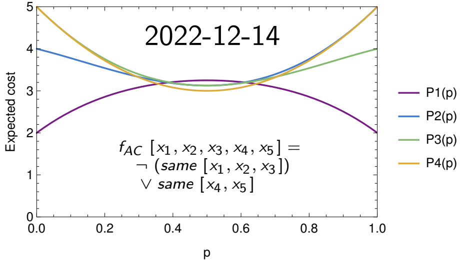
Level-p-complexity of Boolean Functions

Using thinning, memoization, and polynomials

Patrik Jansson[1]

Functional Programming unit, Chalmers University of Technology



[1] Joint work with Julia Jansson, PhD student, Mathematical sciences, Chalmers

Some key results (teaser)

We want to find a decision tree with minimal cost for a boolean function.

We specify the problem as "generate all, select a smallest".

For an example 5-bit function f_{AC} there are already **54192** different decision trees.

$$t_5 = \mathit{size}\ (\mathit{genAlg}\ 5\ f_{AC} :: \mathit{Set}\ \mathit{DecTree})$$

Some key results (teaser)

We want to find a decision tree with minimal cost for a boolean function.

We specify the problem as "generate all, select a smallest".

For an example 5-bit function f_{AC} there are already **54192** different decision trees.

$$t_5 = \mathit{size}\ (\mathit{genAlg}\ 5\ f_{AC} :: \mathit{Set}\ \mathit{DecTree})$$

For a 9-bit function we can estimate the number of trees as

$$t_9 = 9 * t_8^2; \quad t_8 = 8 * t_7^2; \quad t_7 = 7 * t_6^2; \quad t_6 = 6 * t_5^2$$

This estimate gives us $1.29 * 10^{89}$ decision trees!
Clearly, a naive implementation will take far too long to terminate.

Some key results (teaser)

We want to find a decision tree with minimal cost for a boolean function.

We specify the problem as "generate all, select a smallest".

For an example 5-bit function f_{AC} there are already **54192** different decision trees.

$$t_5 = \mathit{size}\ (\mathit{genAlg}\ 5\ f_{AC} :: \mathit{Set}\ \mathit{DecTree})$$

For a 9-bit function we can estimate the number of trees as

$$t_9 = 9 * t_8^2; \quad t_8 = 8 * t_7^2; \quad t_7 = 7 * t_6^2; \quad t_6 = 6 * t_5^2$$

This estimate gives us $1.29 * 10^{89}$ decision trees!
Clearly, a naive implementation will take far too long to terminate.

By using **thinning**, **memoization**, and exact comparisions of **polynomials** we get down to a **singleton set** for iterated majority:

$$\mathit{ps9} = \mathit{genAlgThinMemoPoly}\ 9\ \mathit{maj2} :: \mathit{Set}\ (\mathit{Poly}\ \mathit{Rational})$$
$$\text{-- } \mathit{ps9} = \mathit{fromList}\ [P\ [4, 4, 6, 9, -61, 23, 67, -64, 16]]$$

Some key types and definitions

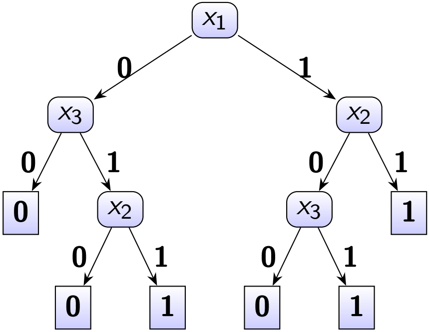
- A Boolean function $f : BoolFun\ n$ takes a tuple of n bits to one bit.
- A tuple $t : Tuple\ n$ is just a vector of n bits (**type** $Bit = Bool$; $\mathbf{0}$ = False; $\mathbf{1}$ = True).
- A decision tree describes a way to evaluate a $BoolFun$:

Haskell version:

type *Index* $= \mathbb{N}$
data *DecTree* **where**
  Res :: *Bool* → *DecTree*
  Pick :: *Index* → *DecTree* → *DecTree* → *DecTree*

ex1 :: *DecTree*
ex1 = *Pick* 1 (*Pick* 3 (*Res* **0**)
                (*Pick* 2 (*Res* **0**) (*Res* **1**)))
         (*Pick* 2 (*Pick* 3 (*Res* **0**) (*Res* **1**))
                (*Res* **1**))

A decision tree (*ex1*) for the 3-bit majority function:



Some key types and definitions

A decision tree describes a way to evaluate a *BoolFun*

Haskell version:

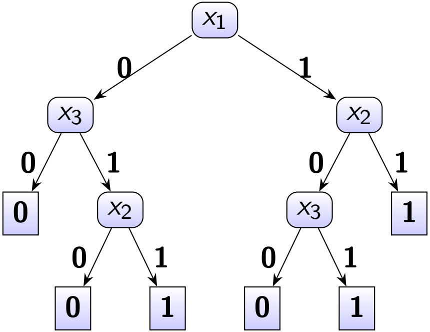
$$
\begin{array}{l}
\textbf{type}\ \mathit{Index} = \mathbb{N} \\
\textbf{data}\ \mathit{DecTree}\ \textbf{where} \\
\quad \mathit{Res}\ \ ::\ \mathit{Bool} \to \mathit{DecTree} \\
\quad \mathit{Pick} ::\ \mathit{Index} \to \mathit{DecTree} \to \mathit{DecTree} \to \mathit{DecTree}
\end{array}
$$

Agda version:

$$
\begin{array}{l}
\textbf{data}\ \mathit{DecTree} : (n : \mathbb{N}) \to (f : \mathit{BoolFun}\ n) \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{Res}\ \ : (b : \mathit{Bool}) \to \mathit{DecTree}\ n\ (\mathit{const}\ b) \\
\quad \mathit{Pick} : \{f : \mathit{BoolFun}\ (\mathit{suc}\ n)\} \to (i : \mathit{Fin}\ (\mathit{suc}\ n)) \to \mathit{DecTree}\ n\ (\mathit{setBit}\ i\ \mathbf{0}\ f) \to \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{DecTree}\ n\ (\mathit{setBit}\ i\ \mathbf{1}\ f) \to \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{DecTree}\ (\mathit{suc}\ n)\ f
\end{array}
$$

$$
\mathit{setBit} : \mathit{Fin}\ (\mathit{suc}\ n) \to \mathit{Bit} \to \mathit{BoolFun}\ (\mathit{suc}\ n) \to \mathit{BoolFun}\ n
$$

Decision trees, cost and complexity

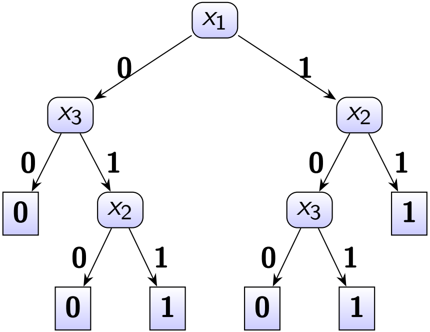


The decision tree *ex1* for the 3-bit majority function *maj*:

- Note that the left subtree is for the function $f_0 = \mathit{setBit}\ 1\ \mathbf{0}\ \mathit{maj} = (\wedge)$
- and the right subtree for the function $f_1 = \mathit{setBit}\ 1\ \mathbf{1}\ \mathit{maj} = (\vee)$

[2] We assume n independent bits, all with probability p to be $\mathbf{1}$. The polynomials are defined over any *Ring*.

Decision trees, cost and complexity



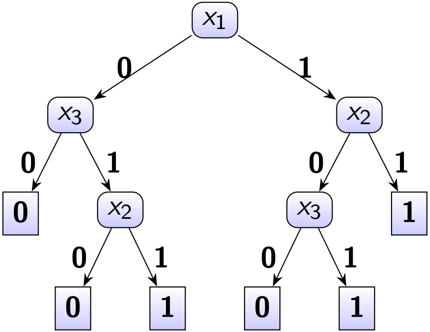
The decision tree *ex1* for the 3-bit majority function *maj*:

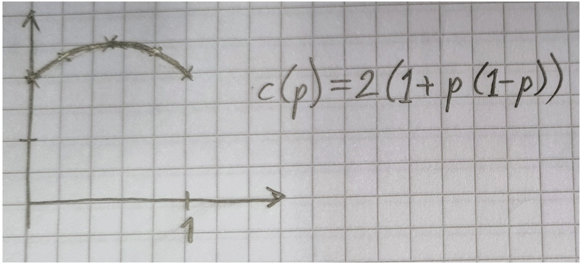
- Note that the left subtree is for the function $f_0 = setBit\ 1\ \mathbf{0}\ maj = (\wedge)$
- and the right subtree for the function $f_1 = setBit\ 1\ \mathbf{1}\ maj = (\vee)$

- The *cost* of a *DecTree* is a function from *Tuple n* to $\mathbb{N}$:
  $cost : \{n : \mathbb{N}\} \to \{f : BoolFun\ n\} \to DecTree\ n\ f \to Tuple\ n \to \mathbb{N}$

[2] We assume n independent bits, all with probability p to be $\mathbf{1}$. The polynomials are defined over any *Ring*.

Decision trees, cost and complexity

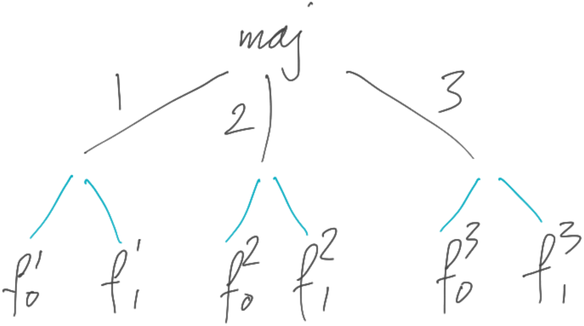




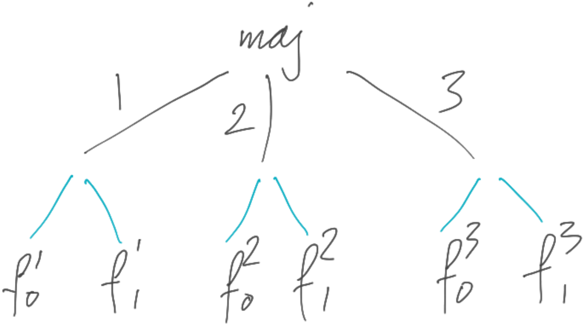
expCost ex1 == $P\ [2, 2, -2]$

- Finally, the *expected cost*[2], is a polynomial in the probability p that a bit is $\mathbf{1}$:
  $expCost : \{n : \mathbb{N}\} \to \{f : BoolFun\ n\} \to DecTree\ n\ f \to Poly\ n$

[2]We assume n independent bits, all with probability p to be $\mathbf{1}$. The polynomials are defined over any *Ring*.



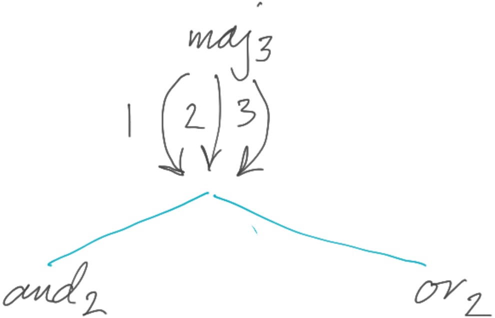
$$f^i_b = \mathrm{setBit}\ i\ b\ \mathrm{maj}$$



$$f_b^i = \text{setBit}\ i\ b\ \text{maj}$$

$$f_0^i = \wedge$$

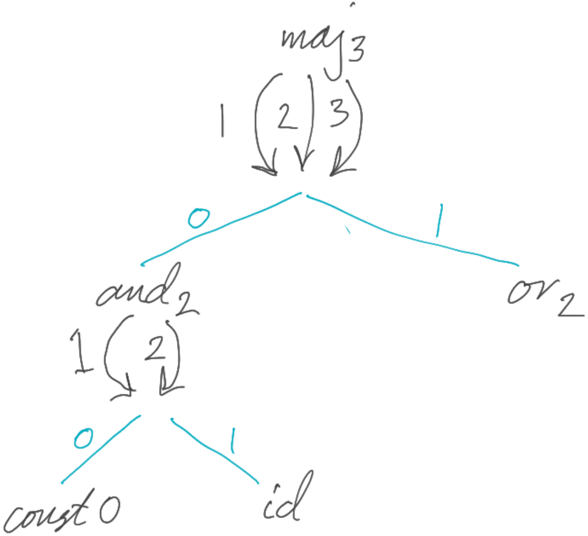
$$f_1^i = \vee$$



$$f^i_b = \mathrm{setBit}\ i\ b\ \mathrm{maj}$$

$$f^i_0 = \wedge = \mathrm{and}_2$$

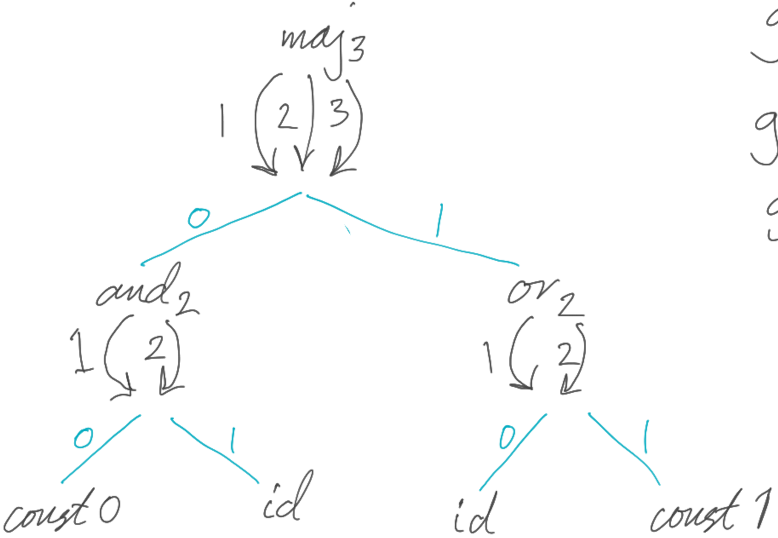
$$f^i_1 = \vee = \mathrm{or}_2$$



$$g^i_b = \mathrm{setBit}\ i\ b\ \mathrm{and}_2$$

$$g^i_0 = \mathrm{const}\ 0$$

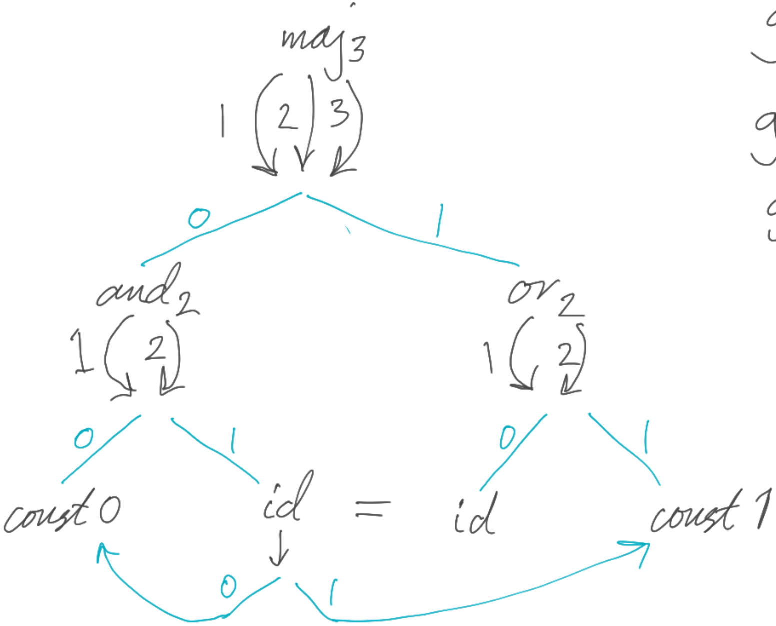
$$g^i_1 = id_{\mathbb{B}}$$



$$g_b^i = \mathit{setBit}\ i\ b\ or_2$$

$$g_0^i = id_{\mathbb{B}}$$

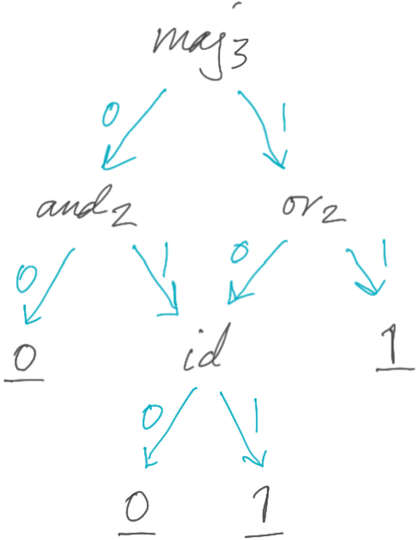
$$g_1^i = \mathit{const}\ 1$$

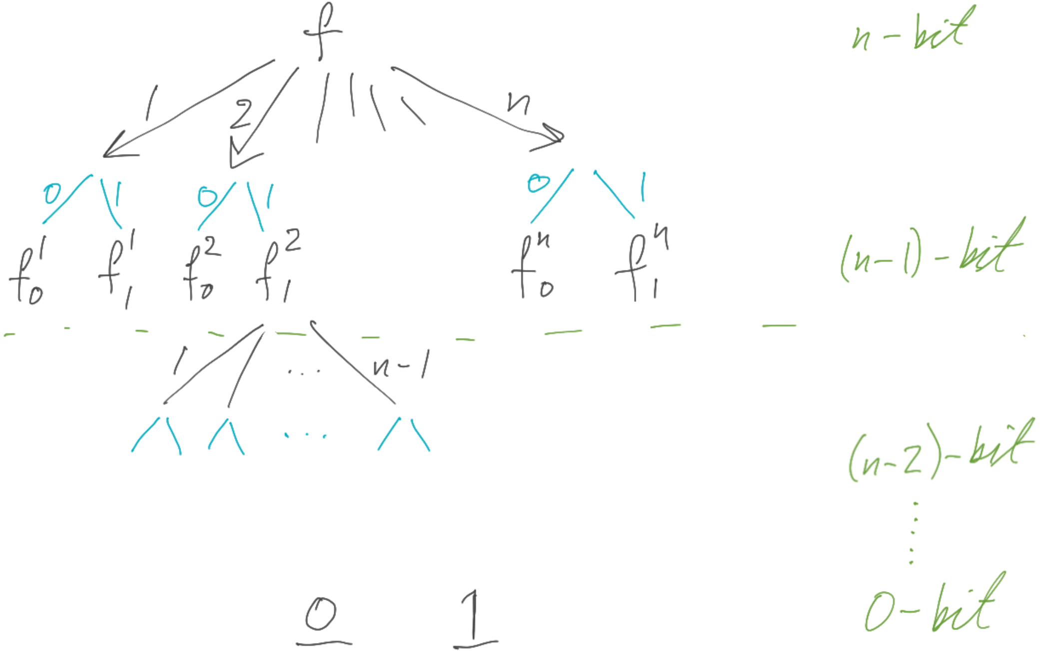


$$g_b^i = \mathit{setBit}\ i\ b\ \mathit{or}_2$$

$$g_0^i = \mathit{id}_{\mathbb{B}}$$

$$g_1^i = \mathit{const}\ 1$$


maj_3
0
1
and_2
or_2
0
1
0
1
0
id
1
0
1
0
1


f
1
2
n
0
1
f_0^1 f_1^1 f_0^2 f_1^2 f_0^n f_1^n
n-1
...
0
1
n-bit
(n-1)-bit
(n-2)-bit
0-bit

A class of Boolean functions

We abstract from the actual type for Boolean functions to a type class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{BoFun}\ \mathit{bf}\ \textbf{where} \\
\quad \mathit{isConst} :: \mathit{bf} \to \mathit{Maybe}\ \mathit{Bool} \\
\quad \mathit{setBit}\ \ :: \mathit{Index} \to \mathit{Bool} \to \mathit{bf} \to \mathit{bf}
\end{array}
$$

We only use one instance: Binary Decision Diagrams (BDDs) which allows good sharing and fast operations.

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$
\begin{array}{l}
\textbf{class } \mathit{TreeAlg}\ a\ \textbf{where} \\
\quad \mathit{res} :: \mathit{Bool} \to a \\
\quad \mathit{pic} :: \mathit{Index} \to a \to a \to a
\end{array}
$$

$$
\begin{array}{l}
\mathit{ex1} :: \mathit{TreeAlg}\ a \Rightarrow a \\
\mathit{ex1} = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))) \\
\qquad\qquad\quad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1}))
\end{array}
$$

$$
\begin{array}{llll}
\textbf{instance} & \mathit{TreeAlg}\ \mathit{DecTree} & \textbf{where}\ \mathit{res} = \mathit{Res}; & \mathit{pic} = \mathit{Pick}; \\
\textbf{instance} & \mathit{TreeAlg}\ \mathit{CostFun} & \textbf{where}\ \mathit{res} = \mathit{resC}; & \mathit{pic} = \mathit{pickC} \\
\textbf{instance}\ \mathit{Ring}\ a \Rightarrow & \mathit{TreeAlg}\ (\mathit{ExpCost}\ a) & \textbf{where}\ \mathit{res} = \mathit{resPoly}; & \mathit{pic} = \mathit{pickPoly}
\end{array}
$$

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$
\begin{array}{l}
\textbf{class}\ \mathit{TreeAlg}\ a\ \textbf{where} \\
\quad \mathit{res} :: \mathit{Bool} \rightarrow a \\
\quad \mathit{pic} :: \mathit{Index} \rightarrow a \rightarrow a \rightarrow a
\end{array}
$$

$$
\begin{array}{l}
\mathit{ex1} :: \mathit{TreeAlg}\ a \Rightarrow a \\
\mathit{ex1} = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))) \\
\qquad\qquad\quad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1}))
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{TreeAlg}\ \mathit{DecTree}\ \textbf{where}\ \mathit{res} = \mathit{Res}; \mathit{pic} = \mathit{Pick}; \\
\textbf{data}\ \mathit{DecTree}\ \textbf{where} \\
\quad \mathit{Res}\ \ :: \mathit{Bool} \rightarrow \mathit{DecTree} \\
\quad \mathit{Pick} :: \mathit{Index} \rightarrow \mathit{DecTree} \rightarrow \mathit{DecTree} \rightarrow \mathit{DecTree}
\end{array}
$$

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$
\begin{array}{l}
\textbf{class}\ \mathit{TreeAlg}\ a\ \textbf{where} \\
\quad \mathit{res} :: \mathit{Bool} \to a \\
\quad \mathit{pic} :: \mathit{Index} \to a \to a \to a
\end{array}
$$

$$
\begin{array}{l}
\mathit{ex1} :: \mathit{TreeAlg}\ a \Rightarrow a \\
\mathit{ex1} = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))) \\
\qquad\qquad\quad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1}))
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{TreeAlg}\ \mathit{CostFun}\ \textbf{where}\ \mathit{res} = \mathit{resC}; \mathit{pic} = \mathit{pickC} \\
\textbf{type}\ \mathit{CostFun} = \mathit{Tuple} \to \mathit{Int} \\
\mathit{resC}\ b = \mathit{const}\ 0 \\
\mathit{pickC}\ i\ c_0\ c_1 = \lambda t \to 1 + \textbf{if}\ \mathit{index}\ t\ i\ \textbf{then}\ c_1\ t\ \textbf{else}\ c_0\ t
\end{array}
$$

A class of decision tree algebras

Similarly, for decision trees, we use a class of tree algebras:

$$\textbf{class } \mathit{TreeAlg}\ a\ \textbf{where}$$
$$\quad \mathit{res} :: \mathit{Bool} \to a$$
$$\quad \mathit{pic} :: \mathit{Index} \to a \to a \to a$$

$$\mathit{ex1} :: \mathit{TreeAlg}\ a \Rightarrow a$$
$$\mathit{ex1} = \mathit{pic}\ 1\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{pic}\ 2\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1})))$$
$$\qquad\qquad (\mathit{pic}\ 2\ (\mathit{pic}\ 3\ (\mathit{res}\ \mathbf{0})\ (\mathit{res}\ \mathbf{1}))\ (\mathit{res}\ \mathbf{1}))$$

$$\textbf{instance } \mathit{Ring}\ a \Rightarrow \mathit{TreeAlg}\ (\mathit{ExpCost}\ a)\ \textbf{where } \mathit{res} = \mathit{resPoly}; \mathit{pic} = \mathit{pickPoly}$$
$$\textbf{type } \mathit{ExpCost}\ a = \mathit{Poly}\ a$$
$$\mathit{resPoly}\ \_b = \mathit{zero}$$
$$\mathit{pickPoly}\ \_\ p_0\ p_1 = \mathit{one} + (\mathit{one} - \mathit{xP}) * p_0 + \mathit{xP} * p_1$$

Problem formulation: compute the *complexity*

- generate all DecTrees for a given function

$$genAlg :: (BoFun\ bf, TreeAlg\ ta) \Rightarrow bf \to Set\ ta$$

- find a small(est) set of dominating trees

$$minWith :: Preorder\ b \Rightarrow (a \to b) \to Set\ a \to Set\ a$$

- then just keep their costs

$$\begin{aligned} &complexity :: (BoFun\ bf, Ring\ a) \Rightarrow bf \to Set\ (Poly\ a) \\ &complexity = mapS\ expCost \circ minWith\ expCost \circ genAlg \end{aligned}$$

Calculation steps

- We can push the map on the other side of *minWith*

$$
\begin{aligned}
&\mathit{complexity} = \mathit{mapS}\ \mathit{expCost} \circ \mathit{minWith}\ \mathit{expCost} \circ \mathit{genAlg} \\
&\text{-- is equal to} \\
&\mathit{complexity} = \mathit{minWith}\ \mathit{id} \circ \mathit{mapS}\ \mathit{expCost} \circ \mathit{genAlg}
\end{aligned}
$$

- Note that this makes the computation more efficient, beacause several trees map to the same polynomial

Calculation step, cont.

- Next, we push the *expCost* computation into the tree generation, to get another gain in efficiency.

$$complexity = minWith\ id \circ mapS\ expCost \circ genAlg$$
-- is equal to
$$complexity = minWith\ id \circ genPoly$$

- Still generating too many polynomials - we can push some of *minWith* into the *genPoly* computation. This is the thinning step.

$$complexity = minWith\ id \circ thin \circ genPoly$$
-- is equal to (the proof is work in progress)
$$complexity = minWith\ id \circ genPolyThin$$

Unfolding boolean functions

$$
\begin{aligned}
&\mathit{genAlg} :: (\mathit{BoFun}\ \mathit{bf}, \mathit{TreeAlg}\ \mathit{ta}) \Rightarrow \mathit{bf} \to \mathit{Set}\ \mathit{ta}\\
&\mathit{genAlg}\ n\ f = \mathbf{case}\ \mathit{isConst}\ f\ \mathbf{of}\\
&\quad \mathit{Just}\ b \quad \to \mathit{singleton}\ (\mathit{res}\ b)\\
&\quad \mathit{Nothing} \to \mathit{crossSigma}\ (\lambda i\ (t_0, t_1) \to \mathit{pic}\ i\ t_0\ t_1)\ [1\mathinner{..}n]\ \$\ \lambda i \to\\
&\qquad\qquad\qquad \mathit{cross}\ (\mathit{genAlg}\ (n-1)\ (\mathit{setBit}\ i\ \mathbf{0}\ f))\\
&\qquad\qquad\qquad\qquad (\mathit{genAlg}\ (n-1)\ (\mathit{setBit}\ i\ \mathbf{1}\ f))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{crossSigma} :: (a \to b \to c) \to \mathit{Set}\ a \to (a \to \mathit{Set}\ b) \to \mathit{Set}\ c\\
&\qquad \text{-- } \mathit{crossSigma}\ \mathit{op}\ \mathit{xs}\ f = \{\mathit{op}\ x\ y \mid x \leftarrow \mathit{xs}, y \leftarrow f\ x\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cross} :: \mathit{Set}\ a \to \mathit{Set}\ b \to \mathit{Set}\ (a, b)\\
&\mathit{cross}\ \mathit{xs}\ \mathit{ys} = \mathit{crossSigma}\ (,)\ \mathit{xs}\ (\backslash\_ \to \mathit{ys})\\
&\qquad \text{-- } \mathit{cross}\ \mathit{xs}\ \mathit{ys} = \{\mathit{op}\ x\ y \mid x \leftarrow \mathit{xs}, y \leftarrow \mathit{ys}\}
\end{aligned}
$$

Unfold structure

- At this point recall the structure of the computation: We have an "unfold", co-algebra structure when we consume a boolean function and produce all dectrees.
- *isConst* maps an *BoFun* into either a *const b* case or the *setBit* case which for each $i : Fin\ n$ has two subfunctions *setBit i F f* and *setBit i T f*.
- This can cause an enormous number of function calls, which indicates the need to do memoization (dynamic programming) when building up the "call graph".

Memoization

- Memoization reminder: naively computing $\mathit{fib}\ (n+2) = \mathit{fib}\ (n+1) + \mathit{fib}\ n$ leads to exponential growth in the number of function calls.
- But if we fill in a table indexed by n with already computed results we get a the result in linear time.

Memoization in our case

- Similarly, here we "just" need to tabulate the result of the calls to complexity so as to avoid recomputation.
- The challenge is that the input is now a boolean function which is not as nicely structured as a natural number index.
- Fortunately, thanks to Hinze and others we have generic Tries only a hackage library away.
- Finally, putting these components together we can compute the level-p-complexity of (small) boolean functions in reasonable time.

Motivating example: 2-level iterated majority *maj2*:

$$\mathit{ps9} = \mathit{genAlgThinMemoPoly}\ 9\ \mathit{maj2} :: \mathit{Set}\ (\mathit{Poly}\ \mathit{Rational})$$
$$\text{-- } \mathit{ps9} = \mathit{fromList}\ [P\ [4,4,6,9,-61,23,67,-64,16]]$$

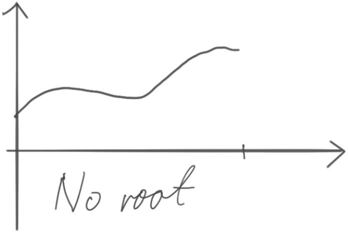
Comparing polynomials

- Now, one of the challenges was actually the implementation of the comparison of polynomials. This is an interesting story in itself but perhaps not the main topic for today.
- Remember that the "complexity" we compute for each decision tree is actually a polynomial in the probability for a bit to be $\mathbf{1}$.
- And the mathematical definition of the level-p-complexity includes a "min" over all dec.trees, which in general results in a piecewise polynomial function.
- Ideally we should have a partition of the unit interval, labelled with polynomials, but we choose to represent such piecewise polynomial function simply as a set of polynomials.

Comparing polynomials, cont.

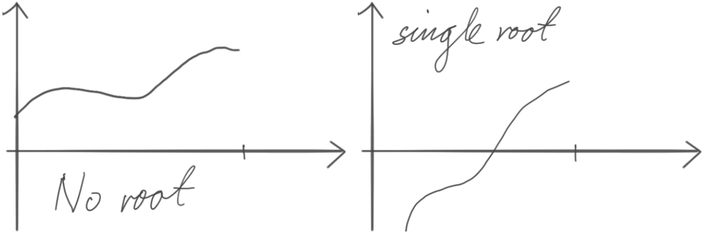
- For thinning we need a preorder on our polynomials, specified as $p \leqslant q = \mathit{forall}\ x.\ \mathit{eval}\ p\ x \leqslant \mathit{eval}\ q\ x.$
- This can be simplified to $\mathit{eval}\ (q - p)\ x \geqslant 0$ where $q - p$ is also a polynomial.
- We started with some ad-hoc special cases, but the sets did not shrink enough.
- Then we went to the wonderful world of polynomial algebra and root-counting.

Root-counting



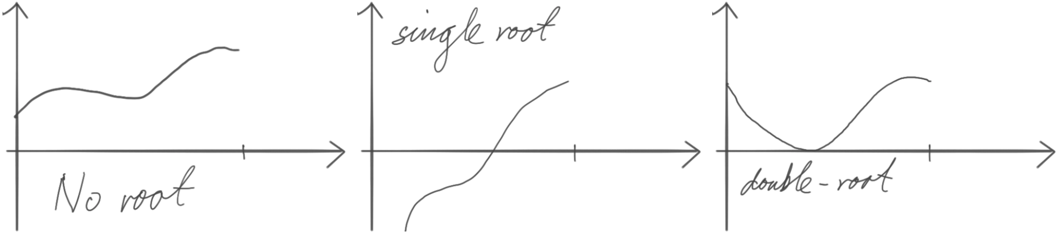
- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.

Root-counting



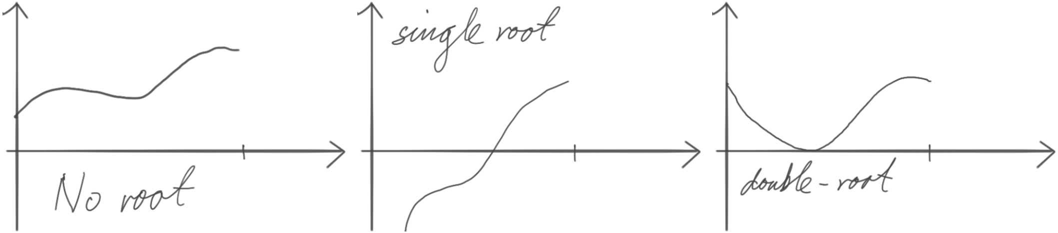
- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $p \leqslant q$ nor $q \leqslant p$).

Root-counting



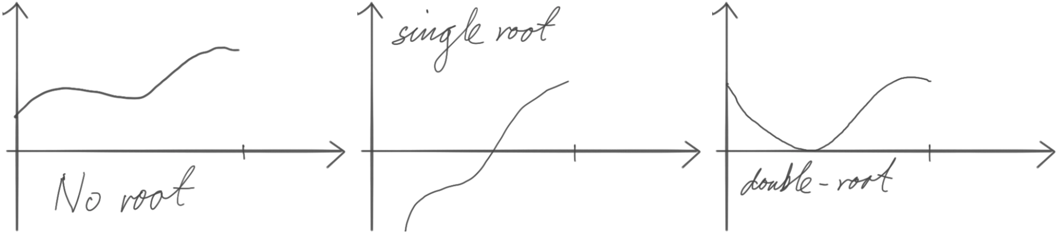
- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $p \leqslant q$ nor $q \leqslant p$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.

Root-counting



- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $p \leqslant q$ nor $q \leqslant p$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.
- Thus, we "only" need to count roots with exact multiplicities in the unit interval.

Root-counting



- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $p \leqslant q$ nor $q \leqslant p$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.
- Thus, we "only" need to count roots with exact multiplicities in the unit interval.
- Note that we do not need to actually compute the roots, only count them.

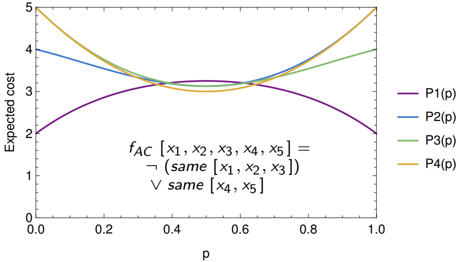
Root-counting

- First, note that if a polynomial has no root in the interval, it is enough to evaluate in one point to find if it is always greater or less than zero.
- Next, if there is any single root, the polynomial is on both sides and thus the original polynomials unrelated (neither $p \leqslant q$ nor $q \leqslant p$).
- For one double-root we are again on one side (or touching). In general, if we only have even-order roots, we are on one side.
- Thus, we "only" need to count roots with exact multiplicities in the unit interval.
- Note that we do not need to actually compute the roots, only count them.
- We use Yun's algorithm for square-free factorisation and Descartes rule of signs in combination with interval-halving.

Conclusions

- By using **thinning**, **memoization**, and exact comparisions of **polynomials** we get down to a **singleton set** for iterated majority:

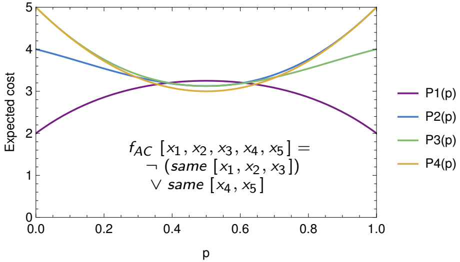
$$
\begin{aligned}
&ps9 = genAlgThinMemoPoly\ 9\ maj2 :: Set\ (Poly\ Rational) \\
&\quad \text{-- } ps9 = fromList\ [P\ [4,4,6,9,-61,23,67,-64,16]] \\
&ps5 = genAlgThinMemoPoly\ 5\ f_{AC} \quad :: Set\ (Poly\ Rational) \\
&check5 = ps5 == fromList\ [P\ [2,6,-10,8,-4], P\ [4,-2,-3,8,-2], \\
&\qquad\qquad\qquad\qquad\qquad P\ [5,-8,9,0,-2],\ \ P\ [5,-8,8]]
\end{aligned}
$$



Conclusions

- By using **thinning**, **memoization**, and exact comparisions of **polynomials** we get down to a **singleton set** for iterated majority:

$$
\begin{aligned}
&\mathit{ps9} = \mathit{genAlgThinMemoPoly}\ 9\ \mathit{maj2} :: \mathit{Set}\ (\mathit{Poly}\ \mathit{Rational})\\
&\quad \text{--}\ \mathit{ps9} = \mathit{fromList}\ [P\ [4,4,6,9,-61,23,67,-64,16]]\\
&\mathit{ps5} = \mathit{genAlgThinMemoPoly}\ 5\ f_{AC}\quad :: \mathit{Set}\ (\mathit{Poly}\ \mathit{Rational})\\
&\mathit{check5} = \mathit{ps5} == \mathit{fromList}\ [P\ [2,6,-10,8,-4], P\ [4,-2,-3,8,-2],\\
&\qquad\qquad\qquad\qquad\qquad P\ [5,-8,9,0,-2],\ \ P\ [5,-8,8]]
\end{aligned}
$$



Future work:

- Clean up the calculations / proofs
- More efficient memoization
- Three-level iterated majority (27 bits)