IsaSAT: Heuristics and Code Generation

Mathias Fleury, Jasmin Blanchette, Peter Lammich

July 17, 2023

Contents

theory *Map-Fun-Rel*
imports *More-Sepref.WB-More-Refinement*
begin

0.0.1 Refinement from function to lists

Throughout our formalization, we often use functions at the most abstract level, that we refine to lists assuming a known domain.

One thing to remark is that I have changed my mind on how to do things. Before we refined things directly and kept the domain implicit. Nowadays, I make the domain explicit – even if this means that we have to duplicate the information of the domain through all the components of our state.

Definition **definition** *map-fun-rel* :: ‹(*nat* × *'key*) *set* ⇒ (*'b* × *'a*) *set* ⇒ (*'b list* × (*'key* ⇒ *'a*)) *set*› **where**
map-fun-rel-def-internal:
‹*map-fun-rel D R* = {(*m*, *f*). ∀(*i*, *j*)∈*D*. *i* < *length m* ∧ (*m* ! *i*, *f j*) ∈ *R*}›

lemma *map-fun-rel-def*:
‹⟨*R*⟩*map-fun-rel D* = {(*m*, *f*). ∀(*i*, *j*)∈*D*. *i* < *length m* ∧ (*m* ! *i*, *f j*) ∈ *R*}›
⟨*proof*⟩

lemma *map-fun-rel-nth*:
‹(*xs*,*ys*)∈⟨*R*⟩*map-fun-rel D* ⟹ (*i*,*j*)∈*D* ⟹ (*xs* ! *i* , *ys j*) ∈ *R*›
⟨*proof*⟩

In combination with lists **definition** *length-ll-f* **where**
‹*length-ll-f W L* = *length* (*W L*)›

lemma *map-fun-rel-length*:
‹(*xs*,*ys*)∈⟨⟨*R*⟩*list-rel*⟩*map-fun-rel D* ⟹ (*i*,*j*)∈*D* ⟹ (*length-ll xs i*, *length-ll-f ys j*) ∈ *nat-rel*›

⟨*proof*⟩

definition *append-update* :: ‹(′*a* ⇒ ′*b list*) ⇒ ′*a* ⇒ ′*b* ⇒ ′*a* ⇒ ′*b list*› **where**
‹*append-update W L a* = *W*(*L*:= *W* (*L*) @ [*a*])›

end

0.1 Pairing Heap According to Oksaki (Modified)

theory *Ordered-Pairing-Heap-List2*
imports
HOL−Library.Multiset
HOL−Data-Structures.Priority-Queue-Specs
begin

Chapter 1

Pairing heaps

To make it useful we simply parametrized the formalization by the order. We ruse the formalization of Tobias Nipkow, but make it *useful* for refinement by separating node and score. We also need to add way to increase the score.

1.0.1 Definitions

This version of pairing heaps is a modified version of the one by Okasaki [**?**] that avoids structural invariants.

datatype (*'b*, *'a*) *hp* = *Hp* (*node*: *'b*) (*score*: *'a*) (*hps*: (*'b*, *'a*) *hp list*)

type-synonym (*'a*, *'b*) *heap* = (*'a*, *'b*) *hp option*

hide-const (**open**) *insert*

fun *get-min* :: (*'b*, *'a*) *heap* ⇒ *'a* **where**
get-min (*Some*(*Hp* - *x* -)) = *x*

This is basically the useful version:

fun *get-min2* :: (*'b*, *'a*) *heap* ⇒ *'b* **where**
get-min2 (*Some*(*Hp n x* -)) = *n*

locale *pairing-heap-assms* =
  fixes *lt* :: ‹*'a* ⇒ *'a* ⇒ *bool*› **and**
    le :: ‹*'a* ⇒ *'a* ⇒ *bool*›
begin

fun *link* :: (*'b*, *'a*) *hp* ⇒ (*'b*, *'a*) *hp* ⇒ (*'b*, *'a*) *hp* **where**
link (*Hp m x lx*) (*Hp n y ly*) =
  (*if lt x y then Hp m x* (*Hp n y ly* # *lx*) *else Hp n y* (*Hp m x lx* # *ly*))

fun *merge* :: (*'b*, *'a*) *heap* ⇒ (*'b*, *'a*) *heap* ⇒ (*'b*, *'a*) *heap* **where**
merge h None = *h* |
merge None h = *h* |
merge (*Some h1*) (*Some h2*) = *Some*(*link h1 h2*)

lemma *merge-None*[*simp*]: *merge None h* = *h*
⟨*proof*⟩

fun *insert* :: $'b \Rightarrow ('a) \Rightarrow ('b, 'a)$ *heap* $\Rightarrow ('b, 'a)$ *heap* **where**
insert n x None = *Some*(*Hp n x* []) |
insert n x (*Some h*) = *Some*(*link* (*Hp n x* []) *h*)

fun $pass_1$ :: $('b, 'a)$ *hp list* $\Rightarrow ('b, 'a)$ *hp list* **where**
$pass_1$ [] = []
| $pass_1$ [*h*] = [*h*]
| $pass_1$ (*h1*#*h2*#*hs*) = *link h1 h2* # $pass_1$ *hs*

fun $pass_2$ :: $('b, 'a)$ *hp list* $\Rightarrow ('b, 'a)$ *heap* **where**
$pass_2$ [] = *None*
| $pass_2$ (*h*#*hs*) = *Some*(*case* $pass_2$ *hs of None* $\Rightarrow$ *h* | *Some h'* $\Rightarrow$ *link h h'*)

fun *merge-pairs* :: $('b, 'a)$ *hp list* $\Rightarrow ('b, 'a)$ *heap* **where**
merge-pairs [] = *None*
| *merge-pairs* [*h*] = *Some h*
| *merge-pairs* (*h1* # *h2* # *hs*) =
Some(*let h12* = *link h1 h2 in case merge-pairs hs of None* $\Rightarrow$ *h12* | *Some h* $\Rightarrow$ *link h12 h*)

fun *del-min* :: $('b, 'a)$ *heap* $\Rightarrow ('b, 'a)$ *heap* **where**
del-min None = *None*
| *del-min* (*Some*(*Hp - x hs*)) = $pass_2$ ($pass_1$ *hs*)

fun (**in** −)*remove-key-children* :: ‹$'b \Rightarrow ('b, 'a)$ *hp list* $\Rightarrow ('b, 'a)$ *hp list*› **where**
‹*remove-key-children k* [] = []› |
‹*remove-key-children k* ((*Hp x n c*) # *xs*) =
(*if k* = *x then remove-key-children k xs else* ((*Hp x n* (*remove-key-children k c*)) # *remove-key-children k xs*))›

fun (**in** −)*remove-key* :: ‹$'b \Rightarrow ('b, 'a)$ *hp* $\Rightarrow ('b, 'a)$ *heap*› **where**
‹*remove-key k* (*Hp x n c*) = (*if x* = *k then None else Some* (*Hp x n* (*remove-key-children k c*)))›

fun (**in** −)*find-key-children* :: ‹$'b \Rightarrow ('b, 'a)$ *hp list* $\Rightarrow ('b, 'a)$ *heap*› **where**
‹*find-key-children k* [] = *None*› |
‹*find-key-children k* ((*Hp x n c*) # *xs*) =
(*if k* = *x then Some* (*Hp x n c*) *else*
(*case find-key-children k c of Some a* $\Rightarrow$ *Some a* | - $\Rightarrow$ *find-key-children k xs*))›

fun (**in** −)*find-key* :: ‹$'b \Rightarrow ('b, 'a)$ *hp* $\Rightarrow ('b, 'a)$ *heap*› **where**
‹*find-key k* (*Hp x n c*) =
(*if k* = *x then Some* (*Hp x n c*) *else find-key-children k c*)›

definition *decrease-key* :: ‹$'b \Rightarrow 'a \Rightarrow ('b, 'a)$ *hp* $\Rightarrow ('b, 'a)$ *heap*› **where**
‹*decrease-key k s hp* = (*case find-key k hp of None* $\Rightarrow$ *Some hp*
| (*Some* (*Hp - - c*)) $\Rightarrow$
(*case remove-key k hp of*
None $\Rightarrow$ *Some* (*Hp k s c*)
| *Some x* $\Rightarrow$ *merge-pairs* [*Hp k s c*, *x*]))›

1.0.2 Correctness Proofs

An optimization:

lemma *pass12-merge-pairs*: $pass_2$ ($pass_1$ *hs*) = *merge-pairs hs*
⟨*proof*⟩

declare *pass12-merge-pairs*[*code-unfold*]

Invariants

fun (**in** −) *set-hp* :: ‹('b, 'a) hp ⇒ 'a set› **where**
‹*set-hp* (*Hp* - *x* *hs*) = ({*x*} ∪ ⋃(*set-hp* ‘ *set* *hs*))›

fun *php* :: ('b, 'a) *hp* ⇒ *bool* **where**
php (*Hp* - *x* *hs*) = (∀ *h* ∈ *set* *hs*. (∀ *y* ∈ *set-hp* *h*. *le* *x* *y*) ∧ *php* *h*)

definition *invar* :: ('b, 'a) *heap* ⇒ *bool* **where**
invar *ho* = (*case* *ho* *of* *None* ⇒ *True* | *Some* *h* ⇒ *php* *h*)
end

locale *pairing-heap* = *pairing-heap-assms* *lt* *le*
for *lt* :: ‹'a ⇒ 'a ⇒ bool› **and**
le :: ‹'a ⇒ 'a ⇒ bool› +
assumes *le*: ‹⋀*a* *b*. *le* *a* *b* ⟷ *a* = *b* ∨ *lt* *a* *b*› **and**
trans: ‹*transp* *le*› **and**
transt: ‹*transp* *lt*› **and**
totalt: ‹*totalp* *lt*›
begin

lemma *php-link*: *php* *h1* ⟹ *php* *h2* ⟹ *php* (*link* *h1* *h2*)
⟨*proof*⟩

lemma *invar-None*[*simp*]: ‹*invar* *None*›
⟨*proof*⟩

lemma *invar-merge*:
⟦ *invar* *h1*; *invar* *h2* ⟧ ⟹ *invar* (*merge* *h1* *h2*)
⟨*proof*⟩

lemma *invar-insert*: *invar* *h* ⟹ *invar* (*insert* *n* *x* *h*)
⟨*proof*⟩

lemma *invar-pass1*: ∀ *h* ∈ *set* *hs*. *php* *h* ⟹ ∀ *h* ∈ *set* ($pass_1$ *hs*). *php* *h*
⟨*proof*⟩

lemma *invar-pass2*: ∀ *h* ∈ *set* *hs*. *php* *h* ⟹ *invar* ($pass_2$ *hs*)
⟨*proof*⟩

lemma *invar-Some*: *invar*(*Some* *h*) = *php* *h*
⟨*proof*⟩

lemma *invar-del-min*: *invar* *h* ⟹ *invar* (*del-min* *h*)
⟨*proof*⟩

lemma (**in** −)*in-remove-key-children-in-childrenD*: ‹*h* ∈ *set* (*remove-key-children* *k* *c*) ⟹ *xa* ∈ *set-hp* *h* ⟹ *xa* ∈ ⋃(*set-hp* ‘ *set* *c*)›
⟨*proof*⟩

lemma *php-remove-key-children*: ‹∀ *h*∈*set* *h1*. *php* *h* ⟹ *h* ∈ *set* (*remove-key-children* *k* *h1*) ⟹ *php* *h*›

⟨*proof*⟩

lemma *php-remove-key*: ‹*php h1* ⟹ *invar* (*remove-key k h1*)›
⟨*proof*⟩

lemma *invar-find-key-children*: ‹∀ *h*∈*set c*. *php h* ⟹ *invar* (*find-key-children k c*)›
⟨*proof*⟩

lemma *invar-find-key*: ‹*php h1* ⟹ *invar* (*find-key k h1*)›
⟨*proof*⟩

lemma (**in** −)*remove-key-None-iff*: ‹*remove-key k h1* = *None* ⟷ *node h1* = *k*›
⟨*proof*⟩

lemma *php-decrease-key*:
‹*php h1* ⟹ (*case* (*find-key k h1*) *of None* ⇒ *True* | *Some a* ⇒ *le s* (*score a*)) ⟹ *invar* (*decrease-key k s h1*)›
⟨*proof*⟩

Functional Correctness

fun (**in** −) *mset-hp* :: (*'b*, *'a*) *hp* ⇒*'a multiset* **where**
mset-hp (*Hp - x hs*) = {#*x*#} + *sum-mset*(*mset*(*map mset-hp hs*))

definition (**in** −) *mset-heap* :: (*'b*, *'a*) *heap* ⇒*'a multiset* **where**
mset-heap ho = (*case ho of None* ⇒ {#} | *Some h* ⇒ *mset-hp h*)

lemma (**in** −) *set-mset-mset-hp*: *set-mset* (*mset-hp h*) = *set-hp h*
⟨*proof*⟩

lemma (**in** −) *mset-hp-empty*[*simp*]: *mset-hp hp* ≠ {#}
⟨*proof*⟩

lemma (**in** −) *mset-heap-Some*: *mset-heap*(*Some hp*) = *mset-hp hp*
⟨*proof*⟩

lemma (**in** −) *mset-heap-empty*: *mset-heap h* = {#} ⟷ *h* = *None*
⟨*proof*⟩

lemma (**in** −)*get-min-in*:
h ≠ *None* ⟹ *get-min h* ∈ *set-hp*(*the h*)
⟨*proof*⟩

lemma *get-min-min*: ⟦ *h* ≠ *None*; *invar h*; *x* ∈ *set-hp*(*the h*) ⟧ ⟹ *le* (*get-min h*) *x*
⟨*proof*⟩

lemma (**in** *pairing-heap-assms*) *mset-link*: *mset-hp* (*link h1 h2*) = *mset-hp h1* + *mset-hp h2*
⟨*proof*⟩

lemma (**in** *pairing-heap-assms*) *mset-merge*: *mset-heap* (*merge h1 h2*) = *mset-heap h1* + *mset-heap h2*
⟨*proof*⟩

lemma (**in** *pairing-heap-assms*) *mset-insert*: *mset-heap* (*insert n a h*) = {#*a*#} + *mset-heap h*
⟨*proof*⟩

lemma (**in** *pairing-heap-assms*) *mset-merge-pairs*: *mset-heap* (*merge-pairs hs*) = *sum-mset*(*image-mset mset-hp* (*mset hs*))
⟨*proof*⟩

lemma (**in** *pairing-heap-assms*) *mset-del-min*: *h* ≠ *None* ⟹
mset-heap (*del-min h*) = *mset-heap h* − {#*get-min h*#}
⟨*proof*⟩

Some more lemmas to make the heaps easier to use:

lemma *invar-merge-pairs*:
⟦∀ *h*∈*set h1*. *invar* (*Some h*)⟧ ⟹ *invar* (*merge-pairs h1*)
⟨*proof*⟩

end

context *pairing-heap-assms*
begin

lemma *merge-pairs-None-iff* [*iff*]: *merge-pairs hs* = *None* ⟷ *hs* = []
⟨*proof*⟩

end

Last step: prove all axioms of the priority queue specification with the right sort:

locale *pairing-heap2* =
fixes *ltype* :: ‹′*a*::*linorder itself*›
begin

sublocale *pairing-heap* **where**
lt =‹(<) :: ′*a* ⇒ ′*a* ⇒ *bool*› **and** *le* = ‹(≤)›
⟨*proof*⟩

interpretation *pairing*: *Priority-Queue-Merge*
where *empty* = *None* **and** *is-empty* = λ*h*. *h* = *None*
and *merge* = *merge* **and** *insert* = ‹*insert default*›
and *del-min* = *del-min* **and** *get-min* = *get-min*
and *invar* = *invar* **and** *mset* = *mset-heap*
⟨*proof*⟩

end

end
theory *Heaps-Abs*
imports *Ordered-Pairing-Heap-List2*
Weidenbach-Book-Base.Explorer
Isabelle-LLVM.IICF
More-Sepref.WB-More-Refinement
begin

We first tried to follow the setup from Isabelle LLVM, but it is not clear how useful this really is. Hence we adapted the definition from the abstract operations.

locale *hmstruct-with-prio* =
fixes *lt* :: ‹′*v* ⇒ ′*v* ⇒ *bool*› **and**
le :: ‹′*v* ⇒ ′*v* ⇒ *bool*›

```
  assumes hm-le: ‹⋀a b. le a b ⟷ a = b ∨ lt a b› and
    hm-trans: ‹transp le› and
    hm-transt: ‹transp lt› and
    hm-totalt: ‹totalp lt›
begin

  definition prio-peek-min where
    prio-peek-min ≡ (λ(𝒜, b, w). (λv.
        v ∈# b
      ∧ (∀ v'∈set-mset b. le (w v) (w v'))))

  definition mop-prio-peek-min where
    mop-prio-peek-min ≡ (λ(𝒜, b, w). doN {ASSERT (b≠{#}); SPEC (prio-peek-min (𝒜, b,w))})

  definition mop-prio-change-weight where
    mop-prio-change-weight ≡ (λv ω (𝒜, b, w). doN {
      ASSERT (v ∈# 𝒜);
      ASSERT (v ∈# b ⟶ le ω (w v));
      RETURN (𝒜, b, w(v := ω))
   })

  definition mop-prio-insert where
    mop-prio-insert ≡ (λv ω (𝒜, b, w). doN {
      ASSERT (v ∉# b ∧ v∈#𝒜);
      RETURN (𝒜, add-mset v b, w(v := ω))
   })

  definition mop-prio-is-in where
    ‹mop-prio-is-in = (λv (𝒜, b, w). do {
    ASSERT (v ∈# 𝒜);
    RETURN (v ∈#b)
    })›
  definition mop-prio-insert-maybe where
    mop-prio-insert-maybe ≡ (λv ω (bw). doN {
      b ← mop-prio-is-in v bw;
      if ¬b then mop-prio-insert v ω (bw)
      else mop-prio-change-weight v ω (bw)
   })
```

TODO this is a shortcut and it could make sense to force w to remember the old values.

```
  definition mop-prio-old-weight where
    mop-prio-old-weight = (λv (𝒜, b, w). doN {
      ASSERT (v ∈# 𝒜);
      b ← mop-prio-is-in v (𝒜, b, w);
      if b then RETURN (w v) else RES UNIV
   })

  definition mop-prio-insert-raw-unchanged where
    mop-prio-insert-raw-unchanged = (λv h. doN {
      ASSERT (v ∉# fst (snd h));
      w ← mop-prio-old-weight v h;
      mop-prio-insert v w h
   })

  definition mop-prio-insert-unchanged where
    mop-prio-insert-unchanged = (λv (bw). doN {
```

b ← *mop-prio-is-in v bw*;
if ¬*b then mop-prio-insert-raw-unchanged v* (*bw*)
else RETURN bw
})

definition *prio-del* **where**
‹*prio-del* = (λv ($\mathcal{A}$, *b*, *w*). ($\mathcal{A}$, *b* − {#*v*#}, *w*))›

definition *mop-prio-del* **where**
mop-prio-del = (λv ($\mathcal{A}$, *b*, *w*). *doN* {
ASSERT ($v \in\# b \wedge v \in\# \mathcal{A}$);
RETURN (*prio-del v* ($\mathcal{A}$, *b*, *w*))
})

definition *mop-prio-pop-min* **where**
mop-prio-pop-min = ($\lambda\mathcal{A}bw$. *doN* {
v ← *mop-prio-peek-min* $\mathcal{A}bw$;
bw ← *mop-prio-del v* $\mathcal{A}bw$;
RETURN (*v*, *bw*)
})

sublocale *pairing-heap*
⟨*proof*⟩

end

end
theory *Pairing-Heaps*
imports *Ordered-Pairing-Heap-List2*
Isabelle-LLVM.IICF
More-Sepref.WB-More-Refinement
Heaps-Abs
begin

1.1 Pairing Heaps

1.1.1 Genealogy Over Pairing Heaps

We first tried to use the heapmap, but this attempt was a terrible failure, because as useful the heapmap is parametrized by the size. This might be useful in some contexts, but I consider this to be the most terrible idea ever, based on past experience. So instead I went for a modification of the pairing heaps.

To increase fun, we reuse the trick from VSIDS to represent the pairing heap inside an array in order to avoid allocation yet another array. As a side effect, it also avoids including the label inside the node (because per definition, the label is exactly the index). But maybe pointers are actually better, because by definition in Isabelle no graph is shared.

fun *mset-nodes* :: (′*b*, ′*a*) *hp* ⇒′*b multiset* **where**
mset-nodes (*Hp x - hs*) = {#*x*#} + $\sum_{\#}$ (*mset-nodes* ‘# *mset hs*)

context *pairing-heap-assms*
begin

lemma *mset-nodes-link*[*simp*]: ‹*mset-nodes* (*link a b*) = *mset-nodes a* + *mset-nodes b*›
⟨*proof*⟩

lemma *mset-nodes-merge-pairs*: ‹*merge-pairs a* $\neq$ *None* $\Longrightarrow$ *mset-nodes* (*the* (*merge-pairs a*)) = *sum-list* (*map mset-nodes a*)›
⟨*proof*⟩

lemma *mset-nodes-pass*$_1$[*simp*]: ‹*sum-list* (*map mset-nodes* (*pass*$_1$ *a*)) = *sum-list* (*map mset-nodes a*)›
⟨*proof*⟩

lemma *mset-nodes-pass*$_2$[*simp*]: ‹*pass*$_2$ *a* $\neq$ *None* $\Longrightarrow$ *mset-nodes* (*the* (*pass*$_2$ *a*)) = *sum-list* (*map mset-nodes a*)›
⟨*proof*⟩

end

lemma *mset-nodes-simps*[*simp*]: ‹*mset-nodes* (*Hp x n hs*) = {#*x*#} + (*sum-list* (*map mset-nodes hs*))›
⟨*proof*⟩

lemmas [*simp del*] = *mset-nodes.simps*

fun *hp-next* **where**
‹*hp-next a* (*Hp m s* (*x* # *y* # *children*)) = (*if a* = *node x then Some y else* (*case hp-next a x of Some a* $\Rightarrow$ *Some a* | *None* $\Rightarrow$ *hp-next a* (*Hp m s* (*y* # *children*))))› |
‹*hp-next a* (*Hp m s* [*b*]) = *hp-next a b*› |
‹*hp-next a* (*Hp m s* []) = *None*›

lemma [*simp*]: ‹*size-list size* (*hps x*) < *Suc* (*size x* + *a*)›
⟨*proof*⟩

fun *hp-prev* **where**
‹*hp-prev a* (*Hp m s* (*x* # *y* # *children*)) = (*if a* = *node y then Some x else* (*case hp-prev a x of Some a* $\Rightarrow$ *Some a* | *None* $\Rightarrow$ *hp-prev a* (*Hp m s* (*y* # *children*))))› |
‹*hp-prev a* (*Hp m s* [*b*]) = *hp-prev a b*› |
‹*hp-prev a* (*Hp m s* []) = *None*›

fun *hp-child* **where**
‹*hp-child a* (*Hp m s* (*x* # *children*)) = (*if a* = *m then Some x else* (*case hp-child a x of None* $\Rightarrow$ *hp-child a* (*Hp m s children*) | *Some a* $\Rightarrow$ *Some a*))› |
‹*hp-child a* (*Hp m s* -) = *None*›

fun *hp-node* **where**
‹*hp-node a* (*Hp m s* (*x*#*children*)) = (*if a* = *m then Some* (*Hp m s* (*x*#*children*)) *else* (*case hp-node a x of None* $\Rightarrow$ *hp-node a* (*Hp m s children*) | *Some a* $\Rightarrow$ *Some a*))› |
‹*hp-node a* (*Hp m s* []) = (*if a* = *m then Some* (*Hp m s* []) *else None*)›

lemma *node-in-mset-nodes*[*simp*]: ‹*node x* $\in$# *mset-nodes x*›
⟨*proof*⟩

lemma *hp-next-None-notin*[*simp*]: ‹*m* $\notin$# *mset-nodes a* $\Longrightarrow$ *hp-next m a* = *None*›
⟨*proof*⟩

lemma *hp-prev-None-notin*[*simp*]: ‹*m* $\notin$# *mset-nodes a* $\Longrightarrow$ *hp-prev m a* = *None*›
⟨*proof*⟩

lemma *hp-child-None-notin*[*simp*]: ‹*m* $\notin$# *mset-nodes a* $\Longrightarrow$ *hp-child m a* = *None*›
⟨*proof*⟩

lemma *hp-node-None-notin2*[*iff*]: ‹*hp-node m a = None* ⟷ *m* ∉# *mset-nodes a*›
⟨*proof*⟩

lemma *hp-node-None-notin*[*simp*]: ‹*m* ∉# *mset-nodes a* ⟹ *hp-node m a = None*›
⟨*proof*⟩

lemma *hp-node-simps*[*simp*]: ‹*hp-node m* (*Hp m* w_m ch_m) = *Some* (*Hp m* w_m ch_m)›
⟨*proof*⟩

lemma *hp-next-None-notin-children*[*simp*]: ‹*a* ∉# *sum-list* (*map mset-nodes children*) ⟹ *hp-next a* (*Hp m* w_m (*children*)) = *None*›
⟨*proof*⟩

lemma *hp-prev-None-notin-children*[*simp*]: ‹*a* ∉# *sum-list* (*map mset-nodes children*) ⟹ *hp-prev a* (*Hp m* w_m (*children*)) = *None*›
⟨*proof*⟩

lemma *hp-child-None-notin-children*[*simp*]: ‹*a* ∉# *sum-list* (*map mset-nodes children*) ⟹ *a* ≠ *m* ⟹ *hp-child a* (*Hp m* w_m (*children*)) = *None*›
⟨*proof*⟩

The function above are nicer for definition than for usage. Instead we define the list version and change the simplification lemmas. We initially tried to use a recursive function, but the proofs did not go through (and it seemed that the induction principle were to weak).

fun *hp-next-children* **where**
‹*hp-next-children a* (*x* # *y* # *children*) = (*if a* = *node x then Some y else* (*case hp-next a x of Some a* ⇒ *Some a* | *None* ⇒ *hp-next-children a* (*y* # *children*)))› |
‹*hp-next-children a* [*b*] = *hp-next a b*› |
‹*hp-next-children a* [] = *None*›

lemma *hp-next-simps*[*simp*]:
‹*hp-next a* (*Hp m s children*) = *hp-next-children a children*›
⟨*proof*⟩

lemma *hp-next-children-None-notin*[*simp*]: ‹*m* ∉# $\sum_{\#}$ (*mset-nodes* ‘# *mset children*) ⟹ *hp-next-children m children = None*›
⟨*proof*⟩

lemma [*simp*]: ‹*distinct-mset* (*mset-nodes a*) ⟹ *hp-next* (*node a*) *a* = *None*›
⟨*proof*⟩

lemma [*simp*]:
‹ch_m ≠ [] ⟹ *hp-next-children* (*node a*) (*a* # ch_m) = *Some* (*hd* ch_m)›
⟨*proof*⟩

fun *hp-prev-children* **where**
‹*hp-prev-children a* (*x* # *y* # *children*) = (*if a* = *node y then Some x else* (*case hp-prev a x of Some a* ⇒ *Some a* | *None* ⇒ *hp-prev-children a* (*y* # *children*)))› |
‹*hp-prev-children a* [*b*] = *hp-prev a b*› |
‹*hp-prev-children a* [] = *None*›

lemma *hp-prev-simps*[*simp*]:
‹*hp-prev a* (*Hp m s children*) = *hp-prev-children a children*›
⟨*proof*⟩

lemma *hp-prev-children-None-notin*[*simp*]: ‹$m \notin\# \sum_{\#}$ (*mset-nodes* '# *mset children*) $\Longrightarrow$ *hp-prev-children m children* = *None*›
⟨*proof*⟩

lemma [*simp*]: ‹*distinct-mset* (*mset-nodes a*) $\Longrightarrow$ *hp-prev* (*node a*) *a* = *None*›
⟨*proof*⟩

lemma *hp-next-in-first-child* [*simp*]: ‹*distinct-mset* (*sum-list* (*map mset-nodes* ch_m) + (*mset-nodes a*)) $\Longrightarrow$
xa $\in\#$ *mset-nodes a* $\Longrightarrow$ *xa* $\neq$ *node a* $\Longrightarrow$
hp-next-children xa (*a* # ch_m) = (*hp-next xa a*)›
⟨*proof*⟩

lemma *hp-next-skip-hd-children*:
‹*distinct-mset* (*sum-list* (*map mset-nodes* ch_m) + (*mset-nodes a*)) $\Longrightarrow$ *xa* $\in\# \sum_{\#}$ (*mset-nodes* '# *mset* ch_m) $\Longrightarrow$
xa $\neq$ *node a* $\Longrightarrow$ *hp-next-children xa* (*a* # ch_m) = *hp-next-children xa* (ch_m)›
⟨*proof*⟩

lemma *hp-prev-in-first-child* [*simp*]: ‹*distinct-mset*
(*sum-list* (*map mset-nodes* ch_m) + (*mset-nodes a*)) $\Longrightarrow$ *xa* $\in\#$ *mset-nodes a* $\Longrightarrow$ *hp-prev-children xa* (*a* # ch_m) = *hp-prev xa a*›
⟨*proof*⟩

lemma *hp-prev-skip-hd-children*:
‹*distinct-mset* (*sum-list* (*map mset-nodes* ch_m) + (*mset-nodes a*)) $\Longrightarrow$ *xa* $\in\# \sum_{\#}$ (*mset-nodes* '# *mset* ch_m) $\Longrightarrow$
xa $\neq$ *node* (*hd* ch_m) $\Longrightarrow$ *hp-prev-children xa* (*a* # ch_m) = *hp-prev-children xa* ch_m›
⟨*proof*⟩

lemma *node-hd-in-sum*[*simp*]: ‹$ch_m \neq$ [] $\Longrightarrow$ *node* (*hd* ch_m) $\in\#$ *sum-list* (*map mset-nodes* ch_m)›
⟨*proof*⟩

lemma *hp-prev-cadr-node*[*simp*]: ‹$ch_m \neq$ [] $\Longrightarrow$ *hp-prev-children* (*node* (*hd* ch_m)) (*a* # ch_m) = *Some a*›
⟨*proof*⟩

lemma *hp-next-children-simps*[*simp*]:
‹*a* = *node x* $\Longrightarrow$ *hp-next-children a* (*x* # *y* # *children*) = *Some y*›
‹*a* $\neq$ *node x* $\Longrightarrow$ *hp-next a x* $\neq$ *None* $\Longrightarrow$ *hp-next-children a* (*x* # *children*) = *hp-next a x*›
‹*a* $\neq$ *node x* $\Longrightarrow$ *hp-next a x* = *None* $\Longrightarrow$ *hp-next-children a* (*x* # *children*) = *hp-next-children a* (*children*)›
⟨*proof*⟩

lemma *hp-prev-children-simps*[*simp*]:
‹*a* = *node y* $\Longrightarrow$ *hp-prev-children a* (*x* # *y* # *children*) = *Some x*›
‹*a* $\neq$ *node y* $\Longrightarrow$ *hp-prev a x* $\neq$ *None* $\Longrightarrow$ *hp-prev-children a* (*x* # *y* # *children*) = *hp-prev a x*›
‹*a* $\neq$ *node y* $\Longrightarrow$ *hp-prev a x* = *None* $\Longrightarrow$ *hp-prev-children a* (*x* # *y* # *children*) = *hp-prev-children a* (*y* # *children*)›
⟨*proof*⟩

lemmas [*simp del*] = *hp-next-children.simps*(*1*) *hp-next.simps*(*1*) *hp-prev.simps*(*1*) *hp-prev-children.simps*(*1*)

lemma *hp-next-children-skip-first-append*[*simp*]:
‹*xa* $\notin\# \sum_{\#}$ (*mset-nodes* '# *mset ch*) $\Longrightarrow$ *hp-next-children xa* (*ch* @ *ch'*) = *hp-next-children xa ch'*›

⟨proof⟩

lemma *hp-prev-children-skip-first-append*[*simp*]:
‹$xa \notin\# \sum_{\#}$ (*mset-nodes* '# *mset ch*) $\Longrightarrow$ $xa \neq$ *node m* $\Longrightarrow$ *hp-prev-children xa* (*ch* @ *m* # *ch'*) = *hp-prev-children xa* (*m*#*ch'*)›
⟨proof⟩

lemma *hp-prev-children-skip-Cons*[*simp*]:
‹$xa \notin\# \sum_{\#}$ (*mset-nodes* '# *mset ch'*) $\Longrightarrow$ $xa \in\#$ *mset-nodes m* $\Longrightarrow$ *hp-prev-children xa* (*m* # *ch'*) = *hp-prev xa m*›
⟨proof⟩

definition *hp-child-children* **where**
‹*hp-child-children a* = *option-hd o* (*List.map-filter* (*hp-child a*))›

lemma *hp-child-children-Cons-if*:
‹*hp-child-children a* (*x* # *y*) = (*if hp-child a x* = *None then hp-child-children a y else hp-child a x*)›
⟨proof⟩

lemma *hp-child-children-simps*[*simp*]:
‹*hp-child-children a* [] = *None*›
‹*hp-child a x* =*None* $\Longrightarrow$ *hp-child-children a* (*x* # *y*) = *hp-child-children a y*›
‹*hp-child a x* $\neq$ *None* $\Longrightarrow$ *hp-child-children a* (*x* # *y*) = *hp-child a x*›
⟨proof⟩

lemma *hp-child-hp-children-simps2*[*simp*]:
‹$x \neq a$ $\Longrightarrow$ *hp-child x* (*Hp a b child*) = *hp-child-children x child*›
⟨proof⟩

lemma *hp-child-children-None-notin*[*simp*]: ‹$m \notin\# \sum_{\#}$ (*mset-nodes* '# *mset children*) $\Longrightarrow$ *hp-child-children m children* = *None*›
⟨proof⟩

definition *hp-node-children* **where**
‹*hp-node-children a* = *option-hd o* (*List.map-filter* (*hp-node a*))›

lemma *hp-node-children-Cons-if*:
‹*hp-node-children a* (*x* # *y*) = (*if hp-node a x* = *None then hp-node-children a y else hp-node a x*)›
⟨proof⟩

lemma *hp-node-children-simps*[*simp*]:
‹*hp-node-children a* [] = *None*›
‹*hp-node a x* =*None* $\Longrightarrow$ *hp-node-children a* (*x* # *y*) = *hp-node-children a y*›
‹*hp-node a x* $\neq$ *None* $\Longrightarrow$ *hp-node-children a* (*x* # *y*) = *hp-node a x*›
⟨proof⟩

lemma *hp-node-children-simps2*[*simp*]:
‹$x \neq a$ $\Longrightarrow$ *hp-node x* (*Hp a b child*) = *hp-node-children x child*›
⟨proof⟩

lemma *hp-node-children-None-notin2*: ‹*hp-node-children m children* = *None* $\longleftrightarrow$ $m \notin\# \sum_{\#}$ (*mset-nodes* '# *mset children*)›
⟨proof⟩

lemma *hp-node-children-None-notin*[*simp*]: ‹$m \notin\# \sum_{\#}$ (*mset-nodes* '# *mset children*) $\Longrightarrow$ *hp-node-children m children* = *None*›

⟨*proof*⟩

lemma *hp-next-children-hd-simps*[*simp*]:
‹$a = node\ x \Longrightarrow distinct\text{-}mset\ (sum\text{-}list\ (map\ mset\text{-}nodes\ (x\ \#\ children))) \Longrightarrow$
$hp\text{-}next\text{-}children\ a\ (x\ \#\ children) = option\text{-}hd\ children$›
⟨*proof*⟩

lemma *hp-next-children-simps-if*:
‹ $distinct\text{-}mset\ (sum\text{-}list\ (map\ mset\text{-}nodes\ (x\ \#\ children))) \Longrightarrow$
$hp\text{-}next\text{-}children\ a\ (x\ \#\ children) = (if\ a = node\ x\ then\ option\text{-}hd\ children\ else\ case\ hp\text{-}next\ a\ x\ of\ None \Rightarrow hp\text{-}next\text{-}children\ a\ children \mid a \Rightarrow a)$›
⟨*proof*⟩

lemma *hp-next-children-skip-end*[*simp*]:
‹$n \in\#\ mset\text{-}nodes\ a \Longrightarrow n \neq node\ a \Longrightarrow n \notin\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ b) \Longrightarrow$
$distinct\text{-}mset\ (mset\text{-}nodes\ a) \Longrightarrow$
$hp\text{-}next\text{-}children\ n\ (a\ \#\ b) = hp\text{-}next\ n\ a$›
⟨*proof*⟩

lemma *hp-next-children-append2*[*simp*]:
‹$x \neq n \Longrightarrow x \notin\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ ch_m) \Longrightarrow hp\text{-}next\text{-}children\ x\ (Hp\ n\ w_n\ ch_n\ \#\ ch_m) = hp\text{-}next\text{-}children\ x\ ch_n$›
⟨*proof*⟩

lemma *hp-next-children-skip-Cons-append*[*simp*]:
‹$NO\text{-}MATCH\ []\ b \Longrightarrow x \in\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ a) \Longrightarrow$
$distinct\text{-}mset\ (sum\text{-}list\ (map\ mset\text{-}nodes\ (a\ @\ m\ \#\ b))) \Longrightarrow$
$hp\text{-}next\text{-}children\ x\ (a\ @\ m\ \#\ b) = hp\text{-}next\text{-}children\ x\ (a\ @\ m\ \#\ [])$›
⟨*proof*⟩

lemma *hp-next-children-append-single-remove-children*:
‹$NO\text{-}MATCH\ []\ ch_m \Longrightarrow x \in\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ a) \Longrightarrow$
$distinct\text{-}mset\ (sum\text{-}list\ (map\ mset\text{-}nodes\ (a\ @\ [Hp\ m\ w_m\ ch_m]))) \Longrightarrow$
$map\text{-}option\ node\ (hp\text{-}next\text{-}children\ x\ (a\ @\ [Hp\ m\ w_m\ ch_m])) =$
$map\text{-}option\ node\ (hp\text{-}next\text{-}children\ x\ (a\ @\ [Hp\ m\ w_m\ []]))$›
⟨*proof*⟩

lemma *hp-prev-children-first-child*[*simp*]:
‹$m \neq n \Longrightarrow n \notin\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ b) \Longrightarrow n \notin\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ ch_n) \Longrightarrow$
$n \in\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ child) \Longrightarrow$
$hp\text{-}prev\text{-}children\ n\ (Hp\ m\ w_m\ child\ \#\ b) = hp\text{-}prev\text{-}children\ n\ child$›
⟨*proof*⟩

lemma *hp-prev-children-skip-last-append*[*simp*]:
‹$NO\text{-}MATCH\ []\ ch' \Longrightarrow$
$distinct\text{-}mset\ (sum\text{-}list\ (map\ mset\text{-}nodes\ (ch\ @ch'))) \Longrightarrow$
$xa \notin\#\ \sum_{\#}\ (mset\text{-}nodes\ `\#\ mset\ ch') \Longrightarrow xa \in\#\ \sum_{\#}\ (mset\text{-}nodes\ `\#\ mset\ (ch\ )) \Longrightarrow hp\text{-}prev\text{-}children\ xa\ (ch\ @\ ch') = hp\text{-}prev\text{-}children\ xa\ (ch)$›
⟨*proof*⟩

lemma *hp-prev-children-Cons-append-found*[*simp*]:
‹$m \notin\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ a) \Longrightarrow m \notin\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ ch) \Longrightarrow m \notin\#\ sum\text{-}list\ (map\ mset\text{-}nodes\ b) \Longrightarrow hp\text{-}prev\text{-}children\ m\ (a\ @\ Hp\ m\ w_m\ ch\ \#\ b) = option\text{-}last\ a$›
⟨*proof*⟩

lemma *hp-prev-children-append-single-remove-children*:
‹$\mathit{NO\text{-}MATCH}\ []\ ch_m \Longrightarrow x \in\#\ \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ a) \Longrightarrow$
$\mathit{distinct\text{-}mset}\ (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ (\mathit{Hp}\ m\ w_m\ ch_m\ \#\ a))) \Longrightarrow$
$\mathit{map\text{-}option}\ \mathit{node}\ (\mathit{hp\text{-}prev\text{-}children}\ x\ (\mathit{Hp}\ m\ w_m\ ch_m\ \#\ a)) =$
$\mathit{map\text{-}option}\ \mathit{node}\ (\mathit{hp\text{-}prev\text{-}children}\ x\ (\mathit{Hp}\ m\ w_m\ []\ \#\ a))$›
⟨*proof*⟩

lemma *map-option-skip-in-child*:
‹$\mathit{distinct\text{-}mset}\ (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch_m) + (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch_n) + \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ a))) \Longrightarrow m \notin\#\ \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch_m) \Longrightarrow$
$ch_m \neq [] \Longrightarrow$
$\mathit{hp\text{-}prev\text{-}children}\ (\mathit{node}\ (\mathit{hd}\ ch_m))\ (a\ @\ [\mathit{Hp}\ m\ w_m\ (\mathit{Hp}\ n\ w_n\ ch_n\ \#\ ch_m)]) = \mathit{Some}\ (\mathit{Hp}\ n\ w_n\ ch_n)$›
⟨*proof*⟩

lemma *hp-child-children-skip-first*[*simp*]:
‹$x \in\#\ \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch') \Longrightarrow$
$\mathit{distinct\text{-}mset}\ (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch) + \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch')) \Longrightarrow$
$\mathit{hp\text{-}child\text{-}children}\ x\ (ch\ @\ ch') = \mathit{hp\text{-}child\text{-}children}\ x\ ch'$›
⟨*proof*⟩

lemma *hp-child-children-skip-last*[*simp*]:
‹$x \in\#\ \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch) \Longrightarrow$
$\mathit{distinct\text{-}mset}\ (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch) + \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ ch')) \Longrightarrow$
$\mathit{hp\text{-}child\text{-}children}\ x\ (ch\ @\ ch') = \mathit{hp\text{-}child\text{-}children}\ x\ ch$›
⟨*proof*⟩

lemma *hp-child-children-skip-last-in-first*:
‹$\mathit{distinct\text{-}mset}\ (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ (\mathit{Hp}\ m\ w_m\ (\mathit{Hp}\ n\ w_n\ ch_n\ \#\ ch_m)\ \#\ b))) \Longrightarrow$
$\mathit{hp\text{-}child\text{-}children}\ n\ (\mathit{Hp}\ m\ w_m\ (\mathit{Hp}\ n\ w_n\ ch_n\ \#\ ch_m)\ \#\ b) = \mathit{hp\text{-}child}\ n\ (\mathit{Hp}\ m\ w_m\ (\mathit{Hp}\ n\ w_n\ ch_n\ \#\ ch_m))$›
⟨*proof*⟩

lemma *hp-child-children-hp-child*[*simp*]: ‹$\mathit{hp\text{-}child\text{-}children}\ x\ [a] = \mathit{hp\text{-}child}\ x\ a$›
⟨*proof*⟩

lemma *hp-next-children-last*[*simp*]:
‹$\mathit{distinct\text{-}mset}\ (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ a)) \Longrightarrow a \neq [] \Longrightarrow$
$\mathit{hp\text{-}next\text{-}children}\ (\mathit{node}\ (\mathit{last}\ a))\ (a\ @\ b) = \mathit{option\text{-}hd}\ b$›
⟨*proof*⟩

lemma *hp-next-children-skip-last-not-last*:
‹$\mathit{distinct\text{-}mset}\ (\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ a) + \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ b)) \Longrightarrow$
$a \neq [] \Longrightarrow$
$x \neq \mathit{node}\ (\mathit{last}\ a) \Longrightarrow x \in\#\ \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{mset\text{-}nodes}\ a) \Longrightarrow$
$\mathit{hp\text{-}next\text{-}children}\ x\ (a\ @\ b) = \mathit{hp\text{-}next\text{-}children}\ x\ a$›
⟨*proof*⟩

lemma *hp-node-children-append-case*:
‹$\mathit{hp\text{-}node\text{-}children}\ x\ (a\ @\ b) = (\mathit{case}\ \mathit{hp\text{-}node\text{-}children}\ x\ a\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{hp\text{-}node\text{-}children}\ x\ b \mid x \Rightarrow x)$›
⟨*proof*⟩

lemma *hp-node-children-append*[*simp*]:
‹*hp-node-children x a* = *None* ⟹ *hp-node-children x* (*a* @ *b*) = *hp-node-children x b*›
‹*hp-node-children x a* ≠ *None* ⟹ *hp-node-children x* (*a* @ *b*) = *hp-node-children x a*›
⟨*proof*⟩

lemma *ex-hp-node-children-Some-in-mset-nodes*:
‹(∃ *y*. *hp-node-children xa a* = *Some y*) ⟷ *xa* ∈# *sum-list* (*map mset-nodes a*)›
⟨*proof*⟩

hide-const (**open**) *NEMonad.ASSERT NEMonad.RETURN NEMonad.SPEC*

lemma *hp-node-node-itself*[*simp*]: ‹*hp-node* (*node x2*) *x2* = *Some x2*›
⟨*proof*⟩

lemma *hp-child-hd*[*simp*]: ‹*hp-child x1* (*Hp x1 x2 x3*) = *option-hd x3*›
⟨*proof*⟩

lemma *drop-is-single-iff*: ‹*drop e xs* = [*a*] ⟷ *last xs* = *a* ∧ *e* = *length xs* − *1* ∧ *xs* ≠ []›
⟨*proof*⟩

lemma *distinct-mset-mono'*: ‹*distinct-mset D* ⟹ *D'* ⊆# *D* ⟹ *distinct-mset D'*›
⟨*proof*⟩

context *pairing-heap-assms*
begin

lemma *pass*$_1$*-append-even*: ‹*even* (*length xs*) ⟹ *pass*$_1$ (*xs* @ *ys*) = *pass*$_1$ *xs* @ *pass*$_1$ *ys*›
⟨*proof*⟩

lemma *pass*$_2$*-None-iff*[*simp*]: ‹*pass*$_2$ *list* = *None* ⟷ *list* = []›
⟨*proof*⟩

lemma *last-pass*$_1$[*simp*]: *odd* (*length xs*) ⟹ *last* (*pass*$_1$ *xs*) = *last xs*
⟨*proof*⟩
end

lemma *get-min2-alt-def*: ‹*get-min2* (*Some h*) = *node h*›
⟨*proof*⟩

fun *hp-parent* :: ‹*'a* ⇒ (*'a*, *'b*) *hp* ⇒ (*'a*, *'b*)*hp option*› **where**
‹*hp-parent n* (*Hp a sc* (*x* # *children*)) = (*if n* = *node x then Some* (*Hp a sc* (*x* # *children*)) *else map-option the* (*option-hd* (*filter* ((≠) *None*) (*map* (*hp-parent n*) (*x*#*children*)))))› |
‹*hp-parent n* - = *None*›

definition *hp-parent-children* :: ‹*'a* ⇒ (*'a*, *'b*) *hp list* ⇒ (*'a*, *'b*)*hp option*› **where**
‹*hp-parent-children n xs* = *map-option the* (*option-hd* (*filter* ((≠) *None*) (*map* (*hp-parent n*) *xs*)))›

lemma *hp-parent-None-notin*[*simp*]: ‹*m* ∉# *mset-nodes a* ⟹ *hp-parent m a* = *None*›
⟨*proof*⟩

lemma *hp-parent-children-None-notin*[*simp*]: ‹(*m*) ∉# *sum-list* (*map mset-nodes a*) ⟹ *hp-parent-children m a* = *None*›

⟨*proof*⟩

lemma *hp-parent-children-cons*: ‹*hp-parent-children a* (*x* # *children*) = (*case hp-parent a x of None* ⇒ *hp-parent-children a children* | *Some a* ⇒ *Some a*)›
⟨*proof*⟩

lemma *hp-parent-simps-if*:
‹*hp-parent n* (*Hp a sc* (*x* # *children*)) = (*if n* = *node x then Some* (*Hp a sc* (*x* # *children*)) *else hp-parent-children n* (*x*#*children*))›
⟨*proof*⟩

lemmas [*simp del*] = *hp-parent.simps*(*1*)

lemma *hp-parent-simps*:
‹*n* = *node x* ⟹ *hp-parent n* (*Hp a sc* (*x* # *children*)) = *Some* (*Hp a sc* (*x* # *children*))›
‹*n* ≠ *node x* ⟹ *hp-parent n* (*Hp a sc* (*x* # *children*)) = *hp-parent-children n* (*x* # *children*)›
⟨*proof*⟩

lemma *hp-parent-itself*[*simp*]: ‹*distinct-mset* (*mset-nodes x*) ⟹ *hp-parent* (*node x*) *x* = *None*›
⟨*proof*⟩

lemma *hp-parent-children-itself*[*simp*]:
‹*distinct-mset* (*mset-nodes x* + *sum-list* (*map mset-nodes children*)) ⟹ *hp-parent-children* (*node x*) (*x* # *children*) = *None*›
⟨*proof*⟩

lemma *hp-parent-in-nodes*: ‹*hp-parent n x* ≠ *None* ⟹ *node* (*the* (*hp-parent n x*)) ∈# *mset-nodes x*›
⟨*proof*⟩

lemma *hp-parent-children-Some-iff*:
‹*hp-parent-children a xs* = *Some y* ⟷ (∃ *u b as*. *xs* = *u* @ *b* # *as* ∧ (∀ *x*∈*set u*. *hp-parent a x* = *None*) ∧ *hp-parent a b* = *Some y*)›
⟨*proof*⟩

lemma *hp-parent-children-in-nodes*:
‹*hp-parent-children b xs* ≠ *None* ⟹ *node* (*the* (*hp-parent-children b xs*)) ∈# $\sum_{\#}$ (*mset-nodes* '# *mset xs*)›
⟨*proof*⟩

lemma *hp-parent-hp-child*:
‹*distinct-mset* ((*mset-nodes* (*a*::('*a*,*nat*)*hp*))) ⟹ *hp-child n a* ≠ *None* ⟹ *map-option node* (*hp-parent* (*node* (*the* (*hp-child n a*))) *a*) = *Some n*›
⟨*proof*⟩

lemma *hp-child-hp-parent*:
‹*distinct-mset* ((*mset-nodes* (*a*::('*a*,*nat*)*hp*))) ⟹ *hp-parent n a* ≠ *None* ⟹ *map-option node* (*hp-child* (*node* (*the* (*hp-parent n a*))) *a*) = *Some n*›
⟨*proof*⟩

lemma *hp-parent-children-append-case*:
‹*hp-parent-children a* (*xs* @ *ys*) = (*case hp-parent-children a xs of None* ⇒ *hp-parent-children a ys* | *Some a* ⇒ *Some a*)›
⟨*proof*⟩

lemma *hp-parent-children-append-skip-first*[*simp*]:

‹$a \notin\# \sum_{\#}$ (*mset-nodes* ‘# *mset xs*) $\Longrightarrow$ *hp-parent-children a* (*xs* @ *ys*) = *hp-parent-children a ys*›
⟨*proof*⟩

lemma *hp-parent-children-append-skip-second*[*simp*]:
‹$a \notin\# \sum_{\#}$ (*mset-nodes* ‘# *mset ys*) $\Longrightarrow$ *hp-parent-children a* (*xs* @ *ys*) = *hp-parent-children a xs*›
⟨*proof*⟩

lemma *hp-parent-simps-single-if*:
‹*hp-parent n* (*Hp a sc* (*children*)) =
(*if children* = [] *then None else if n* = *node* (*hd children*) *then Some* (*Hp a sc* (*children*))
else hp-parent-children n children)›
⟨*proof*⟩

lemma *hp-parent-children-remove-key-children*:
‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* ‘# *mset xs*)) $\Longrightarrow$ *hp-parent-children a* (*remove-key-children a xs*) = *None*›
⟨*proof*⟩

lemma *remove-key-children-notin-unchanged*[*simp*]: ‹$x \notin\#$ *sum-list* (*map mset-nodes c*) $\Longrightarrow$ *remove-key-children x c* = *c*›
⟨*proof*⟩

lemma *remove-key-notin-unchanged*[*simp*]: ‹$x \notin\#$ *mset-nodes c* $\Longrightarrow$ *remove-key x c* = *Some c*›
⟨*proof*⟩

lemma *remove-key-remove-all*: ‹$k \notin\# \sum_{\#}$ (*mset-nodes* ‘# *mset* (*remove-key-children k c*))›
⟨*proof*⟩

lemma *hd-remove-key-node-same*: ‹$c \neq [] \Longrightarrow$ *remove-key-children k c* $\neq [] \Longrightarrow$
node (*hd* (*remove-key-children k c*)) = *node* (*hd c*) $\longleftrightarrow$ *node* (*hd c*) $\neq k$›
⟨*proof*⟩

lemma *hd-remove-key-node-same'*: ‹$c \neq [] \Longrightarrow$ *remove-key-children k c* $\neq [] \Longrightarrow$
node (*hd c*) = *node* (*hd* (*remove-key-children k c*)) $\longleftrightarrow$ *node* (*hd c*) $\neq k$›
⟨*proof*⟩

lemma *remove-key-children-node-hd*[*simp*]: ‹$c \neq [] \Longrightarrow$ *remove-key-children* (*node* (*hd c*)) *c*= *remove-key-children* (*node* (*hd c*)) (*tl c*)›
⟨*proof*⟩

lemma *remove-key-children-alt-def*:
‹*remove-key-children k xs* = *map* (λx. *case x of Hp a b c* $\Rightarrow$ *Hp a b* (*remove-key-children k c*)) (*filter* (λn. *node n* $\neq k$) *xs*)›
⟨*proof*⟩

lemma *not-orig-notin-remove-key*: ‹$b \notin\#$ *sum-list* (*map mset-nodes xs*) $\Longrightarrow$
$b \notin\#$ *sum-list* (*map mset-nodes* (*remove-key-children a xs*))›
⟨*proof*⟩

lemma *hp-parent-None-notin-same-hd*[*simp*]: ‹$b \notin\#$ *sum-list* (*map mset-nodes x3*) $\Longrightarrow$ *hp-parent b* (*Hp b x2 x3*) = *None*›
⟨*proof*⟩

lemma *hp-parent-children-remove-key-children*:
‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* ‘# *mset xs*)) $\Longrightarrow a \neq b \Longrightarrow$ *hp-parent-children b* (*remove-key-children*

a xs) = *hp-parent-children b xs*›
⟨*proof*⟩

lemma *hp-parent-remove-key*:
‹*distinct-mset* ((*mset-nodes xs*)) ⟹ *a* ≠ *node xs* ⟹ *hp-parent a* (*the* (*remove-key a xs*)) = *None*›
⟨*proof*⟩

lemma *find-key-children-None-or-itself*[*simp*]:
‹*find-key-children a h* ≠ *None* ⟹ *node* (*the* (*find-key-children a h*)) = *a*›
⟨*proof*⟩

lemma *find-key-None-or-itself*[*simp*]:
‹*find-key a h* ≠ *None* ⟹ *node* (*the* (*find-key a h*)) = *a*›
⟨*proof*⟩

lemma *find-key-children-notin*[*simp*]:
‹*a* ∉# $\sum_{\#}$ (*mset-nodes* ‘# *mset xs*) ⟹ *find-key-children a xs* = *None*›
⟨*proof*⟩

lemma *find-key-notin*[*simp*]:
‹*a* ∉# *mset-nodes h* ⟹ *find-key a h* = *None*›
⟨*proof*⟩

lemma *mset-nodes-find-key-children-subset*:
‹*find-key-children a h* ≠ *None* ⟹ *mset-nodes* (*the* (*find-key-children a h*)) ⊆# $\sum_{\#}$ (*mset-nodes* ‘# *mset h*)›
⟨*proof*⟩

lemma *hp-parent-None-iff-children-None*:
‹*hp-parent z* (*Hp x n c*) = *None* ⟷ (*c* ≠ [] ⟶ *z* ≠ *node* (*hd c*)) ∧ *hp-parent-children* (*z*) *c* = *None*›
⟨*proof*⟩

lemma *mset-nodes-find-key-subset*:
‹*find-key a h* ≠ *None* ⟹ *mset-nodes* (*the* (*find-key a h*)) ⊆# *mset-nodes h*›
⟨*proof*⟩

lemma *find-key-none-iff*[*simp*]:
‹*find-key-children a h* = *None* ⟷ *a* ∉# $\sum_{\#}$ (*mset-nodes* ‘# *mset h*)›
⟨*proof*⟩

lemma *find-key-noneD*:
‹*find-key-children a h* = *Some x* ⟹ *a* ∈# $\sum_{\#}$ (*mset-nodes* ‘# *mset h*)›
⟨*proof*⟩

lemma *hp-parent-children-hd-None*[*simp*]:
‹*xs* ≠ [] ⟹ *distinct-mset* ($\sum_{\#}$ (*mset-nodes* ‘# *mset xs*)) ⟹ *hp-parent-children* (*node* (*hd xs*)) *xs* = *None*›
⟨*proof*⟩

lemma *hp-parent-hd-None*[*simp*]:
‹*x* ∉# ($\sum_{\#}$ (*mset-nodes* ‘# *mset xs*)) ⟹*x* ∉# *sum-list* (*map mset-nodes c*) ⟹ *hp-parent-children x* (*Hp x n c* # *xs*) = *None*›
⟨*proof*⟩

lemma *hp-parent-none-children*: ‹*hp-parent-children z c = None* ⟹
hp-parent z (*Hp x n c*) = *Some x2a* ⟷ (*c* ≠ [] ∧ *z* = *node* (*hd c*) ∧ *x2a* = *Hp x n c*)›
⟨*proof*⟩

lemma *hp-parent-children-remove-key-children*:
‹*distinct-mset* (∑# (*mset-nodes* '# *mset xs*)) ⟹ *a* ≠ *b* ⟹ *hp-parent-children b* (*remove-key-children a xs*) =
(*if find-key-children b xs* ≠ *None then None else hp-parent-children b xs*)›
⟨*proof*⟩

lemma *in-the-default-empty-iff*: ‹*b* ∈# *the-default* {#} *M* ⟷ *M* ≠ *None* ∧ *b* ∈# *the M*›
⟨*proof*⟩

lemma *remove-key-children-hd-tl*: ‹*distinct-mset* (*sum-list* (*map mset-nodes c*)) ⟹ *c* ≠ [] ⟹ *remove-key-children* (*node* (*hd c*)) (*tl c*) = *tl c*›
⟨*proof*⟩

lemma *in-find-key-children-notin-remove-key*:
‹*find-key-children k c* = *Some x2* ⟹ *distinct-mset* (∑# (*mset-nodes* '# *mset c*)) ⟹
b ∈# *mset-nodes x2* ⟹
b ∉# ∑#(*mset-nodes* '# *mset* (*remove-key-children k c*))›
⟨*proof*⟩

lemma *hp-parent-children-None-hp-parent-iff*: ‹*hp-parent-children b list* = *None* ⟹ *hp-parent b* (*Hp x n list*) = *Some x2a* ⟷ *list* ≠ [] ∧ *node* (*hd list*) = *b* ∧ *x2a* = *Hp x n list*›
⟨*proof*⟩

lemma *hp-parent-children-not-hd-node*:
‹*distinct-mset* (∑# (*mset-nodes* '# *mset c*)) ⟹ *node* (*hd c*) = *node x2a* ⟹ *c* ≠ [] ⟹ *remove-key-children* (*node x2a*) *c* ≠ [] ⟹
hp-parent-children (*node* (*hd* (*remove-key-children* (*node x2a*) *c*))) *c* = *Some x2a* ⟹ *False*›
⟨*proof*⟩

lemma *hp-parent-children-hd-tl-None*[*simp*]: ‹*distinct-mset* (∑# (*mset-nodes* '# *mset c*)) ⟹ *c* ≠ [] ⟹ *a* ∈ *set* (*tl c*)⟹ *hp-parent-children* (*node a*) *c* = *None*›
⟨*proof*⟩

lemma *hp-parent-hp-parent-remove-key-not-None-same*:
assumes ‹*distinct-mset* (∑# (*mset-nodes* '# *mset c*))› **and**
‹*x* ∉# ∑# (*mset-nodes* '# *mset c*)› **and**
‹*hp-parent b* (*Hp x n c*) = *Some x2a*› ‹*b* ∉# *mset-nodes x2a*›
‹*hp-parent b* (*Hp x n* (*remove-key-children k c*)) = *Some x2b*›
shows ‹*remove-key k x2a* ≠ *None* ∧ (*case remove-key k x2a of Some a* ⇒ (*x2b*) = *a* | *None* ⇒ *node x2a* = *k*)›
⟨*proof*⟩

lemma *in-remove-key-children-changed*: ‹*k* ∈# *sum-list* (*map mset-nodes c*) ⟹ *remove-key-children k c* ≠ *c*›
⟨*proof*⟩

lemma *hp-parent-in-nodes2*: ‹*hp-parent* (*z*) *xs* = *Some a* ⟹ *node a* ∈# *mset-nodes xs*›
⟨*proof*⟩

lemma *hp-parent-children-in-nodes2*: ‹*hp-parent-children z xs* = *Some a* ⟹ *node a* ∈# $\sum_{\#}$ (*mset-nodes* '# *mset xs*)›
⟨*proof*⟩

lemma *hp-next-in-nodes2*: ‹*hp-next* (*z*) *xs* = *Some a* ⟹ *node a* ∈# *mset-nodes xs*›
⟨*proof*⟩

lemma *hp-next-children-in-nodes2*: ‹*hp-next-children* (*z*) *xs* = *Some a* ⟹ *node a* ∈# $\sum_{\#}$ (*mset-nodes* '# *mset xs*)›
⟨*proof*⟩

lemma *in-remove-key-changed*: ‹*remove-key k a* ≠ *None* ⟹ *a* = *the* (*remove-key k a*) ⟷ *k* ∉# *mset-nodes a*›
⟨*proof*⟩

lemma *node-remove-key-children-in-mset-nodes*: ‹$\sum_{\#}$ (*mset-nodes* '# *mset* (*remove-key-children k c*)) ⊆# ($\sum_{\#}$ (*mset-nodes* '# *mset c*))›
⟨*proof*⟩

lemma *remove-key-children-hp-parent-children-hd-None*: ‹*remove-key-children k c* = *a* # *list* ⟹
distinct-mset (*sum-list* (*map mset-nodes c*)) ⟹
hp-parent-children (*node a*) (*a* # *list*) = *None*›
⟨*proof*⟩

lemma *hp-next-not-same-node*: ‹*distinct-mset* (*mset-nodes b*) ⟹ *hp-next x b* = *Some y* ⟹ *x* ≠ *node y*›
⟨*proof*⟩

lemma *hp-next-children-not-same-node*: ‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* '# *mset c*)) ⟹ *hp-next-children x c* = *Some y* ⟹ *x* ≠ *node y*›
⟨*proof*⟩

lemma *hp-next-children-hd-is-hd-tl*: ‹*c* ≠ [] ⟹ *distinct-mset* ($\sum_{\#}$ (*mset-nodes* '# *mset c*)) ⟹ *hp-next-children* (*node* (*hd c*)) *c* = *option-hd* (*tl c*)›
⟨*proof*⟩

lemma *hp-parent-children-remove-key-children-other*:
assumes ‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* '# *mset xs*))›
shows ‹*hp-parent-children b* (*remove-key-children a xs*) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key-children a xs*))) *then None*
else if map-option node (*hp-next-children a xs*) = *Some b then map-option* (*the o remove-key a*) (*hp-parent-children a xs*)
else map-option (*the o remove-key a*) (*hp-parent-children b xs*))›
⟨*proof*⟩

lemma *hp-parent-remove-key-other*:
assumes ‹*distinct-mset* ((*mset-nodes xs*))› ‹(*remove-key a xs*) ≠ *None*›
shows ‹*hp-parent b* (*the* (*remove-key a xs*)) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key a xs*))) *then None*
else if map-option node (*hp-next a xs*) = *Some b then map-option* (*the o remove-key a*) (*hp-parent a xs*)
else map-option (*the o remove-key a*) (*hp-parent b xs*))›
⟨*proof*⟩

lemma *hp-prev-in-nodes*: ‹*hp-prev k c≠ None* ⟹ *node* (*the* (*hp-prev k c*)) ∈# ((*mset-nodes c*))›
⟨*proof*⟩

lemma *hp-prev-children-in-nodes*: ‹*hp-prev-children k c≠ None* ⟹ *node* (*the* (*hp-prev-children k c*)) ∈# (∑# (*mset-nodes* '# *mset c*))›
⟨*proof*⟩

lemma *hp-next-children-notin-end*:
‹*distinct-mset* (∑# (*mset-nodes* '# *mset* (*x#xs*))) ⟹ *hp-next-children a xs* = *None* ⟹ *hp-next-children a* (*x* # *xs*) = (*if a* = *node x then option-hd xs else hp-next a x*)›
⟨*proof*⟩

lemma *hp-next-children-remove-key-children-other*:
fixes *xs* :: ('*b*, '*a*) *hp list*
assumes ‹*distinct-mset* (∑# (*mset-nodes* '# *mset xs*))›
shows ‹*hp-next-children b* (*remove-key-children a xs*) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key-children a xs*))) *then None*
else if map-option node (*hp-prev-children a xs*) = *Some b then* (*hp-next-children a xs*)
else map-option (*the o remove-key a*) (*hp-next-children b xs*))›
⟨*proof*⟩

lemma *hp-next-remove-key-other*:
assumes ‹*distinct-mset* (*mset-nodes xs*)› ‹*remove-key a xs* ≠ *None*›
shows ‹*hp-next b* (*the* (*remove-key a xs*)) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key a xs*))) *then None*
else if map-option node (*hp-prev a xs*) = *Some b then* (*hp-next a xs*)
else map-option (*the o remove-key a*) (*hp-next b xs*))›
⟨*proof*⟩

lemma *hp-prev-children-cons-if*:
‹*hp-prev-children b* (*a* # *xs*) = (*if map-option node* (*option-hd xs*) = *Some b then Some a*
else (*case hp-prev-children b* (*hps a*) *of None* ⇒ *hp-prev-children b xs* | *Some a* ⇒ *Some a*))›
⟨*proof*⟩

lemma *hp-prev-children-remove-key-children-other*:
assumes ‹*distinct-mset* (∑# (*mset-nodes* '# *mset xs*))›
shows ‹*hp-prev-children b* (*remove-key-children a xs*) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key-children a xs*))) *then None*
else if map-option node (*hp-next-children a xs*) = *Some b then* (*hp-prev-children a xs*)
else map-option (*the o remove-key a*) (*hp-prev-children b xs*))›
⟨*proof*⟩

lemma *hp-prev-remove-key-other*:
assumes ‹*distinct-mset* (*mset-nodes xs*)› ‹*remove-key a xs* ≠ *None*›
shows ‹*hp-prev b* (*the* (*remove-key a xs*)) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key a xs*))) *then None*
else if map-option node (*hp-next a xs*) = *Some b then* (*hp-prev a xs*)
else map-option (*the o remove-key a*) (*hp-prev b xs*))›
⟨*proof*⟩

lemma *hp-next-find-key-children*:
‹*distinct-mset* (∑# (*mset-nodes* '# *mset h*)) ⟹ *find-key-children a h* ≠ *None* ⟹
x ∈# *mset-nodes* (*the* (*find-key-children a h*)) ⟹ *x* ≠*a* ⟹
hp-next x (*the* (*find-key-children a h*)) = *hp-next-children x h*›

⟨proof⟩

lemma *hp-next-find-key*:
‹distinct-mset (mset-nodes h) ⟹ find-key a h ≠ None ⟹ x ∈# mset-nodes (the (find-key a h)) ⟹ x ≠a ⟹
hp-next x (the (find-key a h)) = hp-next x h›
⟨proof⟩

lemma *hp-next-find-key-itself*:
‹distinct-mset (mset-nodes h) ⟹ (find-key a h) ≠ None ⟹ hp-next a (the (find-key a h)) = None›
⟨proof⟩

lemma *hp-prev-find-key-children*:
‹distinct-mset (∑# (mset-nodes '# mset h)) ⟹ find-key-children a h ≠ None ⟹
x ∈# mset-nodes (the (find-key-children a h)) ⟹ x ≠a ⟹
hp-prev x (the (find-key-children a h)) = hp-prev-children x h›
⟨proof⟩

lemma *hp-prev-find-key*:
‹distinct-mset (mset-nodes h) ⟹ find-key a h ≠ None ⟹ x ∈# mset-nodes (the (find-key a h)) ⟹ x ≠a ⟹
hp-prev x (the (find-key a h)) = hp-prev x h›
⟨proof⟩

lemma *hp-prev-find-key-itself*:
‹distinct-mset (mset-nodes h) ⟹ (find-key a h) ≠ None ⟹ hp-prev a (the (find-key a h)) = None›
⟨proof⟩

lemma *hp-child-find-key-children*:
‹distinct-mset (∑# (mset-nodes '# mset h)) ⟹ find-key-children a h ≠ None ⟹
x ∈# mset-nodes (the (find-key-children a h)) ⟹
hp-child x (the (find-key-children a h)) = hp-child-children x h›
⟨proof⟩

lemma *hp-child-find-key*:
‹distinct-mset (mset-nodes h) ⟹ find-key a h ≠ None ⟹ x ∈# mset-nodes (the (find-key a h)) ⟹
hp-child x (the (find-key a h)) = hp-child x h›
⟨proof⟩

lemma *find-remove-children-mset-nodes-full*:
‹distinct-mset (∑# (mset-nodes '# mset h)) ⟹ find-key-children a h = Some x ⟹
(∑# (mset-nodes '# mset (remove-key-children a h))) + mset-nodes x = ∑# (mset-nodes '# mset h)›
⟨proof⟩

lemma *find-remove-mset-nodes-full*:
‹distinct-mset (mset-nodes h) ⟹ remove-key a h = Some y ⟹
find-key a h = Some ya ⟹ (mset-nodes y + mset-nodes ya) = mset-nodes h›
⟨proof⟩

lemma *in-remove-key-in-nodes*: ‹remove-key a h ≠ None ⟹ x' ∈# mset-nodes (the (remove-key a h)) ⟹ x' ∈# mset-nodes h›
⟨proof⟩

lemma *in-find-key-in-nodes*: ‹find-key a h ≠ None ⟹ x' ∈# mset-nodes (the (find-key a h)) ⟹ x'

∈# *mset-nodes h*›
⟨*proof*⟩

lemma *in-find-key-notin-remove-key-children*:
‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* '# *mset h*)) $\Longrightarrow$ *find-key-children a h* $\neq$ *None* $\Longrightarrow$ *x* ∈# *mset-nodes* (*the* (*find-key-children a h*))$\Longrightarrow$ *x* ∉# $\sum_{\#}$ (*mset-nodes* '# *mset* (*remove-key-children a h*))›
⟨*proof*⟩

lemma *in-find-key-notin-remove-key*:
‹*distinct-mset* (*mset-nodes h*) $\Longrightarrow$ *find-key a h* $\neq$ *None* $\Longrightarrow$ *remove-key a h* $\neq$ *None* $\Longrightarrow$ *x* ∈# *mset-nodes* (*the* (*find-key a h*))$\Longrightarrow$ *x* ∉# *mset-nodes* (*the* (*remove-key a h*))›
⟨*proof*⟩

lemma *map-option-node-hp-next-remove-key*:
‹*distinct-mset* (*mset-nodes h*) $\Longrightarrow$ *map-option node* (*hp-prev a h*) $\neq$ *Some x'* $\Longrightarrow$ *map-option node* (*hp-next x' h*) =
map-option (λ*x*. *node* (*the* (*remove-key a x*))) (*hp-next x' h*)›
⟨*proof*⟩

lemma *has-prev-still-in-remove-key*: ‹*distinct-mset* (*mset-nodes h*) $\Longrightarrow$ *hp-prev a h* $\neq$ *None* $\Longrightarrow$
remove-key a h $\neq$ *None* $\Longrightarrow$ *node* (*the* (*hp-prev a h*)) ∈# *mset-nodes* (*the* (*remove-key a h*))›
⟨*proof*⟩
lemma *find-key-head-node-iff*: ‹*node h* = *node m'* $\Longrightarrow$ *find-key* (*node m'*) *h* = *Some m'* $\longleftrightarrow$ *h* = *m'*›
⟨*proof*⟩

lemma *map-option-node-hp-prev-remove-key*:
‹*distinct-mset* (*mset-nodes h*) $\Longrightarrow$ *map-option node* (*hp-next a h*) $\neq$ *Some x'* $\Longrightarrow$ *map-option node* (*hp-prev x' h*) =
map-option (λ*x*. *node* (*the* (*remove-key a x*))) (*hp-prev x' h*)›
⟨*proof*⟩

lemma ‹*distinct-mset* (*mset-nodes h*) $\Longrightarrow$ *node y* ∈# *mset-nodes h* $\Longrightarrow$ *find-key* (*node y*) *h* = *Some y*$\Longrightarrow$
mset-nodes (*the* (*find-key* (*node y*) *h*)) = *mset-nodes y*›
⟨*proof*⟩

lemma *distinct-mset-find-node-next*:
‹*distinct-mset* (*mset-nodes h*) $\Longrightarrow$ *find-key n h* = *Some y* $\Longrightarrow$
distinct-mset (*mset-nodes y* + (*if hp-next n h* = *None then* {#} *else* (*mset-nodes* (*the* (*hp-next n h*)))))›
⟨*proof*⟩

lemma *hp-child-node-itself*[*simp*]: ‹*hp-child* (*node a*) *a* = *option-hd* (*hps a*)›
⟨*proof*⟩

lemma *find-key-children-itself-hd*[*simp*]:
‹*find-key-children* (*node a*) [*a*] = *Some a*›
⟨*proof*⟩

lemma *hp-prev-and-next-same-node*:
fixes *y h* :: ‹('*b*, '*a*) *hp*›
assumes ‹*distinct-mset* (*mset-nodes h*)› ‹*hp-prev x' y* $\neq$ *None*›
‹*node yb* = *x'*›
‹*hp-next* (*node y*) *h* = *Some yb*›
‹*find-key* (*node y*) *h* = *Some y*›
shows ‹*False*›

⟨*proof*⟩

lemma *hp-child-children-remove-is-remove-hp-child-children*:
‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* '# *mset c*)) ⟹
hp-child-children b (*c*) ≠ *None* ⟹
hp-parent-children k (*c*) = *None* ⟹
hp-child-children b ((*remove-key-children k c*)) ≠ *None* ⟹
(*hp-child-children b* (*remove-key-children k c*)) = (*remove-key k* (*the* (*hp-child-children b* (*c*))))›
⟨*proof*⟩

lemma *hp-child-remove-is-remove-hp-child*:
‹*distinct-mset* (*mset-nodes* (*Hp x n c*)) ⟹
hp-child b (*Hp x n c*) ≠ *None* ⟹
hp-parent k (*Hp x n c*) = *None* ⟹
remove-key k (*Hp x n c*) ≠ *None* ⟹
hp-child b (*the* (*remove-key k* (*Hp x n c*))) ≠ *None* ⟹
hp-child b (*the* (*remove-key k* (*Hp x n c*))) = *remove-key k* (*the* (*hp-child b* (*Hp x n c*)))›
⟨*proof*⟩

lemma *remove-key-children-itself-hd*[*simp*]: ‹*distinct-mset* (*mset-nodes a* + *sum-list* (*map mset-nodes list*)) ⟹
remove-key-children (*node a*) (*a* # *list*) = *list*›
⟨*proof*⟩

lemma *hp-child-children-remove-key-children-other-helper*:
assumes
K: ‹*hp-child-children b* (*remove-key-children k c*) = *map-option* ((*the* ∘∘ *remove-key*) *k*) (*hp-child-children b c*)› **and**
H: ‹*node x2a* ≠ *b*›
‹*hp-parent k* (*Hp x n c*) = *Some x2a*›
‹*hp-child b* (*Hp x n c*) = *Some y*›
‹*hp-child b* (*Hp x n* (*remove-key-children k c*)) = *Some ya*›
shows
‹*ya* = *the* (*remove-key k y*)›
⟨*proof*⟩

lemma *hp-child-children-remove-key-children-other*:
assumes ‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* '# *mset xs*))›
shows ‹*hp-child-children b* (*remove-key-children a xs*) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key-children a xs*))) *then None*
else if map-option node (*hp-parent-children a xs*) = *Some b then* (*hp-next-children a xs*)
else map-option (*the o remove-key a*) (*hp-child-children b xs*))›
⟨*proof*⟩

lemma *hp-child-remove-key-other*:
assumes ‹*distinct-mset* (*mset-nodes xs*)› ‹*remove-key a xs* ≠ *None*›
shows ‹*hp-child b* (*the* (*remove-key a xs*)) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key a xs*))) *then None*
else if map-option node (*hp-parent a xs*) = *Some b then* (*hp-next a xs*)
else map-option (*the o remove-key a*) (*hp-child b xs*))›
⟨*proof*⟩

abbreviation *hp-score-children* **where**
‹*hp-score-children a xs* ≡ *map-option score* (*hp-node-children a xs*)›

lemma *hp-score-children-remove-key-children-other*:
assumes ‹*distinct-mset* ($\sum_{\#}$ (*mset-nodes* '# *mset xs*))›
shows ‹*hp-score-children b* (*remove-key-children a xs*) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key-children a xs*))) *then None*
else (*hp-score-children b xs*))›
⟨*proof*⟩

abbreviation *hp-score* **where**
‹*hp-score a xs* ≡ *map-option score* (*hp-node a xs*)›

lemma *hp-score-remove-key-other*:
assumes ‹*distinct-mset* (*mset-nodes xs*)› ‹*remove-key a xs* ≠ *None*›
shows ‹*hp-score b* (*the* (*remove-key a xs*)) =
(*if b* ∈# (*the-default* {#} (*map-option mset-nodes* (*find-key a xs*))) *then None*
else (*hp-score b xs*))›
⟨*proof*⟩

lemma *map-option-node-remove-key-iff*:
‹(*h* ≠ *None* ⟹ *distinct-mset* (*mset-nodes* (*the h*))) ⟹ (*h* ≠ *None* ⟹ *remove-key a* (*the h*) ≠ *None*) ⟹
map-option node h = *map-option node* (*map-option* (λ*x*. *the* (*remove-key a x*)) *h*) ⟷ *h* = *None* ∨ (*h* ≠ *None* ∧ *a* ≠ *node* (*the h*))›
⟨*proof*⟩

lemma *sum-next-prev-child-subset*:
‹*distinct-mset* (*mset-nodes h*) ⟹
((*if hp-next n h* = *None then* {#} *else* (*mset-nodes* (*the* (*hp-next n h*)))) +
(*if hp-prev n h* = *None then* {#} *else* (*mset-nodes* (*the* (*hp-prev n h*)))) +
(*if hp-child n h* = *None then* {#} *else* (*mset-nodes* (*the* (*hp-child n h*))))) ⊆# *mset-nodes h*›
⟨*proof*⟩

lemma *distinct-sum-next-prev-child*:
‹*distinct-mset* (*mset-nodes h*) ⟹
distinct-mset ((*if hp-next n h* = *None then* {#} *else* (*mset-nodes* (*the* (*hp-next n h*)))) +
(*if hp-prev n h* = *None then* {#} *else* (*mset-nodes* (*the* (*hp-prev n h*)))) +
(*if hp-child n h* = *None then* {#} *else* (*mset-nodes* (*the* (*hp-child n h*)))))›
⟨*proof*⟩

lemma *node-remove-key-in-mset-nodes*:
‹*remove-key a h* ≠ *None* ⟹ *mset-nodes* (*the* (*remove-key a h*)) ⊆# *mset-nodes h*›
⟨*proof*⟩

lemma *no-relative-ancestor-or-notin*: ‹*hp-parent* (m') *h* = *None* ⟹ *hp-prev* m' *h* = *None* ⟹
hp-next m' *h* = *None* ⟹ m' = *node h* ∨ m' ∉# *mset-nodes h*›
⟨*proof*⟩

lemma *hp-node-in-find-key-children*:
distinct-mset (*sum-list* (*map mset-nodes h*)) ⟹ *find-key-children x h* = *Some* m' ⟹ *a* ∈# *mset-nodes* m' ⟹
hp-node a m' = *hp-node-children a h*
⟨*proof*⟩

lemma *hp-node-in-find-key0*:

distinct-mset (*mset-nodes h*) $\Longrightarrow$ *find-key x h* = *Some m'* $\Longrightarrow$ *a* $\in\#$ *mset-nodes m'* $\Longrightarrow$
hp-node a m' = *hp-node a h*
⟨*proof*⟩

lemma *hp-node-in-find-key*:
distinct-mset (*mset-nodes h*) $\Longrightarrow$ *find-key x h* $\neq$ *None* $\Longrightarrow$ *a* $\in\#$ *mset-nodes* (*the* (*find-key x h*)) $\Longrightarrow$
hp-node a (*the* (*find-key x h*)) = *hp-node a h*
⟨*proof*⟩

context *hmstruct-with-prio*
begin

definition *hmrel* :: ‹(('*a multiset* × ('*a*, '*v*) *hp option*) × ('*a multiset* × '*a multiset* × ('*a* ⇒ '*v*))) *set*›
where
‹*hmrel* = {((ℬ, *xs*), (𝒜, *b*, *w*)). *invar xs* ∧ *distinct-mset b* ∧ 𝒜 = ℬ ∧
((*xs* = *None* ∧ *b* = {#}) ∨
(*xs* ≠ *None* ∧ *b* = *mset-nodes* (*the xs*) ∧
(∀ *v*∈#*b*. *hp-node v* (*the xs*) ≠ *None*) ∧
(∀ *v*∈#*b*. *score* (*the* (*hp-node v* (*the xs*))) = *w v*)))}›

lemma *hp-score-children-iff-hp-score*: ‹*xa* ∈# *sum-list* (*map mset-nodes list*) ⟹ *distinct-mset* (*sum-list* (*map mset-nodes list*)) ⟹
hp-score-children xa list ≠ *None* ⟷ (∃ *x*∈*set list*. *hp-score xa x* ≠ *None* ∧ *hp-score-children xa list* = *hp-score xa x* ∧ (∀ *x*∈*set list* − {*x*}. *hp-score xa x* = *None*))›
⟨*proof*⟩

lemma *hp-score-children-in-iff*: ‹*xa* ∈# *sum-list* (*map mset-nodes list*) ⟹ *distinct-mset* (*sum-list* (*map mset-nodes list*)) ⟹
the (*hp-score-children xa list*) ∈ *A* ⟷ (∃ *x*∈*set list*. *hp-score xa x* ≠ *None* ∧ *the* (*hp-score xa x*) ∈ *A*)›
⟨*proof*⟩

lemma *set-hp-is-hp-score-mset-nodes*:
assumes ‹*distinct-mset* (*mset-nodes a*)›
shows ‹*set-hp a* = (λ*v'*. *the* (*hp-score v' a*)) ‘ *set-mset* (*mset-nodes a*)›
⟨*proof*⟩

definition *mop-get-min2* :: ‹-› **where**
‹*mop-get-min2* = (λ(ℬ, *x*). *do* {
ASSERT (*x* ≠ *None*);
RETURN (*get-min2 x*)
})›

lemma *get-min2-mop-prio-peek-min*:
‹(*xs*, *ys*) ∈ *hmrel* ⟹ *fst ys* ≠ {#} ⟹
mop-get-min2 xs ≤ ⇓(*Id*) (*mop-prio-peek-min ys*)›
⟨*proof*⟩

lemma *get-min2-mop-prio-peek-min2*:
‹(*xs*, *ys*) ∈ *hmrel* ⟹
mop-get-min2 xs ≤ ⇓{(*a*,*b*). (*a*,*b*)∈*Id* ∧ *b* = *get-min2* (*snd xs*)} (*mop-prio-peek-min ys*)›
⟨*proof*⟩

lemma *del-min-None-iff*: ‹*del-min a* = *None* ⟷ *a* = *None* ∨ *hps* (*the a*) = []›
⟨*proof*⟩

lemma *score-hp-node-pass*$_1$: ‹*distinct-mset* (*sum-list* (*map mset-nodes x3*)) ⟹ *score* (*the* (*hp-node-children* *v* (*pass*$_1$ *x3*))) = *score* (*the* (*hp-node-children v x3*))›
⟨*proof*⟩

lemma *node-pass*$_2$*-in-nodes*: ‹*pass*$_2$ *hs* ≠ *None* ⟹ *mset-nodes* (*the* (*pass*$_2$ *hs*)) ⊆# *sum-list* (*map* *mset-nodes hs*)›
⟨*proof*⟩

lemma *score-pass2-same*:
‹*distinct-mset* (*sum-list* (*map mset-nodes x3*)) ⟹ *pass*$_2$ *x3* ≠ *None* ⟹*v* ∈# *sum-list* (*map mset-nodes* *x3*) ⟹
score (*the* (*hp-node v* (*the* (*pass*$_2$ *x3*)))) = *score* (*the* (*hp-node-children v x3*))›
⟨*proof*⟩

lemma *score-hp-node-merge-pairs-same*: ‹*distinct-mset* (*sum-list* (*map mset-nodes x3*)) ⟹ *v* ∈# *sum-list* (*map mset-nodes x3*) ⟹
score (*the* (*hp-node v* (*the* (*merge-pairs x3*)))) = *score* (*the* (*hp-node-children v x3*))›
⟨*proof*⟩
term *mop-get-min2*

definition *mop-hm-pop-min* :: ‹-› **where**
‹*mop-hm-pop-min* = (λ(ℬ, *x*). *do* {
ASSERT (*x* ≠ *None*);
m ← *mop-get-min2* (ℬ, *x*);
RETURN (*m*, (ℬ, *del-min x*))
})›

lemma *get-min2-del-min2-mop-prio-pop-min*:
assumes ‹(*xs*, *ys*) ∈ *hmrel*›
shows ‹*mop-hm-pop-min xs* ≤ ⇓(*Id* $\times_r$ *hmrel*) (*mop-prio-pop-min ys*)›
⟨*proof*⟩

definition *mop-hm-insert* :: ‹-› **where**
‹*mop-hm-insert* = (λ*w v* (ℬ, *xs*). *do* {
ASSERT (*w* ∈# ℬ ∧ (*xs* ≠ *None* ⟶ *w* ∉# *mset-nodes* (*the xs*)));
RETURN (ℬ, *insert w v xs*)
})›

lemma *mop-prio-insert*:
‹(*xs*, *ys*) ∈ *hmrel* ⟹
mop-hm-insert w v xs ≤ ⇓(*hmrel*) (*mop-prio-insert w v ys*)›
⟨*proof*⟩

lemma *find-key-node-itself*[*simp*]: ‹*find-key* (*node y*) *y* = *Some y*›
⟨*proof*⟩

lemma *invar-decrease-key*: ‹*le v x* ⟹
invar (*Some* (*Hp w x x3*)) ⟹ *invar* (*Some* (*Hp w v x3*))›
⟨*proof*⟩

lemma *find-key-children-single*[*simp*]: ‹*find-key-children k* [*x*] = *find-key k x*›
⟨*proof*⟩

lemma *hp-node-find-key-children*:
‹*distinct-mset* (*sum-list* (*map mset-nodes a*)) ⟹ *find-key-children x a* ≠ *None* ⟹

hp-node x (the (find-key-children x a)) ≠ None ⟹
hp-node x (the (find-key-children x a)) = hp-node-children x a›
⟨*proof*⟩

lemma *hp-node-find-key*:
‹*distinct-mset (mset-nodes a) ⟹ find-key x a ≠ None ⟹ hp-score x (the (find-key x a)) ≠ None ⟹*
hp-score x (the (find-key x a)) = hp-score x a›
⟨*proof*⟩

lemma *score-hp-node-link*:
‹*distinct-mset (mset-nodes a + mset-nodes b) ⟹*
map-option score (hp-node w (link a b)) = (case hp-node w a of Some a ⇒ Some (score a)
| - ⇒ map-option score (hp-node w b))›
⟨*proof*⟩

lemma *hp-node-link-none-iff-parents*: ‹*hp-node va (link a b) = None ⟷ hp-node va a = None ∧ hp-node va b = None*›
⟨*proof*⟩

lemma *score-hp-node-link2*:
‹*distinct-mset (mset-nodes a + mset-nodes b) ⟹ (hp-node w (link a b)) ≠ None ⟹*
score (the (hp-node w (link a b))) = (case hp-node w a of Some a ⇒ (score a)
| - ⇒ score (the (hp-node w b)))›
⟨*proof*⟩

definition *mop-hm-decrease-key* :: ‹-› **where**
‹*mop-hm-decrease-key = (λw v (𝓑, xs). do {*
ASSERT (w ∈# 𝓑);
if xs = None then RETURN (𝓑, xs)
else RETURN (𝓑, decrease-key w v (the xs))
})›

lemma *decrease-key-mop-prio-change-weight*:
assumes ‹*(xs, ys) ∈ hmrel*›
shows ‹*mop-hm-decrease-key w v xs ≤ ⇓(hmrel) (mop-prio-change-weight w v ys)*›
⟨*proof*⟩

lemma *$pass_1$-empty-iff*[*simp*]: ‹*$pass_1$ x = [] ⟷ x= []*›
⟨*proof*⟩

lemma *sum-list-map-mset-nodes-empty-iff*[*simp*]: ‹*sum-list (map mset-nodes x3) = {#} ⟷ x3 = []*›
⟨*proof*⟩

lemma *hp-score-link*:
‹*a ∈# mset-nodes h1 ⟹ distinct-mset (mset-nodes h1 + mset-nodes h2) ⟹ hp-score a (link h1 h2) = hp-score a h1*›
⟨*proof*⟩

lemma *hp-score-link-skip-first*[*simp*]:
‹*a ∉# mset-nodes h1 ⟹ hp-score a (link h1 h2) = hp-score a h2*›
⟨*proof*⟩

lemma *hp-score-merge-pairs*:
‹*distinct-mset (sum-list (map mset-nodes ys)) ⟹ merge-pairs ys ≠ None ⟹*
hp-score a (the (merge-pairs (ys))) = hp-score-children a (ys)›

⟨*proof*⟩

definition *decrease-key2* **where**
‹*decrease-key2 a w h* = (*if h* = *None then None else decrease-key a w* (*the h*))›
lemma *hp-mset-rel-def*: ‹*hmrel* = {((ℬ, *h*), (𝒜, *m*, *w*)). *distinct-mset m* ∧ 𝒜=ℬ ∧
(*h* = *None* ⟷ *m* = {#}) ∧
(*m* ≠ {#} ⟶ (*mset-nodes* (*the h*) = *m* ∧ (∀ *a*∈#*m*. *Some* (*w a*) = *hp-score a* (*the h*)) ∧ *invar h*))}›
⟨*proof*⟩

lemma (**in** −)*find-key-None-remove-key-ident*: ‹*find-key a h* = *None* ⟹ *remove-key a h* = *Some h*›
⟨*proof*⟩

lemma *decrease-key2*:
assumes ‹(*x*, *m*) ∈ *hmrel*› ‹(*a*,*a*′)∈*Id*› ‹(*w*,*w*′)∈*Id*› ‹*le w* (*snd* (*snd m*) *a*)›
shows ‹*mop-hm-decrease-key a w x* ≤ ⇓ (*hmrel*) (*mop-prio-change-weight a*′ *w*′ *m*)›
⟨*proof*⟩

end

interpretation *ACIDS*: *hmstruct-with-prio* **where**
le = ‹(≥) :: *nat* ⇒ *nat* ⇒ *bool*› **and**
lt = ‹(>)›
⟨*proof*⟩

end
theory *Relational-Pairing-Heaps*
imports *Pairing-Heaps*
begin

1.1.2 Flat Version of Pairing Heaps

Splitting genealogy to Relations

In this subsection, we replace the tree version by several arrays that represent the relations (parent, child, next, previous) of the same trees.

type-synonym (′*a*, ′*b*) *hp-fun* = ‹((′*a* ⇒ ′*a option*) × (′*a* ⇒ ′*a option*) × (′*a* ⇒ ′*a option*) × (′*a* ⇒ ′*a option*) × (′*a* ⇒ ′*b option*))›

definition *hp-set-all* :: ‹′*a* ⇒ ′*a option* ⇒ ′*a option* ⇒ ′*a option* ⇒ ′*a option* ⇒ ′*b option* ⇒ (′*a*, ′*b*) *hp-fun* ⇒ (′*a*, ′*b*) *hp-fun* › **where**
‹*hp-set-all i prev nxt child par sc* = (λ(*prevs*, *nxts*, *childs*, *parents*, *scores*). (*prevs*(*i*:=*prev*), *nxts*(*i*:=*nxt*), *childs*(*i*:=*child*), *parents*(*i*:=*par*), *scores*(*i*:=*sc*)))›

definition *hp-update-prev* :: ‹′*a* ⇒ ′*a option* ⇒ (′*a*, ′*b*) *hp-fun* ⇒ (′*a*, ′*b*) *hp-fun*› **where**
‹*hp-update-prev i prev* = (λ(*prevs*, *nxts*, *childs*, *parents*, *score*). (*prevs*(*i*:=*prev*), *nxts*, *childs*, *parents*, *score*))›

definition *hp-update-nxt* :: ‹′*a* ⇒ ′*a option* ⇒ (′*a*, ′*b*) *hp-fun* ⇒ (′*a*, ′*b*) *hp-fun*› **where**
‹*hp-update-nxt i nxt* = (λ(*prevs*, *nxts*, *childs*, *parents*, *score*). (*prevs*, *nxts*(*i*:=*nxt*), *childs*, *parents*, *score*))›

definition *hp-update-parents* :: ‹′*a* ⇒ ′*a option* ⇒ (′*a*, ′*b*) *hp-fun* ⇒ (′*a*, ′*b*) *hp-fun*› **where**

‹*hp-update-parents i nxt* = (λ(*prevs*, *nxts*, *childs*, *parents*, *score*). (*prevs*, *nxts*, *childs*, *parents*(*i*:=*nxt*), *score*))›

definition *hp-update-score* :: ‹′*a* ⇒ ′*b option* ⇒ (′*a*, ′*b*) *hp-fun* ⇒ (′*a*, ′*b*) *hp-fun*› **where**
‹*hp-update-score i nxt* = (λ(*prevs*, *nxts*, *childs*, *parents*, *score*). (*prevs*, *nxts*, *childs*, *parents*, *score*(*i*:=*nxt*)))›

fun *hp-read-nxt* :: ‹- ⇒ (′*a*, ′*b*) *hp-fun* ⇒ -› **where** ‹*hp-read-nxt i* (*prevs*, *nxts*, *childs*) = *nxts i*›
fun *hp-read-prev* :: ‹- ⇒ (′*a*, ′*b*) *hp-fun* ⇒ -› **where** ‹*hp-read-prev i* (*prevs*, *nxts*, *childs*) = *prevs i*›
fun *hp-read-child* :: ‹- ⇒ (′*a*, ′*b*) *hp-fun* ⇒ -› **where** ‹*hp-read-child i* (*prevs*, *nxts*, *childs*, *parents*, *scores*) = *childs i*›
fun *hp-read-parent* :: ‹- ⇒ (′*a*, ′*b*) *hp-fun* ⇒ -› **where** ‹*hp-read-parent i* (*prevs*, *nxts*, *childs*, *parents*, *scores*) = *parents i*›
fun *hp-read-score* :: ‹- ⇒ (′*a*, ′*b*) *hp-fun* ⇒ -› **where** ‹*hp-read-score i* (*prevs*, *nxts*, *childs*, *parents*, *scores*) = *scores i*›

It was not entirely clear from the ground up whether we would actually need to have the conditions of emptyness of the previous or the parent. However, these are the only conditions to know whether a node is in the treen or not, so we decided to include them. It is critical to not add that the scores are empty, because this is the only way to track the scores after removing a node.

We initially inlined the definition of *empty-outside*, but the simplifier immediatly hung himself.

definition *empty-outside* :: ‹-› **where**
‹*empty-outside* $\mathcal{V}$ *prevs* = (∀ *x*. *x* ∉# $\mathcal{V}$ ⟶ *prevs x* = *None*)›

definition *encoded-hp-prop* :: ‹′*e multiset* ⇒ (′*e*,′*f*) *hp multiset* ⇒ (′*e*, ′*f*) *hp-fun* ⇒ -› **where**
‹*encoded-hp-prop* $\mathcal{V}$ *m* = (λ(*prevs*,*nxts*,*children*, *parents*, *scores*). *distinct-mset* ($\sum_{\#}$ (*mset-nodes* ‘# *m*)) ∧
set-mset ($\sum_{\#}$ (*mset-nodes* ‘# *m*)) ⊆ *set-mset* $\mathcal{V}$ ∧
(∀ *m*∈# *m*. ∀ *x* ∈# *mset-nodes m*. *prevs x* = *map-option node* (*hp-prev x m*)) ∧
(∀ *m*′∈#*m*. ∀ *x* ∈# *mset-nodes m*′. *nxts x* = *map-option node* (*hp-next x m*′)) ∧
(∀ *m*∈# *m*. ∀ *x* ∈# *mset-nodes m*. *children x* = *map-option node* (*hp-child x m*)) ∧
(∀ *m*∈# *m*. ∀ *x* ∈# *mset-nodes m*. *parents x* = *map-option node* (*hp-parent x m*)) ∧
(∀ *m*∈# *m*. ∀ *x* ∈# *mset-nodes m*. *scores x* = *map-option score* (*hp-node x m*)) ∧
empty-outside ($\sum_{\#}$ (*mset-nodes* ‘# *m*)) *prevs* ∧
empty-outside ($\sum_{\#}$ (*mset-nodes* ‘# *m*)) *parents*)›

lemma *empty-outside-alt-def*: ‹*empty-outside* $\mathcal{V}$ *f* = (*dom f* ∩ *UNIV* − *set-mset* $\mathcal{V}$ = {})›
⟨*proof*⟩

lemma *empty-outside-add-mset*[*simp*]:
‹*f v* = *None* ⟹ *empty-outside* (*add-mset v* $\mathcal{V}$) *f* ⟷ *empty-outside* $\mathcal{V}$ *f*›
⟨*proof*⟩

lemma *empty-outside-notin-None*: ‹*empty-outside* $\mathcal{V}$ *prevs* ⟹ *a* ∉# $\mathcal{V}$ ⟹ *prevs a* = *None*›
⟨*proof*⟩

lemma *empty-outside-update-already-in*[*simp*]: ‹*x* ∈# $\mathcal{V}$ ⟹ *empty-outside* $\mathcal{V}$ (*prevs*(*x* := *a*)) = *empty-outside* $\mathcal{V}$ *prevs*›
⟨*proof*⟩

lemma *encoded-hp-prop-irrelevant*:
assumes ‹*a* ∉# $\sum_{\#}$ (*mset-nodes* ‘# *m*)› **and** ‹*a*∈#$\mathcal{V}$› **and**
‹*encoded-hp-prop* $\mathcal{V}$ *m* (*prevs*, *nxts*, *children*, *parents*, *scores*)›
shows
‹*encoded-hp-prop* $\mathcal{V}$ (*add-mset* (*Hp a sc* []) *m*) (*prevs*, *nxts*(*a*:=*None*), *children*(*a*:=*None*), *parents*,

scores(*a*:=*Some sc*))›
⟨*proof*⟩

lemma *hp-parent-single-child*[*simp*]: ‹*hp-parent* (*node a*) (*Hp m* w_m [*a*]) = *Some* (*Hp m* w_m [*a*])›
⟨*proof*⟩

lemma *hp-parent-children-single-hp-parent*[*simp*]: ‹*hp-parent-children b* [*a*] = *hp-parent b a*›
⟨*proof*⟩

lemma *hp-parent-single-child-If*:
‹*b* ≠ *m* ⟹ *hp-parent b* (*Hp m* w_m (*a* # [])) = (*if b* = *node a then Some* (*Hp m* w_m [*a*]) *else hp-parent b a*)›
⟨*proof*⟩

lemma *hp-parent-single-child-If2*:
‹*distinct-mset* (*add-mset m* (*mset-nodes a*)) ⟹
hp-parent b (*Hp m* w_m (*a* # [])) = (*if b* = *m then None else if b* = *node a then Some* (*Hp m* w_m [*a*]) *else hp-parent b a*)›
⟨*proof*⟩

lemma *hp-parent-single-child-If3*:
‹*distinct-mset* (*add-mset m* (*mset-nodes a* + *sum-list* (*map mset-nodes xs*))) ⟹
hp-parent b (*Hp m* w_m (*a* # *xs*)) = (*if b* = *m then None else if b* = *node a then Some* (*Hp m* w_m (*a* # *xs*)) *else hp-parent-children b* (*a* # *xs*))›
⟨*proof*⟩

lemma *hp-parent-is-first-child*[*simp*]: ‹*hp-parent* (*node a*) (*Hp m* w_m (*a* # ch_m)) = *Some* (*Hp m* w_m (*a* # ch_m))›
⟨*proof*⟩

lemma *hp-parent-children-in-first-child*[*simp*]: ‹*distinct-mset* (*mset-nodes a* + *sum-list* (*map mset-nodes* ch_m)) ⟹
xa ∈# *mset-nodes a* ⟹ *hp-parent-children xa* (*a* # ch_m) = *hp-parent xa a*›
⟨*proof*⟩

lemma *encoded-hp-prop-link*:
fixes ch_m *a prevs parents m*
defines ‹*prevs*′ ≡ (*if* ch_m = [] *then prevs else prevs* (*node* (*hd* ch_m) := *Some* (*node a*)))›
defines ‹*parents*′ ≡ (*if* ch_m = [] *then parents else parents* (*node* (*hd* ch_m) := *None*))›
assumes ‹*encoded-hp-prop* 𝒱 (*add-mset* (*Hp m* w_m ch_m) (*add-mset a x*)) (*prevs*, *nxts*, *children*, *parents*, *scores*)›
shows
‹*encoded-hp-prop* 𝒱 (*add-mset* (*Hp m* w_m (*a* # ch_m)) *x*) (*prevs*′, *nxts*(*node a*:= *if* ch_m = [] *then None else Some* (*node* (*hd* ch_m))),
children(*m* := *Some* (*node a*)), *parents*′(*node a*:= *Some m*), *scores*(*m*:=*Some* w_m))›
⟨*proof*⟩

fun *find-first-not-none* **where**
‹*find-first-not-none* (*None* # *xs*) = *find-first-not-none xs*› |
‹*find-first-not-none* (*Some a* # -) = *Some a*›|
‹*find-first-not-none* [] = *None*›

lemma *find-first-not-none-alt-def*:

‹find-first-not-none xs = map-option the (option-hd (filter ((≠) None) xs))›
⟨proof⟩

In the following we distinguish between the tree part and the tree part without parent (aka the list part). The latter corresponds to a tree where we have removed the source, but the leafs remains in the correct form. They are different for first level nexts and first level children.

definition *encoded-hp-prop-list* :: ‹'e multiset ⇒ ('e,'f) hp multiset ⇒ ('e,'f) hp list ⇒ -› **where**
‹encoded-hp-prop-list 𝒱 m xs = (λ(prevs,nxts,children, parents, scores). distinct-mset (∑# (mset-nodes '# m + mset-nodes '# (mset xs))) ∧
set-mset (∑# (mset-nodes '# m + mset-nodes '# (mset xs))) ⊆ set-mset 𝒱 ∧
(∀ m'∈#m. ∀ x ∈# mset-nodes m'. nxts x = map-option node (hp-next x m')) ∧
(∀ m∈# m. ∀ x ∈# mset-nodes m. prevs x = map-option node (hp-prev x m)) ∧
(∀ m∈# m. ∀ x ∈# mset-nodes m. children x = map-option node (hp-child x m)) ∧
(∀ m∈# m. ∀ x ∈# mset-nodes m. parents x = map-option node (hp-parent x m)) ∧
(∀ m∈# m. ∀ x ∈# mset-nodes m. scores x = map-option score (hp-node x m)) ∧
(∀ x ∈# ∑# (mset-nodes '# mset xs). nxts x = map-option node (hp-next-children x xs)) ∧
(∀ x ∈# ∑# (mset-nodes '# mset xs). prevs x = map-option node (hp-prev-children x xs)) ∧
(∀ x ∈# ∑# (mset-nodes '# mset xs). children x = map-option node (hp-child-children x xs)) ∧
(∀ x ∈# ∑# (mset-nodes '# mset xs). parents x = map-option node (hp-parent-children x xs)) ∧
(∀ x ∈# ∑# (mset-nodes '# mset xs). scores x = map-option score (hp-node-children x xs)) ∧
empty-outside (∑# (mset-nodes '# m + mset-nodes '# (mset xs))) prevs ∧
empty-outside (∑# (mset-nodes '# m + mset-nodes '# (mset xs))) parents)
›

lemma *encoded-hp-prop-list-encoded-hp-prop*[*simp*]: ‹encoded-hp-prop-list 𝒱 arr [] h = encoded-hp-prop 𝒱 arr h›
⟨proof⟩

lemma *encoded-hp-prop-list-encoded-hp-prop-single*[*simp*]: ‹encoded-hp-prop-list 𝒱 {#} [arr] h = encoded-hp-prop 𝒱 {#arr#} h›
⟨proof⟩

lemma *empty-outside-set-none-outside*[*simp*]: ‹empty-outside 𝒱 prevs ⟹ a ∉# 𝒱 ⟹ empty-outside 𝒱 (prevs(a := None))›
⟨proof⟩

lemma *encoded-hp-prop-list-remove-min*:
fixes *parents a child children*
defines ‹parents' ≡ (if children a = None then parents else parents(the (children a) := None))›
assumes ‹encoded-hp-prop-list 𝒱 (add-mset (Hp a b child) xs) [] (prevs, nxts, children, parents, scores)›
shows ‹encoded-hp-prop-list 𝒱 xs child (prevs(a:=None), nxts, children(a:=None), parents', scores)›
⟨proof⟩

lemma *hp-parent-children-skip-first*[*simp*]:
‹x ∈# sum-list (map mset-nodes ch') ⟹
distinct-mset (sum-list (map mset-nodes ch) + sum-list (map mset-nodes ch')) ⟹
hp-parent-children x (ch @ ch') = hp-parent-children x ch'›
⟨proof⟩

lemma *hp-parent-children-skip-last*[*simp*]:
‹x ∈# sum-list (map mset-nodes ch) ⟹
distinct-mset (sum-list (map mset-nodes ch) + sum-list (map mset-nodes ch')) ⟹
hp-parent-children x (ch @ ch') = hp-parent-children x ch›
⟨proof⟩

lemma *hp-parent-first-child*[*simp*]:
‹$n \neq m \Longrightarrow$ *hp-parent n* (*Hp m* w_m (*Hp n* w_n ch_n # ch_m)) = *Some* (*Hp m* w_m (*Hp n* w_n ch_n # ch_m))›
⟨*proof*⟩

lemma *encoded-hp-prop-list-link*:
fixes *m* ch_m *prevs b* hp_m *n nxts children parents*
defines ‹$prevs_0$ ≡ (*if* ch_m = [] *then prevs else prevs* (*node* (*hd* ch_m) := *Some n*))›
defines ‹*prevs'* ≡ (*if b* = [] *then* $prevs_0$ *else* $prevs_0$ (*node* (*hd b*) := *Some m*)) (*n*:= *None*)›
defines ‹*nxts'* ≡ *nxts* (*m* := *map-option node* (*option-hd b*), *n* := *map-option node* (*option-hd* ch_m))›
defines ‹*children'* ≡ *children* (*m* := *Some n*)›
defines ‹*parents'* ≡ (*if* ch_m = [] *then parents else parents*(*node* (*hd* ch_m) := *None*))(*n* := *Some m*)›
assumes ‹*encoded-hp-prop-list* $\mathcal{V}$ (*xs*) (*a* @ [*Hp m* w_m ch_m, *Hp n* w_n ch_n] @ *b*) (*prevs*, *nxts*, *children*, *parents*, *scores*)›
shows ‹*encoded-hp-prop-list* $\mathcal{V}$ *xs* (*a* @ [*Hp m* w_m (*Hp n* w_n ch_n # ch_m)] @ *b*)
(*prevs'*, *nxts'*, *children'*, *parents'*, *scores*)›
⟨*proof*⟩

lemma *encoded-hp-prop-list-link2*:
fixes *m* ch_m *prevs b* hp_m *n nxts children* ch_n *a parents*
defines ‹*prevs'* ≡ (*if* ch_n = [] *then prevs else prevs* (*node* (*hd* ch_n) := *Some m*))(*m* := *None*, *n* := *map-option node* (*option-last a*))›
defines ‹$nxts_0$ ≡ (*if a* = [] *then nxts else nxts*(*node* (*last a*) := *Some n*))›
defines ‹*nxts'* ≡ $nxts_0$ (*n* := *map-option node* (*option-hd b*), *m* := *map-option node* (*option-hd* ch_n))›
defines ‹*children'* ≡ *children* (*n* := *Some m*)›
defines ‹*parents'* ≡ (*if* ch_n = [] *then parents else parents*(*node* (*hd* ch_n) := *None*))(*m* := *Some n*)›
assumes ‹*encoded-hp-prop-list* $\mathcal{V}$ (*xs*) (*a* @ [*Hp m* w_m ch_m, *Hp n* w_n ch_n] @ *b*) (*prevs*, *nxts*, *children*, *parents*, *scores*)›
shows ‹*encoded-hp-prop-list* $\mathcal{V}$ *xs* (*a* @ [*Hp n* w_n (*Hp m* w_m ch_m # ch_n)] @ *b*)
(*prevs'*, *nxts'*, *children'*, *parents'*, *scores*)›
⟨*proof*⟩

definition *encoded-hp-prop-list-conc*
:: *'a*::*linorder multiset* × (*'a*, *'b*) *hp-fun* × *'a option* ⇒
'a multiset × (*'a*, *'b*) *hp option* ⇒ *bool*
where
‹*encoded-hp-prop-list-conc* = (λ($\mathcal{V}$, *arr*, *h*) ($\mathcal{V}'$, *x*). $\mathcal{V} = \mathcal{V}'$ ∧
(*case x of None* ⇒ *encoded-hp-prop-list* $\mathcal{V}'$ {#} ([]:: (*'a*, *'b*) *hp list*) *arr* ∧ *h* = *None*
| *Some x* ⇒ *encoded-hp-prop-list* $\mathcal{V}'$ {#*x*#} [] *arr* ∧ *set-mset* (*mset-nodes x*) ⊆ *set-mset* $\mathcal{V}$ ∧ *h* = *Some* (*node x*)))›

lemma *encoded-hp-prop-list-conc-alt-def*:
‹*encoded-hp-prop-list-conc* = (λ($\mathcal{V}$, *arr*, *h*) ($\mathcal{V}'$, *x*). $\mathcal{V} = \mathcal{V}'$ ∧
(*case x of None* ⇒ *encoded-hp-prop-list* $\mathcal{V}'$ {#} ([]:: (*'a*::*linorder*, *'b*) *hp list*) *arr* ∧ *h* = *None*
| *Some x* ⇒ *encoded-hp-prop-list* $\mathcal{V}'$ {#*x*#} [] *arr* ∧ *h* = *Some* (*node x*)))›
⟨*proof*⟩

definition *encoded-hp-prop-list2-conc*
:: *'a*::*linorder multiset* × (*'a*, *'b*) *hp-fun* × *'a option* ⇒
'a multiset × (*'a*, *'b*) *hp list* ⇒ *bool*
where
‹*encoded-hp-prop-list2-conc* = (λ($\mathcal{V}$, *arr*, *h*) ($\mathcal{V}'$, *x*). $\mathcal{V}' = \mathcal{V}$ ∧
(*encoded-hp-prop-list* $\mathcal{V}$ {#} *x arr* ∧ *set-mset* (*sum-list* (*map mset-nodes x*)) ⊆ *set-mset* $\mathcal{V}$ ∧ *h* = *None*))›

lemma *encoded-hp-prop-list2-conc-alt-def*:

‹*encoded-hp-prop-list2-conc* = (λ(𝒱, *arr*, *h*) (𝒱′, *x*). 𝒱 = 𝒱′ ∧
(*encoded-hp-prop-list* 𝒱 {#} *x arr* ∧ *h* = *None*))›
⟨*proof*⟩

lemma *encoded-hp-prop-list-update-score*:
fixes *h* :: ‹(′*a*, *nat*) *hp*› **and** *a arr* **and** *hs* :: ‹(′*a*, *nat*) *hp multiset*› **and** *x*
defines *arr*′: ‹*arr*′ ≡ *hp-update-score a* (*Some x*) *arr*›
assumes *enc*: ‹*encoded-hp-prop-list* 𝒱 (*add-mset* (*Hp a b c*) *hs*) [] *arr*›
shows ‹*encoded-hp-prop-list* 𝒱 (*add-mset* (*Hp a x c*) *hs*) []
arr′›
⟨*proof*⟩

Refinement to Imperative version

definition *hp-insert* :: ‹′*a* ⇒ ′*b*::*linorder* ⇒ ′*a multiset* × (′*a*,′*b*) *hp-fun* × ′*a option* ⇒ (′*a multiset* × (′*a*,′*b*) *hp-fun* × ′*a option*) *nres*› **where**
‹*hp-insert* = (λ(*i*::′*a*) (*w*::′*b*) (𝒱::′*a multiset*, *arr* :: (′*a*, ′*b*) *hp-fun*, *h* :: ′*a option*). *do* {
if h = *None then do* {
ASSERT (*i* ∈# 𝒱);
RETURN (𝒱, *hp-set-all i None None None None* (*Some w*) *arr*, *Some i*)
} *else do* {
ASSERT (*i* ∈# 𝒱);
ASSERT (*hp-read-prev i arr* = *None*);
ASSERT (*hp-read-parent i arr* = *None*);
let (*j*::′*a*) = ((*the h*) :: ′*a*);
ASSERT (*j* ∈# 𝒱 ∧ *j* ≠ *i*);
ASSERT (*hp-read-score j* (*arr* :: (′*a*, ′*b*) *hp-fun*) ≠ *None*);
ASSERT (*hp-read-prev j arr* = *None* ∧ *hp-read-nxt j arr* = *None* ∧ *hp-read-parent j arr* = *None*);
let y = (*the* (*hp-read-score j arr*)::′*b*);
if y < *w*
then do {
let arr = *hp-set-all i None None* (*Some j*) *None* (*Some* (*w*::′*b*)) (*arr*::(′*a*, ′*b*) *hp-fun*);
let arr = *hp-update-parents j* (*Some i*) *arr*;
let nxt = *hp-read-nxt j arr*;
RETURN (𝒱, *arr* :: (′*a*, ′*b*) *hp-fun*, *Some i*)
}
else do {
let child = *hp-read-child j arr*;
ASSERT (*child* ≠ *None* ⟶ *the child* ∈# 𝒱);
let arr = *hp-set-all j None None* (*Some i*) *None* (*Some y*) *arr*;
let arr = *hp-set-all i None child None* (*Some j*) (*Some* (*w*::′*b*)) *arr*;
let arr = (*if child* = *None then arr else hp-update-prev* (*the child*) (*Some i*) *arr*);
let arr = (*if child* = *None then arr else hp-update-parents* (*the child*) *None arr*);
RETURN (𝒱, *arr* :: (′*a*, ′*b*) *hp-fun*, *h*)
}
}
})›

lemma *hp-insert-spec*:
assumes ‹*encoded-hp-prop-list-conc arr h*› **and**
‹*snd h* ≠ *None* ⟹ *i* ∉# *mset-nodes* (*the* (*snd h*))› **and**
‹*i* ∈# *fst arr*›
shows ‹*hp-insert i w arr* ≤ ⇓ {(*arr*, *h*). *encoded-hp-prop-list-conc arr h*} (*ACIDS.mop-hm-insert i w h*)›
⟨*proof*⟩

definition *hp-link* :: ‹'a ⇒ 'a ⇒ 'a multiset × ('a,'b::order) hp-fun × 'a option ⇒ (('a multiset × ('a,'b) hp-fun × 'a option) × 'a) nres› **where**
‹*hp-link* = (λ(i::'a) j (𝒱::'a multiset, arr :: ('a, 'b) hp-fun, h :: 'a option). do {
ASSERT (i ≠ j);
ASSERT (i ∈# 𝒱);
ASSERT (j ∈# 𝒱);
ASSERT (hp-read-score i arr ≠ None);
ASSERT (hp-read-score j arr ≠ None);
let x = (the (hp-read-score i arr)::'b);
let y = (the (hp-read-score j arr)::'b);
let prev = hp-read-prev i arr;
let nxt = hp-read-nxt j arr;
ASSERT (nxt ≠ Some i ∧ nxt ≠ Some j);
ASSERT (prev ≠ Some i ∧ prev ≠ Some j);
let (parent,ch,w_p, w_{ch}) = (if y < x then (i, j, x, y) else (j, i, y, x));
let child = hp-read-child parent arr;
ASSERT (child ≠ Some i ∧ child ≠ Some j);
let $child_{ch}$ = hp-read-child ch arr;
ASSERT ($child_{ch}$ ≠ Some i ∧ $child_{ch}$ ≠ Some j ∧ ($child_{ch}$ ≠ None ⟶ $child_{ch}$ ≠ child));
ASSERT (distinct ([i, j] @ (if $child_{ch}$ ≠ None then [the $child_{ch}$] else [])
@ (if child ≠ None then [the child] else [])
@ (if prev ≠ None then [the prev] else [])
@ (if nxt ≠ None then [the nxt] else []))
);
ASSERT (ch ∈# 𝒱);
ASSERT (parent ∈# 𝒱);
ASSERT (child ≠ None ⟶ the child ∈# 𝒱);
ASSERT (nxt ≠ None ⟶ the nxt ∈# 𝒱);
ASSERT (prev ≠ None ⟶ the prev ∈# 𝒱);
let arr = hp-set-all parent prev nxt (Some ch) None (Some (w_p::'b)) (arr::('a, 'b) hp-fun);
let arr = hp-set-all ch None child $child_{ch}$ (Some parent) (Some (w_{ch}::'b)) (arr::('a, 'b) hp-fun);
let arr = (if child = None then arr else hp-update-prev (the child) (Some ch) arr);
let arr = (if nxt = None then arr else hp-update-prev (the nxt) (Some parent) arr);
let arr = (if prev = None then arr else hp-update-nxt (the prev) (Some parent) arr);
let arr = (if child = None then arr else hp-update-parents (the child) None arr);
RETURN ((𝒱, arr :: ('a, 'b) hp-fun, h), parent)
})›

lemma *fun-upd-twist2*: a ≠ c ⟹ a ≠ e ⟹ c ≠ e ⟹ m(a := b, c := d, e := f) = (m(e := f, c := d))(a := b)
⟨*proof*⟩

lemma *hp-link*:
assumes *enc*: ‹encoded-hp-prop-list2-conc arr (𝒱', xs @ x # y # ys)› **and**
‹i = node x› **and**
‹j = node y›
shows ‹hp-link i j arr ≤ SPEC (λ(arr, n). encoded-hp-prop-list2-conc arr (𝒱', xs @ ACIDS.link x y # ys) ∧
n = node (ACIDS.link x y))›
⟨*proof*⟩

In an imperative setting use the two pass approaches is better than the alternative.
The e of the loop is a dummy counter.

definition *vsids-pass*$_1$ **where**
‹*vsids-pass*$_1$ = (λ(𝒱::'*a multiset*, *arr* :: ('*a*, '*b*::*order*) *hp-fun*, *h* :: '*a option*) (*j*::'*a*). *do* {
((𝒱, *arr*, *h*), *j*, -, *n*) ← *WHILE*$_T$(λ((𝒱, *arr*, *h*), *j*, *e*, *n*). *j* ≠ *None*)
(λ((𝒱, *arr*, *h*), *j*, *e*::*nat*, *n*). *do* {
if j = *None then RETURN* ((𝒱, *arr*, *h*), *None*, *e*, *n*)
else do {
let j = *the j*;
ASSERT (*j* ∈# 𝒱);
let nxt = *hp-read-nxt j arr*;
if nxt = *None then RETURN* ((𝒱, *arr*, *h*), *nxt*, *e*+*1*, *j*)
else do {
ASSERT (*nxt* ≠ *None*);
ASSERT (*the nxt* ∈# 𝒱);
let nnxt = *hp-read-nxt* (*the nxt*) *arr*;
((𝒱, *arr*, *h*), *n*) ← *hp-link j* (*the nxt*) (𝒱, *arr*, *h*);
RETURN ((𝒱, *arr*, *h*), *nnxt*, *e*+*2*, *n*)
}}
})
((𝒱, *arr*, *h*), *Some j*, *0*::*nat*, *j*);
RETURN ((𝒱, *arr*, *h*), *n*)
})›

lemma *vsids-pass*$_1$:
fixes *arr* :: ‹'*a*::*linorder multiset* × ('*a*, *nat*) *hp-fun* × '*a option*›
assumes ‹*encoded-hp-prop-list2-conc arr* (𝒱', *xs*)› **and** ‹*xs* ≠ []› **and** ‹*j* = *node* (*hd xs*)›
shows ‹*vsids-pass*$_1$ *arr j* ≤ *SPEC*(λ(*arr*, *j*). *encoded-hp-prop-list2-conc arr* (𝒱', *ACIDS.pass*$_1$ *xs*) ∧ *j* = *node* (*last* (*ACIDS.pass*$_1$ *xs*)))›
⟨*proof*⟩

definition *vsids-pass*$_2$ **where**
‹*vsids-pass*$_2$ = (λ(𝒱::'*a multiset*, *arr* :: ('*a*, '*b*::*order*) *hp-fun*, *h* :: '*a option*) (*j*::'*a*). *do* {
ASSERT (*j* ∈# 𝒱);
let nxt = *hp-read-prev j arr*;
((𝒱, *arr*, *h*), *j*, *leader*, -) ← *WHILE*$_T$(λ((𝒱, *arr*, *h*), *j*, *leader*, *e*). *j* ≠ *None*)
(λ((𝒱, *arr*, *h*), *j*, *leader*, *e*::*nat*). *do* {
if j = *None then RETURN* ((𝒱, *arr*, *h*), *None*, *leader*, *e*)
else do {
let j = *the j*;
ASSERT (*j* ∈# 𝒱);
let nnxt = *hp-read-prev j arr*;
((𝒱, *arr*, *h*), *n*) ← *hp-link j leader* (𝒱, *arr*, *h*);
RETURN ((𝒱, *arr*, *h*), *nnxt*, *n*, *e*+*1*)
}
})
((𝒱, *arr*, *h*), *nxt*, *j*, *1*::*nat*);
RETURN (𝒱, *arr*, *Some leader*)
})›

lemma *vsids-pass*$_2$:
fixes *arr* :: ‹'*a*::*linorder multiset* × ('*a*, *nat*) *hp-fun* × '*a option*›
assumes ‹*encoded-hp-prop-list2-conc arr* (𝒱', *xs*)› **and** ‹*xs* ≠ []› **and** ‹*j* = *node* (*last xs*)›
shows ‹*vsids-pass*$_2$ *arr j* ≤ *SPEC*(λ(*arr*). *encoded-hp-prop-list-conc arr* (𝒱', *ACIDS.pass*$_2$ *xs*))›
⟨*proof*⟩

definition *merge-pairs* **where**
‹*merge-pairs arr j* = *do* {
  (*arr*, *j*) ← *vsids-pass*$_1$ *arr j*;
  vsids-pass$_2$ *arr j*
}›

lemma *vsids-merge-pairs*:
fixes *arr* :: ‹′*a*::*linorder multiset* × (′*a*, *nat*) *hp-fun* × ′*a option*›
assumes ‹*encoded-hp-prop-list2-conc arr* (𝒱′, *xs*)› **and** ‹*xs* ≠ []› **and** ‹*j* = *node* (*hd xs*)›
shows ‹*merge-pairs arr j* ≤ *SPEC*(λ(*arr*). *encoded-hp-prop-list-conc arr* (𝒱′, *ACIDS*.*merge-pairs xs*))›
⟨*proof*⟩

definition *hp-update-child* **where**
‹*hp-update-child i nxt* = (λ(*prevs*, *nxts*, *childs*, *scores*). (*prevs*, *nxts*, *childs*(*i*:=*nxt*), *scores*))›

definition *vsids-pop-min* :: ‹-› **where**
‹*vsids-pop-min* = (λ(𝒱::′*a multiset*, *arr* :: (′*a*, ′*b*::*order*) *hp-fun*, *h* :: ′*a option*). *do* {
if h = *None then RETURN* (*None*, (𝒱, *arr*, *h*))
else do {
    ASSERT (*the h* ∈# 𝒱);
    let j = *hp-read-child* (*the h*) *arr*;
    if j = *None then RETURN* (*h*, (𝒱, *arr*, *None*))
    else do {
      ASSERT (*the j* ∈# 𝒱);
      let arr = *hp-update-prev* (*the h*) *None arr*;
      let arr = *hp-update-child* (*the h*) *None arr*;
      let arr = *hp-update-parents* (*the j*) *None arr*;
      arr ← *merge-pairs* (𝒱, *arr*, *None*) (*the j*);
      RETURN (*h*, *arr*)
    }
  }
})›

lemma *node-remove-key-itself-iff*[*simp*]: ‹*remove-key* (*y*) *z* ≠ *None* ⟹ *node z* = *node* (*the* (*remove-key* (*y*) *z*)) ⟷ *y* ≠ *node z*›
⟨*proof*⟩

lemma *vsids-pop-min*:
fixes *arr* :: ‹′*a*::*linorder multiset* × (′*a*, *nat*) *hp-fun* × ′*a option*›
assumes ‹*encoded-hp-prop-list-conc arr* (𝒱, *h*)›
shows ‹*vsids-pop-min arr* ≤ *SPEC*(λ(*j*, *arr*). *j* = (*if h* = *None then None else Some* (*get-min2 h*)) ∧ *encoded-hp-prop-list-conc arr* (𝒱, *ACIDS*.*del-min h*))›
⟨*proof*⟩

Unconditionnal version of the previous function

definition *vsids-pop-min2* :: ‹-› **where**
‹*vsids-pop-min2* = (λ(𝒱::′*a multiset*, *arr* :: (′*a*, ′*b*::*order*) *hp-fun*, *h* :: ′*a option*). *do* {
  ASSERT (*h*≠*None*);
  ASSERT (*the h* ∈# 𝒱);
  let j = *hp-read-child* (*the h*) *arr*;
  if j = *None then RETURN* (*the h*, (𝒱, *arr*, *None*))
  else do {
    ASSERT (*the j* ∈# 𝒱);
    let arr = *hp-update-prev* (*the h*) *None arr*;

let arr = *hp-update-child* (*the h*) *None arr*;
let arr = *hp-update-parents* (*the j*) *None arr*;
arr ← *merge-pairs* ($\mathcal{V}$, *arr*, *None*) (*the j*);
RETURN (*the h*, *arr*)
}
}
)›

lemma *vsids-pop-min2*:
fixes *arr* :: ‹′*a*::*linorder multiset* × (′*a*, *nat*) *hp-fun* × ′*a option*›
assumes ‹*encoded-hp-prop-list-conc arr* ($\mathcal{V}$, *h*)› **and** ‹*h* ≠ *None*›
shows ‹*vsids-pop-min2 arr* ≤ *SPEC*(λ(*j*, *arr*). *j* = (*get-min2 h*) ∧ *encoded-hp-prop-list-conc arr* ($\mathcal{V}$, *ACIDS*.*del-min h*))›
⟨*proof*⟩

lemma *in-remove-key-in-find-keyD*:
‹*m*′ ∈# (*if remove-key a h* = *None then* {#} *else* {#*the* (*remove-key a h*)#}) +
(*if find-key a h* = *None then* {#} *else* {#*the* (*find-key a h*)#}) ⟹
distinct-mset (*mset-nodes h*) ⟹
x′ ∈# *mset-nodes m*′ ⟹ *x*′ ∈# *mset-nodes h*›
⟨*proof*⟩

lemma *map-option-node-map-option-node-iff*:
‹(*x* ≠ *None* ⟹ *distinct-mset* (*mset-nodes* (*the x*))) ⟹ (*x* ≠ *None* ⟹ *y* ≠ *node* (*the x*)) ⟹
map-option node x = *map-option node* (*map-option* (λ*x*. *the* (*remove-key y x*)) *x*)›
⟨*proof*⟩

lemma *distinct-mset-hp-parent*: ‹*distinct-mset* (*mset-nodes h*) ⟹ *hp-parent a h* = *Some ya* ⟹ *distinct-mset* (*mset-nodes ya*)›
⟨*proof*⟩

lemma *in-find-key-children-same-hp-parent*:
‹*hp-parent k* (*Hp x n c*) = *None* ⟹
x′ ∈# *mset-nodes m*′ ⟹
x ∉# *sum-list* (*map mset-nodes c*) ⟹
distinct-mset (*sum-list* (*map mset-nodes c*)) ⟹
find-key-children k c = *Some m*′ ⟹ *hp-parent x*′ (*Hp x n c*) = *hp-parent x*′ *m*′›
⟨*proof*⟩

lemma *in-find-key-same-hp-parent*:
‹*x*′ ∈# *mset-nodes m*′ ⟹
distinct-mset (*mset-nodes h*) ⟹
find-key a h = *Some m*′ ⟹
hp-parent a h = *None* ⟹
∃ *y*. *hp-prev a h* = *Some y* ⟹
hp-parent x′ *h* = *hp-parent x*′ *m*′›
⟨*proof*⟩

lemma *in-find-key-children-same-hp-parent2*:
‹*x*′ ≠ *k* ⟹
x′ ∈# *mset-nodes m*′ ⟹
x ∉# *sum-list* (*map mset-nodes c*) ⟹
distinct-mset (*sum-list* (*map mset-nodes c*)) ⟹
find-key-children k c = *Some m*′ ⟹ *hp-parent x*′ (*Hp x n c*) = *hp-parent x*′ *m*′›
⟨*proof*⟩

lemma *in-find-key-same-hp-parent2*:
‹$x' \in\#$ *mset-nodes* $m' \Longrightarrow$
distinct-mset (*mset-nodes h*) $\Longrightarrow$
find-key a h = *Some* $m' \Longrightarrow$
$x' \neq a \Longrightarrow$
hp-parent x' *h* = *hp-parent* x' m'›
⟨*proof*⟩

lemma *encoded-hp-prop-list-remove-find*:
fixes *h* :: ‹('*a*, *nat*) *hp*› **and** *a arr* **and** *hs* :: ‹('*a*, *nat*) *hp multiset*›
defines ‹$arr_1 \equiv$ (*if hp-parent a h* = *None then arr else hp-update-child* (*node* (*the* (*hp-parent a h*))) (*map-option node* (*hp-next a h*)) *arr*)›
defines ‹$arr_2 \equiv$ (*if hp-prev a h* = *None then* arr_1 *else hp-update-nxt* (*node* (*the* (*hp-prev a h*))) (*map-option node* (*hp-next a h*)) arr_1)›
defines ‹$arr_3 \equiv$ (*if hp-next a h* = *None then* arr_2 *else hp-update-prev* (*node* (*the* (*hp-next a h*))) (*map-option node* (*hp-prev a h*)) arr_2)›
defines ‹$arr_4 \equiv$ (*if hp-next a h* = *None then* arr_3 *else hp-update-parents* (*node* (*the* (*hp-next a h*))) (*map-option node* (*hp-parent a h*)) arr_3)›
defines ‹$arr' \equiv$ *hp-update-parents a None* (*hp-update-prev a None* (*hp-update-nxt a None* arr_4))›
assumes *enc*: ‹*encoded-hp-prop-list* $\mathcal{V}$ (*add-mset h* {#}) [] *arr*›
shows ‹*encoded-hp-prop-list* $\mathcal{V}$ ((*if remove-key a h* = *None then* {#} *else* {#*the* (*remove-key a h*)#}) +
(*if find-key a h* = *None then* {#} *else* {#*the* (*find-key a h*)#})) []
arr'›
⟨*proof*⟩

In the kissat implementation prev and parent are merged.

lemma *in-node-iff-prev-parent-or-root*:
assumes ‹*distinct-mset* (*mset-nodes h*)›
shows ‹$i \in\#$ *mset-nodes h* $\longleftrightarrow$ *hp-prev i h* $\neq$ *None* $\vee$ *hp-parent i h* $\neq$ *None* $\vee$ *i* = *node h*›
⟨*proof*⟩

lemma *encoded-hp-prop-list-in-node-iff-prev-parent-or-root*:
assumes ‹*encoded-hp-prop-list-conc arr h*› **and** ‹*snd h* $\neq$ *None*›
shows ‹$i \in\#$ *mset-nodes* (*the* (*snd h*)) $\longleftrightarrow$ *hp-read-prev i* (*fst* (*snd arr*)) $\neq$ *None* $\vee$ *hp-read-parent i* (*fst* (*snd arr*)) $\neq$ *None* $\vee$ *Some i* = *snd* (*snd arr*)›
⟨*proof*⟩

fun *update-source-node* **where**
‹*update-source-node i* ($\mathcal{V}$,*arr*,-) = ($\mathcal{V}$, *arr*, *i*)›
fun *source-node* :: ‹(*nat multiset* × (*nat*,'*c*) *hp-fun* × *nat option*) ⇒ -› **where**
‹*source-node* ($\mathcal{V}$,*arr*,*h*) = *h*›
fun *hp-read-nxt'* :: ‹-› **where**
‹*hp-read-nxt' i* ($\mathcal{V}$, *arr*, *h*) = *hp-read-nxt i arr*›
fun *hp-read-parent'* :: ‹-› **where**
‹*hp-read-parent' i* ($\mathcal{V}$, *arr*, *h*) = *hp-read-parent i arr*›

fun *hp-read-score'* :: ‹-› **where**
‹*hp-read-score' i* ($\mathcal{V}$, *arr*, *h*) = (*hp-read-score i arr*)›
fun *hp-read-child'* :: ‹-› **where**
‹*hp-read-child' i* ($\mathcal{V}$, *arr*, *h*) = *hp-read-child i arr*›

fun *hp-read-prev'* :: ‹-› **where**
‹*hp-read-prev' i* ($\mathcal{V}$, *arr*, *h*) = *hp-read-prev i arr*›

fun *hp-update-child′* **where**
‹*hp-update-child′ i p*($\mathcal{V}$, *u*, *h*) = ($\mathcal{V}$, *hp-update-child i p u*, *h*)›

fun *hp-update-parents′* **where**
‹*hp-update-parents′ i p*($\mathcal{V}$, *u*, *h*) = ($\mathcal{V}$, *hp-update-parents i p u*, *h*)›

fun *hp-update-prev′* **where**
‹*hp-update-prev′ i p* ($\mathcal{V}$, *u*, *h*) = ($\mathcal{V}$, *hp-update-prev i p u*, *h*)›

fun *hp-update-nxt′* **where**
‹*hp-update-nxt′ i p*($\mathcal{V}$, *u*, *h*) = ($\mathcal{V}$, *hp-update-nxt i p u*, *h*)›

fun *hp-update-score′* **where**
‹*hp-update-score′ i p*($\mathcal{V}$, *u*, *h*) = ($\mathcal{V}$, *hp-update-score i p u*, *h*)›

definition *maybe-hp-update-prev′* **where**
‹*maybe-hp-update-prev′ child ch arr* =
(*if child* = *None then arr else hp-update-prev′* (*the child*) *ch arr*)›

definition *maybe-hp-update-nxt′* **where**
‹*maybe-hp-update-nxt′ child ch arr* =
(*if child* = *None then arr else hp-update-nxt′* (*the child*) *ch arr*)›

definition *maybe-hp-update-parents′* **where**
‹*maybe-hp-update-parents′ child ch arr* =
(*if child* = *None then arr else hp-update-parents′* (*the child*) *ch arr*)›

definition *maybe-hp-update-child′* **where**
‹*maybe-hp-update-child′ child ch arr* =
(*if child* = *None then arr else hp-update-child′* (*the child*) *ch arr*)›

definition *unroot-hp-tree* **where**

```
‹unroot-hp-tree arr h = do {
  ASSERT (h ∈# fst arr);
  let a = source-node arr;
  ASSERT (a ≠ None ⟶ the a ∈# fst arr);
  let nnext = hp-read-nxt′ h arr;
  let parent = hp-read-parent′ h arr;
  let prev = hp-read-prev′ h arr;
  if prev = None ∧ parent = None ∧ Some h ≠ a then RETURN (update-source-node None arr)
  else if Some h = a then RETURN (update-source-node None arr)
  else do {
    ASSERT (a ≠ None);
    ASSERT (nnext ≠ None ⟶ the nnext ∈# fst arr);
    ASSERT (parent ≠ None ⟶ the parent ∈# fst arr);
    ASSERT (prev ≠ None ⟶ the prev ∈# fst arr);
    let a′ = the a;
    let arr = maybe-hp-update-child′ parent nnext arr;
    let arr = maybe-hp-update-nxt′ prev nnext arr;
    let arr = maybe-hp-update-prev′ nnext prev arr;
    let arr = maybe-hp-update-parents′ nnext parent arr;

    let arr = hp-update-nxt′ h None arr;
    let arr = hp-update-prev′ h None arr;
```

```
      let arr = hp-update-parents' h None arr;

      let arr = hp-update-nxt' h (Some a') arr;
      let arr = hp-update-prev' a' (Some h) arr;
      RETURN (update-source-node None arr)
    }
}›
```

lemma *unroot-hp-tree-alt-def*:

```
  ‹unroot-hp-tree arr h = do {
    ASSERT (h ∈# fst arr);
    let a = source-node arr;
    ASSERT (a ≠ None ⟶ the a ∈# fst arr);
    let nnext = hp-read-nxt' h arr;
    let parent = hp-read-parent' h arr;
    let prev = hp-read-prev' h arr;
    if prev = None ∧ parent = None ∧ Some h ≠ a then RETURN (update-source-node None arr)
    else if Some h = a then RETURN (update-source-node None arr)
    else do {
      ASSERT (a ≠ None);
      ASSERT (nnext ≠ None ⟶ the nnext ∈# fst arr);
      ASSERT (parent ≠ None ⟶ the parent ∈# fst arr);
      ASSERT (prev ≠ None ⟶ the prev ∈# fst arr);
      let a' = the a;
       arr ← do {
         let arr = maybe-hp-update-child' parent nnext arr;
         let arr = maybe-hp-update-nxt' prev nnext arr;
         let arr = maybe-hp-update-prev' nnext prev arr;
         let arr = maybe-hp-update-parents' nnext parent arr;

         let arr = hp-update-nxt' h None arr;
         let arr = hp-update-prev' h None arr;
         let arr = hp-update-parents' h None arr;

        RETURN (update-source-node None arr)
      };

      let arr = hp-update-nxt' h (Some a') arr;
      let arr = hp-update-prev' a' (Some h) arr;
      RETURN (arr)
      }
}›
  ⟨proof⟩
```

lemma *hp-update-fst-snd*:

```
  ‹hp-update-prev i j (fst (snd arr)) = fst (snd (hp-update-prev' i j arr))›
  ‹hp-update-nxt i j (fst (snd arr)) = fst (snd (hp-update-nxt' i j arr))›
  ‹hp-update-parents i j (fst (snd arr)) = fst (snd (hp-update-parents' i j arr))›
  ‹hp-update-child i j (fst (snd arr)) = fst (snd (hp-update-child' i j arr))›
  ⟨proof⟩
```

lemma *find-remove-mset-nodes-full2*:

```
  ‹distinct-mset (mset-nodes h) ⟹ sum-mset (mset-nodes `# ((if remove-key a h = None then {#} else
{#the (remove-key a h)#}) +
        (if find-key a h = None then {#} else {#the (find-key a h)#}))) = mset-nodes h›
  ⟨proof⟩
```

definition *encoded-hp-prop-mset2-conc*
:: *'a::linorder multiset* × (*'a*, *'b*) *hp-fun* × *'a option* ⇒
'a::linorder multiset × (*'a*, *'b*) *hp multiset* ⇒ *bool*
where
‹*encoded-hp-prop-mset2-conc* = (λ(𝒱, *arr*, *h*) (𝒱′, *x*). 𝒱 = 𝒱′ ∧
(*encoded-hp-prop-list* 𝒱 *x* [] *arr*))›

lemma *fst-update*[*simp*]:
‹ *fst* (*hp-update-prev′ a b x*) = *fst x*›
‹*fst* (*hp-update-nxt′ a b x*) = *fst x*›
‹*fst* (*update-source-node y x*) = *fst x*›
⟨*proof*⟩

lemma *encoded-hp-prop-mset2-conc-combine-list2-conc*:
‹*encoded-hp-prop-mset2-conc arr* (𝒱, {#*a*,*b*#}) ⟹
encoded-hp-prop-list2-conc (*hp-update-prev′* (*node b*) (*Some* (*node a*)) (*hp-update-nxt′* (*node a*) (*Some* (*node b*)) (*update-source-node None arr*))) (𝒱, [*a*,*b*])›
⟨*proof*⟩

lemma *update-source-node-fst-simps*[*simp*]:
‹*fst* (*snd* (*update-source-node None arr*)) = *fst* (*snd arr*)›
‹*fst* (*update-source-node None arr*) = *fst arr*›
‹*snd* (*snd* (*update-source-node None arr*)) = *None*›
⟨*proof*⟩

lemma *maybe-hp-update-fst-snd*: ‹*fst* (*snd* (*maybe-hp-update-child′* (*map-option node b*) *x arr*)) =
(*if b* = *None then fst* (*snd arr*) *else fst* (*snd* (*hp-update-child′* (*node* (*the b*)) *x arr*)))›
‹*fst* (*snd* (*maybe-hp-update-prev′* (*map-option node b*) *x arr*)) =
(*if b* = *None then fst* (*snd arr*) *else fst* (*snd* (*hp-update-prev′* (*node* (*the b*)) *x arr*)))›
‹*fst* (*snd* (*maybe-hp-update-nxt′* (*map-option node b*) *x arr*)) =
(*if b* = *None then fst* (*snd arr*) *else fst* (*snd* (*hp-update-nxt′* (*node* (*the b*)) *x arr*)))›
‹*fst* (*snd* (*maybe-hp-update-parents′* (*map-option node b*) *x arr*)) =
(*if b* = *None then fst* (*snd arr*) *else fst* (*snd* (*hp-update-parents′* (*node* (*the b*)) *x arr*)))› **and**
maybe-hp-update-fst-snd2:
‹((*maybe-hp-update-child′* (*map-option node b*) *x arr′*)) =
(*if b* = *None then* (*arr′*) *else* ((*hp-update-child′* (*node* (*the b*)) *x arr′*)))›
‹((*maybe-hp-update-prev′* (*map-option node b*) *x arr′*)) =
(*if b* = *None then* (*arr′*) *else* ((*hp-update-prev′* (*node* (*the b*)) *x arr′*)))›
‹((*maybe-hp-update-nxt′* (*map-option node b*) *x arr′*)) =
(*if b* = *None then* (*arr′*) *else* ((*hp-update-nxt′* (*node* (*the b*)) *x arr′*)))›
‹((*maybe-hp-update-parents′* (*map-option node b*) *x arr′*)) =
(*if b* = *None then* (*arr′*) *else* ((*hp-update-parents′* (*node* (*the b*)) *x arr′*)))›
for *x b arr*
⟨*proof*⟩

lemma *fst-hp-update-simp*[*simp*]:
‹*fst* (*hp-update-prev′ i x arr*) = *fst arr*›
‹*fst* (*hp-update-nxt′ i x arr*) = *fst arr*›
‹*fst* (*hp-update-child′ i x arr*) = *fst arr*›
‹*fst* (*hp-update-parents′ i x arr*) = *fst arr*›
⟨*proof*⟩

lemma *fst-maybe-hp-update-simp*[*simp*]:
‹*fst* (*maybe-hp-update-prev′ i y arr*) = *fst arr*›
‹*fst* (*maybe-hp-update-nxt′ i y arr*) = *fst arr*›

‹fst (maybe-hp-update-child′ i y arr) = fst arr›
‹fst (maybe-hp-update-parents′ i y arr) = fst arr›
⟨proof⟩

lemma *encoded-hp-prop-list-remove-find2*:
fixes *h* :: ‹(′a::linorder, nat) hp› **and** *a arr* **and** *hs* :: ‹(′a, nat) hp multiset›
defines ‹arr_1 ≡ (if hp-parent a h = None then arr else hp-update-child′ (node (the (hp-parent a h))) (map-option node (hp-next a h)) arr)›
defines ‹arr_2 ≡ (if hp-prev a h = None then arr_1 else hp-update-nxt′ (node (the (hp-prev a h))) (map-option node (hp-next a h)) arr_1)›
defines ‹arr_3 ≡ (if hp-next a h = None then arr_2 else hp-update-prev′ (node (the (hp-next a h))) (map-option node (hp-prev a h)) arr_2)›
defines ‹arr_4 ≡ (if hp-next a h = None then arr_3 else hp-update-parents′ (node (the (hp-next a h))) (map-option node (hp-parent a h)) arr_3)›
defines ‹arr′ ≡ hp-update-parents′ a None (hp-update-prev′ a None (hp-update-nxt′ a None arr_4))›
assumes *enc*: ‹encoded-hp-prop-mset2-conc arr (𝒱, add-mset h {#})›
shows ‹encoded-hp-prop-mset2-conc arr′ (𝒱, (if remove-key a h = None then {#} else {#the (remove-key a h)#}) +
(if find-key a h = None then {#} else {#the (find-key a h)#}))›
⟨proof⟩

lemma *hp-read-fst-snd-simps*[*simp*]:
‹hp-read-nxt j (fst (snd arr)) = hp-read-nxt′ j arr›
‹hp-read-prev j (fst (snd arr)) = hp-read-prev′ j arr›
‹hp-read-child j (fst (snd arr)) = hp-read-child′ j arr›
‹hp-read-parent j (fst (snd arr)) = hp-read-parent′ j arr›
‹hp-read-score j (fst (snd arr)) = hp-read-score′ j arr›
⟨proof⟩

lemma *unroot-hp-tree*:
fixes *h* :: ‹(nat, nat)hp option›
assumes *enc*: ‹encoded-hp-prop-list-conc arr (𝒱, h)› ‹a ∈# fst arr› ‹h ≠ None›
shows ‹unroot-hp-tree arr a ≤ SPEC (λarr′. fst arr′ = fst arr ∧ encoded-hp-prop-list2-conc arr′
(𝒱, (if find-key a (the h) = None then [] else [the (find-key a (the h))]) @
(if remove-key a (the h) = None then [] else [the (remove-key a (the h))])))›
⟨proof⟩

definition *rescale-and-reroot* **where**
```
‹rescale-and-reroot h w′ arr = do {
  ASSERT (h ∈# fst arr);
  let nnext = hp-read-nxt′ h arr;
  let parent = hp-read-parent′ h arr;
  let prev = hp-read-prev′ h arr;
  if source-node arr = None then RETURN (hp-update-score′ h (Some w′) arr)
  else if prev = None ∧ parent = None ∧ Some h ≠ source-node arr then RETURN (hp-update-score′ h (Some w′) arr)
  else if Some h = source-node arr then RETURN (hp-update-score′ h (Some w′) arr)
  else do {
    arr ← unroot-hp-tree arr h;
    ASSERT (h ∈# fst arr);
    let arr = (hp-update-score′ h (Some w′) arr);
    merge-pairs arr h
  }
}›
```

lemma *fst-update2*[*simp*]:
‹*fst* (*hp-update-score′ a b h*) = *fst h*›
⟨*proof*⟩

lemma *encoded-hp-prop-list2-conc-update-score*:
‹*encoded-hp-prop-list2-conc h* ($\mathcal{V}$, [*x*,*y*]) ⟹ *node x* = *a* ⟹ *encoded-hp-prop-list2-conc* (*hp-update-score′ a* (*Some w′*) *h*) ($\mathcal{V}$, [*Hp* (*node x*) *w′* (*hps x*), *y*])›
⟨*proof*⟩

lemma *encoded-hp-prop-list-conc-update-score*: ‹*encoded-hp-prop-list-conc arr* ($\mathcal{V}$, *Some* (*Hp a x2 x3*)) ⟹
encoded-hp-prop-list-conc (*hp-update-score′ a* (*Some w′*) *arr*) ($\mathcal{V}$, *Some* (*Hp a w′ x3*))›
⟨*proof*⟩

lemma *encoded-hp-prop-list-conc-update-outside*:
‹(*snd h* ≠ *None* ⟹ *a* ∉# *mset-nodes* (*the* (*snd h*))) ⟹ *encoded-hp-prop-list-conc arr h* ⟹
encoded-hp-prop-list-conc (*hp-update-score′ a w′ arr*) *h*›
⟨*proof*⟩

definition *ACIDS-decrease-key′* **where**
‹*ACIDS-decrease-key′* = (λ*a w* ($\mathcal{V}$, *h*). ($\mathcal{V}$, *ACIDS.decrease-key a w* (*the h*)))›

lemma *rescale-and-reroot*:
fixes *h* :: ‹*nat multiset* × (*nat*, *nat*)*hp option*›
assumes *enc*: ‹*encoded-hp-prop-list-conc arr h*›
shows ‹*rescale-and-reroot a w′ arr* ≤ ⇓ {(*arr*, *h*). *encoded-hp-prop-list-conc arr h*} (*ACIDS.mop-hm-decrease-key a w′ h*)›
⟨*proof*⟩

definition *acids-encoded-hmrel* **where**
‹*acids-encoded-hmrel* = {(*arr*, *h*). *encoded-hp-prop-list-conc arr h*} *O ACIDS.hmrel*›

lemma *hp-insert-spec-mop-prio-insert*:
assumes ‹(*arr*, *h*) ∈ *acids-encoded-hmrel*›
shows ‹*hp-insert i w arr* ≤ ⇓*acids-encoded-hmrel* (*ACIDS.mop-prio-insert i w h*)›
⟨*proof*⟩

lemma *hp-insert-spec-mop-prio-insert2*:
‹(*uncurry2 hp-insert*, *uncurry2 ACIDS.mop-prio-insert*) ∈
nat-rel $\times_f$ *nat-rel* $\times_f$ *acids-encoded-hmrel* $\to_f$ ⟨*acids-encoded-hmrel*⟩*nres-rel*›
⟨*proof*⟩

lemma *rescale-and-reroot-mop-prio-change-weight*:
assumes ‹(*arr*, *h*) ∈ *acids-encoded-hmrel*›
shows ‹*rescale-and-reroot a w arr* ≤ ⇓*acids-encoded-hmrel* (*ACIDS.mop-prio-change-weight a w h*)›
⟨*proof*⟩

lemma *rescale-and-reroot-mop-prio-change-weight2*:
‹(*uncurry2 rescale-and-reroot*, *uncurry2 ACIDS.mop-prio-change-weight*) ∈
nat-rel $\times_f$ *nat-rel* $\times_f$ *acids-encoded-hmrel* $\to_f$ ⟨*acids-encoded-hmrel*⟩*nres-rel*›
⟨*proof*⟩

context *hmstruct-with-prio*
begin

definition *mop-hm-is-in* :: ‹-› **where**
‹*mop-hm-is-in w* = (λ(𝒜, *xs*). *do* {
ASSERT (*w* ∈# 𝒜);
RETURN (*xs* ≠ *None* ∧ *w* ∈# *mset-nodes* (*the xs*))
})›

lemma *mop-hm-is-in-mop-prio-is-in*:
assumes ‹(*xs*, *ys*) ∈ *hmrel*›
shows ‹*mop-hm-is-in w xs* ≤ ⇓*bool-rel* (*mop-prio-is-in w ys*)›
⟨*proof*⟩

lemma *del-min-None-iff*: ‹*del-min* (*Some ya*) = *None* ⟷ *mset-nodes ya* = {#*node ya*#}› **and**
del-min-Some-mset-nodes: ‹*del-min* (*Some ya*) = *Some yb* ⟹ *mset-nodes ya* = *add-mset* (*node ya*) (*mset-nodes yb*)›
⟨*proof*⟩

lemma *mset-nodes-del-min*[*simp*]:
‹*del-min* (*Some ya*) ≠ *None* ⟹ *mset-nodes* (*the* (*del-min* (*Some ya*))) = *remove1-mset* (*node ya*) (*mset-nodes ya*)›
⟨*proof*⟩

lemma *hp-score-del-min*:
‹*h* ≠ *None* ⟹ *del-min h* ≠ *None* ⟹ *distinct-mset* (*mset-nodes* (*the h*)) ⟹ *hp-score a* (*the* (*del-min h*)) = (*if a* = *get-min2 h then None else hp-score a* (*the h*))›
⟨*proof*⟩

lemma *del-min-prio-del*: ‹(*j*, *h*) ∈ *hmrel* ⟹ *fst* (*snd h*) ≠ {#}⟹
((*fst j*, *del-min* (*snd j*)), *prio-del* (*get-min2* (*snd j*)) *h*) ∈ *hmrel*›
⟨*proof*⟩

definition *mop-hm-old-weight* :: ‹-› **where**
‹*mop-hm-old-weight w* = (λ(𝒜, *xs*). *do* {
ASSERT (*w* ∈# 𝒜);
if xs ≠ *None* ∧ *w* ∈# *mset-nodes* (*the xs*) *then RETURN* (*the* (*hp-score w* (*the xs*)))
else RES UNIV
})›

This requires a stronger invariant than what we want to do.

lemma *mop-hm-old-weight-mop-prio-old-weight*:
‹(*xs*, *ys*) ∈ *hmrel* ⟹ *mop-hm-old-weight w xs* ≤ ⇓*Id* (*mop-prio-old-weight w ys*)›
⟨*proof*⟩

end

definition *hp-is-in* :: ‹-› **where**
‹*hp-is-in w* = (λ*bw*. *do* {
ASSERT (*w* ∈# *fst bw*);
RETURN (*source-node bw* ≠ *None* ∧ (*hp-read-prev′ w bw* ≠ *None* ∨ *hp-read-parent′ w bw* ≠ *None* ∨ *the* (*source-node bw*) = *w*))
})›

lemma *hp-is-in*:
assumes ‹*encoded-hp-prop-list-conc arr h*›

shows ‹*hp-is-in i arr* $\leq$ $\Downarrow$*bool-rel* (*ACIDS.mop-hm-is-in i h*)›
⟨*proof*⟩

lemma *hp-is-in-mop-prio-is-in*:
assumes ‹(*arr*, *h*) $\in$ *acids-encoded-hmrel*›
shows ‹*hp-is-in a arr* $\leq$ $\Downarrow$*bool-rel* (*ACIDS.mop-prio-is-in a h*)›
⟨*proof*⟩

lemma *hp-is-in-mop-prio-is-in2*:
‹(*uncurry hp-is-in*, *uncurry ACIDS.mop-prio-is-in*) $\in$ *nat-rel* $\times_f$ *acids-encoded-hmrel* $\rightarrow_f$ ⟨*bool-rel*⟩*nres-rel*›
⟨*proof*⟩

lemma *vsids-pop-min2-mop-prio-pop-min*:
fixes *arr* :: ‹*'a::linorder multiset* $\times$ (*'a*, *nat*) *hp-fun* $\times$ *'a option*›
assumes ‹(*arr*, *h*) $\in$ *acids-encoded-hmrel*›
shows ‹*vsids-pop-min2 arr* $\leq$ $\Downarrow$(*Id*$\times_r$*acids-encoded-hmrel*)(*ACIDS.mop-prio-pop-min h*)›
⟨*proof*⟩

lemma *vsids-pop-min2-mop-prio-pop-min2*:
‹(*vsids-pop-min2*, *ACIDS.mop-prio-pop-min*) $\in$ *acids-encoded-hmrel* $\rightarrow_f$ ⟨*nat-rel* $\times_r$ *acids-encoded-hmrel*⟩*nres-rel*›
⟨*proof*⟩

definition *mop-hp-read-score* :: ‹-› **where**
‹*mop-hp-read-score x* = (λ($\mathcal{A}$, *w*, *h*). *do* {
ASSERT (*x* $\in\#$ $\mathcal{A}$);
if hp-read-score x w $\neq$ *None then RETURN* (*the* (*hp-read-score x w*)) *else RES UNIV*
}) ›

lemma *mop-hp-read-score-mop-hm-old-weight*:
assumes ‹*encoded-hp-prop-list-conc arr h*›
shows
‹*mop-hp-read-score w arr* $\leq$ $\Downarrow$*Id* (*ACIDS.mop-hm-old-weight w h*)›
⟨*proof*⟩

lemma *mop-hp-read-score-mop-prio-old-weight*:
fixes *arr* :: ‹*'a::linorder multiset* $\times$ (*'a*, *nat*) *hp-fun* $\times$ *'a option*›
assumes ‹(*arr*, *h*) $\in$ *acids-encoded-hmrel*›
shows ‹*mop-hp-read-score w arr* $\leq$ $\Downarrow$(*Id*)(*ACIDS.mop-prio-old-weight w h*)›
⟨*proof*⟩

lemma *mop-hp-read-score-mop-prio-old-weight2*:
‹(*uncurry mop-hp-read-score*, *uncurry ACIDS.mop-prio-old-weight*) $\in$ *nat-rel* $\times_r$ *acids-encoded-hmrel* $\rightarrow_f$ ⟨*Id*⟩*nres-rel*›
⟨*proof*⟩

thm *ACIDS.mop-prio-insert-raw-unchanged-def*
thm *ACIDS.mop-prio-insert-maybe-def*
term *ACIDS.prio-peek-min*
thm *ACIDS.mop-prio-old-weight-def*
thm *ACIDS.mop-prio-insert-raw-unchanged-def*
term *ACIDS.mop-prio-insert-unchanged*
end
theory *Pairing-Heaps-Impl*
imports *Relational-Pairing-Heaps*
Map-Fun-Rel

begin

hide-const (**open**) *NEMonad.ASSERT NEMonad.RETURN NEMonad.SPEC*

1.2 Imperative Pairing heaps

type-synonym ('a,'b)*pairing-heaps-imp* = ‹('a option list × 'a option list × 'a option list × 'a option list × 'b list × 'a option)›
definition *pairing-heaps-rel* :: ‹('a option × nat option) set ⇒ ('b option × 'c option) set ⇒ (('a,'b)pairing-heaps-imp × (nat multiset × (nat,'c) hp-fun × nat option)) set› **where**
pairing-heaps-rel-def-internal:
‹*pairing-heaps-rel R S* = {((*prevs'*, *nxts'*, *children'*, *parents'*, *scores'*, *h'*), ($\mathcal{V}$, (*prevs*, *nxts*, *children*, *parents*, *scores*), *h*)).
(*h'*, *h*) ∈ *R* ∧
(*prevs'*, *prevs*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*nxts'*, *nxts*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*children'*, *children*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*parents'*, *parents*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*map Some scores'*, *scores*) ∈ ⟨*S*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$)
}›

lemma *pairing-heaps-rel-def*:
‹⟨*R*,*S*⟩*pairing-heaps-rel* =
{((*prevs'*, *nxts'*, *children'*, *parents'*, *scores'*, *h'*), ($\mathcal{V}$, (*prevs*, *nxts*, *children*, *parents*, *scores*), *h*)).
(*h'*, *h*) ∈ *R* ∧
(*prevs'*, *prevs*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*nxts'*, *nxts*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*children'*, *children*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*parents'*, *parents*) ∈ ⟨*R*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$) ∧
(*map Some scores'*, *scores*) ∈ ⟨*S*⟩*map-fun-rel* ((λ*a*. (*a*,*a*))' *set-mset* $\mathcal{V}$)
}›
⟨*proof*⟩

definition *op-hp-read-nxt-imp* **where**
‹*op-hp-read-nxt-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
(*nxts* ! *i*)
})›
definition *mop-hp-read-nxt-imp* **where**
‹*mop-hp-read-nxt-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
ASSERT (*i* < *length nxts*);
RETURN (*nxts* ! *i*)
})›

lemma *op-hp-read-nxt-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
(*op-hp-read-nxt-imp i xs*, *hp-read-nxt' j ys*) ∈ *R*›
⟨*proof*⟩

lemma *mop-hp-read-nxt-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
mop-hp-read-nxt-imp i xs ≤ *SPEC* (λ*a*. (*a*, *hp-read-nxt' j ys*) ∈ *R*)›
⟨*proof*⟩

definition *op-hp-read-prev-imp* **where**

‹*op-hp-read-prev-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
prevs ! *i*
})›

definition *mop-hp-read-prev-imp* **where**
‹*mop-hp-read-prev-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
ASSERT (*i* < *length prevs*);
RETURN (*prevs* ! *i*)
})›

lemma *op-hp-read-prev-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
(*op-hp-read-prev-imp i xs*, *hp-read-prev′ j ys*) ∈ *R*›
⟨*proof*⟩

lemma *mop-hp-read-prev-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
mop-hp-read-prev-imp i xs ≤ *SPEC* (λ*a*. (*a*, *hp-read-prev′ j ys*) ∈ *R*)›
⟨*proof*⟩

definition *op-hp-read-child-imp* **where**
‹*op-hp-read-child-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
(*children* ! *i*)
})›

definition *mop-hp-read-child-imp* **where**
‹*mop-hp-read-child-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
ASSERT (*i* < *length children*);
RETURN (*children* ! *i*)
})›

lemma *op-hp-read-child-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
(*op-hp-read-child-imp i xs*, *hp-read-child′ j ys*) ∈ *R*›
⟨*proof*⟩

lemma *mop-hp-read-child-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
mop-hp-read-child-imp i xs ≤ *SPEC* (λ*a*. (*a*, *hp-read-child′ j ys*) ∈ *R*)›
⟨*proof*⟩

definition *mop-hp-read-parent-imp* **where**
‹*mop-hp-read-parent-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
ASSERT (*i* < *length parents*);
RETURN (*parents* ! *i*)
})›
definition *op-hp-read-parent-imp* **where**
‹*op-hp-read-parent-imp* = (λ*i* (*prevs*, *nxts*, *children*, *parents*, *scores*, *h*). *do* {
(*parents* ! *i*)
})›

lemma *op-hp-read-parent-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
(*op-hp-read-parent-imp i xs*, *hp-read-parent′ j ys*) ∈ *R*›
⟨*proof*⟩

lemma *mop-hp-read-parent-imp-spec*:
‹$(xs, ys) \in \langle R,S\rangle pairing\text{-}heaps\text{-}rel \Longrightarrow (i,j) \in nat\text{-}rel \Longrightarrow j \in\# fst\ ys \Longrightarrow$
$mop\text{-}hp\text{-}read\text{-}parent\text{-}imp\ i\ xs \le SPEC\ (\lambda a.\ (a,\ hp\text{-}read\text{-}parent'\ j\ ys) \in R)$›
⟨*proof*⟩

definition *op-hp-read-score-imp* :: ‹$nat \Rightarrow ('a,'b) pairing\text{-}heaps\text{-}imp \Rightarrow 'b$› **where**
‹$op\text{-}hp\text{-}read\text{-}score\text{-}imp = (\lambda i\ (prevs,\ nxts,\ children,\ parents,\ scores,\ h).\ do\ \{$
$((scores\ !\ i))$
$\})$›

definition *mop-hp-read-score-imp* :: ‹$nat \Rightarrow ('a,'b) pairing\text{-}heaps\text{-}imp \Rightarrow 'b\ nres$› **where**
‹$mop\text{-}hp\text{-}read\text{-}score\text{-}imp = (\lambda i\ (prevs,\ nxts,\ children,\ parents,\ scores,\ h).\ do\ \{$
$ASSERT\ (i < length\ scores);$
$RETURN\ ((scores\ !\ i))$
$\})$›

lemma *mop-hp-read-score-imp-spec*:
‹$(xs, ys) \in \langle R,S\rangle pairing\text{-}heaps\text{-}rel \Longrightarrow (i,j) \in nat\text{-}rel \Longrightarrow j \in\# fst\ ys \Longrightarrow$
$mop\text{-}hp\text{-}read\text{-}score\text{-}imp\ i\ xs \le SPEC\ (\lambda a.\ (Some\ a,\ hp\text{-}read\text{-}score'\ j\ ys) \in S)$›
⟨*proof*⟩

fun *hp-set-all′* **where**
‹$hp\text{-}set\text{-}all'\ i\ p\ q\ r\ s\ t\ (\mathcal{V},\ u,\ h) = (\mathcal{V},\ hp\text{-}set\text{-}all\ i\ p\ q\ r\ s\ t\ u,\ h)$›

definition *mop-hp-set-all-imp* :: ‹$nat \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow ('a,'b) pairing\text{-}heaps\text{-}imp \Rightarrow ('a,'b) pairing\text{-}heaps\text{-}imp$
$nres$›**where**
‹$mop\text{-}hp\text{-}set\text{-}all\text{-}imp = (\lambda i\ p\ q\ r\ s\ t\ (prevs,\ nxts,\ children,\ parents,\ scores,\ h).\ do\ \{$
$ASSERT\ (i < length\ nxts \land i < length\ prevs \land i < length\ parents \land i < length\ children \land i <$
$length\ scores);$
$RETURN\ (prevs[i := p],\ nxts[i:=q],\ children[i:=r],\ parents[i:=s],\ scores[i:=t],\ h)$
$\})$›

lemma *mop-hp-set-all-imp-spec*:
‹$(xs, ys) \in \langle R,S\rangle pairing\text{-}heaps\text{-}rel \Longrightarrow (i,j) \in nat\text{-}rel \Longrightarrow$
$(p',p) \in R \Longrightarrow (q',q) \in R \Longrightarrow (r',r) \in R \Longrightarrow (s',s) \in R \Longrightarrow (Some\ t',t) \in S \Longrightarrow j \in\# fst\ ys \Longrightarrow$
$mop\text{-}hp\text{-}set\text{-}all\text{-}imp\ i\ p'\ q'\ r'\ s'\ t'\ xs \le SPEC\ (\lambda a.\ (a,\ hp\text{-}set\text{-}all'\ j\ p\ q\ r\ s\ t\ ys) \in \langle R,S\rangle pairing\text{-}heaps\text{-}rel)$›
⟨*proof*⟩

lemma *fst-hp-set-all′*[*simp*]: ‹$fst\ (hp\text{-}set\text{-}all'\ i\ p\ q\ r\ s\ t\ x) = fst\ x$›
⟨*proof*⟩

fun *update-source-node-impl* :: ‹$- \Rightarrow ('a,'b) pairing\text{-}heaps\text{-}imp \Rightarrow ('a,'b) pairing\text{-}heaps\text{-}imp$› **where**
‹$update\text{-}source\text{-}node\text{-}impl\ i\ (prevs,\ nxts,\ parents,\ children,\ scores,-) = (prevs,\ nxts,\ parents,\ children,$
$scores,\ i)$›

fun *source-node-impl* :: ‹$('a,'b) pairing\text{-}heaps\text{-}imp \Rightarrow 'a\ option$› **where**
‹$source\text{-}node\text{-}impl\ (prevs,\ nxts,\ parents,\ children,\ scores,h) = h$›

lemma *update-source-node-impl-spec*:
‹$(xs, ys) \in \langle R,S\rangle pairing\text{-}heaps\text{-}rel \Longrightarrow (i,j) \in R \Longrightarrow$
$(update\text{-}source\text{-}node\text{-}impl\ i\ xs,\ update\text{-}source\text{-}node\ j\ ys) \in \langle R,S\rangle pairing\text{-}heaps\text{-}rel$›
⟨*proof*⟩

lemma *source-node-spec*:
‹$(xs, ys) \in \langle R,S\rangle pairing\text{-}heaps\text{-}rel \Longrightarrow$

(*source-node-impl xs*, *source-node ys*) ∈ *R*›
⟨*proof*⟩

lemma *hp-insert-alt-def*:
‹*hp-insert* = (λ*i w arr*. *do* {
let h = *source-node arr*;
if h = *None then do* {
  ASSERT (*i* ∈# *fst arr*);
  let arr = (*hp-set-all′ i None None None None* (*Some w*) *arr*);
  RETURN (*update-source-node* (*Some i*) *arr*)
} *else do* {
  ASSERT (*i* ∈# *fst arr*);
  ASSERT (*hp-read-prev′ i arr* = *None*);
  ASSERT (*hp-read-parent′ i arr* = *None*);
  let j = *the h*;
  ASSERT (*j* ∈# (*fst arr*) ∧ *j* ≠ *i*);
  ASSERT (*hp-read-score′ j* (*arr*) ≠ *None*);
  ASSERT (*hp-read-prev′ j arr* = *None* ∧ *hp-read-nxt′ j arr* = *None* ∧ *hp-read-parent′ j arr* = *None*);
  let y = (*the* (*hp-read-score′ j arr*));
  if y < *w*
  then do {
    let arr = *hp-set-all′ i None None* (*Some j*) *None* (*Some w*) *arr*;
    ASSERT (*j* ∈# *fst arr*);
    let arr = *hp-update-parents′ j* (*Some i*) *arr*;
    RETURN (*update-source-node* (*Some i*) *arr*)
  }
  else do {
    let child = *hp-read-child′ j arr*;
    ASSERT (*child* ≠ *None* ⟶ *the child* ∈# *fst arr*);
    let arr = *hp-set-all′ j None None* (*Some i*) *None* (*Some y*) *arr*;
    ASSERT (*i* ∈# *fst arr*);
    let arr = *hp-set-all′ i None child None* (*Some j*) (*Some* (*w*)) *arr*;
    ASSERT (*child* ≠ *None* ⟶ *the child* ∈# *fst arr*);
    let arr = (*if child* = *None then arr else hp-update-prev′* (*the child*) (*Some i*) *arr*);
    ASSERT (*child* ≠ *None* ⟶ *the child* ∈# *fst arr*);
    let arr = (*if child* = *None then arr else hp-update-parents′* (*the child*) *None arr*);
    RETURN arr
  }
}
})› (**is** ‹*?A* = *?B*›)
⟨*proof*⟩

definition *mop-hp-update-prev′-imp* :: ‹*nat* ⇒ ′*a option* ⇒ (′*a*,′*b*)*pairing-heaps-imp* ⇒ (′*a*,′*b*)*pairing-heaps-imp nres*› **where**
‹*mop-hp-update-prev′-imp* = (λ*i v* (*prevs*, *nxts*, *parents*, *children*). *do* {
  ASSERT (*i* < *length prevs*);
  RETURN (*prevs*[*i*:=*v*], *nxts*, *parents*, *children*)
})›

lemma *mop-hp-update-prev′-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ *j* ∈# *fst ys* ⟹ (*i*,*j*)∈*nat-rel* ⟹
(*p′*,*p*)∈*R* ⟹
mop-hp-update-prev′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *hp-update-prev′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

definition *mop-hp-update-parent′-imp* :: ‹*nat* ⇒ ′*a option* ⇒ (′*a*,′*b*)*pairing-heaps-imp* ⇒ (′*a*,′*b*)*pairing-heaps-imp nres*› **where**
‹*mop-hp-update-parent′-imp* = (λ*i v* (*prevs*, *nxts*,*children*, *parents*, *scores*). *do* {
ASSERT (*i* < *length parents*);
RETURN (*prevs*, *nxts*, *children*, *parents*[*i*:=*v*], *scores*)
})›

lemma *mop-hp-update-parent′-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ *j* ∈# *fst ys* ⟹ (*i*,*j*)∈*nat-rel* ⟹
(*p′*,*p*)∈*R* ⟹
mop-hp-update-parent′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *hp-update-parents′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

definition *mop-hp-update-nxt′-imp* :: ‹*nat* ⇒ ′*a option* ⇒ (′*a*,′*b*)*pairing-heaps-imp* ⇒ (′*a*,′*b*)*pairing-heaps-imp nres*› **where**
‹*mop-hp-update-nxt′-imp* = (λ*i v* (*prevs*, *nxts*, *parents*, *children*). *do* {
ASSERT (*i* < *length nxts*);
RETURN (*prevs*, *nxts*[*i*:=*v*], *parents*, *children*)
})›

lemma *mop-hp-update-nxt′-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ *j* ∈# *fst ys* ⟹ (*i*,*j*)∈*nat-rel* ⟹
(*p′*,*p*)∈*R* ⟹
mop-hp-update-nxt′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *hp-update-nxt′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

definition *mop-hp-update-score-imp* :: ‹*nat* ⇒ ′*b* ⇒ (′*a*,′*b*)*pairing-heaps-imp* ⇒ (′*a*,′*b*)*pairing-heaps-imp nres*› **where**
‹*mop-hp-update-score-imp* = (λ*i v* (*prevs*, *nxts*, *parents*, *children*, *scores*, *h*). *do* {
ASSERT (*i* < *length scores*);
RETURN (*prevs*, *nxts*, *parents*, *children*, *scores*[*i*:=*v*], *h*)
})›

lemma *mop-hp-update-score-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈*nat-rel* ⟹ *j* ∈# *fst ys* ⟹
(*Some p′*, *p*)∈*S* ⟹
mop-hp-update-score-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *hp-update-score′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

definition *mop-hp-update-child′-imp* :: ‹*nat* ⇒ ′*a option* ⇒ (′*a*,′*b*)*pairing-heaps-imp* ⇒ (′*a*,′*b*)*pairing-heaps-imp nres*› **where**
‹*mop-hp-update-child′-imp* = (λ*i v* (*prevs*, *nxts*, *children*, *parents*, *scores*). *do* {
ASSERT (*i* < *length children*);
RETURN (*prevs*, *nxts*, *children*[*i*:=*v*], *parents*, *scores*)
})›

lemma *mop-hp-update-child′-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ *j* ∈# *fst ys* ⟹ (*i*,*j*)∈*nat-rel* ⟹
(*p′*,*p*)∈*R* ⟹
mop-hp-update-child′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *hp-update-child′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›

⟨proof⟩

definition *mop-hp-insert-impl* :: ‹nat ⇒ 'b::linorder ⇒ (nat,'b)pairing-heaps-imp ⇒ (nat,'b)pairing-heaps-imp nres› **where**

```
  ‹mop-hp-insert-impl = (λi (w::'b) (arr :: (nat,'b)pairing-heaps-imp). do {
  let h = source-node-impl arr;
  if h = None then do {
    arr ← mop-hp-set-all-imp i None None None None w arr;
    RETURN (update-source-node-impl (Some i) arr)
   } else do {
    ASSERT (op-hp-read-prev-imp i arr = None);
    ASSERT (op-hp-read-parent-imp i arr = None);
    let j = (the h);
   ASSERT (op-hp-read-prev-imp j arr = None ∧ op-hp-read-nxt-imp j arr = None ∧ op-hp-read-parent-imp
j arr = None);
    y ← mop-hp-read-score-imp j arr;
    if y < w
    then do {
      arr ← mop-hp-set-all-imp i None None (Some j) None ((w)) (arr);
      arr← mop-hp-update-parent'-imp j (Some i) arr;
      RETURN (update-source-node-impl (Some i) arr)
    }
    else do {
      child ← mop-hp-read-child-imp j arr;
      arr ← mop-hp-set-all-imp j None None (Some i) None (y) arr;
      arr ← mop-hp-set-all-imp i None child None (Some j) w arr;
     arr ← (if child = None then RETURN arr else mop-hp-update-prev'-imp (the child) (Some i) arr);
      arr ← (if child = None then RETURN arr else mop-hp-update-parent'-imp (the child) None arr);
      RETURN arr
    }
   }
  })›
```

lemma *Some-x-y-option-theD*: ‹(Some x, y) ∈ ⟨S⟩option-rel ⟹ (x, the y) ∈ S›
⟨proof⟩

context
begin
private lemma *in-pairing-heaps-rel-still*: ‹(arra, arr') ∈ ⟨⟨nat-rel⟩option-rel, ⟨S⟩option-rel⟩pairing-heaps-rel ⟹ arr' = arr'' ⟹
    (arra, arr'') ∈ ⟨⟨nat-rel⟩option-rel, ⟨S⟩option-rel⟩pairing-heaps-rel›
⟨proof⟩

lemma *mop-hp-insert-impl-spec*:
 assumes ‹(xs, ys) ∈ ⟨⟨nat-rel⟩option-rel,⟨nat-rel⟩option-rel⟩pairing-heaps-rel› ‹(i,j)∈nat-rel› ‹(w,w')∈nat-rel›
 shows ‹mop-hp-insert-impl i w xs ≤ ⇓(⟨⟨nat-rel⟩option-rel,⟨nat-rel⟩option-rel⟩pairing-heaps-rel) (hp-insert j w' ys)›
⟨proof⟩

lemma *hp-link-alt-def*:

```
   ‹hp-link = (λ(i::'a) j arr. do {
    ASSERT (i ≠ j);
    ASSERT (i ∈# fst arr);
    ASSERT (j ∈# fst arr);
```

$ASSERT$ (*hp-read-score' i arr* $\neq$ *None*);
$ASSERT$ (*hp-read-score' j arr* $\neq$ *None*);
let x = (*the* (*hp-read-score' i arr*)::'*b*::*order*);
let y = (*the* (*hp-read-score' j arr*)::'*b*);
let prev = *hp-read-prev' i arr*;
let nxt = *hp-read-nxt' j arr*;
$ASSERT$ (*nxt* $\neq$ *Some i* $\wedge$ *nxt* $\neq$ *Some j*);
$ASSERT$ (*prev* $\neq$ *Some i* $\wedge$ *prev* $\neq$ *Some j*);
let (*parent*,*ch*,w_p, w_{ch}) = (*if* $y < x$ *then* (i, j, x, y) *else* (j, i, y, x));
$ASSERT$ (*parent* $\in$# *fst arr*);
$ASSERT$ (*ch* $\in$# *fst arr*);
let child = *hp-read-child' parent arr*;
$ASSERT$ (*child* $\neq$ *Some i* $\wedge$ *child* $\neq$ *Some j*);
let $child_{ch}$ = *hp-read-child' ch arr*;
$ASSERT$ ($child_{ch}$ $\neq$ *Some i* $\wedge$ $child_{ch}$ $\neq$ *Some j* $\wedge$ ($child_{ch}$ $\neq$ *None* $\longrightarrow$ $child_{ch}$ $\neq$ *child*));
$ASSERT$ (*distinct* ([i, j] @ (*if* $child_{ch}$ $\neq$ *None then* [*the* $child_{ch}$] *else* [])
@ (*if child* $\neq$ *None then* [*the child*] *else* [])
@ (*if prev* $\neq$ *None then* [*the prev*] *else* [])
@ (*if nxt* $\neq$ *None then* [*the nxt*] *else* []))
);
$ASSERT$ (*ch* $\in$# *fst arr*);
$ASSERT$ (*parent* $\in$# *fst arr*);
$ASSERT$ (*child* $\neq$ *None* $\longrightarrow$ *the child* $\in$# *fst arr*);
$ASSERT$ (*nxt* $\neq$ *None* $\longrightarrow$ *the nxt* $\in$# *fst arr*);
$ASSERT$ (*prev* $\neq$ *None* $\longrightarrow$ *the prev* $\in$# *fst arr*);
let arr = *hp-set-all' parent prev nxt* (*Some ch*) *None* (*Some* (w_p::'*b*)) *arr*;
let arr = *hp-set-all' ch None child* $child_{ch}$ (*Some parent*) (*Some* (w_{ch}::'*b*)) *arr*;
let arr = (*if child* = *None then arr else hp-update-prev'* (*the child*) (*Some ch*) *arr*);
let arr = (*if nxt* = *None then arr else hp-update-prev'* (*the nxt*) (*Some parent*) *arr*);
let arr = (*if prev* = *None then arr else hp-update-nxt'* (*the prev*) (*Some parent*) *arr*);
let arr = (*if child* = *None then arr else hp-update-parents'* (*the child*) *None arr*);
$RETURN$ (*arr*, *parent*)
})› (**is** ‹*?A* = *?B*›)
⟨*proof*⟩

definition *maybe-mop-hp-update-prev'-imp* **where**
‹*maybe-mop-hp-update-prev'-imp child ch arr* =
(*if child* = *None then RETURN arr else mop-hp-update-prev'-imp* (*the child*) *ch arr*)›

definition *maybe-mop-hp-update-nxt'-imp* **where**
‹*maybe-mop-hp-update-nxt'-imp child ch arr* =
(*if child* = *None then RETURN arr else mop-hp-update-nxt'-imp* (*the child*) *ch arr*)›

definition *maybe-mop-hp-update-child'-imp* **where**
‹*maybe-mop-hp-update-child'-imp child ch arr* =
(*if child* = *None then RETURN arr else mop-hp-update-child'-imp* (*the child*) *ch arr*)›

definition *maybe-mop-hp-update-parent'-imp* **where**
‹*maybe-mop-hp-update-parent'-imp child ch arr* =
(*if child* = *None then RETURN arr else mop-hp-update-parent'-imp* (*the child*) *ch arr*)›

lemma *maybe-mop-hp-update-prev'-imp-spec*:
‹(*xs*, *ys*) $\in$ ⟨*R*,*S*⟩*pairing-heaps-rel* $\Longrightarrow$ (*i*,*j*)$\in$⟨*nat-rel*⟩*option-rel* $\Longrightarrow$ (*j* $\neq$ *None* $\Longrightarrow$ *the j* $\in$# *fst ys*) $\Longrightarrow$
(p',p)$\in$*R* $\Longrightarrow$

maybe-mop-hp-update-prev′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *maybe-hp-update-prev′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

lemma *maybe-mop-hp-update-nxt′-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈⟨*nat-rel*⟩*option-rel* ⟹ (*j* ≠ *None* ⟹ *the j* ∈# *fst ys*) ⟹
(*p′*,*p*)∈*R* ⟹
maybe-mop-hp-update-nxt′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *maybe-hp-update-nxt′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

lemma *maybe-mop-hp-update-parent′-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈⟨*nat-rel*⟩*option-rel* ⟹ (*j* ≠ *None* ⟹ *the j* ∈# *fst ys*) ⟹
(*p′*,*p*)∈*R* ⟹
maybe-mop-hp-update-parent′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *maybe-hp-update-parents′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

lemma *maybe-mop-hp-update-child′-imp-spec*:
‹(*xs*, *ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel* ⟹ (*i*,*j*)∈⟨*nat-rel*⟩*option-rel* ⟹ (*j* ≠ *None* ⟹ *the j* ∈# *fst ys*) ⟹
(*p′*,*p*)∈*R* ⟹
maybe-mop-hp-update-child′-imp i p′ xs ≤ *SPEC* (λ*a*. (*a*, *maybe-hp-update-child′ j p ys*) ∈ ⟨*R*,*S*⟩*pairing-heaps-rel*)›
⟨*proof*⟩

definition *mop-hp-link-imp* :: ‹*nat* ⇒*nat* ⇒(*nat*, ′*b*::*ord*)*pairing-heaps-imp* ⇒ - *nres*› **where**
‹*mop-hp-link-imp* = (λ*i j arr*. *do* {
  ASSERT (*i* ≠ *j*);
  x ← *mop-hp-read-score-imp i arr*;
  y ← *mop-hp-read-score-imp j arr*;
  prev ← *mop-hp-read-prev-imp i arr*;
  nxt ← *mop-hp-read-nxt-imp j arr*;
  let (*parent*,*ch*,w_p, w_{ch}) = (*if y* < *x then* (*i*, *j*, *x*, *y*) *else* (*j*, *i*, *y*, *x*));
  child ← *mop-hp-read-child-imp parent arr*;
  $child_{ch}$ ← *mop-hp-read-child-imp ch arr*;
  arr ← *mop-hp-set-all-imp parent prev nxt* (*Some ch*) *None* ((w_p)) *arr*;
  arr ← *mop-hp-set-all-imp ch None child* $child_{ch}$ (*Some parent*) ((w_{ch})) *arr*;
  arr ← (*if child* = *None then RETURN arr else mop-hp-update-prev′-imp* (*the child*) (*Some ch*) *arr*);
  arr ← (*if nxt* = *None then RETURN arr else mop-hp-update-prev′-imp* (*the nxt*) (*Some parent*) *arr*);
  arr ← (*if prev* = *None then RETURN arr else mop-hp-update-nxt′-imp* (*the prev*) (*Some parent*) *arr*);
  arr ← (*if child* = *None then RETURN arr else mop-hp-update-parent′-imp* (*the child*) *None arr*);
  RETURN (*arr*, *parent*)
  })›

lemma *mop-hp-link-imp-spec*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*› ‹(*i*,*j*)∈*nat-rel*› ‹(*w*,*w′*)∈*nat-rel*›
shows ‹*mop-hp-link-imp i w xs* ≤ ⇓(⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel* $\times_r$ *nat-rel*) (*hp-link j w′ ys*)›
⟨*proof*⟩

lemma *vsids-pass*$_1$*-alt-def*:
‹*vsids-pass*$_1$ = (λ(*arr*::′*a multiset* × (′*a*,′*c*::*order*) *hp-fun* × ′*a option*) (*j*::′*a*). *do* {
(*arr*, *j*, -, *n*) ← $WHILE_T$(λ(*arr*, *j*,-, -). *j* ≠ *None*)
(λ(*arr*, *j*, *e*::*nat*, *n*). *do* {
  if j = *None then RETURN* (*arr*, *None*, *e*, *n*)
  else do {

let j = the j;
ASSERT (j ∈# fst arr);
let nxt = hp-read-nxt′ j arr;
if nxt = None then RETURN (arr, nxt, e+1, j)
else do {
ASSERT (nxt ≠ None);
ASSERT (the nxt ∈# fst arr);
let nnxt = hp-read-nxt′ (the nxt) arr;
(arr, n) ← hp-link j (the nxt) arr;
RETURN (arr, nnxt, e+2, n)
}}
})
(arr, Some j, 0::nat, j);
RETURN (arr, n)
})› (**is** ‹*?A = ?B*›)
⟨*proof*⟩

definition *mop-vsids-pass*$_1$*-imp* :: ‹*(nat, ′b::ord)pairing-heaps-imp ⇒ nat ⇒ - nres*› **where**
‹*mop-vsids-pass*$_1$*-imp = (λarr j. do {*
(arr, j, n) ← $WHILE_T$*(λ(arr, j, -). j ≠ None)*
(λ(arr, j, n). do {
if j = None then RETURN (arr, None, n)
else do {
let j = the j;
nxt ← mop-hp-read-nxt-imp j arr;
if nxt = None then RETURN (arr, nxt, j)
else do {
ASSERT (nxt ≠ None);
nnxt ← mop-hp-read-nxt-imp (the nxt) arr;
(arr, n) ← mop-hp-link-imp j (the nxt) arr;
RETURN (arr, nnxt, n)
}}
})
(arr, Some j, j);
RETURN (arr, n)
})›

lemma *mop-vsids-pass*$_1$*-imp-spec*:
assumes ‹*(xs, ys) ∈ ⟨⟨nat-rel⟩option-rel,⟨nat-rel⟩option-rel⟩pairing-heaps-rel*› ‹*(i,j)∈nat-rel*›
shows ‹*mop-vsids-pass*$_1$*-imp xs i ≤ ⇓(⟨⟨nat-rel⟩option-rel,⟨nat-rel⟩option-rel⟩pairing-heaps-rel* $\times_r$ *nat-rel)*
(vsids-pass$_1$ *ys j)*›
⟨*proof*⟩

lemma *vsids-pass*$_2$*-alt-def*:
‹*vsids-pass*$_2$ *= (λarr (j::′a). do {*
ASSERT (j ∈# fst arr);
let nxt = hp-read-prev′ j arr;
(arr, j, leader, -) ← $WHILE_T$*(λ(arr, j, leader, e). j ≠ None)*
(λ(arr, j, leader, e::nat). do {
if j = None then RETURN (arr, None, leader, e)
else do {
let j = the j;
ASSERT (j ∈# fst arr);
let nnxt = hp-read-prev′ j arr;
(arr, n) ← hp-link j leader arr;

RETURN (*arr*, *nnxt*, *n*, *e+1*)
}
})
(*arr*, *nxt*, *j*, *1*::*nat*);
RETURN (*update-source-node* (*Some leader*) *arr*)
})› (**is** ‹*?A* = *?B*›)
⟨*proof*⟩

definition *mop-vsids-pass*$_2$*-imp* **where**
‹*mop-vsids-pass*$_2$*-imp* = (λ*arr* (*j*::*nat*). *do* {
nxt ← *mop-hp-read-prev-imp j arr*;
(*arr*, *j*, *leader*) ← *WHILE*$_T$(λ(*arr*, *j*, *leader*). *j* ≠ *None*)
(λ(*arr*, *j*, *leader*). *do* {
if j = *None then RETURN* (*arr*, *None*, *leader*)
else do {
let j = *the j*;
nnxt ← *mop-hp-read-prev-imp j arr*;
(*arr*, *n*) ← *mop-hp-link-imp j leader arr*;
RETURN (*arr*, *nnxt*, *n*)
}
})
(*arr*, *nxt*, *j*);
RETURN (*update-source-node-impl* (*Some leader*) *arr*)
})›

lemma *mop-vsids-pass*$_2$*-imp-spec*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*› ‹(*i*,*j*)∈*nat-rel*›
shows ‹*mop-vsids-pass*$_2$*-imp xs i* ≤ ⇓(⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*) (*vsids-pass*$_2$ *ys j*)›
⟨*proof*⟩

definition *mop-merge-pairs-imp* **where**
‹*mop-merge-pairs-imp arr j* = *do* {
(*arr*, *j*) ← *mop-vsids-pass*$_1$*-imp arr j*;
mop-vsids-pass$_2$*-imp arr j*
}›

lemma *mop-merge-pairs-imp-spec*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*› ‹(*i*,*j*)∈*nat-rel*›
shows ‹*mop-merge-pairs-imp xs i* ≤ ⇓(⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*) (*merge-pairs ys j*)›
⟨*proof*⟩

lemma *vsids-pop-min-alt-def*:
‹*vsids-pop-min* = (λ*arr*. *do* {
let h = *source-node arr*;
if h = *None then RETURN* (*None*, *arr*)
else do {
ASSERT (*the h* ∈# *fst arr*);
let j = *hp-read-child′* (*the h*) *arr*;
if j = *None then RETURN* (*h*, (*update-source-node None arr*))
else do {
ASSERT (*the j* ∈# *fst arr*);
let arr = *hp-update-prev′* (*the h*) *None arr*;
let arr = *hp-update-child′* (*the h*) *None arr*;

let arr = *hp-update-parents'* (*the j*) *None arr*;
arr ← *merge-pairs* (*update-source-node None arr*) (*the j*);
RETURN (*h*, *arr*)
}
}
})› (**is** ‹*?A* = *?B*›)
⟨*proof*⟩

definition *mop-vsids-pop-min-impl* **where**
‹*mop-vsids-pop-min-impl* = (λ*arr*. *do* {
let h = *source-node-impl arr*;
if h = *None then RETURN* (*None*, *arr*)
else do {
j ← *mop-hp-read-child-imp* (*the h*) *arr*;
if j = *None then RETURN* (*h*, *update-source-node-impl None arr*)
else do {
arr ← *mop-hp-update-prev'-imp* (*the h*) *None arr*;
arr ← *mop-hp-update-child'-imp* (*the h*) *None arr*;
arr ← *mop-hp-update-parent'-imp* (*the j*) *None arr*;
arr ← *mop-merge-pairs-imp* (*update-source-node-impl None arr*) (*the j*);
RETURN (*h*, *arr*)
}
}
})›

lemma *mop-vsids-pop-min-impl*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*›
shows ‹*mop-vsids-pop-min-impl xs* ≤ ⇓(⟨*nat-rel*⟩*option-rel* $\times_r$ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*)
(*vsids-pop-min ys*)›
⟨*proof*⟩

definition *mop-vsids-pop-min2-impl* **where**
‹*mop-vsids-pop-min2-impl* = (λ*arr*. *do* {
let h = *source-node-impl arr*;
ASSERT (*h* ≠ *None*);
j ← *mop-hp-read-child-imp* (*the h*) *arr*;
if j = *None then RETURN* (*the h*, *update-source-node-impl None arr*)
else do {
arr ← *mop-hp-update-prev'-imp* (*the h*) *None arr*;
arr ← *mop-hp-update-child'-imp* (*the h*) *None arr*;
arr ← *mop-hp-update-parent'-imp* (*the j*) *None arr*;
arr ← *mop-merge-pairs-imp* (*update-source-node-impl None arr*) (*the j*);
RETURN (*the h*, *arr*)
}
})›

lemma *vsids-pop-min2-alt-def*:
‹*vsids-pop-min2* = (λ*arr*. *do* {
let h = *source-node arr*;
ASSERT (*h* ≠ *None*);
ASSERT (*the h* ∈# *fst arr*);

let j = hp-read-child′ (the h) arr;
if j = None then RETURN (the h, (update-source-node None arr))
else do {
ASSERT (the j ∈# fst arr);
let arr = hp-update-prev′ (the h) None arr;
let arr = hp-update-child′ (the h) None arr;
let arr = hp-update-parents′ (the j) None arr;
arr ← merge-pairs (update-source-node None arr) (the j);
RETURN (the h, arr)
}
})› (**is** ‹?A = ?B›)
⟨*proof*⟩

lemma *mop-vsids-pop-min2-impl*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*›
shows ‹*mop-vsids-pop-min2-impl xs* ≤ ⇓(*nat-rel* $\times_r$ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*) (*vsids-pop-min2 ys*)›
⟨*proof*⟩

definition *mop-unroot-hp-tree* **where**
‹*mop-unroot-hp-tree arr h = do {*
let a = source-node-impl arr;
nnext ← mop-hp-read-nxt-imp h arr;
parent ← mop-hp-read-parent-imp h arr;
prev ← mop-hp-read-prev-imp h arr;
if prev = None ∧ parent = None ∧ Some h ≠ a then RETURN (update-source-node-impl None arr)
else if Some h = a then RETURN (update-source-node-impl None arr)
else do {
ASSERT (a ≠ None);
let a′ = the a;
arr ← maybe-mop-hp-update-child′-imp parent nnext arr;
arr ← maybe-mop-hp-update-nxt′-imp prev nnext arr;
arr ← maybe-mop-hp-update-prev′-imp nnext prev arr;
arr ← maybe-mop-hp-update-parent′-imp nnext parent arr;

arr ← mop-hp-update-nxt′-imp h None arr;
arr ← mop-hp-update-prev′-imp h None arr;
arr ← mop-hp-update-parent′-imp h None arr;

arr ← mop-hp-update-nxt′-imp h (Some a′) arr;
arr ← mop-hp-update-prev′-imp a′ (Some h) arr;
RETURN (update-source-node-impl None arr)
}
}›

lemma *mop-unroot-hp-tree-spec*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*› **and** ‹(*h*,*i*)∈*nat-rel*›
shows ‹*mop-unroot-hp-tree xs h* ≤ ⇓(⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*) (*unroot-hp-tree ys i*)›
⟨*proof*⟩

definition *mop-rescale-and-reroot* **where**
‹*mop-rescale-and-reroot h w′ arr = do {*
nnext ← mop-hp-read-nxt-imp h arr;
parent ← mop-hp-read-parent-imp h arr;
prev ← mop-hp-read-prev-imp h arr;

if source-node-impl arr = None then mop-hp-update-score-imp h w′ arr
else if prev = None ∧ parent = None ∧ Some h ≠ source-node-impl arr then mop-hp-update-score-imp h w′ arr
else if Some h = source-node-impl arr then mop-hp-update-score-imp h w′ arr
else do {
arr ← mop-unroot-hp-tree arr h;
arr ← mop-hp-update-score-imp h w′ arr;
mop-merge-pairs-imp arr h
}
}›

lemma *mop-rescale-and-reroot-spec*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*› **and** ‹(*h*,*i*)∈*nat-rel*› ‹(*w*, *w*′) ∈ *nat-rel*›
shows ‹*mop-rescale-and-reroot h w xs* ≤ ⇓(⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*) (*rescale-and-reroot i w*′ *ys*)›
⟨*proof*⟩

definition *mop-hp-is-in* :: ‹-› **where**
‹*mop-hp-is-in h* = (λ*arr*. *do* {
parent ← mop-hp-read-parent-imp h arr;
prev ← mop-hp-read-prev-imp h arr;
let s = source-node-impl arr;
RETURN (*s* ≠ *None* ∧ (*prev* ≠ *None* ∨ *parent* ≠ *None* ∨ *the s* = *h*))
})›

lemma *mop-hp-is-in-spec*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*› **and** ‹(*h*,*i*)∈*nat-rel*›
shows ‹*mop-hp-is-in h xs* ≤ ⇓*bool-rel* (*hp-is-in i ys*)›
⟨*proof*⟩

lemma *mop-hp-read-score-imp-mop-hp-read-score*:
assumes ‹(*xs*, *ys*) ∈ ⟨⟨*nat-rel*⟩*option-rel*,⟨*nat-rel*⟩*option-rel*⟩*pairing-heaps-rel*› **and** ‹(*h*,*i*)∈*nat-rel*›
shows ‹*mop-hp-read-score-imp h xs* ≤ ⇓*nat-rel* (*mop-hp-read-score i ys*)›
⟨*proof*⟩

end
end