Monadification, Memoization and Dynamic Programming

Simon Wimmer Shuwei Hu Tobias Nipkow

Technical University of Munich

March 17, 2025

Abstract

We present a lightweight framework for the automatic verified (functional or imperative) memoization of recursive functions. Our tool can turn a pure Isabelle/HOL function definition into a monadified version in a state monad or the Imperative HOL heap monad, and prove a correspondence theorem. We provide a variety of memory implementations for the two types of monads. A number of simple techniques allow us to achieve bottom-up computation and space-efficient memoization. The framework's utility is demonstrated on a number of representative dynamic programming problems. A detailed description of our work can be found in the accompanying paper [2].

Contents

0.1 State Monad

theory *State_Monad_Ext*
  imports *HOL−Library.State_Monad*
begin

definition *fun_app_lifted* :: $('M,'a \Rightarrow ('M, 'b)\ state)\ state \Rightarrow ('M,'a)\ state \Rightarrow ('M,'b)\ state$ **where**
  $fun\_app\_lifted\ f_T\ x_T \equiv do\ \{\ f \leftarrow f_T;\ x \leftarrow x_T;\ f\ x\ \}$

bundle *state_monad_syntax* **begin**

notation *fun_app_lifted* (**infixl** ‹.› 999)
type_synonym $('a,'M,'b)$ *fun_lifted* $= 'a \Rightarrow ('M,'b)$ *state* (‹_ ==_⟹ _› [3,1000,2] 2)
type_synonym $('a,'b)$ *dpfun* $= 'a$ ==$('a \rightharpoonup 'b)$⟹ $'b$ (**infixr** ‹$\Rightarrow_T$› 2)
type_notation *state* (‹[_| _]›)

notation *State_Monad.return* (‹⟨_⟩›)
notation (*ASCII*) *State_Monad.return* (‹(#_#)›)
notation *Transfer.Rel* (‹*Rel*›)

end

context includes *state_monad_syntax* **begin**

qualified lemma *return_app_return*:
  $\langle f\rangle\ .\ \langle x\rangle = f\ x$
  ⟨*proof*⟩ **lemma** *return_app_return_meta*:
  $\langle f\rangle\ .\ \langle x\rangle \equiv f\ x$
  ⟨*proof*⟩ **definition** if_T :: $('M,\ bool)\ state \Rightarrow ('M,\ 'a)\ state \Rightarrow ('M,\ 'a)\ state \Rightarrow ('M,\ 'a)\ state$ **where**
  $if_T\ b_T\ x_T\ y_T \equiv do\ \{b \leftarrow b_T;\ if\ b\ then\ x_T\ else\ y_T\}$
end

end

1 Monadification

1.1 Monads

theory *Pure_Monad*
  imports *Main*
begin

definition *Wrap* :: $'a \Rightarrow 'a$ **where**
  Wrap $x \equiv x$

definition *App* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
  App $f \equiv f$

lemma *Wrap_App_Wrap*:
  App (*Wrap* *f*) (*Wrap* *x*) $\equiv$ *f x*
  ⟨*proof*⟩

end

1.2 Parametricity of the State Monad

theory *DP_CRelVS*
  imports *./State_Monad_Ext ../Pure_Monad*
begin

definition *lift_p* :: $('s \Rightarrow bool) \Rightarrow ('s, 'a)$ *state* $\Rightarrow$ *bool* **where**
  lift_p P f =
    ($\forall$ *heap*. *P heap* $\longrightarrow$ (*case State_Monad.run_state f heap of* (_, *heap*)
$\Rightarrow$ *P heap*))

context
  fixes *P f heap*
  assumes *lift*: *lift_p P f* **and** *P*: *P heap*
begin

lemma *run_state_cases*:
  case State_Monad.run_state f heap of (_, *heap*) $\Rightarrow$ *P heap*
  ⟨*proof*⟩

lemma *lift_p_P*:
  P heap' **if** *State_Monad.run_state f heap* = (*v*, *heap'*)
  ⟨*proof*⟩

end

locale *state_mem_defs* =
  fixes *lookup* :: $'param \Rightarrow ('mem, 'result\ option)$ *state*
    and *update* :: $'param \Rightarrow 'result \Rightarrow ('mem, unit)$ *state*
begin

definition *checkmem* :: *'param* ⇒ (*'mem*, *'result*) *state* ⇒ (*'mem*, *'result*) *state* **where**
  checkmem param calc ≡ *do* {
    x ← *lookup param*;
    case x of
      Some x ⇒ *State_Monad.return x*
    | *None* ⇒ *do* {
        x ← *calc*;
        update param x;
        State_Monad.return x
      }
  }

abbreviation *checkmem_eq* ::
  (*'param* ⇒ (*'mem*, *'result*) *state*) ⇒ *'param* ⇒ (*'mem*, *'result*) *state* ⇒ *bool*
  (‹_$ _ =*CHECKMEM*= _› [1000,51] 51) **where**
  ($dp_T$$ *param* =*CHECKMEM*= *calc*) ≡ (dp_T *param* = *checkmem param calc*)
term 0

definition *map_of* **where**
  map_of heap k = *fst* (*run_state* (*lookup k*) *heap*)

definition *checkmem'* :: *'param* ⇒ (*unit* ⇒ (*'mem*, *'result*) *state*) ⇒ (*'mem*, *'result*) *state* **where**
  checkmem' param calc ≡ *do* {
    x ← *lookup param*;
    case x of
      Some x ⇒ *State_Monad.return x*
    | *None* ⇒ *do* {
        x ← *calc* ();
        update param x;
        State_Monad.return x
      }
  }

lemma *checkmem_checkmem'*:
  checkmem' param (λ_. *calc*) = *checkmem param calc*
  ⟨*proof*⟩

lemma *checkmem_eq_alt*:
  checkmem_eq dp param calc = (*dp param* = *checkmem' param* (λ _. *calc*))

⟨*proof*⟩

end

locale *mem_correct* = *state_mem_defs* +
fixes P
assumes *lookup_inv*: *lift_p* P (*lookup* k) **and** *update_inv*: *lift_p* P (*update* k v)
assumes
lookup_correct: $P\ m \Longrightarrow$ *map_of* (*snd* (*State_Monad.run_state* (*lookup* k) m)) $\subseteq_m$ (*map_of* m)
and
update_correct: $P\ m \Longrightarrow$ *map_of* (*snd* (*State_Monad.run_state* (*update* k v) m)) $\subseteq_m$ (*map_of* m)($k \mapsto v$)

locale *dp_consistency* =
mem_correct lookup update P
for *lookup* :: $'param \Rightarrow ('mem, 'result\ option)$ *state* **and** *update* **and** P +
fixes *dp* :: $'param \Rightarrow 'result$
begin

context
includes *lifting_syntax* **and** *state_monad_syntax*
begin

definition *cmem* :: $'mem \Rightarrow bool$ **where**
cmem $M \equiv \forall param \in dom\ (map\_of\ M).\ map\_of\ M\ param = Some\ (dp\ param)$

definition *crel_vs* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow ('mem, 'b)\ state \Rightarrow bool$ **where**
crel_vs $R\ v\ s \equiv \forall M.\ cmem\ M \wedge P\ M \longrightarrow (case\ State\_Monad.run\_state\ s\ M\ of\ (v', M') \Rightarrow R\ v\ v' \wedge cmem\ M' \wedge P\ M')$

abbreviation *rel_fun_lifted* :: $('a \Rightarrow 'c \Rightarrow bool) \Rightarrow ('b \Rightarrow 'd \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('c ==\_\Longrightarrow 'd) \Rightarrow bool$ (**infixr** ‹$===>_T$› 55) **where**
rel_fun_lifted $R\ R' \equiv R\ ===>\ crel\_vs\ R'$
term 0

definition *consistentDP* :: $('param == 'mem \Longrightarrow 'result) \Rightarrow bool$ **where**
consistentDP $\equiv ((=)\ ===>\ crel\_vs\ (=))\ dp$
term 0

private lemma *cmem_intro*:
assumes $\bigwedge param\ v\ M'.\ State\_Monad.run\_state\ (lookup\ param)\ M = (Some\ v,\ M') \Longrightarrow v = dp\ param$
shows *cmem M*
$\langle proof \rangle$

lemma *cmem_elim*:
assumes $cmem\ M\ State\_Monad.run\_state\ (lookup\ param)\ M = (Some\ v,\ M')$
obtains $dp\ param = v$
$\langle proof \rangle$
term 0

lemma *crel_vs_intro*:
assumes $\bigwedge M\ v'\ M'.\ [\![cmem\ M;\ P\ M;\ State\_Monad.run\_state\ v_T\ M = (v',\ M')]\!] \Longrightarrow R\ v\ v' \wedge cmem\ M' \wedge P\ M'$
shows $crel\_vs\ R\ v\ v_T$
$\langle proof \rangle$
term 0

lemma *crel_vs_elim*:
assumes $crel\_vs\ R\ v\ v_T\ \ cmem\ M\ P\ M$
obtains $v'\ M'$ **where** $State\_Monad.run\_state\ v_T\ M = (v',\ M')\ R\ v\ v'\ cmem\ M'\ P\ M'$
$\langle proof \rangle$
term 0

lemma *consistentDP_intro*:
assumes $\bigwedge param.\ Transfer.Rel\ (crel\_vs\ (=))\ (dp\ param)\ (dp_T\ param)$
shows $consistentDP\ dp_T$
$\langle proof \rangle$

lemma *crel_vs_return*:
$[\![Transfer.Rel\ R\ x\ y]\!] \Longrightarrow Transfer.Rel\ (crel\_vs\ R)\ (Wrap\ x)\ (State\_Monad.return\ y)$
$\langle proof \rangle$
term 0

lemma *crel_vs_return_ext*:
$[\![Transfer.Rel\ R\ x\ y]\!] \Longrightarrow Transfer.Rel\ (crel\_vs\ R)\ x\ (State\_Monad.return$

$y)$
  $\langle proof \rangle$
term 0

private lemma *cmem_upd*:
  cmem M' **if** *cmem* M P M *State_Monad.run_state* (*update param* (*dp param*)) $M = (v, M')$
  $\langle proof \rangle$ **lemma** *P_upd*:
  P M' **if** P M *State_Monad.run_state* (*update param* (*dp param*)) $M = (v, M')$
  $\langle proof \rangle$ **lemma** *crel_vs_get*:
  $[\![\bigwedge M.\ cmem\ M \Longrightarrow crel\_vs\ R\ v\ (sf\ M)]\!] \Longrightarrow crel\_vs\ R\ v\ (State\_Monad.get \gg\!= sf)$
  $\langle proof \rangle$
term 0

private lemma *crel_vs_set*:
  $[\![crel\_vs\ R\ v\ sf;\ cmem\ M;\ P\ M]\!] \Longrightarrow crel\_vs\ R\ v\ (State\_Monad.set\ M \gg sf)$
  $\langle proof \rangle$
term 0

private lemma *crel_vs_bind_eq*:
  $[\![crel\_vs\ (=)\ v\ s;\ crel\_vs\ R\ (f\ v)\ (sf\ v)]\!] \Longrightarrow crel\_vs\ R\ (f\ v)\ (s \gg\!= sf)$
  $\langle proof \rangle$
term 0

lemma *bind_transfer*[*transfer_rule*]:
  $(crel\_vs\ R0 ===> (R0 ===>_T R1) ===> crel\_vs\ R1)\ (\lambda v\ f.\ f\ v)\ (\gg\!=)$
  $\langle proof \rangle$ **lemma** *cmem_lookup*:
  cmem M' **if** *cmem* M P M *State_Monad.run_state* (*lookup param*) $M = (v, M')$
  $\langle proof \rangle$ **lemma** *P_lookup*:
  P M' **if** P M *State_Monad.run_state* (*lookup param*) $M = (v, M')$
  $\langle proof \rangle$

lemma *crel_vs_lookup*:
  $crel\_vs\ (\lambda\ v\ v'.\ case\ v'\ of\ None \Rightarrow True\ |\ Some\ v' \Rightarrow v = v' \wedge v = dp\ param)\ (dp\ param)\ (lookup\ param)$
  $\langle proof \rangle$

lemma *crel_vs_update*:
  $crel\_vs\ (=)\ ()\ (update\ param\ (dp\ param))$

⟨*proof*⟩ **lemma** *crel_vs_checkmem*:
$[\![$*is_equality R*; *Transfer.Rel* (*crel_vs R*) (*dp param*) *s*$]\!]$
$\Longrightarrow$ *Transfer.Rel* (*crel_vs R*) (*dp param*) (*checkmem param s*)
⟨*proof*⟩

lemma *crel_vs_checkmem_tupled*:
assumes *v* = *dp param*
shows $[\![$*is_equality R*; *Transfer.Rel* (*crel_vs R*) *v s*$]\!]$
$\Longrightarrow$ *Transfer.Rel* (*crel_vs R*) *v* (*checkmem param s*)
⟨*proof*⟩

lemma *return_transfer*[*transfer_rule*]:
(R ===>$_T$ R) *Wrap State_Monad.return*
⟨*proof*⟩

lemma *fun_app_lifted_transfer*[*transfer_rule*]:
(*crel_vs* ($R0$ ===>$_T$ $R1$) ===> *crel_vs* $R0$ ===> *crel_vs* $R1$) *App* (.)
⟨*proof*⟩

lemma *crel_vs_fun_app*:
$[\![$*Transfer.Rel* (*crel_vs* $R0$) x x_T; *Transfer.Rel* (*crel_vs* ($R0$ ===>$_T$ $R1$)) f $f_T$$]\!]$ $\Longrightarrow$ *Transfer.Rel* (*crel_vs* $R1$) (*App f x*) (f_T . x_T)
⟨*proof*⟩

lemma if_T*_transfer*[*transfer_rule*]:
(*crel_vs* (=) ===> *crel_vs R* ===> *crel_vs R* ===> *crel_vs R*) *If State_Monad_Ext.*if_T
⟨*proof*⟩
end

end
end

1.3 Miscellaneous Parametricity Theorems

theory *State_Heap_Misc*
imports *Main*
begin
context **includes** *lifting_syntax* **begin**
lemma *rel_fun_comp*:
assumes ($R1$ ===> $S1$) *f g* ($R2$ ===> $S2$) *g h*

shows $(R1\ OO\ R2 ===> S1\ OO\ S2)\ f\ h$
⟨*proof*⟩

lemma *rel_fun_comp1*:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h\ R' = R1\ OO\ R2$
shows $(R' ===> S1\ OO\ S2)\ f\ h$
⟨*proof*⟩

lemma *rel_fun_comp2*:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h\ S' = S1\ OO\ S2$
shows $(R1\ OO\ R2 ===> S')\ f\ h$
⟨*proof*⟩

lemma *rel_fun_relcompp*:
$((R1 ===> S1)\ OO\ (R2 ===> S2))\ a\ b \Longrightarrow ((R1\ OO\ R2) ===> (S1\ OO\ S2))\ a\ b$
⟨*proof*⟩

lemma *rel_fun_comp1'*:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h \bigwedge a\ b.\ R'\ a\ b \Longrightarrow (R1\ OO\ R2)\ a\ b$
shows $(R' ===> S1\ OO\ S2)\ f\ h$
⟨*proof*⟩

lemma *rel_fun_comp2'*:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h \bigwedge a\ b.\ (S1\ OO\ S2)\ a\ b \Longrightarrow S'\ a\ b$
shows $(R1\ OO\ R2 ===> S')\ f\ h$
⟨*proof*⟩

end
end

1.4 Heap Monad

theory *Heap_Monad_Ext*
imports *HOL−Imperative_HOL.Imperative_HOL*
begin

definition *fun_app_lifted* :: $('a \Rightarrow 'b\ Heap)\ Heap \Rightarrow 'a\ Heap \Rightarrow 'b\ Heap$
where
fun_app_lifted $f_T\ x_T \equiv do\ \{\ f \leftarrow f_T;\ x \leftarrow x_T;\ f\ x\ \}$

bundle *heap_monad_syntax* **begin**

notation *fun_app_lifted* (**infixl** ‹.› 999)
type_synonym ($'a$, $'b$) *fun_lifted* = $'a \Rightarrow 'b$ *Heap* (‹_ ==*H*⟹ _› [3,2] 2)
type_notation *Heap* (‹[_]›)

notation *Heap_Monad.return* (‹⟨_⟩›)
notation (*ASCII*) *Heap_Monad.return* (‹(#_#)›)
notation *Transfer.Rel* (‹*Rel*›)

end

context includes *heap_monad_syntax* **begin**

qualified lemma *return_app_return*:
  $\langle f\rangle$. $\langle x\rangle = f\ x$
  ⟨*proof*⟩ **lemma** *return_app_return_meta*:
  $\langle f\rangle$. $\langle x\rangle \equiv f\ x$
  ⟨*proof*⟩ **definition** if_T :: *bool Heap* ⇒ *'a Heap* ⇒ *'a Heap* ⇒ *'a Heap* **where**
  $if_T\ b_T\ x_T\ y_T \equiv$ *do* $\{b \leftarrow b_T;$ *if b then* x_T *else* $y_T\}$
end

end

1.5 Relation Between the State and the Heap Monad

theory *State_Heap*
  imports
    ../state_monad/DP_CRelVS
    HOL−Imperative_HOL.Imperative_HOL
    State_Heap_Misc
    Heap_Monad_Ext
begin

definition *lift_p* :: (*heap* ⇒ *bool*) ⇒ *'a Heap* ⇒ *bool* **where**
  lift_p P f =
    (∀ *heap*. *P heap* ⟶ (*case execute f heap of None* ⇒ *False* | *Some* (_, *heap*) ⇒ *P heap*))

context
  fixes *P f heap*
  assumes *lift*: *lift_p P f* **and** *P*: *P heap*
begin

lemma *execute_cases*:
  case execute f heap of None $\Rightarrow$ *False* | *Some* (_, *heap*) $\Rightarrow$ *P heap*
  ⟨*proof*⟩

lemma *execute_cases*′:
  case execute f heap of Some (_, *heap*) $\Rightarrow$ *P heap*
  ⟨*proof*⟩

lemma *lift_p_None*[*simp*, *dest*]:
  False **if** *execute f heap* = *None*
  ⟨*proof*⟩

lemma *lift_p_P*:
  case the (*execute f heap*) *of* (_, *heap*) $\Rightarrow$ *P heap*
  ⟨*proof*⟩

lemma *lift_p_P*′:
  P heap′ **if** *the* (*execute f heap*) = (*v*, *heap*′)
  ⟨*proof*⟩

lemma *lift_p_P*′′:
  P heap′ **if** *execute f heap* = *Some* (*v*, *heap*′)
  ⟨*proof*⟩

lemma *lift_p_the_Some*[*simp*]:
  execute f heap = *Some* (*v*, *heap*′) **if** *the* (*execute f heap*) = (*v*, *heap*′)
  ⟨*proof*⟩

lemma *lift_p_E*:
  obtains *v heap*′ **where** *execute f heap* = *Some* (*v*, *heap*′) *P heap*′
  ⟨*proof*⟩

end

definition *state_of s* $\equiv$ *State* (λ *heap*. *the* (*execute s heap*))

locale *heap_mem_defs* =
  fixes *P* :: *heap* $\Rightarrow$ *bool*
    and *lookup* :: ′*k* $\Rightarrow$ ′*v option Heap*
    and *update* :: ′*k* $\Rightarrow$ ′*v* $\Rightarrow$ *unit Heap*
begin

definition *rel_state* :: (′*a* $\Rightarrow$ ′*b* $\Rightarrow$ *bool*) $\Rightarrow$ (*heap*, ′*a*) *state* $\Rightarrow$ ′*b Heap* $\Rightarrow$

bool **where**
  rel_state R f g ≡
    ∀ *heap. P heap* ⟶
      (*case State_Monad.run_state f heap of* (*v*1, *heap*1) ⇒ *case execute g heap of*
        Some (*v*2, *heap*2) ⇒ *R v*1 *v*2 ∧ *heap*1 = *heap*2 ∧ *P heap*2 | *None* ⇒ *False*)

definition *lookup′ k* ≡ *State* (λ *heap. the* (*execute* (*lookup k*) *heap*))

definition *update′ k v* ≡ *State* (λ *heap. the* (*execute* (*update k v*) *heap*))

definition *heap_get* = *Heap_Monad.Heap* (λ *heap. Some* (*heap*, *heap*))

definition *checkmem* :: ′*k* ⇒ ′*v Heap* ⇒ ′*v Heap* **where**
  checkmem param calc ≡
    Heap_Monad.bind (*lookup param*) (λ *x.*
    case x of
      Some x ⇒ *return x*
    | *None* ⇒ *Heap_Monad.bind calc* (λ *x.*
        Heap_Monad.bind (*update param x*) (λ _.
        return x
)
)
)

definition *checkmem′* :: ′*k* ⇒ (*unit* ⇒ ′*v Heap*) ⇒ ′*v Heap* **where**
  checkmem′ param calc ≡
    Heap_Monad.bind (*lookup param*) (λ *x.*
    case x of
      Some x ⇒ *return x*
    | *None* ⇒ *Heap_Monad.bind* (*calc* ()) (λ *x.*
        Heap_Monad.bind (*update param x*) (λ _.
        return x
)
)
)

lemma *checkmem_checkmem′*:
  checkmem′ param (λ_. *calc*) = *checkmem param calc*
  ⟨*proof*⟩

definition *map_of_heap* **where**
map_of_heap heap k = *fst* (*the* (*execute* (*lookup k*) *heap*))

lemma *rel_state_elim*:
assumes *rel_state R f g P heap*
obtains *heap′ v v′* **where**
State_Monad.run_state f heap = (*v*, *heap′*) *execute g heap* = *Some* (*v′*, *heap′*) *R v v′ P heap′*
⟨*proof*⟩

lemma *rel_state_intro*:
assumes
$\bigwedge$ *heap v heap′*. *P heap* $\Longrightarrow$ *State_Monad.run_state f heap* = (*v*, *heap′*)
$\Longrightarrow$ ∃ *v′*. *R v v′* ∧ *execute g heap* = *Some* (*v′*, *heap′*)
$\bigwedge$ *heap v heap′*. *P heap* $\Longrightarrow$ *State_Monad.run_state f heap* = (*v*, *heap′*)
$\Longrightarrow$ *P heap′*
shows *rel_state R f g*
⟨*proof*⟩

context
includes *lifting_syntax* **and** *state_monad_syntax*
begin

lemma *transfer_bind*[*transfer_rule*]:
(*rel_state R* ===> (*R* ===> *rel_state Q*) ===> *rel_state Q*) *State_Monad.bind Heap_Monad.bind*
⟨*proof*⟩

lemma *transfer_return*[*transfer_rule*]:
(*R* ===> *rel_state R*) *State_Monad.return Heap_Monad.return*
⟨*proof*⟩

lemma *fun_app_lifted_transfer*:
(*rel_state* (*R* ===> *rel_state Q*) ===> *rel_state R* ===> *rel_state Q*)
State_Monad_Ext.fun_app_lifted Heap_Monad_Ext.fun_app_lifted
⟨*proof*⟩

lemma *transfer_get*[*transfer_rule*]:
rel_state (=) *State_Monad.get heap_get*
⟨*proof*⟩

end

end

locale *heap_inv* = *heap_mem_defs* _ *lookup* **for** *lookup* :: $'k \Rightarrow 'v$ *option Heap* +
  assumes *lookup_inv*: *lift_p P* (*lookup k*)
  assumes *update_inv*: *lift_p P* (*update k v*)
begin

lemma *rel_state_lookup*:
  rel_state (=) (*lookup′ k*) (*lookup k*)
  ⟨*proof*⟩

lemma *rel_state_update*:
  rel_state (=) (*update′ k v*) (*update k v*)
  ⟨*proof*⟩

context
  includes *lifting_syntax*
begin

lemma *transfer_lookup*:
  ((=) ===> *rel_state* (=)) *lookup′ lookup*
  ⟨*proof*⟩

lemma *transfer_update*:
  ((=) ===> (=) ===> *rel_state* (=)) *update′ update*
  ⟨*proof*⟩

lemma *transfer_checkmem*:
  ((=) ===> *rel_state* (=) ===> *rel_state* (=))
    (*state_mem_defs.checkmem lookup′ update′*) *checkmem*
  ⟨*proof*⟩

end

end

locale *heap_correct* =
  heap_inv +
  assumes *lookup_correct*:
      $P\ m \Longrightarrow$ *map_of_heap* (*snd* (*the* (*execute* (*lookup k*) *m*))) $\subseteq_m$ (*map_of_heap m*)
  and *update_correct*:
      $P\ m \Longrightarrow$ *map_of_heap* (*snd* (*the* (*execute* (*update k v*) *m*))) $\subseteq_m$

(map_of_heap m)($k \mapsto v$)
begin

lemma *lookup′_correct*:
state_mem_defs.map_of lookup′ (*snd* (*State_Monad.run_state* (*lookup′* *k*) *m*)) $\subseteq_m$ (*state_mem_defs.map_of lookup′ m*) **if** *P m*
⟨*proof*⟩

lemma *update′_correct*:
state_mem_defs.map_of lookup′ (*snd* (*State_Monad.run_state* (*update′* *k v*) *m*)) $\subseteq_m$ (*state_mem_defs.map_of lookup′ m*)($k \mapsto v$)
if *P m*
⟨*proof*⟩

lemma *lookup′_inv*:
DP_CRelVS.lift_p P (*lookup′ k*)
⟨*proof*⟩

lemma *update′_inv*:
DP_CRelVS.lift_p P (*update′ k v*)
⟨*proof*⟩

lemma *mem_correct_heap*: *mem_correct lookup′ update′ P*
⟨*proof*⟩

end

context *heap_mem_defs*
begin

context
includes *lifting_syntax*
begin

lemma *mem_correct_heap_correct*:
assumes *correct*: *mem_correct* $lookup_s$ $update_s$ *P*
and *lookup*: ((=) ===> *rel_state* (=)) $lookup_s$ *lookup*
and *update*: ((=) ===> (=) ===> *rel_state* (=)) $update_s$ *update*
shows *heap_correct P update lookup*
⟨*proof*⟩

end

end

end

1.6 Parametricity of the Heap Monad

theory *DP_CRelVH*
  imports *State_Heap*
begin

locale *dp_heap* =
 state_dp_consistency: *dp_consistency lookup_st update_st P dp* + *heap_mem_defs Q lookup update*
  for $P\ Q :: heap \Rightarrow bool$ **and** $dp :: {}'k \Rightarrow {}'v$ **and** $lookup :: {}'k \Rightarrow {}'v\ option\ Heap$
  and *lookup_st update update_st* +
  assumes
    rel_state_lookup: *rel_fun* (=) (*rel_state* (=)) *lookup_st lookup*
      and
    rel_state_update: *rel_fun* (=) (*rel_fun* (=) (*rel_state* (=))) *update_st update*
begin

context
  includes *lifting_syntax* **and** *heap_monad_syntax*
begin

definition *crel_vs R v f* ≡
  ∀ *heap*. *P heap* ∧ *Q heap* ∧ *state_dp_consistency.cmem heap* ⟶
    (*case execute f heap of*
      None ⇒ *False* |
     Some (*v′*, *heap′*) ⇒ *P heap′* ∧ *Q heap′* ∧ *R v v′* ∧ *state_dp_consistency.cmem heap′*
)

abbreviation *rel_fun_lifted* :: $({}'a \Rightarrow {}'c \Rightarrow bool) \Rightarrow ({}'b \Rightarrow {}'d \Rightarrow bool) \Rightarrow ({}'a \Rightarrow {}'b) \Rightarrow ({}'c \mathrel{==H\Longrightarrow} {}'d) \Rightarrow bool$ (**infixr** ‹===>$_T$› 55) **where**
  rel_fun_lifted R R′ ≡ *R* ===> *crel_vs R′*

definition *consistentDP* :: $({}'k \Rightarrow {}'v\ Heap) \Rightarrow bool$ **where**
  consistentDP ≡ ((=) ===> *crel_vs* (=)) *dp*

lemma *consistentDP_intro*:

assumes $\bigwedge param.\ Transfer.Rel\ (crel\_vs\ (=))\ (dp\ param)\ (dp_T\ param)$
shows $consistentDP\ dp_T$
$\langle proof \rangle$

lemma *crel_vs_execute_None*:
$False$ **if** $crel\_vs\ R\ a\ b\ execute\ b\ heap = None\ P\ heap\ Q\ heap\ state\_dp\_consistency.cmem\ heap$
$\langle proof \rangle$

lemma *crel_vs_execute_Some*:
assumes $crel\_vs\ R\ a\ b\ P\ heap\ Q\ heap\ state\_dp\_consistency.cmem\ heap$
obtains $x\ heap'$ **where** $execute\ b\ heap = Some\ (x,\ heap')\ P\ heap'\ Q\ heap'$
$\langle proof \rangle$

lemma *crel_vs_executeD*:
assumes $crel\_vs\ R\ a\ b\ P\ heap\ Q\ heap\ state\_dp\_consistency.cmem\ heap$
obtains $x\ heap'$ **where**
$execute\ b\ heap = Some\ (x,\ heap')\ P\ heap'\ Q\ heap'\ state\_dp\_consistency.cmem\ heap'\ R\ a\ x$
$\langle proof \rangle$

lemma *crel_vs_success*:
assumes $crel\_vs\ R\ a\ b\ P\ heap\ Q\ heap\ state\_dp\_consistency.cmem\ heap$
shows $success\ b\ heap$
$\langle proof \rangle$

lemma *crel_vsI*: $crel\_vs\ R\ a\ b$ **if** $(state\_dp\_consistency.crel\_vs\ R\ OO\ rel\_state\ (=))\ a\ b$
$\langle proof \rangle$

lemma *transfer'_return*[*transfer_rule*]:
$(R ===> crel\_vs\ R)\ Wrap\ return$
$\langle proof \rangle$

lemma *crel_vs_return*:
$Transfer.Rel\ (crel\_vs\ R)\ (Wrap\ x)\ (return\ y)$ **if** $Transfer.Rel\ R\ x\ y$
$\langle proof \rangle$

lemma *crel_vs_return_ext*:
$[\![Transfer.Rel\ R\ x\ y]\!] \Longrightarrow Transfer.Rel\ (crel\_vs\ R)\ x\ (Heap\_Monad.return\ y)$
$\langle proof \rangle$
term 0

lemma *bind_transfer*[*transfer_rule*]:
  (*crel_vs R*0 ===> (*R*0 ===> *crel_vs R*1) ===> *crel_vs R*1) ($\lambda v\, f.\ f$ v) ($\gg=$)
  ⟨*proof*⟩

lemma *crel_vs_update*:
  crel_vs (=) () (*update param* (*dp param*))
  ⟨*proof*⟩

lemma *crel_vs_lookup*:
  crel_vs
    (λ v v'. *case* v' *of None* $\Rightarrow$ *True* | *Some* v' $\Rightarrow$ $v = v' \wedge v =$ *dp param*) (*dp param*) (*lookup param*)
  ⟨*proof*⟩

lemma *crel_vs_eq_eq_onp*:
  crel_vs (*eq_onp* (λ x. $x = v$)) v s **if** *crel_vs* (=) v s
  ⟨*proof*⟩

lemma *crel_vs_bind_eq*:
  ⟦*crel_vs* (=) v s; *crel_vs R* (f v) (*sf* v)⟧ $\Longrightarrow$ *crel_vs R* (f v) ($s \gg=$ *sf*)
  ⟨*proof*⟩

lemma *crel_vs_checkmem*:
  Transfer.Rel (*crel_vs R*) (*dp param*) (*checkmem param s*) **if** *is_equality* *R Transfer.Rel* (*crel_vs R*) (*dp param*) *s*
  ⟨*proof*⟩

lemma *crel_vs_checkmem_tupled*:
  assumes $v =$ *dp param*
  shows ⟦*is_equality R*; *Transfer.Rel* (*crel_vs R*) v s⟧
      $\Longrightarrow$ *Transfer.Rel* (*crel_vs R*) v (*checkmem param s*)
  ⟨*proof*⟩

lemma *transfer_fun_app_lifted*[*transfer_rule*]:
  (*crel_vs* (*R*0 ===> *crel_vs R*1) ===> *crel_vs R*0 ===> *crel_vs R*1)
    App Heap_Monad_Ext.fun_app_lifted
  ⟨*proof*⟩

lemma *crel_vs_fun_app*:
  ⟦*Transfer.Rel* (*crel_vs R*0) x x_T; *Transfer.Rel* (*crel_vs* (*R*0 $===>_T$ *R*1)) f f_T⟧ $\Longrightarrow$ *Transfer.Rel* (*crel_vs R*1) (*App f x*) (f_T . x_T)
  ⟨*proof*⟩

end

end

locale *dp_consistency_heap* = *heap_correct* +
  fixes *dp* :: $'a \Rightarrow 'b$
begin

interpretation *state_mem_correct*: *mem_correct lookup′ update′ P*
  ⟨*proof*⟩

interpretation *state_dp_consistency*: *dp_consistency lookup′ update′ P dp*
⟨*proof*⟩

lemma *dp_heap*: *dp_heap P P lookup lookup′ update update′*
  ⟨*proof*⟩

sublocale *dp_heap P P dp lookup lookup′ update update′*
  ⟨*proof*⟩

notation *rel_fun_lifted* (**infixr** ‹===>$_T$› 55)
end

locale *heap_correct_empty* = *heap_correct* +
  fixes *empty*
  assumes *empty_correct*: *map_of_heap empty* $\subseteq_m$ *Map.empty* **and** *P_empty*: *P empty*

locale *dp_consistency_heap_empty* =
  dp_consistency_heap + *heap_correct_empty*
begin

lemma *cmem_empty*:
  state_dp_consistency.cmem empty
  ⟨*proof*⟩

corollary *memoization_correct*:
  dp x = *v state_dp_consistency.cmem m* **if**
  consistentDP dp_T *Heap_Monad.execute* (dp_T *x*) *empty* = *Some* (*v*, *m*)
  ⟨*proof*⟩

lemma *memoized_success*:
  success (dp_T *x*) *empty* **if** *consistentDP* dp_T

$\langle proof \rangle$

lemma *memoized*:
$dp\ x = fst\ (the\ (Heap\_Monad.execute\ (dp_T\ x)\ empty))$ **if** $consistentDP\ dp_T$
$\langle proof \rangle$

lemma *cmem_result*:
$state\_dp\_consistency.cmem\ (snd\ (the\ (Heap\_Monad.execute\ (dp_T\ x)\ empty)))$ **if** $consistentDP\ dp_T$
$\langle proof \rangle$

end

end

2 Memoization

2.1 Memory Implementations for the State Monad

theory *Memory*
imports *DP_CRelVS HOL−Library.Mapping*
begin

lemma *lift_pI*[*intro?*]:
$lift\_p\ P\ f$ **if** $\bigwedge heap\ x\ heap'.\ P\ heap \Longrightarrow run\_state\ f\ heap = (x,\ heap') \Longrightarrow P\ heap'$
$\langle proof \rangle$

lemma *mem_correct_default*:
$mem\_correct$
$(\lambda\ k.\ do\ \{m \leftarrow State\_Monad.get;\ State\_Monad.return\ (m\ k)\})$
$(\lambda\ k\ v.\ do\ \{m \leftarrow State\_Monad.get;\ State\_Monad.set\ (m(k \mapsto v))\})$
$(\lambda\ \_.\ True)$
$\langle proof \rangle$

lemma *mem_correct_rbt_mapping*:
$mem\_correct$
$(\lambda\ k.\ do\ \{m \leftarrow State\_Monad.get;\ State\_Monad.return\ (Mapping.lookup\ m\ k)\})$
$(\lambda\ k\ v.\ do\ \{m \leftarrow State\_Monad.get;\ State\_Monad.set\ (Mapping.update\ k\ v\ m)\})$
$(\lambda\ \_.\ True)$

⟨*proof*⟩

locale *mem_correct_empty* = *mem_correct* +
fixes *empty*
assumes *empty_correct*: *map_of empty* $\subseteq_m$ *Map.empty* **and** *P_empty*: *P empty*

lemma (**in** *mem_correct_empty*) *dom_empty*[*simp*]:
dom (*map_of empty*) = {}
⟨*proof*⟩

lemma *mem_correct_empty_default*:
mem_correct_empty
(λ *k*. *do* {*m* ← *State_Monad.get*; *State_Monad.return* (*m k*)})
(λ *k v*. *do* {*m* ← *State_Monad.get*; *State_Monad.set* (*m*(*k*↦*v*))})
(λ _. *True*)
Map.empty
⟨*proof*⟩

lemma *mem_correct_rbt_empty_mapping*:
mem_correct_empty
(λ *k*. *do* {*m* ← *State_Monad.get*; *State_Monad.return* (*Mapping.lookup m k*)})
(λ *k v*. *do* {*m* ← *State_Monad.get*; *State_Monad.set* (*Mapping.update k v m*)})
(λ _. *True*)
Mapping.empty
⟨*proof*⟩

locale *dp_consistency_empty* =
dp_consistency + *mem_correct_empty*
begin

lemma *cmem_empty*:
cmem empty
⟨*proof*⟩

corollary *memoization_correct*:
dp x = *v cmem m* **if** *consistentDP* dp_T *State_Monad.run_state* (dp_T *x*) *empty* = (*v*, *m*)
⟨*proof*⟩

lemma *memoized*:
$dp\ x = fst\ (State\_Monad.run\_state\ (dp_T\ x)\ empty)$ **if** $consistentDP\ dp_T$
⟨*proof*⟩

lemma *cmem_result*:
$cmem\ (snd\ (State\_Monad.run\_state\ (dp_T\ x)\ empty))$ **if** $consistentDP\ dp_T$
⟨*proof*⟩

end

locale *dp_consistency_default* =
fixes $dp :: 'param \Rightarrow 'result$
begin

sublocale *dp_consistency_empty*
$\lambda\ k.\ do\ \{(m::'param \rightharpoonup 'result) \leftarrow State\_Monad.get;\ State\_Monad.return\ (m\ k)\}$
$\lambda\ k\ v.\ do\ \{m \leftarrow State\_Monad.get;\ State\_Monad.set\ (m(k \mapsto v))\}$
$\lambda\ (\_::'param \rightharpoonup 'result).\ True$
dp
Map.empty
⟨*proof*⟩

end

locale *dp_consistency_mapping* =
fixes $dp :: 'param \Rightarrow 'result$
begin

sublocale *dp_consistency_empty*
$(\lambda\ k.\ do\ \{(m::('param,'result)\ mapping) \leftarrow State\_Monad.get;\ State\_Monad.return\ (Mapping.lookup\ m\ k)\})$
$(\lambda\ k\ v.\ do\ \{m \leftarrow State\_Monad.get;\ State\_Monad.set\ (Mapping.update\ k\ v\ m)\})$
$(\lambda\ \_::('param,'result)\ mapping.\ True)\ dp\ Mapping.empty$
⟨*proof*⟩

end

2.1.1 Tracing Memory

context *state_mem_defs*
begin

definition
lookup_trace k =
State (λ (*log*, *m*). *case State_Monad.run_state* (*lookup k*) *m of*
(*None*, *m*) ⇒ (*None*, ((''*Missed*'', *k*) # *log*, *m*)) |
(*Some v*, *m*) ⇒ (*Some v*, ((''*Found*'', *k*) # *log*, *m*))
)

definition
update_trace k v =
State (λ (*log*, *m*). *case State_Monad.run_state* (*update k v*) *m of*
(_, *m*) ⇒ ((), ((''*Stored*'', *k*) # *log*, *m*))
)

end

context *mem_correct*
begin

lemma *map_of_simp*:
state_mem_defs.map_of lookup_trace = *map_of o snd*
⟨*proof*⟩

lemma *mem_correct_tracing*: *mem_correct lookup_trace update_trace* (*P o snd*)
⟨*proof*⟩

end

context *mem_correct_empty*
begin

lemma *mem_correct_tracing_empty*:
mem_correct_empty lookup_trace update_trace (*P o snd*) ([], *empty*)
⟨*proof*⟩

end

locale *dp_consistency_mapping_tracing* =
fixes *dp* :: '*param* ⇒ '*result*
begin

interpretation *mapping*: *dp_consistency_mapping* ⟨*proof*⟩

sublocale *dp_consistency_empty*

mapping.lookup_trace mapping.update_trace (λ _. *True*) *o snd dp* ([], *Mapping.empty*)
⟨*proof*⟩

end

end

2.2 Pair Memory

theory *Pair_Memory*
imports ../*state_monad*/*Memory*
begin

lemma *map_add_mono*:
$(m1 \mathbin{++} m2) \subseteq_m (m1' \mathbin{++} m2')$ **if** $m1 \subseteq_m m1'$ $m2 \subseteq_m m2'$ $dom\ m1 \cap dom\ m2' = \{\}$
⟨*proof*⟩

lemma *map_add_upd2*:
$f(x \mapsto y) \mathbin{++} g = (f \mathbin{++} g)(x \mapsto y)$ **if** $dom\ f \cap dom\ g = \{\}$ $x \notin dom\ g$
⟨*proof*⟩

locale *pair_mem_defs* =
fixes *lookup*1 *lookup*2 :: ′*a* ⇒ (′*mem*, ′*v option*) *state*
and *update*1 *update*2 :: ′*a* ⇒ ′*v* ⇒ (′*mem*, *unit*) *state*
and *move*12 :: ′*k*1 ⇒ (′*mem*, *unit*) *state*
and *get_k*1 *get_k*2 :: (′*mem*, ′*k*1) *state*
and *P* :: ′*mem* ⇒ *bool*
fixes *key*1 :: ′*k* ⇒ ′*k*1 **and** *key*2 :: ′*k* ⇒ ′*a*
begin

We assume that look-ups happen on the older row, so it is biased towards the second entry.

definition
lookup_pair k = *do* {
let k′ = *key*1 *k*;
*k*2 ← *get_k*2;
if k′ = *k*2
*then lookup*2 (*key*2 *k*)
else do {
*k*1 ← *get_k*1;
if k′ = *k*1

$then\ lookup1\ (key2\ k)$
$else\ State\_Monad.return\ None$
$\}$
$\}$

We assume that updates happen on the newer row, so it is biased towards the first entry.

definition
$update\_pair\ k\ v = do\ \{$
$let\ k' = key1\ k;$
$k1 \leftarrow get\_k1;$
$if\ k' = k1$
$then\ update1\ (key2\ k)\ v$
$else\ do\ \{$
$k2 \leftarrow get\_k2;$
$if\ k' = k2$
$then\ update2\ (key2\ k)\ v$
$else\ (move12\ k' \gg update1\ (key2\ k)\ v)$
$\}$
$\}$

sublocale $pair$: $state\_mem\_defs\ lookup\_pair\ update\_pair\ \langle proof \rangle$

sublocale $mem1$: $state\_mem\_defs\ lookup1\ update1\ \langle proof \rangle$

sublocale $mem2$: $state\_mem\_defs\ lookup2\ update2\ \langle proof \rangle$

definition
$inv\_pair\ heap \equiv$
let
$k1 = fst\ (State\_Monad.run\_state\ get\_k1\ heap);$
$k2 = fst\ (State\_Monad.run\_state\ get\_k2\ heap)$
in
$(\forall\ k \in dom\ (mem1.map\_of\ heap).\ \exists\ k'.\ key1\ k' = k1 \wedge key2\ k' = k) \wedge$
$(\forall\ k \in dom\ (mem2.map\_of\ heap).\ \exists\ k'.\ key1\ k' = k2 \wedge key2\ k' = k) \wedge$
$k1 \neq k2 \wedge P\ heap$

definition
$map\_of1\ m\ k = (if\ key1\ k = fst\ (State\_Monad.run\_state\ get\_k1\ m)\ then\ mem1.map\_of\ m\ (key2\ k)\ else\ None)$

definition
$map\_of2\ m\ k = (if\ key1\ k = fst\ (State\_Monad.run\_state\ get\_k2\ m)\ then\ mem2.map\_of\ m\ (key2\ k)\ else\ None)$

end

locale $pair\_mem = pair\_mem\_defs\ +$
assumes get_state:
$State\_Monad.run\_state\ get\_k1\ m = (k, m') \Longrightarrow m' = m$
$State\_Monad.run\_state\ get\_k2\ m = (k, m') \Longrightarrow m' = m$
assumes $move12\_correct$:
$P\ m \Longrightarrow State\_Monad.run\_state\ (move12\ k1)\ m = (x, m') \Longrightarrow mem1.map\_of\ m' \subseteq_m Map.empty$
$P\ m \Longrightarrow State\_Monad.run\_state\ (move12\ k1)\ m = (x, m') \Longrightarrow mem2.map\_of\ m' \subseteq_m mem1.map\_of\ m$
assumes $move12\_keys$:
$State\_Monad.run\_state\ (move12\ k1)\ m = (x, m') \Longrightarrow fst\ (State\_Monad.run\_state\ get\_k1\ m') = k1$
$State\_Monad.run\_state\ (move12\ k1)\ m = (x, m') \Longrightarrow fst\ (State\_Monad.run\_state\ get\_k2\ m') = fst\ (State\_Monad.run\_state\ get\_k1\ m)$
assumes $move12\_inv$:
$lift\_p\ P\ (move12\ k1)$
assumes $lookup\_inv$:
$lift\_p\ P\ (lookup1\ k')\ lift\_p\ P\ (lookup2\ k')$
assumes $update\_inv$:
$lift\_p\ P\ (update1\ k'\ v)\ lift\_p\ P\ (update2\ k'\ v)$
assumes $lookup\_keys$:
$P\ m \Longrightarrow State\_Monad.run\_state\ (lookup1\ k')\ m = (v', m') \Longrightarrow$
$fst\ (State\_Monad.run\_state\ get\_k1\ m') = fst\ (State\_Monad.run\_state\ get\_k1\ m)$
$P\ m \Longrightarrow State\_Monad.run\_state\ (lookup1\ k')\ m = (v', m') \Longrightarrow$
$fst\ (State\_Monad.run\_state\ get\_k2\ m') = fst\ (State\_Monad.run\_state\ get\_k2\ m)$
$P\ m \Longrightarrow State\_Monad.run\_state\ (lookup2\ k')\ m = (v', m') \Longrightarrow$
$fst\ (State\_Monad.run\_state\ get\_k1\ m') = fst\ (State\_Monad.run\_state\ get\_k1\ m)$
$P\ m \Longrightarrow State\_Monad.run\_state\ (lookup2\ k')\ m = (v', m') \Longrightarrow$
$fst\ (State\_Monad.run\_state\ get\_k2\ m') = fst\ (State\_Monad.run\_state\ get\_k2\ m)$
assumes $update\_keys$:
$P\ m \Longrightarrow State\_Monad.run\_state\ (update1\ k'\ v)\ m = (x, m') \Longrightarrow$
$fst\ (State\_Monad.run\_state\ get\_k1\ m') = fst\ (State\_Monad.run\_state\ get\_k1\ m)$
$P\ m \Longrightarrow State\_Monad.run\_state\ (update1\ k'\ v)\ m = (x, m') \Longrightarrow$

fst ($State\_Monad.run\_state$ get_k2 m') = fst ($State\_Monad.run\_state$ get_k2 m)
$P\ m \Longrightarrow State\_Monad.run\_state\ (update2\ k'\ v)\ m = (x,\ m') \Longrightarrow$
fst ($State\_Monad.run\_state$ get_k1 m') = fst ($State\_Monad.run\_state$ get_k1 m)
$P\ m \Longrightarrow State\_Monad.run\_state\ (update2\ k'\ v)\ m = (x,\ m') \Longrightarrow$
fst ($State\_Monad.run\_state$ get_k2 m') = fst ($State\_Monad.run\_state$ get_k2 m)

assumes

lookup_correct:
$P\ m \Longrightarrow mem1.map\_of\ (snd\ (State\_Monad.run\_state\ (lookup1\ k')\ m)) \subseteq_m (mem1.map\_of\ m)$
$P\ m \Longrightarrow mem2.map\_of\ (snd\ (State\_Monad.run\_state\ (lookup1\ k')\ m)) \subseteq_m (mem2.map\_of\ m)$
$P\ m \Longrightarrow mem1.map\_of\ (snd\ (State\_Monad.run\_state\ (lookup2\ k')\ m)) \subseteq_m (mem1.map\_of\ m)$
$P\ m \Longrightarrow mem2.map\_of\ (snd\ (State\_Monad.run\_state\ (lookup2\ k')\ m)) \subseteq_m (mem2.map\_of\ m)$

assumes

update_correct:
$P\ m \Longrightarrow mem1.map\_of\ (snd\ (State\_Monad.run\_state\ (update1\ k'\ v)\ m)) \subseteq_m (mem1.map\_of\ m)(k' \mapsto v)$
$P\ m \Longrightarrow mem2.map\_of\ (snd\ (State\_Monad.run\_state\ (update2\ k'\ v)\ m)) \subseteq_m (mem2.map\_of\ m)(k' \mapsto v)$
$P\ m \Longrightarrow mem2.map\_of\ (snd\ (State\_Monad.run\_state\ (update1\ k'\ v)\ m)) \subseteq_m mem2.map\_of\ m$
$P\ m \Longrightarrow mem1.map\_of\ (snd\ (State\_Monad.run\_state\ (update2\ k'\ v)\ m)) \subseteq_m mem1.map\_of\ m$

begin

lemma *map_of_le_pair*:
$pair.map\_of\ m \subseteq_m map\_of1\ m\ {+}{+}\ map\_of2\ m$
if $inv\_pair\ m$
⟨*proof*⟩

lemma *pair_le_map_of*:
$map\_of1\ m\ {+}{+}\ map\_of2\ m \subseteq_m pair.map\_of\ m$
if $inv\_pair\ m$
⟨*proof*⟩

lemma *map_of_eq_pair*:
$map\_of1\ m\ {+}{+}\ map\_of2\ m = pair.map\_of\ m$
if $inv\_pair\ m$
⟨*proof*⟩

lemma $inv\_pair\_neq[simp]$:
$False$ **if** $inv\_pair\ m\ fst\ (State\_Monad.run\_state\ get\_k1\ m) = fst\ (State\_Monad.run\_state\ get\_k2\ m)$
$\langle proof \rangle$

lemma $inv\_pair\_P\_D$:
$P\ m$ **if** $inv\_pair\ m$
$\langle proof \rangle$

lemma $inv\_pair\_domD[intro]$:
$dom\ (map\_of1\ m) \cap dom\ (map\_of2\ m) = \{\}$ **if** $inv\_pair\ m$
$\langle proof \rangle$

lemma $move12\_correct1$:
$map\_of1\ heap' \subseteq_m Map.empty$ **if** $State\_Monad.run\_state\ (move12\ k1)\ heap = (x,\ heap')\ P\ heap$
$\langle proof \rangle$

lemma $move12\_correct2$:
$map\_of2\ heap' \subseteq_m map\_of1\ heap$ **if** $State\_Monad.run\_state\ (move12\ k1)\ heap = (x,\ heap')\ P\ heap$
$\langle proof \rangle$

lemma $dom\_empty[simp]$:
$dom\ (map\_of1\ heap') = \{\}$ **if** $State\_Monad.run\_state\ (move12\ k1)\ heap = (x,\ heap')\ P\ heap$
$\langle proof \rangle$

lemma $inv\_pair\_lookup1$:
$inv\_pair\ m'$ **if** $State\_Monad.run\_state\ (lookup1\ k)\ m = (v,\ m')\ inv\_pair\ m$
$\langle proof \rangle$

lemma $inv\_pair\_lookup2$:
$inv\_pair\ m'$ **if** $State\_Monad.run\_state\ (lookup2\ k)\ m = (v,\ m')\ inv\_pair\ m$
$\langle proof \rangle$

lemma $inv\_pair\_update1$:
$inv\_pair\ m'$
if $State\_Monad.run\_state\ (update1\ (key2\ k)\ v)\ m = (v',\ m')\ inv\_pair\ m\ fst\ (State\_Monad.run\_state\ get\_k1\ m) = key1\ k$
$\langle proof \rangle$

lemma *inv_pair_update2*:
inv_pair m'
if *State_Monad.run_state* (*update2* (*key2* k) v) $m = (v', m')$ *inv_pair* m
fst (*State_Monad.run_state* *get_k2* m) = *key1* k
⟨*proof*⟩

lemma *inv_pair_move12*:
inv_pair m'
if *State_Monad.run_state* (*move12* k) $m = (v', m')$ *inv_pair* m *fst* (*State_Monad.run_state* *get_k1* m) $\neq k$
⟨*proof*⟩

lemma *mem_correct_pair*:
mem_correct lookup_pair update_pair inv_pair
if *injective*: $\forall\ k\ k'.\ key1\ k = key1\ k' \land key2\ k = key2\ k' \longrightarrow k = k'$
⟨*proof*⟩

lemma *emptyI*:
assumes *inv_pair* m *mem1.map_of* $m \subseteq_m$ *Map.empty* *mem2.map_of* m $\subseteq_m$ *Map.empty*
shows *pair.map_of* $m \subseteq_m$ *Map.empty*
⟨*proof*⟩

end

datatype $('k, 'v)$ *pair_storage* = *Pair_Storage* $'k\ 'k\ 'v\ 'v$

context *mem_correct_empty*
begin

context
fixes *key* :: $'a \Rightarrow 'k$
begin

We assume that look-ups happen on the older row, so it is biased towards the second entry.

definition
lookup_pair k =
State (λ *mem*.
(
case mem of Pair_Storage $k1\ k2\ m1\ m2 \Rightarrow$ *let* $k' = key\ k$ *in*
if $k' = k2$ *then case State_Monad.run_state* (*lookup* k) $m2$ *of* (v, m)

$\Rightarrow$ $(v,\ Pair\_Storage\ k1\ k2\ m1\ m)$
  $else\ if\ k' = k1\ then\ case\ State\_Monad.run\_state\ (lookup\ k)\ m1\ of$ $(v,\ m) \Rightarrow (v,\ Pair\_Storage\ k1\ k2\ m\ m2)$
  $else\ (None,\ mem)$
)
)

We assume that updates happen on the newer row, so it is biased towards the first entry.

definition
$update\_pair\ k\ v =$
$State\ (\lambda\ mem.$
(
$case\ mem\ of\ Pair\_Storage\ k1\ k2\ m1\ m2 \Rightarrow let\ k' = key\ k\ in$
$if\ k' = k1\ then\ case\ State\_Monad.run\_state\ (update\ k\ v)\ m1\ of\ (\_,$ $m) \Rightarrow ((),\ Pair\_Storage\ k1\ k2\ m\ m2)$
$else\ if\ k' = k2\ then\ case\ State\_Monad.run\_state\ (update\ k\ v)\ m2\ of$ $(\_,\ m) \Rightarrow ((),Pair\_Storage\ k1\ k2\ m1\ m)$
$else\ case\ State\_Monad.run\_state\ (update\ k\ v)\ empty\ of\ (\_,\ m) \Rightarrow$ $((),\ Pair\_Storage\ k'\ k1\ m\ m1)$
)
)

interpretation $pair$: $state\_mem\_defs\ lookup\_pair\ update\_pair$ $\langle proof \rangle$

definition
$inv\_pair\ p = (case\ p\ of\ Pair\_Storage\ k1\ k2\ m1\ m2 \Rightarrow$
$key\ \text{‘}\ dom\ (map\_of\ m1) \subseteq \{k1\} \wedge key\ \text{‘}\ dom\ (map\_of\ m2) \subseteq \{k2\} \wedge k1$ $\neq k2 \wedge P\ m1 \wedge P\ m2$
)

lemma $map\_of\_le\_pair$:
$pair.map\_of\ (Pair\_Storage\ k1\ k2\ m1\ m2) \subseteq_m (map\_of\ m1\ {+}{+}\ map\_of$ $m2)$
if $inv\_pair\ (Pair\_Storage\ k1\ k2\ m1\ m2)$
$\langle proof \rangle$

lemma $pair\_le\_map\_of$:
$map\_of\ m1\ {+}{+}\ map\_of\ m2 \subseteq_m pair.map\_of\ (Pair\_Storage\ k1\ k2\ m1$ $m2)$
if $inv\_pair\ (Pair\_Storage\ k1\ k2\ m1\ m2)$
$\langle proof \rangle$

lemma *map_of_eq_pair*:
map_of $m1$ ++ *map_of* $m2$ = *pair.map_of* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
if *inv_pair* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
⟨*proof*⟩

lemma *inv_pair_neq*[*simp*, *dest*]:
False **if** *inv_pair* (*Pair_Storage* k k x y)
⟨*proof*⟩

lemma *inv_pair_P_D1*:
P $m1$ **if** *inv_pair* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
⟨*proof*⟩

lemma *inv_pair_P_D2*:
P $m2$ **if** *inv_pair* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
⟨*proof*⟩

lemma *inv_pair_domD*[*intro*]:
dom (*map_of* $m1$) ∩ *dom* (*map_of* $m2$) = {} **if** *inv_pair* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
⟨*proof*⟩

lemma *mem_correct_pair*:
mem_correct lookup_pair update_pair inv_pair
⟨*proof*⟩

end

end

end

2.3 Indexing

theory *Indexing*
imports *Main*
begin

definition *injective* :: *nat* ⇒ ('*k* ⇒ *nat*) ⇒ *bool* **where**
injective size to_index ≡ ∀ *a b*.
to_index a = *to_index b*
∧ *to_index a* < *size*
∧ *to_index b* < *size*

$\longrightarrow a = b$
for *size to_index*

lemma *index_mono*:
fixes $a\ b\ a0\ b0 :: nat$
assumes a: $a < a0$ **and** b: $b < b0$
shows $a * b0 + b < a0 * b0$
⟨*proof*⟩

lemma *index_eq_iff*:
fixes $a\ b\ c\ d\ b0 :: nat$
assumes $b < b0\ \ d < b0\ \ a * b0 + b = c * b0 + d$
shows $a = c \wedge b = d$
⟨*proof*⟩

locale *prod_order_def* =
order0: *ord less_eq0 less0* +
order1: *ord less_eq1 less1*
for *less_eq0 less0 less_eq1 less1*
begin

fun *less* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool$ **where**
less (a,b) (c,d) $\longleftrightarrow$ *less0* a c $\wedge$ *less1* b d

fun *less_eq* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool$ **where**
less_eq ab cd $\longleftrightarrow$ *less ab cd* $\vee$ $ab = cd$

end

locale *prod_order* =
prod_order_def less_eq0 less0 less_eq1 less1 +
order0: *order less_eq0 less0* +
order1: *order less_eq1 less1*
for *less_eq0 less0 less_eq1 less1*
begin

sublocale *order less_eq less*
⟨*proof*⟩

end

locale *option_order* =
order0: *order less_eq0 less0*

for *less_eq0 less0*
begin

fun *less_eq_option* :: $'a$ *option* $\Rightarrow$ $'a$ *option* $\Rightarrow$ *bool* **where**
less_eq_option None _ $\longleftrightarrow$ *True*
| *less_eq_option* (*Some _*) *None* $\longleftrightarrow$ *False*
| *less_eq_option* (*Some a*) (*Some b*) $\longleftrightarrow$ *less_eq0 a b*

fun *less_option* :: $'a$ *option* $\Rightarrow$ $'a$ *option* $\Rightarrow$ *bool* **where**
less_option ao bo $\longleftrightarrow$ *less_eq_option ao bo* $\wedge$ *ao* $\neq$ *bo*

sublocale *order less_eq_option less_option*
$\langle proof \rangle$

end

datatype $'a$ *bound* = *Bound* (*lower*: $'a$) (*upper*:$'a$)

definition *in_bound* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *bound* $\Rightarrow$ $'a \Rightarrow$ *bool* **where**
in_bound less_eq less bound x $\equiv$ *case bound of Bound l r* $\Rightarrow$ *less_eq l x* $\wedge$ *less x r* **for** *less_eq less*

locale *index_locale_def* = *ord less_eq less* **for** *less_eq less* :: $'a \Rightarrow 'a \Rightarrow$ *bool* +
fixes *idx* :: $'a$ *bound* $\Rightarrow$ $'a \Rightarrow$ *nat*
and *size* :: $'a$ *bound* $\Rightarrow$ *nat*

locale *index_locale* = *index_locale_def* + *idx_ord*: *order* +
assumes *idx_valid*: *in_bound less_eq less bound x* $\Longrightarrow$ *idx bound x* < *size bound*
and *idx_inj* : $[\![$*in_bound less_eq less bound x*; *in_bound less_eq less bound y*; *idx bound x* = *idx bound y*$]\!]$ $\Longrightarrow$ *x* = *y*

locale *prod_index_def* =
index0: *index_locale_def less_eq0 less0 idx0 size0* +
index1: *index_locale_def less_eq1 less1 idx1 size1*
for *less_eq0 less0 idx0 size0 less_eq1 less1 idx1 size1*
begin

fun *idx* :: ($'a \times 'b$) *bound* $\Rightarrow$ $'a \times 'b \Rightarrow$ *nat* **where**
idx (*Bound* (*l0*, *r0*) (*l1*, *r1*)) (*a*, *b*) = (*idx0* (*Bound l0 l1*) *a*) $*$ (*size1* (*Bound r0 r1*)) + *idx1* (*Bound r0 r1*) *b*

fun *size* :: $('a \times 'b)$ *bound* $\Rightarrow$ *nat* **where**
size (*Bound* (*l*0, *r*0) (*l*1, *r*1)) = *size*0 (*Bound* *l*0 *l*1) $*$ *size*1 (*Bound* *r*0 *r*1)

end

locale *prod_index* = *prod_index_def* *less_eq*0 *less*0 *idx*0 *size*0 *less_eq*1 *less*1 *idx*1 *size*1 +
*index*0: *index_locale* *less_eq*0 *less*0 *idx*0 *size*0 +
*index*1: *index_locale* *less_eq*1 *less*1 *idx*1 *size*1
for *less_eq*0 *less*0 *idx*0 *size*0 *less_eq*1 *less*1 *idx*1 *size*1
begin

sublocale *prod_order* *less_eq*0 *less*0 *less_eq*1 *less*1 ⟨*proof*⟩

sublocale *index_locale* *less_eq* *less* *idx* *size* ⟨*proof*⟩
end

locale *option_index* =
*index*0: *index_locale* *less_eq*0 *less*0 *idx*0 *size*0
for *less_eq*0 *less*0 *idx*0 *size*0
begin

fun *idx* :: *'a option bound* $\Rightarrow$ *'a option* $\Rightarrow$ *nat* **where**
idx (*Bound* (*Some* *l*) (*Some* *r*)) (*Some* *a*) = *idx*0 (*Bound* *l* *r*) *a*
| *idx* _ _ = *undefined*

end

locale *nat_index_def* = *ord* $(\le)$ :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* $(<)$
begin

fun *idx* :: *nat bound* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
idx (*Bound* *l* _) *i* = $i - l$

fun *size* :: *nat bound* $\Rightarrow$ *nat* **where**
size (*Bound* *l* *r*) = $r - l$

sublocale *index_locale* $(\le)$ $(<)$ *idx* *size*
⟨*proof*⟩

end

locale *nat_index* = *nat_index_def* + *order* $(\le)$ :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* $(<)$

locale *int_index_def* = *ord* (≤) :: *int* ⇒ *int* ⇒ *bool* (<)
begin

fun *idx* :: *int bound* ⇒ *int* ⇒ *nat* **where**
  idx (*Bound l* _) *i* = *nat* (*i* − *l*)

fun *size* :: *int bound* ⇒ *nat* **where**
  size (*Bound l r*) = *nat* (*r* − *l*)

sublocale *index_locale* (≤) (<) *idx size*
⟨*proof*⟩

end

locale *int_index* = *int_index_def* + *order* (≤) :: *int* ⇒ *int* ⇒ *bool* (<)

class *index* =
  fixes *less_eq less* :: ′*a* ⇒ ′*a* ⇒ *bool*
    and *idx* :: ′*a bound* ⇒ ′*a* ⇒ *nat*
    and *size* :: ′*a bound* ⇒ *nat*
  assumes *is_locale*: *index_locale less_eq less idx size*

locale *bounded_index* =
  fixes *bound* :: ′*k* :: *index bound*
begin

interpretation *index_locale less_eq less idx size*
  ⟨*proof*⟩

definition *size* ≡ *index_class.size bound* **for** *size*

definition *checked_idx x* ≡ *if in_bound less_eq less bound x then idx bound x else size*

lemma *checked_idx_injective*:
  injective size checked_idx
  ⟨*proof*⟩
end

instantiation *nat* :: *index*
begin

interpretation *nat_index* ⟨*proof*⟩

thm *index_locale_axioms*

definition [*simp*]: *less_eq_nat* $\equiv$ ($\leq$) :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
definition [*simp*]: *less_nat* $\equiv$ (<) :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
definition [*simp*]: *idx_nat* $\equiv$ *idx*
definition *size_nat* **where** [*simp*]: *size_nat* $\equiv$ *size*

instance ⟨*proof*⟩

end

instantiation *int* :: *index*
begin

interpretation *int_index* ⟨*proof*⟩
thm *index_locale_axioms*

definition [*simp*]: *less_eq_int* $\equiv$ ($\leq$) :: *int* $\Rightarrow$ *int* $\Rightarrow$ *bool*
definition [*simp*]: *less_int* $\equiv$ (<) :: *int* $\Rightarrow$ *int* $\Rightarrow$ *bool*
definition [*simp*]: *idx_int* $\equiv$ *idx*
definition [*simp*]: *size_int* $\equiv$ *size*

lemmas *size_int* = *size.simps*

instance ⟨*proof*⟩
end

instantiation *prod* :: (*index*, *index*) *index*
begin

interpretation *prod_index*
  less_eq::'*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool less idx size*
  less_eq::'*b* $\Rightarrow$ '*b* $\Rightarrow$ *bool less idx size*
  ⟨*proof*⟩
thm *index_locale_axioms*

definition [*simp*]: *less_eq_prod* $\equiv$ *less_eq*
definition [*simp*]: *less_prod* $\equiv$ *less*
definition [*simp*]: *idx_prod* $\equiv$ *idx*
definition [*simp*]: *size_prod* $\equiv$ *size* **for** *size_prod*

lemmas *size_prod* = *size.simps*

instance ⟨*proof*⟩

end

lemma *bound_int_simp*[*code*]:
bounded_index.size (*Bound* (*l*1, *l*2) (*u*1, *u*2)) = *nat* (*u*1 − *l*1) ∗ *nat* (*u*2 − *l*2)
⟨*proof*⟩

lemmas [*code*] = *bounded_index.size_def bounded_index.checked_idx_def*

lemmas [*code*] =
nat_index_def.size.simps
nat_index_def.idx.simps

lemmas [*code*] =
int_index_def.size.simps
int_index_def.idx.simps

lemmas [*code*] =
prod_index_def.size.simps
prod_index_def.idx.simps

lemmas [*code*] =
prod_order_def.less_eq.simps
prod_order_def.less.simps

lemmas *index_size_defs* =
prod_index_def.size.simps int_index_def.size.simps nat_index_def.size.simps bounded_index.size_def

end

2.4 Heap Memory Implementations

theory *Memory_Heap*
imports *State_Heap DP_CRelVH Pair_Memory HOL−Eisbach.Eisbach ../Indexing*
begin

Move

abbreviation *result_of c h* ≡ *fst* (*the* (*execute c h*))
abbreviation *heap_of c h* ≡ *snd* (*the* (*execute c h*))

lemma *map_emptyI*:

$m \subseteq_m Map.empty$ **if** $\bigwedge x.\ m\ x = None$
⟨proof⟩

lemma *result_of_return*[*simp*]:
$result\_of\ (Heap\_Monad.return\ x)\ h = x$
⟨proof⟩

lemma *get_result_of_lookup*:
$result\_of\ (!r)\ heap = x$ **if** $Ref.get\ heap\ r = x$
⟨proof⟩

context
fixes $size :: nat$
and $to\_index :: ('k2 :: heap) \Rightarrow nat$
begin

definition
$mem\_empty = (Array.new\ size\ (None :: ('v :: heap)\ option))$

lemma *success_empty*[*intro*]:
success mem_empty heap
⟨proof⟩

lemma *length_mem_empty*:
Array.length
$(heap\_of\ (mem\_empty :: (('b :: heap)\ option\ array)\ Heap)\ h)$
$(result\_of\ (mem\_empty :: ('b\ option\ array)\ Heap)\ h) = size$
⟨proof⟩

lemma *nth_mem_empty*:
result_of
$(Array.nth\ (result\_of\ (mem\_empty :: ('b\ option\ array)\ Heap)\ h)\ i)$
$(heap\_of\ (mem\_empty :: (('b :: heap)\ option\ array)\ Heap)\ h) = None$
if $i < size$
⟨proof⟩

context
fixes $mem :: ('v :: heap)\ option\ array$
begin

definition
$mem\_lookup\ k = (let\ i = to\_index\ k\ in$
$if\ i < size\ then\ Array.nth\ mem\ i\ else\ return\ None$
$)$

definition
mem_update k v = (*let i* = *to_index k in*
if i < *size then* (*Array.upd i* (*Some v*) *mem* ≫= (λ _. *return* ()))
else return ()
)

context assumes *injective*: *injective size to_index*
begin

interpretation *heap_correct* λ*heap. Array.length heap mem* = *size mem_update mem_lookup*
⟨*proof*⟩

lemmas *mem_heap_correct* = *heap_correct_axioms*

context
assumes [*simp*]: *mem* = *result_of mem_empty Heap.empty*
begin

interpretation *heap_correct_empty*
λ*heap. Array.length heap mem* = *size mem_update mem_lookup*
heap_of (*mem_empty* :: ′*v option array Heap*) *Heap.empty*
⟨*proof*⟩

lemmas *array_heap_emptyI* = *heap_correct_empty_axioms*

context
fixes *dp* :: ′*k2* ⇒ ′*v*
begin

interpretation *dp_consistency_heap_empty*
λ*heap. Array.length heap mem* = *size mem_update mem_lookup dp*
heap_of (*mem_empty* :: ′*v option array Heap*) *Heap.empty*
⟨*proof*⟩

lemmas *array_consistentI* = *dp_consistency_heap_empty_axioms*

end

end

end

end

lemma *execute_bind_success'*:
  assumes *success f h execute* $(f \ggg g)$ $h = Some$ $(y,\ h'')$
  obtains $x\ h'$ **where** *execute f h* $= Some\ (x,\ h')$ *execute* $(g\ x)\ h' = Some$ $(y,\ h'')$
  $\langle proof \rangle$

lemma *success_bind_I*:
  assumes *success f h*
    and $\bigwedge x\ h'.\ execute\ f\ h = Some\ (x,\ h') \Longrightarrow success\ (g\ x)\ h'$
  shows *success* $(f \ggg g)$ h
  $\langle proof \rangle$

definition
  $alloc\_pair\ a\ b \equiv do\ \{$
    $r1 \leftarrow ref\ a;$
    $r2 \leftarrow ref\ b;$
    $return\ (r1,\ r2)$
  $\}$

lemma *alloc_pair_alloc*:
  $Ref.get\ heap'\ r1 = a\ Ref.get\ heap'\ r2 = b$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')$
  $\langle proof \rangle$

lemma *alloc_pairD1*:
  $r =!= r1 \wedge r =!= r2 \wedge Ref.present\ heap'\ r$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')\ Ref.present\ heap\ r$
  $\langle proof \rangle$

lemma *alloc_pairD2*:
  $r1 =!= r2 \wedge Ref.present\ heap'\ r2 \wedge Ref.present\ heap'\ r1$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')$
  $\langle proof \rangle$

lemma *alloc_pairD3*:
  $Array.present\ heap'\ r$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')\ Array.present\ heap\ r$
  $\langle proof \rangle$

lemma *alloc_pairD4*:
  $Ref.get\ heap'\ r = x$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')$
    $Ref.get\ heap\ r = x$ $Ref.present\ heap\ r$
  ⟨*proof*⟩

lemma *alloc_pair_array_get*:
  $Array.get\ heap'\ r = x$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')$ $Array.get\ heap\ r = x$
  ⟨*proof*⟩

lemma *alloc_pair_array_length*:
  $Array.length\ heap'\ r = Array.length\ heap\ r$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')$
  ⟨*proof*⟩

lemma *alloc_pair_nth*:
  $result\_of\ (Array.nth\ r\ i)\ heap' = result\_of\ (Array.nth\ r\ i)\ heap$
  if $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1,\ r2),\ heap')$
  ⟨*proof*⟩

lemma *succes_alloc_pair*[*intro*]:
  $success\ (alloc\_pair\ a\ b)\ heap$
  ⟨*proof*⟩

definition
  $init\_state\_inner\ k1\ k2\ m1\ m2 \equiv\ do\ \{$
    $(k\_ref1,\ k\_ref2) \leftarrow alloc\_pair\ k1\ k2;$
    $(m\_ref1,\ m\_ref2) \leftarrow alloc\_pair\ m1\ m2;$
    $return\ (k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2)$
  $\}$

lemma *init_state_inner_alloc*:
  assumes
    $execute\ (init\_state\_inner\ k1\ k2\ m1\ m2)\ heap = Some\ ((k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2),\ heap')$
  shows
    $Ref.get\ heap'\ k\_ref1 = k1$ $Ref.get\ heap'\ k\_ref2 = k2$
    $Ref.get\ heap'\ m\_ref1 = m1$ $Ref.get\ heap'\ m\_ref2 = m2$
  ⟨*proof*⟩

lemma *init_state_inner_distinct*:

assumes
execute (*init_state_inner* *k*1 *k*2 *m*1 *m*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap'*)
shows
*m_ref*1 =!= *m_ref*2 ∧ *m_ref*1 =!= *k_ref*1 ∧ *m_ref*1 =!= *k_ref*2 ∧ *m_ref*2 =!= *k_ref*1
∧ *m_ref*2 =!= *k_ref*2 ∧ *k_ref*1 =!= *k_ref*2
⟨*proof*⟩

lemma *init_state_inner_present*:
assumes
execute (*init_state_inner* *k*1 *k*2 *m*1 *m*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap'*)
shows
*Ref.present heap' k_ref*1 *Ref.present heap' k_ref*2
*Ref.present heap' m_ref*1 *Ref.present heap' m_ref*2
⟨*proof*⟩

lemma *inite_state_inner_present'*:
assumes
execute (*init_state_inner* *k*1 *k*2 *m*1 *m*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap'*)
Array.present heap a
shows
Array.present heap' a
⟨*proof*⟩

lemma *succes_init_state_inner*[*intro*]:
success (*init_state_inner* *k*1 *k*2 *m*1 *m*2) *heap*
⟨*proof*⟩

lemma *init_state_inner_nth*:
result_of (*Array.nth r i*) *heap'* = *result_of* (*Array.nth r i*) *heap*
if *execute* (*init_state_inner* *k*1 *k*2 *m*1 *m*2) *heap* = *Some* ((*r*1, *r*2), *heap'*)
⟨*proof*⟩

definition
init_state *k*1 *k*2 ≡ *do* {
*m*1 ← *mem_empty*;
*m*2 ← *mem_empty*;
init_state_inner *k*1 *k*2 *m*1 *m*2
}

lemma *succes_init_state*[*intro*]:

success (*init_state* *k*1 *k*2) *heap*
⟨*proof*⟩

definition
inv_distinct *k_ref*1 *k_ref*2 *m_ref*1 *m_ref*2 ≡
*m_ref*1 =!= *m_ref*2 ∧ *m_ref*1 =!= *k_ref*1 ∧ *m_ref*1 =!= *k_ref*2 ∧ *m_ref*2 =!= *k_ref*1
∧ *m_ref*2 =!= *k_ref*2 ∧ *k_ref*1 =!= *k_ref*2

lemma *init_state_distinct*:
assumes
execute (*init_state* *k*1 *k*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap*′)
shows
inv_distinct *k_ref*1 *k_ref*2 *m_ref*1 *m_ref*2
⟨*proof*⟩

lemma *init_state_present*:
assumes
execute (*init_state* *k*1 *k*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap*′)
shows
Ref.present heap′ *k_ref*1 *Ref.present heap*′ *k_ref*2
Ref.present heap′ *m_ref*1 *Ref.present heap*′ *m_ref*2
⟨*proof*⟩

lemma *empty_present*:
Array.present *h*′ *x* **if** *execute mem_empty heap* = *Some* (*x*, *h*′)
⟨*proof*⟩

lemma *empty_present*′:
Array.present *h*′ *a* **if** *execute mem_empty heap* = *Some* (*x*, *h*′) *Array.present heap a*
⟨*proof*⟩

lemma *init_state_present*2:
assumes
execute (*init_state* *k*1 *k*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap*′)
shows
Array.present heap′ (*Ref.get heap*′ *m_ref*1) *Array.present heap*′ (*Ref.get heap*′ *m_ref*2)
⟨*proof*⟩

lemma *init_state_neq*:
assumes
$execute\ (init\_state\ k1\ k2)\ heap = Some\ ((k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2),\ heap')$
shows
$Ref.get\ heap'\ m\_ref1 =!!= Ref.get\ heap'\ m\_ref2$
⟨*proof*⟩

lemma *present_alloc_get*:
$Array.get\ heap'\ a = Array.get\ heap\ a$
if $Array.alloc\ xs\ heap = (a',\ heap')$ $Array.present\ heap\ a$
⟨*proof*⟩

lemma *init_state_length*:
assumes
$execute\ (init\_state\ k1\ k2)\ heap = Some\ ((k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2),\ heap')$
shows
$Array.length\ heap'\ (Ref.get\ heap'\ m\_ref1) = size$
$Array.length\ heap'\ (Ref.get\ heap'\ m\_ref2) = size$
⟨*proof*⟩

context
fixes $key1 :: 'k \Rightarrow ('k1 :: heap)$ **and** $key2 :: 'k \Rightarrow 'k2$
and $m\_ref1\ m\_ref2 :: ('v :: heap)\ option\ array\ ref$
and $k\_ref1\ k\_ref2 :: ('k1 :: heap)\ ref$
begin

We assume that look-ups happen on the older row, so this is biased towards the second entry.

definition
```
lookup_pair k = do {
  let k' = key1 k;
  k2 ← !k_ref2;
  if k' = k2 then
    do {
      m2 ← !m_ref2;
      mem_lookup m2 (key2 k)
    }
  else
    do {
    k1 ← !k_ref1;
    if k' = k1 then
```

```
      do {
        m1 ← !m_ref1;
        mem_lookup m1 (key2 k)
      }
    else
      return None
  }
}
```

We assume that updates happen on the newer row, so this is biased towards the first entry.

definition

```
update_pair k v = do {
  let k' = key1 k;
    k1 ← !k_ref1;
    if k' = k1 then do {
      m ← !m_ref1;
      mem_update m (key2 k) v
    }
    else do {
      k2 ← !k_ref2;
      if k' = k2 then do {
        m ← !m_ref2;
        mem_update m (key2 k) v
      }
      else do {
        do {
          k1 ← !k_ref1;
          m ← mem_empty;
          m1 ← !m_ref1;
          k_ref2 := k1;
          k_ref1 := k';
          m_ref2 := m1;
          m_ref1 := m
        }
      ;
      m ← !m_ref1;
      mem_update m (key2 k) v
    }
  }
 }
```

definition
  inv_pair_weak heap = (
    let
      *m*1 = *Ref.get heap m_ref*1;
      *m*2 = *Ref.get heap m_ref*2
    *in Array.length heap m*1 = *size* ∧ *Array.length heap m*2 = *size*
      ∧ *Ref.present heap k_ref*1 ∧ *Ref.present heap k_ref*2
      ∧ *Ref.present heap m_ref*1 ∧ *Ref.present heap m_ref*2
      ∧ *Array.present heap m*1 ∧ *Array.present heap m*2
      ∧ *m*1 =!!= *m*2
)

definition
  inv_pair heap ≡ *inv_pair_weak heap* ∧ *inv_distinct k_ref*1 *k_ref*2 *m_ref*1 *m_ref*2

lemma *init_state_inv*:
  assumes
    execute (*init_state k*1 *k*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap*′)
  shows *inv_pair_weak heap*′
  ⟨*proof*⟩

lemma *inv_pair_lengthD*1:
  Array.length heap (*Ref.get heap m_ref*1) = *size* **if** *inv_pair_weak heap*
  ⟨*proof*⟩

lemma *inv_pair_lengthD*2:
  Array.length heap (*Ref.get heap m_ref*2) = *size* **if** *inv_pair_weak heap*
  ⟨*proof*⟩

lemma *inv_pair_presentD*:
  Array.present heap (*Ref.get heap m_ref*1) *Array.present heap* (*Ref.get heap m_ref*2)
  if *inv_pair_weak heap*
  ⟨*proof*⟩

lemma *inv_pair_presentD*2:
  *Ref.present heap m_ref*1 *Ref.present heap m_ref*2
  *Ref.present heap k_ref*1 *Ref.present heap k_ref*2
  if *inv_pair_weak heap*
  ⟨*proof*⟩

lemma *inv_pair_not_eqD*:
Ref.get heap m_ref1 =!!= *Ref.get heap m_ref2* **if** *inv_pair_weak heap*
⟨*proof*⟩

definition *lookup1 k* ≡ *state_of* (*do* {*m* ← !*m_ref1*; *mem_lookup m k*})

definition *lookup2 k* ≡ *state_of* (*do* {*m* ← !*m_ref2*; *mem_lookup m k*})

definition *update1 k v* ≡ *state_of* (*do* {*m* ← !*m_ref1*; *mem_update m k v*})

definition *update2 k v* ≡ *state_of* (*do* {*m* ← !*m_ref2*; *mem_update m k v*})

definition *move12 k* ≡ *state_of* (*do* {
k1 ← !*k_ref1*;
m ← *mem_empty*;
m1 ← !*m_ref1*;
k_ref2 := *k1*;
k_ref1 := *k*;
m_ref2 := *m1*;
m_ref1 := *m*
})

definition *get_k1* ≡ *state_of* (!*k_ref1*)

definition *get_k2* ≡ *state_of* (!*k_ref2*)

lemma *run_state_state_of*[*simp*]:
State_Monad.run_state (*state_of p*) *m* = *the* (*execute p m*)
⟨*proof*⟩

context assumes *injective*: *injective size to_index*
begin

context
assumes *inv_distinct*: *inv_distinct k_ref1 k_ref2 m_ref1 m_ref2*
begin

lemma *disjoint*[*simp*]:
m_ref1 =!= *m_ref2* *m_ref1* =!= *k_ref1* *m_ref1* =!= *k_ref2*
m_ref2 =!= *k_ref1* *m_ref2* =!= *k_ref2*
k_ref1 =!= *k_ref2*

⟨*proof*⟩

lemmas [*simp*] = *disjoint*[*THEN noteq_sym*]

lemma [*simp*]:
Array.get (*snd* (*Array.alloc xs heap*)) *a* = *Array.get heap a* **if** *Array.present heap a*
⟨*proof*⟩

lemma [*simp*]:
Ref.get (*snd* (*Array.alloc xs heap*)) *r* = *Ref.get heap r* **if** *Ref.present heap r*
⟨*proof*⟩

lemma *alloc_present*:
Array.present (*snd* (*Array.alloc xs heap*)) *a* **if** *Array.present heap a*
⟨*proof*⟩

lemma *alloc_present'*:
Ref.present (*snd* (*Array.alloc xs heap*)) *r* **if** *Ref.present heap r*
⟨*proof*⟩

lemma *length_get_upd*[*simp*]:
length (*Array.get* (*Array.update a i x heap*) *r*) = *length* (*Array.get heap r*)
⟨*proof*⟩

method *solve*1 =
(*frule inv_pair_lengthD*1, *frule inv_pair_lengthD*2, *frule inv_pair_not_eqD*)?,
auto split: *if_split_asm dest*: *Array.noteq_sym*

interpretation *pair*: *pair_mem lookup*1 *lookup*2 *update*1 *update*2 *move*12 *get_k*1 *get_k*2 *inv_pair_weak*
⟨*proof*⟩

lemmas *mem_correct_pair* = *pair.mem_correct_pair*

definition
*mem_lookup*1 *k* = *do* {*m* ← !*m_ref*1; *mem_lookup m k*}

definition
*mem_lookup*2 *k* = *do* {*m* ← !*m_ref*2; *mem_lookup m k*}

definition *get_k*1' ≡ !*k_ref*1

definition *get_k2'* ≡ !*k_ref2*

definition *update1' k v* ≡ *do* {*m* ← !*m_ref1*; *mem_update m k v*}

definition *update2' k v* ≡ *do* {*m* ← !*m_ref2*; *mem_update m k v*}

definition *move12' k* ≡ *do* {
k1 ← !*k_ref1*;
m ← *mem_empty*;
m1 ← !*m_ref1*;
k_ref2 := *k1*;
k_ref1 := *k*;
m_ref2 := *m1*;
m_ref1 := *m*
}

interpretation *heap_mem_defs inv_pair_weak lookup_pair update_pair*
⟨*proof*⟩

lemma *rel_state_ofI*:
rel_state (=) (*state_of m*) *m* **if**
∀ *heap*. *inv_pair_weak heap* ⟶ *success m heap*
lift_p inv_pair_weak m
⟨*proof*⟩

lemma *inv_pair_iff*:
inv_pair_weak = *inv_pair*
⟨*proof*⟩

lemma *lift_p_inv_pairI*:
State_Heap.lift_p inv_pair m **if** *State_Heap.lift_p inv_pair_weak m*
⟨*proof*⟩

lemma *lift_p_success*:
State_Heap.lift_p inv_pair_weak m
if *DP_CRelVS.lift_p inv_pair_weak* (*state_of m*) ∀ *heap*. *inv_pair_weak heap* ⟶ *success m heap*
⟨*proof*⟩

lemma *rel_state_ofI2*:
rel_state (=) (*state_of m*) *m* **if**
∀ *heap*. *inv_pair_weak heap* ⟶ *success m heap*
DP_CRelVS.lift_p inv_pair_weak (*state_of m*)
⟨*proof*⟩

context
  includes *lifting_syntax*
begin

lemma [*transfer_rule*]:
  $((=) ===> rel\_state\ (=))\ move12\ move12'$
  ⟨*proof*⟩

lemma [*transfer_rule*]:
  $((=) ===> rel\_state\ (rel\_option\ (=)))\ lookup1\ mem\_lookup1$
  ⟨*proof*⟩

lemma [*transfer_rule*]:
  $((=) ===> rel\_state\ (rel\_option\ (=)))\ lookup2\ mem\_lookup2$
  ⟨*proof*⟩

lemma [*transfer_rule*]:
  $rel\_state\ (=)\ get\_k1\ get\_k1'$
  ⟨*proof*⟩

lemma [*transfer_rule*]:
  $rel\_state\ (=)\ get\_k2\ get\_k2'$
  ⟨*proof*⟩

lemma [*transfer_rule*]:
  $((=) ===> (=) ===> rel\_state\ (=))\ update1\ update1'$
  ⟨*proof*⟩

lemma [*transfer_rule*]:
  $((=) ===> (=) ===> rel\_state\ (=))\ update2\ update2'$
  ⟨*proof*⟩

lemma [*transfer_rule*]:
  $((=) ===> rel\_state\ (rel\_option\ (=)))\ lookup1\ mem\_lookup1$
  ⟨*proof*⟩

lemma *rel_state_lookup*:
  $((=) ===> rel\_state\ (=))\ pair.lookup\_pair\ lookup\_pair$
  ⟨*proof*⟩

lemma *rel_state_update*:
  $((=) ===> (=) ===> rel\_state\ (=))\ pair.update\_pair\ update\_pair$
  ⟨*proof*⟩

interpretation *mem*: *heap_mem_defs pair.inv_pair lookup_pair update_pair*
⟨*proof*⟩

lemma *inv_pairD*:
inv_pair_weak heap **if** *pair.inv_pair heap*
⟨*proof*⟩

lemma *mem_rel_state_ofI*:
mem.rel_state (=) m' *m* **if**
rel_state (=) m' *m*
$\bigwedge$ *heap*. *pair.inv_pair heap* $\Longrightarrow$
(*case State_Monad.run_state* m' *heap of* (_, *heap*) $\Rightarrow$ *inv_pair_weak heap* $\longrightarrow$ *pair.inv_pair heap*)
⟨*proof*⟩

lemma *mem_rel_state_ofI*$'$:
mem.rel_state (=) m' *m* **if**
rel_state (=) m' *m*
DP_CRelVS.lift_p pair.inv_pair m'
⟨*proof*⟩

context
assumes *keys*: $\forall k\ k'.\ key1\ k = key1\ k' \wedge key2\ k = key2\ k' \longrightarrow k = k'$
begin

interpretation *mem_correct pair.lookup_pair pair.update_pair pair.inv_pair*
⟨*proof*⟩

lemma *rel_state_lookup*$'$:
((=) ===> *mem.rel_state* (=)) *pair.lookup_pair lookup_pair*
⟨*proof*⟩

lemma *rel_state_update*$'$:
((=) ===> (=) ===> *mem.rel_state* (=)) *pair.update_pair update_pair*
⟨*proof*⟩

interpretation *heap_correct pair.inv_pair update_pair lookup_pair*
⟨*proof*⟩

lemmas *heap_correct_pairI* = *heap_correct_axioms*

lemma *mem_rel_state_resultD*:

$\mathit{result\_of}\ m\ \mathit{heap} = \mathit{fst}\ (\mathit{run\_state}\ m'\ \mathit{heap})$ **if** $\mathit{mem.rel\_state}\ (=)\ m'\ m$ $\mathit{pair.inv\_pair}\ \mathit{heap}$
⟨*proof*⟩

lemma *map_of_heap_eq*:
$\mathit{mem.map\_of\_heap}\ \mathit{heap} = \mathit{pair.pair.map\_of}\ \mathit{heap}$ **if** $\mathit{pair.inv\_pair}\ \mathit{heap}$
⟨*proof*⟩

context
fixes $k1\ k2\ \mathit{heap}\ \mathit{heap}'$
assumes *init*: $\mathit{execute}\ (\mathit{init\_state}\ k1\ k2)\ \mathit{heap} = \mathit{Some}\ ((\mathit{k\_ref1},\ \mathit{k\_ref2},\ \mathit{m\_ref1},\ \mathit{m\_ref2}),\ \mathit{heap}')$
begin

lemma *init_state_empty1*:
$\mathit{pair.mem1.map\_of}\ \mathit{heap}'\ k = \mathit{None}$
⟨*proof*⟩

lemma *init_state_empty2*:
$\mathit{pair.mem2.map\_of}\ \mathit{heap}'\ k = \mathit{None}$
⟨*proof*⟩

lemma
shows *init_state_k1*: $\mathit{result\_of}\ (!\mathit{k\_ref1})\ \mathit{heap}' = k1$
and *init_state_k2*: $\mathit{result\_of}\ (!\mathit{k\_ref2})\ \mathit{heap}' = k2$
⟨*proof*⟩

context
assumes *neq*: $k1 \neq k2$
begin

lemma *init_state_inv'*:
$\mathit{pair.inv\_pair}\ \mathit{heap}'$
⟨*proof*⟩

lemma *init_state_empty*:
$\mathit{pair.pair.map\_of}\ \mathit{heap}' \subseteq_m \mathit{Map.empty}$
⟨*proof*⟩

interpretation *heap_correct_empty pair.inv_pair update_pair lookup_pair heap'*
⟨*proof*⟩

lemmas *heap_correct_empty_pairI* = *heap_correct_empty_axioms*

context
  fixes *dp* :: $'k \Rightarrow 'v$
begin

interpretation *dp_consistency_heap_empty*
  pair.inv_pair update_pair lookup_pair dp heap'
  ⟨*proof*⟩

lemmas *consistent_empty_pairI* = *dp_consistency_heap_empty_axioms*

end

end

end

end

end

end

end

end

end

end

2.5 Tool Setup

theory *Transform_Cmd*
  imports
    ../Pure_Monad
    ../state_monad/DP_CRelVS
    ../heap_monad/DP_CRelVH
  keywords
    memoize_fun :: *thy_decl*
    and *monadifies* :: *thy_decl*
    and *memoize_correct* :: *thy_goal*
    and *with_memory* :: *quasi_command*
    and *default_proof* :: *quasi_command*

begin

⟨*ML*⟩

end

2.6 Bottom-Up Computation

theory *Bottom_Up_Computation*
  imports *../state_monad/Memory ../state_monad/DP_CRelVS*
begin

fun *iterate_state* **where**
  iterate_state f [] = *State_Monad.return* () |
  iterate_state f (*x* # *xs*) = *do* {*f x*; *iterate_state f xs*}

locale *iterator_defs* =
  fixes *cnt* :: *'a* ⇒ *bool* **and** *nxt* :: *'a* ⇒ *'a*
begin

definition
  iter_state f ≡
    wfrec
      {(*nxt x*, *x*) | *x*. *cnt x*}
      (λ *rec x*. *if cnt x then do* {*f x*; *rec* (*nxt x*)} *else State_Monad.return*
())

definition
  iterator_to_list ≡
    wfrec {(*nxt x*, *x*) | *x*. *cnt x*} (λ *rec x*. *if cnt x then x* # *rec* (*nxt x*) *else*
[])

end

locale *iterator* = *iterator_defs* +
  fixes *sizef* :: *'a* ⇒ *nat*
  assumes *terminating*:
    finite {*x*. *cnt x*} ∀ *x*. *cnt x* ⟶ *sizef x* < *sizef* (*nxt x*)
begin

lemma *admissible*:
  adm_wf
      {(*nxt x*, *x*) | *x*. *cnt x*}

(λ *rec x. if cnt x then do* {*f x*; *rec* (*nxt x*)} *else State_Monad.return* ())
⟨*proof*⟩

lemma *wellfounded*:
wf {(*nxt x*, *x*) | *x*. *cnt x*} (**is** *wf ?S*)
⟨*proof*⟩

lemma *iter_state_unfold*:
iter_state f x = (*if cnt x then do* {*f x*; *iter_state f* (*nxt x*)} *else State_Monad.return* ())
⟨*proof*⟩

lemma *iterator_to_list_unfold*:
iterator_to_list x = (*if cnt x then x* # *iterator_to_list* (*nxt x*) *else* [])
⟨*proof*⟩

lemma *iter_state_iterate_state*:
iter_state f x = *iterate_state f* (*iterator_to_list x*)
⟨*proof*⟩

end

context *dp_consistency*
begin

context
includes *lifting_syntax*
begin

lemma *crel_vs_iterate_state*:
crel_vs (=) () (*iterate_state f xs*) **if** ((=) $===>_T$ *R*) *g f*
⟨*proof*⟩

lemma *crel_vs_bind_ignore*:
crel_vs R a (*do* {*d*; *b*}) **if** *crel_vs R a b crel_vs S c d*
⟨*proof*⟩

lemma *crel_vs_iterate_and_compute*:
assumes ((=) $===>_T$ *R*) *g f*
shows *crel_vs R* (*g x*) (*do* {*iterate_state f xs*; *f x*})
⟨*proof*⟩

end

end

locale *dp_consistency_iterator* =
  dp_consistency lookup update + *iterator cnt nxt sizef*
  for *lookup* :: $'a \Rightarrow ('b, 'c\ option)$ *state* **and** *update*
    and *cnt* :: $'a \Rightarrow$ *bool* **and** *nxt* **and** *sizef*
begin

lemma *crel_vs_iter_and_compute*:
  assumes $((=) ===>_T R)\ g\ f$
  shows *crel_vs R* (*g x*) (*do* {*iter_state f y*; *f x*})
  ⟨*proof*⟩

lemma *consistentDP_iter_and_compute*:
  assumes *consistentDP f*
  shows *crel_vs* (=) (*dp x*) (*do* {*iter_state f y*; *f x*})
  ⟨*proof*⟩

end

locale *dp_consistency_iterator_empty* =
  dp_consistency_iterator + *dp_consistency_empty*
begin

lemma *memoized*:
  dp x = *fst* (*run_state* (*do* {*iter_state f y*; *f x*}) *empty*) **if** *consistentDP f*
  ⟨*proof*⟩

lemma *cmem_result*:
  cmem (*snd* (*run_state* (*do* {*iter_state f y*; *f x*}) *empty*)) **if** *consistentDP*
f
  ⟨*proof*⟩

end

lemma *dp_consistency_iterator_emptyI*:
  dp_consistency_iterator_empty P lookup update cnt
    nxt sizef empty
  if *dp_consistency_empty lookup update P empty*
     iterator cnt nxt sizef
   for *empty*
  ⟨*proof*⟩

context
  fixes m :: *nat* — Width of a row
    and n :: *nat* — Number of rows
begin

lemma *table_iterator_up*:
  iterator
    $(\lambda\ (x,\ y).\ x \leq n \wedge y \leq m)$
    $(\lambda\ (x,\ y).\ \textit{if}\ y < m\ \textit{then}\ (x,\ y + 1)\ \textit{else}\ (x + 1,\ 0))$
    $(\lambda\ (x,\ y).\ x * (m + 1) + y)$
  ⟨*proof*⟩

lemma *table_iterator_down*:
  iterator
    $(\lambda\ (x,\ y).\ x \leq n \wedge y \leq m \wedge x > 0)$
    $(\lambda\ (x,\ y).\ \textit{if}\ y > 0\ \textit{then}\ (x,\ y - 1)\ \textit{else}\ (x - 1,\ m))$
    $(\lambda\ (x,\ y).\ (n - x) * (m + 1) + (m - y))$
  ⟨*proof*⟩

end

end
theory *Bottom_Up_Computation_Heap*
  imports *../state_monad/Bottom_Up_Computation ../heap_monad/DP_CRelVH*
begin

definition (**in** *iterator_defs*)
  iter_heap f ≡
    wfrec
      $\{(nxt\ x,\ x) \mid x.\ cnt\ x\}$
      $(\lambda\ rec\ x.\ \textit{if}\ cnt\ x\ \textit{then}\ do\ \{f\ x;\ rec\ (nxt\ x)\}\ \textit{else}\ return\ ())$

lemma (**in** *iterator*) *iter_heap_unfold*:
  $iter\_heap\ f\ x = (\textit{if}\ cnt\ x\ \textit{then}\ do\ \{f\ x;\ iter\_heap\ f\ (nxt\ x)\}\ \textit{else}\ return\ ())$
  ⟨*proof*⟩

locale *dp_consistency_iterator_heap* =
  dp_consistency_heap P update lookup dp + *iterator cnt nxt sizef*
  for *lookup* :: $'a \Rightarrow ('c\ option)\ Heap$ **and** *update* **and** *P dp*
    and *cnt* :: $'a \Rightarrow bool$ **and** *nxt* **and** *sizef*
begin

context
  includes *lifting_syntax*

begin

term *iter_heap*

term *crel_vs*

lemma *crel_vs_iterate_state*:
  crel_vs (=) () (*iter_heap f x*) **if** ((=) ===> *crel_vs R*) *g f*
  ⟨*proof*⟩

lemma *crel_vs_bind_ignore*:
  crel_vs R a (*do* {*d*; *b*}) **if** *crel_vs R a b crel_vs S c d*
⟨*proof*⟩

lemma *crel_vs_iter_and_compute*:
  assumes ((=) ===> *crel_vs R*) *g f*
  shows *crel_vs R* (*g x*) (*do* {*iter_heap f y*; *f x*})
  ⟨*proof*⟩

lemma *consistent_DP_iter_and_compute*:
  assumes *consistentDP f*
  shows *consistentDP* (λ *x*. *do* {*iter_heap f y*; *f x*})
  ⟨*proof*⟩

end

end

end

2.7 Setup for the Heap Monad

theory *Solve_Cong*
  imports *Main HOL−Eisbach.Eisbach*
begin

Method for solving trivial equalities with congruence reasoning

named_theorems *cong_rules*

method *solve_cong* **methods** *solve* =
  rule HOL.refl |
  rule cong_rules; *solve_cong solve* |
  solve; *fail*

end
theory *Heap_Main*
  imports
    ../heap_monad/Memory_Heap
    ../transform/Transform_Cmd
    Bottom_Up_Computation_Heap
    ../util/Solve_Cong
begin

context includes *heap_monad_syntax* **begin**

thm *if_cong*
lemma *ifT_cong*:
  assumes $b = c$ $c \Longrightarrow x = u$ $\neg c \Longrightarrow y = v$
  shows $Heap\_Monad\_Ext.if_T\ \langle b\rangle\ x\ y = Heap\_Monad\_Ext.if_T\ \langle c\rangle\ u\ v$
  ⟨*proof*⟩

lemma *return_app_return_cong*:
  assumes $f\ x = g\ y$
  shows $\langle f\rangle\ .\ \langle x\rangle = \langle g\rangle\ .\ \langle y\rangle$
  ⟨*proof*⟩

lemmas [*fundef_cong*] =
  return_app_return_cong
  ifT_cong
end
memoize_fun $comp_T$: *comp* **monadifies** (*heap*) *comp_def*
thm $comp_T'.simps$
lemma (**in** *dp_consistency_heap*) **shows** $comp_T\_transfer[transfer\_rule]$:
  $crel\_vs\ ((R1 ===>_T R2) ===>_T (R0 ===>_T R1) ===>_T (R0 ===>_T R2))\ comp\ comp_T$
  ⟨*proof*⟩

memoize_fun map_T: *map* **monadifies** (*heap*) *list.map*
lemma (**in** *dp_consistency_heap*) $map_T\_transfer[transfer\_rule]$:
  $crel\_vs\ ((R0 ===>_T R1) ===>_T list\_all2\ R0 ===>_T list\_all2\ R1)\ map\ map_T$
  ⟨*proof*⟩

memoize_fun $fold_T$: *fold* **monadifies** (*heap*) *fold.simps*
lemma (**in** *dp_consistency_heap*) $fold_T\_transfer[transfer\_rule]$:
  $crel\_vs\ ((R0 ===>_T R1 ===>_T R1) ===>_T list\_all2\ R0 ===>_T R1 ===>_T R1)\ fold\ fold_T$
  ⟨*proof*⟩

context includes *heap_monad_syntax* **begin**

thm *map_cong*
lemma *mapT_cong*:
  assumes $xs = ys \bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
  shows $map_T\ .\ \langle f \rangle\ .\ \langle xs \rangle = map_T\ .\ \langle g \rangle\ .\ \langle ys \rangle$
  ⟨*proof*⟩

thm *fold_cong*
lemma *foldT_cong*:
  assumes $xs = ys \bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
  shows $fold_T\ .\ \langle f \rangle\ .\ \langle xs \rangle = fold_T\ .\ \langle g \rangle\ .\ \langle ys \rangle$
  ⟨*proof*⟩

lemma *abs_unit_cong*:

  assumes $x = y$
  shows $(\lambda\_::unit.\ x) = (\lambda\_.\ y)$
  ⟨*proof*⟩

lemma *arg_cong4*:
  $f\ a\ b\ c\ d = f\ a'\ b'\ c'\ d'$ **if** $a = a'\ b = b'\ c = c'\ d = d'$
  ⟨*proof*⟩

lemmas [*fundef_cong*, *cong_rules*] =
  return_app_return_cong
  ifT_cong
  mapT_cong
  foldT_cong
  abs_unit_cong
lemmas [*cong_rules*] =
  arg_cong4[**where** f = *heap_mem_defs.checkmem*]
  arg_cong2[**where** f = *fun_app_lifted*]
end

context *dp_consistency_heap* **begin**
context includes *lifting_syntax* **and** *heap_monad_syntax* **begin**

named_theorems *dp_match_rule*

thm *if_cong*

lemma if_T_cong2:
assumes $Rel\ (=)\ b\ c\ c \Longrightarrow Rel\ (crel\_vs\ R)\ x\ x_T\ \neg c \Longrightarrow Rel\ (crel\_vs\ R)\ y\ y_T$
shows $Rel\ (crel\_vs\ R)\ (if\ (Wrap\ b)\ then\ x\ else\ y)\ (Heap\_Monad\_Ext.if_T\ \langle c\rangle\ x_T\ y_T)$
$\langle proof\rangle$

lemma map_T_cong2:
assumes
$is\_equality\ R$
$Rel\ R\ xs\ ys$
$\bigwedge x.\ x{\in}set\ ys \Longrightarrow Rel\ (crel\_vs\ S)\ (f\ x)\ (f_T{}'\ x)$
shows $Rel\ (crel\_vs\ (list\_all2\ S))\ (App\ (App\ map\ (Wrap\ f))\ (Wrap\ xs))\ (map_T\ .\ \langle f_T{}'\rangle\ .\ \langle ys\rangle)$
$\langle proof\rangle$

lemma $fold_T\_cong2$:
assumes
$is\_equality\ R$
$Rel\ R\ xs\ ys$
$\bigwedge x.\ x{\in}set\ ys \Longrightarrow Rel\ (crel\_vs\ (S\ {===}{>}\ crel\_vs\ S))\ (f\ x)\ (f_T{}'\ x)$
shows
$Rel\ (crel\_vs\ (S\ {===}{>}\ crel\_vs\ S))\ (fold\ f\ xs)\ (fold_T\ .\ \langle f_T{}'\rangle\ .\ \langle ys\rangle)$
$\langle proof\rangle$

lemma $refl2$:
$is\_equality\ R \Longrightarrow Rel\ R\ x\ x$
$\langle proof\rangle$

lemma rel_fun2:
assumes $is\_equality\ R0\ \bigwedge x.\ Rel\ R1\ (f\ x)\ (g\ x)$
shows $Rel\ (rel\_fun\ R0\ R1)\ f\ g$
$\langle proof\rangle$

lemma $crel\_vs\_return\_app\_return$:
assumes $Rel\ R\ (f\ x)\ (g\ x)$
shows $Rel\ R\ (App\ (Wrap\ f)\ (Wrap\ x))\ (\langle g\rangle\ .\ \langle x\rangle)$
$\langle proof\rangle$

thm $option.case\_cong[no\_vars]$
lemma $option\_case\_cong'$:
$Rel\ (=)\ option'\ option \Longrightarrow$
$(option = None \Longrightarrow Rel\ R\ f1\ g1) \Longrightarrow$
$(\bigwedge x2.\ option = Some\ x2 \Longrightarrow Rel\ R\ (f2\ x2)\ (g2\ x2)) \Longrightarrow$

$Rel\ R\ (case\ option'\ of\ None \Rightarrow f1 \mid Some\ x2 \Rightarrow f2\ x2)$
$(case\ option\ of\ None \Rightarrow g1 \mid Some\ x2 \Rightarrow g2\ x2)$
⟨*proof*⟩

thm *prod.case_cong[no_vars]*
lemma *prod_case_cong'*: **fixes** *prod prod'* **shows**
$Rel\ (=)\ prod\ prod' \Longrightarrow$
$(\bigwedge x1\ x2.\ prod' = (x1, x2) \Longrightarrow Rel\ R\ (f\ x1\ x2)\ (g\ x1\ x2)) \Longrightarrow$
$Rel\ R\ (case\ prod\ of\ (x1, x2) \Rightarrow f\ x1\ x2)$
$(case\ prod'\ of\ (x1, x2) \Rightarrow g\ x1\ x2)$
⟨*proof*⟩

lemmas [*dp_match_rule*] = *prod_case_cong' option_case_cong'*

lemmas [*dp_match_rule*] =
crel_vs_return_app_return

lemmas [*dp_match_rule*] =
map_T*_cong2*
$fold_T$*_cong2*
if_T*_cong2*

lemmas [*dp_match_rule*] =
crel_vs_return
crel_vs_fun_app
refl2
rel_fun2

end
end

2.7.1 More Heap

lemma *execute_heap_ofD*:
heap_of c h = h' **if** *execute c h = Some (v, h')*
⟨*proof*⟩

lemma *execute_result_ofD*:
result_of c h = v **if** *execute c h = Some (v, h')*
⟨*proof*⟩

locale *heap_correct_init_defs* =
fixes $P :: 'm \Rightarrow heap \Rightarrow bool$
and $lookup :: 'm \Rightarrow 'k \Rightarrow 'v\ option\ Heap$
and $update :: 'm \Rightarrow 'k \Rightarrow 'v \Rightarrow unit\ Heap$
begin

definition *map_of_heap′* **where**
$map\_of\_heap'\ m\ heap\ k = fst\ (the\ (execute\ (lookup\ m\ k)\ heap))$

end

locale *heap_correct_init_inv* = *heap_correct_init_defs* +
assumes *lookup_inv*: $\bigwedge m.\ lift\_p\ (P\ m)\ (lookup\ m\ k)$
assumes *update_inv*: $\bigwedge m.\ lift\_p\ (P\ m)\ (update\ m\ k\ v)$

locale *heap_correct_init* =
heap_correct_init_inv +
assumes *lookup_correct*:
$\bigwedge a.\ P\ a\ m \Longrightarrow map\_of\_heap'\ a\ (snd\ (the\ (execute\ (lookup\ a\ k)\ m)))$
$\subseteq_m (map\_of\_heap'\ a\ m)$
and *update_correct*:
$\bigwedge a.\ P\ a\ m \Longrightarrow$
$map\_of\_heap'\ a\ (snd\ (the\ (execute\ (update\ a\ k\ v)\ m))) \subseteq_m (map\_of\_heap'$
$a\ m)(k \mapsto v)$
begin

end

locale *dp_consistency_heap_init* = *heap_correct_init* _ *lookup* **for** *lookup*
$:: 'm \Rightarrow 'k \Rightarrow 'v\ option\ Heap$ +
fixes $dp :: 'k \Rightarrow 'v$
fixes $init :: 'm\ Heap$
assumes *success*: *success init Heap.empty*
assumes *empty_correct*:
$\bigwedge empty\ heap.\ execute\ init\ Heap.empty = Some\ (empty,\ heap) \Longrightarrow$
$map\_of\_heap'\ empty\ heap \subseteq_m Map.empty$
and *P_empty*: $\bigwedge empty\ heap.\ execute\ init\ Heap.empty = Some\ (empty,$
$heap) \Longrightarrow P\ empty\ heap$
begin

definition $init\_mem = result\_of\ init\ Heap.empty$

sublocale *dp_consistency_heap*
where $P{=}P\ init\_mem$

and *lookup=lookup init_mem*
and *update=update init_mem*
⟨*proof*⟩

interpretation *consistent*: *dp_consistency_heap_empty*
where *P=P init_mem*
and *lookup=lookup init_mem*
and *update=update init_mem*
and *empty= heap_of init Heap.empty*
⟨*proof*⟩

lemma *memoized_empty*:
$dp\ x = result\_of\ (init \ggg= (\lambda mem.\ dp_T\ mem\ x))\ Heap.empty$
if $consistentDP\ (dp_T\ (result\_of\ init\ Heap.empty))$
⟨*proof*⟩

end

locale *dp_consistency_heap_init'* = *heap_correct_init_ lookup* **for** *lookup* :: $'m \Rightarrow 'k \Rightarrow 'v$ *option Heap* +
fixes *dp* :: $'k \Rightarrow 'v$
fixes *init* :: $'m$ *Heap*
assumes *success*: *success init Heap.empty*
assumes *empty_correct*:
$\bigwedge$ *empty heap. execute init Heap.empty = Some (empty, heap)* $\Longrightarrow$ *map_of_heap' empty heap* $\subseteq_m$ *Map.empty*
and *P_empty*: $\bigwedge$ *empty heap. execute init Heap.empty = Some (empty, heap)* $\Longrightarrow$ *P empty heap*
begin

sublocale *dp_consistency_heap*
where *P=P init_mem*
and *lookup=lookup init_mem*
and *update=update init_mem*
⟨*proof*⟩

definition *init_mem = result_of init Heap.empty*

interpretation *consistent*: *dp_consistency_heap_empty*
where *P=P init_mem*
and *lookup=lookup init_mem*
and *update=update init_mem*
and *empty= heap_of init Heap.empty*
⟨*proof*⟩

lemma *memoized_empty*:
  $dp\ x = result\_of\ (init \gg= (\lambda mem.\ dp_T\ mem\ x))\ Heap.empty$
  if $consistentDP\ init\_mem\ (dp_T\ (result\_of\ init\ Heap.empty))$
  ⟨*proof*⟩

end

locale *dp_consistency_new* =
  fixes $dp :: 'k \Rightarrow 'v$
  fixes $P :: 'm \Rightarrow heap \Rightarrow bool$
    and $lookup :: 'm \Rightarrow 'k \Rightarrow 'v\ option\ Heap$
    and $update :: 'm \Rightarrow 'k \Rightarrow 'v \Rightarrow unit\ Heap$
    and *init*
  assumes
    success: *success init Heap.empty*
  assumes
    inv_init: $\bigwedge empty\ heap.\ execute\ init\ Heap.empty = Some\ (empty, heap)$
$\Longrightarrow P\ empty\ heap$
  assumes *consistent*:
    $\bigwedge empty\ heap.\ execute\ init\ Heap.empty = Some\ (empty, heap)$
      $\Longrightarrow dp\_consistency\_heap\_empty\ (P\ empty)\ (update\ empty)\ (lookup$
$empty)\ heap$
begin

sublocale *dp_consistency_heap_empty*
  where $P=P\ (result\_of\ init\ Heap.empty)$
    and $lookup=lookup\ (result\_of\ init\ Heap.empty)$
    and $update=update\ (result\_of\ init\ Heap.empty)$
    and $empty=\ heap\_of\ init\ Heap.empty$
  ⟨*proof*⟩

lemma *memoized_empty*:
  $dp\ x = result\_of\ (init \gg= (\lambda mem.\ dp_T\ mem\ x))\ Heap.empty$
  if $consistentDP\ (dp_T\ (result\_of\ init\ Heap.empty))$
  ⟨*proof*⟩

end

locale *dp_consistency_new'* =
  fixes $dp :: 'k \Rightarrow 'v$
  fixes $P :: 'm \Rightarrow heap \Rightarrow bool$
    and $lookup :: 'm \Rightarrow 'k \Rightarrow 'v\ option\ Heap$
    and $update :: 'm \Rightarrow 'k \Rightarrow 'v \Rightarrow unit\ Heap$

and *init*
and *mem* :: $'m$
assumes *mem_is_init*: *mem* = *result_of init Heap.empty*
assumes
success: *success init Heap.empty*
assumes
inv_init: $\bigwedge$ *empty heap. execute init Heap.empty* = *Some* (*empty*, *heap*) $\Longrightarrow$ *P empty heap*
assumes *consistent*:
$\bigwedge$ *empty heap. execute init Heap.empty* = *Some* (*empty*, *heap*)
$\Longrightarrow$ *dp_consistency_heap_empty* (*P empty*) (*update empty*) (*lookup empty*) *heap*
begin

sublocale *dp_consistency_heap_empty*
where *P*=*P mem*
and *lookup*=*lookup mem*
and *update*=*update mem*
and *empty*= *heap_of init Heap.empty*
⟨*proof*⟩

lemma *memoized_empty*:
dp x = *result_of* (*init* $\gg=$ (λ*mem*. dp_T *mem x*)) *Heap.empty*
if *consistentDP* (dp_T (*result_of init Heap.empty*))
⟨*proof*⟩

end

locale *dp_consistency_heap_array_new'* =
fixes *size* :: *nat*
and *to_index* :: ($'k$ :: *heap*) $\Rightarrow$ *nat*
and *mem* :: ($'v$::*heap*) *option array*
and *dp* :: $'k \Rightarrow 'v$::*heap*
assumes *mem_is_init*: *mem* = *result_of* (*mem_empty size*) *Heap.empty*
assumes *injective*: *injective size to_index*
begin

sublocale *dp_consistency_new'*
where *P* = λ *mem heap. Array.length heap mem* = *size*
and *lookup* = λ *mem. mem_lookup size to_index mem*
and *update* = λ *mem. mem_update size to_index mem*
and *init* = *mem_empty size*
and *mem* = *mem*
⟨*proof*⟩

```
thm memoized_empty

end

locale dp_consistency_heap_array_new =
  fixes size :: nat
    and to_index :: ('k :: heap) ⇒ nat
    and dp :: 'k ⇒ 'v::heap
  assumes injective: injective size to_index
begin

sublocale dp_consistency_new
  where P      = λ mem heap. Array.length heap mem = size
    and lookup = λ mem. mem_lookup size to_index mem
    and update = λ mem. mem_update size to_index mem
    and init   = mem_empty size
  ⟨proof⟩

thm memoized_empty

end

locale dp_consistency_heap_array =
  fixes size :: nat
    and to_index :: ('k :: heap) ⇒ nat
    and dp :: 'k ⇒ 'v::heap
  assumes injective: injective size to_index
begin

sublocale dp_consistency_heap_init
  where P=λmem heap. Array.length heap mem = size
    and lookup=λ mem. mem_lookup size to_index mem
    and update=λ mem. mem_update size to_index mem
    and init=mem_empty size
  ⟨proof⟩

end


locale dp_consistency_heap_array_pair' =
  fixes size :: nat
  fixes key1 :: 'k ⇒ ('k1 :: heap) and key2 :: 'k ⇒ 'k2 :: heap
    and to_index :: 'k2 ⇒ nat
```

```
    and dp :: 'k ⇒ 'v::heap
    and k1 k2 :: 'k1
    and mem :: ('k1 ref ×
             'k1 ref ×
             'v option array ref ×
             'v option array ref)
  assumes mem_is_init: mem = result_of (init_state size k1 k2) Heap.empty
  assumes injective: injective size to_index
      and keys_injective: ∀ k k'. key1 k = key1 k' ∧ key2 k = key2 k' ⟶ k
= k'
      and keys_neq: k1 ≠ k2
begin

definition
  inv_pair' = (λ (k_ref1, k_ref2, m_ref1, m_ref2).
      pair_mem_defs.inv_pair (lookup1 size to_index m_ref1)
        (lookup2 size to_index m_ref2) (get_k1 k_ref1)
        (get_k2 k_ref2)
        (inv_pair_weak size m_ref1 m_ref2 k_ref1 k_ref2) key1 key2)

sublocale dp_consistency_new'
  where P=inv_pair'
    and lookup=λ (k_ref1, k_ref2, m_ref1, m_ref2).
      lookup_pair size to_index key1 key2 m_ref1 m_ref2 k_ref1 k_ref2
    and update=λ (k_ref1, k_ref2, m_ref1, m_ref2).
      update_pair size to_index key1 key2 m_ref1 m_ref2 k_ref1 k_ref2
    and init=init_state size k1 k2
  ⟨proof⟩

end

locale dp_consistency_heap_array_pair_iterator =
  dp_consistency_heap_array_pair' where dp = dp + iterator where cnt
= cnt
  for dp :: 'k ⇒ 'v::heap and cnt :: 'k ⇒ bool
begin

sublocale dp_consistency_iterator_heap
  where P = inv_pair' mem
  and update = (case mem of
  (k_ref1, k_ref2, m_ref1, m_ref2) ⇒
    update_pair size to_index key1 key2 m_ref1 m_ref2 k_ref1 k_ref2)
  and lookup = (case mem of
  (k_ref1, k_ref2, m_ref1, m_ref2) ⇒
```

*lookup_pair size to_index key*1 *key*2 *m_ref*1 *m_ref*2 *k_ref*1 *k_ref*2)
⟨*proof*⟩

end

locale *dp_consistency_heap_array_pair* =
fixes *size* :: *nat*
fixes *key*1 :: *'k* ⇒ (*'k*1 :: *heap*) **and** *key*2 :: *'k* ⇒ *'k*2 :: *heap*
and *to_index* :: *'k*2 ⇒ *nat*
and *dp* :: *'k* ⇒ *'v*::*heap*
and *k*1 *k*2 :: *'k*1
assumes *injective*: *injective size to_index*
and *keys_injective*: ∀ *k k'*. *key*1 *k* = *key*1 *k'* ∧ *key*2 *k* = *key*2 *k'* ⟶ *k*
= *k'*
and *keys_neq*: *k*1 ≠ *k*2
begin

definition
inv_pair' = (λ (*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2).
pair_mem_defs.inv_pair (*lookup*1 *size to_index m_ref*1)
(*lookup*2 *size to_index m_ref*2) (*get_k*1 *k_ref*1)
(*get_k*2 *k_ref*2)
(*inv_pair_weak size m_ref*1 *m_ref*2 *k_ref*1 *k_ref*2) *key*1 *key*2)

sublocale *dp_consistency_new*
where *P*=*inv_pair'*
and *lookup*=λ (*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2).
*lookup_pair size to_index key*1 *key*2 *m_ref*1 *m_ref*2 *k_ref*1 *k_ref*2
and *update*=λ (*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2).
*update_pair size to_index key*1 *key*2 *m_ref*1 *m_ref*2 *k_ref*1 *k_ref*2
and *init*=*init_state size k*1 *k*2
⟨*proof*⟩

end

2.7.2 Code Setup

lemmas [*code_unfold*] = *heap_mem_defs.checkmem_checkmem'*[*symmetric*]
lemmas [*code*] =
heap_mem_defs.checkmem'_def
Heap_Main.map$_T$*_def*

end

2.8 Setup for the State Monad

theory *State_Main*
  imports
    ../transform/Transform_Cmd
    Memory
begin

context includes *state_monad_syntax* **begin**

thm *if_cong*
lemma *ifT_cong*:
  assumes $b = c$ $c \Longrightarrow x = u$ $\neg c \Longrightarrow y = v$
  shows *State_Monad_Ext.if*$_T$ $\langle b \rangle$ x y = *State_Monad_Ext.if*$_T$ $\langle c \rangle$ u v
  $\langle proof \rangle$

lemma *return_app_return_cong*:
  assumes $f\ x = g\ y$
  shows $\langle f \rangle$. $\langle x \rangle$ = $\langle g \rangle$. $\langle y \rangle$
  $\langle proof \rangle$

lemmas [*fundef_cong*] =
  return_app_return_cong
  ifT_cong
end

memoize_fun *comp*$_T$: *comp* **monadifies** (*state*) *comp_def*
lemma (**in** *dp_consistency*) *comp*$_T$*_transfer*[*transfer_rule*]:
  crel_vs ((*R*1 ===>$_T$ *R*2) ===>$_T$ (*R*0 ===>$_T$ *R*1) ===>$_T$ (*R*0 ===>$_T$ *R*2)) *comp* *comp*$_T$
  $\langle proof \rangle$

memoize_fun *map*$_T$: *map* **monadifies** (*state*) *list.map*
lemma (**in** *dp_consistency*) *map*$_T$*_transfer*[*transfer_rule*]:
  crel_vs ((*R*0 ===>$_T$ *R*1) ===>$_T$ *list_all*2 *R*0 ===>$_T$ *list_all*2 *R*1) *map* *map*$_T$
  $\langle proof \rangle$

memoize_fun *fold*$_T$: *fold* **monadifies** (*state*) *fold.simps*
lemma (**in** *dp_consistency*) *fold*$_T$*_transfer*[*transfer_rule*]:
  crel_vs ((*R*0 ===>$_T$ *R*1 ===>$_T$ *R*1) ===>$_T$ *list_all*2 *R*0 ===>$_T$ *R*1 ===>$_T$ *R*1) *fold* *fold*$_T$
  $\langle proof \rangle$

context includes *state_monad_syntax* **begin**

thm *map_cong*
lemma *mapT_cong*:
  assumes $xs = ys$ $\bigwedge x.\ x{\in}set\ ys \Longrightarrow f\ x = g\ x$
  shows $map_T\ .\ \langle f\rangle\ .\ \langle xs\rangle = map_T\ .\ \langle g\rangle\ .\ \langle ys\rangle$
  ⟨*proof*⟩

thm *fold_cong*
lemma *foldT_cong*:
  assumes $xs = ys$ $\bigwedge x.\ x{\in}set\ ys \Longrightarrow f\ x = g\ x$
  shows $fold_T\ .\ \langle f\rangle\ .\ \langle xs\rangle = fold_T\ .\ \langle g\rangle\ .\ \langle ys\rangle$
  ⟨*proof*⟩

lemma *abs_unit_cong*:

  assumes $x = y$
  shows $(\lambda\_::unit.\ x) = (\lambda\_.\ y)$
  ⟨*proof*⟩

lemmas [*fundef_cong*] =
  return_app_return_cong
  ifT_cong
  mapT_cong
  foldT_cong
  abs_unit_cong
end

context *dp_consistency* **begin**
context includes *lifting_syntax* **and** *state_monad_syntax* **begin**

named_theorems *dp_match_rule*

thm *if_cong*
lemma if_T*_cong2*:
  assumes $Rel\ (=)\ b\ c\ c \Longrightarrow Rel\ (crel\_vs\ R)\ x\ x_T\ \neg c \Longrightarrow Rel\ (crel\_vs\ R)\ y\ y_T$
  shows $Rel\ (crel\_vs\ R)\ (if\ (Wrap\ b)\ then\ x\ else\ y)\ (State\_Monad\_Ext.if_T\ \langle c\rangle\ x_T\ y_T)$
  ⟨*proof*⟩

lemma map_T*_cong2*:
  assumes
    is_equality R

$Rel\ R\ xs\ ys$
$\bigwedge x.\ x{\in}set\ ys \Longrightarrow Rel\ (crel\_vs\ S)\ (f\ x)\ (f_T{}'\ x)$
shows $Rel\ (crel\_vs\ (list\_all2\ S))\ (App\ (App\ map\ (Wrap\ f))\ (Wrap\ xs))$
$(map_T\ .\ \langle f_T{}'\rangle\ .\ \langle ys\rangle)$
$\langle proof\rangle$

lemma $fold_T\_cong2$:
assumes
$is\_equality\ R$
$Rel\ R\ xs\ ys$
$\bigwedge x.\ x{\in}set\ ys \Longrightarrow Rel\ (crel\_vs\ (S\ {=}{=}{=}{>}\ crel\_vs\ S))\ (f\ x)\ (f_T{}'\ x)$
shows
$Rel\ (crel\_vs\ (S\ {=}{=}{=}{>}\ crel\_vs\ S))\ (fold\ f\ xs)\ (fold_T\ .\ \langle f_T{}'\rangle\ .\ \langle ys\rangle)$
$\langle proof\rangle$

lemma $refl2$:
$is\_equality\ R \Longrightarrow Rel\ R\ x\ x$
$\langle proof\rangle$

lemma rel_fun2:
assumes $is\_equality\ R0\ \bigwedge x.\ Rel\ R1\ (f\ x)\ (g\ x)$
shows $Rel\ (rel\_fun\ R0\ R1)\ f\ g$
$\langle proof\rangle$

lemma $crel\_vs\_return\_app\_return$:
assumes $Rel\ R\ (f\ x)\ (g\ x)$
shows $Rel\ R\ (App\ (Wrap\ f)\ (Wrap\ x))\ (\langle g\rangle\ .\ \langle x\rangle)$
$\langle proof\rangle$

thm $option.case\_cong[no\_vars]$
lemma $option\_case\_cong'$:
$Rel\ (=)\ option'\ option \Longrightarrow$
$(option = None \Longrightarrow Rel\ R\ f1\ g1) \Longrightarrow$
$(\bigwedge x2.\ option = Some\ x2 \Longrightarrow Rel\ R\ (f2\ x2)\ (g2\ x2)) \Longrightarrow$
$Rel\ R\ (case\ option'\ of\ None \Rightarrow f1\ |\ Some\ x2 \Rightarrow f2\ x2)$
$(case\ option\ of\ None \Rightarrow g1\ |\ Some\ x2 \Rightarrow g2\ x2)$
$\langle proof\rangle$

thm $prod.case\_cong[no\_vars]$
lemma $prod\_case\_cong'$: **fixes** $prod\ prod'$ **shows**
$Rel\ (=)\ prod\ prod' \Longrightarrow$
$(\bigwedge x1\ x2.\ prod' = (x1,\ x2) \Longrightarrow Rel\ R\ (f\ x1\ x2)\ (g\ x1\ x2)) \Longrightarrow$
$Rel\ R\ (case\ prod\ of\ (x1,\ x2) \Rightarrow f\ x1\ x2)$
$(case\ prod'\ of\ (x1,\ x2) \Rightarrow g\ x1\ x2)$

⟨*proof*⟩

thm *nat.case_cong*[*no_vars*]
lemma *nat_case_cong′*: **fixes** *nat nat′* **shows**
$Rel\ (=)\ nat\ nat' \Longrightarrow$
$(nat' = 0 \Longrightarrow Rel\ R\ f1\ g1) \Longrightarrow$
$(\bigwedge x2.\ nat' = Suc\ x2 \Longrightarrow Rel\ R\ (f2\ x2)\ (g2\ x2)) \Longrightarrow$
$Rel\ R\ (case\ nat\ of\ 0 \Rightarrow f1 \mid Suc\ x2 \Rightarrow f2\ x2)\ (case\ nat'\ of\ 0 \Rightarrow g1 \mid Suc\ x2 \Rightarrow g2\ x2)$
⟨*proof*⟩

lemmas [*dp_match_rule*] =
prod_case_cong′
option_case_cong′
nat_case_cong′

lemmas [*dp_match_rule*] =
crel_vs_return_app_return

lemmas [*dp_match_rule*] =
map_T*_cong2*
$fold_T$*_cong2*
if_T*_cong2*

lemmas [*dp_match_rule*] =
crel_vs_return
crel_vs_fun_app
refl2
rel_fun2

end
end

2.8.1 Code Setup

lemmas [*code_unfold*] =
state_mem_defs.checkmem_checkmem′[*symmetric*]
state_mem_defs.checkmem′_def
map_T*_def*

end

3 Examples

3.1 Misc

theory *Example_Misc*
  imports
    Main
    HOL−Library.Extended
    ../state_monad/State_Main
begin

Lists **fun** *min_list* :: $'a$::*ord list* $\Rightarrow$ $'a$ **where**
  $min\_list\ (x\ \#\ xs) = (case\ xs\ of\ [] \Rightarrow x \mid \_ \Rightarrow min\ x\ (min\_list\ xs))$

lemma *fold_min_commute*:
  $fold\ min\ xs\ (min\ a\ b) = min\ a\ (fold\ min\ xs\ b)$ **for** $a :: 'a :: linorder$
  $\langle proof \rangle$

lemma *min_list_fold*:
  $min\_list\ (x\ \#\ xs) = fold\ min\ xs\ x$ **for** $x :: 'a :: linorder$
  $\langle proof \rangle$

lemma *induct_list*012:
  $[\![P\ [];\ \bigwedge x.\ P\ [x];\ \bigwedge x\ y\ zs.\ P\ (y\ \#\ zs) \Longrightarrow P\ (x\ \#\ y\ \#\ zs)]\!] \Longrightarrow P\ xs$
$\langle proof \rangle$

lemma *min_list_Min*: $xs \neq [] \Longrightarrow min\_list\ xs = Min\ (set\ xs)$
$\langle proof \rangle$

Extended Data Type **lemma** *Pinf_add_right*[*simp*]:
  $\infty + x = \infty$
  $\langle proof \rangle$

Syntax **bundle** *app_syntax* **begin**

notation *App* (**infixl** ‹\$› 999)
notation *Wrap* (‹⟪_⟫›)

end

end
theory *Tracing*
  imports
    ../heap_monad/Heap_Main
    HOL−Library.Code_Target_Numeral
    Show.Show_Instances
begin

NB: A more complete solution could be built by using the following entry: `https://www.isa-afp.org/entries/Show.html`.

definition *writeln* :: *String.literal* ⇒ *unit* **where**
  writeln = (λ *s*. ())

code_printing
  constant *writeln* ⇀ (*SML*) *writeln* _

definition *trace* **where**
  trace s x = (*let a* = *writeln s in x*)

lemma *trace_alt_def*[*simp*]:
  trace s x = (λ _. *x*) (*writeln s*)
  ⟨*proof*⟩

definition (**in** *heap_mem_defs*) *checkmem_trace* ::
  (′*k* ⇒ *String.literal*) ⇒ ′*k* ⇒ (*unit* ⇒ ′*v Heap*) ⇒ ′*v Heap*
  where
  checkmem_trace trace_key param calc ≡
    Heap_Monad.bind (*lookup param*) (λ *x*.
    case x of
      Some x ⇒ *trace* (*STR* ′′*Hit* ′′ + *trace_key param*) (*return x*)
    | *None* ⇒ *trace* (*STR* ′′*Miss* ′′ + *trace_key param*)
       Heap_Monad.bind (*calc* ()) (λ *x*.
        Heap_Monad.bind (*update param x*) (λ _.
        return x
)
)
)

lemma (**in** *heap_mem_defs*) *checkmem_checkmem_trace*:
  checkmem param calc = *checkmem_trace trace_key param* (λ_. *calc*)
  ⟨*proof*⟩

definition *nat_to_string* :: *nat* ⇒ *String.literal* **where**
nat_to_string x = *String.implode* (*show x*)

definition *nat_pair_to_string* :: *nat* × *nat* ⇒ *String.literal* **where**
nat_pair_to_string x = *String.implode* (*show x*)

value *show* (3 :: *nat*)

Code Setup **lemmas** [*code*] =
heap_mem_defs.checkmem_trace_def

lemmas [*code_unfold*] =
heap_mem_defs.checkmem_checkmem_trace[**where** *trace_key* = *nat_to_string*]
heap_mem_defs.checkmem_checkmem_trace[**where** *trace_key* = *nat_pair_to_string*]

end
theory *Ground_Function*
imports *Main*
keywords
ground_function :: *thy_decl*
begin

⟨*ML*⟩

end

3.2 The Bellman-Ford Algorithm

theory *Bellman_Ford*
imports
HOL−Library.IArray
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../heap_monad/Heap_Main
Example_Misc
../util/Tracing
../util/Ground_Function
begin

3.2.1 Misc

lemma *nat_le_cases*:
fixes *n* :: *nat*

assumes $i \le n$
obtains $i < n \mid i = n$
⟨*proof*⟩

context *dp_consistency_iterator*
begin

lemma *crel_vs_iterate_state*:
crel_vs $(=)$ $()$ $(iter\_state\ f\ x)$ **if** $((=) \Longrightarrow_T R)\ g\ f$
⟨*proof*⟩

lemma *consistent_crel_vs_iterate_state*:
crel_vs $(=)$ $()$ $(iter\_state\ f\ x)$ **if** *consistentDP f*
⟨*proof*⟩

end

instance *extended* :: (*countable*) *countable*
⟨*proof*⟩

instance *extended* :: (*heap*) *heap* ⟨*proof*⟩

instantiation *extended* :: (*conditionally_complete_lattice*) *complete_lattice*
begin

definition
Inf A = (
if $A = \{\} \lor A = \{\infty\}$ *then* ∞
else if $-\infty \in A \lor \neg$ *bdd_below* (*Fin* –‘ *A*) *then* $-\infty$
else Fin (*Inf* (*Fin* –‘ *A*)))

definition
Sup A = (
if $A = \{\} \lor A = \{-\infty\}$ *then* $-\infty$
else if $\infty \in A \lor \neg$ *bdd_above* (*Fin* –‘ *A*) *then* ∞
else Fin (*Sup* (*Fin* –‘ *A*)))

instance
⟨*proof*⟩

end

instance *extended* :: ({*conditionally_complete_lattice*,*linorder*}) *complete_linorder*
⟨*proof*⟩

lemma *Minf_eq_zero*[*simp*]: $-\infty = 0 \longleftrightarrow$ *False* **and** *Pinf_eq_zero*[*simp*]: $\infty = 0 \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *Sup_int*:
fixes x :: *int* **and** X :: *int set*
assumes $X \neq \{\}$ *bdd_above* X
shows *Sup* $X \in X \wedge (\forall y \in X.\ y \leq$ *Sup* $X)$
⟨*proof*⟩

lemmas *Sup_int_in* = *Sup_int*[*THEN conjunct*1]

lemma *Inf_int_in*:
fixes S :: *int set*
assumes $S \neq \{\}$ *bdd_below* S
shows *Inf* $S \in S$
⟨*proof*⟩

lemma *finite_setcompr_eq_image*: *finite* $\{f\ x\ |x.\ P\ x\} \longleftrightarrow$ *finite* $(f$ ‘ $\{x.\ P\ x\})$
⟨*proof*⟩

lemma *finite_lists_length_le*1: *finite* $\{xs.$ *length* $xs \leq i \wedge$ *set* $xs \subseteq \{0..(n::nat)\}\}$ **for** i
⟨*proof*⟩

lemma *finite_lists_length_le*2: *finite* $\{xs.$ *length* $xs + 1 \leq i \wedge$ *set* $xs \subseteq \{0..(n::nat)\}\}$ **for** i
⟨*proof*⟩

lemmas [*simp*] =
*finite_setcompr_eq_image finite_lists_length_le*2[*simplified*] *finite_lists_length_le*1

lemma *get_return*:
run_state (*State_Monad.bind State_Monad.get* (λ *m*. *State_Monad.return* (*f m*))) *m* = (*f m*, *m*)
⟨*proof*⟩

lemma *list_pidgeonhole*:

assumes *set xs* $\subseteq$ *S card S* $<$ *length xs finite S*
obtains *as a bs cs* **where** *xs* = *as* @ *a* # *bs* @ *a* # *cs*
⟨*proof*⟩

lemma *path_eq_cycleE*:
assumes *v* # *ys* @ [*t*] = *as* @ *a* # *bs* @ *a* # *cs*
obtains (*Nil_Nil*) *as* = [] *cs* = [] *v* = *a a* = *t ys* = *bs*
| (*Nil_Cons*) *cs'* **where** *as* = [] *v* = *a ys* = *bs* @ *a* # *cs' cs* = *cs'* @ [*t*]
| (*Cons_Nil*) *as'* **where** *as* = *v* # *as' cs* = [] *a* = *t ys* = *as'* @ *a* # *bs*
| (*Cons_Cons*) *as' cs'* **where** *as* = *v* # *as' cs* = *cs'* @ [*t*] *ys* = *as'* @ *a* # *bs* @ *a* # *cs'*
⟨*proof*⟩

lemma *le_add_same_cancel1*:
$a + b \geq a \longleftrightarrow b \geq 0$ **if** $a < \infty$ $-\infty < a$ **for** *a b* :: *int extended*
⟨*proof*⟩

lemma *add_gt_minfI*:
assumes $-\infty < a$ $-\infty < b$
shows $-\infty < a + b$
⟨*proof*⟩

lemma *add_lt_infI*:
assumes $a < \infty$ $b < \infty$
shows $a + b < \infty$
⟨*proof*⟩

lemma *sum_list_not_infI*:
sum_list xs $< \infty$ **if** $\forall$ $x \in$ *set xs*. $x < \infty$ **for** *xs* :: *int extended list*
⟨*proof*⟩

lemma *sum_list_not_minfI*:
sum_list xs $> -\infty$ **if** $\forall$ $x \in$ *set xs*. $x > -\infty$ **for** *xs* :: *int extended list*
⟨*proof*⟩

3.2.2 Single-Sink Shortest Path Problem

datatype *bf_result* = *Path nat list int* | *No_Path* | *Computation_Error*

context
fixes *n* :: *nat* **and** *W* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *int extended*
begin

context

fixes t :: *nat* — Final node
begin

The correctness proof closely follows Kleinberg & Tardos: "Algorithm Design", chapter "Dynamic Programming" [1]

fun *weight* :: *nat list* $\Rightarrow$ *int extended* **where**
  $weight\ [v] = 0$
$|\ weight\ (v\ \#\ w\ \#\ xs) = W\ v\ w + weight\ (w\ \#\ xs)$

definition
  $OPT\ i\ v = ($
    $Min\ ($
      $\{weight\ (v\ \#\ xs\ @\ [t]) \mid xs.\ length\ xs + 1 \leq i \wedge set\ xs \subseteq \{0..n\}\} \cup$
      $\{if\ t = v\ then\ 0\ else\ \infty\}$
    $)$
  $)$

lemma *weight_alt_def'*:
  $weight\ (s\ \#\ xs) + w = snd\ (fold\ (\lambda j\ (i,\ x).\ (j,\ W\ i\ j + x))\ xs\ (s,\ w))$
  $\langle proof \rangle$

lemma *weight_alt_def*:
  $weight\ (s\ \#\ xs) = snd\ (fold\ (\lambda j\ (i,\ x).\ (j,\ W\ i\ j + x))\ xs\ (s,\ 0))$
  $\langle proof \rangle$

lemma *weight_append*:
  $weight\ (xs\ @\ a\ \#\ ys) = weight\ (xs\ @\ [a]) + weight\ (a\ \#\ ys)$
  $\langle proof \rangle$

lemma *OPT_0*:
  $OPT\ 0\ v = (if\ t = v\ then\ 0\ else\ \infty)$
  $\langle proof \rangle$

3.2.3 Functional Correctness

lemma *OPT_cases*:
  obtains (*path*) *xs* **where** $OPT\ i\ v = weight\ (v\ \#\ xs\ @\ [t])$ $length\ xs + 1 \leq i$ $set\ xs \subseteq \{0..n\}$
  $|$ (*sink*) $v = t$ $OPT\ i\ v = 0$
  $|$ (*unreachable*) $v \neq t$ $OPT\ i\ v = \infty$
  $\langle proof \rangle$

lemma *OPT_Suc*:
  $OPT\ (Suc\ i)\ v = min\ (OPT\ i\ v)\ (Min\ \{OPT\ i\ w + W\ v\ w \mid w.\ w \leq n\})$

(**is** *?lhs = ?rhs*)
if $t \leq n$
⟨*proof*⟩

fun *bf* :: *nat* ⇒ *nat* ⇒ *int extended* **where**
bf 0 *v* = (*if* *t* = *v* *then* 0 *else* ∞)
| *bf* (*Suc i*) *v* = *min_list*
(*bf i v* # [*W v w* + *bf i w* . *w* ← [0 ..< *Suc n*]])

lemmas [*simp del*] = *bf.simps*
lemmas *bf_simps*[*simp*] = *bf.simps*[*unfolded min_list_fold*]

lemma *bf_correct*:
OPT i j = *bf i j* **if** ‹$t \leq n$›
⟨*proof*⟩

3.2.4 Functional Memoization

memoize_fun bf_m: *bf* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *bf.simps*

Generated Definitions

context includes *state_monad_syntax* **begin**
thm bf_m'*.simps* bf_m*_def*
end

Correspondence Proof

memoize_correct
⟨*proof*⟩
print_theorems
lemmas [*code*] = bf_m*.memoized_correct*

interpretation *iterator*
λ (*x*, *y*). $x \leq n \wedge y \leq n$
λ (*x*, *y*). *if* $y < n$ *then* (*x*, *y* + 1) *else* (*x* + 1, 0)
λ (*x*, *y*). $x * (n + 1) + y$
⟨*proof*⟩

interpretation *bottom_up*: *dp_consistency_iterator_empty*
λ (_::(*nat* × *nat*, *int extended*) *mapping*). *True*
λ (*x*, *y*). *bf x y*
λ *k*. *do* {*m* ← *State_Monad.get*; *State_Monad.return* (*Mapping.lookup m k* :: *int extended option*)}

λ k v. *do* $\{m \leftarrow$ *State_Monad.get*; *State_Monad.set* (*Mapping.update* k v m)$\}$
λ (x, y). $x \leq n \wedge y \leq n$
λ (x, y). *if* $y < n$ *then* $(x, y + 1)$ *else* $(x + 1, 0)$
λ (x, y). $x * (n + 1) + y$
Mapping.empty ⟨*proof*⟩

definition
iter_bf = *iter_state* (λ (x, y). $bf_m{}'$ x y)

lemma *iter_bf_unfold*[*code*]:
iter_bf = (λ (i, j).
(*if* $i \leq n \wedge j \leq n$
then *do* {
$bf_m{}'$ i j;
iter_bf (*if* $j < n$ *then* $(i, j + 1)$ *else* $(i + 1, 0)$)
}
else *State_Monad.return* ()))
⟨*proof*⟩

lemmas *bf_memoized* = bf_m.*memoized*[*OF* bf_m.*crel*]
lemmas *bf_bottom_up* = *bottom_up.memoized*[*OF* bf_m.*crel*, *folded iter_bf_def*]

This will be our final implementation, which includes detection of negative cycles. See the corresponding section below for the correctness proof.

definition
bellman_ford ≡
do {
_ ← *iter_bf* (n, n);
xs ← *State_Main.map*$_T{}'$ (λi. $bf_m{}'$ n i) [0..<n+1];
ys ← *State_Main.map*$_T{}'$ (λi. $bf_m{}'$ $(n + 1)$ i) [0..<n+1];
State_Monad.return (*if xs* = *ys then Some xs else None*)
}

context
includes *state_monad_syntax*
begin

lemma *bellman_ford_alt_def*:
bellman_ford ≡
do {
_ ← *iter_bf* (n, n);
(⟨λ*xs*. ⟨λ*ys*. *State_Monad.return* (*if xs* = *ys then Some xs else None*)⟩
. (*State_Main.map*$_T$. ⟨λi. $bf_m{}'$ $(n + 1)$ i⟩ . ⟨[0..<n+1]⟩)⟩)

. $(State\_Main.map_T$. $\langle\lambda i.\ bf_m{}'\ n\ i\rangle$. $\langle[0..<n+1]\rangle)$
}
$\langle proof\rangle$

end

3.2.5 Imperative Memoization

context
fixes $mem :: nat\ ref \times nat\ ref \times int\ extended\ option\ array\ ref \times int\ extended\ option\ array\ ref$
assumes mem_is_init: $mem = result\_of\ (init\_state\ (n + 1)\ 1\ 0)\ Heap.empty$
begin

lemma $[intro]$:
$dp\_consistency\_heap\_array\_pair'\ (n + 1)\ fst\ snd\ id\ 1\ 0\ mem$
$\langle proof\rangle$

interpretation $iterator$
$\lambda\ (x, y).\ x \leq n \wedge y \leq n$
$\lambda\ (x, y).\ if\ y < n\ then\ (x, y + 1)\ else\ (x + 1, 0)$
$\lambda\ (x, y).\ x * (n + 1) + y$
$\langle proof\rangle$

lemma $[intro]$:
$dp\_consistency\_heap\_array\_pair\_iterator\ (n + 1)\ fst\ snd\ id\ 1\ 0\ mem$
$(\lambda\ (x, y).\ if\ y < n\ then\ (x, y + 1)\ else\ (x + 1, 0))$
$(\lambda\ (x, y).\ x * (n + 1) + y)$
$(\lambda\ (x, y).\ x \leq n \wedge y \leq n)$
$\langle proof\rangle$

memoize_fun bf_h: bf
with_memory (**default_proof**) $dp\_consistency\_heap\_array\_pair\_iterator$
where $size = n + 1$
and $key1 = fst :: nat \times nat \Rightarrow nat$ **and** $key2 = snd :: nat \times nat \Rightarrow nat$
and $k1 = 1 :: nat$ **and** $k2 = 0 :: nat$
and $to\_index = id :: nat \Rightarrow nat$
and $mem = mem$
and $cnt = \lambda\ (x, y).\ x \leq n \wedge y \leq n$
and $nxt = \lambda\ (x :: nat, y).\ if\ y < n\ then\ (x, y + 1)\ else\ (x + 1, 0)$
and $sizef = \lambda\ (x, y).\ x * (n + 1) + y$
monadifies $(heap)\ bf.simps$

memoize_correct

$\langle proof \rangle$

lemmas $memoized\_empty = bf_h.memoized\_empty[OF\ bf_h.consistent\_DP\_iter\_and\_compute[OF\ bf_h.crel]]$
lemmas $iter\_heap\_unfold = iter\_heap\_unfold$

end

3.2.6 Detecting Negative Cycles

definition
$shortest\ v = ($
$Inf\ ($
$\{weight\ (v\ \#\ xs\ @\ [t])\ |\ xs.\ set\ xs \subseteq \{0..n\}\} \cup$
$\{if\ t = v\ then\ 0\ else\ \infty\}$
$)$
$)$

definition
$is\_path\ xs \equiv weight\ (xs\ @\ [t]) < \infty$

definition
$has\_negative\_cycle \equiv$
$\exists xs\ a\ ys.\ set\ (a\ \#\ xs\ @\ ys) \subseteq \{0..n\} \wedge weight\ (a\ \#\ xs\ @\ [a]) < 0 \wedge is\_path\ (a\ \#\ ys)$

definition
$reaches\ a \equiv \exists xs.\ is\_path\ (a\ \#\ xs) \wedge a \leq n \wedge set\ xs \subseteq \{0..n\}$

lemma $fold\_sum\_aux'$:
assumes $\forall u \in set\ (a\ \#\ xs).\ \forall v \in set\ (xs\ @\ [b]).\ f\ v + W\ u\ v \geq f\ u$
shows $sum\_list\ (map\ f\ (a\ \#\ xs)) \leq sum\_list\ (map\ f\ (xs\ @\ [b])) + weight\ (a\ \#\ xs\ @\ [b])$
$\langle proof \rangle$

lemma $fold\_sum\_aux$:
assumes $\forall u \in set\ (a\ \#\ xs).\ \forall v \in set\ (a\ \#\ xs).\ f\ v + W\ u\ v \geq f\ u$
shows $sum\_list\ (map\ f\ (a\ \#\ xs\ @\ [a])) \leq sum\_list\ (map\ f\ (a\ \#\ xs\ @\ [a])) + weight\ (a\ \#\ xs\ @\ [a])$
$\langle proof \rangle$

context
begin

private definition $is\_path2\ xs \equiv weight\ xs < \infty$

private lemma $is\_path2\_remove\_cycle$:
  assumes $is\_path2\ (as\ @\ a\ \#\ bs\ @\ a\ \#\ cs)$
  shows $is\_path2\ (as\ @\ a\ \#\ cs)$
$\langle proof \rangle$ **lemma** is_path_eq:
  $is\_path\ xs \longleftrightarrow is\_path2\ (xs\ @\ [t])$
  $\langle proof \rangle$

lemma $is\_path\_remove\_cycle$:
  assumes $is\_path\ (as\ @\ a\ \#\ bs\ @\ a\ \#\ cs)$
  shows $is\_path\ (as\ @\ a\ \#\ cs)$
  $\langle proof \rangle$

lemma $is\_path\_remove\_cycle2$:
  assumes $is\_path\ (as\ @\ t\ \#\ cs)$
  shows $is\_path\ as$
  $\langle proof \rangle$

end

lemma $is\_path\_shorten$:
  assumes $is\_path\ (i\ \#\ xs)\ i \leq n\ set\ xs \subseteq \{0..n\}\ t \leq n\ t \neq i$
  obtains xs **where** $is\_path\ (i\ \#\ xs)\ i \leq n\ set\ xs \subseteq \{0..n\}\ length\ xs < n$
$\langle proof \rangle$

lemma $reaches\_non\_inf\_path$:
  assumes $reaches\ i\ i \leq n\ t \leq n$
  shows $OPT\ n\ i < \infty$
$\langle proof \rangle$

lemma $OPT\_sink\_le\_0$:
  $OPT\ i\ t \leq 0$
  $\langle proof \rangle$

lemma $is\_path\_appendD$:
  assumes $is\_path\ (as\ @\ a\ \#\ bs)$
  shows $is\_path\ (a\ \#\ bs)$
  $\langle proof \rangle$

lemma $has\_negative\_cycleI$:
  assumes $set\ (a\ \#\ xs\ @\ ys) \subseteq \{0..n\}\ weight\ (a\ \#\ xs\ @\ [a]) < 0\ is\_path\ (a\ \#\ ys)$
  shows $has\_negative\_cycle$

$\langle proof \rangle$

lemma *OPT_cases2*:
obtains (*path*) *xs* **where**
$v \neq t$ $OPT\ i\ v \neq \infty$ $OPT\ i\ v = weight\ (v\ \#\ xs\ @\ [t])$ $length\ xs + 1 \leq i$ $set\ xs \subseteq \{0..n\}$
| (*unreachable*) $v \neq t$ $OPT\ i\ v = \infty$
| (*sink*) $v = t$ $OPT\ i\ v \leq 0$
$\langle proof \rangle$

lemma *shortest_le_OPT*:
assumes $v \leq n$
shows $shortest\ v \leq OPT\ i\ v$
$\langle proof \rangle$

context
assumes *W_wellformed*: $\forall\, i \leq n.\ \forall\, j \leq n.\ W\ i\ j > -\infty$
assumes $t \leq n$
begin

lemma *weight_not_minfI*:
$-\infty < weight\ xs$ **if** $set\ xs \subseteq \{0..n\}$ $xs \neq []$
$\langle proof \rangle$

lemma *OPT_not_minfI*:
$OPT\ n\ i > -\infty$ **if** $i \leq n$
$\langle proof \rangle$

theorem *detects_cycle*:
assumes *has_negative_cycle*
shows $\exists\, i \leq n.\ OPT\ (n + 1)\ i < OPT\ n\ i$
$\langle proof \rangle$

corollary *bf_detects_cycle*:
assumes *has_negative_cycle*
shows $\exists\, i \leq n.\ bf\ (n + 1)\ i < bf\ n\ i$
$\langle proof \rangle$

lemma *shortest_cases*:
assumes $v \leq n$
obtains (*path*) *xs* **where** $shortest\ v = weight\ (v\ \#\ xs\ @\ [t])$ $set\ xs \subseteq \{0..n\}$
| (*sink*) $v = t$ $shortest\ v = 0$

| (*unreachable*) $v \neq t$ *shortest* $v = \infty$
| (*negative_cycle*) *shortest* $v = -\infty$ $\forall\, x.\ \exists\, xs.$ *set* $xs \subseteq \{0..n\} \wedge$ *weight* $(v \# xs @ [t]) <$ *Fin* x
⟨*proof*⟩

lemma *simple_paths*:
assumes $\neg$ *has_negative_cycle* *weight* $(v \# xs @ [t]) < \infty$ *set* $xs \subseteq \{0..n\}$ $v \leq n$
obtains *ys* **where**
weight $(v \# ys @ [t]) \leq$ *weight* $(v \# xs @ [t])$ *set* $ys \subseteq \{0..n\}$ *length* $ys < n$ | $v = t$
⟨*proof*⟩

theorem *shorter_than_OPT_n_has_negative_cycle*:
assumes *shortest* $v <$ *OPT* $n\ v\ v \leq n$
shows *has_negative_cycle*
⟨*proof*⟩

corollary *detects_cycle_has_negative_cycle*:
assumes *OPT* $(n + 1)\ v <$ *OPT* $n\ v\ v \leq n$
shows *has_negative_cycle*
⟨*proof*⟩

corollary *bellman_ford_detects_cycle*:
has_negative_cycle $\longleftrightarrow (\exists\, v \leq n.$ *OPT* $(n + 1)\ v <$ *OPT* $n\ v)$
⟨*proof*⟩

corollary *bellman_ford_shortest_paths*:
assumes $\neg$ *has_negative_cycle*
shows $\forall\, v \leq n.$ *bf* $n\ v =$ *shortest* v
⟨*proof*⟩

lemma *OPT_mono*:
OPT $m\ v \leq$ *OPT* $n\ v$ **if** ‹$v \leq n$› ‹$n \leq m$›
⟨*proof*⟩

corollary *bf_fix*:
assumes $\neg$ *has_negative_cycle* $m \geq n$
shows $\forall\, v \leq n.$ *bf* $m\ v =$ *bf* $n\ v$
⟨*proof*⟩

lemma *bellman_ford_correct'*:
bf_m.*crel_vs* (=) (*if has_negative_cycle then None else Some* (*map shortest* $[0..<n+1]$)) *bellman_ford*

⟨*proof*⟩
  include *state_monad_syntax* **and** *app_syntax*
  ⟨*proof*⟩

theorem *bellman_ford_correct*:
  fst (*run_state bellman_ford Mapping.empty*) =
  (*if has_negative_cycle then None else Some* (*map shortest* [0..<*n*+1]))
  ⟨*proof*⟩

end

end

end

3.2.7 Extracting an Executable Constant for the Imperative Implementation

ground_function (*prove_termination*) $bf_h{}'$*_impl*: $bf_h{}'$*.simps*

lemma $bf_h{}'$*_impl_def*:
  fixes *n* :: *nat*
  fixes *mem* :: *nat ref* × *nat ref* × *int extended option array ref* × *int extended option array ref*
  assumes *mem_is_init*: *mem* = *result_of* (*init_state* (*n* + 1) 1 0) *Heap.empty*
  shows $bf_h{}'$*_impl n w t mem* = $bf_h{}'$ *n w t mem*
⟨*proof*⟩

definition
  iter_bf_heap n w t mem = *iterator_defs.iter_heap*
    $(\lambda(x,\ y).\ x \leq n \land y \leq n)$
    $(\lambda(x,\ y).$ *if* $y < n$ *then* $(x,\ y + 1)$ *else* $(x + 1,\ 0))$
    $(\lambda(x,\ y).$ $bf_h{}'$*_impl n w t mem x y*)

lemma *iter_bf_heap_unfold*[*code*]:
  iter_bf_heap n w t mem = (λ (*i*, *j*).
    (*if* $i \leq n \land j \leq n$
     then do {
          $bf_h{}'$*_impl n w t mem i j*;
          iter_bf_heap n w t mem (*if* $j < n$ *then* $(i,\ j + 1)$ *else* $(i + 1,\ 0)$)
        }
     else Heap_Monad.return ()))
  ⟨*proof*⟩

definition
$bf\_impl\ n\ w\ t\ i\ j = do\ \{$
$mem \leftarrow (init\_state\ (n + 1)\ (1{::}nat)\ (0{::}nat) ::$
$(nat\ ref \times nat\ ref \times int\ extended\ option\ array\ ref \times int\ extended\ option\ array\ ref)\ Heap);$
$iter\_bf\_heap\ n\ w\ t\ mem\ (0, 0);$
$bf_h{}'\_impl\ n\ w\ t\ mem\ i\ j$
$\}$

lemma *bf_impl_correct*:
$bf\ n\ w\ t\ i\ j = result\_of\ (bf\_impl\ n\ w\ t\ i\ j)\ Heap.empty$
$\langle proof \rangle$

3.2.8 Test Cases

definition
$G_1\_list = [[(1 :: nat, -6 :: int), (2,4), (3,5)], [(3,10)], [(3,2)], []]$

definition
$G_2\_list = [[(1 :: nat, -6 :: int), (2,4), (3,5)], [(3,10)], [(3,2)], [(0, -5)]]$

definition
$G_3\_list = [[(1 :: nat, -1 :: int), (2,2)], [(2,5), (3,4)], [(3,2), (4,3)], [(2,-2), (4,2)], []]$

definition
$G_4\_list = [[(1 :: nat, -1 :: int), (2,2)], [(2,5), (3,4)], [(3,2), (4,3)], [(2,-3), (4,2)], []]$

definition
$graph\_of\ a\ i\ j = case\_option\ \infty\ (Fin\ o\ snd)\ (List.find\ (\lambda\ p.\ fst\ p = j)\ (a\ !!\ i))$

definition $test\_bf = bf\_impl\ 3\ (graph\_of\ (IArray\ G_1\_list))\ 3\ 3\ 0$

code_reflect *Test* **functions** *test_bf*

One can see a trace of the calls to the memory in the output

$\langle ML \rangle$

lemma *bottom_up_alt*[*code*]:
$bf\ n\ W\ t\ i\ j =$
$fst\ (run\_state$
$(iter\_bf\ n\ W\ t\ (0, 0) \gg= (\lambda\_.\ bf_m{}'\ n\ W\ t\ i\ j))$

Mapping.empty)
⟨*proof*⟩

definition
$bf\_ia\ n\ W\ t\ i\ j = (let\ W' = graph\_of\ (IArray\ W)\ in$
$fst\ (run\_state$
$(iter\_bf\ n\ W'\ t\ (i, j) \gg= (\lambda\_.\ bf_m{}'\ n\ W'\ t\ i\ j))$
$Mapping.empty)$
)

— Component tests.
lemma
$fst\ (run\_state\ (bf_m{}'\ 3\ (graph\_of\ (IArray\ G_1\_list))\ 3\ 3\ 0)\ Mapping.empty) = 4$
$bf\ 3\ (graph\_of\ (IArray\ G_1\_list))\ 3\ 3\ 0 = 4$
⟨*proof*⟩
lemma
$fst\ (run\_state\ (bellman\_ford\ 3\ (graph\_of\ (IArray\ G_1\_list))\ 3)\ Mapping.empty) = Some\ [4, 10, 2, 0]$
$fst\ (run\_state\ (bellman\_ford\ 4\ (graph\_of\ (IArray\ G_3\_list))\ 4)\ Mapping.empty) = Some\ [4, 5, 3, 1, 0]$
⟨*proof*⟩
lemma
$fst\ (run\_state\ (bellman\_ford\ 3\ (graph\_of\ (IArray\ G_2\_list))\ 3)\ Mapping.empty) = None$
$fst\ (run\_state\ (bellman\_ford\ 4\ (graph\_of\ (IArray\ G_4\_list))\ 4)\ Mapping.empty) = None$
⟨*proof*⟩

end
theory *Heap_Default*
imports
Heap_Main
../Indexing
begin

locale *dp_consistency_heap_default* =
fixes $bound :: 'k :: \{index, heap\}\ bound$
and $mem :: 'v::heap\ option\ array$
and $dp :: 'k \Rightarrow 'v$
begin

interpretation *idx*: *bounded_index bound* ⟨*proof*⟩

sublocale *dp_consistency_heap*
  where $P{=}\lambda heap.$ *Array.length heap mem* = *idx.size*
    and *lookup=mem_lookup idx.size idx.checked_idx mem*
    and *update=mem_update idx.size idx.checked_idx mem*
  ⟨*proof*⟩

context
  fixes *empty*
  assumes *empty*: *map_of_heap empty* $\subseteq_m$ *Map.empty*
    and *len*: *Array.length empty mem* = *idx.size*
begin

interpretation *consistent*: *dp_consistency_heap_empty*
  where $P{=}\lambda heap.$ *Array.length heap mem* = *idx.size*
    and *lookup=mem_lookup idx.size idx.checked_idx mem*
    and *update=mem_update idx.size idx.checked_idx mem*
  ⟨*proof*⟩

lemmas *memoizedI* = *consistent.memoized*
lemmas *successI* = *consistent.memoized_success*

end

lemma *mem_empty_empty*:
  map_of_heap (*heap_of* (*mem_empty idx.size* :: *'v option array Heap*) *Heap.empty*) $\subseteq_m$ *Map.empty*
  if *mem* = *result_of* (*mem_empty idx.size*) *Heap.empty*
  ⟨*proof*⟩

lemma *memoized_empty*:
  dp x = *result_of* ((*mem_empty idx.size* :: *'v option array Heap*) $\gg=$ ($\lambda mem.$ dp_T *mem x*)) *Heap.empty*
  if *consistentDP* (dp_T *mem*) *mem* = *result_of* (*mem_empty idx.size*) *Heap.empty*
  ⟨*proof*⟩

lemma *init_success*:
  success ((*mem_empty idx.size* :: *'v option array Heap*) $\gg=$ ($\lambda mem.$ dp_T *mem x*)) *Heap.empty*
  if *consistentDP* (dp_T *mem*) *mem* = *result_of* (*mem_empty idx.size*) *Heap.empty*
  ⟨*proof*⟩

end

end

3.3 The Knapsack Problem

theory *Knapsack*
  imports
    HOL−Library.Code_Target_Numeral
    ../state_monad/State_Main
    ../heap_monad/Heap_Default
    Example_Misc
begin

3.3.1 Definitions

context
  fixes $w :: nat \Rightarrow nat$
begin

context
  fixes $v :: nat \Rightarrow nat$
begin

fun *knapsack* $:: nat \Rightarrow nat \Rightarrow nat$ **where**
  $knapsack\ 0\ \ W = 0\ |$
  $knapsack\ (Suc\ i)\ \ W = (if\ W < w\ (Suc\ i)$
    $then\ knapsack\ i\ W$
    $else\ max\ (knapsack\ i\ W)\ (v\ (Suc\ i) + knapsack\ i\ (W - w\ (Suc\ i))))$

no_notation *fun_app_lifted* (**infixl** ‹.› 999)

The correctness proof closely follows Kleinberg & Tardos: "Algorithm Design", chapter "Dynamic Programming" [1]

definition
  $OPT\ n\ W = Max\ \{\sum\ i \in S.\ v\ i\ |\ S.\ S \subseteq \{1..n\} \wedge (\sum\ i \in S.\ w\ i) \leq W\}$

lemma *OPT_0*:
  $OPT\ 0\ \ W = 0$
  ⟨*proof*⟩

3.3.2 Functional Correctness

lemma *Max_add_left*:
  $(x :: nat) + Max\ S = Max\ (((+)\ x)\ `\ S)$ (**is** $?A = ?B$) **if** $finite\ S\ S \neq \{\}$

⟨*proof*⟩

lemma *OPT_Suc*:
  $OPT\ (Suc\ i)\ W = ($
    if $W < w\ (Suc\ i)$
    then $OPT\ i\ W$
    else $max(v\ (Suc\ i) + OPT\ i\ (W - w\ (Suc\ i)))\ (OPT\ i\ W)$
) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

theorem *knapsack_correct*:
  $OPT\ n\ W = knapsack\ n\ W$
  ⟨*proof*⟩

3.3.3 Functional Memoization

memoize_fun $knapsack_m$: *knapsack* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *knapsack.simps*

Generated Definitions

context includes *state_monad_syntax* **begin**
thm $knapsack_m'$*.simps* $knapsack_m$*_def*
end

Correspondence Proof

memoize_correct
  ⟨*proof*⟩
print_theorems
lemmas [*code*] = $knapsack_m$*.memoized_correct*

3.3.4 Imperative Memoization

context fixes
  mem :: *nat option array*
  and *n W* :: *nat*
begin

memoize_fun $knapsack_T$: *knapsack*
  with_memory *dp_consistency_heap_default* **where** *bound* = *Bound*
(0, 0) (*n*, *W*) **and** *mem*=*mem*
  monadifies (*heap*) *knapsack.simps*

context includes *heap_monad_syntax* **begin**
thm $knapsack_T'$*.simps* $knapsack_T$*_def*

end

memoize_correct
  ⟨*proof*⟩

lemmas *memoized_empty* = $knapsack_T$.*memoized_empty*

end

Adding Memory Initialization

context
  includes *heap_monad_syntax*
  notes [*simp del*] = $knapsack_T$*′.simps*
begin

definition
  $knapsack_h$ ≡ λ *i j*. *Heap_Monad.bind* (*mem_empty* (*i* ∗ *j*)) (λ *mem*. $knapsack_T$*′ mem i j i j*)

lemmas *memoized_empty′* = *memoized_empty*[
    of mem n W λ *m*. λ(*i*,*j*). $knapsack_T$*′ m n W i j*,
    OF $knapsack_T$.*crel*[*of mem n W*], *of* (*n*, *W*) **for** *mem n W*
  ]

lemma *knapsack_heap*:
  knapsack n W = *result_of* ($knapsack_h$ *n W*) *Heap.empty*
  ⟨*proof*⟩

end

end

fun *su* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  su 0 *W* = 0 |
  su (*Suc i*) *W* = (*if W* < *w* (*Suc i*)
    then su i W
    else max (*su i W*) (*w* (*Suc i*) + *su i* (*W* − *w* (*Suc i*))))

lemma *su_knapsack*:
  su n W = *knapsack w n W*
  ⟨*proof*⟩

lemma *su_correct*:
  Max {$\sum$ *i* ∈ *S*. *w i* | *S*. *S* ⊆ {1..*n*} ∧ ($\sum$ *i* ∈ *S*. *w i*) ≤ *W*} = *su n W*

⟨*proof*⟩

3.3.5 Memoization

memoize_fun su_m: *su* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *su.simps*

Generated Definitions

context includes *state_monad_syntax* **begin**
thm su_m*'.simps* su_m*_def*
end

Correspondence Proof

memoize_correct
⟨*proof*⟩
print_theorems
lemmas [*code*] = su_m*.memoized_correct*

end

3.3.6 Regression Test

definition
knapsack_test = ($knapsack_h$ (λ *i*. [2,3,4] ! (*i* − 1)) (λ *i*. [2,3,4] ! (*i* − 1)) 3 8)

code_reflect *Test* **functions** *knapsack_test*

⟨*ML*⟩

end
theory *Counting_Tiles*
imports
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../state_monad/State_Main
Example_Misc
begin

3.4 A Counting Problem

This formalization contains verified solutions for Project Euler problems

- #114 (`https://projecteuler.net/problem=114`) and

- #115 (`https://projecteuler.net/problem=115`).

This is the problem description for #115:

> A row measuring n units in length has red blocks with a minimum length of m units placed on it, such that any two red blocks (which are allowed to be different lengths) are separated by at least one black square. Let the fill-count function, F(m, n), represent the number of ways that a row can be filled.
>
> For example, F(3, 29) = 673135 and F(3, 30) = 1089155.
>
> That is, for m = 3, it can be seen that n = 30 is the smallest value for which the fill-count function first exceeds one million. In the same way, for m = 10, it can be verified that F(10, 56) = 880711 and F(10, 57) = 1148904, so n = 57 is the least value for which the fill-count function first exceeds one million.
>
> For m = 50, find the least value of n for which the fill-count function first exceeds one million.

3.4.1 Misc

lemma *lists_of_len_fin*1:
  finite (*lists* $A \cap \{l.\ length\ l = n\}$) **if** *finite* A
  ⟨*proof*⟩

lemma *disjE*1:
  $A \vee B \Longrightarrow (A \Longrightarrow P) \Longrightarrow (\neg\ A \Longrightarrow B \Longrightarrow P) \Longrightarrow P$
  ⟨*proof*⟩

3.4.2 Problem Specification

Colors

datatype *color* = R | B

Direct natural definition of a valid line

context
  fixes *m* :: *nat*
begin

inductive *valid* **where**
  valid [] |
  valid xs $\Longrightarrow$ *valid* (B # *xs*) |

$valid\ xs \Longrightarrow n \geq m \Longrightarrow valid\ (replicate\ n\ R\ @\ xs)$

Definition of the fill-count function

definition $F\ n = card\ \{l.\ length\ l = n \wedge valid\ l\}$

3.4.3 Combinatorial Identities

This alternative variant helps us to prove the split lemma below.

inductive *valid′* **where**
$valid'\ []\ |$
$n \geq m \Longrightarrow valid'\ (replicate\ n\ R)\ |$
$valid'\ xs \Longrightarrow valid'\ (B\ \#\ xs)\ |$
$valid'\ xs \Longrightarrow n \geq m \Longrightarrow valid'\ (replicate\ n\ R\ @\ B\ \#\ xs)$

lemma *valid_valid′*:
$valid\ l \Longrightarrow valid'\ l$
⟨*proof*⟩

lemmas *valid_red* = *valid.intros*(3)[*OF valid.intros*(1), *simplified*]

lemma *valid′_valid*:
$valid'\ l \Longrightarrow valid\ l$
⟨*proof*⟩

lemma *valid_eq_valid′*:
$valid'\ l = valid\ l$
⟨*proof*⟩

Additional Facts on Replicate

lemma *replicate_iff*:
$(\forall\, i{<}length\ l.\ l\ !\ i = R) \longleftrightarrow (\exists\ n.\ l = replicate\ n\ R)$
⟨*proof*⟩

lemma *replicate_iff2*:
$(\forall\, i{<}n.\ l\ !\ i = R) \longleftrightarrow (\exists\ l'.\ l = replicate\ n\ R\ @\ l')$ **if** $n < length\ l$
⟨*proof*⟩

lemma *replicate_Cons_eq*:
$replicate\ n\ x = y\ \#\ ys \longleftrightarrow (\exists\ n'.\ n = Suc\ n' \wedge x = y \wedge replicate\ n'\ x = ys)$
⟨*proof*⟩

Main Case Analysis on @*term valid*

lemma *valid_split*:

$valid\ l \longleftrightarrow$
$l = [] \lor$
$(l!0 = B \land valid\ (tl\ l)) \lor$
$length\ l \geq m \land (\forall\ i < length\ l.\ l\ !\ i = R) \lor$
$(\exists\ j < length\ l.\ j \geq m \land (\forall\ i < j.\ l\ !\ i = R) \land l\ !\ j = B \land valid\ (drop\ (j + 1)\ l))$
⟨*proof*⟩

Base cases

lemma *valid_line_just_B*:
$valid\ (replicate\ n\ B)$
⟨*proof*⟩

lemma *F_base_0_aux*:
$\{l.\ l = [] \land valid\ l\} = \{[]\}$
⟨*proof*⟩

lemma *F_base_0*: $F\ 0 = 1$
⟨*proof*⟩

lemma *F_base_aux*: $\{l.\ length\ l{=}n \land valid\ l\} = \{replicate\ n\ B\}$ **if** $n > 0$ $n < m$
⟨*proof*⟩

lemma *F_base_1*:
$F\ n = 1$ **if** $n > 0$ $n < m$
⟨*proof*⟩

lemma *valid_m_Rs* [*simp*]:
$valid\ (replicate\ m\ R)$
⟨*proof*⟩

lemma *F_base_aux_2*: $\{l.\ length\ l{=}m \land valid\ l\} = \{replicate\ m\ R,\ replicate\ m\ B\}$
⟨*proof*⟩

lemma *F_base_2*:
$F\ m = 2$ **if** $0 < m$
⟨*proof*⟩

The recursion case

lemma *finite_valid_length*:
$finite\ \{l.\ length\ l = n \land valid\ l\}$ (**is** *finite ?S*)
⟨*proof*⟩

lemma *valid_line_aux*:
  $\{l.\ length\ l = n \wedge valid\ l\} \neq \{\}$ (**is** $?S \neq \{\}$)
  $\langle proof \rangle$

lemma *replicate_unequal_aux*:
  $replicate\ x\ R\ @\ B\ \#\ l \neq replicate\ y\ R\ @\ B\ \#\ l'$ (**is** $?l \neq ?r$) **if** ‹$x < y$›
for $l\ l'$
$\langle proof \rangle$

lemma *valid_prepend_B_iff*:
  $valid\ (B\ \#\ xs) \longleftrightarrow valid\ xs$ **if** $m > 0$
  $\langle proof \rangle$

lemma *F_rec*: $F\ n = F\ (n-1) + 1 + (\sum i{=}m..{<}n.\ F\ (n-i-1))$ **if** ‹$n>m$›
$m > 0$
$\langle proof \rangle$

3.4.4 Computing the Fill-Count Function

fun *lcount* :: $nat \Rightarrow nat$ **where**
  $lcount\ n = ($
    if $n < m$ *then* 1
    else if $n = m$ *then* 2
    else $lcount\ (n - 1) + 1 + (\sum i \leftarrow [m..{<}n].\ lcount\ (n - i - 1))$
  $)$

lemmas [*simp del*] = *lcount.simps*

lemma *lcount_correct*:
  $lcount\ n = F\ n$ **if** $m > 0$
$\langle proof \rangle$

3.4.5 Memoization

memoize_fun $lcount_m$: *lcount* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *lcount.simps*

memoize_correct
  $\langle proof \rangle$

lemmas [*code*] = $lcount_m$*.memoized_correct*

end

3.4.6 Problem solutions

Example and solution for problem #114

value *lcount* 3 7
value *lcount* 3 50

Examples for problem #115

value *lcount* 3 29
value *lcount* 3 30
value *lcount* 10 56
value *lcount* 10 57

Binary search for the solution of problem #115

value *lcount* 50 100
value *lcount* 50 150
value *lcount* 50 163
value *lcount* 50 166
value *lcount* 50 167
value *lcount* 50 168 — The solution
value *lcount* 50 169
value *lcount* 50 175
value *lcount* 50 200
value *lcount* 50 300
value *lcount* 50 500
value *lcount* 50 1000

We prove that 168 is the solution for problem #115

theorem
  $(LEAST\ n.\ F\ 50\ n > 1000000) = 168$
⟨*proof*⟩

end

3.5 The CYK Algorithm

theory *CYK*
imports
  HOL−Library.IArray
  HOL−Library.Code_Target_Numeral
  HOL−Library.Product_Lexorder
  HOL−Library.RBT_Mapping
  ../state_monad/State_Main
  ../heap_monad/Heap_Default
  Example_Misc
begin

3.5.1 Misc

lemma *append_iff_take_drop*:
$w = u@v \longleftrightarrow (\exists\, k \in \{0..length\ w\}.\ u = take\ k\ w \wedge v = drop\ k\ w)$
$\langle proof \rangle$

lemma *append_iff_take_drop1*: $u \neq [] \Longrightarrow v \neq [] \Longrightarrow$
$w = u@v \longleftrightarrow (\exists\, k \in \{1..length\ w - 1\}.\ u = take\ k\ w \wedge v = drop\ k\ w)$
$\langle proof \rangle$

3.5.2 Definitions

datatype $('n, 't)\ rhs = NN\ 'n\ 'n \mid T\ 't$

type_synonym $('n, 't)\ prods = ('n \times ('n, 't)\ rhs)\ list$

context
fixes $P :: ('n :: heap, 't)\ prods$
begin

inductive $yield :: 'n \Rightarrow 't\ list \Rightarrow bool$ **where**
$(A,\ T\ a) \in set\ P \Longrightarrow yield\ A\ [a] \mid$
$[\![\ (A,\ NN\ B\ C) \in set\ P;\ yield\ B\ u;\ yield\ C\ v\ ]\!] \Longrightarrow yield\ A\ (u@v)$

lemma *yield_not_Nil*: $yield\ A\ w \Longrightarrow w \neq []$
$\langle proof \rangle$

lemma *yield_eq1*:
$yield\ A\ [a] \longleftrightarrow (A,\ T\ a) \in set\ P$ (**is** $?L = ?R$)
$\langle proof \rangle$

lemma *yield_eq2*: **assumes** $length\ w > 1$
shows $yield\ A\ w \longleftrightarrow (\exists\, B\ u\ C\ v.\ yield\ B\ u \wedge yield\ C\ v \wedge w = u@v \wedge (A,$
$NN\ B\ C) \in set\ P)$
(**is** $?L = ?R$)
$\langle proof \rangle$

3.5.3 CYK on Lists

fun $cyk :: 't\ list \Rightarrow 'n\ list$ **where**
$cyk\ [] = [] \mid$
$cyk\ [a] = [A\ .\ (A,\ T\ a') <- P,\ a'= a] \mid$
$cyk\ w =$
$[A.\ k <- [1..<length\ w],\ B <- cyk\ (take\ k\ w),\ C <- cyk\ (drop\ k\ w),\ (A,$
$NN\ B'\ C') <- P,\ B' = B,\ C' = C]$

lemma *set_cyk_simp2[simp]*: *length w* $\geq 2 \Longrightarrow$ *set(cyk w)* =
($\bigcup k \in \{1..length\ w - 1\}.\ \bigcup B \in set(cyk\ (take\ k\ w)).\ \bigcup C \in set(cyk\ (drop\ k\ w)).\ \{A.\ (A,\ NN\ B\ C) \in set\ P\}$)
⟨*proof*⟩

declare *cyk.simps(3)[simp del]*

lemma *cyk_correct*: $set(cyk\ w) = \{N.\ yield\ N\ w\}$
⟨*proof*⟩

3.5.4 CYK on Lists and Index

fun *cyk2* :: *'t list* ⇒ *nat* ∗ *nat* ⇒ *'n list* **where**
cyk2 w (i,0) = [] |
cyk2 w (i,Suc 0) = [*A* . (*A*, *T a*) <− *P*, *a* = *w*!*i*] |
cyk2 w (i,n) =
[*A*. *k* <− [1..<*n*], *B* <− *cyk2 w (i,k)*, *C* <− *cyk2 w (i+k,n−k)*, (*A*, *NN B' C'*) <− *P*, *B'* = *B*, *C'* = *C*]

lemma *set_aux*: ($\bigcup xb \in set\ P.\ \{A.\ (A,\ NN\ B\ C) = xb\}) = \{A.\ (A,\ NN\ B\ C) \in set\ P\}$
⟨*proof*⟩

lemma *cyk2_eq_cyk*: $i+n \leq length\ w \Longrightarrow set(cyk2\ w\ (i,n)) = set(cyk\ (take\ n\ (drop\ i\ w)))$
⟨*proof*⟩

definition *CYK S w* = ($S \in set(cyk2\ w\ (0,\ length\ w))$)

theorem *CYK_correct*: *CYK S w* = *yield S w*
⟨*proof*⟩

3.5.5 CYK With Index Function

context
fixes *w* :: *nat* ⇒ *'t*
begin

fun *cyk_ix* :: *nat* ∗ *nat* ⇒ *'n list* **where**
cyk_ix (i,0) = [] |
cyk_ix (i,Suc 0) = [*A* . (*A*, *T a*) <− *P*, *a* = *w i*] |
cyk_ix (i,n) =
[*A*. *k* <− [1..<*n*], *B* <− *cyk_ix (i,k)*, *C* <− *cyk_ix (i+k,n−k)*, (*A*, *NN*

B' C') <− P, $B' = B$, $C' = C$]

3.5.6 Correctness Proof

lemma *cyk_ix_simp2*: $set(cyk\_ix\ (i,Suc(Suc\ n))) =$
$(\bigcup k \in \{1..Suc\ n\}.\ \bigcup B \in set(cyk\_ix\ (i,k)).\ \bigcup C \in set(cyk\_ix\ (i+k,n+2-k)).$
$\{A.\ (A,\ NN\ B\ C) \in set\ P\})$
⟨*proof*⟩

declare *cyk_ix.simps*(3)[*simp del*]

abbreviation (*input*) *slice f i j* ≡ *map f* [*i*..<*j*]

lemma *slice_append_iff_take_drop1*: $u \neq [] \Longrightarrow v \neq [] \Longrightarrow$
$slice\ w\ i\ j = u\ @\ v \longleftrightarrow (\exists k.\ 1 \leq k \wedge k \leq j{-}i{-}1 \wedge slice\ w\ i\ (i + k) = u$
$\wedge\ slice\ w\ (i + k)\ j = v)$
⟨*proof*⟩

lemma *cyk_ix_correct*:
$set(cyk\_ix\ (i,n)) = \{N.\ yield\ N\ (slice\ w\ i\ (i+n))\}$
⟨*proof*⟩

3.5.7 Functional Memoization

memoize_fun cyk_ix_m: *cyk_ix* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *cyk_ix.simps*
thm $cyk\_ix_m{}'.simps$

memoize_correct
⟨*proof*⟩
print_theorems

lemmas [*code*] = $cyk\_ix_m.memoized\_correct$

3.5.8 Imperative Memoization

context
fixes *n* :: *nat*
begin

context
fixes *mem* :: *'n list option array*
begin

memoize_fun cyk_ix_h: *cyk_ix*

with_memory *dp_consistency_heap_default* **where** *bound* = *Bound* (0, 0) (*n*, *n*) **and** *mem*=*mem*
monadifies (*heap*) *cyk_ix.simps*

context includes *heap_monad_syntax* **begin**
thm $cyk\_ix_h{}'$*.simps* $cyk\_ix_h\_def$
end

memoize_correct
⟨*proof*⟩

lemmas *memoized_empty* = cyk_ix_h*.memoized_empty*

lemmas *init_success* = cyk_ix_h*.init_success*

end

definition *cyk_ix_impl i j* = *do* {*mem* ← *mem_empty* $(n * n)$; $cyk\_ix_h{}'$ *mem* (*i*, *j*)}

lemma *cyk_ix_impl_success*:
success (*cyk_ix_impl i j*) *Heap.empty*
⟨*proof*⟩

lemma *min_wpl_heap*:
cyk_ix (*i*, *j*) = *result_of* (*cyk_ix_impl i j*) *Heap.empty*
⟨*proof*⟩

end

end

definition *CYK_ix S w n* = (*S* ∈ *set*(*cyk_ix w* (0,*n*)))

theorem *CYK_ix_correct*: *CYK_ix S w n* = *yield S* (*slice w* 0 *n*)
⟨*proof*⟩

definition *cyk_list w* = *cyk_ix* (λ*i*. *w* ! *i*) (0,*length w*)

definition
CYK_ix_impl S w n = *do* {*R* ← *cyk_ix_impl w n* 0 *n*; *return* (*S* ∈ *set R*)}

lemma *CYK_ix_impl_correct*:

$result\_of\ (CYK\_ix\_impl\ S\ w\ n)\ Heap.empty = yield\ S\ (slice\ w\ 0\ n)$
$\langle proof \rangle$

end

3.5.9 Functional Test Case

value
$(let\ P = [(0::int,\ NN\ 1\ 2),\ (0,\ NN\ 2\ 3),$
$(1,\ NN\ 2\ 1),\ (1,\ T\ (CHR\ ''a'')),$
$(2,\ NN\ 3\ 3),\ (2,\ T\ (CHR\ ''b'')),$
$(3,\ NN\ 1\ 2),\ (3,\ T\ (CHR\ ''a''))]$
$in\ map\ (\lambda w.\ cyk2\ P\ w\ (0,length\ w))\ [''baaba'',\ ''baba''])$

value
$(let\ P = [(0::int,\ NN\ 1\ 2),\ (0,\ NN\ 2\ 3),$
$(1,\ NN\ 2\ 1),\ (1,\ T\ (CHR\ ''a'')),$
$(2,\ NN\ 3\ 3),\ (2,\ T\ (CHR\ ''b'')),$
$(3,\ NN\ 1\ 2),\ (3,\ T\ (CHR\ ''a''))]$
$in\ map\ (cyk\_list\ P)\ [''baaba'',\ ''baba''])$

definition $cyk\_ia\ P\ w = (let\ a = IArray\ w\ in\ cyk\_ix\ P\ (\lambda i.\ a\ !!\ i)\ (0,length\ w))$

value
$(let\ P = [(0::int,\ NN\ 1\ 2),\ (0,\ NN\ 2\ 3),$
$(1,\ NN\ 2\ 1),\ (1,\ T\ (CHR\ ''a'')),$
$(2,\ NN\ 3\ 3),\ (2,\ T\ (CHR\ ''b'')),$
$(3,\ NN\ 1\ 2),\ (3,\ T\ (CHR\ ''a''))]$
$in\ map\ (cyk\_ia\ P)\ [''baaba'',\ ''baba''])$

3.5.10 Imperative Test Case

definition $cyk\_ia'\ P\ w = (let\ a = IArray\ w\ in\ cyk\_ix\_impl\ P\ (\lambda i.\ a\ !!\ i)\ (length\ w)\ 0\ (length\ w))$

definition
$test = (let\ P = [(0::int,\ NN\ 1\ 2),\ (0,\ NN\ 2\ 3),$
$(1,\ NN\ 2\ 1),\ (1,\ T\ (CHR\ ''a'')),$
$(2,\ NN\ 3\ 3),\ (2,\ T\ (CHR\ ''b'')),$
$(3,\ NN\ 1\ 2),\ (3,\ T\ (CHR\ ''a''))]$
$in\ map\ (cyk\_ia'\ P)\ [''baaba'',\ ''baba''])$

code_reflect *Test* **functions** *test*

⟨*ML*⟩

end

3.6 Minimum Edit Distance

theory *Min_Ed_Dist*0
imports
HOL−Library.IArray
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../state_monad/State_Main
../heap_monad/Heap_Main
Example_Misc
../util/Tracing
../util/Ground_Function
begin

3.6.1 Misc

Executable argmin

fun *argmin* :: ($'a \Rightarrow 'b$::*order*) $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ **where**
argmin f [*a*] = *a* |
argmin f (*a*#*as*) = (*let m* = *argmin f as in if f a* $\leq$ *f m then a else m*)

fun *argmin*2 :: ($'a \Rightarrow 'b$::*order*) $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a * 'b$ **where**
*argmin*2 *f* [*a*] = (*a*, *f a*) |
*argmin*2 *f* (*a*#*as*) = (*let fa* = *f a*; (*am*,*m*) = *argmin*2 *f as in if fa* $\leq$ *m then* (*a*, *fa*) *else* (*am*,*m*))

3.6.2 Edit Distance

datatype $'a$ *ed* = *Copy* | *Repl* $'a$ | *Ins* $'a$ | *Del*

fun *edit* :: $'a$ *ed list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* **where**
edit (*Copy* # *es*) (*x* # *xs*) = *x* # *edit es xs* |
edit (*Repl a* # *es*) (*x* # *xs*) = *a* # *edit es xs* |
edit (*Ins a* # *es*) *xs* = *a* # *edit es xs* |
edit (*Del* # *es*) (*x* # *xs*) = *edit es xs* |
edit (*Copy* # *es*) [] = *edit es* [] |

$edit\ (Repl\ a\ \#\ es)\ []\ =\ edit\ es\ []\ |$
$edit\ (Del\ \#\ es)\ []\ =\ edit\ es\ []\ |$
$edit\ []\ xs\ =\ xs$

abbreviation *cost* **where**
$cost\ es \equiv length\ [e <- es.\ e \neq Copy]$

3.6.3 Minimum Edit Sequence

fun $min\_eds :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ ed\ list$ **where**
$min\_eds\ []\ []\ =\ []\ |$
$min\_eds\ []\ (y\#ys)\ =\ Ins\ y\ \#\ min\_eds\ []\ ys\ |$
$min\_eds\ (x\#xs)\ []\ =\ Del\ \#\ min\_eds\ xs\ []\ |$
$min\_eds\ (x\#xs)\ (y\#ys)\ =$
  $argmin\ cost\ [Ins\ y\ \#\ min\_eds\ (x\#xs)\ ys,\ Del\ \#\ min\_eds\ xs\ (y\#ys),$
    $(if\ x=y\ then\ Copy\ else\ Repl\ y)\ \#\ min\_eds\ xs\ ys]$

lemma $min\_eds\ ''vintner''\ ''writers'' =$
  $[Ins\ CHR\ ''w'',\ Repl\ CHR\ ''r'',\ Copy,\ Del,\ Copy,\ Del,\ Copy,\ Copy,\ Ins\ CHR\ ''s'']$
$\langle proof \rangle$

lemma $min\_eds\_correct$: $edit\ (min\_eds\ xs\ ys)\ xs\ =\ ys$
$\langle proof \rangle$

lemma min_eds_same: $min\_eds\ xs\ xs\ =\ replicate\ (length\ xs)\ Copy$
$\langle proof \rangle$

lemma $min\_eds\_eq\_Nil\_iff$: $min\_eds\ xs\ ys\ =\ []\ \longleftrightarrow\ xs\ =\ []\ \wedge\ ys\ =\ []$
$\langle proof \rangle$

lemma min_eds_Nil: $min\_eds\ []\ ys\ =\ map\ Ins\ ys$
$\langle proof \rangle$

lemma min_eds_Nil2: $min\_eds\ xs\ []\ =\ replicate\ (length\ xs)\ Del$
$\langle proof \rangle$

lemma if_edit_Nil2: $edit\ es\ ([]::'a\ list)\ =\ ys \Longrightarrow length\ ys \leq cost\ es$
$\langle proof \rangle$

lemma $if\_edit\_eq\_Nil$: $edit\ es\ xs\ =\ [] \Longrightarrow length\ xs \leq cost\ es$
$\langle proof \rangle$

lemma $min\_eds\_minimal$: $edit\ es\ xs\ =\ ys \Longrightarrow cost(min\_eds\ xs\ ys) \leq cost$

es
⟨*proof*⟩

3.6.4 Computing the Minimum Edit Distance

fun *min_ed* :: *'a list* ⇒ *'a list* ⇒ *nat* **where**
min_ed [] [] = 0 |
min_ed [] (*y*#*ys*) = 1 + *min_ed* [] *ys* |
min_ed (*x*#*xs*) [] = 1 + *min_ed xs* [] |
min_ed (*x*#*xs*) (*y*#*ys*) =
  Min {1 + *min_ed* (*x*#*xs*) *ys*, 1 + *min_ed xs* (*y*#*ys*), (*if x*=*y then* 0 *else* 1) + *min_ed xs ys*}

lemma *min_ed_min_eds*: *min_ed xs ys* = *cost*(*min_eds xs ys*)
⟨*proof*⟩

lemma *min_ed* ''*madagascar*'' ''*bananas*'' = 6
⟨*proof*⟩

Exercise: Optimization of the Copy case

fun *min_eds*2 :: *'a list* ⇒ *'a list* ⇒ *'a ed list* **where**
*min_eds*2 [] [] = [] |
*min_eds*2 [] (*y*#*ys*) = *Ins y* # *min_eds*2 [] *ys* |
*min_eds*2 (*x*#*xs*) [] = *Del* # *min_eds*2 *xs* [] |
*min_eds*2 (*x*#*xs*) (*y*#*ys*) =
  (*if x*=*y then Copy* # *min_eds*2 *xs ys*
   else argmin cost
     [*Ins y* # *min_eds*2 (*x*#*xs*) *ys*, *Del* # *min_eds*2 *xs* (*y*#*ys*), *Repl y* # *min_eds*2 *xs ys*])

value *min_eds*2 ''*madagascar*'' ''*bananas*''

lemma *cost_Copy_Del*: *cost*(*min_eds xs ys*) ≤ *cost* (*min_eds xs* (*x*#*ys*)) + 1
⟨*proof*⟩

lemma *cost_Copy_Ins*: *cost*(*min_eds xs ys*) ≤ *cost* (*min_eds* (*x*#*xs*) *ys*) + 1
⟨*proof*⟩

lemma *cost*(*min_eds*2 *xs ys*) = *cost*(*min_eds xs ys*)
⟨*proof*⟩

lemma *min_eds*2 *xs ys* = *min_eds xs ys*

⟨*proof*⟩

3.6.5 Indexing

Indexing lists

context
fixes *xs ys* :: *'a list*
fixes *m n* :: *nat*
begin

function (*sequential*)
  min_ed_ix' :: *nat* ∗ *nat* ⇒ *nat* **where**
min_ed_ix' (*i*,*j*) =
  (*if* *i* ≥ *m* *then*
     if *j* ≥ *n* *then* 0 *else* 1 + *min_ed_ix'* (*i*,*j*+1) *else*
   if *j* ≥ *n* *then* 1 + *min_ed_ix'* (*i*+1, *j*)
   else
   Min {1 + *min_ed_ix'* (*i*,*j*+1), 1 + *min_ed_ix'* (*i*+1, *j*),
       (*if* *xs*!*i* = *ys*!*j* *then* 0 *else* 1) + *min_ed_ix'* (*i*+1,*j*+1)})
⟨*proof*⟩
termination ⟨*proof*⟩

declare *min_ed_ix'*.*simps*[*simp del*]

end

lemma *min_ed_ix'_min_ed*:
  min_ed_ix' *xs ys* (*length xs*) (*length ys*) (*i*, *j*) = *min_ed* (*drop i xs*) (*drop j ys*)
⟨*proof*⟩

Indexing functions

context
fixes *xs ys* :: *nat* ⇒ *'a*
fixes *m n* :: *nat*
begin

function (*sequential*)
  min_ed_ix :: *nat* × *nat* ⇒ *nat* **where**
min_ed_ix (*i*, *j*) =
  (*if* *i* ≥ *m* *then*
     if *j* ≥ *n* *then* 0 *else* *n*−*j* *else*
   if *j* ≥ *n* *then* *m*−*i*

else
min_list [1 + *min_ed_ix* (*i*, *j*+1), 1 + *min_ed_ix* (*i*+1, *j*),
(*if xs i* = *ys j then* 0 *else* 1) + *min_ed_ix* (*i*+1, *j*+1)])
⟨*proof*⟩
termination ⟨*proof*⟩

3.6.6 Functional Memoization

memoize_fun $min\_ed\_ix_m$: *min_ed_ix* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *min_ed_ix.simps*
thm $min\_ed\_ix_m'$*.simps*

memoize_correct
⟨*proof*⟩
print_theorems

lemmas [*code*] = $min\_ed\_ix_m$*.memoized_correct*

declare *min_ed_ix.simps*[*simp del*]

3.6.7 Imperative Memoization

context
fixes *mem* :: *nat ref* × *nat ref* × *nat option array ref* × *nat option array ref*
assumes *mem_is_init*: *mem* = *result_of* (*init_state* (*n* + 1) *m* (*m* + 1)) *Heap.empty*
begin

interpretation *iterator*
λ (*x*, *y*). $x \leq m \wedge y \leq n \wedge x > 0$
λ (*x*, *y*). *if* $y > 0$ *then* $(x, y - 1)$ *else* $(x - 1, n)$
λ (*x*, *y*). $(m - x) * (n + 1) + (n - y)$
⟨*proof*⟩

lemma [*intro*]:
dp_consistency_heap_array_pair' (*n* + 1) *fst snd id m* (*m* + 1) *mem*
⟨*proof*⟩

lemma [*intro*]:
dp_consistency_heap_array_pair_iterator (*n* + 1) *fst snd id m* (*m* + 1) *mem*
(λ (*x*, *y*). *if* $y > 0$ *then* $(x, y - 1)$ *else* $(x - 1, n)$)
(λ (*x*, *y*). $(m - x) * (n + 1) + (n - y)$)

$(\lambda\ (x,\ y).\ x \leq m \wedge y \leq n \wedge x > 0)$

⟨*proof*⟩

memoize_fun *min_ed_ix*$_h$: *min_ed_ix*
with_memory (**default_proof**) *dp_consistency_heap_array_pair_iterator*
where *size* = *n* + 1
and *key1*=*fst* :: *nat* × *nat* ⇒ *nat* **and** *key2*=*snd* :: *nat* × *nat* ⇒ *nat*
and *k1*=*m* :: *nat* **and** *k2*=*m* + 1 :: *nat*
and *to_index* = *id* :: *nat* ⇒ *nat*
and *mem* = *mem*
and *cnt* = $\lambda\ (x,\ y).\ x \leq m \wedge y \leq n \wedge x > 0$
and *nxt* = $\lambda\ (x{::}nat,\ y).$ *if* $y > 0$ *then* $(x,\ y - 1)$ *else* $(x - 1,\ n)$
and *sizef* = $\lambda\ (x,\ y).\ (m - x) * (n + 1) + (n - y)$
monadifies (*heap*) *min_ed_ix.simps*

memoize_correct
⟨*proof*⟩

lemmas *memoized_empty* =
min_ed_ix$_h$.*memoized_empty*[*OF* *min_ed_ix*$_h$.*consistent_DP_iter_and_compute*[*OF* *min_ed_ix*$_h$.*crel*]]
lemmas *iter_heap_unfold* = *iter_heap_unfold*

end

end

3.6.8 Test Cases

abbreviation (*input*) *slice xs i j* ≡ *map xs* [*i*..<*j*]

lemma *min_ed_Nil1*: *min_ed* [] *ys* = *length ys*
⟨*proof*⟩

lemma *min_ed_Nil2*: *min_ed xs* [] = *length xs*
⟨*proof*⟩

lemma *min_ed_ix_min_ed*: *min_ed_ix xs ys m n* (*i*,*j*) = *min_ed* (*slice xs i m*) (*slice ys j n*)
⟨*proof*⟩

Functional Test Cases

definition $min\_ed\_list\ xs\ ys = min\_ed\_ix\ (\lambda i.\ xs!i)\ (\lambda i.\ ys!i)\ (length\ xs)$ $(length\ ys)\ (0{,}0)$

lemma $min\_ed\_list\ ''madagascar''\ ''bananas'' = 6$
$\langle proof \rangle$

definition $min\_ed\_ia\ xs\ ys = (let\ a = IArray\ xs;\ b = IArray\ ys$
$in\ min\_ed\_ix\ (\lambda i.\ a!!i)\ (\lambda i.\ b!!i)\ (length\ xs)\ (length\ ys)\ (0{,}0))$

lemma $min\_ed\_ia\ ''madagascar''\ ''bananas'' = 6$
$\langle proof \rangle$

Extracting an Executable Constant for the Imperative Implementation

ground_function $min\_ed\_ix_h'\_impl$: $min\_ed\_ix_h'.simps$
termination
$\langle proof \rangle$

lemmas $[simp\ del] = min\_ed\_ix_h'\_impl.simps\ min\_ed\_ix_h'.simps$

lemma $min\_ed\_ix_h'\_impl\_def$:
includes $heap\_monad\_syntax$
fixes $m\ n :: nat$
fixes $mem :: nat\ ref \times nat\ ref \times nat\ option\ array\ ref \times nat\ option\ array\ ref$
assumes mem_is_init: $mem = result\_of\ (init\_state\ (n + 1)\ m\ (m + 1))\ Heap.empty$
shows $min\_ed\_ix_h'\_impl\ xs\ ys\ m\ n\ mem = min\_ed\_ix_h'\ xs\ ys\ m\ n\ mem$
$\langle proof \rangle$

definition
$iter\_min\_ed\_ix\ xs\ ys\ m\ n\ mem = iterator\_defs.iter\_heap$
$(\lambda\ (x,\ y).\ x \le m \land y \le n \land x > 0)$
$(\lambda\ (x,\ y).\ if\ y > 0\ then\ (x,\ y - 1)\ else\ (x - 1,\ n))$
$(min\_ed\_ix_h'\_impl\ xs\ ys\ m\ n\ mem)$

lemma $iter\_min\_ed\_ix\_unfold[code]$:
$iter\_min\_ed\_ix\ xs\ ys\ m\ n\ mem = (\lambda\ (i,\ j).$
$(if\ i > 0 \land i \le m \land j \le n$
$then\ do\ \{$
$min\_ed\_ix_h'\_impl\ xs\ ys\ m\ n\ mem\ (i,\ j);$
$iter\_min\_ed\_ix\ xs\ ys\ m\ n\ mem\ (if\ j > 0\ then\ (i,\ j - 1)\ else\ (i - 1,\ n))$
$\}$

else Heap_Monad.return ()))
⟨*proof*⟩

definition
min_ed_ix_impl xs ys m n i j = *do* {
mem ← (*init_state* (*n* + 1) (*m*::*nat*) (*m* + 1) ::
(*nat ref* × *nat ref* × *nat option array ref* × *nat option array ref*) *Heap*);
iter_min_ed_ix xs ys m n mem (*m*, *n*);
$min\_ed\_ix_h{}'\_impl$ *xs ys m n mem* (*i*, *j*)
}

lemma *bf_impl_correct*:
min_ed_ix xs ys m n (*i*, *j*) = *result_of* (*min_ed_ix_impl xs ys m n i j*) *Heap.empty*
⟨*proof*⟩

Imperative Test Case

definition
$min\_ed\_ia_h$ *xs ys* = (*let a* = *IArray xs*; *b* = *IArray ys*
in min_ed_ix_impl (λ*i*. *a*!!*i*) (λ*i*. *b*!!*i*) (*length xs*) (*length ys*) 0 0)

definition
test_case = $min\_ed\_ia_h$ ''*madagascar*'' ''*bananas*''

export_code *min_ed_ix* **in** *SML* **module_name** *Test*

code_reflect *Test* **functions** *test_case*

One can see a trace of the calls to the memory in the output

⟨*ML*⟩

end

3.7 Optimal Binary Search Trees

The material presented in this section just contains a simple and non-optimal version (cubic instead of quadratic in the number of keys). It can now be viewed to be superseded by the AFP entry *Optimal_BST*. It is kept here as a more easily understandable example and for archival purposes.

theory *OptBST*
imports
HOL−Library.Tree
HOL−Library.Code_Target_Numeral

$../state\_monad/State\_Main$
$../heap\_monad/Heap\_Default$
$Example\_Misc$
$HOL{-}Library.Product\_Lexorder$
$HOL{-}Library.RBT\_Mapping$
begin

3.7.1 Function *argmin*

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

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

Note that *arg_min_list* is similar but returns the leftmost element.

lemma $argmin\_forall$: $xs \neq [] \Longrightarrow (\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x) \Longrightarrow P\ (argmin\ f\ xs)$
$\langle proof \rangle$

lemma $argmin\_Min$: $xs \neq [] \Longrightarrow f\ (argmin\ f\ xs) = Min\ (f\ `\ set\ xs)$
$\langle proof \rangle$

3.7.2 Misc

lemma $upto\_join$: $[\![\ i \le j;\ j \le k\ ]\!] \Longrightarrow [i..j{-}1]\ @\ j\ \#\ [j{+}1..k] = [i..k]$
$\langle proof \rangle$

lemma $atLeastAtMost\_split$:
$\{i..j\} = \{i..k\} \cup \{k{+}1..j\}$ **if** $i \le k\ k \le j$ **for** $i\ j\ k :: int$
$\langle proof \rangle$

lemma $atLeastAtMost\_split\_insert$:
$\{i..k\} = insert\ k\ \{i..k{-}1\}$ **if** $k \ge i$ **for** $i :: int$
$\langle proof \rangle$

3.7.3 Definitions

context
fixes $W :: int \Rightarrow int \Rightarrow nat$
begin

fun $wpl :: int \Rightarrow int \Rightarrow int\ tree \Rightarrow nat$ **where**

$wpl\ i\ j\ Leaf = 0$
$|\ wpl\ i\ j\ (Node\ l\ k\ r) = wpl\ i\ (k{-}1)\ l + wpl\ (k{+}1)\ j\ r + W\ i\ j$

function $min\_wpl :: int \Rightarrow int \Rightarrow nat$ **where**
$min\_wpl\ i\ j =$
$(if\ i > j\ then\ 0$
$else\ min\_list\ (map\ (\lambda k.\ min\_wpl\ i\ (k{-}1) + min\_wpl\ (k{+}1)\ j + W\ i\ j)\ [i..j]))$
$\langle proof\rangle$
termination $\langle proof\rangle$
declare $min\_wpl.simps[simp\ del]$

function $opt\_bst :: int \Rightarrow int \Rightarrow int\ tree$ **where**
$opt\_bst\ i\ j =$
$(if\ i > j\ then\ Leaf\ else\ argmin\ (wpl\ i\ j)\ [\langle opt\_bst\ i\ (k{-}1),\ k,\ opt\_bst\ (k{+}1)\ j\rangle.\ k \leftarrow [i..j]])$
$\langle proof\rangle$
termination $\langle proof\rangle$
declare $opt\_bst.simps[simp\ del]$

3.7.4 Functional Memoization

context
fixes $n :: nat$
begin

context fixes
$mem :: nat\ option\ array$
begin

memoize_fun min_wpl_T: min_wpl
with_memory $dp\_consistency\_heap\_default$ **where** $bound = Bound\ (0, 0)\ (int\ n, int\ n)$ **and** $mem{=}mem$
monadifies $(heap)\ min\_wpl.simps$

context includes $heap\_monad\_syntax$ **begin**
thm $min\_wpl_T'.simps\ min\_wpl_T\_def$
end

memoize_correct
$\langle proof\rangle$

lemmas $memoized\_empty = min\_wpl_T.memoized\_empty$

end

context
includes *heap_monad_syntax*
notes [*simp del*] = $min\_wpl_T{}'.simps$
begin

definition $min\_wpl_h \equiv \lambda\ i\ j.\ Heap\_Monad.bind\ (mem\_empty\ (n * n))\ (\lambda\ mem.\ min\_wpl_T{}'\ mem\ i\ j)$

lemma *min_wpl_heap*:
$min\_wpl\ i\ j = result\_of\ (min\_wpl_h\ i\ j)\ Heap.empty$
⟨*proof*⟩

end

end

context includes *state_monad_syntax* **begin**

memoize_fun min_wpl_m: *min_wpl* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *min_wpl.simps*
thm $min\_wpl_m{}'.simps$

memoize_correct
⟨*proof*⟩
print_theorems
lemmas [*code*] = $min\_wpl_m.memoized\_correct$

memoize_fun opt_bst_m: *opt_bst* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *opt_bst.simps*
thm $opt\_bst_m{}'.simps$

memoize_correct
⟨*proof*⟩
print_theorems
lemmas [*code*] = $opt\_bst_m.memoized\_correct$

end

3.7.5 Correctness Proof

lemma *min_wpl_minimal*:
$inorder\ t = [i..j] \Longrightarrow min\_wpl\ i\ j \le wpl\ i\ j\ t$

⟨*proof*⟩

lemma *opt_bst_correct*: *inorder* (*opt_bst i j*) = [*i..j*]
⟨*proof*⟩

lemma *wpl_opt_bst*: *wpl i j* (*opt_bst i j*) = *min_wpl i j*
⟨*proof*⟩

lemma *opt_bst_is_optimal*:
inorder t = [*i..j*] ⟹ *wpl i j* (*opt_bst i j*) ≤ *wpl i j t*
⟨*proof*⟩

end

3.7.6 Access Frequencies

Usually, the problem is phrased in terms of access frequencies. We now give an interpretation of *wpl* in this view and show that we have actually computed the right thing.

context
— We are given a range [*i..j*] of integer keys with access frequencies *p*. These can be thought of as a probability distribution but are not required to represent one. This model assumes that the tree will contain all keys in the range [*i..j*]. See *Optimal_BST* for a model with missing keys.
fixes *p* :: *int* ⇒ *nat*
begin

— The *weighted path path length* (or *cost*) of a tree.
fun *cost* :: *int tree* ⇒ *nat* **where**
cost Leaf = 0
| *cost* (*Node l k r*) = *sum p* (*set_tree l*) + *cost l* + *p k* + *cost r* + *sum p* (*set_tree r*)

— Deriving a weight function from *p*.
qualified definition *W* **where**
W i j = *sum p* {*i..j*}

— We will use this later for computing *W* efficiently.
lemma *W_rec*:
W i j = (*if j* ≥ *i then W i* (*j* − 1) + *p j else* 0)
⟨*proof*⟩
lemma *inorder_wpl_correct*:
inorder t = [*i..j*] ⟹ *wpl W i j t* = *cost t*

$\langle proof \rangle$

The optimal binary search tree has minimal cost among all binary search trees.

lemma *opt_bst_has_optimal_cost*:
  $inorder\ t = [i..j] \Longrightarrow cost\ (opt\_bst\ W\ i\ j) \le cost\ t$
  $\langle proof \rangle$

The function *min_wpl* correctly computes the minimal cost among all binary search trees:

- Its cost is a lower bound for the cost of all binary search trees
- Its cost actually corresponds to an optimal binary search tree

lemma *min_wpl_minimal_cost*:
  $inorder\ t = [i..j] \Longrightarrow min\_wpl\ W\ i\ j \le cost\ t$
  $\langle proof \rangle$

lemma *min_wpl_tree*:
  $cost\ (opt\_bst\ W\ i\ j) = min\_wpl\ W\ i\ j$
  $\langle proof \rangle$

An alternative view of costs. **fun** $depth :: 'a \Rightarrow 'a\ tree \Rightarrow nat\ extended$ **where**
  $depth\ x\ Leaf = \infty$
$|\ depth\ x\ (Node\ l\ k\ r) = (if\ x = k\ then\ 1\ else\ min\ (depth\ x\ l)\ (depth\ x\ r) + 1)$

fun *the_fin* **where**
  $the\_fin\ (Fin\ x) = x \mid the\_fin\ \_ = undefined$

definition $cost' :: int\ tree \Rightarrow nat$ **where**
  $cost'\ t = sum\ (\lambda x.\ the\_fin\ (depth\ x\ t) * p\ x)\ (set\_tree\ t)$

lemma [*simp*]:
  $the\_fin\ 1 = 1$
  $\langle proof \rangle$

lemma *set_tree_depth*:
  assumes $x \notin set\_tree\ t$
  shows $depth\ x\ t = \infty$
  $\langle proof \rangle$

lemma *depth_inf_iff*:
  depth x t = ∞ ⟷ *x* ∉ *set_tree t*
  ⟨*proof*⟩

lemma *depth_not_neg_inf*[*simp*]:
  depth x t = −∞ ⟷ *False*
  ⟨*proof*⟩

lemma *depth_FinD*:
  assumes *x* ∈ *set_tree t*
  obtains *d* **where** *depth x t* = *Fin d*
  ⟨*proof*⟩

lemma *cost′_Leaf*[*simp*]:
  cost′ Leaf = 0
  ⟨*proof*⟩

lemma *cost′_Node*:
  distinct (*inorder* ⟨*l*, *x*, *r*⟩) ⟹
  cost′ ⟨*l*, *x*, *r*⟩ = *sum p* (*set_tree l*) + *cost′ l* + *p x* + *cost′ r* + *sum p* (*set_tree r*)
  ⟨*proof*⟩

lemma *weight_correct*:
  distinct (*inorder t*) ⟹ *cost′ t* = *cost t*
  ⟨*proof*⟩

3.7.7 Memoizing Weights

function *W_fun* **where**
  W_fun i j = (*if i* > *j then* 0 *else W_fun i* (*j* − 1) + *p j*)
  ⟨*proof*⟩

termination
  ⟨*proof*⟩

lemma *W_fun_correct*:
  W_fun i j = *W i j*
  ⟨*proof*⟩

memoize_fun W_m: *W_fun*
  with_memory *dp_consistency_mapping*
  monadifies (*state*) *W_fun.simps*

memoize_correct

$\langle proof \rangle$

definition
$compute\_W\ n = snd\ (run\_state\ (State\_Main.map_T{}'\ (\lambda i.\ W_m{}'\ i\ n)\ [0..n])$
$Mapping.empty)$

notation $W_m.crel\_vs$ ($\langle crel \rangle$)

lemmas $W_m\_crel = W_m.crel[unfolded\ W_m.consistentDP\_def,\ THEN\ rel\_funD,$
$of\ (m,\ x)\ (m,\ y)$ **for** $m\ x\ y,\ unfolded\ prod.case]$

lemma $compute\_W\_correct$:
assumes $Mapping.lookup\ (compute\_W\ n)\ (i,\ j) = Some\ x$
shows $W\ i\ j = x$
$\langle proof \rangle$
include $state\_monad\_syntax$ **and** app_syntax **and** $lifting\_syntax$
$\langle proof \rangle$

definition
$min\_wpl'\ i\ j \equiv$
let
$M = compute\_W\ j;$
$W = (\lambda i\ j.\ case\ Mapping.lookup\ M\ (i,\ j)\ of\ None \Rightarrow W\ i\ j\ |\ Some\ x \Rightarrow$
$x)$
$in\ min\_wpl\ W\ i\ j$

lemma $W\_compute$: $W\ i\ j = (case\ Mapping.lookup\ (compute\_W\ n)\ (i,\ j)$
$of\ None \Rightarrow W\ i\ j\ |\ Some\ x \Rightarrow x)$
$\langle proof \rangle$

lemma $min\_wpl'\_correct$:
$min\_wpl'\ i\ j = min\_wpl\ W\ i\ j$
$\langle proof \rangle$

definition
$opt\_bst'\ i\ j \equiv$
let
$M = compute\_W\ j;$
$W = (\lambda i\ j.\ case\ Mapping.lookup\ M\ (i,\ j)\ of\ None \Rightarrow W\ i\ j\ |\ Some\ x \Rightarrow$
$x)$
$in\ opt\_bst\ W\ i\ j$

lemma $opt\_bst'\_correct$:
$opt\_bst'\ i\ j = opt\_bst\ W\ i\ j$

$\langle proof \rangle$

end

3.7.8 Test Case

Functional Implementations

lemma $min\_wpl\ (\lambda i\ j.\ nat(i+j))\ 0\ 4 = 10$
  $\langle proof \rangle$

lemma $opt\_bst\ (\lambda i\ j.\ nat(i+j))\ 0\ 4 = \langle\langle\langle\langle\langle\langle\rangle, 0, \langle\rangle\rangle, 1, \langle\rangle\rangle, 2, \langle\rangle\rangle, 3, \langle\rangle\rangle, 4, \langle\rangle\rangle$
  $\langle proof \rangle$

Using Frequencies

definition
  $list\_to\_p\ xs\ (i{::}int) = (if\ i - 1 \geq 0 \wedge nat\ (i - 1) < length\ xs\ then\ xs\ !\ nat\ (i - 1)\ else\ 0)$

definition
  $ex\_p\_1 = [10, 30, 15, 25, 20]$

definition
  $opt\_tree\_1 =$
  $\langle$
    $\langle$
      $\langle\langle\rangle, 1{::}int, \langle\rangle\rangle,$
      $2,$
      $\langle\langle\rangle, 3, \langle\rangle\rangle$
    $\rangle,$
    $4,$
    $\langle\langle\rangle, 5, \langle\rangle\rangle$
  $\rangle$

lemma $opt\_bst'\ (list\_to\_p\ ex\_p\_1)\ 1\ 5 = opt\_tree\_1$
  $\langle proof \rangle$

Imperative Implementation

code_thms min_wpl

definition $min\_wpl\_test = min\_wpl_h\ (\lambda i\ j.\ nat(i+j))\ 4\ 0\ 4$

code_reflect *Test* **functions** min_wpl_test

⟨*ML*⟩

end

3.8 Longest Common Subsequence

theory *Longest_Common_Subsequence*
  imports
    HOL−Library.Sublist
    HOL−Library.IArray
    HOL−Library.Code_Target_Numeral
    HOL−Library.Product_Lexorder
    HOL−Library.RBT_Mapping
    ../state_monad/State_Main
begin

3.8.1 Misc

lemma *finite_subseq*:
  finite {*xs*. *subseq xs ys*} (**is** *finite ?S*)
⟨*proof*⟩

lemma *subseq_singleton_right*:
  $subseq\ xs\ [x] = (xs = [x] \lor xs = [])$
  ⟨*proof*⟩

lemma *subseq_append_single_right*:
  $subseq\ xs\ (ys\ @\ [x]) = ((\exists\ xs'.\ subseq\ xs'\ ys \land xs = xs'\ @\ [x]) \lor subseq\ xs\ ys)$
  ⟨*proof*⟩

lemma *Max_nat_plus*:
  $Max\ (((+)\ n)\ `\ S) = (n :: nat) + Max\ S$ **if** $finite\ S\ S \neq \{\}$
  ⟨*proof*⟩

3.8.2 Definitions

context
  fixes $A\ B :: 'a\ list$
begin

fun $lcs :: nat \Rightarrow nat \Rightarrow nat$ **where**
  $lcs\ 0\ \_ = 0\ |$
  $lcs\ \_\ 0 = 0\ |$

$lcs\ (Suc\ i)\ (Suc\ j) = (if\ A!i = B!j\ then\ 1 + lcs\ i\ j\ else\ max\ (lcs\ i\ (j + 1))\ (lcs\ (i + 1)\ j))$

definition $OPT\ i\ j = Max\ \{length\ xs \mid xs.\ subseq\ xs\ (take\ i\ A) \wedge subseq\ xs\ (take\ j\ B)\}$

lemma *finite_OPT*:
$finite\ \{xs.\ subseq\ xs\ (take\ i\ A) \wedge subseq\ xs\ (take\ j\ B)\}$ (**is** $finite\ ?S$)
$\langle proof \rangle$

3.8.3 Correctness Proof

lemma *non_empty_OPT*:
$\{xs.\ subseq\ xs\ (take\ i\ A) \wedge subseq\ xs\ (take\ j\ B)\} \neq \{\}$
$\langle proof \rangle$

lemma *OPT_0_left*:
$OPT\ 0\ j = 0$
$\langle proof \rangle$

lemma *OPT_0_right*:
$OPT\ i\ 0 = 0$
$\langle proof \rangle$

lemma *OPT_rec1*:
$OPT\ (i + 1)\ (j + 1) = 1 + OPT\ i\ j$ (**is** $?l = ?r$)
if $A!i = B!j\ i < length\ A\ j < length\ B$
$\langle proof \rangle$

lemma *OPT_rec2*:
$OPT\ (i + 1)\ (j + 1) = max\ (OPT\ i\ (j + 1))\ (OPT\ (i + 1)\ j)$ (**is** $?l = ?r$)
if $A!i \neq B!j\ i < length\ A\ j < length\ B$
$\langle proof \rangle$

lemma *lcs_correct'*:
$OPT\ i\ j = lcs\ i\ j$ **if** $i \leq length\ A\ j \leq length\ B$
$\langle proof \rangle$

theorem *lcs_correct*:
$Max\ \{length\ xs \mid xs.\ subseq\ xs\ A \wedge subseq\ xs\ B\} = lcs\ (length\ A)\ (length\ B)$
$\langle proof \rangle$

end

3.8.4 Functional Memoization

context
  fixes A B :: $'a$ *iarray*
begin

fun *lcs_ia* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  lcs_ia 0 _ = 0 |
  lcs_ia _ 0 = 0 |
  lcs_ia (*Suc i*) (*Suc j*) =
    (*if A*!!*i* = *B*!!*j then* 1 + *lcs_ia i j else max* (*lcs_ia i* (*j* + 1)) (*lcs_ia* (*i*
+ 1) *j*))

lemma *lcs_lcs_ia*:
  lcs xs ys i j = *lcs_ia i j* **if** *A* = *IArray xs B* = *IArray ys*
  ⟨*proof*⟩

memoize_fun lcs_m: *lcs_ia* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *lcs_ia.simps*

memoize_correct
  ⟨*proof*⟩

lemmas [*code*] = lcs_m.*memoized_correct*

end

3.8.5 Test Case

definition lcs_a **where**
  lcs_a *xs ys* = (*let A* = *IArray xs*; *B* = *IArray ys in lcs_ia A B* (*length xs*)
(*length ys*))

lemma lcs_a*_correct*:
  lcs xs ys (*length xs*) (*length ys*) = lcs_a *xs ys*
  ⟨*proof*⟩

value lcs_a *''ABCDGH'' ''AEDFHR''*

value lcs_a *''AGGTAB'' ''GXTXAYB''*

end

theory *All_Examples*
imports
Bellman_Ford
Knapsack
Counting_Tiles
CYK
*Min_Ed_Dist*0
OptBST
Longest_Common_Subsequence
begin

end

References

[1] J. M. Kleinberg and É. Tardos. *Algorithm Design.* Addison-Wesley, 2006.

[2] S. Wimmer, S. Hu, and T. Nipkow. Verified memoization and dynamic programming. In J. Avigad and A. Mahboubi, editors, *ITP 2018, Proceedings*, Lecture Notes in Computer Science. Springer, 2018.