Alpha-Beta Pruning

Tobias Nipkow
Technical University of Munich

March 17, 2025

Abstract

Alpha-beta pruning is an efficient search strategy for two-player game trees. It was invented in the late 1950s and is at the heart of most implementations of combinatorial game playing programs. These theories formalize and verify a number of variations of alpha-beta pruning, in particular fail-hard and fail-soft, and valuations into linear orders, distributive lattices and domains with negative values.

A detailed presentation of these theories can be found in the chapter *Alpha-Beta Pruning* in the (forthcoming) book Functional Data Structures and Algorithms — A Proof Assistant Approach.

Chapter 1

Overview

1.1 Introduction

Alpha-beta pruning is an efficient search strategy for two-player game trees. It was invented in the late 1950s and is at the heart of most implementations of combinatorial game playing programs. Most publications assume that the game values are numbers augmented with $\pm\infty$. This generalizes easily to an arbitrary linear order with $\bot$ and $\top$ values. We consider this standard case first. Later it was realized that alpha-beta pruning can be generalized to distributive lattices. We consider this case separately. In both cases we analyze two variants: *fail-hard* (analyzed by Knuth and Moore [3]) and *fail-soft* (introduced by Fishburn [2]). Our analysis focusses on functional correctness, not quantitative results. All algorithms operate on game trees of this type:

$$\textbf{datatype}\ {}'a\ tree = Lf\ {}'a \mid Nd\ ({}'a\ tree\ list)$$

1.2 Linear Orders

We assume that the type of values is a bounded linear order with $\bot$ and $\top$. Game trees are evaluated in the standard manner via functions *maxmin* (the maximizer) and the dual *minmax* (the minimizer).

$$maxmin :: {}'a\ tree \Rightarrow {}'a$$

$$maxmin\ (Lf\ x) = x$$
$$maxmin\ (Nd\ ts) = maxs\ (map\ minmax\ ts)$$

$$minmax :: {}'a\ tree \Rightarrow {}'a$$

$$minmax\ (Lf\ x) = x$$
$$minmax\ (Nd\ ts) = mins\ (map\ maxmin\ ts)$$

$$maxs :: {}'a\ list \Rightarrow {}'a$$

$$
\begin{aligned}
&\mathit{maxs}\ [] = \bot\\
&\mathit{maxs}\ (x \# \mathit{xs}) = \mathit{max}\ x\ (\mathit{maxs}\ \mathit{xs})
\end{aligned}
$$

$$\mathit{mins} :: {}'a\ \mathit{list} \Rightarrow {}'a$$

$$
\begin{aligned}
&\mathit{mins}\ [] = \top\\
&\mathit{mins}\ (x \# \mathit{xs}) = \mathit{min}\ x\ (\mathit{mins}\ \mathit{xs})
\end{aligned}
$$

The maximizer and minimizer functions are dual to each other. In this overview we will only show the maximizer side from now on.

1.2.1 Fail-Hard

The fail-hard variant of alpha-beta pruning is defined like this:

$$\mathit{ab\_max} :: {}'a \Rightarrow {}'a \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a$$

$$
\begin{aligned}
&\mathit{ab\_max}\ \_\ \_\ (\mathit{Lf}\ x) = x\\
&\mathit{ab\_max}\ a\ b\ (\mathit{Nd}\ \mathit{ts}) = \mathit{ab\_maxs}\ a\ b\ \mathit{ts}
\end{aligned}
$$

$$\mathit{ab\_maxs} :: {}'a \Rightarrow {}'a \Rightarrow {}'a\ \mathit{tree}\ \mathit{list} \Rightarrow {}'a$$

$$
\begin{aligned}
&\mathit{ab\_maxs}\ a\ \_\ [] = a\\
&\mathit{ab\_maxs}\ a\ b\ (t \# \mathit{ts})\\
&= (\mathbf{let}\ a' = \mathit{max}\ a\ (\mathit{ab\_min}\ a\ b\ t)\\
&\quad\ \mathbf{in\ if}\ b \le a'\ \mathbf{then}\ a'\ \mathbf{else}\ \mathit{ab\_maxs}\ a'\ b\ \mathit{ts})
\end{aligned}
$$

The intuitive meaning of $\mathit{ab\_max}\ a\ b\ t$ roughly is this: search t, focussing on values in the interval from a to b. That is, a is the maximum value that the maximizer is already assured of and b is the minimum value that the minimizer is already assured of (by the search so far). During the search, the maximizer will increase a, the minimizer will decrease b.

The desired correctness property is that alpha-beta pruning with the full interval yields the value of the game tree:

$$\mathit{ab\_max}\ \bot\ \top\ t = \mathit{maxmin}\ t \tag{1.1}$$

Knuth and Moore generalize this property for arbitrary a and b, using the following predicate:

$$
\begin{aligned}
&x \cong y\ (\mathrm{mod}\ a,b) \equiv\\
&((y \le a \longrightarrow x \le a) \wedge\\
&(a < y \wedge y < b \longrightarrow y = x) \wedge\\
&(b \le y \longrightarrow b \le x))
\end{aligned}
$$

It follows easily that $x \cong y\ (\mathrm{mod}\ \bot,\top)$ implies $x = y$. (Also interesting to note is commutativity: $a < b \Longrightarrow x \cong y\ (\mathrm{mod}\ a,b) = y \cong x\ (\mathrm{mod}\ a,b)$.) We have verified Knuth and Moore's correctness theorem

$$a < b \Longrightarrow \mathit{maxmin}\ t \cong \mathit{ab\_max}\ a\ b\ t\ (\mathrm{mod}\ a,b)$$

which immediately implies (1.1).

1.2.2 Fail-Soft

Fishburn introduced the fail-soft variant that agrees with fail-hard if the value is in between a and b but is more precise otherwise, where fail-hard just returns a or b:

$$ab\_max' :: 'a \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$$

$$\begin{aligned}
&ab_max'\ _\ _\ (Lf\ x) = x\\
&ab_max'\ a\ b\ (Nd\ ts) = ab_maxs'\ a\ b\ \bot\ ts
\end{aligned}$$

$$ab\_maxs' :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ tree\ list \Rightarrow 'a$$

$$\begin{aligned}
&ab_maxs'\ _\ _\ m\ [] = m\\
&ab_maxs'\ a\ b\ m\ (t\ \#\ ts)\\
&= (\mathbf{let}\ m' = max\ m\ (ab_min'\ (max\ m\ a)\ b\ t)\\
&\quad\ \mathbf{in\ if}\ b \le m'\ \mathbf{then}\ m'\ \mathbf{else}\ ab_maxs'\ a\ b\ m'\ ts)
\end{aligned}$$

Fishburn claims that fail-soft searches the same part of the tree as fail-hard but that sometimes fail-soft bounds the real value more tightly than fail-hard because fail-soft satisfies

$$a < b \Longrightarrow ab\_max'\ a\ b\ t \le maxmin\ t\ (\mathrm{mod}\ a,b) \tag{1.2}$$

where $\le$ is a strengthened version of $\cong$:

$$\begin{aligned}
&ab \le v\ (\mathrm{mod}\ a,b) \equiv\\
&((ab \le a \longrightarrow v \le ab)\ \wedge\\
&(a < ab \wedge ab < b \longrightarrow ab = v)\ \wedge\\
&(b \le ab \longrightarrow ab \le v))
\end{aligned}$$

We can confirm both claims. However, what Fishburn misses is that fail-hard already satisfies *fishburn*:

$$a < b \Longrightarrow ab\_max\ a\ b\ t \le maxmin\ t\ (\mathrm{mod}\ a,b)$$

Thus (1.2) does not imply that fail-soft is better. However, we have proved

$$a < b \Longrightarrow ab\_max\ a\ b\ t \le ab\_max'\ a\ b\ t\ (\mathrm{mod}\ a,b)$$

which does indeed show that fail-soft approximates the real value at least as well as fail-hard.

Fail-soft does not improve matters if one only looks at the top-level call with $\bot$ and $\top$. It comes into its own when the tree is actually a graph and nodes are visited repeatedly from different directions with different a and b which are typically not $\bot$ and $\top$. Then it becomes crucial to store the results of previous alpha-beta calls in a cache and use it to (possibly) narrow the search window on successive searches of the same subgraph. The justification: Let ab be some search function that *fishburn* the real value. If on a previous call $b \le ab\ a\ b\ t$, then in a subsequent search of the same t with possibly different a' and b', we can replace a' by $max\ a'\ (ab\ a\ b\ t)$:

$$\llbracket\forall a\ b.\ abf\ a\ b\ t \le maxmin\ t\ (\mathrm{mod}\ a{,}b);\ b \le abf\ a\ b\ t;\ max\ a'\ (abf\ a\ b\ t) < b'\rrbracket \Longrightarrow abf\ (max\ a'\ (abf\ a\ b\ t))\ b'\ t \le maxmin\ t\ (\mathrm{mod}\ a'{,}b')$$

There is a dual lemma for replacing b' by $min\ b'\ (ab\ a\ b\ t)$.

We have a verified version of alpha-beta pruning with a cache, but it is not yet part of this development.

1.2.3 Negation

Traditionally the definition of both the evaluation and of alpha-beta pruning does not define a minimizer and maximizer separately. Instead it defines only one function and uses negation (on numbers!) to flip between the two players. This is evaluation and the fail-hard and fail-soft variants of alpha-beta pruning:

$negmax :: 'a\ tree \Rightarrow 'a$

$negmax\ (Lf\ x) = x$
$negmax\ (Nd\ ts) = maxs\ (map\ (\lambda t.\ -\ negmax\ t)\ ts)$

$ab\_negmax :: 'a \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$

$ab\_negmax\ \_\ \_\ (Lf\ x) = x$
$ab\_negmax\ a\ b\ (Nd\ ts) = ab\_negmaxs\ a\ b\ ts$

$ab\_negmaxs :: 'a \Rightarrow 'a \Rightarrow 'a\ tree\ list \Rightarrow 'a$

$ab\_negmaxs\ a\ \_\ [] = a$
$ab\_negmaxs\ a\ b\ (t\ \#\ ts)$
$= (\mathbf{let}\ a' = max\ a\ (-\ ab\_negmax\ (-\ b)\ (-\ a)\ t)$
$\quad \mathbf{in\ if}\ b \le a'\ \mathbf{then}\ a'\ \mathbf{else}\ ab\_negmaxs\ a'\ b\ ts)$

$ab\_negmax' :: 'a \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$

$ab\_negmax'\ \_\ \_\ (Lf\ x) = x$
$ab\_negmax'\ a\ b\ (Nd\ ts) = ab\_negmaxs'\ a\ b\ \bot\ ts$

$ab\_negmaxs' :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ tree\ list \Rightarrow 'a$

$ab\_negmaxs'\ \_\ \_\ m\ [] = m$
$ab\_negmaxs'\ a\ b\ m\ (t\ \#\ ts)$
$= (\mathbf{let}\ m' = max\ m\ (-\ ab\_negmax'\ (-\ b)\ (-\ max\ m\ a)\ t)$
$\quad \mathbf{in\ if}\ b \le m'\ \mathbf{then}\ m'\ \mathbf{else}\ ab\_negmaxs'\ a\ b\ m'\ ts)$

However, what does "$-$" on a linear order mean? It turns out that the following two properties of "$-$" are required to make things work:

$$-\ min\ x\ y = max\ (-\ x)\ (-\ y) \qquad -\ (-\ x) = x$$

We call such linear orders *de Morgan orders*. We have proved correctness of alpha-beta pruning on de Morgan orders:

$a < b \Longrightarrow ab\_negmax\ a\ b\ t \leq negmax\ t\ (\mathrm{mod}\ a{,}b)$
$a < b \Longrightarrow ab\_negmax'\ a\ b\ t \leq negmax\ t\ (\mathrm{mod}\ a{,}b)$
$a < b \Longrightarrow ab\_negmax\ a\ b\ t \leq ab\_negmax'\ a\ b\ t\ (\mathrm{mod}\ a{,}b)$

1.3 Lattices

Bird and Hughes [1] were the first to generalize alpha-beta pruning from linear orders to lattices. The generalization of the code is easy: simply replace *min* and *max* by $(\sqcap)$ and $(\sqcup)$. Thus, the value of a game tree is now defined like this:

$supinf :: 'a\ tree \Rightarrow 'a$

$supinf\ (Lf\ x) = x$
$supinf\ (Nd\ ts) = sups\ (map\ infsup\ ts)$

$sups :: 'a\ list \Rightarrow 'a$

$sups\ [] = \bot$
$sups\ (x\ \#\ xs) = x \sqcup sups\ xs$

And similarly fail-hard and fail-soft alpha-beta pruning:

$ab\_sup :: 'a \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$

$ab\_sup\ \_\ \_\ (Lf\ x) = x$
$ab\_sup\ a\ b\ (Nd\ ts) = ab\_sups\ a\ b\ ts$

$ab\_sups :: 'a \Rightarrow 'a \Rightarrow 'a\ tree\ list \Rightarrow 'a$

$ab\_sups\ a\ \_\ [] = a$
$ab\_sups\ a\ b\ (t\ \#\ ts)$
$= (\mathbf{let}\ a' = a \sqcup ab\_inf\ a\ b\ t$
$\quad \mathbf{in\ if}\ b \leq a'\ \mathbf{then}\ a'\ \mathbf{else}\ ab\_sups\ a'\ b\ ts)$

$ab\_sup' :: 'a \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$

$ab\_sup'\ \_\ \_\ (Lf\ x) = x$
$ab\_sup'\ a\ b\ (Nd\ ts) = ab\_sups'\ a\ b\ \bot\ ts$

$ab\_sups' :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ tree\ list \Rightarrow 'a$

$ab\_sups'\ \_\ \_\ m\ [] = m$
$ab\_sups'\ a\ b\ m\ (t\ \#\ ts)$
$= (\mathbf{let}\ m' = m \sqcup ab\_inf'\ (m \sqcup a)\ b\ t$
$\quad \mathbf{in\ if}\ b \leq m'\ \mathbf{then}\ m'\ \mathbf{else}\ ab\_sups'\ a\ b\ m'\ ts)$

It turns out that this version of alpha-beta pruning works for bounded distributive lattices. To prove this we can generalize both $\cong$ and $\leq$ by first rephrasing them (for linear orders)

$$a < b \Longrightarrow x \cong y \ (\mathrm{mod}\ a,b) = (\mathit{max}\ a\ (\mathit{min}\ x\ b) = \mathit{max}\ a\ (\mathit{min}\ y\ b))$$
$$a < b \Longrightarrow ab \leq v \ (\mathrm{mod}\ a,b) = (\mathit{min}\ v\ b \leq ab \wedge ab \leq \mathit{max}\ v\ a)$$

and then deriving from the right-hand sides new versions $\simeq$ and $\sqsubseteq$ appropriate for lattices:

$$x \simeq y \ (\mathrm{mod}\ a,b) \equiv a \sqcup x \sqcap b = a \sqcup y \sqcap b$$
$$ab \sqsubseteq v \ (\mathrm{mod}\ a,b) \equiv b \sqcap v \leq ab \wedge ab \leq a \sqcup v$$

As for linear orders, $\sqsubseteq$ implies $\simeq$, but not the other way around:

$$ab \sqsubseteq v \ (\mathrm{mod}\ a,b) \Longrightarrow ab \simeq v \ (\mathrm{mod}\ a,b)$$

Both fail-hard and fail-soft are correct w.r.t. $\sqsubseteq$:

$$\mathit{ab\_sup}\ a\ b\ t \sqsubseteq \mathit{supinf}\ t\ (\mathrm{mod}\ a,b)$$
$$\mathit{ab\_sup'}\ a\ b\ t \sqsubseteq \mathit{supinf}\ t\ (\mathrm{mod}\ a,b)$$

1.3.1 Negation

This time we extend bounded distributive lattices to *de Morgan algebras* by adding "$-$" and assuming

$$-\ (x \sqcap y) = -\ x \sqcup -\ y \qquad -\ (-\ x) = x$$

Game tree evaluation:

$$\mathit{negsup} :: {}'a\ \mathit{tree} \Rightarrow {}'a$$
$$\mathit{negsup}\ (\mathit{Lf}\ x) = x$$
$$\mathit{negsup}\ (\mathit{Nd}\ \mathit{ts}) = \mathit{sups}\ (\mathit{map}\ (\lambda t.\ -\ \mathit{negsup}\ t)\ \mathit{ts})$$

Fail-hard alpha-beta pruning:

$$\mathit{ab\_negsup} :: {}'a \Rightarrow {}'a \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a$$
$$\mathit{ab\_negsup}\ \_\ \_\ (\mathit{Lf}\ x) = x$$
$$\mathit{ab\_negsup}\ a\ b\ (\mathit{Nd}\ \mathit{ts}) = \mathit{ab\_negsups}\ a\ b\ \mathit{ts}$$
$$\mathit{ab\_negsups} :: {}'a \Rightarrow {}'a \Rightarrow {}'a\ \mathit{tree}\ \mathit{list} \Rightarrow {}'a$$
$$\mathit{ab\_negsups}\ a\ \_\ [] = a$$
$$\mathit{ab\_negsups}\ a\ b\ (t\ \#\ \mathit{ts})$$
$$= (\textbf{let}\ a' = a \sqcup -\ \mathit{ab\_negsup}\ (-\ b)\ (-\ a)\ t$$
$$\quad \textbf{in if}\ b \leq a'\ \textbf{then}\ a'\ \textbf{else}\ \mathit{ab\_negsups}\ a'\ b\ \mathit{ts})$$

Fail-soft alpha-beta pruning:

$$\mathit{ab\_negsup'} :: {}'a \Rightarrow {}'a \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a$$
$$\mathit{ab\_negsup'}\ \_\ \_\ (\mathit{Lf}\ x) = x$$
$$\mathit{ab\_negsup'}\ a\ b\ (\mathit{Nd}\ \mathit{ts}) = \mathit{ab\_negsups'}\ a\ b\ \bot\ \mathit{ts}$$

$$ab\_negsups' :: {}'a \Rightarrow {}'a \Rightarrow {}'a \Rightarrow {}'a\ tree\ list \Rightarrow {}'a$$

$$\begin{array}{l}
ab_negsups'\ _\ _\ m\ [] = m \\
ab_negsups'\ a\ b\ m\ (t\ \#\ ts) \\
= (\mathbf{let}\ m' = m \sqcup -\ ab_negsup'\ (-\ b)\ (-\ (m \sqcup a))\ t \\
\quad \mathbf{in}\ \mathbf{if}\ b \leq m'\ \mathbf{then}\ m'\ \mathbf{else}\ ab_negsups'\ a\ b\ m'\ ts)
\end{array}$$

Correctness:

$$\begin{array}{l}
ab_negsup\ a\ b\ t \sqsubseteq negsup\ t\ (\mathrm{mod}\ a{,}b) \\
ab_negsup'\ a\ b\ t \sqsubseteq negsup\ t\ (\mathrm{mod}\ a{,}b)
\end{array}$$

Bibliography

[1] R. S. Bird and J. Hughes. The alpha-beta algorithm: An exercise in program transformation. *Inf. Process. Lett.*, 24(1):53–57, 1987.

[2] J. P. Fishburn. An optimization of alpha-beta search. *SIGART Newsl.*, 72:29–31, 1980.

[3] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artif. Intell.*, 6(4):293–326, 1975.

Contents

Chapter 2

Linear Orders

theory *Alpha_Beta_Linear*
imports
  HOL−Library.Extended_Real
begin

2.1 Classes

notation
  bot (‹⊥›) **and**
  top (‹⊤›)

class *bounded_linorder* = *linorder* + *order_top* + *order_bot*
begin

lemma *bounded_linorder_collapse*:
assumes ¬ ⊥ < ⊤ **shows** (*x*::′*a*) = *y*
proof −
  from *assms* **have** ⊤ ≤ ⊥ **by**(*rule leI*)
  have *x* = ⊤ **using** *order.trans*[*OF* ‹⊤ ≤ ⊥› *bot_least*[*of x*]] *top_unique* **by** *metis*
  moreover
  have *y* = ⊤ **using** *order.trans*[*OF* ‹⊤ ≤ ⊥› *bot_least*[*of y*]] *top_unique* **by** *metis*
  ultimately show *?thesis* **by** *blast*
qed

end

class *de_morgan_order* = *bounded_linorder* + *uminus* +
assumes *de_morgan_min*: − *min x y* = *max* (− *x*) (− *y*)
assumes *neg_neg*[*simp*]: − (− *x*) = *x*
begin

lemma *de_morgan_max*: − *max x y* = *min* (− *x*) (− *y*)
by (*metis de_morgan_min neg_neg*)

lemma *neg_top*[*simp*]: $- \top = \bot$
by (*metis de_morgan_max max_top*2 *min_bot neg_neg*)

lemma *neg_bot*[*simp*]: $- \bot = \top$
using *neg_neg neg_top* **by** *blast*

lemma *uminus_eq_iff*[*simp*]: $-a = -b \longleftrightarrow a = b$
by (*metis neg_neg*)

lemma *uminus_le_reorder*: $(- a \leq b) = (- b \leq a)$
by (*metis de_morgan_max max.absorb_iff*1 *min.absorb_iff*1 *neg_neg*)

lemma *uminus_less_reorder*: $(- a < b) = (- b < a)$
by (*metis order.strict_iff_not neg_neg uminus_le_reorder*)

lemma *minus_le_minus*[*simp*]: $- a \leq - b \longleftrightarrow b \leq a$
by (*metis neg_neg uminus_le_reorder*)

lemma *minus_less_minus*[*simp*]: $- a < - b \longleftrightarrow b < a$
by (*metis neg_neg uminus_less_reorder*)

lemma *less_uminus_reorder*: $a < - b \longleftrightarrow b < - a$
by (*metis neg_neg uminus_less_reorder*)

end

instance *bool* :: *bounded_linorder* **..**

instantiation *ereal* :: *de_morgan_order*
begin

instance
proof (*standard*, *goal_cases*)
  case 1
  thus *?case*
    by (*simp add*: *max_def*)
next
  case 2
  thus *?case* **by** (*simp add*: *min_def*)
qed

end

2.2 Game Tree Evaluation

datatype *'a tree* = *Lf 'a* | *Nd 'a tree list*

datatype_compat *tree*

fun *maxs* :: ('*a*::*bounded_linorder*) *list* ⇒ '*a* **where**
maxs [] = ⊥ |
maxs (*x*#*xs*) = *max x* (*maxs xs*)

fun *mins* :: ('*a*::*bounded_linorder*) *list* ⇒ '*a***where**
mins [] = ⊤ |
mins (*x*#*xs*) = *min x* (*mins xs*)

fun *maxmin* :: ('*a*::*bounded_linorder*) *tree* ⇒ '*a*
and *minmax* :: ('*a*::*bounded_linorder*) *tree* ⇒ '*a* **where**
maxmin (*Lf x*) = *x* |
maxmin (*Nd ts*) = *maxs* (*map minmax ts*) |
minmax (*Lf x*) = *x* |
minmax (*Nd ts*) = *mins* (*map maxmin ts*)

Cannot use *Max* instead of *maxs* because *Max* {} is undefined.

No need for bounds if lists are nonempty:

fun *invar* :: '*a tree* ⇒ *bool* **where**
invar (*Lf x*) = *True* |
invar (*Nd ts*) = (*ts* ≠ [] ∧ (∀ *t* ∈ *set ts*. *invar t*))

fun *maxs*1 :: ('*a*::*linorder*) *list* ⇒ '*a* **where**
*maxs*1 [*x*] = *x* |
*maxs*1 (*x*#*xs*) = *max x* (*maxs*1 *xs*)

fun *mins*1 :: ('*a*::*linorder*) *list* ⇒ '*a***where**
*mins*1 [*x*] = *x* |
*mins*1 (*x*#*xs*) = *min x* (*mins*1 *xs*)

fun *maxmin*1 :: ('*a*::*bounded_linorder*) *tree* ⇒ '*a*
and *minmax*1 :: ('*a*::*bounded_linorder*) *tree* ⇒ '*a* **where**
*maxmin*1 (*Lf x*) = *x* |
*maxmin*1 (*Nd ts*) = *maxs*1 (*map minmax*1 *ts*) |
*minmax*1 (*Lf x*) = *x* |
*minmax*1 (*Nd ts*) = *mins*1 (*map maxmin*1 *ts*)

lemma *maxs*1_*maxs*: *xs* ≠ [] ⟹ *maxs*1 *xs* = *maxs xs*
by(*induction xs rule*: *maxs*1.*induct*) *auto*

lemma *mins*1_*mins*: *xs* ≠ [] ⟹ *mins*1 *xs* = *mins xs*
by(*induction xs rule*: *mins*1.*induct*) *auto*

lemma *maxmin*1_*maxmin*:
shows *invar t* ⟹ *maxmin*1 *t* = *maxmin t*
and *invar t* ⟹ *minmax*1 *t* = *minmax t*
proof(*induction t rule*: *maxmin*1_*minmax*1.*induct*)
  case 2 **thus** *?case* **by** (*simp add*: *maxs*1_*maxs cong*: *map_cong*)

next
  case 4 **thus** *?case* **by** (*simp add*: *mins*1_*mins cong*: *map_cong*)
qed *auto*

2.2.1 Parameterized by the orderings

fun *maxs_le* :: $'a \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a$ **where**
maxs_le bo le [] = *bo* |
maxs_le bo le (*x*#*xs*) = (*let m* = *maxs_le bo le xs in if le x m then m else x*)

fun *maxmin_le* :: $'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ tree \Rightarrow 'a$ **where**
maxmin_le _ _ _ (*Lf x*) = *x* |
maxmin_le bo to le (*Nd ts*) = *maxs_le bo le* (*map* (*maxmin_le to bo* ($\lambda x\ y.$ *le y x*)) *ts*)

lemma *maxs_le_maxs*: *maxs_le* $\bot$ ($\leq$) *xs* = *maxs xs*
by(*induction xs*) (*auto simp*: *Let_def*)

lemma *maxs_le_mins*: *maxs_le* $\top$ ($\geq$) *xs* = *mins xs*
by(*induction xs*) (*auto simp*: *Let_def*)

lemma *maxmin_le_maxmin*:
  shows *maxmin_le* $\bot$ $\top$ ($\leq$) *t* = *maxmin t*
  and *maxmin_le* $\top$ $\bot$ ($\geq$) *t* = *minmax t*
by(*induction t* **and** *t rule*: *maxmin_minmax.induct*)
 (*auto simp add*: *Let_def max_def maxs_le_maxs maxs_le_mins cong*: *map_cong*)

2.2.2 Negamax: de Morgan orders

fun *negmax* :: ($'a$::*de_morgan_order*) *tree* $\Rightarrow 'a$ **where**
negmax (*Lf x*) = *x* |
negmax (*Nd ts*) = *maxs* (*map* ($\lambda t.$ − *negmax t*) *ts*)

lemma *de_morgan_mins*:
fixes *f* :: $'a \Rightarrow 'b$::*de_morgan_order*
shows − *mins* (*map f xs*) = *maxs* (*map* ($\lambda x.$ − *f x*) *xs*)
by(*induction xs*)(*auto simp*: *de_morgan_min*)

fun *negate* :: *bool* $\Rightarrow$ ($'a$::*de_morgan_order*) *tree* $\Rightarrow$ ($'a$::*de_morgan_order*) *tree* **where**
negate b (*Lf x*) = *Lf* (*if b then* −*x else x*) |
negate b (*Nd ts*) = *Nd* (*map* (*negate* ($\neg$*b*)) *ts*)

lemma *negate_negate*: *negate f* (*negate f t*) = *t*
by(*induction t arbitrary*: *f*)(*auto cong*: *map_cong*)

lemma *maxmin_negmax*: *maxmin t* = *negmax* (*negate False t*)
and *minmax_negmax*: *minmax t* = − *negmax* (*negate True t*)
by(*induction t* **and** *t rule*: *maxmin_minmax.induct*)
  (*auto simp flip*: *de_morgan_mins cong*: *map_cong*)

```
lemma maxmin t = negmax (negate False t)
and  minmax t = − negmax (negate True t)
proof(induction t and t rule: maxmin_minmax.induct)
  case (2 ts)
  have maxmin (Nd ts) = maxs (map minmax ts)
    by simp
  also have ... = maxs (map (λt. − negmax (negate True t)) ts)
    using 2.IH by (simp cong: map_cong)
  also have ... = maxs (map ((λt. − negmax t) o (negate True)) ts)
    by (metis comp_apply)
  also have ... = maxs (map (λt. − negmax t) (map (negate True) ts))
    by(simp)
  also have ... = negmax (Nd (map (negate True) ts))
    by simp
  also have ... = negmax (negate False (Nd ts))
    by simp
  finally show ?case .
next
  case (4 ts)
  have minmax (Nd ts) = mins (map maxmin ts)
    by simp
  also have ... = mins (map (λt. negmax (negate False t)) ts)
    using 4.IH by (simp cong: map_cong)
  also have ... = − (− mins (map (λt. negmax (negate False t)) ts))
    by simp
  also have ... = − maxs (map (λt. − negmax (negate False t)) ts)
    by(simp only: de_morgan_mins)
  also have ... = − maxs (map ((λt. − negmax t) o (negate False)) ts)
    by (metis comp_apply)
  also have ... = − maxs (map (λt. − negmax t) (map (negate False) ts))
    by(simp)
  also have ... = − negmax (Nd (map (negate False) ts))
    by simp
  also have ... = − negmax (negate True (Nd ts))
    by simp
  finally show ?case .
qed (auto)

lemma shows negmax_maxmin: negmax t = maxmin(negate False t)
and negmax t = − minmax(negate True t)
  apply (simp add: maxmin_negmax negate_negate)
  by (simp add: minmax_negmax negate_negate)


lemma maxs_append: maxs (xs @ ys) = max (maxs xs) (maxs ys)
by(induction xs) (auto simp: max.assoc)
```

lemma *maxs_rev*: *maxs* (*rev xs*) = *maxs xs*
by(*induction xs*) (*auto simp*: *max.commute maxs_append*)

2.3 Specifications

2.3.1 The squash operator *max a* (*min x b*)

abbreviation *mm* **where** *mm a x b* == *min* (*max a x*) *b*

lemma *max_min_commute*: (*a*::_::*linorder*) ≤ *b* ⟹ *max a* (*min x b*) = *min b* (*max x a*)
by (*metis max.absorb2 max.commute min.commute max_min_distrib1*)

lemma *max_min_commute2*: (*a*::_::*linorder*) ≤ *b* ⟹ *max a* (*min x b*) = *min* (*max a x*) *b*
by (*metis max.absorb2 max.commute max_min_distrib1*)

lemma *max_min_neg*: *a*<*b* ⟹ *max* (*a*::_::*de_morgan_order*) (*min x b*) = − *max* (−*b*) (*min* (−*x*) (−*a*))
by(*simp add*: *max_min_commute de_morgan_min de_morgan_max*)

2.3.2 Fail-Hard and Soft

Specification of fail-hard; symmetric in x and y!

abbreviation
knuth (*a*::_::*linorder*) *b x y* ==
((*y* ≤ *a* ⟶ *x* ≤ *a*) ∧ (*a* < *y* ∧ *y* < *b* ⟶ *y* = *x*) ∧ (*b* ≤ *y* ⟶ *b* ≤ *x*))

abbreviation *knuth2* :: ('*a*::*linorder*) ⇒ '*a* ⇒ '*a* ⇒ '*a* ⇒ *bool* (‹(_ ≅/ _/ '(mod _,_'))› [51,51,0,0])
where *knuth2 x y a b* ≡ *knuth a b x y*

notation (*latex* **output**) *knuth2* (‹(_ ≅/ _/ '(mod _,_'))› [51,51,0,0])

lemma *knuth_bot_top*: *knuth* ⊥ ⊤ *x y* ⟹ *x* = (*y*::_::*bounded_linorder*)
by (*metis bot.extremum_uniqueI linorder_le_less_linear top.extremum_uniqueI*)

The equational version of *knuth*. First, automatically:

lemma *knuth_iff_max_min*: *a* < *b* ⟹ *knuth a b x y* ⟷ *max a* (*min x b*) = *max a* (*min y b*)
by (*smt* (*verit*) *linorder_not_le max.absorb4 max_min_distrib2 max_min_same*(1) *min.absorb_iff2*)

Needs $a < b$: take everything = ∞, x = 0

lemma *knuth_if_mm*: *a* < *b* ⟹ *mm a y b* = *mm a x b* ⟹ *knuth a b x y*
by (*smt* (*verit*, *del_insts*) *le_max_iff_disj min_def nle_le nless_le*)

lemma *mm_if_knuth*: *knuth a b y x* ⟹ *mm a y b* = *mm a x b*
by (*metis leI max.orderE min.absorb_iff2 min_max_distrib1*)

Now readable:

lemma *mm_iff_knuth*: **assumes** $(a::\_::linorder) < b$
shows $max\ a\ (min\ x\ b) = max\ a\ (min\ y\ b) \longleftrightarrow knuth\ a\ b\ y\ x$ (**is** *?mm = ?h*)
proof −
 have $max\ a\ (min\ x\ b) = max\ a\ (min\ y\ b) \longleftrightarrow$
  $(min\ x\ b \le a \longleftrightarrow min\ y\ b \le a) \wedge (a < min\ x\ b \longrightarrow min\ x\ b = min\ y\ b)$
  by (*metis linorder_not_le max_def nle_le*)
 also have $\ldots \longleftrightarrow (x \le a \longleftrightarrow y \le a) \wedge (a < x \longrightarrow min\ x\ b = min\ y\ b)$
  using *assms* **apply** (*simp add*: *linorder_not_le*) **by** (*metis leD min_le_iff_disj*)
 also have $\ldots \longleftrightarrow (x \le a \longleftrightarrow y \le a) \wedge (a < x \longrightarrow (b \le x \longleftrightarrow b \le y) \wedge (x < b \longrightarrow x{=}y))$
  by (*metis leI min.strict_order_iff min_absorb2*)
 also have $\ldots \longleftrightarrow (x \le a \longleftrightarrow y \le a) \wedge (a < x \longrightarrow (b \le x \longleftrightarrow b \le y)) \wedge (a < x \wedge x < b \longrightarrow x{=}y)$
  by *blast*
 also have $\ldots \longleftrightarrow (x \le a \longleftrightarrow y \le a) \wedge (b \le x \longleftrightarrow b \le y) \wedge (a < x \wedge x < b \longrightarrow x{=}y)$
  using *assms dual_order.strict_trans2 linorder_not_less* **by** *blast*
 also have $\ldots \longleftrightarrow (x \le a \longrightarrow y \le a) \wedge (x \ge b \longrightarrow y \ge b) \wedge (a < x \wedge x < b \longrightarrow x{=}y)$
  by (*metis assms order.strict_trans linorder_le_less_linear nless_le*)
 finally show *?thesis* **by** *blast*
qed

corollary *mm_iff_knuth'*: $a < b \Longrightarrow max\ a\ (min\ x\ b) = max\ a\ (min\ y\ b) \longleftrightarrow knuth\ a\ b\ x\ y$
using *mm_iff_knuth* **by** (*metis mm_iff_knuth*)

corollary *knuth_comm*: $a{<}b \Longrightarrow knuth\ a\ b\ x\ y \longleftrightarrow knuth\ a\ b\ y\ x$
using *mm_iff_knuth*[*of a b x y*] *mm_iff_knuth*[*of a b y x*]
by *metis*

Specification of fail-soft: v is the actual value, ab the approximation.

abbreviation
 fishburn $(a::\_::linorder)\ b\ v\ ab ==$
 $((ab \le a \longrightarrow v \le ab) \wedge (a < ab \wedge ab < b \longrightarrow ab = v) \wedge (b \le ab \longrightarrow ab \le v))$

abbreviation *fishburn2* :: $('a::linorder) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ (‹(_ ≤/ _/ '(mod _,_'))› [51,51,0,0])
where *fishburn2 ab v a b* ≡ *fishburn a b v ab*

notation (*latex* **output**) *fishburn2* (‹(_ ≤/ _/ '(mod _,_'))› [51,51,0,0])

lemma *fishburn_iff_min_max*: $a < b \Longrightarrow fishburn\ a\ b\ v\ ab \longleftrightarrow min\ v\ b \le ab \wedge ab \le max\ v\ a$
by (*metis* (*full_types*) *le_max_iff_disj linorder_not_le min_le_iff_disj nle_le*)

lemma *knuth_if_fishburn*: $\mathit{fishburn}\ a\ b\ x\ y \Longrightarrow \mathit{knuth}\ a\ b\ x\ y$
using *order_trans* **by** *blast*

corollary *fishburn_bot_top*: $\mathit{fishburn}\ \bot\ \top\ (x{::}\_{::}\mathit{bounded\_linorder})\ y \Longrightarrow x = y$
by (*metis bot.extremum bot.not_eq_extremum nle_le top.not_eq_extremum top_greatest*)

lemma *trans_fishburn*: $\mathit{fishburn}\ a\ b\ x\ y \Longrightarrow \mathit{fishburn}\ a\ b\ y\ z \Longrightarrow \mathit{fishburn}\ a\ b\ x\ z$
using *order.trans* **by** *blast*

An simple alternative formulation:

lemma *fishburn2*: $a < b \Longrightarrow \mathit{fishburn}\ a\ b\ f\ g = ((g > a \longrightarrow f \geq g) \wedge (g < b \longrightarrow f \leq g))$
by *auto*

Like *fishburn2* above, but exchanging f and g. Not clearly related to *knuth* and *fishburn*.

abbreviation $\mathit{lb\_ub}\ a\ b\ f\ g \equiv ((f \geq a \longrightarrow g \geq a) \wedge (f \leq b \longrightarrow g \leq b))$

lemma $(a{::}\mathit{nat}) < b \Longrightarrow \mathit{knuth}\ a\ b\ f\ g \Longrightarrow \mathit{lb\_ub}\ a\ b\ f\ g$
quickcheck[*expect=counterexample*]
oops

lemma $(a{::}\mathit{nat}) < b \Longrightarrow \mathit{lb\_ub}\ a\ b\ f\ g \Longrightarrow \mathit{knuth}\ a\ b\ f\ g$
quickcheck[*expect=counterexample*]
oops

lemma $\mathit{fishburn}\ a\ b\ f\ g \Longrightarrow \mathit{lb\_ub}\ a\ b\ f\ g$
by (*metis order.trans nle_le*)

lemma $(a{::}\mathit{nat}) < b \Longrightarrow \mathit{lb\_ub}\ a\ b\ f\ g \Longrightarrow \mathit{fishburn}\ a\ b\ f\ g$
quickcheck[*expect=counterexample*]
oops

lemma $a<(b{::}\mathit{int}) \Longrightarrow \mathit{fishburn}\ a\ b\ f\ g \Longrightarrow \mathit{fishburn}\ a\ b\ g\ f$
quickcheck[*expect=counterexample*]
oops

lemma $a<(b{::}\mathit{int}) \Longrightarrow \mathit{knuth}\ a\ b\ f\ g \Longrightarrow \mathit{fishburn}\ a\ b\ f\ g$
quickcheck[*expect=counterexample*]
oops

lemma *fishburn_trans*: $\mathit{fishburn}\ a\ b\ f\ g \Longrightarrow \mathit{fishburn}\ a\ b\ g\ h \Longrightarrow \mathit{fishburn}\ a\ b\ f\ h$
by *auto*

Exactness: if the real value is within the bounds, ab is exact. More interesting would be the other way around. The impact of the exactness lemmas below is unclear.

lemma *fishburn_exact*: $a \leq v \wedge v \leq b \Longrightarrow \mathit{fishburn}\ a\ b\ v\ ab \Longrightarrow ab = v$
by *auto*

Let everyting $= 0$ and $ab = 1$:

lemma *mm_not_exact*: $a \le (v{::}bool) \wedge v \le b \Longrightarrow mm\ a\ v\ b = mm\ a\ ab\ b \Longrightarrow ab = v$
quickcheck[*expect*=*counterexample*]
oops
lemma *knuth_not_exact*: $a \le (v{::}ereal) \wedge v \le b \Longrightarrow knuth\ a\ b\ v\ ab \Longrightarrow ab = v$
quickcheck[*expect*=*counterexample*]
oops
lemma *mm_not_exact*: $a < b \Longrightarrow (a{::}ereal) \le v \wedge v \le b \Longrightarrow mm\ a\ v\ b = mm\ a\ ab\ b \Longrightarrow ab = v$
quickcheck[*expect*=*counterexample*]
oops

2.4 Alpha-Beta for Linear Orders

2.4.1 From the Left

Hard

fun *ab_max* :: $'a \Rightarrow 'a \Rightarrow ('a{::}linorder)tree \Rightarrow 'a$ **and** *ab_maxs ab_min ab_mins* **where**
ab_max a b (Lf x) = x |
ab_max a b (Nd ts) = ab_maxs a b ts |

ab_maxs a b [] = a |
ab_maxs a b (t#ts) = (let a′ = max a (ab_min a b t) in if a′ ≥ b then a′ else ab_maxs a′ b ts) |

ab_min a b (Lf x) = x |
ab_min a b (Nd ts) = ab_mins a b ts |

ab_mins a b [] = b |
ab_mins a b (t#ts) = (let b′ = min b (ab_max a b t) in if b′ ≤ a then b′ else ab_mins a b′ ts)

lemma *ab_maxs_ge_a*: *ab_maxs a b ts ≥ a*
apply(*induction ts arbitrary*: *a*)
by (*auto simp*: *Let_def*)(*use max.bounded_iff* **in** *blast*)

lemma *ab_mins_le_b*: *ab_mins a b ts ≤ b*
apply(*induction ts arbitrary*: *b*)
by (*auto simp*: *Let_def*)(*use min.bounded_iff* **in** *blast*)

Automatic *fishburn* proof:

theorem
shows *a < b ⟹ fishburn a b (maxmin t) (ab_max a b t)*
and *a < b ⟹ fishburn a b (maxmin (Nd ts)) (ab_maxs a b ts)*
and *a < b ⟹ fishburn a b (minmax t) (ab_min a b t)*
and *a < b ⟹ fishburn a b (minmax (Nd ts)) (ab_mins a b ts)*

```
proof(induction a b t and a b ts and a b t and a b ts rule: ab_max_ab_maxs_ab_min_ab_mins.induct)
  case (4 a b t ts)
  then show ?case
    apply(simp add: Let_def)
     by (smt (verit, del_insts) ab_maxs_ge_a le_max_iff_disj linorder_not_le
nle_le)
next
  case (8 a b t ts)
  then show ?case
    apply(simp add: Let_def)
     by (smt (verit, del_insts) ab_mins_le_b linorder_not_le min.bounded_iff
nle_le)
qed auto
```

Detailed *fishburn* proof:

```
theorem fishburn_val_ab:
  shows a < b ⟹ fishburn a b (maxmin t)   (ab_max a b t)
    and a < b ⟹ fishburn a b (maxmin (Nd ts)) (ab_maxs a b ts)
    and a < b ⟹ fishburn a b (minmax t)   (ab_min a b t)
    and a < b ⟹ fishburn a b (minmax (Nd ts)) (ab_mins a b ts)
proof(induction a b t and a b ts and a b t and a b ts rule: ab_max_ab_maxs_ab_min_ab_mins.induct)
  case (4 a b t ts)
  let ?abt = ab_min a b t let ?a' = max a ?abt let ?abts = ab_maxs ?a' b ts
  let ?mmt = minmax t let ?mmts = maxmin (Nd ts)
  let ?abtts = ab_maxs a b (t # ts) let ?mmtts = maxmin (Nd (t # ts))
  note IH1 = 4.IH(1)[OF ‹a<b›]

  have 1: ?mmtts ≤ ?abtts if ab: ?abtts ≤ a
  proof (cases b ≤ ?a')
    case True
    note ‹a<b›
    also note True
    also have ?a' = ?abtts using True by simp
    also note ‹... ≤ a›
    finally have False by simp
    thus ?thesis ..
  next
    case False
   hence IH2: fishburn ?a' b ?mmts ?abts using 4(2)[OF refl] ‹a<b› linorder_not_le
by blast
    from False ab have ab: ?abts ≤ a by(simp)
    have ?a' ≤ ?abts by(rule ab_maxs_ge_a)

     hence ?mmt ≤ ?abt using IH1 ab by (metis order.trans linorder_not_le
max.absorb4)
    have ?abts ≤ ?a' using ab le_max_iff_disj by blast
    have ?a' ≤ a using ‹?a' ≤ ?abts› ab by(rule order.trans)
    hence ?mmts ≤ ?abts using IH2 ‹?abts ≤ ?a'› by blast
    with ‹?mmt ≤ ?abt› show ?thesis using False ‹?a' ≤ ?abts› by(auto)
```

qed

have 2: *?abtts* $\leq$ *?mmtts* **if** *ab*: *b* $\leq$ *?abtts*
proof (*cases b* $\leq$ *?a'*)
case *True*
hence *b* $\leq$ *?abt* **using** ‹*a*<*b*› **by** (*metis linorder_not_le max_less_iff_conj*)
hence *?abt* $\leq$ *?mmt* **using** *IH*1 **by** *blast*
moreover
then have *a* $\leq$ *?mmt* **using** ‹*a*<*b*› ‹*b* $\leq$ *?abt*› **by** (*simp*)
ultimately show *?thesis* **using** *True* **by** (*simp add*: *max.coboundedI*1)
next
case *False*
hence *b* $\leq$ *?abts* **using** *ab* **by** *simp*
hence *?abts* $\leq$ *?mmts* **using** 4.*IH*(2)[*OF refl*] *False* **by** (*meson linorder_not_le*)
then show *?thesis* **using** *False* **by** (*simp add*: *le_max_iff_disj*)
qed

have 3: *?abtts* = *?mmtts* **if** *ab*: *a* < *?abtts ?abtts* < *b*
proof (*cases b* $\leq$ *?a'*)
case *True*
also have *?a'* = *?abtts* **using** *True* **by** *simp*
also note *ab*(2)
finally have *False* **by** *simp*
thus *?thesis* **..**
next
case *False*
hence *IH*2: *fishburn ?a' b ?mmts ?abts* **using** 4(2)[*OF refl*] ‹*a*<*b*› *linorder_not_le* **by** *blast*
have *?abtts* = *?abts* **using** *False* **by**(*simp*)
have *?mmtts* = *max ?mmt ?mmts* **by** *simp*
note *IH*11 = *IH*1[*THEN conjunct*1] **note** *IH*12 = *IH*1[*THEN conjunct*2,*THEN conjunct*1]
note *IH*21 = *IH*2[*THEN conjunct*1] **note** *IH*22 = *IH*2[*THEN conjunct*2,*THEN conjunct*1]
have *arecb*: *a* < *?abts* $\wedge$ *?abts* < *b* **using** *ab False* **by**(*auto*)
have *?abt* $\leq$ *a* $\vee$ *a* < *?abt* **by** *auto*
hence *?abts* = *max ?mmt ?mmts*
proof
assume *?abt* $\leq$ *a*
hence *?a'* = *a* **by** *simp*
hence *?abts* = *?mmts* **using** *IH*22 *arecb* **by** *presburger*
moreover
have *?mmt* $\leq$ *?mmts* **using** *IH*11 ‹*?abt* $\leq$ *a*› *arecb* ‹*?abts* = *?mmts*› **by** *auto*
ultimately show *?thesis* **by**(*simp add*:*Let_def*)
next
assume *a* < *?abt*
have *?abt* < *b* **by** (*meson False linorder_not_le max_less_iff_conj*)
hence *?abt* = *?mmt* **using** *IH*12 ‹*a* < *?abt*› **by** *blast*
have *?a'* < *?abts* $\vee$ *?a'* = *?abts* **using** *ab_maxs_ge_a*[*of ?a' b ts*] *or-*

der_le_less **by** *blast*
      thus *?thesis*
      proof
        assume $?a' < ?abts$
        thus *?thesis* **using** ‹$?abt = ?mmt$› *IH22 arecb* **by**(*simp*)
      next
        assume $?a' = ?abts$
        then show *?thesis* **using** ‹$?abt = ?mmt$› ‹$a < ?abt$› *IH21* **by**(*simp*)
      qed
    qed
    thus *?thesis* **using** ‹$?abtts = ?abts$› ‹$?mmtts = max\ ?mmt\ ?mmts$› **by** *simp*
  qed

  show *?case* **using** 1 2 3 **by** *blast*
next
  case 8 **thus** *?case*
    apply(*simp add*: *Let_def*)
    by (*smt* (*verit*, *del_insts*) *ab_mins_le_b linorder_le_cases linorder_not_le min_le_iff_disj order_antisym*)
qed *auto*

corollary *ab_max_bot_top*: $ab\_max \perp \top\ t = maxmin\ t$
by (*metis bounded_linorder_collapse fishburn_val_ab*(1) *fishburn_bot_top*)

A detailed *knuth* proof, similar to $a < b \Longrightarrow ab\_max\ a\ b\ t \le maxmin\ t$ (mod a,b)
$a < b \Longrightarrow ab\_maxs\ a\ b\ ts \le maxmin\ (Nd\ ts)$ (mod a,b)
$a < b \Longrightarrow ab\_min\ a\ b\ t \le minmax\ t$ (mod a,b)
$a < b \Longrightarrow ab\_mins\ a\ b\ ts \le minmax\ (Nd\ ts)$ (mod a,b) proof:

theorem *knuth_val_ab*:
  shows $a < b \Longrightarrow knuth\ a\ b\ (maxmin\ t)\ \ (ab\_max\ a\ b\ t)$
    and $a < b \Longrightarrow knuth\ a\ b\ (maxmin\ (Nd\ ts))\ (ab\_maxs\ a\ b\ ts)$
    and $a < b \Longrightarrow knuth\ a\ b\ (minmax\ t)\ \ (ab\_min\ a\ b\ t)$
    and $a < b \Longrightarrow knuth\ a\ b\ (minmax\ (Nd\ ts))\ (ab\_mins\ a\ b\ ts)$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_max_ab_maxs_ab_min_ab_mins.induct*)
  case (4 *a b t ts*)
  let $?abt = ab\_min\ a\ b\ t$ **let** $?a' = max\ a\ ?abt$ **let** $?abts = ab\_maxs\ ?a'\ b\ ts$
  let $?mmt = minmax\ t$ **let** $?mmts = maxmin\ (Nd\ ts)$
  note $IH1 = 4.IH(1)[OF$ ‹$a<b$›$]$

  have 1: $maxmin\ (Nd\ (t\ \#\ ts)) \le a$ **if** *ab*: $ab\_maxs\ a\ b\ (t\ \#\ ts) \le a$
  proof (*cases* $b \le ?a'$)
    case *True*
    note ‹$a<b$›
    also note *True*
    also have $?a' = ab\_maxs\ a\ b\ (t\ \#\ ts)$ **using** *True* **by** *simp*
    also note ‹$\ldots \le a$›
    finally have *False* **by** *simp*
    thus *?thesis* **..**

```
  next
    case False
    hence IH2: knuth ?a' b ?mmts ?abts
      using 4(2)[OF refl] ‹a<b› linorder_not_le by blast
    from False ab have ab: ab_maxs ?a' b ts ≤ a by(simp)
    have ?a' ≤ ?abts by(rule ab_maxs_ge_a)

   hence ?mmt ≤ a using IH1 ab by (metis order.trans linorder_not_le max.absorb4)
    have ?abts ≤ ?a' using ab le_max_iff_disj by blast
    from ‹?a' ≤ ?abts› ab have ?a' ≤ a by(rule order.trans)
    hence ?mmts ≤ a using IH2 ‹?abts ≤ ?a'› order.trans by blast
    with ‹?mmt ≤ a› show ?thesis by simp
  qed

  have 2: b ≤ maxmin (Nd (t # ts)) if ab: b ≤ ab_maxs a b (t # ts)
  proof (cases b ≤ ?a')
    case True
    hence b ≤ ?abt using ‹a<b› by (metis linorder_not_le max_less_iff_conj)
    hence b ≤ ?mmt using IH1 by blast
    thus ?thesis by (simp add: le_max_iff_disj)
  next
    case False
    hence b ≤ ?abts using ab by simp
    hence b ≤ ?mmts using 4.IH(2)[OF refl] False by (meson linorder_not_le)
    then show ?thesis by (simp add: le_max_iff_disj)
  qed

  have 3: ab_maxs a b (t # ts) = maxmin (Nd (t # ts))
   if ab: a < ab_maxs a b (t # ts) ab_maxs a b (t # ts) < b
  proof (cases b ≤ ?a')
    case True
    also have ?a' = ab_maxs a b (t # ts) using True by simp
    also note ab(2)
    finally have False by simp
    thus ?thesis ..
  next
    case False
   hence IH2: knuth ?a' b ?mmts ?abts using 4(2)[OF refl] ‹a<b› linorder_not_le
by blast
    have ab_maxs a b (t # ts) = ?abts using False by(simp)
    note IH11 = IH1[THEN conjunct1] note IH12 = IH1[THEN conjunct2,THEN
conjunct1]
    note IH21 = IH2[THEN conjunct1] note IH22 = IH2[THEN conjunct2,THEN
conjunct1]
    have ?abt < b by (meson False linorder_not_le max_less_iff_conj)
    have arecb: a < ?abts ∧ ?abts < b using ab False by(auto)
    have ?abt ≤ a ∨ a < ?abt by auto
    thus ?thesis
    proof
```

```
      assume ?abt ≤ a
      hence ?a' = a by simp
      hence ?abts = ?mmts using IH22 arecb by presburger
      moreover
      have ?mmt ≤ ?mmts using IH11 ‹?abt ≤ a› arecb ‹?abts = ?mmts› by auto
      ultimately show ?thesis using ‹ab_maxs a b (t # ts) = ?abts› by(simp add:Let_def)
    next
      assume a < ?abt
      have ?abt = minmax t using IH12 ‹a < ?abt› ‹?abt < b› by blast
      have ?a' < ?abts ∨ ?a' = ?abts using ab_maxs_ge_a[of ?a' b ts] order_le_less by blast
      thus ?thesis
      proof
        assume ?a' < ?abts
        then show ?thesis using ‹?abt = ?mmt› False IH22 arecb by(simp)
      next
        assume ?a' = ?abts
        then show ?thesis using ‹?abt = ?mmt› False ‹a < ?abt› IH21 by(simp)
      qed
    qed
  qed

  show ?case using 1 2 3 by blast
next
  case 8 thus ?case
    apply(simp add: Let_def)
    by (smt (verit, del_insts) ab_mins_le_b linorder_le_cases linorder_not_le min_le_iff_disj order_antisym)
qed auto
```

Towards exactness:

```
lemma ab_max_le_b: ⟦ a ≤ b; maxmin t ≤ b ⟧ ⟹ ab_max a b t ≤ b
and ⟦ a ≤ b; maxmin (Nd ts) ≤ b ⟧ ⟹ ab_maxs a b ts ≤ b
and ⟦ a ≤ minmax t; a ≤ b ⟧ ⟹ a ≤ ab_min a b t
and ⟦ a ≤ minmax (Nd ts); a ≤ b ⟧ ⟹ a ≤ ab_mins a b ts
proof(induction a b t and a b ts and a b t and a b ts rule: ab_max_ab_maxs_ab_min_ab_mins.induct)
  case (4 a b t ts)
  show ?case
  proof (cases t)
    case Lf
    then show ?thesis using 4.IH(2) 4.prems by(simp add: Let_def)
  next
    case Nd
    then show ?thesis
      apply(simp add: Let_def leI ab_mins_le_b)
      using 4.IH(2) 4.prems ab_mins_le_b by auto
  qed
next
```

```
  case (8 a b t ts)
  show ?case
  proof (cases t)
    case Lf
    then show ?thesis using 8.IH(2) 8.prems by(simp add: Let_def)
  next
    case Nd
    then show ?thesis
      apply(simp add: Let_def leI ab_maxs_ge_a)
      using 8.IH(2) 8.prems ab_maxs_ge_a by auto
  qed
qed auto

lemma ab_max_exact:
assumes v = maxmin t a ≤ v ∧ v ≤ b
shows ab_max a b t = v
proof (cases t)
  case Lf with assms show ?thesis by simp
next
  case Nd
  then show ?thesis using assms
   by (smt (verit) ab_max.simps(2) ab_max_le_b ab_maxs_ge_a order.order_iff_strict
order_le_less_trans fishburn_val_ab(1))
qed
```

Hard, max/min flag

```
fun ab_minmax :: bool ⇒ ('a::linorder) ⇒ 'a ⇒ 'a tree ⇒ 'a and ab_minmaxs
where
ab_minmax mx a b (Lf x) = x |
ab_minmax mx a b (Nd ts) = ab_minmaxs mx a b ts |

ab_minmaxs mx a b [] = a |
ab_minmaxs mx a b (t#ts) =
 (let abt = ab_minmax (¬mx) b a t;
      a' = (if mx then max else min) a abt
  in if (if mx then (≥) else (≤)) a' b then a' else ab_minmaxs mx a' b ts)

lemma ab_max_ab_minmax:
  shows ab_max a b t = ab_minmax True a b t
  and   ab_maxs a b ts = ab_minmaxs True a b ts
  and   ab_min b a t = ab_minmax False a b t
  and   ab_mins b a ts = ab_minmaxs False a b ts
proof(induction a b t and a b ts and b a t and b a ts rule: ab_max_ab_maxs_ab_min_ab_mins.induct)
qed (auto simp add: Let_def)
```

Hard, abstracted over ≤

```
fun ab_le :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ ('a::linorder)tree ⇒ 'a and ab_les
where
```

$ab\_le\ le\ a\ b\ (Lf\ x) = x$ |
$ab\_le\ le\ a\ b\ (Nd\ ts) = ab\_les\ le\ a\ b\ ts$ |

$ab\_les\ le\ a\ b\ [] = a$ |
$ab\_les\ le\ a\ b\ (t\#ts) = (let\ abt = ab\_le\ (\lambda x\ y.\ le\ y\ x)\ b\ a\ t;$
$a' = if\ le\ a\ abt\ then\ abt\ else\ a\ in\ if\ le\ b\ a'\ then\ a'\ else\ ab\_les\ le\ a'\ b\ ts)$

lemma *ab_max_ab_le*:
  shows $ab\_max\ a\ b\ t = ab\_le\ (\leq)\ a\ b\ t$
  and $ab\_maxs\ a\ b\ ts = ab\_les\ (\leq)\ a\ b\ ts$
  and $ab\_min\ b\ a\ t = ab\_le\ (\geq)\ a\ b\ t$
  and $ab\_mins\ b\ a\ ts = ab\_les\ (\geq)\ a\ b\ ts$
proof(*induction a b t* **and** *a b ts* **and** *b a t* **and** *b a ts rule*: *ab_max_ab_maxs_ab_min_ab_mins.induct*)
qed (*auto simp add*: *Let_def*)

Delayed test:

fun *ab_le3* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a::linorder)tree \Rightarrow 'a$ **and** *ab_les3*
where
$ab\_le3\ le\ a\ b\ (Lf\ x) = x$ |
$ab\_le3\ le\ a\ b\ (Nd\ ts) = ab\_les3\ le\ a\ b\ ts$ |

$ab\_les3\ le\ a\ b\ [] = a$ |
$ab\_les3\ le\ a\ b\ (t\#ts) =$
$(if\ le\ b\ a\ then\ a\ else$
$let\ abt = ab\_le3\ (\lambda x\ y.\ le\ y\ x)\ b\ a\ t;$
$a' = if\ le\ a\ abt\ then\ abt\ else\ a$
$in\ ab\_les3\ le\ a'\ b\ ts)$

lemma *ab_max_ab_le3*:
  shows $a < b \Longrightarrow ab\_max\ a\ b\ t = ab\_le3\ (\leq)\ a\ b\ t$
  and $a < b \Longrightarrow ab\_maxs\ a\ b\ ts = ab\_les3\ (\leq)\ a\ b\ ts$
  and $a > b \Longrightarrow ab\_min\ b\ a\ t = ab\_le3\ (\geq)\ a\ b\ t$
  and $a > b \Longrightarrow ab\_mins\ b\ a\ ts = ab\_les3\ (\geq)\ a\ b\ ts$
proof(*induction a b t* **and** *a b ts* **and** *b a t* **and** *b a ts rule*: *ab_max_ab_maxs_ab_min_ab_mins.induct*)
  case (4 *a b t ts*)
  show *?case*
  proof (*cases ts*)
    case *Nil*
    then show *?thesis* **using** 4 **by** (*simp add*: *Let_def*)
  next
    case *Cons*
    then show *?thesis* **using** 4 **by** (*auto simp add*: *Let_def le_max_iff_disj*)
  qed
next
  case (8 *a b t ts*)
  show *?case*
  proof (*cases ts*)
    case *Nil*
    then show *?thesis* **using** 8 **by** (*simp add*: *Let_def*)

next
case *Cons*
then show *?thesis* **using** 8 **by** (*auto simp add*: *Let_def min_le_iff_disj*)
qed
qed *auto*

corollary *ab_le3_bot_top*: $ab\_le3\ (\le)\ \bot\ \top\ t = maxmin\ t$
by (*metis* (*mono_tags*, *lifting*) *ab_max_ab_le3*(1) *ab_max_bot_top bounded_linorder_collapse*)

Hard, *max/min* in *Lf*

Idea due to Bird and Hughes

fun $ab\_max2 :: 'a \Rightarrow 'a \Rightarrow ('a::linorder)tree \Rightarrow 'a$ **and** *ab_maxs2* **and** *ab_min2* **and** *ab_mins2* **where**
$ab\_max2\ a\ b\ (Lf\ x) = max\ a\ (min\ x\ b)$ |
$ab\_max2\ a\ b\ (Nd\ ts) = ab\_maxs2\ a\ b\ ts$ |

$ab\_maxs2\ a\ b\ [] = a$ |
$ab\_maxs2\ a\ b\ (t\#ts) = (let\ a' = ab\_min2\ a\ b\ t\ in\ if\ a' = b\ then\ a'\ else\ ab\_maxs2\ a'\ b\ ts)$ |

$ab\_min2\ a\ b\ (Lf\ x) = max\ a\ (min\ x\ b)$ |
$ab\_min2\ a\ b\ (Nd\ ts) = ab\_mins2\ a\ b\ ts$ |

$ab\_mins2\ a\ b\ [] = b$ |
$ab\_mins2\ a\ b\ (t\#ts) = (let\ b' = ab\_max2\ a\ b\ t\ in\ if\ a = b'\ then\ b'\ else\ ab\_mins2\ a\ b'\ ts)$

lemma *ab_max2_max_min_maxmin*:
shows $a \le b \Longrightarrow ab\_max2\ a\ b\ t = max\ a\ (min\ (maxmin\ t)\ b)$
and $a \le b \Longrightarrow ab\_maxs2\ a\ b\ ts = max\ a\ (min\ (maxmin\ (Nd\ ts))\ b)$
and $a \le b \Longrightarrow ab\_min2\ a\ b\ t = max\ a\ (min\ (minmax\ t)\ b)$
and $a \le b \Longrightarrow ab\_mins2\ a\ b\ ts = max\ a\ (min\ (minmax\ (Nd\ ts))\ b)$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_max2_ab_maxs2_ab_min2_ab_mins2.induct*)
case 4 **thus** *?case* **apply** (*simp add*: *Let_def*)
by (*metis* (*no_types*, *lifting*) *max.assoc max_min_same*(4) *min_max_distrib1*)
next
case 8 **thus** *?case* **apply** (*simp add*: *Let_def*)
by (*metis* (*no_types*, *opaque_lifting*) *max.left_idem max_min_distrib2 max_min_same*(1) *min.assoc min.commute*)
qed *auto*

corollary *ab_max2_bot_top*: $ab\_max2\ \bot\ \top\ t = maxmin\ t$
by(*simp add*: *ab_max2_max_min_maxmin*)

Now for the *ab* version parameterized with *le*:

fun $ab\_le2 :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a::linorder)tree \Rightarrow 'a$ **and** *ab_les2* **where**
$ab\_le2\ le\ a\ b\ (Lf\ x) =$

$(let\ xb = if\ le\ x\ b\ then\ x\ else\ b$
$in\ if\ le\ a\ xb\ then\ xb\ else\ a)$ |
$ab\_le2\ le\ a\ b\ (Nd\ ts) = ab\_les2\ le\ a\ b\ ts$ |

$ab\_les2\ le\ a\ b\ [] = a$ |
$ab\_les2\ le\ a\ b\ (t\#ts) = (let\ a' = ab\_le2\ (\lambda x\ y.\ le\ y\ x)\ b\ a\ t\ in\ if\ a' = b\ then\ a'$
$else\ ab\_les2\ le\ a'\ b\ ts)$

Relate ab_le2 back to ab_max2 (using $a \leq b \Longrightarrow ab\_max2\ a\ b\ t = max$
$a\ (min\ (maxmin\ t)\ b)$

$a \leq b \Longrightarrow ab\_maxs2\ a\ b\ ts = max\ a\ (min\ (maxmin\ (Nd\ ts))\ b)$

$a \leq b \Longrightarrow ab\_min2\ a\ b\ t = max\ a\ (min\ (minmax\ t)\ b)$

$a \leq b \Longrightarrow ab\_mins2\ a\ b\ ts = max\ a\ (min\ (minmax\ (Nd\ ts))\ b)!$):

lemma $ab\_le2\_ab\_max2$:
fixes $a :: \_ :: bounded\_linorder$
shows $a \leq b \Longrightarrow ab\_le2\ (\leq)\ a\ b\ t = ab\_max2\ a\ b\ t$
and $a \leq b \Longrightarrow ab\_les2\ (\leq)\ a\ b\ ts = ab\_maxs2\ a\ b\ ts$
and $a \leq b \Longrightarrow ab\_le2\ (\geq)\ b\ a\ t = ab\_min2\ a\ b\ t$
and $a \leq b \Longrightarrow ab\_les2\ (\geq)\ b\ a\ ts = ab\_mins2\ a\ b\ ts$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: $ab\_max2\_ab\_maxs2\_ab\_min2\_ab\_mins2.induct$)
  case (4 *a b t ts*) **thus** *?case*
    apply (*simp add*: *Let_def*)
    by (*metis* $ab\_max2\_max\_min\_maxmin(3)$ *max.boundedI* $min.cobounded2$)
next
  case 8 **thus** *?case*
    apply (*simp add*: *Let_def*)
    by (*metis* $ab\_max2\_max\_min\_maxmin(1)$ $max.cobounded1$)
qed *auto*

corollary $ab\_le2\_bot\_top$: $ab\_le2\ (\leq)\ \bot\ \top\ t = maxmin\ t$
by (*simp add*: $ab\_le2\_ab\_max2(1)$ $ab\_max2\_bot\_top$)

Hard, Delayed Test

fun $ab\_max3 :: 'a \Rightarrow 'a \Rightarrow ('a::linorder)tree \Rightarrow 'a$ **and** ab_maxs3 **and** ab_min3 **and** ab_mins3 **where**
$ab\_max3\ a\ b\ (Lf\ x) = x$ |
$ab\_max3\ a\ b\ (Nd\ ts) = ab\_maxs3\ a\ b\ ts$ |

$ab\_maxs3\ a\ b\ [] = a$ |
$ab\_maxs3\ a\ b\ (t\#ts) = (if\ a \geq b\ then\ a\ else\ ab\_maxs3\ (max\ a\ (ab\_min3\ a\ b\ t))$
$b\ ts)$ |

$ab\_min3\ a\ b\ (Lf\ x) = x$ |
$ab\_min3\ a\ b\ (Nd\ ts) = ab\_mins3\ a\ b\ ts$ |

$ab\_mins3\ a\ b\ [] = b$ |
$ab\_mins3\ a\ b\ (t\#ts) = (if\ a \geq b\ then\ b\ else\ ab\_mins3\ a\ (min\ b\ (ab\_max3\ a\ b\ t))$
$ts)$

lemma *ab_max3_ab_max*:
shows $a<b \Longrightarrow ab\_max3\ a\ b\ t = ab\_max\ a\ b\ t$
and $a<b \Longrightarrow ab\_maxs3\ a\ b\ ts = ab\_maxs\ a\ b\ ts$
and $a<b \Longrightarrow ab\_min3\ a\ b\ t = ab\_min\ a\ b\ t$
and $a<b \Longrightarrow ab\_mins3\ a\ b\ ts = ab\_mins\ a\ b\ ts$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_max3_ab_maxs3_ab_min3_ab_mins3.induct*)
  case (4 *a b t ts*)
  show *?case*
  proof (*cases ts*)
    case *Nil*
    then show *?thesis* **using** 4 **by** (*simp add*: *Let_def*)
  next
    case *Cons*
    then show *?thesis* **using** 4 **by** (*auto simp add*: *Let_def le_max_iff_disj*)
  qed
next
  case (8 *a b t ts*)
  show *?case*
  proof (*cases ts*)
    case *Nil*
    then show *?thesis* **using** 8 **by** (*simp add*: *Let_def*)
  next
    case *Cons*
    then show *?thesis* **using** 8 **by** (*auto simp add*: *Let_def min_le_iff_disj*)
  qed
qed *auto*

corollary *ab_max3_bot_top*: $ab\_max3\ \bot\ \top\ t = maxmin\ t$
by(*metis fishburn_bot_top ab_max3_ab_max*(1) *fishburn_val_ab*(1) *bounded_linorder_collapse*)

Soft

Fishburn

fun $ab\_max' :: 'a::bounded\_linorder \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$ **and** $ab\_maxs'\ ab\_min'$
ab_mins' **where**
$ab\_max'\ a\ b\ (Lf\ x) = x\ |$
$ab\_max'\ a\ b\ (Nd\ ts) = ab\_maxs'\ a\ b\ \bot\ ts\ |$

$ab\_maxs'\ a\ b\ m\ [] = m\ |$
$ab\_maxs'\ a\ b\ m\ (t\#ts) =$
  $(let\ m' = max\ m\ (ab\_min'\ (max\ m\ a)\ b\ t)\ in\ if\ m' \geq b\ then\ m'\ else\ ab\_maxs'$
$a\ b\ m'\ ts)\ |$

$ab\_min'\ a\ b\ (Lf\ x) = x\ |$
$ab\_min'\ a\ b\ (Nd\ ts) = ab\_mins'\ a\ b\ \top\ ts\ |$

$ab\_mins'\ a\ b\ m\ [] = m\ |$
$ab\_mins'\ a\ b\ m\ (t\#ts) =$

($let\ m' = min\ m\ (ab\_max'\ a\ (min\ m\ b)\ t)\ in\ if\ m' \leq a\ then\ m'\ else\ ab\_mins'\ a$
$b\ m'\ ts$)

lemma $ab\_maxs'\_ge\_a$: $ab\_maxs'\ a\ b\ m\ ts \geq m$
apply(*induction ts arbitrary*: $a\ b\ m$)
by (*auto simp*: *Let_def*)(*use max.bounded_iff* **in** *blast*)

lemma $ab\_mins'\_le\_a$: $ab\_mins'\ a\ b\ m\ ts \leq m$
apply(*induction ts arbitrary*: $a\ b\ m$)
by (*auto simp*: *Let_def*)(*use min.bounded_iff* **in** *blast*)

Find a, b and t such that $a < b$ and fail-soft is closer to the real value than fail-hard.

lemma $let\ a = -\infty;\ b = ereal\ 0;\ t = Nd\ [Nd\ []]$
$in\ a < b \wedge ab\_max\ a\ b\ t = 0 \wedge ab\_max'\ a\ b\ t = \infty \wedge maxmin\ t = \infty$
by *eval*

theorem $fishburn\_val\_ab'$:
shows $a < b \Longrightarrow fishburn\ a\ b\ (maxmin\ t)\ (ab\_max'\ a\ b\ t)$
and $max\ m\ a < b \Longrightarrow fishburn\ (max\ m\ a)\ b\ (maxmin\ (Nd\ ts))\ (ab\_maxs'\ a\ b\ m$
$ts)$
and $a < b \Longrightarrow fishburn\ a\ b\ (minmax\ t)\ (ab\_min'\ a\ b\ t)$
and $a < min\ m\ b \Longrightarrow fishburn\ a\ (min\ m\ b)\ (minmax\ (Nd\ ts))\ (ab\_mins'\ a\ b\ m$
$ts)$
proof(*induction a b t* **and** *a b m ts* **and** *a b t* **and** *a b m ts rule*: *ab_max'_ab_maxs'_ab_min'_ab_mins'.indu*
case (4 $a\ b\ m\ t\ ts$)
then show *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit*, *best*) *ab_maxs'_ge_a max.absorb_iff2 max.coboundedI1 max.commute nle_le nless_le*)
next
case (8 $a\ b\ m\ t\ ts$)
then show *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit*) *ab_mins'_le_a linorder_not_le min.absorb_iff2 min.coboundedI1 min_def*)
qed *auto*

theorem $fishburn\_ab'\_ab$:
shows $a < b \Longrightarrow fishburn\ a\ b\ (ab\_max'\ a\ b\ t)\quad (ab\_max\ a\ b\ t)$
and $max\ m\ a < b \Longrightarrow fishburn\ a\ b\ (ab\_maxs'\ a\ b\ m\ ts)\ (ab\_maxs\ (max\ m\ a)$
$b\ ts)$
and $a < b \Longrightarrow fishburn\ a\ b\ (ab\_min'\ a\ b\ t)\quad (ab\_min\ a\ b\ t)$
and $a < min\ m\ b \Longrightarrow a < m \Longrightarrow fishburn\ a\ b\ (ab\_mins'\ a\ b\ m\ ts)\ (ab\_mins$
$a\ (min\ m\ b)\ ts)$
proof(*induction a b t* **and** *a b m ts* **and** *a b t* **and** *a b m ts rule*: *ab_max'_ab_maxs'_ab_min'_ab_mins'.indu*

case 3 **thus** *?case* **apply** *simp*
by (*metis linorder_not_le max.absorb*4 *max.order_iff*)
next
case (4 *a b m t ts*)
thus *?case* **using** [[*simp_depth_limit*=2]] **apply** (*simp add*: *Let_def*)
by (*smt* (*verit*, *ccfv_threshold*) *linorder_not_le max.absorb*2 *max_less_iff_conj nle_le*)
next
case 6 **thus** *?case*
apply *simp* **using** *top.extremum_strict top.not_eq_extremum* **by** *blast*
next
case 7 **thus** *?case* **apply** *simp*
by (*metis linorder_not_le min.absorb*4 *min.order_iff*)
next
case (8 *a b m t ts*)
thus *?case* **using** [[*simp_depth_limit*=2]] **apply** (*simp add*: *Let_def*)
by (*smt* (*verit*) *linorder_not_le min_def nle_le fishburn*2)
qed *auto*

Fail-soft can be more precise than fail-hard:

lemma *let* a = *ereal* 0; b = 1; t = *Nd* [] *in*
maxmin t = *ab_max′* a b t ∧ *maxmin* t ≠ *ab_max* a b t
by *eval*

lemma *ab_max′_lb_ub*:
shows $a \le b \Longrightarrow$ *lb_ub* a b (*maxmin* t) (*ab_max′* a b t)
and $a \le b \Longrightarrow$ *lb_ub* a b (*max* i (*maxmin* (*Nd ts*))) (*ab_maxs′* a b i *ts*)
and $a \le b \Longrightarrow$ *lb_ub* a b (*minmax* t) (*ab_min′* a b t)
and $a \le b \Longrightarrow$ *lb_ub* a b (*min* i (*minmax* (*Nd ts*))) (*ab_mins′* a b i *ts*)
proof(*induction a b t* **and** *a b i ts* **and** *a b t* **and** *a b i ts rule*: *ab_max′_ab_maxs′_ab_min′_ab_mins′.induct*)
case (4 *a b m t ts*)
then show *?case*
apply(*simp_all add*: *Let_def*)
by (*smt* (*verit*) *max.coboundedI*1 *max.coboundedI*2 *max_def*)
next
case (8 *a b m t ts*)
then show *?case*
apply(*simp_all add*: *Let_def*)
by (*smt* (*verit*, *del_insts*) *min.coboundedI*1 *min.coboundedI*2 *min_def*)
qed *auto*

lemma *ab_max′_exact_less*: ⟦ $a<b$; v = *maxmin* t; $a \le v \wedge v \le b$ ⟧ ⟹ *ab_max′* a b t = v
using *fishburn_val_ab′*(1) **by** *force*

lemma *ab_max′_exact*: ⟦ v = *maxmin* t; $a \le v \wedge v \le b$ ⟧ ⟹ *ab_max′* a b t = v
using *ab_max′_exact_less ab_max′_lb_ub*(1)
by (*metis order.strict_trans*2 *nless_le*)

Searched trees

Hard:

fun *abt_max* :: (*'a::linorder*) ⇒ *'a* ⇒ *'a tree* ⇒ *'a tree* **and** *abt_maxs abt_min*
abt_mins **where**
abt_max a b (*Lf x*) = *Lf x* |
abt_max a b (*Nd ts*) = *Nd* (*abt_maxs a b ts*) |

abt_maxs a b [] = [] |
abt_maxs a b (*t#ts*) = (*let u* = *abt_min a b t*; *a'* = *max a* (*ab_min a b t*) *in*
  u # (*if a'* ≥ *b then* [] *else abt_maxs a' b ts*)) |

abt_min a b (*Lf x*) = *Lf x* |
abt_min a b (*Nd ts*) = *Nd* (*abt_mins a b ts*) |

abt_mins a b [] = [] |
abt_mins a b (*t#ts*) = (*let u* = *abt_max a b t*; *b'* = *min b* (*ab_max a b t*) *in*
  u # (*if b'* ≤ *a then* [] *else abt_mins a b' ts*))

Soft:

fun *abt_max'* :: (*'a::bounded_linorder*) ⇒ *'a* ⇒ *'a tree* ⇒ *'a tree* **and** *abt_maxs'*
abt_min' abt_mins' **where**
abt_max' a b (*Lf x*) = *Lf x* |
abt_max' a b (*Nd ts*) = *Nd* (*abt_maxs' a b* ⊥ *ts*) |

abt_maxs' a b m [] = [] |
abt_maxs' a b m (*t#ts*) =
 (*let u* = *abt_min'* (*max m a*) *b t*; *m'* = *max m* (*ab_min'* (*max m a*) *b t*) *in*
  u # (*if m'* ≥ *b then* [] *else abt_maxs' a b m' ts*)) |

abt_min' a b (*Lf x*) = *Lf x* |
abt_min' a b (*Nd ts*) = *Nd* (*abt_mins' a b* ⊤ *ts*) |

abt_mins' a b m [] = [] |
abt_mins' a b m (*t#ts*) =
 (*let u* = *abt_max' a* (*min m b*) *t*; *m'* = *min m* (*ab_max' a* (*min m b*) *t*) *in*
  u # (*if m'* ≤ *a then* [] *else abt_mins' a b m' ts*))

lemma *abt_max'_abt_max*:
shows *a* < *b* ⟹ *abt_max' a b t* = *abt_max a b t*
and *max m a* < *b* ⟹ *abt_maxs' a b m ts* = *abt_maxs* (*max m a*) *b ts*
and *a* < *b* ⟹ *abt_min' a b t* = *abt_min a b t*
and *a* < *min m b* ⟹ *abt_mins' a b m ts* = *abt_mins a* (*min m b*) *ts*
proof(*induction a b t* **and** *a b m ts* **and** *a b t* **and** *a b m ts rule*: *abt_max'_abt_maxs'_abt_min'_abt_mins'.in*
  case (4 *a b m t ts*)
  thus *?case* **unfolding** *abt_maxs'.simps*(2) *abt_maxs.simps*(2) *Let_def*
    using *fishburn_ab'_ab*(3)
    by (*smt* (*verit*, *best*) *le_max_iff_disj linorder_not_le max_def nless_le*)
next

case (8 *a b m t ts*)
then show *?case* **unfolding** *abt_mins'.simps*(2) *abt_mins.simps*(2) *Let_def*
using *fishburn_ab'_ab*(1)
by (*smt* (*verit*, *del_insts*) *linorder_not_le min.absorb*1 *min.absorb*3 *min.commute min_le_iff_disj*)
qed (*auto*)

An annotated tree of *ab* calls with the *a*,*b* window.

datatype *'a tri* = *Ma 'a 'a 'a tr* | *Mi 'a 'a 'a tr*
and *'a tr* = *No 'a tri list* | *Le 'a*

fun *abtr_max* :: (*'a*::*linorder*) ⇒ *'a* ⇒ *'a tree* ⇒ *'a tri* **and** *abtr_maxs abtr_min abtr_mins* **where**
abtr_max a b (*Lf x*) = *Ma a b* (*Le x*) |
abtr_max a b (*Nd ts*) = *Ma a b* (*No* (*abtr_maxs a b ts*)) |

abtr_maxs a b [] = [] |
abtr_maxs a b (*t#ts*) = (*let u* = *abtr_min a b t*; *a'* = *max a* (*ab_min a b t*) *in*
u # (*if a'* ≥ *b then* [] *else abtr_maxs a' b ts*)) |

abtr_min a b (*Lf x*) = *Mi a b* (*Le x*) |
abtr_min a b (*Nd ts*) = *Mi a b* (*No* (*abtr_mins a b ts*)) |

abtr_mins a b [] = [] |
abtr_mins a b (*t#ts*) = (*let u* = *abtr_max a b t*; *b'* = *min b* (*ab_max a b t*) *in*
u # (*if b'* ≤ *a then* [] *else abtr_mins a b' ts*))

For better readability get rid of *ereal*:

fun *de* :: *ereal* ⇒ *real* **where**
de (*ereal x*) = *x* |
de PInfty = 100 |
de MInfty = −100

fun *detri* **and** *detr* **where**
detri (*Ma a b t*) = *Ma* (*de a*) (*de b*) (*detr t*) |
detri (*Mi a b t*) = *Mi* (*de a*) (*de b*) (*detr t*) |
detr (*No ts*) = *No* (*map detri ts*) |
detr (*Le x*) = *Le* (*de x*)

Example in Knuth and Moore. Evaluation confirms that all subtrees *u* are pruned.

value *let*
*t*11 = *Nd*[*Nd*[*Lf* 3,*Lf* 1,*Lf* 4], *Nd*[*Lf* 1,*t*], *Nd*[*Lf* 2,*Lf* 6,*Lf* 5]];
*t*12 = *Nd*[*Nd*[*Lf* 3,*Lf* 5,*Lf* 8], *u*]; *t*13 = *Nd*[*Nd*[*Lf* 8,*Lf* 4,*Lf* 6], *u*];
*t*21 = *Nd*[*Nd*[*Lf* 3,*Lf* 2],*Nd*[*Lf* 9,*Lf* 5,*Lf* 0],*Nd*[*Lf* 2,*u*]];
*t*31 = *Nd*[*Nd*[*Lf* 0,*u*],*Nd*[*Lf* 4,*Lf* 9,*Lf* 4],*Nd*[*Lf* 4,*u*]];
*t*32 = *Nd*[*Nd*[*Lf* 2,*u*],*Nd*[*Lf* 7,*Lf* 8,*Lf* 1],*Nd*[*Lf* 6,*Lf* 4,*Lf* 0]];
t = *Nd*[*Nd*[*t*11, *t*12, *t*13], *Nd*[*t*21,*u*], *Nd*[*t*31,*t*32,*u*]]
in (*ab_max* (−∞::*ereal*) ∞ *t*,*abt_max* (−∞::*ereal*) ∞ *t*,*detri*(*abtr_max* (−∞::*ereal*) ∞ *t*))

Soft, generalized, attempts

Attempts to prove correct General version due to Junkang Li et al.

This first version (not worth following!) stops the list iteration as soon as $max\ m\ a \geq b$ (I call this "delayed test", it complicates proofs a little.) and the initial value is fixed a (not $i0/1$)

```
fun abil0' :: ('a::bounded_linorder)tree ⇒ 'a ⇒ 'a ⇒ 'a and abils0' abil1' abils1'
where
abil0' (Lf x) a b = x |
abil0' (Nd ts) a b = abils0' ts a b a |

abils0' [] a b m = m |
abils0' (t#ts) a b m =
  (if max m a ≥ b then m else abils0' ts (max m a) b (max m (abil1' t b (max m
a)))) |

abil1' (Lf x) a b = x |
abil1' (Nd ts) a b = abils1' ts a b a |

abils1' [] a b m = m |
abils1' (t#ts) a b m =
  (if min m a ≤ b then m else  abils1' ts (min m a) b (min m (abil0' t b (min m
a))))

lemma abils0'_ge_i: abils0' ts a b i ≥ i
apply(induction ts arbitrary: i a)
by (auto simp: Let_def)(use max.bounded_iff in blast)

lemma abils1'_le_i: abils1' ts a b i ≤ i
apply(induction ts arbitrary: i a)
by (auto simp: Let_def)(use min.bounded_iff in blast)

lemma fishburn_abil01':
  shows a < b ⟹ fishburn a b (maxmin t)           (abil0' t a b)
    and a < b ⟹ i < b ⟹ fishburn (max a i) b (maxmin (Nd ts)) (abils0' ts a
b i)
    and a > b ⟹ fishburn b a (minmax t)           (abil1' t a b)
    and a > b ⟹ i > b ⟹ fishburn b (min a i) (minmax (Nd ts)) (abils1' ts a b
i)
proof(induction t a b and ts a b i and t a b and ts a b i rule: abil0'_abils0'_abil1'_abils1'.induct)
  case (4 t ts a b m)
  thus ?case apply (simp add: Let_def)
    by (smt (verit) abils0'.elims abils0'_ge_i max.absorb_iff1 max.coboundedI2
max_def nless_le)
next
  case (8 t ts a b m)
  thus ?case apply (simp add: Let_def)
    by (smt (verit, best) abils1'.elims abils1'_le_i linorder_not_le min.absorb2
```

min_def min_le_iff_disj)
qed *auto*

This second computes the value of t before deciding if it needs to look at *ts* as well. This simplifies the proof (also in other versions, independently of initialization). The initial value is not fixed but determined by *i*0/1. The "real" constraint on *i*0/1 is commented out and replaced by the simplified value a.

locale *LeftSoft* =
fixes *i*0 *i*1 :: *'a::bounded_linorder tree list* ⇒ *'a* ⇒ *'a*
assumes *i*0: *i*0 *ts a* ≤ *a* — max a (maxmin (Nd ts)) **and** *i*1: *i*1 *ts a* ≥ *a*— min a (minmax (Nd ts))
begin

fun *abil*0*'* :: (*'a::bounded_linorder*)*tree* ⇒ *'a* ⇒ *'a* ⇒ *'a* **and** *abils*0*'* *abil*1*'* *abils*1*'*
where
*abil*0*'* (*Lf x*) *a b* = *x* |
*abil*0*'* (*Nd ts*) *a b* = *abils*0*'* *ts a b* (*i*0 *ts a*) |

*abils*0*'* [] *a b m* = *m* |
*abils*0*'* (*t#ts*) *a b m* =
  (*let m'* = *max m* (*abil*1*'* *t b* (*max m a*)) *in if m'* ≥ *b then m' else abils*0*'* *ts a b m'*) |

*abil*1*'* (*Lf x*) *a b* = *x* |
*abil*1*'* (*Nd ts*) *a b* = *abils*1*'* *ts a b* (*i*1 *ts a*) |

*abils*1*'* [] *a b m* = *m* |
*abils*1*'* (*t#ts*) *a b m* =
  (*let m'* = *min m* (*abil*0*'* *t b* (*min m a*)) *in if m'* ≤ *b then m' else abils*1*'* *ts a b m'*)

lemma *abils*0*'_ge_i*: *abils*0*'* *ts a b i* ≥ *i*
apply(*induction ts arbitrary*: *i*)
by (*auto simp*: *Let_def*)(*use max.bounded_iff* **in** *blast*)

lemma *abils*1*'_le_i*: *abils*1*'* *ts a b i* ≤ *i*
apply(*induction ts arbitrary*: *i*)
by (*auto simp*: *Let_def*)(*use min.bounded_iff* **in** *blast*)

Generalizations that don't seem to work: a) *max a i* → *max* (*max a* (*maxmin* (*Nd ts*))) *i* b) *?*

lemma *fishburn_abil*01*'*:
  shows *a* < *b* ⟹ *fishburn a b* (*maxmin t*)   (*abil*0*'* *t a b*)
   and *a* < *b* ⟹ *i* < *b* ⟹ *fishburn* (*max a i*) *b* (*maxmin* (*Nd ts*)) (*abils*0*'* *ts a b i*)
   and *a* > *b* ⟹ *fishburn b a* (*minmax t*)   (*abil*1*'* *t a b*)
   and *a* > *b* ⟹ *i* > *b* ⟹ *fishburn b* (*min a i*) (*minmax* (*Nd ts*)) (*abils*1*'* *ts a b i*)

proof(*induction t a b* **and** *ts a b i* **and** *t a b* **and** *ts a b i rule*: *abil0′_abils0′_abil1′_abils1′.induct*)
  case (2 *ts a b*)
  thus *?case* **using** *i0*[*of ts a*] **by** *auto*
next
  case (4 *i t ts a b*)
  thus *?case* **apply** (*simp add*: *Let_def*)
    by (*smt* (*verit*) *abils0′_ge_i le_max_iff_disj linorder_not_le max.absorb1*
nle_le)
next
  case (6 *ts a b*)
  thus *?case* **using** *i1*[*of a ts*] **by** *simp*
next
  case (8 *i t ts a b*)
  thus *?case* **apply** (*simp add*: *Let_def*)
   by (*smt* (*verit*, *best*) *abils1′_le_i linorder_not_le min_def min_less_iff_conj*
nle_le)
qed *auto*

Note the $a \leq b$ instead of the $a < b$ in theorem *fishburn_abir01′*:

lemma *abil0′lb_ub*:
shows $a \leq b \Longrightarrow$ *lb_ub a b* (*maxmin t*) (*abil0′ t a b*)
and $a \leq b \Longrightarrow$ *lb_ub a b* (*max i* (*maxmin* (*Nd ts*))) (*abils0′ ts a b i*)
and $a \geq b \Longrightarrow$ *lb_ub b a* (*minmax t*) (*abil1′ t a b*)
and $a \geq b \Longrightarrow$ *lb_ub b a* (*min i* (*minmax* (*Nd ts*))) (*abils1′ ts a b i*)
proof(*induction t a b* **and** *ts a b i* **and** *t a b* **and** *ts a b i rule*: *abil0′_abils0′_abil1′_abils1′.induct*)
  case (2 *ts a b*)
  then show *?case* **by** *simp* (*meson order.trans i0 le_max_iff_disj*)
next
  case (4 *t ts a b m*)
  then show *?case*
   apply(*simp add*: *Let_def*)
   by (*smt* (*verit*, *best*) *max.coboundedI1 max.coboundedI2 max_def*)
next
  case (6 *ts a b*)
  then show *?case*
   by *simp* (*meson i1 min.coboundedI2 order_trans*)
next
  case (8 *t ts a b m*)
  then show *?case*
   apply(*simp add*: *Let_def*)
   by (*smt* (*verit*, *del_insts*) *min.coboundedI1 min.coboundedI2 min_def*)
qed *auto*

lemma *abil0′_exact_less*: $\llbracket$ $a<b$; $v =$ *maxmin t*; $a \leq v \wedge v \leq b$ $\rrbracket \Longrightarrow$ *abil0′ t a b*
$= v$
using *fishburn_abil01′*(1) **by** *force*

lemma *abil0′_exact*: $\llbracket$ $v =$ *maxmin t*; $a \leq v \wedge v \leq b\rrbracket \Longrightarrow$ *abil0′ t a b* $= v$
by (*metis abil0′_exact_less abil0′lb_ub*(1) *order.trans leD order_le_imp_less_or_eq*)

end

Transposition Table / Graph / Repeated AB

lemma *ab_twice_lb*:
$[\![$ $\forall a\ b.$ *fishburn a b* (*maxmin t*) (*abf a b t*); $b \le$ *abf a b t*; *max* a' (*abf a b t*) $< b'$
$]\!] \Longrightarrow$
fishburn a' b' (*maxmin t*) (*abf* (*max* a' (*abf a b t*)) b' *t*)
by (*smt* (*verit*, *del_insts*) *order.eq_iff order.strict_trans leI max_less_iff_conj*)

lemma *ab_twice_ub*:
$[\![$ $\forall a\ b.$ *fishburn a b* (*maxmin t*) (*abf a b t*); *abf a b t* $\le a$; *min* b' (*abf a b t*) $> a'$
$]\!] \Longrightarrow$
fishburn a' b' (*maxmin t*) (*abf* a' (*min* b' (*abf a b t*)) *t*)
by (*smt* (*verit*, *best*) *linorder_not_le min.absorb1 min.absorb2 min.strict_boundedE nless_le*)

But what does a narrower window achieve? Less precise bounds but prefix of search space. For fail-hard and fail-soft.

fun *prefix prefixs* **where**
prefix (*Lf x*) (*Lf y*) = (*x=y*) |
prefix (*Nd ts*) (*Nd us*) = *prefixs ts us* |
prefix _ _ = *False* |

prefixs [] *us* = *True* |
prefixs (*t#ts*) (*u#us*) = (*prefix t u* ∧ *prefixs ts us*) |
prefixs _ _ = *False*

lemma *fishburn_ab_max_windows*:
shows $[\![$ $a < b$; $a' \le a$; $b \le b'$ $]\!] \Longrightarrow$ *fishburn a b* (*ab_max* a' b' *t*) (*ab_max a b t*)
and $[\![$ $a < b$; $a' \le a$; $b \le b'$ $]\!] \Longrightarrow$ *fishburn a b* (*ab_maxs* a' b' *ts*) (*ab_maxs a b ts*)
and $[\![$ $a < b$; $a' \le a$; $b \le b'$ $]\!] \Longrightarrow$ *fishburn a b* (*ab_min* a' b' *t*) (*ab_min a b t*)
and $[\![$ $a < b$; $a' \le a$; $b \le b'$ $]\!] \Longrightarrow$ *fishburn a b* (*ab_mins* a' b' *ts*) (*ab_mins a b ts*)
proof (*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts*
arbitrary: a' b' **and** a' b' **and** a' b' **and** a' b'
rule: *ab_max_ab_maxs_ab_min_ab_mins.induct*)
case 2 **show** *?case*
using 2.*prems* **apply** *simp*
using 2.*IH* **by** *presburger*
next
case (4 *a b t ts*) **show** *?case* **using** 4.*prems*
apply (*simp add*: *Let_def*)
using 4.*IH*
by (*smt* (*verit*, *del_insts*) *ab_maxs_ge_a le_max_iff_disj linorder_not_le max_def nle_le*)
next

case 6 **show** *?case*
using 6.*prems* **apply** *simp*
using 6.*IH* **by** *presburger*
next
case (8 *a b t ts*) **show** *?case* **using** 8.*prems*
apply (*simp add*: *Let_def*)
using 8.*IH*
by (*smt* (*verit*, *best*) *ab_mins_le_b order.strict_trans*1 *linorder_not_le min.absorb*1 *min.absorb*2 *nle_le*)
qed *auto*

lemma *abt_max_prefix_windows*:
shows ⟦ $a' \le a$; $b \le b'$ ⟧ ⟹ *prefix* (*abt_max a b t*) (*abt_max a' b' t*)
and ⟦ $a' \le a$; $b \le b'$ ⟧ ⟹ *prefixs* (*abt_maxs a b ts*) (*abt_maxs a' b' ts*)
and ⟦ $a' \le a$; $b \le b'$ ⟧ ⟹ *prefix* (*abt_min a b t*) (*abt_min a' b' t*)
and ⟦ $a' \le a$; $b \le b'$ ⟧ ⟹ *prefixs* (*abt_mins a b ts*) (*abt_mins a' b' ts*)
proof (*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts*
arbitrary: *a' b'* **and** *a' b'* **and** *a' b'* **and** *a' b'*
rule: *abt_max_abt_maxs_abt_min_abt_mins.induct*)
case (4 *a b t ts*)
then show *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit*, *del_insts*) *fishburn_ab_max_windows*(3) *linorder_not_le max.coboundedI*1 *max.orderI max_def*)
next
case (8 *a b t ts*)
then show *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit*, *del_insts*) *fishburn_ab_max_windows*(1) *le_cases*3 *linorder_not_less min_def_raw knuth_if_fishburn*)
qed *auto*

lemma *fishburn_ab_max'_windows*:
shows ⟦ $a < b$; $a' \le a$; $b \le b'$ ⟧ ⟹ *fishburn a b* (*ab_max' a' b' t*) (*ab_max' a b t*)
and ⟦ $max\ m\ a < b$; $a' \le a$; $b \le b'$; $m' \le m$ ⟧ ⟹ *fishburn* (*max m a*) *b* (*ab_maxs' a' b' m' ts*) (*ab_maxs' a b m ts*)
and ⟦ $a < b$; $a' \le a$; $b \le b'$ ⟧ ⟹ *fishburn a b* (*ab_min' a' b' t*) (*ab_min' a b t*)
and ⟦ $a < min\ m\ b$; $a' \le a$; $b \le b'$; $m \le m'$ ⟧ ⟹ *fishburn a* (*min m b*) (*ab_mins' a' b' m' ts*) (*ab_mins' a b m ts*)
proof (*induction a b t* **and** *a b m ts* **and** *a b t* **and** *a b m ts*
arbitrary: *a' b'* **and** *a' b' m'* **and** *a' b'* **and** *a' b' m'*
rule: *ab_max'_ab_maxs'_ab_min'_ab_mins'.induct*)
case (2 *a b ts*)
show *?case*
using 2.*prems* **apply** *simp*
using 2.*IH* **by** (*metis max_bot nle_le*)
next
case (4 *a b m t ts*)

show *?case*
using 4.*prems* **apply** (*simp add*: *Let_def*)
using 4.*IH ab_maxs'_ge_a*
by (*smt* (*verit*) *knuth_comm linorder_le_cases max.absorb*1 *max.absorb*2 *max.absorb*4 *max.bounded_iff*)
next
case (6 *a b ts*)
show *?case*
using 6.*prems* **apply** *simp*
using 6.*IH* **by** (*metis min_top nle_le*)
next
case (8 *a b m t ts*)
show *?case*
using 8.*prems* **apply** (*simp add*: *Let_def*)
using 8.*IH ab_mins'_le_a*
by (*smt* (*verit*) *leD linorder_linear min.absorb*1 *min.absorb*2 *min.bounded_iff not_le_imp_less*)
qed *auto*

Example of reduced search space:

lemma *let* $a = 0$; $b = (1::ereal)$; $a' = 0$; $b' = 2$; $t = Nd$ [*Lf* 1, *Lf* 0]
in abt_max' a b t = *Nd* [*Lf* 1] ∧ *abt_max' a' b' t* = *t*
by *eval*

lemma *abt_max'_prefix_windows*:
shows ⟦ $a < b$; $a' \le a$; $b \le b'$ ⟧ ⟹ *prefix* (*abt_max' a b t*) (*abt_max' a' b' t*)
and ⟦ *max m a* < *b*; $a' \le a$; $b \le b'$; $m' \le m$ ⟧ ⟹ *prefixs* (*abt_maxs' a b m ts*) (*abt_maxs' a' b' m' ts*)
and ⟦ $a < b$; $a' \le a$; $b \le b'$ ⟧ ⟹ *prefix* (*abt_min' a b t*) (*abt_min' a' b' t*)
and ⟦ *a* < *min m b*; $a' \le a$; $b \le b'$; $m \le m'$ ⟧ ⟹ *prefixs* (*abt_mins' a b m ts*) (*abt_mins' a' b' m' ts*)
proof (*induction a b t* **and** *a b m ts* **and** *a b t* **and** *a b m ts*
arbitrary: *a' b'* **and** *a' b' m'* **and** *a' b'* **and** *a' b' m'*
rule: *abt_max'_abt_maxs'_abt_min'_abt_mins'.induct*)
case (4 *a b m t ts*)
then show *?case* **apply** (*simp add*: *Let_def*)
using *fishburn_ab_max'_windows*(3)
by (*smt* (*verit*, *ccfv_threshold*) *add_mono linorder_le_cases max.absorb*2 *max.assoc max.order_iff order.strict_iff_not trans_le_add*1)
next
case (8 *a b t ts*)
then show *?case* **apply** (*simp add*: *Let_def*)
using *fishburn_ab_max'_windows*(1)
by (*smt* (*verit*, *best*)*add_le_mono le_add*1 *linorder_not_less min.mono min_less_iff_conj order.trans nle_le*)
qed *auto*

2.4.2 From the Right

The literature uniformly considers iteration from the left only. Iteration from the right is technically simpler but needs to go through all successors, which means generating all of them. This is typically done anyway to reorder them based on heuristic evaluations. This rules out an infinite list of successors, but it is unclear if there are any applications.

Naming convention: 0 = max, 1 = min

Hard

fun *abr0* :: (*'a::linorder*)*tree* ⇒ *'a* ⇒ *'a* ⇒ *'a* **and** *abrs0* **and** *abr1* **and** *abrs1*
where
abr0 (*Lf x*) *a b* = *x* |
abr0 (*Nd ts*) *a b* = *abrs0 ts a b* |

abrs0 [] *a b* = *a* |
abrs0 (*t#ts*) *a b* = (*let m* = *abrs0 ts a b in if m* ≥ *b then m else max* (*abr1 t b m*) *m*) |

abr1 (*Lf x*) *a b* = *x* |
abr1 (*Nd ts*) *a b* = *abrs1 ts a b* |

abrs1 [] *a b* = *a* |
abrs1 (*t#ts*) *a b* = (*let m* = *abrs1 ts a b in if m* ≤ *b then m else min* (*abr0 t b m*) *m*)

lemma *abrs0_ge_a*: *abrs0 ts a b* ≥ *a*
apply(*induction ts arbitrary*: *a*)
by (*auto simp*: *Let_def*)(*use max.coboundedI2* **in** *blast*)

lemma *abrs1_le_a*: *abrs1 ts a b* ≤ *a*
apply(*induction ts arbitrary*: *a*)
by (*auto simp*: *Let_def*)(*use min.coboundedI2* **in** *blast*)

theorem *abr01_mm*:
  shows *mm a* (*abr0 t a b*) *b* = *mm a* (*maxmin t*) *b*
    and *mm a* (*abrs0 ts a b*) *b* = *mm a* (*maxmin* (*Nd ts*)) *b*
    and *mm b* (*abr1 t a b*) *a* = *mm b* (*minmax t*) *a*
    and *mm b* (*abrs1 ts a b*) *a* = *mm b* (*minmax* (*Nd ts*)) *a*
proof(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abr0_abrs0_abr1_abrs1.induct*)
  case (4 *t ts a b*)
  then show *?case*
    apply(*simp add*: *Let_def*)
     by (*smt* (*verit*) *max.left_commute max_def_raw min.commute min.orderE min_max_distrib1*)
next
  case (8 *t ts a b*)

then show *?case*
apply(*simp add*: *Let_def*)
by (*smt* (*verit*) *max.left_idem max.orderE max_min_distrib*2 *max_min_same*(4) *min.assoc*)
qed *auto*

As a corollary:

corollary *knuth_abr01_cor*: $a < b \Longrightarrow$ *knuth a b* (*maxmin t*) (*abr*0 *t a b*)
by (*meson knuth_if_mm abr*01_*mm*(1))

corollary *maxmin_mm_abr*0: ⟦ $a \le$ *maxmin t*; *maxmin t* $\le b$ ⟧ $\Longrightarrow$ *maxmin t* = *mm a* (*abr*0 *t a b*) *b*
by (*metis max_def min.absorb*1 *abr*01_*mm*(1))
corollary *maxmin_mm_abrs*0: ⟦ $a \le$ *maxmin* (*Nd ts*); *maxmin* (*Nd ts*) $\le b$ ⟧ $\Longrightarrow$ *maxmin* (*Nd ts*) = *mm a* (*abrs*0 *ts a b*) *b*
by (*metis max_def min.absorb*1 *abr*01_*mm*(2))

The stronger *fishburn* spec:

Needs $a<b$.

theorem *fishburn_abr*01:
shows $a < b \Longrightarrow$ *fishburn a b* (*maxmin t*) (*abr*0 *t a b*)
and $a < b \Longrightarrow$ *fishburn a b* (*maxmin* (*Nd ts*)) (*abrs*0 *ts a b*)
and $a > b \Longrightarrow$ *fishburn b a* (*minmax t*) (*abr*1 *t a b*)
and $a > b \Longrightarrow$ *fishburn b a* (*minmax* (*Nd ts*)) (*abrs*1 *ts a b*)
proof(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abr*0_*abrs*0_*abr*1_*abrs*1.*induct*)
case (4 *t ts a b*)
then show *?case*
apply(*simp add*: *Let_def*)
by (*smt* (*verit*, *best*) *leI max.absorb_iff*2 *max.coboundedI*1 *max.orderE order.strict_iff_not*)
next
case (8 *t ts a b*)
then show *?case*
apply(*simp add*: *Let_def*)
by (*smt* (*verit*, *best*) *order.trans linorder_not_le min.absorb*3 *min.absorb_iff*2 *nle_le*)
qed *auto*

Above lemma does not work for $a = b$ and $a > b$. Not fishburn: *abr*0 $\le a$ but not *maxmin* $\le$ *abr*0. Not knuth: *abr*0 $\le a$ but not *maxmin* $\le a$

lemma *let* $a = 0$::*ereal*; $t =$ *Nd* [*Lf* 1, *Lf* 0] *in abr*0 *t a a* $= 0 \wedge$ *maxmin t* $= 1$
by *eval*
lemma *let* $a = 0$::*ereal*; $b = -1$; $t =$ *Nd* [*Lf* 1, *Lf* 0] *in abr*0 *t a b* $= 0 \wedge$ *maxmin t* $= 1$
by *eval*

The following lemma does not follow from *fishburn* because of the weaker assumption $a \le b$ that is required for the later exactness lemma.

lemma *abr*0_*le_b*: ⟦ $a \le b$; *maxmin t* $\le b$ ⟧ $\Longrightarrow$ *abr*0 *t a b* $\le b$

and $[\![\ a \leq b;\ maxmin\ (Nd\ ts) \leq b\ ]\!] \Longrightarrow abrs0\ ts\ a\ b \leq b$
and $[\![\ b \leq minmax\ t;\ b \leq a\ ]\!] \Longrightarrow b \leq abr1\ t\ a\ b$
and $[\![\ b \leq minmax\ (Nd\ ts);\ b \leq a\ ]\!] \Longrightarrow b \leq abrs1\ ts\ a\ b$
proof(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abr0_abrs0_abr1_abrs1.induct*)
  case (4 *t ts a b*)
  show *?case*
  proof (*cases t*)
    case *Lf*
    then show *?thesis* **using** 4.*IH*(1) 4.*prems* **by**(*simp add*: *Let_def*)
  next
    case *Nd*
    then show *?thesis*
      apply(*simp add*: *Let_def leI abrs1_le_a*)
      using 4.*IH*(1) 4.*prems* **by** *auto*
  qed
next
  case (8 *t ts a b*)
  show *?case*
  proof (*cases t*)
    case *Lf*
    then show *?thesis* **using** 8.*IH*(1) 8.*prems* **by**(*simp add*: *Let_def*)
  next
    case *Nd*
    then show *?thesis*
      apply(*simp add*: *Let_def leI abrs0_ge_a*)
      using 8.*IH*(1) 8.*prems* **by** *auto*
  qed
qed *auto*

lemma *abr0_exact_less*:
assumes $a < b$ $v = maxmin\ t$ $a \leq v \wedge v \leq b$
shows $abr0\ t\ a\ b = v$
using *fishburn_exact*[*OF assms*(3)] *fishburn_abr01*[*OF assms*(1)] *assms*(2) **by** *metis*

lemma *abr0_exact*:
assumes $v = maxmin\ t$ $a \leq v \wedge v \leq b$
shows $abr0\ t\ a\ b = v$
using *abr0_exact_less abr0_le_b abrs0_ge_a assms*(1,2) *abr0.elims*
by (*smt* (*verit*, *best*) *dual_order.trans leD maxmin.simps*(1) *order_le_imp_less_or_eq*)

Another proof:

lemma *abr0_exact2*:
assumes $v = maxmin\ t$ $a \leq v \wedge v \leq b$
shows $abr0\ t\ a\ b = v$
proof (*cases t*)
  case *Lf* **with** *assms* **show** *?thesis* **by** *simp*
next
  case *Nd*
  then show *?thesis* **using** *assms*

by $(smt\ (verit,\ del\_insts)\ abr0.simps(2)\ abr0\_le\_b\ abrs0\_ge\_a\ order.order\_iff\_strict\ order\_le\_less\_trans\ fishburn\_abr01(1))$

qed

Soft

Starting at $\perp$ (after Fishburn)

fun $abr0' :: ('a::bounded\_linorder)tree \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **and** $abrs0'$ **and** $abr1'$ **and** $abrs1'$ **where**
$abr0'\ (Lf\ x)\ a\ b = x\ |$
$abr0'\ (Nd\ ts)\ a\ b = abrs0'\ ts\ a\ b\ |$

$abrs0'\ []\ a\ b = \perp\ |$
$abrs0'\ (t\#ts)\ a\ b = (let\ m = abrs0'\ ts\ a\ b\ in\ if\ m \geq b\ then\ m\ else\ max\ (abr1'\ t\ b\ (max\ m\ a))\ m)\ |$

$abr1'\ (Lf\ x)\ a\ b = x\ |$
$abr1'\ (Nd\ ts)\ a\ b = abrs1'\ ts\ a\ b\ |$

$abrs1'\ []\ a\ b = \top\ |$
$abrs1'\ (t\#ts)\ a\ b = (let\ m = abrs1'\ ts\ a\ b\ in\ if\ m \leq b\ then\ m\ else\ min\ (abr0'\ t\ b\ (min\ m\ a))\ m)$

theorem $fishburn\_abr01'$:
  shows $a < b \Longrightarrow fishburn\ a\ b\ (maxmin\ t)\quad (abr0'\ t\ a\ b)$
    and $a < b \Longrightarrow fishburn\ a\ b\ (maxmin\ (Nd\ ts))\ (abrs0'\ ts\ a\ b)$
    and $a > b \Longrightarrow fishburn\ b\ a\ (minmax\ t)\quad (abr1'\ t\ a\ b)$
    and $a > b \Longrightarrow fishburn\ b\ a\ (minmax\ (Nd\ ts))\ (abrs1'\ ts\ a\ b)$
proof$(induction\ t\ a\ b$ **and** $ts\ a\ b$ **and** $t\ a\ b$ **and** $ts\ a\ b\ rule\colon abr0'\_abrs0'\_abr1'\_abrs1'.induct)$
  case $(4\ t\ ts\ a\ b)$
  then show $?case$
    apply$(simp\ add\colon Let\_def)$
   by $(smt\ (verit,\ ccfv\_SIG)\ le\_max\_iff\_disj\ linorder\_not\_le\ max.absorb3\ max.absorb\_iff2)$
next
  case $(8\ t\ ts\ a\ b)$
  then show $?case$
    apply$(simp\ add\colon Let\_def)$
    by $(smt\ (verit,\ best)\ linorder\_not\_le\ min.absorb\_iff1\ min.absorb\_iff2\ nle\_le\ fishburn2)$
qed $auto$

Same as for $abr0$: Above lemma does not work for $a = b$ and $a > b$. Not fishburn: $abr0' \leq a$ but not $maxmin \leq abr0'$. Not knuth: $abr0' \leq a$ but not $maxmin \leq a$

lemma $let\ a = 0::ereal;\ t = Nd\ [Lf\ 1,\ Lf\ 0]\ in\ abr0'\ t\ a\ a = 0 \wedge maxmin\ t = 1$
by $eval$
lemma $let\ a = 0::ereal;\ b = -1;\ t = Nd\ [Lf\ 1,\ Lf\ 0]\ in\ abr0'\ t\ a\ b = 0 \wedge maxmin\ t = 1$

by *eval*

Fails for a=b=-1 and t = Nd []

theorem *fishburn2_abr01_abr01'*:
shows $a < b \Longrightarrow$ *fishburn a b* (*abr0' t a b*) (*abr0 t a b*)
and $a < b \Longrightarrow$ *fishburn a b* (*abrs0' ts a b*) (*abrs0 ts a b*)
and $a > b \Longrightarrow$ *fishburn b a* (*abr1' t a b*) (*abr1 t a b*)
and $a > b \Longrightarrow$ *fishburn b a* (*abrs1' ts a b*) (*abrs1 ts a b*)
proof(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abr0_abrs0_abr1_abrs1.induct*)
case (4 *t ts a b*)
thus *?case* **apply** (*simp add*: *Let_def*)
by (*smt* (*verit*) *abrs0_ge_a max.absorb2 max.assoc max.commute nle_le order_le_imp_less_or_eq*)
next
case (8 *t ts a b*)
thus *?case* **apply** (*simp add*: *Let_def*)
by (*smt* (*verit*, *ccfv_threshold*) *abrs1_le_a min.absorb_iff2 min.bounded_iff nle_le order.strict_iff_order*)
qed *auto*

Towards 'exactness':

No need for restricting *a*,*b*, but only corollaries:

corollary *abr0'_mm*: *mm a* (*abr0' t a b*) *b* = *mm a* (*maxmin t*) *b*
by (*smt* (*verit*) *max.absorb1 max.cobounded2 max.strict_order_iff min.absorb2 min.absorb3 min.cobounded1 mm_if_knuth fishburn_abr01'*(1))
corollary *abrs0'_mm*: *mm a* (*abrs0' ts a b*) *b* = *mm a* (*maxmin* (*Nd ts*)) *b*
by (*metis abr0'.simps*(2) *abr0'_mm*)
corollary *abr1'_mm*: *mm b* (*abr1' t a b*) *a* = *mm b* (*minmax t*) *a*
by (*smt* (*verit*, *best*) *linorder_not_le max.absorb1 max_less_iff_conj min.absorb2 fishburn_abr01'*(3))
corollary *abrs1'_mm*: *mm b* (*abrs1' ts a b*) *a* = *mm b* (*minmax* (*Nd ts*)) *a*
by (*metis abr1'.simps*(2) *abr1'_mm*)

corollary *li1*: $[\![a \le$ *maxmin t*; *maxmin t* $\le b]\!] \Longrightarrow$ *mm a* (*abr0' t a b*) *b* = *maxmin t*
by (*simp add*: *abr0'_mm*)

Note the $a \le b$ instead of the $a < b$ in $a < b \Longrightarrow$ *abr0' t a b* $\le$ *maxmin t* (mod *a*,*b*)

$a < b \Longrightarrow$ *abrs0' ts a b* $\le$ *maxmin* (*Nd ts*) (mod *a*,*b*)

$b < a \Longrightarrow$ *abr1' t a b* $\le$ *minmax t* (mod *b*,*a*)

$b < a \Longrightarrow$ *abrs1' ts a b* $\le$ *minmax* (*Nd ts*) (mod *b*,*a*):

lemma *abr01'lb_ub*:
shows $a \le b \Longrightarrow$ *lb_ub a b* (*maxmin t*) (*abr0' t a b*)
and $a \le b \Longrightarrow$ *lb_ub a b* (*maxmin* (*Nd ts*)) (*abrs0' ts a b*)
and $a \ge b \Longrightarrow$ *lb_ub b a* (*minmax t*) (*abr1' t a b*)
and $a \ge b \Longrightarrow$ *lb_ub b a* (*minmax* (*Nd ts*)) (*abrs1' ts a b*)
apply(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abr0'_abrs0'_abr1'_abrs1'.induct*)

by(*auto simp add*: *Let_def le_max_iff_disj min_le_iff_disj*)

lemma *abr0'_exact_less*: $[\![$ $a<b$; v = *maxmin* t; $a \le v \wedge v \le b$ $]\!] \Longrightarrow$ *abr0'* t a b $= v$
using *fishburn_abr01'*(1) **by** *force*

lemma *abr0'_exact*: $[\![$ v = *maxmin* t; $a \le v \wedge v \le b$ $]\!] \Longrightarrow$ *abr0'* t a b $= v$
by (*metis abr0'_exact_less abr01'lb_ub*(1) *order.trans leD order_le_imp_less_or_eq*)

Also returning the searched tree

Hard:

fun *abtr0* :: ('*a*::*linorder*) *tree* ⇒ '*a* ⇒ '*a* ⇒ '*a* ∗ '*a tree* **and** *abtrs0* **and** *abtr1* **and** *abtrs1* **where**
abtr0 (*Lf x*) *a b* = (*x*, *Lf x*) |
abtr0 (*Nd ts*) *a b* = (*let* (*m*,*us*) = *abtrs0 ts a b in* (*m*, *Nd us*)) |

abtrs0 [] *a b* = (*a*,[]) |
abtrs0 (*t*#*ts*) *a b* = (*let* (*m*,*us*) = *abtrs0 ts a b in*
  if m ≥ *b then* (*m*,*us*) *else let* (*n*,*u*) = *abtr1 t b m in* (*max n m*,*u*#*us*)) |

abtr1 (*Lf x*) *a b* = (*x*, *Lf x*) |
abtr1 (*Nd ts*) *a b* = (*let* (*m*,*us*) = *abtrs1 ts a b in* (*m*, *Nd us*)) |

abtrs1 [] *a b* = (*a*,[]) |
abtrs1 (*t*#*ts*) *a b* = (*let* (*m*,*us*) = *abtrs1 ts a b in*
  if m ≤ *b then* (*m*,*us*) *else let* (*n*,*u*) = *abtr0 t b m in* (*min n m*,*u*#*us*))

Soft:

fun *abtr0'* :: ('*a*::*bounded_linorder*) *tree* ⇒ '*a* ⇒ '*a* ⇒ '*a* ∗ '*a tree* **and** *abtrs0'* **and** *abtr1'* **and** *abtrs1'* **where**
abtr0' (*Lf x*) *a b* = (*x*, *Lf x*) |
abtr0' (*Nd ts*) *a b* = (*let* (*m*,*us*) = *abtrs0' ts a b in* (*m*, *Nd us*)) |

abtrs0' [] *a b* = (⊥,[]) |
abtrs0' (*t*#*ts*) *a b* = (*let* (*m*,*us*) = *abtrs0' ts a b in*
  if m ≥ *b then* (*m*,*us*) *else let* (*n*,*u*) = *abtr1' t b* (*max m a*) *in* (*max n m*,*u*#*us*)) |

abtr1' (*Lf x*) *a b* = (*x*, *Lf x*) |
abtr1' (*Nd ts*) *a b* = (*let* (*m*,*us*) = *abtrs1' ts a b in* (*m*, *Nd us*)) |

abtrs1' [] *a b* = (⊤,[]) |
abtrs1' (*t*#*ts*) *a b* = (*let* (*m*,*us*) = *abtrs1' ts a b in*
  if m ≤ *b then* (*m*,*us*) *else let* (*n*,*u*) = *abtr0' t b* (*min m a*) *in* (*min n m*,*u*#*us*))

lemma *fst_abtr01*:
shows *fst*(*abtr0 t a b*) = *abr0 t a b*
and *fst*(*abtrs0 ts a b*) = *abrs0 ts a b*
and *fst*(*abtr1 t a b*) = *abr1 t a b*

and *fst*(*abtrs*1 *ts a b*) = *abrs*1 *ts a b*
by(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abtr*0_*abtrs*0_*abtr*1_*abtrs*1.*induct*)
(*auto simp*: *Let_def split*: *prod.split*)

lemma *fst_abtr*01′:
shows *fst*(*abtr*0′ *t a b*) = *abr*0′ *t a b*
and *fst*(*abtrs*0′ *ts a b*) = *abrs*0′ *ts a b*
and *fst*(*abtr*1′ *t a b*) = *abr*1′ *t a b*
and *fst*(*abtrs*1′ *ts a b*) = *abrs*1′ *ts a b*
by(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abtr*0′_*abtrs*0′_*abtr*1′_*abtrs*1′.*induct*)
(*auto simp*: *Let_def split*: *prod.split*)

lemma *snd_abtr*01′_*abtr*01:
shows $a < b \Longrightarrow$ *snd*(*abtr*0′ *t a b*) = *snd*(*abtr*0 *t a b*)
and $a < b \Longrightarrow$ *snd*(*abtrs*0′ *ts a b*) = *snd*(*abtrs*0 *ts a b*)
and $a > b \Longrightarrow$ *snd*(*abtr*1′ *t a b*) = *snd*(*abtr*1 *t a b*)
and $a > b \Longrightarrow$ *snd*(*abtrs*1′ *ts a b*) = *snd*(*abtrs*1 *ts a b*)
proof(*induction t a b* **and** *ts a b* **and** *t a b* **and** *ts a b rule*: *abtr*0′_*abtrs*0′_*abtr*1′_*abtrs*1′.*induct*)
case (4 *t ts a b*)
then show *?case*
apply(*simp add*: *Let_def split*: *prod.split*)
using *fst_abtr*01(2) *fst_abtr*01′(2) *fishburn*2_*abr*01_*abr*01′(2) *abrs*0_*ge_a*
by (*smt* (*verit*, *best*) *fst_conv le_max_iff_disj linorder_not_le max.absorb*1
nle_le sndI)
next
case (8 *t ts a b*)
then show *?case*
apply(*simp add*: *Let_def split*: *prod.split*)
using *fst_abtr*01(4) *fst_abtr*01′(4) *fishburn*2_*abr*01_*abr*01′(4) *abrs*1_*le_a*
by (*smt* (*verit*, *ccfv_threshold*) *fst_conv linorder_not_le min.absorb*1 *min.absorb_iff*2
*order.strict_trans*2 *snd_conv*)
qed (*auto simp add*: *split_beta*)

Generalized

General version due to Junkang Li et al.:

locale *SoftGeneral* =
fixes *i*0 *i*1 :: ′*a*::*bounded_linorder tree list* ⇒ ′*a* ⇒ ′*a*
assumes *i*0: *i*0 *ts a* ≤ *max a* (*maxmin*(*Nd ts*)) **and** *i*1: *i*1 *ts a* ≥ *min a* (*minmax*
(*Nd ts*))
begin

fun *abir*0′ :: (′*a*::*bounded_linorder*)*tree* ⇒ ′*a* ⇒ ′*a* ⇒ ′*a* **and** *abirs*0′ **and** *abir*1′
and *abirs*1′ **where**
*abir*0′ (*Lf x*) *a b* = *x* |
*abir*0′ (*Nd ts*) *a b* = *abirs*0′ (*i*0 *ts a*) *ts a b* |

*abirs*0′ *i* [] *a b* = *i* |
*abirs*0′ *i* (*t*#*ts*) *a b* =

$(let\ m = abirs0'\ i\ ts\ a\ b\ in\ if\ m \geq b\ then\ m\ else\ max\ (abir1'\ t\ b\ (max\ m\ a))\ m)$ |

$abir1'\ (Lf\ x)\ a\ b = x$ |
$abir1'\ (Nd\ ts)\ a\ b = abirs1'\ (i1\ ts\ a)\ ts\ a\ b$ |

$abirs1'\ i\ []\ a\ b = i$ |
$abirs1'\ i\ (t\#ts)\ a\ b =$
$(let\ m = abirs1'\ i\ ts\ a\ b\ in\ if\ m \leq b\ then\ m\ else\ min\ (abir0'\ t\ b\ (min\ m\ a))\ m)$

Unused:

lemma *abirs0'_ge_i*: $abirs0'\ i\ ts\ a\ b \geq i$
apply(*induction ts*)
by (*auto simp*: *Let_def*) (*metis max.coboundedI2*)

lemma *abirs1'_le_i*: $abirs1'\ i\ ts\ a\ b \leq i$
apply(*induction ts*)
by (*auto simp*: *Let_def*) (*metis min.coboundedI2*)

lemma *fishburn_abir01'*:
shows $a < b \Longrightarrow fishburn\ a\ b\ (maxmin\ t)\ (abir0'\ t\ a\ b)$
and $a < b \Longrightarrow fishburn\ a\ b\ (max\ i\ (maxmin\ (Nd\ ts)))\ (abirs0'\ i\ ts\ a\ b)$
and $a > b \Longrightarrow fishburn\ b\ a\ (minmax\ t)\ (abir1'\ t\ a\ b)$
and $a > b \Longrightarrow fishburn\ b\ a\ (min\ i\ (minmax\ (Nd\ ts)))\ (abirs1'\ i\ ts\ a\ b)$
proof(*induction t a b* **and** *i ts a b* **and** *t a b* **and** *i ts a b rule*: *abir0'_abirs0'_abir1'_abirs1'.induct*)
case (2 *ts a b*)
thus *?case* **using** *i0*[*of ts a*] **apply** *simp*
by (*smt* (*verit*, *best*) *leD le_max_iff_disj max_def*)
next
case (4 *i t ts a b*)
thus *?case* **apply** (*simp add*: *Let_def*)
by (*smt* (*verit*, *ccfv_SIG*) *linorder_not_le max.coboundedI2 max_def nle_le*)
next
case (6 *ts a b*)
thus *?case*
using *i1*[*of a ts*] **apply** *simp* **by** (*smt* (*verit*, *del_insts*) *leD min_le_iff_disj nle_le*)
next
case (8 *i t ts a b*)
thus *?case* **apply** (*simp add*: *Let_def*)
by (*smt* (*verit*, *ccfv_SIG*) *linorder_not_le min.absorb2 min_le_iff_disj nle_le*)
qed *auto*

Note the $a \leq b$ instead of the $a < b$ in $a < b \Longrightarrow abir0'\ t\ a\ b \leq maxmin\ t\ (\mathrm{mod}\ a,b)$

$a < b \Longrightarrow abirs0'\ i\ ts\ a\ b \leq max\ i\ (maxmin\ (Nd\ ts))\ (\mathrm{mod}\ a,b)$
$b < a \Longrightarrow abir1'\ t\ a\ b \leq minmax\ t\ (\mathrm{mod}\ b,a)$
$b < a \Longrightarrow abirs1'\ i\ ts\ a\ b \leq min\ i\ (minmax\ (Nd\ ts))\ (\mathrm{mod}\ b,a)$:

lemma *abir0'lb_ub*:
shows $a \leq b \Longrightarrow lb\_ub\ a\ b\ (maxmin\ t)\ (abir0'\ t\ a\ b)$

and $a \leq b \Longrightarrow lb\_ub\ a\ b\ (max\ i\ (maxmin\ (Nd\ ts)))\ (abirs0'\ i\ ts\ a\ b)$
and $a \geq b \Longrightarrow lb\_ub\ b\ a\ (minmax\ t)\ (abir1'\ t\ a\ b)$
and $a \geq b \Longrightarrow lb\_ub\ b\ a\ (min\ i\ (minmax\ (Nd\ ts)))\ (abirs1'\ i\ ts\ a\ b)$
by(*induction t a b* **and** *i ts a b* **and** *t a b* **and** *i ts a b rule*: *abir0'_abirs0'_abir1'_abirs1'.induct*)
(*auto simp add*: *Let_def le_max_iff_disj min_le_iff_disj*
intro: *order_trans*[*OF i0*] *order_trans*[*OF _ i1*])

lemma *abir0'_exact_less*: $\llbracket\ a<b;\ v = maxmin\ t;\ a \leq v \wedge v \leq b\ \rrbracket \Longrightarrow abir0'\ t\ a\ b = v$
using *fishburn_abir01'*(1) **by** *force*

lemma *abir0'_exact*: $\llbracket\ v = maxmin\ t;\ a \leq v \wedge v \leq b\rrbracket \Longrightarrow abir0'\ t\ a\ b = v$
by (*metis abir0'_exact_less abir0'lb_ub*(1) *order.trans leD order_le_imp_less_or_eq*)

end

Now with explicit parameters *i0* and *i1* such that we can vary them:

fun *abir0'* :: _ ⇒ _ ⇒ (*'a::bounded_linorder*)*tree* ⇒ *'a* ⇒ *'a* ⇒ *'a* **and** *abirs0'* **and** *abir1'* **and** *abirs1'* **where**
abir0' i0 i1 (*Lf x*) *a b* = *x* |
abir0' i0 i1 (*Nd ts*) *a b* = *abirs0' i0 i1* (*i0 ts a*) *ts a b* |

abirs0' i0 i1 i [] *a b* = *i* |
abirs0' i0 i1 i (*t#ts*) *a b* =
(*let m* = *abirs0' i0 i1 i ts a b in if m* ≥ *b then m else max* (*abir1' i0 i1 t b* (*max m a*)) *m*) |

abir1' i0 i1 (*Lf x*) *a b* = *x* |
abir1' i0 i1 (*Nd ts*) *a b* = *abirs1' i0 i1* (*i1 ts a*) *ts a b* |

abirs1' i0 i1 i [] *a b* = *i* |
abirs1' i0 i1 i (*t#ts*) *a b* =
(*let m* = *abirs1' i0 i1 i ts a b in if m* ≤ *b then m else min* (*abir0' i0 i1 t b* (*min m a*)) *m*)

First, the same theorem as in the locale *SoftGeneral*:

definition *bnd i0 i1* ≡
$\forall\, ts\ a.\ i0\ ts\ a \leq max\ a\ (maxmin(Nd\ ts)) \wedge i1\ ts\ a \geq min\ a\ (minmax\ (Nd\ ts))$

declare [[*unify_search_bound*=400,*unify_trace_bound*=400]]

lemma *fishburn_abir01'*:
shows $a < b \Longrightarrow bnd\ i0\ i1 \Longrightarrow fishburn\ a\ b\ (maxmin\ t)\ \ (abir0'\ i0\ i1\ t\ a\ b)$
and $a < b \Longrightarrow bnd\ i0\ i1 \Longrightarrow fishburn\ a\ b\ (max\ i\ (maxmin\ (Nd\ ts)))\ (abirs0'\ i0\ i1\ i\ ts\ a\ b)$
and $a > b \Longrightarrow bnd\ i0\ i1 \Longrightarrow fishburn\ b\ a\ (minmax\ t)\ \ (abir1'\ i0\ i1\ t\ a\ b)$
and $a > b \Longrightarrow bnd\ i0\ i1 \Longrightarrow fishburn\ b\ a\ (min\ i\ (minmax\ (Nd\ ts)))\ (abirs1'\ i0\ i1\ i\ ts\ a\ b)$

proof(*induction* $i0$ $i1$ t a b **and** $i0$ $i1$ i ts a b **and** $i0$ $i1$ t a b **and** $i0$ $i1$ i ts a b
    rule: *abir0′_abirs0′_abir1′_abirs1′.induct*)
  case (2 *ts a b*)
  thus *?case* **unfolding** *bnd_def* **apply** *simp*
    by (*smt* (*verit*, *best*) *leD le_max_iff_disj max_def*)
next
  case (4 *i t ts a b*)
  thus *?case* **apply** (*simp add*: *Let_def*)
    by (*smt* (*verit*, *ccfv_SIG*) *linorder_not_le max.coboundedI2 max_def nle_le*)
next
  case (6 *ts a b*)
  thus *?case*
    unfolding *bnd_def* **apply** *simp*
   by (*smt* (*verit*, *ccfv_threshold*) *linorder_not_le min.absorb2 min_def min_le_iff_disj*)
next
  case (8 *i t ts a b*)
  thus *?case* **apply** (*simp add*: *Let_def*)
   by (*smt* (*verit*, *ccfv_SIG*) *linorder_not_le min.absorb2 min_le_iff_disj nle_le*)
qed (*auto*)

Unused:

lemma *abirs0′_ge_i*: $abirs0'\ i0\ i1\ i\ ts\ a\ b \geq i$
by(*induction ts*) (*auto simp*: *Let_def max.coboundedI2*)

lemma *abirs0′_eq_i*: $i \geq b \Longrightarrow abirs0'\ i0\ i1\ i\ ts\ a\ b = i$
by(*induction ts*) (*auto simp*: *Let_def*)

lemma *abirs1′_le_i*: $abirs1'\ i0\ i1\ i\ ts\ a\ b \leq i$
by(*induction ts*) (*auto simp*: *Let_def min.coboundedI2*)

Monotonicity wrt the init functions, below/above a:

definition $bnd\_mono\ i0\ i1\ i0'\ i1' =$
$(\forall\, ts\ a.\ i0'\ ts\ a \leq a \wedge i1'\ ts\ a \geq a \wedge i0\ ts\ a \leq i0'\ ts\ a \wedge i1\ ts\ a \geq i1'\ ts\ a)$

lemma *fishburn_abir0′_mono*:
shows $a < b \Longrightarrow bnd\_mono\ i0\ i1\ i0'\ i1' \Longrightarrow fishburn\ a\ b\ (abir0'\ i0\ i1\ t\ a\ b)\ (abir0'\ i0'\ i1'\ t\ a\ b)$
  and $a < b \Longrightarrow bnd\_mono\ i0\ i1\ i0'\ i1' \Longrightarrow i = i0\ (ts0\ @\ ts)\ a \Longrightarrow$
    $fishburn\ a\ b\ (abirs0'\ i0\ i1\ i\ ts\ a\ b)\ (abirs0'\ i0'\ i1'\ (i0'\ (ts0\ @\ ts)\ a)\ ts\ a\ b)$
 and $a > b \Longrightarrow bnd\_mono\ i0\ i1\ i0'\ i1' \Longrightarrow fishburn\ b\ a\ (abir1'\ i0\ i1\ t\ a\ b)\ (abir1'\ i0'\ i1'\ t\ a\ b)$
 and $a > b \Longrightarrow bnd\_mono\ i0\ i1\ i0'\ i1' \Longrightarrow i = i1\ (ts0@ts)\ a \Longrightarrow$
    $fishburn\ b\ a\ (abirs1'\ i0\ i1\ i\ ts\ a\ b)\ (abirs1'\ i0'\ i1'\ (i1'\ (ts0\ @\ ts)\ a)\ ts\ a\ b)$
proof(*induction* $i0$ $i1$ t a b **and** $i0$ $i1$ i ts a b **and** $i0$ $i1$ t a b **and** $i0$ $i1$ i ts a b
  arbitrary: $i0'$ $i1'$ **and** $i0'$ $i1'$ $ts0$ **and** $i0'$ $i1'$ **and** $i0'$ $i1'$ $ts0$
  rule: *abir0′_abirs0′_abir1′_abirs1′.induct*)
  case 1
  then show *?case* **by** *simp*
next

case 2
then show *?case* **by** (*metis abir0'.simps*(2) *append_Nil*)
next
case 3
then show *?case* **unfolding** *bnd_mono_def*
apply *simp* **by** (*metis order.strict_trans1 leD*)
next
case (4 *i0 i1 i t ts a b*)
show *?case*
using 4.*prems* 4.*IH*(2)[*OF refl*, *of i0' i1'*] 4.*IH*(1)[*OF* ‹*a*<*b*›, *of i0' i1' ts0* @ [*t*]]
by (*smt* (*verit*) *abirs0'.simps*(2) *append_Cons append_eq_append_conv2 linorder_not_less max.absorb4*
max.absorb_iff2 max.coboundedI2 max.order_iff self_append_conv)
next
case 5
then show *?case* **by** *simp*
next
case 6
then show *?case* **by** (*metis abir1'.simps*(2) *append_Nil*)
next
case 7
then show *?case* **unfolding** *bnd_mono_def*
apply *simp* **by** (*metis leD order_le_less_trans*)
next
case (8 *i0 i1 i t ts a b*)
then show *?case*
using 8.*prems* 8.*IH*(2)[*OF refl*, *of i0' i1'*] 8.*IH*(1)[*OF* ‹*a*>*b*›, *of i0' i1' ts0* @ [*t*]]
by (*smt* (*verit*) *Cons_eq_appendI abirs1'.simps*(2) *append_eq_append_conv2 linorder_le_less_linear min.absorb2 min.absorb3 min.order_iff min_less_iff_conj self_append_conv*)
qed

The *i0* bound of *a* cannot be increased to *max a* (*maxmin*(*Nd ts*) (as the theorem *fishburn_abir0'* might suggest). Problem: if $b \le i0\ a\ ts < i0'\ a\ ts$ then it can happen that $b \le abirs0'\ i0\ i1\ t\ a\ b < abirs0'\ i0'\ i1'\ t\ a\ b$, which violates *fishburn*.

value *let* $a = -\infty$; $b = 0$::*ereal*; $t = Nd$ [*Lf* (1::*ereal*)] *in*
(*abir0'* (λ*ts a*. *max a* (*maxmin*(*Nd ts*))) *i1' t a b*,
abir0' (λ*ts a*. *max a* (*maxmin*(*Nd ts*))−1) *i1 t a b*)

lemma *let* $a = -\infty$; $b = 0$::*ereal*; *ts* = [*Lf* (1::*ereal*)] *in*
abirs0' (λ*ts a*. *max a* (*maxmin*(*Nd ts*))−1) (λ_ *a*. *a*+1) (*max a* (*maxmin*(*Nd ts*))−1) *ts a b* = 0
unfolding *Let_def*
using [[*simp_trace*]] **by** (*simp add*:*Let_def*)

2.5 Alpha-Beta for De Morgan Orders

2.5.1 From the Left, Fail-Hard

Like Knuth.

fun *ab_negmax* :: $'a \Rightarrow 'a \Rightarrow ('a::de\_morgan\_order)tree \Rightarrow 'a$ **and** *ab_negmaxs*
where
$ab\_negmax\ a\ b\ (Lf\ x) = x\ |$
$ab\_negmax\ a\ b\ (Nd\ ts) = ab\_negmaxs\ a\ b\ ts\ |$

$ab\_negmaxs\ a\ b\ [] = a\ |$
$ab\_negmaxs\ a\ b\ (t\#ts) = (let\ a' = max\ a\ (-\ ab\_negmax\ (-b)\ (-a)\ t)\ in\ if\ a' \geq b\ then\ a'\ else\ ab\_negmaxs\ a'\ b\ ts)$

Via *foldl*. Wasteful: *foldl* consumes whole list.

definition *ab_negmaxf* :: $('a::de\_morgan\_order) \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$ **where**
$ab\_negmaxf\ b = (\lambda a\ t.\ if\ a \geq b\ then\ a\ else\ max\ a\ (-\ ab\_negmax\ (-b)\ (-a)\ t))$

lemma *foldl_ab_negmaxf_idemp*:
$b \leq a \Longrightarrow foldl\ (ab\_negmaxf\ b)\ a\ ts = a$
by(*induction ts*) (*auto simp*: *ab_negmaxf_def*)

lemma *ab_negmaxs_foldl*:
$(a::'a::de\_morgan\_order) < b \Longrightarrow ab\_negmaxs\ a\ b\ ts = foldl\ (ab\_negmaxf\ b)\ a\ ts$
using *foldl_ab_negmaxf_idemp*[**where** *'a='a*]
by(*induction ts arbitrary*: *a*) (*auto simp*: *ab_negmaxf_def Let_def dest*: *not_le_imp_less*)

Also returning the searched tree.

fun *abtl* :: $'a \Rightarrow 'a \Rightarrow ('a::de\_morgan\_order)tree \Rightarrow 'a * ('a::de\_morgan\_order)tree$
and *abtls* **where**
$abtl\ a\ b\ (Lf\ x) = (x,\ Lf\ x)\ |$
$abtl\ a\ b\ (Nd\ ts) = (let\ (m,us) = abtls\ a\ b\ ts\ in\ (m,\ Nd\ us))\ |$

$abtls\ a\ b\ [] = (a,[])\ |$
$abtls\ a\ b\ (t\#ts) = (let\ (a',u) = abtl\ (-b)\ (-a)\ t;\ a' = max\ a\ (-a')\ in$
$if\ a' \geq b\ then\ (a',[u])\ else\ let\ (n,us) = abtls\ a'\ b\ ts\ in\ (n,u\#us))$

lemma *fst_abtl*:
shows $fst(abtl\ a\ b\ t) = ab\_negmax\ a\ b\ t$
and $fst(abtls\ a\ b\ ts) = ab\_negmaxs\ a\ b\ ts$
by(*induction a b t* **and** *a b ts rule*: *abtl_abtls.induct*)
(*auto simp*: *Let_def split*: *prod.split*)

Correctness Proofs

First, a very direct proof.

lemma *ab_negmaxs_ge_a*: $ab\_negmaxs\ a\ b\ ts \geq a$
apply(*induction ts arbitrary*: *a*)

by (*auto simp*: *Let_def*) (*metis max.bounded_iff*)

lemma *fishburn_val_ab_neg*:
shows $a < b \Longrightarrow$ *fishburn a b* (*negmax t*) (*ab_negmax* (*a*) *b t*)
and $a < b \Longrightarrow$ *fishburn a b* (*negmax* (*Nd ts*)) (*ab_negmaxs* (*a*) *b ts*)
proof(*induction a b t* **and** *a b ts rule*: *ab_negmax_ab_negmaxs.induct*)
  case (4 *a b t ts*)
  then show *?case*
    apply(*simp add*: *Let_def less_uminus_reorder*)
   by (*smt* (*verit*, *ccfv_threshold*) *ab_negmaxs_ge_a le_max_iff_disj linorder_not_le minus_less_minus nle_le uminus_less_reorder*)
qed *auto*

Now an indirect one by reduction to the min/max alpha-beta. Direct proof is simpler!

Relate ordinary and negmax ab:

theorem *ab_max_negmax*:
shows *ab_max a b t* = *ab_negmax a b* (*negate False t*)
and *ab_maxs a b ts* = *ab_negmaxs a b* (*map* (*negate True*) *ts*)
and *ab_min a b t* = − *ab_negmax* (−*b*) (−*a*) (*negate True t*)
and *ab_mins a b ts* = − *ab_negmaxs* (−*b*) (−*a*) (*map* (*negate False*) *ts*)
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_max_ab_maxs_ab_min_ab_mins.induct*)
  case 8
  then show *?case* **by**(*simp add*: *Let_def de_morgan_max de_morgan_min uminus_le_reorder*)
qed (*simp_all add*: *Let_def*)

corollary *fishburn_negmax_ab_negmax*: $a < b \Longrightarrow$ *fishburn a b* (*negmax t*) (*ab_negmax a b t*)
using *fishburn_val_ab*(1) *ab_max_negmax*(1) *negmax_maxmin*(1) *negate_negate*
by (*metis* (*no_types*, *lifting*))

lemma *ab_negmax_ab_le*:
  shows *ab_negmax a b t* = *ab_le* (≤) *a b* (*negate False t*)
  and *ab_negmaxs a b ts* = *ab_les* (≤) *a b* (*map* (*negate True*) *ts*)
  and *ab_negmax a b t* = − *ab_le* (≥) (−*a*) (−*b*) (*negate True t*)
  and *ab_negmaxs a b ts* = − *ab_les* (≥) (−*a*) (−*b*) (*map* (*negate False*) *ts*)
by(*induction a b t* **and** *a b ts* **and** *b a t* **and** *b a ts rule*: *ab_max_ab_maxs_ab_min_ab_mins.induct*)
  (*auto simp add*: *Let_def max_def ab_max_ab_le*[*symmetric*] *ab_max_negmax negate_negate o_def*)

Pointless? Weaker than fishburn and direct proof rather than corollary as via *ab_max_negmax*

Weaker max-min property. Proof: Case False one eqn chain, but dualized IH:

theorem

shows *ab_negmax_negmax2*: *max a* (*min* (*ab_negmax a b t*) *b*) = *max a* (*min* (*negmax t*) *b*)
and *ab_negmaxs_maxs_neg3*: *a* < *b* ⟹ *min* (*ab_negmaxs a b ts*) *b* = *max a* (*min* (*negmax* (*Nd ts*)) *b*)
proof(*induction a b t* **and** *a b ts rule*: *ab_negmax_ab_negmaxs.induct*)
  case 2
  thus *?case* **apply** *simp*
    by (*metis leI max_absorb1 max_def min.coboundedI2*)
next
  case (4 *a b t ts*)
  let *?abt* = *ab_negmax* (− *b*) (− *a*) *t* **let** *?a′* = *max a* (− *?abt*)
  let *?T* = *negmax t* **let** *?S* = *negmax* (*Nd ts*)
  show *?case*

  proof (*cases b* ≤ *?a′*)
    case *True*
    have *min b* (*max* (−*?abt*) *a*) = *min b* (*max* (− *?T*) *a*) **using** 4.*IH*(1) 4.*prems*
      by (*metis* (*no_types*) *neg_neg de_morgan_min*)
    hence *b* = *min b* (*max* (− *?T*) *a*) **using** *True*
      by (*metis max.commute min.orderE*)
    hence *b* ≤ *max* (− *?T*) *a*
      by (*metis min.cobounded2*)
    hence *b*: *b* ≤ − *?T*
      by (*meson* 4.*prems leD le_max_iff_disj*)
    have *min* (*ab_negmaxs a b* (*t* # *ts*)) *b* = *min ?a′ b*
      using *True* **by** *simp*
    also have ... = *b*
      using *True min.absorb2* **by** *blast*
    also have ... = *max a* (*max* (*min* (− *?T*) *b*) (*min ?S b*))
      using *b* 4.*prems* **by** *simp*
    also have ... = *max a* (*min* (*max* (− *?T*) *?S*) *b*)
      by (*metis min_max_distrib1*)
    also have ... = *max a* (*min* (*negmax* (*Nd* (*t* # *ts*))) *b*)
      by *simp*
    finally show *?thesis* **.**
  next
    case *False*
    hence 1: − *?abt* < *b*
      by (*metis le_max_iff_disj linorder_not_le*)
    have *IH1*: *max a* (*min* (− *?abt*) *b*) = *max a* (*min* (− *?T*) *b*)
      using 4.*IH*(1) ‹*a*<*b*› **by** (*metis max_min_neg neg_neg*)
    have *min* (*ab_negmaxs a b* (*t* # *ts*)) *b* = *min* (*ab_negmaxs* (*max a* (− *?abt*)) *b ts*) *b*
      using *False* **by**(*simp*)
    also have ... = *max* (*max a* (− *?abt*)) (*min ?S b*)
      using 4.*IH*(2) 4.*prems* 1 *False* **by**(*simp*)
    also have ... = *max* (*max a* (*min* (−*?abt*) *b*)) (*max a* (*min ?S b*))
      using 1 **by** (*simp add*: *max.assoc max.left_commute*)
    also have ... = *max* (*max a* (*min* (− *?T*) *b*)) (*max a* (*min ?S b*))

using *IH*1 **by** *presburger*
also have ... = *max a* (*min* (*max* (− *?T*) *?S*) *b*)
by (*metis* (*no_types*, *lifting*) *max.assoc max.commute max.right_idem min_max_distrib*1)
also have ... = *max a* (*min* (*negmax* (*Nd* (*t* # *ts*))) *b*) **by** *simp*
finally show *?thesis* **.**
qed
qed *auto*

corollary *ab_negmax_negmax_cor*2: *ab_negmax* ⊥ ⊤ *t* = *negmax t*
using *ab_negmax_negmax*2[*of* ⊥ ⊤ *t*] **by** (*simp*)

2.5.2 From the Left, Fail-Soft

After Fishburn

fun *ab_negmax'* :: *'a* ⇒ *'a* ⇒ (*'a*::*de_morgan_order*)*tree* ⇒ *'a* **and** *ab_negmaxs'*
where
ab_negmax' a b (*Lf x*) = *x* |
ab_negmax' a b (*Nd ts*) = (*ab_negmaxs' a b* ⊥ *ts*) |

ab_negmaxs' a b m [] = *m* |
ab_negmaxs' a b m (*t*#*ts*) = (*let m'* = *max m* (− *ab_negmax'* (−*b*) (− *max m a*) *t*) *in*
if m' ≥ *b then m' else ab_negmaxs' a b m' ts*)

lemma *ab_negmaxs'_ge_a*: *ab_negmaxs' a b m ts* ≥ *m*
apply(*induction ts arbitrary*: *a b m*)
by (*auto simp*: *Let_def*) (*metis max.bounded_iff*)

theorem *fishburn_val_ab_neg'*:
shows *a* < *b* ⟹ *fishburn a b* (*negmax t*) (*ab_negmax' a b t*)
and *max a m* < *b* ⟹ *fishburn* (*max a m*) *b* (*negmax* (*Nd ts*)) (*ab_negmaxs' a b m ts*)
proof(*induction a b t* **and** *a b m ts rule*: *ab_negmax'_ab_negmaxs'.induct*)
case (4 *a b m t ts*)
then show *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit*, *del_insts*) *ab_negmaxs'_ge_a minus_le_minus uminus_le_reorder linorder_not_le max.absorb*1 *max.absorb*4 *max.coboundedI*2 *max.commute*)
qed *auto*

theorem *fishburn_ab'_ab_neg*:
shows *a* < *b* ⟹ *fishburn a b* (*ab_negmax' a b t*) (*ab_negmax a b t*)
and *max m a* < *b* ⟹ *fishburn a b* (*ab_negmaxs' a b m ts*) (*ab_negmaxs* (*max m a*) *b ts*)
proof(*induction a b t* **and** *a b m ts rule*: *ab_negmax'_ab_negmaxs'.induct*)
case 1

then show *?case* **by** *auto*
next
case 2
then show *?case*
by *fastforce*
next
case 3
then show *?case* **apply** *simp*
by (*metis linorder_linear linorder_not_le max.commute max.orderE*)
next
case (4 *a b m t ts*)
then show *?case*
apply (*simp*)
by (*smt* (*verit*) *minus_le_minus neg_neg leD linorder_le_less_linear linorder_linear max.absorb_iff1 max.assoc max.commute nle_le*)
qed

Another proof of *fishburn_negmax_ab_negmax*, just by transitivity:

corollary $a < b \Longrightarrow$ *fishburn a b* (*negmax t*) (*ab_negmax a b t*)
by(*rule trans_fishburn*[*OF fishburn_val_ab_neg'*(1) *fishburn_ab'_ab_neg*(1)])

Now fail-soft with traversed trees.

fun *abtl'* :: *'a* ⇒ *'a* ⇒ (*'a*::*de_morgan_order*)*tree* ⇒ *'a* ∗ (*'a*::*de_morgan_order*)*tree*
and *abtls'* **where**
abtl' a b (*Lf x*) = (*x*, *Lf x*) |
abtl' a b (*Nd ts*) = (*let* (*m*,*us*) = *abtls' a b* ⊥ *ts in* (*m*, *Nd us*)) |

abtls' a b m [] = (*m*,[]) |
abtls' a b m (*t*#*ts*) = (*let* (*m'*,*u*) = *abtl'* (−*b*) (− *max m a*) *t*; *m'* = *max m* (−*m'*) *in*
if m' ≥ *b then* (*m'*,[*u*]) *else let* (*n*,*us*) = *abtls' a b m' ts in* (*n*,*u*#*us*))

lemma *fst_abtl'*:
shows *fst*(*abtl' a b t*) = *ab_negmax' a b t*
and *fst*(*abtls' a b m ts*) = *ab_negmaxs' a b m ts*
by(*induction a b t* **and** *a b m ts rule*: *abtl'_abtls'.induct*)
(*auto simp*: *Let_def split*: *prod.split*)

Fail-hard and fail-soft search the same part of the tree:

lemma *snd_abtl'_abtl*:
shows $a < b \Longrightarrow$ *abtl' a b t* = (*ab_negmax' a b t*, *snd*(*abtl a b t*))
and *max m a* < *b* $\Longrightarrow$ *abtls' a b m ts* = (*ab_negmaxs' a b m ts*, *snd*(*abtls* (*max m a*) *b ts*))
proof(*induction a b t* **and** *a b m ts rule*: *abtl'_abtls'.induct*)
case (4 *t ts a b*)
then show *?case*
apply(*simp add*: *Let_def split*: *prod.split*)
by (*smt* (*verit*) *fishburn_ab'_ab_neg*(1) *fst_abtl*(1) *fst_conv linorder_neq_iff max.absorb3 max.cobounded2 max.coboundedI2 max_def minus_less_minus snd_conv uminus_less_reorder*)

qed (*auto simp add*: *split_beta*)

min/*max* in *Lf*

fun $ab\_negmax2$ :: $('a::de\_morgan\_order) \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$ **and** $ab\_negmaxs2$
where
$ab\_negmax2\ a\ b\ (Lf\ x) = max\ a\ (min\ x\ b)\ |$
$ab\_negmax2\ a\ b\ (Nd\ ts) = ab\_negmaxs2\ a\ b\ ts\ |$

$ab\_negmaxs2\ a\ b\ []\ = a\ |$
$ab\_negmaxs2\ a\ b\ (t\#ts) = (let\ a' = -\ ab\_negmax2\ (-b)\ (-a)\ t\ in\ if\ a' = b\ then\ a'\ else\ ab\_negmaxs2\ a'\ b\ ts)$

lemma $ab\_negmax2\_max\_min\_negmax$:
shows $a < b \Longrightarrow ab\_negmax2\ a\ b\ t = max\ a\ (min\ (negmax\ t)\ b)$
and $a < b \Longrightarrow ab\_negmaxs2\ a\ b\ ts = max\ a\ (min\ (negmax\ (Nd\ ts))\ b)$
proof(*induction a b t* **and** *a b ts rule*: *ab_negmax2_ab_negmaxs2.induct*)
next
  case 4 **thus** *?case*
    apply (*simp add*: *Let_def*)
   by (*smt* (*verit*) *less_le_not_le linorder_less_linear max.absorb3 max.cobounded1 max.commute max_min_commute2*
        min.absorb2 min.commute minus_le_minus neg_neg)
qed *auto*

corollary $ab\_negmax2\_bot\_top$: $ab\_negmax2\ \bot\ \top\ t = negmax\ t$
by (*metis ab_negmax2_max_min_negmax*(1) *bounded_linorder_collapse max_bot min_top2*)

Delayed test

Now a variant that delays the test to the next call of *ab_negmaxs*. Like Bird and Hughes' version, except that *ab_negmax3* does not cut off the return value.

fun $ab\_negmax3$ :: $('a::de\_morgan\_order) \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a$ **and** $ab\_negmaxs3$
where
$ab\_negmax3\ a\ b\ (Lf\ x) = x\ |$
$ab\_negmax3\ a\ b\ (Nd\ ts) = ab\_negmaxs3\ a\ b\ ts\ |$

$ab\_negmaxs3\ a\ b\ []\ = a\ |$
$ab\_negmaxs3\ a\ b\ (t\#ts) = (if\ a \geq b\ then\ a\ else\ ab\_negmaxs3\ (max\ a\ (-\ ab\_negmax3\ (-b)\ (-a)\ t))\ b\ ts)$

lemma $ab\_negmax3\_ab\_negmax$:
shows $a{<}b \Longrightarrow ab\_negmax3\ a\ b\ t = ab\_negmax\ a\ b\ t$
and $a{<}b \Longrightarrow ab\_negmaxs3\ a\ b\ ts = ab\_negmaxs\ a\ b\ ts$
proof(*induction a b t* **and** *a b ts rule*: *ab_negmax3_ab_negmaxs3.induct*)
  case (4 *a b t ts*)
  show *?case*

proof (*cases ts*)
  case *Nil*
  then show *?thesis* **using** 4 **by** (*simp add*: *Let_def*)
next
  case *Cons*
  then show *?thesis* **using** 4 **by** (*auto simp add*: *Let_def le_max_iff_disj*)
qed
qed *auto*

corollary *ab_negmax3_bot_top*: $ab\_negmax3\ \bot\ \top\ t = negmax\ t$
by(*metis fishburn_negmax_ab_negmax ab_negmax3_ab_negmax*(1) *bounded_linorder_collapse fishburn_bot_top*)

lemma *ab_negmaxs3_foldl*:
$ab\_negmaxs3\ a\ b\ ts = foldl\ (\lambda a\ t.\ if\ a \geq b\ then\ a\ else\ max\ a\ (-\ ab\_negmax3\ (-b)\ (-a)\ t))\ a\ ts$
apply(*induction ts arbitrary*: *a*)
by (*auto simp*: *Let_def*) (*metis ab_negmaxs3.elims*)

2.5.3 From the Right, Fail-Hard

fun $abr :: ('a::de\_morgan\_order)tree \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **and** *abrs* **where**
$abr\ (Lf\ x)\ a\ b = x\ |$
$abr\ (Nd\ ts)\ a\ b = abrs\ ts\ a\ b\ |$

$abrs\ []\ a\ b = a\ |$
$abrs\ (t\#ts)\ a\ b = (let\ m = abrs\ ts\ a\ b\ in\ if\ m \geq b\ then\ m\ else\ max\ (-\ abr\ t\ (-b)\ (-m))\ m)$

lemma *Lf_eq_negateD*: $Lf\ x = negate\ f\ t \Longrightarrow t = Lf(if\ f\ then\ -x\ else\ x)$
by(*cases t*) *auto*

lemma *Nd_eq_negateD*: $Nd\ ts' = negate\ f\ t \Longrightarrow \exists\, ts.\ t = Nd\ ts \wedge ts' = map\ (negate\ (\neg f))\ ts$
by(*cases t*) (*auto simp*: *comp_def cong*: *map_cong*)

lemma *abr01_negate*:
shows $abr0\ (negate\ f\ t)\ a\ b = -\ abr1\ (negate\ (\neg f)\ t)\ (-a)\ (-b)$
and $abrs0\ (map\ (negate\ f)\ ts)\ a\ b = -\ abrs1\ (map\ (negate\ (\neg f))\ ts)\ (-a)\ (-b)$
and $abr1\ (negate\ f\ t)\ a\ b = -\ abr0\ (negate\ (\neg f)\ t)\ (-a)\ (-b)$
and $abrs1\ (map\ (negate\ f)\ ts)\ a\ b = -\ abrs0\ (map\ (negate\ (\neg f))\ ts)\ (-a)\ (-b)$
proof(*induction negate f t a b* **and** *map* (*negate f*) *ts a b* **and** *negate f t a b* **and** *map* (*negate f*) *ts a b arbitrary*: *f t* **and** *f ts* **and** *f t* **and** *f ts rule*: *abr0_abrs0_abr1_abrs1.induct*)
  case (1 *x a b*)
  from *Lf_eq_negateD*[*OF this*] **show** *?case* **by** *simp*
next
  case (2 *ts a b*)
  from *Nd_eq_negateD*[*OF* 2(2)] 2(1) **show** *?case* **by** *auto*

next
  case (3 *a b*)
  then show *?case* **by** *simp*
next
  case (4 *t ts a b*)
  from *Cons_eq_map_D*[*OF* 4(3)] 4(1) 4(2)[*OF refl*] **show** *?case*
   by (*auto simp*: *Let_def de_morgan_min*) (*metis neg_neg uminus_le_reorder*)+
next
  case (5 *x a b*)
  from *Lf_eq_negateD*[*OF this*] **show** *?case* **by** *simp*
next
  case (6 *ts a b*)
  from *Nd_eq_negateD*[*OF* 6(2)] 6(1) **show** *?case* **by** *auto*
next
  case (7 *a b*)
  then show *?case* **by** *simp*
next
  case (8 *t ts a b*)
  from *Cons_eq_map_D*[*OF* 8(3)] 8(1) 8(2)[*OF refl*] **show** *?case*
    by (*auto simp*: *Let_def de_morgan_max uminus_le_reorder*)
qed

lemma *abr_abr0*:
 shows *abr t a b* = *abr0* (*negate False t*) *a b*
 and *abrs ts a b* = *abrs0* (*map* (*negate True*) *ts*) *a b*
proof (*induction t a b* **and** *ts a b rule*: *abr_abrs.induct*)
  case (1 *x a b*)
  then show *?case* **by** *simp*
next
  case (2 *ts a b*)
  then show *?case* **by** *simp*
next
  case (3 *a b*)
  then show *?case* **by** *simp*
next
  case (4 *t ts a b*)
  then show *?case*
    by (*simp add*: *Let_def abr01_negate*(3))
qed

Relationship to *foldr*

fun *foldr* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ list \Rightarrow 'b$ **where**
foldr f v [] = *v* |
foldr f v (*x*#*xs*) = *f x* (*foldr f v xs*)

definition *abrsf b* = ($\lambda t\ m.$ *if* $m \geq b$ *then m else max* ($-$ *abr t* ($-b$) ($-m$)) *m*)

lemma *abrs_foldr*: *abrs ts a b* = *foldr* (*abrsf b*) *a ts*

by(*induction ts arbitrary*: *a*) (*auto simp*: *abrsf_def Let_def*)

A direct (rather than mutually) recursive def of *abr*

lemma *abr_Nd_foldr*:
  abr (*Nd ts*) *a b* = *foldr* (*abrsf b*) *a ts*
by (*simp add*: *abrs_foldr*)

Direct correctness proof of *foldr* version is no simpler than proof via *abr*/*abrs*:

lemma *fishburn_abr_foldr*: $a < b \Longrightarrow$ *fishburn a b* (*negmax t*) (*abr t a b*)
proof(*induction t arbitrary*: *a b*)
  case (*Lf*)
  then show *?case* **by** *simp*
next
  case (*Nd ts*)
  then show *?case*
  proof(*induction ts*)
    case *Nil*
    then show *?case* **apply** *simp* **using** *linorder_not_less* **by** *blast*
  next
    case (*Cons t ts*)
    then show *?case* **using** *Nd*
      apply (*simp add*: *abrsf_def abr_Nd_foldr*)
     by (*smt* (*verit*) *max.absorb*3 *max.bounded_iff max.commute minus_le_minus nle_le nless_le uminus_less_reorder*)
  qed
qed

The long proofs that follows are duplicated from the *bounded_linorder* section.

fishburn Proofs

lemma *abrs_ge_a*: *abrs ts a b* $\geq$ *a*
by (*simp add*: *abr_abr*0(2) *abrs*0*_ge_a*)

Automatic correctness proof, also works for *knuth* instead of *fishburn*:

corollary *fishburn_abr_negmax*:
  shows $a < b \Longrightarrow$ *fishburn a b* (*negmax t*)   (*abr t a b*)
    and $a < b \Longrightarrow$ *fishburn a b* (*negmax* (*Nd ts*)) (*abrs ts a b*)
apply (*metis abr_abr*0(1) *negmax_maxmin fishburn_abr*01(1))
by (*metis abr.simps*(2) *abr_abr*0(1) *negmax_maxmin fishburn_abr*01(1))

corollary *knuth_abr_negmax*: $a < b \Longrightarrow$ *knuth a b* (*negmax t*) (*abr t a b*)
by (*meson order.trans fishburn_abr_negmax*(1))

corollary *abr_cor*: *abr t* $\perp$ $\top$ = *negmax t*
by (*metis* (*mono_tags*) *bot.extremum_strict knuth_abr_negmax knuth_bot_top linorder_not_less*)

Detailed *fishburn*2 proof (85 lines):

```
theorem fishburn2_abr:
  shows a < b ⟹ fishburn a b (negmax t)   (abr t a b)
    and a < b ⟹ fishburn a b (negmax (Nd ts)) (abrs ts a b)
unfolding fishburn2
proof(induction t a b and ts a b rule: abr_abrs.induct)
  case (4 t ts a b)

  let ?m = abrs ts a b
  let ?ab = abrs (t # ts) a b
  let ?nm1 = negmax t
  let ?nms = negmax (Nd ts)
  let ?nm1s = negmax (Nd (t # ts))
  let ?r = abr t (- b) (- ?m)

  have 1: ?nm1s ≤ ?ab if asm: ?ab < b
  proof -
    have ¬ b ≤ ?m using asm by(auto simp add: Let_def)
    hence ?ab = max (- ?r) ?m by(simp add: Let_def)
    hence - b < ?r using uminus_less_reorder asm by auto
    have ‹?nm1s = max (- ?nm1) ?nms› by simp
    also have ... ≤ max (- ?r) ?m
    proof -
      have - ?nm1 ≤ - ?r
        using 4.IH(2)[OF refl, THEN conjunct1] ‹- b < ?r› ‹¬ b ≤ ?m› by auto
      moreover have ?nms ≤ ?m
        using 4.IH(1)[OF ‹a < b›, THEN conjunct2] ‹¬ b ≤ ?m› leI by blast
      ultimately show ?thesis by (metis max.mono)
    qed
    also note ‹?ab = max (- ?r) ?m›[symmetric]
    finally show ?thesis .
  qed

  have 2: ?ab ≤ ?nm1s if a < ?ab
  proof cases
    assume b ≤ ?m
    have ?ab = ?m using ‹b ≤ ?m› by(simp)
    also have ... ≤ ?nms using 4.IH(1)[OF ‹a < b›, THEN conjunct1]
      by (metis order.strict_trans2[OF ‹a<b› ‹b ≤ ?m›])
    also have ... ≤ max (-?nm1) ?nms
      using max.cobounded2 by blast
    also have ... = ?nm1s by simp
    finally show ?thesis .
  next
    assume ¬ b ≤ ?m
    hence IH2: ?r < - ?m ⟶ ?nm1 ≤ ?r
      using 4.IH(2)[OF refl,THEN conjunct2] minus_less_minus linorder_not_le
by blast
    have a < ?m ∨ ¬ a < ?m ∧ a < - ?r using ‹a < ?ab› ‹¬ b ≤ ?m›
```

```
      by(auto simp: Let_def less_max_iff_disj)
    then show ?thesis
    proof
      assume a < ?m
      hence IH1: ?m ≤ ?nms
        using 4.IH(1)[OF ‹a<b›, THEN conjunct1] by blast
      have 1: − ?r ≤ ?nm1s
      proof cases
        assume ?r < − ?m
        thus ?thesis using IH2 by (simp add: le_max_iff_disj)
      next
        assume ¬ ?r < − ?m
        thus ?thesis using IH1
          apply simp
          by (smt (verit) le_max_iff_disj linorder_le_less_linear order.trans that
uminus_le_reorder)
      qed
      have 2: ?m ≤ ?nm1s
        using IH1 by(auto simp: le_max_iff_disj)
      show ?thesis
        using ‹¬ b ≤ ?m› 1 2 by(simp add: Let_def not_le_imp_less)
    next
      assume a: ¬ a < ?m ∧ a < − ?r
      have 1 : − ?r ≤ − ?nm1
        using ‹¬ a < ?m ∧ a < − ?r› IH2 less_uminus_reorder
        by (metis abrs_ge_a linorder_not_le minus_le_minus nle_le)
      have 2: ?m ≤ − ?nm1
        using a 4.IH(1)[OF ‹a<b›, THEN conjunct1] 1
        by (meson dual_order.strict_iff_not order.trans nle_le)
      have ?ab = max (− ?r) ?m
        using a ‹¬ b ≤ ?m› by(simp add: Let_def)
      also have ... ≤ max (− ?nm1) ?nms
        using 1 2 by (simp add: max.coboundedI1)
      also have ... = ?nm1s
        by simp
      finally show ?thesis .
    qed
  qed
  show ?case using 1 2 by blast
qed auto
```

Detailed *fishburn* proof (100 lines):

```
theorem fishburn_abr:
  shows a < b ⟹ fishburn a b (negmax t)   (abr t a b)
    and a < b ⟹ fishburn a b (negmax (Nd ts)) (abrs ts a b)
proof(induction t a b and ts a b rule: abr_abrs.induct)
  case (4 t ts a b)
  let ?m = abrs ts a b
  let ?ab = abrs (t # ts) a b
```

let *?r* = *abr t* (− *b*) (− *?m*)
let *?nm*1 = *negmax t*
let *?nms* = *negmax* (*Nd ts*)
let *?nm*1*s* = *negmax* (*Nd* (*t* # *ts*))
have *?nm*1*s* = *max* (− *?nm*1) *?nms* **by** *simp*

have 1: *?nm*1*s* ≤ *?ab* **if** *asm*: *?ab* ≤ *a*
proof −
have ¬ *b* ≤ *?m* **by** (*rule ccontr*) (*use* ‹*?ab* ≤ *a*› ‹*a* < *b*› **in** *simp*)
hence ∗: *?ab* = *max* (− *?r*) *?m* **by**(*simp add*: *Let_def*)
hence *?m* ≤ *a* − *?r* ≤ *a* **using** *asm* **by** *auto*
have − *b* < *?r* **using** ‹− *?r* ≤ *a*› ‹*a* < *b*› *uminus_less_reorder order_le_less_trans* **by** *blast*
note ‹*?nm*1*s* = _›
also have *max* (− *?nm*1) *?nms* ≤ *max* (− *?r*) *?m*
proof −
have − *?nm*1 ≤ − *?r*
proof *cases*
assume − *?m* ≤ *?r*
thus *?thesis* **using** 4.*IH*(2)[*THEN conjunct*2, *THEN conjunct*2] ‹¬ *b* ≤ *?m*› **by** *auto*
next
assume ¬ − *?m* ≤ *?r*
thus *?thesis* **using** 4.*IH*(2)[*THEN conjunct*2, *THEN conjunct*1] ‹¬ *b* ≤ *?m*› ‹− *b* < *?r*› **by** (*simp*)
qed
moreover have *?nms* ≤ *?m*
using ‹*?m* ≤ *a*› 4.*IH*(1)[*OF* 4.*prems*, *THEN conjunct*1] **by** (*auto*)
ultimately show *?thesis* **by** (*metis max.mono*)
qed
also note ∗[*symmetric*]
finally show *?thesis* **.**
qed

have 2: *?ab* ≤ *?nm*1*s* **if** *b* ≤ *?ab*
proof *cases*
assume *?m* ≥ *b*
show *?thesis* **using** ‹*?m* ≥ *b*›
using 4.*IH*(1)[*OF* ‹*a* < *b*›,*THEN conjunct*2,*THEN conjunct*2] *max.coboundedI*2 **by** *auto*
next
assume ¬ *?m* ≥ *b*
hence *?ab* = *max* (−*?r*) *?m* **by**(*simp add*: *Let_def*)
hence −*?r* ≥ *b* **using** ‹*b* ≤ *?ab*› ‹¬ *?m* ≥ *b*› **by** (*metis le_max_iff_disj*)
hence *?ab* = −*?r*
using ‹*?ab* = _› ‹¬ *b* ≤ *?m*› **by**(*metis not_le_imp_less less_le_trans max.absorb*3)
also have . . . ≤ − *?nm*1 **using** 4.*IH*(2)[*OF refl*,*THEN conjunct*1] ‹¬ *b* ≤ *?m*› ‹*b* ≤ −*?r*›

    by (metis minus_le_minus uminus_le_reorder linorder_not_le)
  also have ... ≤ max (− ?nm1) ?nms by auto
  also note ‹?nm1s = ...›[symmetric]
  finally show ?thesis .
qed

have 3: ?ab = ?nm1s if asm: a < ?ab ?ab < b
proof −
  have ¬ b ≤ ?m by (rule ccontr) (use ‹?ab < b› in simp)
  hence *: ?ab = max (− ?r) ?m by(simp add: Let_def)
  hence − ?r < b using ‹?ab < b› by auto
  note *
  also have max (− ?r) ?m = ?nm1s
  proof −
    have ?r = ?nm1 ∧ ?nms ≤ − ?r if ‹¬ − ?r ≤ ?m›
    proof
      have − b < ?r ∧ ?r < − ?m
        using ‹− ?r < b› ‹¬ − ?r ≤ ?m›
        by (metis minus_less_minus neg_neg linorder_not_le)
      thus ?r = ?nm1
        using 4.IH(2)[OF refl, THEN conjunct2,THEN conjunct1] ‹¬ b ≤ ?m›
        using order.strict_trans by blast
      show ?nms ≤ − ?r
      proof cases
        assume ?m ≤ a
         thus ?thesis using 4.IH(1)[OF ‹a<b›,THEN conjunct1] ‹¬ − ?r ≤ ?m›
by simp
      next
        assume ¬ ?m ≤ a
        thus ?thesis
          using 4.IH(1)[OF ‹a<b›,THEN conjunct2, THEN conjunct1] ‹¬ − ?r ≤
?m› ‹¬ b ≤ ?m› by auto
      qed
    qed
    moreover have ?m = ?nms ∧ − ?nm1 ≤ ?m if − ?r ≤ ?m
    proof
      note ‹a < ?ab›
      also note *
       also have max (− ?r) ?m = ?m using ‹− ?r ≤ ?m› using max.absorb2
by blast
      finally have a < ?m .
      thus ?m = ?nms
        using 4.IH(1)[OF ‹a < b›,THEN conjunct2,THEN conjunct1] ‹¬ b ≤ ?m›
not_le by blast
      have − ?nm1 ≤ − ?r
         using 4.IH(2)[OF refl,THEN conjunct2,THEN conjunct2] ‹¬ b ≤ ?m›
‹− ?r ≤ ?m›
        by (metis neg_neg not_le minus_le_minus)
      also note ‹− ?r ≤ ?m›

finally show $-$ *?nm*1 $\leq$ *?m* **.**
qed
ultimately show *?thesis* **using** ‹*?nm*1*s* = _› **by** *fastforce*
qed
finally show *?thesis* **.**
qed
show *?case* **using** 1 2 3 **by** *blast*
qed *auto*

Explicit equational *knuth* proofs via min/max

Not mm, only min and max. Only min in abrs. $a<b$ required: a=1, b=-1, t=[]

theorem shows *abr_negmax3*: *max a* (*min* (*abr t a b*) *b*) = *max a* (*min* (*negmax t*) *b*)
and $a < b \Longrightarrow$ *min* (*abrs ts a b*) *b* = *max a* (*min* (*negmax* (*Nd ts*)) *b*)
proof(*induction t a b* **and** *ts a b rule*: *abr_abrs.induct*)
case (2 *ts a b*)
then show *?case* **apply** *simp*
by (*metis max_def min.strict_boundedE order_neq_le_trans*)
next
case (4 *t ts a b*)
let *?abts* = *abrs ts a b* **let** *?abt* = *abr t* ($-$ *b*) ($-$ *?abts*)
show *?case*
proof (*cases b* $\leq$ *?abts*)
case *True*
thus *?thesis* **using** 4.*IH*(1)[*OF* 4.*prems*] **apply** (*simp add*: *Let_def*)
by (*metis* (*no_types*) *max.left_commute max_min_same*(2) *min.commute min_max_distrib*2)
next
case *False*

have *IH*2: *min b* (*max* ($-$*?abt*) *?abts*) = *min b* (*max* ($-$ *negmax t*) *?abts*)
using 4.*IH*(2) *False* **by** (*metis* (*no_types*) *neg_neg de_morgan_max*)
have *min* (*abrs* (*t* # *ts*) *a b*) *b* = *min* (*max* ($-$ *?abt*) *?abts*) *b*
using *False* **by** (*simp add*: *Let_def*)
also have . . . = *min b* (*max* ($-$ *?abt*) *?abts*)
by (*metis min.commute*)
also have . . . = *min b* (*max* ($-$ *negmax t*) *?abts*)
using *IH*2 **by** *blast*
also have . . . = *max* (*min* ($-$ *negmax t*) *b*) (*min ?abts b*)
by (*metis min.commute min_max_distrib*2)
also have . . . = *max* (*min* ($-$ *negmax t*) *b*) (*max a* (*min* (*negmax* (*Nd ts*)) *b*))
using 4.*IH*(1)[*OF* 4.*prems*] **by** *presburger*
also have . . . = *max a* (*min* (*max* ($-$ *negmax t*) (*negmax* (*Nd ts*))) *b*)
by (*metis max.left_commute min_max_distrib*1)
finally show *?thesis* **by** *simp*
qed
qed *auto*

Not mm, only min and max. Also max in abrs:

theorem shows *abr_negmax2*: *max a* (*min* (*abr t a b*) *b*) = *max a* (*min* (*negmax t*) *b*)
and *a* < *b* ⟹ *max a* (*min* (*abrs ts a b*) *b*) = *max a* (*min* (*negmax* (*Nd ts*)) *b*)
proof(*induction t a b* **and** *ts a b rule*: *abr_abrs.induct*)
case 2
thus *?case* **apply** *simp*
by (*metis max.orderE min.strict_boundedE not_le_imp_less*)
case (4 *t ts a b*)
let *?abts* = *abrs ts a b* **let** *?abt* = *abr t* (− *b*) (− *?abts*)
show *?case*
proof (*cases b* ≤ *?abts*)
case *True*
thus *?thesis* **using** 4.*IH*(1)[*OF* 4.*prems*] **apply** (*simp add*: *Let_def*)
by (*metis* (*no_types*) *max.left_commute max_min_same*(2) *min.commute min_max_distrib2*)
next
case *False*

hence *max a* (*min* (*abrs* (*t* # *ts*) *a b*) *b*) = *max a* (*min* (*max* (− *?abt*) *?abts*) *b*)
by (*simp add*: *Let_def linorder_not_le*)
also have ... = *max a* (−(*max* (*min ?abt* (− *?abts*)) (−*b*)))
by (*metis neg_neg de_morgan_max*)
also have ... = *max a* (−(*max* (−*b*) (*min ?abt* (− *?abts*))))
by (*metis max.commute*)
also have ... = *max a* (−(*max* (− *b*) (*min* (*negmax t*) (− *?abts*))))
using 4.*IH*(2)[*OF refl*] *False* **by** (*simp add*: *linorder_not_le*)
also have ... = *max a* ((*min b* (*max* (− *negmax t*) *?abts*)))
by (*metis neg_neg de_morgan_min*)
also have ... = *max* (*min* (− *negmax t*) *b*) (*max a* (*min ?abts b*))
by (*metis max.left_commute min.commute min_max_distrib2*)
also have ... = *max* (*min* (− *negmax t*) *b*) (*max a* (*min* (*negmax* (*Nd ts*)) *b*))
using 4.*IH*(1)[*OF* 4.*prems*] **by** *presburger*
also have ... = *max a* (*min* (*max* (− *negmax t*) (*negmax* (*Nd ts*))) *b*)
by (*metis max.left_commute min_max_distrib1*)
finally show *?thesis* **by** *simp*
qed
qed *auto*

Relating iteration from right and left

Enables porting *abr* lemmas to *ab_negmax* lemmas, eg correctness.

fun *mirror* :: *'a tree* ⇒ *'a tree* **where**
mirror (*Lf x*) = *Lf x* |
mirror (*Nd ts*) = *Nd* (*rev* (*map mirror ts*))

lemma *abrs_append*:

abrs (*ts*1 @ *ts*2) *a b* = (*let m* = *abrs ts*2 *a b in if m* ≥ *b then m else abrs ts*1 *m b*)
by(*induction ts*1 *arbitrary*: *ts*2) (*auto simp add*: *Let_def*)

lemma *ab_negmax_abr_mirror*:
shows *a*<*b* ⟹ *ab_negmax a b t* = *abr* (*mirror t*) *a b*
  and *a*<*b* ⟹ *ab_negmaxs a b ts* = *abrs* (*rev* (*map mirror ts*)) *a b*
proof(*induction a b t* **and** *a b ts rule*: *ab_negmax_ab_negmaxs.induct*)
  case 4
  then show *?case* **by** (*fastforce simp*: *Let_def abrs_append max.commute*)
qed *auto*

lemma *negmax_mirror*:
fixes *t* :: *'a*::*de_morgan_order tree* **and** *ts* :: *'a*::*de_morgan_order tree list*
shows *negmax* (*mirror t*) = *negmax t* ∧ *negmax* (*Nd* (*rev* (*map mirror ts*))) = *negmax* (*Nd ts*)
by(*rule compat_tree_tree_list.induct*)(*auto simp*: *max.commute maxs_rev maxs_append*)

Correctness of *ab_negmax* from correctness of *abr*:

theorem *fishburn_ab_negmax_negmax_mirror*:
shows *a* < *b* ⟹ *fishburn a b* (*negmax t*) (*ab_negmax a b t*)
  and *a* < *b* ⟹ *fishburn a b* (*negmax* (*Nd ts*)) (*ab_negmaxs a b ts*)
apply (*metis* (*no_types*) *ab_negmax_abr_mirror*(1) *negmax_mirror fishburn_abr_negmax*(1))
by (*metis* (*no_types*) *ab_negmax_abr_mirror*(2) *negmax_mirror fishburn_abr_negmax*(2))

2.5.4 From the Right, Fail-Soft

Starting at ⊥ (after Fishburn)

fun *abr'* :: (*'a*::*de_morgan_order*)*tree* ⇒ *'a* ⇒ *'a* ⇒ *'a* **and** *abrs'* **where**
abr' (*Lf x*) *a b* = *x* |
abr' (*Nd ts*) *a b* = *abrs' ts a b* |

abrs' [] *a b* = ⊥ |
abrs' (*t*#*ts*) *a b* = (*let m* = *abrs' ts a b in*
  if m ≥ *b then m else max* (− *abr' t* (−*b*) (− *max m a*)) *m*)

lemma *abr*01*'_negate*:
shows *abr*0*'* (*negate f t*) *a b* = − *abr*1*'* (*negate* (¬*f*) *t*) (−*a*) (−*b*)
and *abrs*0*'* (*map* (*negate f*) *ts*) *a b* = − *abrs*1*'* (*map* (*negate* (¬*f*)) *ts*) (−*a*) (−*b*)
and *abr*1*'* (*negate f t*) *a b* = − *abr*0*'* (*negate* (¬*f*) *t*) (−*a*) (−*b*)
and *abrs*1*'* (*map* (*negate f*) *ts*) *a b* = − *abrs*0*'* (*map* (*negate* (¬*f*)) *ts*) (−*a*) (−*b*)
proof(*induction negate f t a b* **and** *map* (*negate f*) *ts a b* **and** *negate f t a b* **and** *map* (*negate f*) *ts a b arbitrary*: *f t* **and** *f ts* **and** *f t* **and** *f ts rule*: *abr*0*'_abrs*0*'_abr*1*'_abrs*1*'.induct*)
  case (1 *x a b*)
  from *Lf_eq_negateD*[*OF this*] **show** *?case* **by** *simp*
next
  case (2 *ts a b*)

```
  from Nd_eq_negateD[OF 2(2)] 2(1) show ?case by auto
next
  case (3 a b)
  then show ?case by simp
next
  case (4 t ts a b)
  from Cons_eq_map_D[OF 4(3)] 4(1) 4(2)[OF refl] show ?case
    apply (clarsimp simp add: Let_def de_morgan_max de_morgan_min)
    by (metis neg_neg uminus_le_reorder)
next
  case (5 x a b)
  from Lf_eq_negateD[OF this] show ?case by simp
next
  case (6 ts a b)
  from Nd_eq_negateD[OF 6(2)] 6(1) show ?case by auto
next
  case (7 a b)
  then show ?case by simp
next
  case (8 t ts a b)
  from Cons_eq_map_D[OF 8(3)] 8(1) 8(2)[OF refl] show ?case
   by (auto simp: Let_def de_morgan_max de_morgan_min uminus_le_reorder)
qed

lemma abr_abr0':
 shows abr' t a b = abr0' (negate False t) a b
 and abrs' ts a b = abrs0' (map (negate True) ts) a b
proof (induction t a b and ts a b rule: abr'_abrs'.induct)
  case (1 x a b)
  then show ?case by simp
next
  case (2 ts a b)
  then show ?case by simp
next
  case (3 a b)
  then show ?case by simp
next
  case (4 t ts a b)
  then show ?case
    by (simp add: Let_def abr01'_negate(3))
qed

corollary fishburn_abr'_negmax_cor:
  shows a < b ⟹ fishburn a b (negmax t)   (abr' t a b)
    and a < b ⟹ fishburn a b (negmax (Nd ts)) (abrs' ts a b)
apply (metis abr_abr0'(1) negmax_maxmin fishburn_abr01'(1))
by (metis abr'.simps(2) abr_abr0'(1) negmax_maxmin fishburn_abr01'(1))

lemma abr'_exact: ⟦ v = negmax t; a ≤ v ∧ v ≤ b⟧ ⟹ abr' t a b = v
```

```
by (simp add: abr0'_exact abr_abr0'(1) negmax_maxmin)
```

Now a lot of copy-paste-modify from *bounded_linorder*.

```
theorem
  shows a < b ⟹ fishburn a b (abr' t a b) (abr t a b)
    and a < b ⟹ fishburn a b (abrs' ts a b) (abrs ts a b)
proof(induction t a b and ts a b rule: abr_abrs.induct)
  case (4 t ts a b)
  then show ?case apply (simp add: Let_def)
    by (smt (verit) order_eq_iff abrs_ge_a le_max_iff_disj less_uminus_reorder
max_def minus_less_minus nless_le)
qed auto

theorem fishburn2_abr_abr':
  shows a < b ⟹ fishburn a b (abr' t a b) (abr t a b)
    and a < b ⟹ fishburn a b (abrs' ts a b) (abrs ts a b)
unfolding fishburn2
proof(induction t a b and ts a b rule: abr_abrs.induct)
  case (4 t ts a b)

  let ?m = abrs ts a b
  let ?ab = abrs (t # ts) a b
  let ?r = abr t (− b) (− ?m)
  let ?m' = abrs' ts a b
  let ?ab' = abrs' (t # ts) a b
  let ?r' = abr' t (− b) (− max ?m' a)
  note IH1 = 4.IH(1)[OF ‹a < b›] note IH11 = IH1[THEN conjunct1] note IH12
= IH1[THEN conjunct2]

  have 1: ?ab ≤ ?ab' if a < ?ab
  proof cases
    assume b ≤ ?m
    thus ?thesis using IH11 ‹a<b› by (auto simp: Let_def)
  next
    assume ¬ b ≤ ?m
    hence ?ab = max (− ?r) ?m by (simp add: Let_def)
    hence a < max (− ?r) ?m using ‹a < ?ab› by presburger
    have IH22: ?r < − ?m ⟶ abr' t (− b) (− ?m) ≤ ?r
      using ‹¬ b ≤ ?m› 4.IH(2) by auto
    have ¬ b ≤ ?m' using IH12 ‹¬ b ≤ ?m› by auto
    hence ?ab' = max (− ?r') ?m' by (simp add: Let_def)
    have ?m' ≤ ?m using IH12 ‹¬ b ≤ ?m› linorder_not_le by blast
    have max ?m' a = ?m
    proof cases
      assume ?m ≤ a
      thus ?thesis using ‹?m' ≤ ?m› abrs_ge_a
        by (metis max.absorb1 max.commute)
    next
      assume ¬ ?m ≤ a
```

thus *?thesis* **using** *IH*11 ‹*?m'* ≤ *?m*› **by** *auto*
qed
have $-$ *?r* ≤ *max* ($-$ *?r'*) *?m'*
proof *cases*
assume *?r* < $-$ *?m*
thus *?thesis* **using** *IH*22 ‹*max ?m' a* = *?m*›
by (*simp add*: *max.coboundedI*1)
next
assume ¬ *?r* < $-$ *?m*
hence *?m* = *?m'* **using** ‹*a* < *max* ($-$*?r*) *?m*› ‹*max ?m' a* = *?m*›
by (*metis linorder_not_le max.commute max.order_iff nle_le uminus_le_reorder*)
thus ‹$-$ *?r* ≤ *max* ($-$ *?r'*) *?m'*› **using** ‹¬ *?r* < $-$ *?m*›
by (*simp add*: *max.coboundedI*2 *uminus_le_reorder*)
qed
moreover have *?m* ≤ *max* ($-$ *?r'*) (*?m'*)
proof *cases*
assume *a* < *?m*
hence *?m* ≤ *?m'* **using** *IH*11 **by** *simp*
then show *?thesis* **using** *le_max_iff_disj* **by** *blast*
next
assume ¬ *a* < *?m*
hence *?m* ≤ *a* **by** *simp*
also have *a* < $-$ *?r* **using** ‹*a* < *max* ($-$*?r*) *?m*› ‹¬ *a* < *?m*› *less_max_iff_disj*
by *blast*
also note ‹$-$ *?r* ≤ *max* ($-$ *?r'*) *?m'*›
finally show *?thesis* **using** *order.order_iff_strict* **by** *blast*
qed
ultimately show *?thesis* **using** ‹*?ab* = _› ‹*?ab'* = _› **by** (*metis max.bounded_iff*)
qed

have 2: *?ab'* ≤ *?ab* **if** *?ab* < *b*
proof *cases*
assume *b* ≤ *?m*
thus *?thesis*
using ‹*?ab* < *b*› **by** (*simp add*: *Let_def*)
next
assume ¬ *b* ≤ *?m*
hence $-$ *?r* < *b* **using** ‹*?ab* < *b*› **by**(*auto simp*: *Let_def*)
with 4.*IH*(2) ‹¬ *b* ≤ *?m*›
have *IH*21: *?r* ≤ *abr' t* ($-$ *b*) ($-$ *?m*)
by (*metis linorder_le_less_linear minus_less_minus uminus_less_reorder*)
have ¬ *b* ≤ *?m' ?m'* ≤ *?m* **using** *IH*12 ‹¬ *b* ≤ *?m*› **by** *auto*
have *?ab* = *max* ($-$ *?r*) *?m* **using** ‹¬ *b* ≤ *?m*› **by** (*simp add*: *Let_def*)
hence *?ab'* = *max* ($-$ *?r'*) *?m'* **using** ‹¬ *b* ≤ *?m'*› **by** (*simp add*: *Let_def*)
have $-$ *?r'* ≤ $-$ *?r*
proof *cases*
assume *a* < *?m*
hence *?m'* = *?m* **using** *IH*11 ‹*?m'* ≤ *?m*› *nle_le* **by** *blast*
hence $-$ *?r'* = $-$ *abr' t* ($-$ *b*) ($-$ *?m*) **using** ‹*a* < *?m*› **by** *simp*

also have . . . $\le -$ *?r* **using** *IH21 minus_le_minus* **by** *blast*
finally show *?thesis* **.**
next
assume $\neg$ *a* $<$ *?m*
have *?m* = *a* **using** ‹$\neg$ *a* < *abrs ts a b*› *abrs_ge_a* **by** (*metis order_le_imp_less_or_eq*)
hence *max ?m′ a* = *a* **using** ‹*?m′* $\le$ *?m*› **by** *simp*
with ‹*?m* = *a*› **show** *?thesis* **using** *IH21* **by** *simp*
qed
then have $-$ *?r′* $\le$ *max* ($-$ *?r*) *?m* **using** *max.coboundedI1* **by** *blast*
hence *max* ($-$ *?r′*) *?m′* $\le$ *max* ($-$ *?r*) *?m*
using ‹*?m′* $\le$ *?m*› **by** (*metis max.absorb2 max.bounded_iff max.cobounded2*)
thus *?thesis* **using** ‹*?ab* = _› ‹*?ab′* = _› **by** *metis*
qed

show *?case* **using** 1 2 **by** *blast*

qed(*auto simp add*: *bot_ereal_def*)

theorem *fishburn_abr′_negmax*:
shows *a* < *b* $\Longrightarrow$ *fishburn a b* (*negmax t*) (*abr′ t a b*)
and *a* < *b* $\Longrightarrow$ *fishburn a b* (*negmax* (*Nd ts*)) (*abrs′ ts a b*)
unfolding *fishburn2*
proof(*induction t a b* **and** *ts a b rule*: *abr′_abrs′.induct*)
case (4 *t ts a b*)

let *?m* = *abrs′ ts a b*
let *?ab* = *abrs′* (*t* # *ts*) *a b*
let *?nm1* = *negmax t*
let *?nms* = *negmax* (*Nd ts*)
let *?nm1s* = *negmax* (*Nd* (*t* # *ts*))
let *?r* = *abr′ t* ($-$ *b*) ($-$ *max ?m a*)

have 1: *?nm1s* $\le$ *?ab* **if** *asm*: *?ab* < *b*
proof $-$
have $\neg$ *b* $\le$ *?m* **using** *asm* **by**(*auto simp add*: *Let_def*)
hence *?ab* = *max* ($-$ *?r*) *?m* **by**(*simp add*: *Let_def*)
hence $-$ *b* < *?r* **using** *uminus_less_reorder asm* **by** *auto*
have ‹*?nm1s* = *max* ($-$ *?nm1*) *?nms*› **by** *simp*
also have . . . $\le$ *max* ($-$ *?r*) *?m*
proof $-$
have $-$ *?nm1* $\le$ $-$ *?r*
using 4.*IH*(2)[*OF refl*, *THEN conjunct1*] ‹*a*<*b*› ‹$-$ *b* < *?r*› ‹$\neg$ *b* $\le$ *?m*› **by**
auto
moreover have *?nms* $\le$ *?m*
using 4.*IH*(1)[*OF* ‹*a* < *b*›, *THEN conjunct2*] ‹$\neg$ *b* $\le$ *?m*› *leI* **by** *blast*
ultimately show *?thesis* **by** (*metis max.mono*)
qed
also note ‹*?ab* = *max* ($-$ *?r*) *?m*›[*symmetric*]
finally show *?thesis* **.**

```
  qed

  have 2: ?ab ≤ ?nm1s if a < ?ab
  proof cases
    assume b ≤ ?m
    have ?ab = ?m using ‹b ≤ ?m› by(simp)
    also have ... ≤ ?nms using 4.IH(1)[OF ‹a < b›, THEN conjunct1]
      by (metis order.strict_trans2[OF ‹a<b› ‹b ≤ ?m›])
    also have ... ≤ max (−?nm1) ?nms
      using max.cobounded2 by blast
    also have ... = ?nm1s by simp
    finally show ?thesis .
  next
    assume ¬ b ≤ ?m
    hence IH2: ?r < − ?m ⟶ ?nm1 ≤ ?r
      using 4.IH(2)[OF refl, THEN conjunct2] ‹a<b› ‹a < ?ab› by (auto simp: less_uminus_reorder)
    have a < ?m ∨ ¬ a < ?m ∧ a < − ?r using ‹a < ?ab› ‹¬ b ≤ ?m›
      by(auto simp: Let_def less_max_iff_disj)
    then show ?thesis
    proof
      assume a < ?m
      hence IH1: ?m ≤ ?nms
        using 4.IH(1)[OF ‹a<b›, THEN conjunct1] by blast
      have 1: − ?r ≤ ?nm1s
      proof cases
        assume ?r < − ?m
        thus ?thesis using IH2 by (simp add: le_max_iff_disj)
      next
        assume ¬ ?r < − ?m
        thus ?thesis using IH1
          by (simp add: less_uminus_reorder max.coboundedI2)
      qed
      have 2: ?m ≤ ?nm1s
        using IH1 by(auto simp: le_max_iff_disj)
      show ?thesis
        using ‹¬ b ≤ ?m› 1 2 by(simp add: Let_def not_le_imp_less)
    next
      assume a: ¬ a < ?m ∧ a < − ?r
      have 1 : − ?r ≤ − ?nm1
        using ‹¬ a < ?m ∧ a < − ?r› IH2 by (auto simp: less_uminus_reorder)
      have 2: ?m ≤ − ?nm1
        using a 4.IH(1)[OF ‹a<b›, THEN conjunct1] 1
        by (meson dual_order.strict_iff_not order.trans nle_le)
      have ?ab = max (− ?r) ?m
        using a ‹¬ b ≤ ?m› by(simp add: Let_def)
      also have ... ≤ max (− ?nm1) ?nms
        using 1 2 by (simp add: max.coboundedI1)
      also have ... = ?nm1s
```

by *simp*
finally show *?thesis* **.**
qed
qed
show *?case* **using** 1 2 **by** *blast*
qed (*auto simp add*: *bot_ereal_def*)

Automatic proof:

theorem
shows $a < b \Longrightarrow$ *fishburn a b* (*negmax t*) (*abr′ t a b*)
and $a < b \Longrightarrow$ *fishburn a b* (*negmax* (*Nd ts*)) (*abrs′ ts a b*)
unfolding *fishburn*2
proof(*induction t a b* **and** *ts a b rule*: *abr′_abrs′.induct*)
case (4 *t ts a b*)
then show *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit*, *best*) *abrs_ge_a less_uminus_reorder uminus_less_reorder linorder_not_le max.absorb*1 *max.absorb_iff*2 *nle_le order.trans*)
qed (*auto simp add*: *bot_ereal_def*)

Also returning the searched tree

Hard:

fun *abtr* :: (*′a*::*de_morgan_order*) *tree* ⇒ *′a* ⇒ *′a* ⇒ *′a* ∗ *′a tree* **and** *abtrs* **where**
abtr (*Lf x*) *a b* = (*x*, *Lf x*) |
abtr (*Nd ts*) *a b* = (*let* (*m*,*us*) = *abtrs ts a b in* (*m*, *Nd us*)) |

abtrs [] *a b* = (*a*,[]) |
abtrs (*t#ts*) *a b* = (*let* (*m*,*us*) = *abtrs ts a b in*
if $m \geq b$ *then* (*m*,*us*) *else let* (*n*,*u*) = *abtr t* (−*b*) (−*m*) *in* (*max* (−*n*) *m*,*u#us*))

Soft:

fun *abtr′* :: (*′a*::*de_morgan_order*) *tree* ⇒ *′a* ⇒ *′a* ⇒ *′a* ∗ *′a tree* **and** *abtrs′* **where**
abtr′ (*Lf x*) *a b* = (*x*, *Lf x*) |
abtr′ (*Nd ts*) *a b* = (*let* (*m*,*us*) = *abtrs′ ts a b in* (*m*, *Nd us*)) |

abtrs′ [] *a b* = (⊥,[]) |
abtrs′ (*t#ts*) *a b* = (*let* (*m*,*us*) = *abtrs′ ts a b in*
if $m \geq b$ *then* (*m*,*us*) *else let* (*n*,*u*) = *abtr′ t* (−*b*) (− *max m a*) *in* (*max* (−*n*) *m*,*u#us*))

lemma *fst_abtr*:
shows *fst*(*abtr t a b*) = *abr t a b*
and *fst*(*abtrs ts a b*) = *abrs ts a b*
by(*induction t a b* **and** *ts a b rule*: *abtr_abtrs.induct*)
(*auto simp*: *Let_def split*: *prod.split*)

lemma *fst_abtr′*:

shows $fst(abtr'\ t\ a\ b) = abr'\ t\ a\ b$
and $fst(abtrs'\ ts\ a\ b) = abrs'\ ts\ a\ b$
by(*induction t a b* **and** *ts a b rule*: *abtr'_abtrs'.induct*)
  (*auto simp*: *Let_def split*: *prod.split*)

lemma *snd_abtr'_abtr*:
shows $a < b \Longrightarrow snd(abtr'\ t\ a\ b) = snd(abtr\ t\ a\ b)$
and $a < b \Longrightarrow snd(abtrs'\ ts\ a\ b) = snd(abtrs\ ts\ a\ b)$
proof(*induction t a b* **and** *ts a b rule*: *abtr'_abtrs'.induct*)
  case (4 *t ts a b*)
  then show *?case*
    apply(*simp add*: *Let_def split*: *prod.split*)
    using *fst_abtr*(2) *fst_abtr'*(2) *fishburn2_abr_abr'*(2) *abrs_ge_a*
     by (*smt* (*verit*, *best*) *fst_conv le_max_iff_disj linorder_not_le max.absorb*1
nle_le sndI)
qed (*auto simp add*: *split_beta*)

Fail-Soft Generalized

fun *abir'* :: _ ⇒ (*'a*::*de_morgan_order*)*tree* ⇒ *'a* ⇒ *'a* ⇒ *'a* **and** *abirs'* **where**
$abir'\ i0\ (Lf\ x)\ a\ b = x\ |$
$abir'\ i0\ (Nd\ ts)\ a\ b = abirs'\ i0\ (i0\ (map\ (negate\ True)\ ts)\ a)\ ts\ a\ b\ |$

$abirs'\ i0\ i\ []\ a\ b = i\ |$
$abirs'\ i0\ i\ (t\#ts)\ a\ b =$
  $(let\ m = abirs'\ i0\ i\ ts\ a\ b$
   $in\ if\ m \geq b\ then\ m\ else\ max\ (-\ abir'\ i0\ t\ (-\ b)\ (-\ max\ m\ a))\ m)$

abbreviation $neg\_all \equiv negate\ True\ o\ negate\ False$

lemma *neg_all_negate*: $neg\_all\ (negate\ f\ t) = negate\ (\neg f)\ t$
proof(*induction t arbitrary*: *f*)
  case (*Nd ts*)
  { fix *t*
    assume $t \in set\ ts$
    from *Nd*[*OF this*, *of True*] **have** $neg\_all\ t = negate\ False\ (negate\ True\ t)$
      by (*metis comp_apply negate_negate*)
  }
  with *Nd*[*of _ False*] **show** *?case*
    by (*cases f*) (*auto simp*: *negate_negate*)
qed *simp*

lemma *neg_all_negate'*: $neg\_all\ o\ negate\ f = negate\ (\neg f)$
  using *neg_all_negate* **by** *fastforce*

lemma *abir01'_negate*:
shows $\forall ts\ a.\ i1\ ts\ a = -\ i0\ (map\ neg\_all\ ts)\ (-a) \Longrightarrow$
  $abir0'\ i0\ i1\ (negate\ f\ t)\ a\ b = -\ abir1'\ i0\ i1\ (negate\ (\neg f)\ t)\ (-a)\ (-b)$
and $\forall ts\ a.\ i1\ ts\ a = -\ i0\ (map\ neg\_all\ ts)\ (-a) \Longrightarrow$

$abirs0'$ $i0$ $i1$ i (map ($negate$ f) ts) a b = − $abirs1'$ $i0$ $i1$ (−i) (map ($negate$ (¬f)) ts) (−a) (−b)
and ∀ ts a. $i1$ ts a = − $i0$ (map neg_all ts) (−a) ⟹
$abir1'$ $i0$ $i1$ ($negate$ f t) a b = − $abir0'$ $i0$ $i1$ ($negate$ (¬f) t) (−a) (−b)
and ∀ ts a. $i1$ ts a = − $i0$ (map neg_all ts) (−a) ⟹
$abirs1'$ $i0$ $i1$ i (map ($negate$ f) ts) a b = − $abirs0'$ $i0$ $i1$ (−i) (map ($negate$ (¬f)) ts) (−a) (−b)
proof(*induction* $i0$ $i1$ *negate f t a b* **and** $i0$ $i1$ *i map* (*negate f*) *ts a b* **and** $i0$ $i1$ *negate f t a b* **and** $i0$ $i1$ *i map* (*negate f*) *ts a b arbitrary*: *f t* **and** *f ts* **and** *f t* **and** *f ts rule*: *abir0'_abirs0'_abir1'_abirs1'.induct*)
case (1 *x a b*)
from *Lf_eq_negateD this* **show** *?case* **by** *fastforce*
next
case (2 $i0$ $i1$ *ts a b*)
from *Nd_eq_negateD*[*OF* 2(2)] 2(1,3) **show** *?case* **by** (*auto simp*: *neg_all_negate'*)
next
case (3 *a b*)
then show *?case* **by** *simp*
next
case (4 $i0$ $i1$ *i t ts a b*)
from *Cons_eq_map_D*[*OF* 4(3)] 4(1,4) 4(2)[*OF refl*] **show** *?case*
apply (*clarsimp simp*: *Let_def de_morgan_min de_morgan_max*)
by (*metis neg_neg uminus_le_reorder*)
next
case (5 $i0$ $i1$ *x a b*)
from *Lf_eq_negateD this* **show** *?case* **by** *fastforce*
next
case (6 $i0$ $i1$ *ts a b*)
from *Nd_eq_negateD*[*OF* 6(2)] **obtain** *us* **where** *t* = *Nd us ts* = *map* (*negate* (¬ *f*)) *us* **by** *blast*
with 6(1)[*of Not f us*] 6(3) **show** *?case* **by** (*auto simp*: *neg_all_negate'*)
next
case (7 $i0$ $i1$ *i a b*)
then show *?case* **by** *simp*
next
case (8 $i0$ $i1$ *i t ts a b*)
from *Cons_eq_map_D*[*OF* 8(3)] 8(1,4) 8(2)[*OF refl*] **show** *?case*
by (*auto simp*: *Let_def de_morgan_max de_morgan_min uminus_le_reorder*)
qed

lemma *abir'_abir0'*:
shows $abir'$ $i0$ t a b
= $abir0'$ $i0$ (λts a. − $i0$ (map neg_all ts) (−a)) ($negate$ $False$ t) a b
and $abirs'$ $i0$ i ts a b
= $abirs0'$ $i0$ (λts a. − $i0$ (map neg_all ts) (−a)) i (map ($negate$ $True$) ts) a b
proof(*induction* $i0$ *t a b* **and** $i0$ *i ts a b rule*: *abir'_abirs'.induct*)
case (1 $i0$ *x a b*)
then show *?case* **by** *simp*

next
  case (2 *i0 ts a b*)
  then show *?case* **by** *simp*
next
  case (3 *i0 i a b*)
  then show *?case* **by** *simp*
next
  case (4 *i0 i t ts a b*)
  then show *?case*
    by (*auto simp add*: *Let_def abir*01′*_negate*(3) *o_def*)
qed

corollary *fishburn_abir′_negmax_cor*:
  shows $a < b \Longrightarrow$ *bnd i0* (λ*ts a*. $-$ *i0* (*map neg_all ts*) ($-a$)) $\Longrightarrow$ *fishburn a b* (*negmax t*)   (*abir′ i0 t a b*)
  and $a < b \Longrightarrow$ *bnd i0* (λ*ts a*. $-$ *i0* (*map neg_all ts*) ($-a$)) $\Longrightarrow$ *fishburn a b* (*max i* (*negmax* (*Nd ts*))) (*abirs′ i0 i ts a b*)
unfolding *bnd_def*
apply (*metis* (*no_types*, *lifting*) *bnd_def abir′_abir*0′(1) *negmax_maxmin fishburn_abir*01′(1))
by (*smt* (*verit*, *ccfv_threshold*) *bnd_def abir′_abir*0′(2) *negate.simps*(2) *negmax_maxmin fishburn_abir*01′(2))

end

Chapter 3

Distributive Lattices

theory *Alpha_Beta_Lattice*
imports *Alpha_Beta_Linear*
begin

class *distrib_bounded_lattice* = *distrib_lattice* + *bounded_lattice*

instance *bool* :: *distrib_bounded_lattice* **..**
instance *ereal* :: *distrib_bounded_lattice* **..**
instance *set* :: (*type*) *distrib_bounded_lattice* **..**

unbundle *lattice_syntax*

3.1 Game Tree Evaluation

fun *sups* :: ('*a*::*bounded_lattice*) *list* ⇒ '*a* **where**
sups [] = ⊥ |
sups (*x*#*xs*) = *x* ⊔ *sups xs*

fun *infs* :: ('*a*::*bounded_lattice*) *list* ⇒ '*a* **where**
infs [] = ⊤ |
infs (*x*#*xs*) = *x* ⊓ *infs xs*

fun *supinf* :: ('*a*::*distrib_bounded_lattice*) *tree* ⇒ '*a*
and *infsup* :: ('*a*::*distrib_bounded_lattice*) *tree* ⇒ '*a* **where**
supinf (*Lf x*) = *x* |
supinf (*Nd ts*) = *sups* (*map infsup ts*) |
infsup (*Lf x*) = *x* |
infsup (*Nd ts*) = *infs* (*map supinf ts*)

3.2 Distributive Lattices

lemma *sup_inf_assoc*:

$(a::\_::distrib\_lattice) \leq b \Longrightarrow a \sqcup (x \sqcap b) = (a \sqcup x) \sqcap b$
by (*metis inf.orderE inf_sup_distrib2*)

lemma *sup_inf_assoc_iff*:
$(a::\_::distrib\_lattice) \sqcup x \sqcap b = a \sqcup y \sqcap b \longleftrightarrow (a \sqcup x) \sqcap b = (a \sqcup y) \sqcap b$
by (*metis* (*no_types*, *opaque_lifting*) *inf.left_idem inf_commute inf_sup_distrib1 sup.left_idem sup_inf_distrib1*)

Generalization of Knuth and Moore's equivalence modulo:

abbreviation
eq_mod :: $('a::lattice) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ (‹(_ ≃/ _/ '(*mod* _,_'))› [51,51,0,0]) **where**
$eq\_mod\ x\ y\ a\ b \equiv a \sqcup x \sqcap b = a \sqcup y \sqcap b$

notation (*latex* **output**) *eq_mod* (‹(_ ≃/ _/ '(mod _,_'))› [51,51,0,0])

ab is bounded by v mod a,b, or the other way around.

abbreviation $bounded\ (a::\_::lattice)\ b\ v\ ab \equiv b \sqcap v \leq ab \wedge ab \leq a \sqcup v$

abbreviation *bounded2* :: $('a::lattice) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ (‹(_ ⊑/ _/ '(*mod* _,_'))› [51,51,0,0])
where $bounded2\ ab\ v\ a\ b \equiv bounded\ a\ b\ v\ ab$

notation (*latex* **output**) *bounded2* (‹(_ ⊑/ _/ '(mod _,_'))› [51,51,0,0])

lemma *bounded_bot_top*:
fixes $v\ ab$ :: *'a::distrib_bounded_lattice*
shows $bounded\ \bot\ \top\ v\ ab \Longrightarrow ab = v$
by (*simp add*: *order_eq_iff*)

bounded implies eq-mod, but not the other way around:

bounded implies eq-mod:

lemma *eq_mod_if_bounded*: **assumes** *bounded a b v ab*
shows $a \sqcup ab \sqcap b = a \sqcup v \sqcap (b::\_::distrib\_lattice)$
proof(*rule antisym*)
 have $a \leq a \sqcup v \sqcap b$ **by** *simp*
 moreover have $ab \sqcap b \leq a \sqcup v \sqcap b$
 proof −
  have $ab \sqcap b \leq (a \sqcup v) \sqcap b$ **by**(*fact inf_mono*[*OF conjunct2*[*OF assms*] *order.refl*])
  also have $\ldots = a \sqcap b \sqcup v \sqcap b$ **by** (*fact inf_sup_distrib2*)
  also have $\ldots \leq a \sqcup v \sqcap b$ **by** (*fact sup.mono*[*OF inf.cobounded1 order.refl*])
  finally show *?thesis* **.**
 qed
 ultimately show $a \sqcup ab \sqcap b \leq a \sqcup v \sqcap b$ **by**(*metis sup.bounded_iff*)
next
 have $a \leq a \sqcup ab \sqcap b$ **by** *simp*
 moreover have $v \sqcap b \leq a \sqcup ab \sqcap b$
 proof −

have $v \sqcap b = (v \sqcap b) \sqcap b$ **by** *simp*
also have $\ldots \le ab \sqcap b$ **by**(*metis inf.commute inf.mono*[*OF conjunct1*[*OF assms*] *order.refl*])
also have $\ldots \le a \sqcup ab \sqcap b$ **by** *simp*
finally show *?thesis* **.**
qed
ultimately show $a \sqcup v \sqcap b \le a \sqcup ab \sqcap b$ **by**(*metis sup.bounded_iff*)
qed

Converse is not true, even for *linorder*, even if $a < b$:

lemma *let* a=0; b=1; ab=2; v=1
in $a \sqcup ab \sqcap b = a \sqcup v \sqcap (b{::}nat) \wedge \neg(b \sqcap v \le ab \wedge ab \le a \sqcup v)$
by *eval*

Because for *linord* we have: $bounded = fishburn$ $(a < b \Longrightarrow ab \le v \pmod{a,b} = (min\ v\ b \le ab \wedge ab \le max\ v\ a))$ and $eq\_mod = knuth$ $(a < b \Longrightarrow (max\ a\ (min\ x\ b) = max\ a\ (min\ y\ b)) = y \cong x \pmod{a,b})$ but we know *fishburn* is stronger than *knuth*.

These equivalences do not even hold as implications in *distrib_lattice*, even if $a < b$. (We need to redefine *knuth* and *fishburn* for *distrib_lattice* first)

context
begin

definition
$knuth'$ (a::_::*distrib_lattice*) $b\ x\ y$ ==
$((y \le a \longrightarrow x \le a) \wedge (a < y \wedge y < b \longrightarrow y = x) \wedge (b \le y \longrightarrow b \le x))$

lemma *let* a={}; b={1::*int*}; ab={}; v={0}
in $\neg\ (a \sqcup ab \sqcap b = a \sqcup v \sqcap b \longrightarrow knuth'\ a\ b\ v\ ab)$
by *eval*

lemma *let* a={}; b={1::*int*}; ab={0}; v={1}
in $\neg\ (knuth'\ a\ b\ v\ ab \longrightarrow a \sqcup ab \sqcap b = a \sqcup v \sqcap b)$
by *eval*

definition
$fishburn'$ (a::_::*distrib_lattice*) $b\ v\ ab$ ==
$((ab \le a \longrightarrow v \le ab) \wedge (a < ab \wedge ab < b \longrightarrow ab = v) \wedge (b \le ab \longrightarrow ab \le v))$

Same counterexamples as above:

lemma *let* a={}; b={1::*int*}; ab={}; v={0}
in $\neg\ (bounded\ a\ b\ v\ ab \longrightarrow fishburn'\ a\ b\ v\ ab)$
by *eval*

lemma *let* a={}; b={1::*int*}; ab={0}; v={1}
in $\neg\ (fishburn'\ a\ b\ v\ ab \longrightarrow bounded\ a\ b\ v\ ab)$
by *eval*

end

3.2.1 Fail-Hard

Basic *ab_sup*

Improved version of Bird and Hughes. No squashing in base case.

fun *ab_sup* :: $'a \Rightarrow 'a \Rightarrow ('a::distrib\_lattice)tree \Rightarrow 'a$ **and** *ab_sups* **and** *ab_inf* **and** *ab_infs* **where**
ab_sup a b (*Lf x*) = *x* |
ab_sup a b (*Nd ts*) = *ab_sups a b ts* |
ab_sups a b [] = *a* |
ab_sups a b (*t#ts*) = (*let* $a' = a \sqcup$ *ab_inf a b t in if* $a' \geq b$ *then* a' *else ab_sups* a' *b ts*) |
ab_inf a b (*Lf x*) = *x* |
ab_inf a b (*Nd ts*) = *ab_infs a b ts* |
ab_infs a b [] = *b* |
ab_infs a b (*t#ts*) = (*let* $b' = b \sqcap$ *ab_sup a b t in if* $b' \leq a$ *then* b' *else ab_infs a* b' *ts*)

lemma *ab_sups_ge_a*: *ab_sups a b ts* $\geq a$
apply(*induction ts arbitrary*: *a*)
by (*auto simp*: *Let_def*)(*use le_sup_iff* **in** *blast*)

lemma *ab_infs_le_b*: *ab_infs a b ts* $\leq b$
apply(*induction ts arbitrary*: *b*)
by (*auto simp*: *Let_def*)(*use le_inf_iff* **in** *blast*)

lemma *eq_mod_ab_val_auto*:
shows $a \sqcup$ *ab_sup a b t* $\sqcap b = a \sqcup$ *supinf t* $\sqcap b$
and $a \sqcup$ *ab_sups a b ts* $\sqcap b = a \sqcup$ *supinf* (*Nd ts*) $\sqcap b$
and $a \sqcup$ *ab_inf a b t* $\sqcap b = a \sqcup$ *infsup t* $\sqcap b$
and $a \sqcup$ *ab_infs a b ts* $\sqcap b = a \sqcup$ *infsup* (*Nd ts*) $\sqcap b$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup_ab_sups_ab_inf_ab_infs.induct*)
  case (4 *a b t ts*)
  then show *?case*
    apply(*simp add*: *Let_def*)
   by (*smt* (*verit*, *ccfv_threshold*) *ab_sups_ge_a inf.absorb_iff2 inf_left_commute inf_sup_distrib2 sup.left_idem sup_absorb1 sup_absorb2 sup_assoc sup_inf_assoc_iff*)
next
  case (8 *a b t ts*)
  then show *?case*
    apply(*simp add*: *Let_def*)
   by (*smt* (*verit*) *ab_infs_le_b inf.absorb_iff2 inf_assoc inf_commute inf_right_idem sup.absorb1 sup_inf_distrib1*)
qed *auto*

A readable proof. Some steps still tricky. Complication: sometimes $a \sqcup x \sqcap b$ is better and sometimes $(a \sqcup x) \sqcap b$.

lemma *eq_mod_ab_val*:
shows $a \sqcup$ *ab_sup* $a\ b\ t \sqcap b = a \sqcup$ *supinf* $t \sqcap b$
and $a \sqcup$ *ab_sups* $a\ b$ *ts* $\sqcap b = a \sqcup$ *supinf* (*Nd ts*) $\sqcap b$
and $a \sqcup$ *ab_inf* $a\ b\ t \sqcap b = a \sqcup$ *infsup* $t \sqcap b$
and $a \sqcup$ *ab_infs* $a\ b$ *ts* $\sqcap b = a \sqcup$ *infsup* (*Nd ts*) $\sqcap b$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup_ab_sups_ab_inf_ab_infs.induct*)
  case (8 *a b t ts*)
  let *?abt* = *ab_sup a b t* **let** *?abts* = *ab_infs a* ($b \sqcap$ *?abt*) *ts*
  let *?vt* = *supinf t* **let** *?vts* = *infsup* (*Nd ts*)
  show *?case*
  proof (*cases* $b \sqcap$ *?abt* $\leq a$)
    case *True*
      hence *b*: $a \sqcup$ *?vt* $\sqcap b = a$ **using** 8.*IH*(1) *True* **by** (*metis sup_absorb*1 *inf_commute*)
    have $a \sqcup$ *ab_infs a b* (*t#ts*) $\sqcap b = a \sqcup b \sqcap$ *?abt* $\sqcap b$ **using** *True* **by** (*simp*)
    also have ... = $a \sqcup$ *?abt* $\sqcap b$ **by** (*simp add*: *inf_commute*)
    also have ... = $a \sqcup$ *?vt* $\sqcap b$ **by** (*simp add*: 8.*IH*(1))
    also have ... = $a \sqcup$ (*?vt* $\sqcup$ *?vt* $\sqcap$ *?vts*) $\sqcap b$ **by** (*simp*)
    also have ... = $a \sqcup$ (*?vt* $\sqcap b \sqcup$ *?vt* $\sqcap$ *?vts* $\sqcap b$) **by** (*metis inf_sup_distrib*2)
    also have ... = $a \sqcup$ *?vt* $\sqcap b \sqcup$ *?vt* $\sqcap$ *?vts* $\sqcap b$ **by** (*metis sup_assoc*)
    also have ... = $a \sqcup$ *?vt* $\sqcap$ *?vts* $\sqcap b$ **by** (*metis b*)
    also have ... = $a \sqcup$ *infsup* (*Nd* (*t # ts*)) $\sqcap b$ **by** (*simp*)
    finally show *?thesis* **.**
  next
    case *False*
    from 8.*IH*(2)[*OF refl False*] *ab_infs_le_b*
    have *IH*2′: $a \sqcup$ *?abts* $\sqcap b = a \sqcup$ *?vts* $\sqcap$ *?abt* $\sqcap b$
      by (*metis* (*no_types*, *lifting*) *inf_absorb*1 *inf_assoc inf_commute inf_idem*)
    have $a \sqcup$ *ab_infs a b* (*t#ts*) $\sqcap b = a \sqcup$ *?abts* $\sqcap b$ **using** *False* **by** (*simp*)
    also have ... = $a \sqcup$ *?abt* $\sqcap$ *?vts* $\sqcap b$ **using** *IH*2′ **by** (*metis inf_commute*)
    also have ... = $a \sqcup$ *?vt* $\sqcap$ *?vts* $\sqcap b$ **using** 8.*IH*(1)
      by (*metis* (*no_types*, *lifting*) *inf_assoc inf_commute sup_inf_distrib*1)
    also have ... = $a \sqcup$ *infsup* (*Nd* (*t # ts*)) $\sqcap b$ **by** (*simp*)
    finally show *?thesis* **.**
  qed
next
  case (4 *a b t ts*)
  let *?abt* = *ab_inf a b t* **let** *?abts* = *ab_sups* ($a \sqcup$ *?abt*) *b ts*
  let *?vt* = *infsup t* **let** *?vts* = *supinf* (*Nd ts*)
  show *?case*
  proof (*cases* $b \leq a \sqcup$ *?abt*)
    case *True*
     have *IH*1′: ‹($a \sqcup$ *?abt*) $\sqcap b$ = ($a \sqcup$ *?vt*) $\sqcap b$› **by**(*metis sup_inf_assoc_iff* 4.*IH*(1))
    hence *b*: ($a \sqcup$ *?vt*) $\sqcap b = b$ **using** *True inf_absorb*2 **by** *metis*
     have ($a \sqcup$ *ab_sups a b* (*t#ts*)) $\sqcap b$ = ($a \sqcup$ ($a \sqcup$ *?abt*)) $\sqcap b$ **using** *True* **by** (*simp*)
    also have ... = ($a \sqcup$ *?abt*) $\sqcap b$ **by** (*simp*)
    also have ... = ($a \sqcup$ *?vt*) $\sqcap b$ **by** (*simp add*: *IH*1′)

also have $\ldots = (a \sqcup ?vt \sqcup ?vts) \sqcap (a \sqcup ?vt) \sqcap b$ **by** (*simp add*: *inf.absorb2*)
also have $\ldots = (a \sqcup ?vt \sqcup ?vts) \sqcap b$ **by** (*simp add*: *b inf_assoc*)
also have $\ldots = (a \sqcup \mathit{supinf}\ (\mathit{Nd}\ (t\ \#\ ts))) \sqcap b$ **by** (*simp add*: *sup.assoc*)
finally show *?thesis*
using *sup_inf_assoc_iff* **by** *blast*
next
case *False*
from $4.\mathit{IH}(2)[\mathit{OF\ refl\ False}]$ *ab_sups_ge_a*
have $\mathit{IH2}'$: $(a \sqcup ?abts) \sqcap b = (a \sqcup ?abt \sqcup ?vts) \sqcap b$
by (*smt* (*verit*, *best*) *le_sup_iff sup_absorb2 sup_inf_assoc_iff*)
have $(a \sqcup \mathit{ab\_sups}\ a\ b\ (t\#ts)) \sqcap b = (a \sqcup ?abts) \sqcap b$ **using** *False* **by** (*simp*)
also have $\ldots = (a \sqcup ?abt \sqcup ?vts) \sqcap b$ **using** $\mathit{IH2}'$ **by** *blast*
also have $\ldots = a \sqcap b \sqcup ?abt \sqcap b \sqcup ?vts \sqcap b$ **by** (*simp add*: *inf_sup_distrib2*)
also have $\ldots = (a \sqcup ?abt \sqcap b) \sqcap b \sqcup ?vts \sqcap b$ **by** (*metis inf_sup_distrib2 inf.right_idem*)
also have $\ldots = (a \sqcup ?vt \sqcap b) \sqcap b \sqcup ?vts \sqcap b$ **using** $4.\mathit{IH}(1)$ **by** *simp*
also have $\ldots = (a \sqcup ?vt \sqcup ?vts) \sqcap b$ **by** (*simp add*: *inf_sup_distrib2*)
also have $\ldots = (a \sqcup \mathit{supinf}\ (\mathit{Nd}\ (t\ \#\ ts))) \sqcap b$ **by** (*simp add*: *sup_assoc*)
finally show *?thesis*
using *sup_inf_assoc_iff* **by** *blast*
qed
qed (*simp_all*)

corollary *ab_sup_bot_top*: $\mathit{ab\_sup}\ \bot\ \top\ t = \mathit{supinf}\ t$
by (*metis eq_mod_ab_val*(1) *inf_top_right sup_bot.left_neutral*)

Predicate *knuth* (and thus *fishburn*) does not hold:

lemma *let* $a = \{\mathit{False}\}$; $b = \{\mathit{False},\ \mathit{True}\}$; $t = \mathit{Nd}\ [\mathit{Lf}\ \{\mathit{True}\}]$;
$ab = \mathit{ab\_sup}\ a\ b\ t$; $v = \mathit{supinf}\ t$ *in* $v = \{\mathit{True}\} \wedge ab = \{\mathit{True},\mathit{False}\} \wedge b \le ab \wedge \neg\ b \le v$
by *eval*

Worse: *fishburn* (and *knuth*) only caters for a "linear" analysis where *ab* lies wrt $a < b$. But *ab* may not satisfy either of the 3 alternatives in *fishburn*:

lemma *let* $a = \{\}$; $b = \{\mathit{True}\}$; $t = \mathit{Nd}\ [\mathit{Lf}\ \{\mathit{False}\}]$; $ab = \mathit{ab\_sup}\ a\ b\ t$; $v = \mathit{supinf}\ t$ *in*
$v = \{\mathit{False}\} \wedge ab{=}\{\mathit{False}\} \wedge \neg\ ab{\le}a \wedge \neg\ ab{\ge}b \wedge \neg\ (a{<}ab \wedge ab < b)$
by *eval*

A stronger correctness property

The stronger correctness property *bounded*:

lemma
shows *bounded a b* (*supinf t*) (*ab_sup a b t*)
and *bounded a b* (*supinf* (*Nd ts*)) (*ab_sups a b ts*)
and *bounded a b* (*infsup t*) (*ab_inf a b t*)
and *bounded a b* (*infsup* (*Nd ts*)) (*ab_infs a b ts*)
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup_ab_sups_ab_inf_ab_infs.induct*)
case (4 *a b t ts*)

thus *?case*
  apply(*simp add*: *Let_def inf.coboundedI*1 *sup.coboundedI*1)
  by (*smt* (*verit*, *best*) *ab_sups_ge_a inf_sup_distrib*1 *sup.absorb_iff*2 *sup_assoc sup_commute*)
next
  case (8 *t ts a b*)
  thus *?case*
  apply(*simp add*: *Let_def inf.coboundedI*1 *sup.coboundedI*1)
  by (*smt* (*verit*) *ab_infs_le_b inf.absorb_iff*2 *inf_commute inf_left_commute sup_inf_distrib*1)
qed *auto*

lemma *bounded_val_ab*:
shows *bounded a b* (*supinf t*) (*ab_sup a b t*)
and *bounded a b* (*supinf* (*Nd ts*)) (*ab_sups a b ts*)
and *bounded a b* (*infsup t*) (*ab_inf a b t*)
and *bounded a b* (*infsup* (*Nd ts*)) (*ab_infs a b ts*)
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup_ab_sups_ab_inf_ab_infs.induct*)
  case (4 *a b t ts*)
  let $?abt = ab\_inf\ a\ b\ t$ **let** $?abts = ab\_sups\ (a \sqcup ?abt)\ b\ ts$
  let $?vt = infsup\ t$ **let** $?vts = sups\ (map\ infsup\ ts)$
  have $b \sqcap supinf\ (Nd\ (t\ \#\ ts)) \le ab\_sups\ a\ b\ (t\ \#\ ts)$
  proof −
    have $b \sqcap supinf\ (Nd\ (t\ \#\ ts)) = b \sqcap (?vt \sqcup ?vts)$ **by**(*simp*)
    also have $\ldots = b \sqcap ?vt \sqcup b \sqcap ?vts$ **by** (*fact inf_sup_distrib*1)
    also have $\ldots \le ?abt \sqcup b \sqcap ?vts$ **using** 4.*IH*(1) **by** (*metis order.refl sup.mono*)
    also have $\ldots \le ab\_sups\ a\ b\ (t\ \#\ ts)$
    proof *cases*
      assume $b \le a \sqcup ?abt$
      have $?abt \sqcup b \sqcap ?vts \le a \sqcup ?abt \sqcup b \sqcap ?vts$ **by** (*simp add*: *sup_assoc*)
      also have $\ldots = a \sqcup ?abt$ **using** ‹$b \le a \sqcup ?abt$› **by** (*meson le_infI*1 *sup.absorb*1)
      also have $\ldots = ab\_sups\ a\ b\ (t\ \#\ ts)$ **using** ‹$b \le a \sqcup ?abt$› **by** *simp*
      finally show *?thesis* **.**
    next
      assume $\neg\ b \le a \sqcup ?abt$
      have $?abt \sqcup b \sqcap ?vts \le ?abt \sqcup ?abts$
        using 4.*IH*(2)[*OF refl* ‹$\neg\ b \le a \sqcup ?abt$›] *sup.mono* **by** *auto*
      also have $\ldots \le ?abts$ **by** (*meson ab_sups_ge_a le_sup_iff order_refl*)
      also have $\ldots = ab\_sups\ a\ b\ (t\ \#\ ts)$ **using** ‹$\neg\ b \le a \sqcup ?abt$› **by** *simp*
      finally show *?thesis* **.**
    qed
    finally show *?thesis* **.**
  qed
  moreover
  have $ab\_sups\ a\ b\ (t\ \#\ ts) \le a \sqcup supinf\ (Nd\ (t\ \#\ ts))$
  proof *cases*
    assume $b \le a \sqcup ?abt$
    then have $ab\_sups\ a\ b\ (t\ \#\ ts) = a \sqcup ?abt$ **by**(*simp add*: *Let_def*)
    also have $\ldots \le a \sqcup ?vt$ **using** 4.*IH*(1) **by** *simp*

also have ... $\leq a \sqcup ?vt \sqcup ?vts$ **by** *simp*
also have ... $= a \sqcup supinf\ (Nd\ (t\ \#\ ts))$ **by** (*simp add*: *sup_assoc*)
finally show *?thesis* **.**
next
assume $\neg\ b \leq a \sqcup ?abt$
then have *ab_sups a b* $(t\ \#\ ts) = ?abts$ **by**(*simp add*: *Let_def*)
also have ... $\leq a \sqcup ?abt \sqcup ?vts$ **using** 4.*IH*(2)[*OF refl* ‹$\neg\ b \leq a \sqcup ?abt$›] **by** *simp*
also have ... $\leq a \sqcup ?vt \sqcup ?vts$ **using** 4.*IH*(1)
by (*metis sup.mono inf_sup_absorb le_inf_iff sup.cobounded2 sup.idem*)
also have ... $= a \sqcup supinf\ (Nd\ (t\ \#\ ts))$ **by** (*simp add*: *sup_assoc*)
finally show *?thesis* **.**
qed
ultimately show *?case* **..**
next
case 8 **thus** *?case*
apply(*simp add*: *Let_def*)
by (*smt* (*verit*) *ab_infs_le_b inf.absorb_iff2 inf.cobounded2 inf.orderE inf_assoc inf_idem sup.coboundedI1 sup_inf_distrib1*)
qed *auto*

Bird and Hughes

fun *ab_sup2* :: (*'a::distrib_lattice*) $\Rightarrow$ *'a* $\Rightarrow$ *'a tree* $\Rightarrow$ *'a* **and** *ab_sups2* **and** *ab_inf2* **and** *ab_infs2* **where**
ab_sup2 a b $(Lf\ x) = a \sqcup x \sqcap b$ |
ab_sup2 a b $(Nd\ ts) =$ *ab_sups2 a b ts* |

ab_sups2 a b $[] = a$ |
ab_sups2 a b $(t\#ts) =$ (*let* $a' =$ *ab_inf2 a b t in if* $a' = b$ *then b else ab_sups2* a' *b ts*) |

ab_inf2 a b $(Lf\ x) = (a \sqcup x) \sqcap b$ |
ab_inf2 a b $(Nd\ ts) =$ *ab_infs2 a b ts* |

ab_infs2 a b $[] = b$ |
ab_infs2 a b $(t\#ts) =$ (*let* $b' =$ *ab_sup2 a b t in if* $a = b'$ *then a else ab_infs2 a* b' *ts*)

lemma *eq_mod_ab2_val*:
shows $a \leq b \Longrightarrow$ *ab_sup2 a b t* $= a \sqcup (supinf\ t \sqcap b)$
and $a \leq b \Longrightarrow$ *ab_sups2 a b ts* $= a \sqcup (supinf\ (Nd\ ts) \sqcap b)$
and $a \leq b \Longrightarrow$ *ab_inf2 a b t* $= (a \sqcup infsup\ t) \sqcap b$
and $a \leq b \Longrightarrow$ *ab_infs2 a b ts* $= (a \sqcup infsup(Nd\ ts)) \sqcap b$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup2_ab_sups2_ab_inf2_ab_infs2.induct*)
case 4 **thus** *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit, best*) *inf_commute inf_sup_distrib2 sup_assoc sup_inf_absorb sup_inf_assoc*)

next
case 8 **thus** *?case*
apply (*simp add*: *Let_def*)
by (*smt* (*verit*, *del_insts*) *inf_assoc inf_commute inf_sup_absorb sup_inf_assoc sup_inf_distrib1*)
qed *simp_all*

corollary *ab_sup2_bot_top*: $ab\_sup2\ \bot\ \top\ t = supinf\ t$
using *eq_mod_ab2_val*(1)[*of* $\bot$ $\top$] **by** *simp*

Simpler proof with sets; not really surprising.

lemma *ab_sup2_bounded_set*:
shows $a \le (b{::}\ \_\ set) \Longrightarrow ab\_sup2\ a\ b\ t = a \sqcup (supinf\ t \sqcap b)$
and $a \le b \Longrightarrow ab\_sups2\ a\ b\ ts = a \sqcup (supinf\ (Nd\ ts) \sqcap b)$
and $a \le b \Longrightarrow ab\_inf2\ a\ b\ t = (a \sqcup infsup\ t) \sqcap b$
and $a \le b \Longrightarrow ab\_infs2\ a\ b\ ts = (a \sqcup infsup(Nd\ ts)) \sqcap b$
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup2_ab_sups2_ab_inf2_ab_infs2.induct*)
qed (*auto simp*: *Let_def*)

Delayed Test

fun *ab_sup3* :: ('*a*::*distrib_lattice*) ⇒ '*a* ⇒ '*a tree* ⇒ '*a* **and** *ab_sups3* **and** *ab_inf3*
and *ab_infs3* **where**
$ab\_sup3\ a\ b\ (Lf\ x) = x\ |$
$ab\_sup3\ a\ b\ (Nd\ ts) = ab\_sups3\ a\ b\ ts\ |$

$ab\_sups3\ a\ b\ [] = a\ |$
$ab\_sups3\ a\ b\ (t\#ts) = (if\ a \ge b\ then\ a\ else\ ab\_sups3\ (a \sqcup ab\_inf3\ a\ b\ t)\ b\ ts)\ |$

$ab\_inf3\ a\ b\ (Lf\ x) = x\ |$
$ab\_inf3\ a\ b\ (Nd\ ts) = ab\_infs3\ a\ b\ ts\ |$

$ab\_infs3\ a\ b\ [] = b\ |$
$ab\_infs3\ a\ b\ (t\#ts) = (if\ a \ge b\ then\ b\ else\ ab\_infs3\ a\ (b \sqcap ab\_sup3\ a\ b\ t)\ ts)$

lemma *ab_sups3_ge_a*: $ab\_sups3\ a\ b\ ts \ge a$
apply(*induction ts arbitrary*: *a*)
by (*auto simp*: *Let_def*)(*use le_sup_iff* **in** *blast*)

lemma *ab_infs3_le_b*: $ab\_infs3\ a\ b\ ts \le b$
apply(*induction ts arbitrary*: *b*)
by (*auto simp*: *Let_def*)(*use le_inf_iff* **in** *blast*)

lemma *ab_sup3_ab_sup*:
shows $a<b \Longrightarrow ab\_sup3\ a\ b\ t = ab\_sup\ a\ b\ t$
and $a<b \Longrightarrow ab\_sups3\ a\ b\ ts = ab\_sups\ a\ b\ ts$
and $a<b \Longrightarrow ab\_inf3\ a\ b\ t = ab\_inf\ a\ b\ t$
and $a<b \Longrightarrow ab\_infs3\ a\ b\ ts = ab\_infs\ a\ b\ ts$

quickcheck[*expect=no_counterexample*]
oops

Bird and Hughes plus delayed test

fun *ab_sup4* :: (*'a*::*distrib_lattice*) ⇒ *'a* ⇒ *'a tree* ⇒ *'a* **and** *ab_sups4* **and** *ab_inf4*
and *ab_infs4* **where**
ab_sup4 a b (*Lf x*) = *a* ⊔ *x* ⊓ *b* |
ab_sup4 a b (*Nd ts*) = *ab_sups4 a b ts* |

ab_sups4 a b [] = *a* |
ab_sups4 a b (*t#ts*) = (*if a* = *b then a else ab_sups4* (*ab_inf4 a b t*) *b ts*) |

ab_inf4 a b (*Lf x*) = (*a* ⊔ *x*) ⊓ *b* |
ab_inf4 a b (*Nd ts*) = *ab_infs4 a b ts* |

ab_infs4 a b [] = *b* |
ab_infs4 a b (*t#ts*) = (*if a* = *b then b else ab_infs4 a* (*ab_sup4 a b t*) *ts*)

lemma *ab_sup4_bounded*:
shows *a*≤*b* ⟹ *ab_sup4 a b t* = *a* ⊔ (*supinf t* ⊓ *b*)
and *a*≤*b* ⟹ *ab_sups4 a b ts* = *a* ⊔ (*supinf* (*Nd ts*) ⊓ *b*)
and *a*≤*b* ⟹ *ab_inf4 a b t* = (*a* ⊔ *infsup t*) ⊓ *b*
and *a*≤*b* ⟹ *ab_infs4 a b ts* = (*a* ⊔ *infsup*(*Nd ts*)) ⊓ *b*
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup4_ab_sups4_ab_inf4_ab_infs4.induct*)
  case 3 **thus** *?case* **by** (*simp add*: *sup_absorb1*)
next
  case 4 **thus** *?case*
    apply (*simp add*: *sup_absorb1*)
    by (*metis* (*no_types*, *lifting*) *inf_sup_distrib2 sup_assoc sup_inf_assoc*)
next
  case 7 **thus** *?case* **by** (*simp add*: *inf.absorb2*)
next
  case (8 *a b t ts*)
  then show *?case*
  apply (*simp add*: *inf.absorb2*)
  by (*simp add*: *inf_assoc inf_commute sup_absorb2 sup_inf_distrib1*)
qed *simp_all*

lemma *ab_sup4_bounded_set*:
shows *a*≤(*b*:: _ *set*) ⟹ *ab_sup4 a b t* = *a* ⊔ (*supinf t* ⊓ *b*)
and *a*≤*b* ⟹ *ab_sups4 a b ts* = *a* ⊔ (*supinf* (*Nd ts*) ⊓ *b*)
and *a*≤*b* ⟹ *ab_inf4 a b t* = (*a* ⊔ *infsup t*) ⊓ *b*
and *a*≤*b* ⟹ *ab_infs4 a b ts* = (*a* ⊔ *infsup*(*Nd ts*)) ⊓ *b*
by (*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup4_ab_sups4_ab_inf4_ab_infs4.induct*)
   auto

3.2.2 Fail-Soft

fun *ab_sup′* :: *′a::distrib_bounded_lattice* ⇒ *′a* ⇒ *′a tree* ⇒ *′a* **and** *ab_sups′*
ab_inf′ ab_infs′ **where**
ab_sup′ a b (*Lf x*) = *x* |
ab_sup′ a b (*Nd ts*) = *ab_sups′ a b* ⊥ *ts* |

ab_sups′ a b m [] = *m* |
ab_sups′ a b m (*t#ts*) =
 (*let m′* = *m* ⊔ (*ab_inf′* (*m* ⊔ *a*) *b t*) *in if m′* ≥ *b then m′ else ab_sups′ a b m′*
ts) |

ab_inf′ a b (*Lf x*) = *x* |
ab_inf′ a b (*Nd ts*) = *ab_infs′ a b* ⊤ *ts* |

ab_infs′ a b m [] = *m* |
ab_infs′ a b m (*t#ts*) =
 (*let m′* = *m* ⊓ (*ab_sup′ a* (*m* ⊓ *b*) *t*) *in if m′* ≤ *a then m′ else ab_infs′ a b m′*
ts)

lemma *ab_sups′_ge_m*: *ab_sups′ a b m ts* ≥ *m*
apply(*induction ts arbitrary*: *a b m*)
by (*auto simp*: *Let_def*)(*use le_sup_iff* **in** *blast*)

lemma *ab_infs′_le_m*: *ab_infs′ a b m ts* ≤ *m*
apply(*induction ts arbitrary*: *a b m*)
by (*auto simp*: *Let_def*)(*use le_inf_iff* **in** *blast*)

Fail-soft correctness:

lemma *bounded_val_ab′*:
shows *bounded* (*a*) *b* (*supinf t*) (*ab_sup′ a b t*)
and *bounded* (*a* ⊔ *m*) *b* (*supinf* (*Nd ts*)) (*ab_sups′ a b m ts*)
and *bounded a b* (*infsup t*) (*ab_inf′ a b t*)
and *bounded a* (*b* ⊓ *m*) (*infsup* (*Nd ts*)) (*ab_infs′ a b m ts*)
proof(*induction a b t* **and** *a b m ts* **and** *a b t* **and** *a b m ts rule*: *ab_sup′_ab_sups′_ab_inf′_ab_infs′.induct*)
 case (4 *a b m t ts*)
 then show *?case*
  apply(*simp add*: *Let_def inf.coboundedI1 sup.coboundedI1*)
  by (*smt* (*verit*) *ab_sups′_ge_m inf_sup_distrib1 sup.absorb_iff1 sup_commute*
sup_left_commute)
next
 case (8 *a b m t ts*)
 then show *?case*
  apply(*simp add*: *Let_def inf.coboundedI1 sup.coboundedI1*)
  by (*smt* (*verit*) *ab_infs′_le_m inf.absorb_iff2 inf_assoc inf_left_commute*
sup_inf_distrib1)
qed *auto*

corollary *ab_sup′* ⊥ ⊤ *t* = *supinf t*

by (*rule bounded_bot_top*[*OF bounded_val_ab*′(1)])

lemma *eq_mod_ab*′*_val*:
shows $a \sqcup$ *ab_sup*′ *a b t* $\sqcap b = a \sqcup$ *supinf t* $\sqcap b$
and $(m \sqcup a) \sqcup$ *ab_sups*′ *a b m ts* $\sqcap b = (m \sqcup a) \sqcup$ *supinf* (*Nd ts*) $\sqcap b$
and $a \sqcup$ *ab_inf*′ *a b t* $\sqcap b = a \sqcup$ *infsup t* $\sqcap b$
and $a \sqcup$ *ab_infs*′ *a b m ts* $\sqcap (m \sqcap b) = a \sqcup$ *infsup* (*Nd ts*) $\sqcap (m \sqcap b)$
  apply (*meson bounded_val_ab*′(1) *eq_mod_if_bounded*)
  apply (*metis bounded_val_ab*′(2) *eq_mod_if_bounded sup_commute*)
  apply (*meson bounded_val_ab*′(3) *eq_mod_if_bounded*)
  by (*metis bounded_val_ab*′(4) *eq_mod_if_bounded inf_commute*)

lemma *ab_sups*′*_le_ab_sups*: *ab_sups*′ *a b c t* $\sqcap b \leq$ *ab_sups* $(a \sqcup c)$ *b t*
by (*smt* (*verit*, *best*) *ab_sups_ge_a bounded_val_ab*(2) *eq_mod_ab*′*_val*(2) *inf_commute sup.absorb_iff2 sup_assoc sup_commute*)

lemma *ab_sup*′*_le_ab_sup*: *ab_sup*′ *a b t* $\sqcap b \leq$ *ab_sup a b t*
by (*metis ab_sup*′*.elims ab_sup.simps*(1) *ab_sup.simps*(2) *ab_sups*′*_le_ab_sups inf.cobounded1 sup_bot_right*)

Towards *bounded* of Fail-Soft

theorem *bounded_ab*′*_ab*:
  shows *bounded* (*a*) *b* (*ab_sup*′ *a b t*)   (*ab_sup a b t*)
    and *bounded a b* (*ab_sups*′ *a b m ts*) (*ab_sups* (*sup m a*) *b ts*)
    and *bounded a b* (*ab_inf*′ *a b t*)   (*ab_inf a b t*)
    and *bounded a b* (*ab_infs*′ *a b m ts*) (*ab_infs a* (*inf m b*) *ts*)
oops

3.3 De Morgan Algebras

Now: also negation. But still not a boolean algebra but only a De Morgan algebra:

class *de_morgan_algebra* = *distrib_bounded_lattice* + *uminus*
opening *lattice_syntax* +
assumes *de_Morgan_inf*: $-(x \sqcap y) = -x \sqcup -y$
assumes *neg_neg*[*simp*]: $-(-x) = x$
begin

lemma *de_Morgan_sup*: $-(x \sqcup y) = -x \sqcap -y$
by (*metis de_Morgan_inf neg_neg*)

lemma *neg_top*[*simp*]: $-\top = \bot$
by (*metis bot_eq_sup_iff de_Morgan_inf inf_top.right_neutral neg_neg*)

lemma *neg_bot*[*simp*]: $- \bot = \top$
using *neg_neg neg_top* **by** *blast*

lemma *uminus_eq_iff*[*simp*]: $-a = -b \longleftrightarrow a = b$
by (*metis neg_neg*)

lemma *uminus_le_reorder*: $(- a \leq b) = (- b \leq a)$
by (*metis de_Morgan_sup inf.absorb_iff2 le_iff_sup neg_neg*)

lemma *uminus_less_reorder*: $(- a < b) = (- b < a)$
by (*metis order.strict_iff_not neg_neg uminus_le_reorder*)

lemma *minus_le_minus*[*simp*]: $- a \leq - b \longleftrightarrow b \leq a$
by (*metis neg_neg uminus_le_reorder*)

lemma *minus_less_minus*[*simp*]: $- a < - b \longleftrightarrow b < a$
by (*metis neg_neg uminus_less_reorder*)

lemma *less_uminus_reorder*: $a < - b \longleftrightarrow b < - a$
by (*metis neg_neg uminus_less_reorder*)

end

instantiation *ereal* :: *de_morgan_algebra*
begin

instance
proof (*standard*, *goal_cases*)
  case 1
  thus *?case* **by** (*simp add*: *min_def*)
next
  case 2
  thus *?case* **by** (*simp*)
qed

end

instantiation *set* :: (*type*)*de_morgan_algebra*
begin

instance
proof (*standard*, *goal_cases*)
  case 1
  thus *?case* **by** (*simp*)
next
  case 2
  thus *?case* **by** (*simp*)
qed

end

fun negsup :: ('a :: de_morgan_algebra)tree ⇒ 'a **where**
negsup (Lf x) = x |
negsup (Nd ts) = sups (map (λt. − negsup t) ts)

fun negate :: bool ⇒ ('a::de_morgan_algebra) tree ⇒ 'a tree **where**
negate b (Lf x) = Lf (if b then −x else x) |
negate b (Nd ts) = Nd (map (negate (¬b)) ts)

lemma negate_negate: negate f (negate f t) = t
by(induction t arbitrary: f)(auto cong: map_cong)

lemma uminus_infs:
  fixes f :: 'a ⇒ 'b::de_morgan_algebra
shows − infs (map f xs) = sups (map (λx. − f x) xs)
by(induction xs) (auto simp: de_Morgan_inf)

lemma supinf_negate: supinf (negate b t) = − infsup (negate (¬b) (t::(_::de_morgan_algebra)tree))
by(induction t) (auto simp: uminus_infs cong: map_cong)

lemma negsup_supinf_negate: negsup t = supinf(negate False t)
by(induction t) (auto simp: supinf_negate cong: map_cong)

3.3.1 Fail-Hard

fun ab_negsup :: 'a ⇒ 'a ⇒ ('a::de_morgan_algebra)tree ⇒ 'a **and** ab_negsups
where
ab_negsup a b (Lf x) = x |
ab_negsup a b (Nd ts) = ab_negsups a b ts |

ab_negsups a b [] = a |
ab_negsups a b (t#ts) =
  (let a' = a ⊔ − ab_negsup (−b) (−a) t
   in if a' ≥ b then a' else ab_negsups a' b ts)

A direct *bounded* proof:

lemma ab_negsups_ge_a: ab_negsups a b ts ≥ a
apply(induction ts arbitrary: a)
by (auto simp: Let_def)(use le_sup_iff **in** blast)

lemma bounded_val_ab_neg:
shows bounded (a) b (negsup t) (ab_negsup (a) b t)
and bounded a b (negsup (Nd ts)) (ab_negsups (a) b ts)
proof(induction a b t **and** a b ts rule: ab_negsup_ab_negsups.induct)
  case (4 a b t ts)
  then show ?case
    apply(simp add: Let_def inf.coboundedI1)
   by (smt (verit, ccfv_threshold) ab_negsups_ge_a de_Morgan_sup de_morgan_algebra_class.neg_neg inf.absorb_iff2 inf_sup_distrib1 le_iff_sup sup_commute sup_left_commute)

qed *auto*

An indirect proof:

theorem *ab_sup_ab_negsup*:
shows *ab_sup a b t = ab_negsup a b (negate False t)*
and *ab_sups a b ts = ab_negsups a b (map (negate True) ts)*
and *ab_inf a b t = − ab_negsup (−b) (−a) (negate True t)*
and *ab_infs a b ts = − ab_negsups (−b) (−a) (map (negate False) ts)*
proof(*induction a b t* **and** *a b ts* **and** *a b t* **and** *a b ts rule*: *ab_sup_ab_sups_ab_inf_ab_infs.induct*)
case 8 **then show** *?case*
by(*simp add*: *Let_def de_Morgan_sup de_Morgan_inf uminus_le_reorder*)
qed (*simp_all add*: *Let_def*)

corollary *ab_negsup_bot_top*: *ab_negsup* ⊥ ⊤ *t = negsup t*
by (*metis ab_sup_bot_top ab_sup_ab_negsup*(1) *negate_negate negsup_supinf_negate*)

Correctness statements derived from non-negative versions:

corollary *eq_mod_ab_negsup_supinf_negate*:
a ⊔ *ab_negsup a b t* ⊓ *b = a* ⊔ *supinf (negate False t)* ⊓ *b*
by (*metis eq_mod_ab_val*(1) *ab_sup_ab_negsup*(1) *negate_negate*)

corollary *bounded_negsup_ab_negsup*:
bounded a b (negsup t) (ab_negsup a b t)
by (*metis negate_negate ab_sup_ab_negsup*(1) *bounded_val_ab*(1) *negsup_supinf_negate*)

3.3.2 Fail-Soft

fun *ab_negsup'* :: *'a* ⇒ *'a* ⇒ (*'a*::*de_morgan_algebra*)*tree* ⇒ *'a* **and** *ab_negsups'*
where
ab_negsup' a b (Lf x) = x |
ab_negsup' a b (Nd ts) = (ab_negsups' a b ⊥ *ts)* |

ab_negsups' a b m [] = m |
ab_negsups' a b m (t#ts) = (let m' = sup m (− ab_negsup' (−b) (− sup m a) t)
in
if m' ≥ b then m' else ab_negsups' a b m' ts)

Relate un-negated to negated:

theorem *ab_sup'_ab_negsup'*:
shows *ab_sup' a b t = ab_negsup' a b (negate False t)*
and *ab_sups' a b m ts = ab_negsups' a b m (map (negate True) ts)*
and *ab_inf' a b t = − ab_negsup' (−b) (−a) (negate True t)*
and *ab_infs' a b m ts = − ab_negsups' (−b) (−a) (−m) (map (negate False) ts)*
proof(*induction a b t* **and** *a b m ts* **and** *a b t* **and** *a b m ts rule*: *ab_sup'_ab_sups'_ab_inf'_ab_infs'.induct*)
case (4 *a b m t ts*)
then show *?case*
by(*simp add*: *Let_def de_Morgan_sup de_Morgan_inf uminus_le_reorder*)
next

case (8 *a b m t ts*)
then show *?case*
by(*simp add*: *Let_def de_Morgan_sup de_Morgan_inf uminus_le_reorder*)
qed *auto*

lemma *ab_negsups′_ge_a*: *ab_negsups′ a b m ts* ≥ *m*
apply(*induction ts arbitrary*: *a b m*)
by (*auto simp*: *Let_def*)(*use le_sup_iff* **in** *blast*)

theorem *bounded_val_ab′_neg*:
shows *bounded a b* (*negsup t*) (*ab_negsup′ a b t*)
and *bounded* (*sup a m*) *b* (*negsup* (*Nd ts*)) (*ab_negsups′ a b m ts*)
proof(*induction a b t* **and** *a b m ts rule*: *ab_negsup′_ab_negsups′.induct*)
case (4 *a b m t ts*)
then show *?case*
apply (*auto simp add*: *Let_def inf.coboundedI*1 *sup.coboundedI*1)
apply (*smt* (*verit*, *ccfv_threshold*) *de_Morgan_sup neg_neg inf.coboundedI*1 *le_iff_sup sup.cobounded*1 *sup_assoc sup_commute*)
apply (*metis* (*no_types*, *lifting*) *ab_negsups′_ge_a de_Morgan_sup neg_neg inf.coboundedI*1 *inf_sup_distrib*1 *le_iff_sup le_sup_iff*)
by (*smt* (*verit*, *ccfv_threshold*) *de_Morgan_inf de_morgan_algebra_class.neg_neg inf.orderE le_iff_sup sup.idem sup_commute sup_left_commute*)
qed *auto*

corollary *bounded a b* (*negsup t*) (*ab_negsup′ a b t*)
by (*metis negate_negate ab_sup′_ab_negsup′*(1) *bounded_val_ab′*(1) *negsup_supinf_negate*)

theorem *bounded_ab_neg′_ab_neg*:
shows *bounded a b* (*ab_negsup′ a b t*) (*ab_negsup a b t*)
and *bounded* (*sup a m*) *b* (*ab_negsups′ a b m ts*) (*ab_negsup* ($a \sqcup m$) *b* (*Nd ts*))
oops

end

Chapter 4

An Application: Tic-Tac-Toe

theory *TicTacToe*
imports
  Alpha_Beta_Pruning.Alpha_Beta_Linear
begin

We formalize a general nxn version of tic-tac-toe (noughts and crosses). The winning condition is very simple: a full horizontal, vertical or diagonal line occupied by one player.

A square is either empty (*None*) or occupied by one of the two players (*Some b*).

type_synonym *sq = bool option*
type_synonym *row = sq list*
type_synonym *position = row list*

Successor positions:

fun *next_rows* :: *sq* ⇒ *row* ⇒ *row list* **where**
next_rows s′ (s#ss) = (if s=None then [s′#ss] else []) @ map ((#) s) (next_rows s′ ss) |
next_rows _ [] = []

fun *next_poss* :: *sq* ⇒ *position* ⇒ *position list* **where**
next_poss s′ (ss#sss) = map (λss′. ss′ # sss) (next_rows s′ ss) @ map ((#) ss) (next_poss s′ sss) |
next_poss _ [] = []

A game is won if a full line is occupied by a given square:

fun *diag* :: *′a list list* ⇒ *′a list* **where**
diag ((x#_) # xss) = x # diag (map tl xss) |
diag [] = []

fun *lines* :: *position* ⇒ *sq list list* **where**
lines sss = diag sss # diag (map rev sss) # sss @ transpose sss

fun *won* :: *sq* ⇒ *position* ⇒ *bool* **where**

$won\ sq\ pos = (\exists\, ss \in set\ (lines\ pos).\ \forall\, s \in set\ ss.\ s = sq)$

How many lines are almost won (i.e. all *sq* except one *None*)? Not actually used for heuristic evaluation, too slow.

fun *awon* :: *sq* ⇒ *position* ⇒ *nat* **where**
awon sq sss = *length* (*filter* (λ*ss*. *filter* (λ*s*. *s*≠*sq*) *ss* = [*None*]) (*lines sss*))

The game tree up to a given depth n. Trees at depth $\geq n$ are replaced by *Lf* 0 for simplicity; no heuristic evaluation.

fun *tree* :: *nat* ⇒ *bool* ⇒ *position* ⇒ *ereal tree* **where**
tree (*Suc n*) *b pos* = (
  if won (*Some* (¬*b*)) *pos then Lf*(*if b then* −∞ *else* ∞) — Opponent won
  else
  case next_poss (*Some b*) *pos of*
    [] ⇒ *Lf* 0 — Draw |
    poss ⇒ *Nd* (*map* (*tree n* (¬*b*)) *poss*)) |
tree 0 *b pos* = *Lf* 0

definition *start* :: *nat* ⇒ *position* **where**
start n = *replicate n* (*replicate n None*)

Now we evaluate the game for small n.

The trivial cases:

lemma *maxmin* (*tree* 2 *True* (*start* 1)) = ∞
by *eval*

lemma *maxmin* (*tree* 5 *True* (*start* 2)) = ∞
by *eval*

3x3, full game tree (depth=10), no noticeable speedup of alpha-beta.

lemma *maxmin* (*tree* 10 *True* (*start* 3)) = 0
by *eval*
lemma *ab_max* (−∞) ∞ (*tree* 10 *True* (*start* 3)) = 0
by *eval*

4x4, game tree up to depth 7, alpha-beta noticeably faster.

lemma *maxmin* (*tree* 7 *True* (*start* 4)) = 0
by *eval*
lemma *ab_max* (−∞) ∞ (*tree* 7 *True* (*start* 4)) = 0
by *eval*

end