t} \cup \{x\}
Why is this implementation correct?

Because \( \text{empty} \quad \text{insert} \quad \text{delete} \quad \text{isin} \)
simulate \( \{ \} \quad \cup \{.\} - \{.\} \in \)

\[
\begin{align*}
\text{set_tree empty} &= \{\} \\
\text{set_tree (insert } x \text{ t)} &= \text{set_tree t} \cup \{x\} \\
\text{set_tree (delete } x \text{ t)} &= \text{set_tree t} - \{x\}
\end{align*}
\]

Why is this implementation correct?

Because \( \text{empty} \quad \text{insert} \quad \text{delete} \quad \text{isin} \)
simulate \( \{\} \quad \cup \{.\} - \{.\} \in \)

\[
\begin{align*}
\text{set_tree empty} &= \{\} \\
\text{set_tree (insert } x \text{ t)} &= \text{set_tree t} \cup \{x\} \\
\text{set_tree (delete } x \text{ t)} &= \text{set_tree t} - \{x\} \\
\text{isin } t \ x &= (x \in \text{set_tree } t)
\end{align*}
\]

Also: \( \text{bst must be invariant} \)

\[
\begin{align*}
\text{bst empty} \\
\text{bst } t &\implies \text{bst (insert } x \text{ t)} \\
\text{bst } t &\implies \text{bst (delete } x \text{ t)}
\end{align*}
\]

Under the assumption \( \text{bst } t \)
Also: \( \text{bst} \) must be invariant

\[
\begin{align*}
\text{bst \ empty} \\
\text{bst } t \implies \text{bst } (\text{insert } x \ t) \\
\text{bst } t \implies \text{bst } (\text{delete } x \ t)
\end{align*}
\]

Why is this implementation correct?

Because \( \text{empty, insert, delete, isin} \)
simulate \( \emptyset \cup \{.\} - \{.\} \in \)

\[
\begin{align*}
\text{set}\_\text{tree } \text{empty} &= \emptyset \\
\text{set}\_\text{tree } (\text{insert } x \ t) &= \text{set}\_\text{tree } t \cup \{x\} \\
\text{set}\_\text{tree } (\text{delete } x \ t) &= \text{set}\_\text{tree } t - \{x\} \\
\text{isin } t &\ x = (x \in \text{set}\_\text{tree } t)
\end{align*}
\]

Under the assumption