Optimal Binary Search Trees

Tobias Nipkow and Dániel Somogyi
Technical University Munich

August 16, 2018

Abstract

This article formalizes recursive algorithms for the construction of optimal binary search trees given fixed access frequencies. We follow Knuth [1], Yao [4] and Mehlhorn [2].

The algorithms are memoized with the help of an AFP entry for memoization [3], thus yielding dynamic programming algorithms.

Contents

1 Introduction
  1.1 Data Representation

2 Weighted Path Length of BST

3 Optimal BSTs: The ‘Cubic’ Algorithm
  3.1 Function \texttt{argmin}
  3.2 The ‘Cubic’ Algorithm
    3.2.1 Functions \texttt{wpl} and \texttt{min_wpl}
    3.2.2 Function \texttt{opt_bst}
    3.2.3 Function \texttt{opt_bst_wpl}

4 Quadrangle Inequality

5 Optimal BSTs: The ‘Quadratic’ Algorithm

6 Code Generation (unmemoized)

7 Memoization

1 Introduction

These theories formalize algorithms for the construction of optimal binary search trees from fixed access frequencies for a fixed list of items. The work
is based on the original article by Knuth [1] and the textbook by Mehlhorn [2, Part III, Chapter 4].

Initially the algorithms are expressed as naive recursive functions and have exponential complexity. Nevertheless we already refer to them as the cubic (Section 3) and the quadratic algorithm (Section 5), their running times of their fully memoized dynamic programming versions. In Section 7 the algorithms are memoized with the help of an existing framework [3].

1.1 Data Representation

Instead of labeling our BSTs with (ascending) keys \( x_i < \ldots < x_j \) we label them with the indices of the actual keys, some interval of integers. Functions taking two integer arguments \( i \) and \( j \) construct or analyze trees such that \( \text{inorder } t = [i..j] \).

The access frequencies are given by two tables (functions) \( a \) and \( b \):

\[
\begin{align*}
\text{ak} & (i \leq k \leq j + 1) \text{ is the frequency of (failing) searches with a key in the interval } (x_{k-1}, x_k). \\
\text{bk} & (i \leq k \leq j) \text{ is the frequency of (successful) searches with key } x_k.
\end{align*}
\]

2 Weighted Path Length of BST

theory Weighted_Path_Length
imports HOL-Library.Tree
begin

This theory presents two definitions of the weighted path length of a BST, the objective function we want to minimize, and proves them equivalent. Function \( Wpl \) is the intuitive global definition that sums \( a \) over all leaves and \( b \) over all nodes, taking their depth (= number of comparisons to reach that point) into account. Function \( wpl \) is a recursive definition and thus suitable for the later dynamic programming approaches to building a BST with the minimal weighted path length.

lemma inorder_upto_split:
  assumes inorder \( \langle l,k,r \rangle = [i..j] \)
  shows inorder \( l = [i..k-1] \) inorder \( r = [k+1..j] \) \( i \leq k \leq j \)
⟨proof⟩

fun incr2 :: \( \text{int} \times \text{nat} \Rightarrow \text{int} \times \text{nat} \) where
incr2 \((x,n)\) = \((x, n + 1)\)

fun leaves :: \( \text{int} \Rightarrow \text{int tree} \Rightarrow (\text{int} \times \text{nat}) \text{ set} \) where
leaves \( i \) Leaf = \\{ \((i,0)\) \}
leaves \( i \) (Node \( l \) \( k \) \( r \)) = incr2 \((\text{leaves } l \cup \text{leaves } (k+1) \) \( r \))

fun nodes :: \( \text{int tree} \Rightarrow (\text{int} \times \text{nat}) \text{ set} \) where

nodes Leaf = {} |
nodes (Node l k r) = {(k,1)} ∪ incr2 ' (nodes l ∪ nodes r)

lemma finite_nodes: finite (nodes t)  ⟨proof⟩

lemma finite_leaves: finite (leaves i t)  ⟨proof⟩

lemmanotin_nodes0: (k, 0) ∉ nodes t  ⟨proof⟩

lemma sum_incr2: sum λ k. f (incr2 k) = sum λ k. f (fst (snd (k+1)))  ⟨proof⟩

lemma fst_nodes: fst ' nodes t = set_tree t  ⟨proof⟩

lemma fst_leaves: \( \mbox{inorder t} = [i..j] ; i \leq j+1 \) ⇒ fst ' leaves i t = \{i..j+1\}  ⟨proof⟩

lemma sum_leaves: \( \mbox{inorder t} = [i..j] ; i \leq j+1 \) ⇒  \( \sum \mbox{leaves i t} . f (\mbox{fst x} :: \mbox{nat}) \) = sum f \{i..j+1\}  ⟨proof⟩

lemma sum_nodes: inorder t = [i..j] ⇒  \( \sum \mbox{nodes t} . f (\mbox{fst xy} :: \mbox{nat}) \) = sum f \{i..j\}  ⟨proof⟩

locale wpl =
fixes w :: int ⇒ int ⇒ nat
begin

fun wpl :: int ⇒ int ⇒ int tree ⇒ nat where
wpl i j Leaf = 0 |
wpl i j (Node l k r) = wpl i (k-1) l + wpl (k+1) j r + w i j

end

locale Wpl =
fixes a b :: int ⇒ nat
begin

definition Wpl :: int ⇒ int ⇒ int tree ⇒ nat where
Wpl i t = sum λ k. c * b k (nodes t) + sum λ k. c * a k (leaves i t)

definition w :: int ⇒ int ⇒ nat where
w i j = sum a \{i..j+1\} + sum b \{i..j\}
sublocale wpl where w = w ⟨proof⟩

lemma inorder t = [i..j] ⇒ wpl i j t = Wpl i t ⟨proof⟩
end
end

3 Optimal BSTs: The ‘Cubic’ Algorithm

theory Optimal_BST
imports Weighted_Path_Length
begin

lemma Min_add_const:
fixes f :: _ ⇒ (\_::ordered_ab_semigroup_add)
shows [ finite S; S ≠ {} ] ⇒ Min ((λx. f x + c) ' S) = Min(f ' S) + c
⟨proof⟩

3.1 Function argmin

Function argmin iterates over a list and returns the rightmost element that minimizes a given function:

fun argmin :: ('a ⇒ ('b::linorder)) ⇒ 'a list ⇒ 'a where
argmin f (x#xs) =
(if xs = [] then x else
let m = argmin f xs in if f x < f m then x else m)

An optimized version that avoids repeated computation of f x:

fun argmin2 :: ('a ⇒ ('b::linorder)) ⇒ 'a list ⇒ 'a * 'b where
argmin2 f (x#xs) =
(let fx = f x
in if xs = [] then (x, fx)
else let mfm = argmin2 f xs
in if fx < snd mfm then (x,fx) else mfm)

lemma argmin2_argmin: xs ≠ [] ⇒ argmin2 f xs = (argmin f xs, f(argmin f xs))
⟨proof⟩

lemma argmin_argmin2[code]: argmin f xs = (if xs = [] then undefined else fst(argmin2 f xs))
⟨proof⟩

lemma argmin_forall: xs ≠ [] ⇒ (∀x. x∈set xs ⇒ P x) ⇒ P (argmin f xs)
⟨proof⟩
lemma argmin_in: \(\mathbf{xs} \neq \emptyset \implies \text{argmin } f \mathbf{xs} \in \text{set } \mathbf{xs}\) 
(proof)

lemma argmin_MIN: \(\mathbf{xs} \neq \emptyset \implies f \left(\text{argmin } f \mathbf{xs}\right) = \text{Min } (f \mapsto \text{set } \mathbf{xs})\)
(proof)

lemma argmin_pairs: \(\mathbf{xs} \neq \emptyset \implies (\text{argmin } f \mathbf{xs}, f (\text{argmin } f \mathbf{xs})) = \text{argmin } \text{snd } \left(\text{map } (\lambda x. (x,f x)) \mathbf{xs}\right)\)
(proof)

lemma argmin_map: \(\mathbf{xs} \neq \emptyset \implies \text{argmin } c \left(\text{map } f \mathbf{xs}\right) = f \left(\text{argmin } (c \circ f) \mathbf{xs}\right)\)
(proof)

3.2 The ‘Cubic’ Algorithm

We hide the details of the access frequencies \(a\) and \(b\) by working with an abstract version of function \(w\) defined above (summing \(a\) and \(b\)). Later we interpret \(w\) accordingly.

locale Optimal_BST =
fixes \(w::\text{int} \Rightarrow \text{int} \Rightarrow \text{nat}\)
begin

3.2.1 Functions \(\text{wpl}\) and \(\text{min_wpl}\)

sublocale \(\text{wpl}\) where \(w = w\) (proof)

Function \(\text{min_wpl } i\ j\) computes the minimal weighted path length of any tree \(t\) where \(\text{inorder } t = [i..j]\). It simply tries all possible indices between \(i\) and \(j\) as the root. Thus it implicitly constructs all possible trees.

declare conj_cong [fundef_cong]
function \(\text{min_wpl}::\text{int} \Rightarrow \text{int} \Rightarrow \text{nat}\) where
\(\text{min_wpl } i\ j =\)
(if \(i > j\) then 0
else \(\text{Min } \left((\lambda k. \text{min_wpl } i\ (k-1) + \text{min_wpl } (k+1)\ j) \cdot \{i..j\}\right) + w\ i\ j\))
(proof)
termination (proof)
declare \(\text{min_wpl}.\text{simps[simp del]}\)

Note that for efficiency reasons we have pulled \(+ w\ i\ j\) out of \(\text{Min}\). In the lemma below this is reversed because it simplifies the proofs. Similar optimizations are possible in other functions below.

lemma \(\text{min_wpl}.\text{simps[simp]}\):
\(i > j \implies \text{min_wpl } i\ j = 0\)
\(i \leq j \implies \text{min_wpl } i\ j =\)
\(\text{Min } \left((\lambda k. \text{min_wpl } i\ (k-1) + \text{min_wpl } (k+1)\ j + w\ i\ j) \cdot \{i..j\}\right)\)
(proof)

5
lemma upto_split:
\[
\begin{array}{c}
i \leq j; \quad j \leq k \quad \Rightarrow \quad [i..k] = [i..j-1] \oplus [j..k]
\end{array}
\]
(proof)

Function min_wpl returns a lower bound for all possible BSTs:

theorem min_wpl_is_optimal:
inorder t = [i..j] \Rightarrow \text{min_wpl } i j \leq \text{wpl } i j t
(proof)

Now we show that the lower bound computed by min_wpl is the wpl of an optimal tree that can be computed in the same manner.

3.2.2 Function opt_bst

This is the functional equivalent of the standard cubic imperative algorithm. Unless it is memoized, the complexity is again exponential. The pattern of recursion is the same as for min_wpl but instead of the minimal weight it computes a tree with the minimal weight:

function opt_bst :: int \Rightarrow int \Rightarrow int \arrow \text{tree} where
opt_bst i j =
\quad (\text{if } i > j \text{ then } \text{Leaf}
\quad \text{else } \text{argmin } \text{wpl } i j\ (\langle \text{opt_bst } i (k-1), k, \text{opt_bst } (k+1) j \rangle, k \leftarrow [i..j]\))
(proof)
termination (proof)
declare opt_bst_simps[simp del]
corollary opt_bst_simps[simp]:
i > j \Rightarrow \text{opt_bst } i j = \text{Leaf}
i \leq j \Rightarrow \text{opt_bst } i j =
\quad (\text{argmin } \text{wpl } i j\ (\langle \text{opt_bst } i (k-1), k, \text{opt_bst } (k+1) j \rangle, k \leftarrow [i..j]\))
(proof)

As promised, opt_bst computes a tree with the minimal wpl:

theorem wpl_opt_bst: wpl i j \text{ (opt_bst } i j) = \text{min_wpl } i j
(proof)
corollary opt_bst_is_optimal:
inorder t = [i..j] \Rightarrow \text{wpl } i j \text{ (opt_bst } i j) \leq \text{wpl } i j t
(proof)

3.2.3 Function opt_bst_wpl

Function opt_bst is simplistic because it computes the wpl of each tree anew rather than returning it with the tree. That is what opt_bst_wpl does:

function opt_bst_wpl :: int \Rightarrow int \Rightarrow int \arrow \text{tree} \times \text{nat} where
opt_bst_wpl i j =
\quad (\text{if } i > j \text{ then } \text{Leaf}, 0)
\quad \text{else } \text{argmin } \text{snd}\ \text{let } \text{(tl,cl)} = \text{opt_bst_wpl } i (k-1);
\[(t_2, c_2) = \text{opt bst wpl} (k+1) j \]
\[\text{in } ((t_1, k, t_2), c_1 + c_2 + w i j). k \leftarrow [i..j])\]

\langle proof \rangle
termination
\langle proof \rangle
declare opt bst wpl.simps[simp del]

Function \text{opt bst wpl} returns an optimal tree and its wpl:

\text{lemma opt bst wpl eq pair:}
\text{opt bst wpl } i \, j = (\text{opt bst } i \, j, \text{wpl } i \, j \, (\text{opt bst } i \, j))
\langle proof \rangle

\text{corollary opt bst wpl eq pair':}
\text{opt bst wpl } i \, j = (\text{opt bst } i \, j, \text{min wpl } i \, j)
\langle proof \rangle

end
end

4 Quadrangle Inequality

\text{theory Quadrilateral Inequality}
\text{imports Main}
\text{begin}

definition is arg min on :: (a ⇒ ('b::linorder)) ⇒ 'a set ⇒ 'a ⇒ bool where
is arg min on \ f \ S \ x = (x \in \ S \land (\forall \ y \in \ S. \ f \ x \leq \ f \ y))

definition Args min on :: (int ⇒ ('b::linorder)) ⇒ int set ⇒ int set where
Args min on \ f \ I = \{k. \ is arg min on \ f \ I \ k\}

\text{lemmas Args min simps = Args min on_def is arg min on_def}

\text{lemma is arg min on antimono: fixes f :: _ ⇒ _::order}
shows [ is arg min on \ f \ S \ x; \ f \ y \leq \ f \ x; \ y \in \ S ] ⇒ is arg min on \ f \ S \ y
\langle proof \rangle

\text{lemma ex is arg min on if finite: fixes f :: 'a ⇒ 'b :: linorder}
shows [ finite \ S; \ S \neq \{\} ] ⇒ \exists \ x. \ is arg min on \ f \ S \ x
\langle proof \rangle

locale QI =
\text{fixes } c k :: int ⇒ int ⇒ int ⇒ nat
\text{fixes } c :: int ⇒ int ⇒ nat
\text{and } w :: int ⇒ int ⇒ nat
\text{assumes QI w: } [i \leq i'; i' < j; j \leq j'] \Rightarrow
w i j + w i' j' \leq w i' j + w i j'

7
assumes monotone \( w \): \[ i \leq i' \land i' < j \land j \leq j' \implies w(i') \leq w(i+j) \]
assumes \( c \): \[ c < j \implies c(i) = \text{Min}((c(k) + (i+1,j)) \]
assumes \( c_k \): \[ i < j \land k \in \{i+1..j\} \implies c(k) = w(i + j) + c(k-1) + c(k)
begin

abbreviation mins i j \equiv \text{Args\_min\_on}(c_k(i)) \{i+1..j\}
definition K i j \equiv (i = j \text{ then } i \text{ else } \text{Max}(mins i j))

lemma c\_def\_rec:
i < j \implies c(i) = \text{Min}((\lambda k. (c(k-1) + c(k) + w(i+j)) \{i+1..j\}))
⟨proof⟩

lemma mins\_subset:
mins i j \subseteq \{i+1..j\}
⟨proof⟩

lemma mins\_nonempty:
i < j \implies mins i j \neq \{
⟨proof⟩

lemma finite\_mins:
finit(e(mins i j))
⟨proof⟩

lemma is\_arg\_min\_on\_Min:
assumes finite A is\_arg\_min\_on f A a shows Min(f A) = f a
⟨proof⟩

lemma c\_k\_with\_K:
i < j \implies c(i) = c_k(i) (K i j)
⟨proof⟩

lemma K\_subset:
assumes i \leq j shows K i j \in \{i..j\}
⟨proof⟩

lemma lemma\_2:
\[ l = \text{nat}(j'-i); i \leq i'; i' \leq j; j \leq j' \implies c(i) + c(i') \leq c(i+j) + c(i')
⟨proof⟩

corollary QI':
assumes i < k \land k \leq k' \land k' \leq j \implies c_k(i) \leq c_k(i) k'
shows c_k(i)(j+1) \leq c_k(i) k'
⟨proof⟩

corollary QI'':
assumes i+1 < k \land k \leq k' \land k' \leq j+1 \implies c_k(i) \leq c_k(i) k'
shows c_k(i+1)(j+1) \leq c_k(i+1)(j+1) k
⟨proof⟩

lemma lemma\_3:\nassumes i \leq j shows K i j \leq K i (j+1)
⟨proof⟩

8
lemma lemma_3.2: assumes $i \leq j$ shows $K_i (j+1) \leq K (i+1) (j+1)$  
(proof)

lemma lemma_3: assumes $i \leq j$ shows $K_i j \leq K_i (j+1) K_i (j+1) \leq K (i+1) (j+1)$  
(proof)

end

end

5 Optimal BSTs: The ‘Quadratic’ Algorithm

theory Optimal_BST2
imports Optimal_BST Quadrilateral_Inequality
begin

Knuth presented an optimization of the previously known cubic dynamic programming algorithm to a quadratic one. A simplified proof of this optimization was found by Yao [4]. Mehlhorn follows Yao closely. The core of the optimization argument is given abstractly in theory Optimal_BST.Quadrilateral_Inequality. In addition we first need to establish some more properties of argmin.

An index-based specification of argmin expressing that the last minimal list-element is picked:

lemma argmin_takes_last: $xs \neq [] \implies argmin f xs = xs \setminus \text{Max \{i. i < length xs \wedge (\forall x \in set xs. f(xs!i) \leq f x)\}}$
\[
\text{(is } \_ \implies \_ = \_ \text{! Max (?M xs))}
\]
(proof)

lemma Min_ex: $\text{finite } F; F \neq \{\} \implies \exists m \in F. \forall n \in F. m \leq (\cdots ::\text{linorder})$
(proof)

A consequence of argmin_takes_last:

lemma argmin_Max_Args_min_on: assumes [arith]: $i \leq j$
shows $\text{argmin } f [i..j] = \text{Max (Args_min_on } f \{i..j\})$
(proof)

As a consequence of argmin_Max_Args_min_on the following lemma allows us to justify the restriction of the index range of argmin used below in the optimized (quadratic) algorithm.

lemma argmin_red_int:
assumes $i \leq i' argmin f [i..j] \in \{i',j\'}$ $j' \leq j$
shows $argmin f [i',j'] = argmin f [i,j]$
(proof)
fun root :: 'a tree ⇒ 'a where
root ⟨_, r, _⟩ = r

Now we can formulate and verify the improved algorithm. This requires
two assumptions on the weight function w.

locale Optimal_BST2 = Optimal_BST +
assumes monotone_w: [i ≤ i'; i' ≤ j; j ≤ j'] ⇒ w i' j ≤ w i j'
assumes QL_w: [i ≤ i'; i' ≤ j; j ≤ j'] ⇒ w i j + w i' j' ≤ w i' j + w i j'
begin

When finding an optimal tree for [i..j] the optimization consists in reduc-
ing the search for the root from [i..j] to [root (opt_bst2 i (j - (1::'b))) ..
root (opt_bst2 (i + (1::'a))) j]]:

function opt_bst2 :: int ⇒ int ⇒ int tree where
opt_bst2 i j = (if i > j then Leaf else
if i = j then Node Leaf i Leaf else
let left = root (opt_bst2 i (j-1)) in
let right = root (opt_bst2 (i+1) j) in
argmin (wpl i j) [(opt_bst2 i (k-1), k, opt_bst2 (k+1) j), k ← [left..right]])
⟨proof⟩

The termination of opt_bst2 is not completely obvious. We first need to
establish some functional properties of the terminating computations. We
start by showing that the root of the returned tree is always between left
and right. This is essentially equivalent to proving that left ≤ right because
otherwise argmin is applied to [], which is undefined.

lemma left_le_right:
opt_bst2_dom(i,j) ⇒
(i=j) → root(opt_bst2 i j) = i ∧
(i<j) → root(opt_bst2 i j) ∈ {root(opt_bst2 i (j-1)) .. root(opt_bst2 (i+1) j)}
⟨proof⟩

Now we can bound the result of opt_bst2 easily:

lemma root_opt_bst2_bound:
opt_bst2_dom(i,j) ⇒ i ≤ j ⇒ root (opt_bst2 i j) ∈ {i..j}
⟨proof⟩

Now termination follows easily:

lemma opt_bst2_dom: ∀ args. opt_bst2_dom args
⟨proof⟩

termination ⟨proof⟩

declare opt_bst2.simps[simp del]

abbreviation min_wpl3 i j k ≡ min_wpl i (k-1) + min_wpl (k+1) j + w i j

10
The correctness proof \cite{?} is based on a general theory of ‘quatrilateral inequalities’ developed in locale QI that we now instantiate:

**interpretation QI**

where

\[
c = \lambda i j. \min_{wpl} (i+1) j
\]
\[
c, k = \lambda i j. \min_{wpl3} (i+1) j
\]
\[
w = \lambda i j. w (i+1) j
\]

⟨proof⟩

**lemma** \(K_{\text{argmin}}: i < j \implies K i j = \arg\min (\min_{wpl3} (i+1) j) \ [i+1..j]\)

⟨proof⟩

**theorem** \(\text{opt}_b\text{st}_2.\text{opt}_b\text{st}_2: \text{opt}_b\text{st}_2 i j = \text{opt}_b\text{st} i j\)

⟨proof⟩

**corollary** \(\text{opt}_b\text{st}_2.\text{is}\_\text{optimal}: wpl i j (\text{opt}_b\text{st}_2 i j) = \min_{wpl} i j\)

⟨proof⟩

**function** \(\text{opt}_b\text{st}_wpl2 :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{nat} \Rightarrow \text{tree} \times \text{nat}\)

\[
\text{opt}_b\text{st}_wpl2 \ i \ j =
\]
\[
(\text{if } i > j \text{ then (Leaf,0) else if } i = j \text{ then (Node Leaf i Leaf, w i i) else let } l = \text{root}(\text{fst}(\text{opt}_b\text{st}_wpl2 i (j-1))));
\]
\[
r = \text{root}(\text{fst}(\text{opt}_b\text{st}_wpl2 (i+1) j)) \text{ in argmin snd}
\]
\[
\left(\text{let} \ (tl, wl) = \text{opt}_b\text{st}_wpl2 i (k-1); (tr, wr) = \text{opt}_b\text{st}_wpl2 (k+1) j
\right.
\]
\[
in ((tl, k, tr), \text{wl} + \text{wr} + w i j) \cdot k \leftarrow [l..r]))
\]

⟨proof⟩

**lemma** \(\text{left}_l\text{e}_r\text{ight}_r\text{ight}_2:\)

\[
\text{opt}_b\text{st}_wpl2.\text{dom}(i,j) \implies (i=j \implies \text{root}(\text{fst}(\text{opt}_b\text{st}_wpl2 i j)) = i) \land
\]
\[
(i<j \implies \text{root}(\text{fst}(\text{opt}_b\text{st}_wpl2 i j)) \in \{\text{root}(\text{fst}(\text{opt}_b\text{st}_wpl2 i (j-1))) \ldots \text{root}(\text{fst}(\text{opt}_b\text{st}_wpl2 (i+1) j))\})
\]

⟨proof⟩

Now we can bound the result of \(\text{opt}_b\text{st}_wpl2\):

**lemma** \(\text{root}_\text{opt}_b\text{st}_wpl2.\text{bound}:\)

\[
\text{opt}_b\text{st}_wpl2.\text{dom} (i,j) \implies i \leq j \implies \text{root (fst(opt}_b\text{st}_wpl2 i j)) \in \{i..j}\}
\]

⟨proof⟩

Now termination follows easily:

**lemma** \(\text{opt}_b\text{st}_wpl2.\text{dom}: \forall \text{args. opt}_b\text{st}_wpl2.\text{dom args}\)

⟨proof⟩

**termination** ⟨proof⟩

**declare** \(\text{opt}_b\text{st}_wpl2.\text{simps}[\text{simp del}]\)
lemma opt bst wpl2_eq_pair:
  opt bst wpl2 i j = (opt bst2 i j, wpl i j (opt bst2 i j))
⟨proof⟩

corollary opt bst wpl2_eq_pair': opt bst wpl2 i j = (opt bst i j, min_wpl i j)
⟨proof⟩
end

theory Optimal_BST_Examples
imports HOL-Library.Tree
begin
  Example by Mehlhorn:
definition a_ex1 :: int ⇒ nat where
    a_ex1 i = [4,0,0,3,10] ! nat i
definition b_ex1 :: int ⇒ nat where
    b_ex1 i = [1,3,3,0] ! nat i
definition t_opt_ex1 :: int tree where
    t_opt_ex1 = ⟨⟨⟨⟩, 0, ⟨⟨⟩, 1, ⟨⟩⟩⟩, 2, ⟨⟨⟩, 3, ⟨⟨⟩, 4, ⟨⟩⟩⟩, 5, ⟨⟨⟩, 6, ⟨⟨⟩, 7, ⟨⟩⟩⟩⟩
  Example by Knuth:
definition a_ex2 :: int ⇒ nat where
    a_ex2 i = 0
definition b_ex2 :: int ⇒ nat where
    b_ex2 i = [32,7,69,13,6,15,10,8,64,142,22,79,18,9] ! nat i
definition t_opt_ex2 :: int tree where
    t_opt_ex2 = { 
      ⟨⟨⟩, 0, ⟨⟨⟩, 1, ⟨⟩⟩⟩, 2, 
      ⟨⟨⟩, 3, ⟨⟨⟩, 4, ⟨⟩⟩⟩, 5, 
      ⟨⟨⟩, 6, ⟨⟨⟩, 7, ⟨⟩⟩⟩ }
}

end
6 Code Generation (unmemoized)

theory Optimal_BST_Code
imports
  Optimal_BST2
  Optimal_BST_Examples
begin

global_interpretation Wpl
where a = a and b = b for a b
defines w_ab = w and wpl_ab = wpl.wpl w_ab ⟨proof⟩

global_interpretation Optimal_BST
where w = w_ab a b rewrites wpl.wpl (w_ab a b) = wpl_ab a b for a b
defines opt_bst_ab = opt_bst ⟨proof⟩

global_interpretation Optimal_BST2
where w = w_ab a b rewrites wpl.wpl (w_ab a b) = wpl_ab a b for a b
defines opt_bst2_ab = opt_bst2 ⟨proof⟩

Examples:
lemma opt_bst_ab a_ex1 b_ex1 0 3 = t_opt_ex1 ⟨proof⟩

lemma opt_bst2_ab a_ex2 b_ex2 0 13 = t_opt_ex2 ⟨proof⟩

end

7 Memoization

theory Optimal_BST_Memo
imports
  Optimal_BST
  Monad_Memo_DP.State_Main
This theory memoizes the recursive algorithms with the help of our
generic memoization framework. Note that currently only the tree build-
ing (function \texttt{Optimal\_BST\_opt\_bst}) is memoized but not the computation
of \(w\).

\textbf{global\_interpretation} \ Wpl\
\textbf{where} \(a = a\) and \(b = b\) for \(a\ b\)
\textbf{defines} \(\texttt{w\_ab} = w\) and \(\texttt{wpl\_ab} = \texttt{wpl\ w\_ab}\) \langle proof \rangle

First we express \texttt{argmin} via \texttt{fold}. Primarily because we have a monadic
version of \texttt{fold} already. At the same time we improve efficiency.

\textbf{lemma} \texttt{fold\_argmin}: \texttt{fold} \(\lambda x\ (m, fm)\). let \(fx = f x\) in if \(fx \leq fm\) then \((x, fx)\) else
\((m, fm)\) \(xs\ (x, f x)\)
\(= (\texttt{argmin} f \ (x\#xs), f(\texttt{argmin} f \ (x\#xs)))\)
\langle proof \rangle

\textbf{lemma} \texttt{argmin\_fold}: \texttt{argmin} \(f\) \(xs\ =\) (case \(xs\) of \([]\Rightarrow \texttt{undefined}\ | \ x\#xs\Rightarrow \texttt{fst}(\texttt{fold} \(\lambda x\ (m, fm)\). let \(fx = f x\) in if \(fx \leq fm\) then \((x, fx)\) else \((m, fm)\)) \(xs\ (x, f x)\))
\langle proof \rangle

The actual memoization of the cubic algorithm:

\textbf{context} \texttt{Optimal\_BST}
\textbf{begin}

\textbf{memoize\_fun} \texttt{opt\_bst} : \texttt{opt\_bst} with memory \texttt{dp\_consistency\_mapping}
\textbf{monadifies} \texttt{(state) opt\_bst.simps\[unfolded argmin\_fold\]}

\textbf{thm} \texttt{opt\_bst\_m\_simps}

\textbf{memoize\_correct}
\langle proof \rangle

\textbf{lemmas} [code] = \texttt{opt\_bst.memoized\_correct}

\textbf{end}

Code generation:

\textbf{global\_interpretation} \texttt{Optimal\_BST}
\textbf{where} \(w = \texttt{w\_ab}\ a\ b\)
\textbf{rewrites} \texttt{wpl\ wpl\ (w\_ab\ a\ b) = wpl\_ab\ a\ b\ for\ a\ b}
\textbf{defines} \texttt{opt\_bst\_ab} = \texttt{opt\_bst} and \texttt{opt\_bst\_ab\’} = \texttt{opt\_bst\_m\’}
\langle proof \rangle

Examples:
\textbf{lemma} opt\_bst\_ab \ a\_ex1 \ b\_ex1 \ 0 \ 3 = t\_opt\_ex1
\langle \text{proof} \rangle

\textbf{lemma} opt\_bst\_ab \ a\_ex2 \ b\_ex2 \ 0 \ 13 = t\_opt\_ex2
\langle \text{proof} \rangle

end

\textbf{References}


