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* :: ($'M$,$'a$ ⇒ ($'M$, $'b$) *state*) *state* ⇒ ($'M$,$'a$) *state* ⇒ ($'M$,$'b$) *state* **where**
  fun_app_lifted f_T x_T ≡ *do* { *f* ← f_T; *x* ← x_T; *f x* }

bundle *state_monad_syntax* **begin**

notation *fun_app_lifted* (**infixl** ‹.› 999)
type_synonym ($'a$,$'M$,$'b$) *fun_lifted* = $'a$ ⇒ ($'M$,$'b$) *state* (‹_ ==_⟹ _› [3,1000,2] 2)
type_synonym ($'a$,$'b$) *dpfun* = $'a$ ==($'a$⇀$'b$)⟹ $'b$ (**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*:
  ⟨*f*⟩ . ⟨*x*⟩ = *f x*
  unfolding *fun_app_lifted_def bind_left_identity* **..**

qualified lemma *return_app_return_meta*:
  ⟨*f*⟩ . ⟨*x*⟩ ≡ *f x*
  unfolding *return_app_return* **.**

qualified definition if_T :: ($'M$, *bool*) *state* ⇒ ($'M$, $'a$) *state* ⇒ ($'M$, $'a$) *state* ⇒ ($'M$, $'a$) *state* **where**
  if_T b_T x_T y_T ≡ *do* {*b* ← 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 ⇒ 'a where
  Wrap x ≡ x

definition App :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b where
  App f ≡ f

lemma Wrap_App_Wrap:
  App (Wrap f) (Wrap x) ≡ f x
  unfolding App_def Wrap_def .


end
```

1.2 Parametricity of the State Monad

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

definition lift_p :: ('s ⇒ bool) ⇒ ('s, 'a) state ⇒ bool where
  lift_p P f =
    (∀ heap. P heap ⟶ (case State_Monad.run_state f heap of (_, heap)
⇒ 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) ⇒ P heap
  using lift P unfolding lift_p_def by auto

lemma lift_p_P:
  P heap' if State_Monad.run_state f heap = (v, heap')
  using that run_state_cases by auto
```

end

locale *state_mem_defs* =
  fixes *lookup* :: *'param* ⇒ (*'mem*, *'result option*) *state*
    and *update* :: *'param* ⇒ *'result* ⇒ (*'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*
unfolding *checkmem'_def checkmem_def* **..**

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

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′ ≡ *R* ===> *crel_vs R′*
term 0

definition *consistentDP* :: (′*param* == ′*mem* ⟹ ′*result*) ⇒ *bool* **where**
consistentDP ≡ ((=) ===> *crel_vs* (=)) *dp*
term 0

private lemma *cmem_intro*:
assumes ⋀*param v M′*. *State_Monad.run_state* (*lookup param*) *M* = (*Some v*, *M′*) ⟹ *v* = *dp param*
shows *cmem M*
unfolding *cmem_def map_of_def*
apply *safe*
subgoal for *param y*
by (*cases State_Monad.run_state* (*lookup param*) *M*) (*auto intro*: *assms*)
done

lemma *cmem_elim*:
assumes *cmem M State_Monad.run_state* (*lookup param*) *M* = (*Some v*, *M′*)
obtains *dp param* = *v*
using *assms* **unfolding** *cmem_def dom_def map_of_def* **by** *auto* (*metis fst_conv option.inject*)
term 0

lemma *crel_vs_intro*:
assumes ⋀*M v′ M′*. ⟦*cmem M*; *P M*; *State_Monad.run_state* v_T *M* = (*v′*, *M′*)⟧ ⟹ *R v v′* ∧ *cmem M′* ∧ *P M′*
shows *crel_vs R v* v_T
using *assms* **unfolding** *crel_vs_def* **by** *blast*
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′*
using *assms* **unfolding** *crel_vs_def* **by** *blast*
term 0

lemma *consistentDP_intro*:
assumes ⋀*param*. *Transfer.Rel* (*crel_vs* (=)) (*dp param*) (dp_T *param*)

shows *consistentDP* dp_T
using *assms* **unfolding** *consistentDP_def Rel_def* **by** *blast*

lemma *crel_vs_return*:
⟦*Transfer.Rel R x y*⟧ ⟹ *Transfer.Rel* (*crel_vs R*) (*Wrap x*) (*State_Monad.return y*)
unfolding *State_Monad.return_def Wrap_def Rel_def* **by** (*fastforce intro*: *crel_vs_intro*)
term 0

lemma *crel_vs_return_ext*:
⟦*Transfer.Rel R x y*⟧ ⟹ *Transfer.Rel* (*crel_vs R*) *x* (*State_Monad.return y*)
by (*fact crel_vs_return*[*unfolded Wrap_def*])
term 0

private lemma *cmem_upd*:
cmem M′ **if** *cmem M P M State_Monad.run_state* (*update param* (*dp param*)) *M* = (*v*, *M′*)
using *update_correct*[*of M param dp param*] *that* **unfolding** *cmem_def map_le_def* **by** *simp force*

private lemma *P_upd*:
P M′ **if** *P M State_Monad.run_state* (*update param* (*dp param*)) *M* = (*v*, *M′*)
by (*meson lift_p_P that update_inv*)

private lemma *crel_vs_get*:
⟦⋀*M*. *cmem M* ⟹ *crel_vs R v* (*sf M*)⟧ ⟹ *crel_vs R v* (*State_Monad.get* ≫= *sf*)
unfolding *State_Monad.get_def State_Monad.bind_def* **by** (*fastforce intro*: *crel_vs_intro elim*: *crel_vs_elim split*: *prod.split*)
term 0

private lemma *crel_vs_set*:
⟦*crel_vs R v sf*; *cmem M*; *P M*⟧ ⟹ *crel_vs R v* (*State_Monad.set M* ≫ *sf*)
unfolding *State_Monad.set_def State_Monad.bind_def* **by** (*fastforce intro*: *crel_vs_intro elim*: *crel_vs_elim split*: *prod.split*)
term 0

private lemma *crel_vs_bind_eq*:
⟦*crel_vs* (=) *v s*; *crel_vs R* (*f v*) (*sf v*)⟧ ⟹ *crel_vs R* (*f v*) (*s* ≫= *sf*)

unfolding *State_Monad.bind_def rel_fun_def* **by** (*fastforce intro*: *crel_vs_intro elim*: *crel_vs_elim split*: *prod.split*)
term 0

lemma *bind_transfer*[*transfer_rule*]:
(*crel_vs R*0 ===> (*R*0 ===>$_T$ *R*1) ===> *crel_vs R*1) ($\lambda v\, f.\ f\ v$) (⪢=)
unfolding *State_Monad.bind_def rel_fun_def* **by** (*fastforce intro*: *crel_vs_intro elim*: *crel_vs_elim split*: *prod.split*)

private lemma *cmem_lookup*:
cmem M′ **if** *cmem M P M State_Monad.run_state* (*lookup param*) *M* = (*v*, *M′*)
using *lookup_correct*[*of M param*] *that* **unfolding** *cmem_def map_le_def*
by *force*

private lemma *P_lookup*:
P M′ **if** *P M State_Monad.run_state* (*lookup param*) *M* = (*v*, *M′*)
by (*meson lift_p_P that lookup_inv*)

lemma *crel_vs_lookup*:
crel_vs (λ *v v′*. *case v′ of None* ⇒ *True* | *Some v′* ⇒ *v* = *v′* ∧ *v* = *dp param*) (*dp param*) (*lookup param*)
by (*auto elim*: *cmem_elim intro*: *cmem_lookup crel_vs_intro P_lookup split*: *option.split*)

lemma *crel_vs_update*:
crel_vs (=) () (*update param* (*dp param*))
by (*auto intro*: *cmem_upd crel_vs_intro P_upd*)

private lemma *crel_vs_checkmem*:
⟦*is_equality R*; *Transfer.Rel* (*crel_vs R*) (*dp param*) *s*⟧
⟹ *Transfer.Rel* (*crel_vs R*) (*dp param*) (*checkmem param s*)
unfolding *checkmem_def Rel_def is_equality_def*
by (*rule bind_transfer*[*unfolded rel_fun_def*, *rule_format*, *OF crel_vs_lookup*])
(*auto* 4 3 *intro*: *crel_vs_lookup crel_vs_update crel_vs_return*[*unfolded Rel_def Wrap_def*] *crel_vs_bind_eq*
split: *option.split_asm*
)

lemma *crel_vs_checkmem_tupled*:
assumes *v* = *dp param*
shows ⟦*is_equality R*; *Transfer.Rel* (*crel_vs R*) *v s*⟧
⟹ *Transfer.Rel* (*crel_vs R*) *v* (*checkmem param s*)
unfolding *assms* **by** (*fact crel_vs_checkmem*)

lemma *return_transfer*[*transfer_rule*]:
  (*R* ===>$_T$ *R*) *Wrap State_Monad.return*
  unfolding *rel_fun_def* **by** (*metis crel_vs_return Rel_def*)

lemma *fun_app_lifted_transfer*[*transfer_rule*]:
  (*crel_vs* (*R*0 ===>$_T$ *R*1) ===> *crel_vs R*0 ===> *crel_vs R*1) *App* (.)
  unfolding *App_def fun_app_lifted_def* **by** *transfer_prover*

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⟧ ⟹ *Transfer.Rel* (*crel_vs R*1) (*App f x*) (f_T . x_T)
  unfolding *Rel_def* **using** *fun_app_lifted_transfer*[*THEN rel_funD*, *THEN rel_funD*] **.**

lemma *if*$_T$*_transfer*[*transfer_rule*]:
  (*crel_vs* (=) ===> *crel_vs R* ===> *crel_vs R* ===> *crel_vs R*) *If State_Monad_Ext.if*$_T$
  unfolding *State_Monad_Ext.if*$_T$*_def* **by** *transfer_prover*
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 (*R*1 ===> *S*1) *f g* (*R*2 ===> *S*2) *g h*
  shows (*R*1 *OO R*2 ===> *S*1 *OO S*2) *f h*
  using *assms* **by** (*auto intro*!: *rel_funI dest*!: *rel_funD*)

lemma *rel_fun_comp*1:
  assumes (*R*1 ===> *S*1) *f g* (*R*2 ===> *S*2) *g h R*′ = *R*1 *OO R*2
  shows (*R*′ ===> *S*1 *OO S*2) *f h*
  using *assms rel_fun_comp* **by** *metis*

lemma *rel_fun_comp*2:

assumes (*R*1 ===> *S*1) *f g* (*R*2 ===> *S*2) *g h S′* = *S*1 *OO S*2
shows (*R*1 *OO R*2 ===> *S′*) *f h*
using *assms rel_fun_comp* **by** *metis*

lemma *rel_fun_relcompp*:
((*R*1 ===> *S*1) *OO* (*R*2 ===> *S*2)) *a b* ⟹ ((*R*1 *OO R*2) ===> (*S*1 *OO S*2)) *a b*
unfolding *OO_def rel_fun_def* **by** *blast*

lemma *rel_fun_comp1′*:
assumes (*R*1 ===> *S*1) *f g* (*R*2 ===> *S*2) *g h* ⋀ *a b*. *R′ a b* ⟹ (*R*1 *OO R*2) *a b*
shows (*R′* ===> *S*1 *OO S*2) *f h*
by (*auto intro*: *assms rel_fun_mono*[*OF rel_fun_comp*1])

lemma *rel_fun_comp2′*:
assumes (*R*1 ===> *S*1) *f g* (*R*2 ===> *S*2) *g h* ⋀ *a b*. (*S*1 *OO S*2) *a b* ⟹ *S′ a b*
shows (*R*1 *OO R*2 ===> *S′*) *f h*
by (*auto intro*: *assms rel_fun_mono*[*OF rel_fun_comp*1])

end
end

1.4 Heap Monad

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

definition *fun_app_lifted* :: (*′a* ⇒ *′b Heap*) *Heap* ⇒ *′a Heap* ⇒ *′b Heap*
where
fun_app_lifted f_T x_T ≡ *do* { *f* ← f_T; *x* ← x_T; *f x* }

bundle *heap_monad_syntax* **begin**

notation *fun_app_lifted* (**infixl** ‹.› 999)
type_synonym (*′a*, *′b*) *fun_lifted* = *′a* ⇒ *′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*
unfolding *fun_app_lifted_def return_bind* **..**

qualified lemma *return_app_return_meta*:
$\langle f\rangle$. $\langle x\rangle \equiv$ *f x*
unfolding *return_app_return* **.**

qualified 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 ⇒ *False* | *Some* (_, *heap*) ⇒ *P heap*
using *lift P* **unfolding** *lift_p_def* **by** *auto*

lemma *execute_cases'*:
  case execute f heap of Some (_, *heap*) ⇒ *P heap*
  using *execute_cases* **by** (*auto split*: *option.split*)

lemma *lift_p_None*[*simp*, *dest*]:
  False **if** *execute f heap* = *None*
  using *that execute_cases* **by** *auto*

lemma *lift_p_P*:
  case the (*execute f heap*) *of* (_, *heap*) ⇒ *P heap*
  using *execute_cases* **by** (*auto split*: *option.split_asm*)

lemma *lift_p_P'*:
  P heap' **if** *the* (*execute f heap*) = (*v*, *heap'*)
  using *that lift_p_P* **by** *auto*

lemma *lift_p_P''*:
  P heap' **if** *execute f heap* = *Some* (*v*, *heap'*)
  using *that lift_p_P* **by** *auto*

lemma *lift_p_the_Some*[*simp*]:
  execute f heap = *Some* (*v*, *heap'*) **if** *the* (*execute f heap*) = (*v*, *heap'*)
  using *that execute_cases* **by** (*auto split*: *option.split_asm*)

lemma *lift_p_E*:
  obtains *v heap'* **where** *execute f heap* = *Some* (*v*, *heap'*) *P heap'*
  using *execute_cases* **by** (*cases execute f heap*) *auto*

end

definition *state_of s* ≡ *State* (λ *heap*. *the* (*execute s heap*))

locale *heap_mem_defs* =
  fixes *P* :: *heap* ⇒ *bool*
    and *lookup* :: *'k* ⇒ *'v option Heap*
    and *update* :: *'k* ⇒ *'v* ⇒ *unit Heap*
begin

definition *rel_state* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*heap*, *'a*) *state* ⇒ *'b Heap* ⇒ *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 (*v2*, *heap2*) ⇒ *R v1 v2* ∧ *heap1* = *heap2* ∧ *P heap2* | *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*
unfolding *checkmem′_def checkmem_def* **..**

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'
  apply atomize_elim
  using assms unfolding rel_state_def
  apply auto
  apply (cases State_Monad.run_state f heap)
  apply auto
  apply (auto split: option.split_asm)
  done

lemma rel_state_intro:
  assumes
    ⋀ heap v heap'. P heap ⟹ State_Monad.run_state f heap = (v, heap')
      ⟹ ∃ v'. R v v' ∧ execute g heap = Some (v', heap')
    ⋀ heap v heap'. P heap ⟹ State_Monad.run_state f heap = (v, heap')
⟹ P heap'
  shows rel_state R f g
  unfolding rel_state_def
  apply auto
  apply (frule assms(1)[rotated])
   apply (auto intro: assms(2))
  done

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
  unfolding rel_fun_def State_Monad.bind_def Heap_Monad.bind_def
  by (force elim!: rel_state_elim intro!: rel_state_intro)

lemma transfer_return[transfer_rule]:
  (R ===> rel_state R) State_Monad.return Heap_Monad.return
 unfolding rel_fun_def State_Monad.return_def Heap_Monad.return_def
 by (fastforce intro: rel_state_intro elim: rel_state_elim simp: execute_heap)

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
 unfolding State_Monad_Ext.fun_app_lifted_def Heap_Monad_Ext.fun_app_lifted_def
```

by *transfer_prover*

lemma *transfer_get*[*transfer_rule*]:
  rel_state (=) *State_Monad.get heap_get*
 unfolding *State_Monad.get_def heap_get_def* **by** (*auto intro*: *rel_state_intro*)

end

end

locale *heap_inv* = *heap_mem_defs* _ *lookup* **for** *lookup* :: *'k* ⇒ *'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*)
  unfolding *rel_state_def lookup'_def* **using** *lookup_inv*[*of k*] **by** (*auto intro*: *lift_p_P'*)

lemma *rel_state_update*:
  rel_state (=) (*update' k v*) (*update k v*)
  unfolding *rel_state_def update'_def* **using** *update_inv*[*of k v*] **by** (*auto intro*: *lift_p_P'*)

context
  includes *lifting_syntax*
begin

lemma *transfer_lookup*:
  ((=) ===> *rel_state* (=)) *lookup' lookup*
  unfolding *rel_fun_def* **by** (*auto intro*: *rel_state_lookup*)

lemma *transfer_update*:
  ((=) ===> (=) ===> *rel_state* (=)) *update' update*
  unfolding *rel_fun_def* **by** (*auto intro*: *rel_state_update*)

lemma *transfer_checkmem*:
  ((=) ===> *rel_state* (=) ===> *rel_state* (=))
    (*state_mem_defs.checkmem lookup' update'*) *checkmem*
  supply [*transfer_rule*] = *transfer_lookup transfer_update*
 unfolding *state_mem_defs.checkmem_def checkmem_def* **by** *transfer_prover*

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*
using ‹*P m*› **unfolding** *state_mem_defs.map_of_def map_le_def lookup′_def*
by *simp* (*metis* (*mono_tags*, *lifting*) *domIff lookup_correct map_le_def map_of_heap_def*)

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*
unfolding *state_mem_defs.map_of_def map_le_def lookup′_def update′_def*
using *update_correct*[*of m k v*] *that* **by** (*auto split*: *if_split_asm simp*: *map_le_def map_of_heap_def*)

lemma *lookup′_inv*:
DP_CRelVS.lift_p P (*lookup′ k*)
unfolding *DP_CRelVS.lift_p_def lookup′_def* **by** (*auto elim*: *lift_p_P′*[*OF lookup_inv*])

lemma *update′_inv*:
DP_CRelVS.lift_p P (*update′ k v*)
unfolding *DP_CRelVS.lift_p_def update′_def* **by** (*auto elim*: *lift_p_P′*[*OF update_inv*])

lemma *mem_correct_heap*: *mem_correct lookup′ update′ P*
by (*intro mem_correct.intro lookup′_correct update′_correct lookup′_inv update′_inv*)

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 -
  interpret mem: mem_correct lookup_s update_s P
    by (rule correct)
  have [simp]: the (execute (lookup k) m) = run_state (lookup_s k) m if P
m for k m
    using lookup[THEN rel_funD, OF HOL.refl, of k] ‹P m› by (auto elim:
rel_state_elim)
  have [simp]: the (execute (update k v) m) = run_state (update_s k v) m if
P m for k v m
    using update[THEN rel_funD, THEN rel_funD, OF HOL.refl HOL.refl,
of k v] ‹P m›
    by (auto elim: rel_state_elim)
  have [simp]: map_of_heap m = mem.map_of m if P m for m
    unfolding map_of_heap_def mem.map_of_def using ‹P m› by simp
  show ?thesis
  apply standard
    subgoal for k
      using mem.lookup_inv[of k] lookup[THEN rel_funD, OF HOL.refl, of
k]
      unfolding lift_p_def DP_CRelVS.lift_p_def
      by (auto split: option.splits elim: rel_state_elim)
    subgoal for k v
     using mem.update_inv[of k] update[THEN rel_funD, THEN rel_funD,
OF HOL.refl HOL.refl, of k v]
      unfolding lift_p_def DP_CRelVS.lift_p_def
      by (auto split: option.splits elim: rel_state_elim)
    subgoal premises prems for m k
    proof -
      have P (snd (run_state (lookup_s k) m))
      by (meson DP_CRelVS.lift_p_P mem.lookup_inv prems prod.exhaust_sel)
      with mem.lookup_correct[OF ‹P m›, of k] ‹P m› show ?thesis
```

by (*simp add*: *prems*)
qed
subgoal premises *prems* **for** *m k v*
proof –
have *P* (*snd* (*run_state* ($update_s$ *k v*) *m*))
by (*meson DP_CRelVS.lift_p_P mem.update_inv prems prod.exhaust_sel*)
with *mem.update_correct*[*OF* ‹*P m*›, *of k*] ‹*P m*› **show** *?thesis*
by (*simp add*: *prems*)
qed
done
qed

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* ⇒ *bool* **and** *dp* :: *'k* ⇒ *'v* **and** *lookup* :: *'k* ⇒ *'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* ⇒ *'c* ⇒ *bool*) ⇒ (*'b* ⇒ *'d* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'c* ==*H*⟹ *'d*) ⇒ *bool* (**infixr** ‹===>$_T$› 55) **where**
rel_fun_lifted R R' ≡ *R* ===> *crel_vs R'*

definition *consistentDP* :: (*'k* ⇒ *'v Heap*) ⇒ *bool* **where**
consistentDP ≡ ((=) ===> *crel_vs* (=)) *dp*

lemma *consistentDP_intro*:
assumes ⋀*param*. *Transfer.Rel* (*crel_vs* (=)) (*dp param*) (dp_T *param*)
shows *consistentDP* dp_T
using *assms* **unfolding** *consistentDP_def Rel_def* **by** *blast*

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*
using *that* **unfolding** *crel_vs_def* **by** *auto*

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'*
using *assms* **unfolding** *crel_vs_def* **by** (*cases execute b heap*) *auto*

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*
using *assms* **unfolding** *crel_vs_def* **by** (*cases execute b heap*) *auto*

lemma *crel_vs_success*:
assumes *crel_vs R a b P heap Q heap state_dp_consistency.cmem heap*
shows *success b heap*
using *assms* **unfolding** *success_def* **by** (*auto elim*: *crel_vs_executeD*)

lemma *crel_vsI*: *crel_vs R a b* **if** (*state_dp_consistency.crel_vs R OO rel_state* (=)) *a b*
using *that* **by** (*auto* 4 3 *elim*: *state_dp_consistency.crel_vs_elim rel_state_elim*

simp: *crel_vs_def*)

lemma *transfer'_return*[*transfer_rule*]:
(*R* ===> *crel_vs R*) *Wrap return*
proof –
have (*R* ===> (*state_dp_consistency.crel_vs R OO rel_state* (=))) *Wrap return*
by (*rule rel_fun_comp*1 *state_dp_consistency.return_transfer transfer_return*)+ *auto*
then show *?thesis*
by (*blast intro*: *rel_fun_mono crel_vsI*)
qed

lemma *crel_vs_return*:
Transfer.Rel (*crel_vs R*) (*Wrap x*) (*return y*) **if** *Transfer.Rel R x y*
using *that* **unfolding** *Rel_def* **by** (*rule transfer'_return*[*unfolded rel_fun_def*, *rule_format*])

lemma *crel_vs_return_ext*:
$\llbracket$*Transfer.Rel R x y*$\rrbracket \Longrightarrow$ *Transfer.Rel* (*crel_vs R*) *x* (*Heap_Monad.return y*)
by (*fact crel_vs_return*[*unfolded Wrap_def*])
term 0

lemma *bind_transfer*[*transfer_rule*]:
(*crel_vs R*0 ===> (*R*0 ===> *crel_vs R*1) ===> *crel_vs R*1) (λ*v f*. *f v*) ($\gg=$)
unfolding *rel_fun_def bind_def*
by *safe* (*subst crel_vs_def*, *auto* 4 4 *elim*: *crel_vs_execute_Some elim*!: *crel_vs_executeD*)

lemma *crel_vs_update*:
crel_vs (=) () (*update param* (*dp param*))
by (*rule*
crel_vsI relcomppI state_dp_consistency.crel_vs_update
rel_state_update[*unfolded rel_fun_def*, *rule_format*] *HOL.refl*
)+

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*)
by (*rule*

crel_vsI relcomppI state_dp_consistency.crel_vs_lookup
rel_state_lookup[*unfolded rel_fun_def*, *rule_format*] *HOL.refl*
)+

lemma *crel_vs_eq_eq_onp*:
crel_vs (*eq_onp* (λ *x*. *x* = *v*)) *v s* **if** *crel_vs* (=) *v s*
using *that* **unfolding** *crel_vs_def* **by** (*auto split*: *option.split simp*: *eq_onp_def*)

lemma *crel_vs_bind_eq*:
⟦*crel_vs* (=) *v s*; *crel_vs R* (*f v*) (*sf v*)⟧ ⟹ *crel_vs R* (*f v*) (*s* ≫= *sf*)
by (*erule bind_transfer*[*unfolded rel_fun_def*, *rule_format*, *OF crel_vs_eq_eq_onp*])
(*auto simp*: *eq_onp_def*)

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*
unfolding *checkmem_def Rel_def that*(1)[*unfolded is_equality_def*]
by (*rule bind_transfer*[*unfolded rel_fun_def*, *rule_format*, *OF crel_vs_lookup*])
(*auto* 4 3 *split*: *option.split_asm intro*: *crel_vs_bind_eq crel_vs_update*
crel_vs_return[*unfolded Wrap_def Rel_def*] *that*(2)[*unfolded Rel_def that*(1)[*unfolded*
is_equality_def]])

lemma *crel_vs_checkmem_tupled*:
assumes *v* = *dp param*
shows ⟦*is_equality R*; *Transfer.Rel* (*crel_vs R*) *v s*⟧
⟹ *Transfer.Rel* (*crel_vs R*) *v* (*checkmem param s*)
unfolding *assms* **by** (*fact crel_vs_checkmem*)

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
unfolding *Heap_Monad_Ext.fun_app_lifted_def App_def* **by** *transfer_prover*

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⟧ ⟹ *Transfer.Rel* (*crel_vs R*1) (*App f x*) (f_T . x_T)
unfolding *Rel_def* **using** *transfer_fun_app_lifted*[*THEN rel_funD*, *THEN*
rel_funD] **.**

end

end

locale *dp_consistency_heap* = *heap_correct* +

fixes *dp* :: *'a* ⇒ *'b*
begin

interpretation *state_mem_correct*: *mem_correct lookup' update' P*
by (*rule mem_correct_heap*)

interpretation *state_dp_consistency*: *dp_consistency lookup' update' P dp*
..

lemma *dp_heap*: *dp_heap P P lookup lookup' update update'*
by (*standard*; *rule transfer_lookup transfer_update*)

sublocale *dp_heap P P dp lookup lookup' update update'*
by (*rule dp_heap*)

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
using *empty_correct*
unfolding *state_dp_consistency.cmem_def*
unfolding *map_of_heap_def*
unfolding *state_dp_consistency.map_of_def*
unfolding *lookup'_def*
unfolding *map_le_def*
by *auto*

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*)
using *that* **unfolding** *consistentDP_def*
by (*auto dest*!: *rel_funD*[**where** *x* = *x*] *elim*!: *crel_vs_executeD intro*: *P_empty cmem_empty*)

lemma *memoized_success*:
success (*dp*$_T$ *x*) *empty* **if** *consistentDP dp*$_T$
using *that cmem_empty P_empty*
by (*auto dest*!: *rel_funD intro*: *crel_vs_success simp*: *consistentDP_def*)

lemma *memoized*:
dp x = *fst* (*the* (*Heap_Monad.execute* (*dp*$_T$ *x*) *empty*)) **if** *consistentDP dp*$_T$
using *surjective_pairing memoization_correct*(1)[*OF that*]
memoized_success[*OF that*, *unfolded success_def*]
by (*cases execute* (*dp*$_T$ *x*) *empty*; *auto*)

lemma *cmem_result*:
state_dp_consistency.cmem (*snd* (*the* (*Heap_Monad.execute* (*dp*$_T$ *x*) *empty*)))
if *consistentDP dp*$_T$
using *surjective_pairing memoization_correct*(2)[*OF that*(1)]
memoized_success[*OF that*, *unfolded success_def*]
by (*cases execute* (*dp*$_T$ *x*) *empty*; *auto*)

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* ⟹ *run_state f heap* = (*x*, *heap*′) ⟹ *P heap*′
unfolding *lift_p_def* **by** (*auto intro*: *that*)

lemma *mem_correct_default*:
mem_correct
(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*)
by *standard*
(*auto simp*: *map_le_def state_mem_defs.map_of_def State_Monad.bind_def State_Monad.get_def State_Monad.return_def State_Monad.set_def lift_p_def*)

lemma *mem_correct_rbt_mapping*:
mem_correct
(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*)
by *standard*
(*auto simp*:
map_le_def state_mem_defs.map_of_def State_Monad.bind_def State_Monad.get_def State_Monad.return_def State_Monad.set_def lookup_update′ lift_p_def
)

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*) = {}
using *empty_correct* **by** (*auto dest*: *map_le_implies_dom_le*)

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
by (*intro mem_correct_empty.intro*[*OF mem_correct_default*] *mem_correct_empty_axioms.intro*)
(*auto simp*: *state_mem_defs.map_of_def map_le_def State_Monad.bind_def State_Monad.get_def State_Monad.return_def*)

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
by (*intro mem_correct_empty.intro*[*OF mem_correct_rbt_mapping*] *mem_correct_empty_axioms*
(*auto simp*: *state_mem_defs.map_of_def map_le_def State_Monad.bind_def*
State_Monad.get_def State_Monad.return_def lookup_empty)

locale *dp_consistency_empty* =
dp_consistency + *mem_correct_empty*
begin

lemma *cmem_empty*:
cmem empty
using *empty_correct* **unfolding** *cmem_def* **by** *auto*

corollary *memoization_correct*:
dp x = *v cmem m* **if** *consistentDP* dp_T *State_Monad.run_state* (dp_T *x*) *empty* = (*v*, *m*)
using *that* **unfolding** *consistentDP_def*
by (*auto dest*!: *rel_funD*[**where** *x* = *x*] *elim*!: *crel_vs_elim intro*: *P_empty* *cmem_empty*)

lemma *memoized*:
dp x = *fst* (*State_Monad.run_state* (dp_T *x*) *empty*) **if** *consistentDP* dp_T
using *surjective_pairing memoization_correct*(1)[*OF that*] **by** *blast*

lemma *cmem_result*:
cmem (*snd* (*State_Monad.run_state* (dp_T *x*) *empty*)) **if** *consistentDP* dp_T
using *surjective_pairing memoization_correct*(2)[*OF that*] **by** *blast*

end

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

sublocale *dp_consistency_empty*
λ *k*. *do* {(*m*::*'param* ⇀ *'result*) ← *State_Monad.get*; *State_Monad.return* (*m k*)}
λ *k v*. *do* {*m* ← *State_Monad.get*; *State_Monad.set* (*m*(*k*↦*v*))}
λ (_::*'param* ⇀ *'result*). *True*
dp
Map.empty
by (*intro*
dp_consistency_empty.intro dp_consistency.intro mem_correct_default *mem_correct_empty_default*

)

end

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

sublocale *dp_consistency_empty*
(λ *k*. *do* {(*m*::(*'param*,*'result*) *mapping*) ← *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 _::(*'param*,*'result*) *mapping*. *True*) *dp Mapping.empty*
by (*intro*
dp_consistency_empty.intro dp_consistency.intro mem_correct_rbt_mapping
mem_correct_rbt_empty_mapping
)

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*
unfolding *state_mem_defs.map_of_def lookup_trace_def*
by (*rule ext*) (*auto split*: *prod.split option.split*)

lemma *mem_correct_tracing*: *mem_correct lookup_trace update_trace* (*P o snd*)
by *standard*
(*auto*
intro!: *lift_pI*
elim: *lift_p_P*[*OF lookup_inv*]
simp: *lookup_trace_def update_trace_def state_mem_defs.map_of_def map_of_simp*
split: *prod.splits option.splits*;
metis snd_conv lookup_correct update_correct lift_p_P update_inv lookup_inv lift_p_P
)+

end

context *mem_correct_empty*
begin

lemma *mem_correct_tracing_empty*:
mem_correct_empty lookup_trace update_trace (*P o snd*) ([], *empty*)
by (*intro mem_correct_empty.intro mem_correct_tracing mem_correct_empty_axioms.intro*)
(*simp add*: *map_of_simp empty_correct P_empty*)+

end

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

interpretation *mapping*: *dp_consistency_mapping* **.**

sublocale *dp_consistency_empty*
mapping.lookup_trace mapping.update_trace (λ _. *True*) *o snd dp* ([], *Mapping.empty*)
by (*rule*
dp_consistency_empty.intro dp_consistency.intro
mapping.mem_correct_tracing_empty mem_correct_empty.axioms(1)
)+

end

end

2.2 Pair Memory

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

lemma *map_add_mono*:
 $(m1 ++ m2) \subseteq_m (m1' ++ m2')$ **if** $m1 \subseteq_m m1'$ $m2 \subseteq_m m2'$ $dom\ m1 \cap dom\ m2' = \{\}$
 using *that* **unfolding** *map_le_def map_add_def dom_def* **by** (*auto split*: *option.splits*)

lemma *map_add_upd2*:
 $f(x \mapsto y) ++ g = (f ++ g)(x \mapsto y)$ **if** $dom\ f \cap dom\ g = \{\}$ $x \notin dom\ g$
 apply (*subst map_add_comm*)
  defer
  apply *simp*
  apply (*subst map_add_comm*)
 using *that*
 by *auto*

locale *pair_mem_defs* =
 fixes *lookup1 lookup2* :: $'a \Rightarrow ('mem, 'v\ option)$ *state*
  and *update1 update2* :: $'a \Rightarrow 'v \Rightarrow ('mem, unit)$ *state*
  and *move12* :: $'k1 \Rightarrow ('mem, unit)$ *state*
  and *get_k1 get_k2* :: $('mem, 'k1)$ *state*
  and $P :: 'mem \Rightarrow bool$
 fixes *key1* :: $'k \Rightarrow 'k1$ **and** *key2* :: $'k \Rightarrow 'a$
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 $k' = key1\ k$;
  $k2 \leftarrow get\_k2$;
  if $k' = k2$
  then lookup2 (*key2 k*)
  else do {

```
      k1 ← get_k1;
      if k' = k1
      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 ← get_k1;
    if k' = k1
    then update1 (key2 k) v
    else do {
      k2 ← get_k2;
      if k' = k2
      then update2 (key2 k) v
      else (move12 k' ≫ update1 (key2 k) v)
    }
  }
```

```
sublocale pair: state_mem_defs lookup_pair update_pair .
```

```
sublocale mem1: state_mem_defs lookup1 update1 .
```

```
sublocale mem2: state_mem_defs lookup2 update2 .
```

```
definition
  inv_pair heap ≡
    let
      k1 = fst (State_Monad.run_state get_k1 heap);
      k2 = fst (State_Monad.run_state get_k2 heap)
    in
    (∀ k ∈ dom (mem1.map_of heap). ∃ k'. key1 k' = k1 ∧ key2 k' = k) ∧
    (∀ k ∈ dom (mem2.map_of heap). ∃ k'. key1 k' = k2 ∧ key2 k' = k) ∧
    k1 ≠ k2 ∧ P heap
```

```
definition
  map_of1 m k = (if key1 k = fst (State_Monad.run_state get_k1 m) then
```

*mem*1.*map_of* *m* (*key*2 *k*) *else* *None*)

definition
*map_of*2 *m k* = (*if* *key*1 *k* = *fst* (*State_Monad.run_state* *get_k*2 *m*) *then* *mem*2.*map_of* *m* (*key*2 *k*) *else* *None*)

end

locale *pair_mem* = *pair_mem_defs* +
assumes *get_state*:
State_Monad.run_state *get_k*1 *m* = (*k*, *m'*) ⟹ *m'* = *m*
State_Monad.run_state *get_k*2 *m* = (*k*, *m'*) ⟹ *m'* = *m*
assumes *move*12*_correct*:
P m ⟹ *State_Monad.run_state* (*move*12 *k*1) *m* = (*x*, *m'*) ⟹ *mem*1.*map_of* *m'* $\subseteq_m$ *Map.empty*
P m ⟹ *State_Monad.run_state* (*move*12 *k*1) *m* = (*x*, *m'*) ⟹ *mem*2.*map_of* *m'* $\subseteq_m$ *mem*1.*map_of* *m*
assumes *move*12*_keys*:
State_Monad.run_state (*move*12 *k*1) *m* = (*x*, *m'*) ⟹ *fst* (*State_Monad.run_state* *get_k*1 *m'*) = *k*1
State_Monad.run_state (*move*12 *k*1) *m* = (*x*, *m'*) ⟹ *fst* (*State_Monad.run_state* *get_k*2 *m'*) = *fst* (*State_Monad.run_state* *get_k*1 *m*)
assumes *move*12*_inv*:
lift_p *P* (*move*12 *k*1)
assumes *lookup_inv*:
lift_p *P* (*lookup*1 *k'*) *lift_p* *P* (*lookup*2 *k'*)
assumes *update_inv*:
lift_p *P* (*update*1 *k' v*) *lift_p* *P* (*update*2 *k' v*)
assumes *lookup_keys*:
P m ⟹ *State_Monad.run_state* (*lookup*1 *k'*) *m* = (*v'*, *m'*) ⟹
fst (*State_Monad.run_state* *get_k*1 *m'*) = *fst* (*State_Monad.run_state* *get_k*1 *m*)
P m ⟹ *State_Monad.run_state* (*lookup*1 *k'*) *m* = (*v'*, *m'*) ⟹
fst (*State_Monad.run_state* *get_k*2 *m'*) = *fst* (*State_Monad.run_state* *get_k*2 *m*)
P m ⟹ *State_Monad.run_state* (*lookup*2 *k'*) *m* = (*v'*, *m'*) ⟹
fst (*State_Monad.run_state* *get_k*1 *m'*) = *fst* (*State_Monad.run_state* *get_k*1 *m*)
P m ⟹ *State_Monad.run_state* (*lookup*2 *k'*) *m* = (*v'*, *m'*) ⟹
fst (*State_Monad.run_state* *get_k*2 *m'*) = *fst* (*State_Monad.run_state* *get_k*2 *m*)
assumes *update_keys*:
P m ⟹ *State_Monad.run_state* (*update*1 *k' v*) *m* = (*x*, *m'*) ⟹
fst (*State_Monad.run_state* *get_k*1 *m'*) = *fst* (*State_Monad.run_state*

*get_k*1 *m*)
  P m ⟹ *State_Monad.run_state* (*update*1 *k′ v*) *m* = (*x*, *m′*) ⟹
  fst (*State_Monad.run_state get_k*2 *m′*) = *fst* (*State_Monad.run_state get_k*2 *m*)
  P m ⟹ *State_Monad.run_state* (*update*2 *k′ v*) *m* = (*x*, *m′*) ⟹
  fst (*State_Monad.run_state get_k*1 *m′*) = *fst* (*State_Monad.run_state get_k*1 *m*)
  P m ⟹ *State_Monad.run_state* (*update*2 *k′ v*) *m* = (*x*, *m′*) ⟹
  fst (*State_Monad.run_state get_k*2 *m′*) = *fst* (*State_Monad.run_state get_k*2 *m*)
assumes
  lookup_correct:
    P m ⟹ *mem*1.*map_of* (*snd* (*State_Monad.run_state* (*lookup*1 *k′*) *m*)) $\subseteq_m$ (*mem*1.*map_of m*)
    P m ⟹ *mem*2.*map_of* (*snd* (*State_Monad.run_state* (*lookup*1 *k′*) *m*)) $\subseteq_m$ (*mem*2.*map_of m*)
    P m ⟹ *mem*1.*map_of* (*snd* (*State_Monad.run_state* (*lookup*2 *k′*) *m*)) $\subseteq_m$ (*mem*1.*map_of m*)
    P m ⟹ *mem*2.*map_of* (*snd* (*State_Monad.run_state* (*lookup*2 *k′*) *m*)) $\subseteq_m$ (*mem*2.*map_of m*)
assumes
  update_correct:
    P m ⟹ *mem*1.*map_of* (*snd* (*State_Monad.run_state* (*update*1 *k′ v*) *m*)) $\subseteq_m$ (*mem*1.*map_of m*)(*k′* ↦ *v*)
    P m ⟹ *mem*2.*map_of* (*snd* (*State_Monad.run_state* (*update*2 *k′ v*) *m*)) $\subseteq_m$ (*mem*2.*map_of m*)(*k′* ↦ *v*)
    P m ⟹ *mem*2.*map_of* (*snd* (*State_Monad.run_state* (*update*1 *k′ v*) *m*)) $\subseteq_m$ *mem*2.*map_of m*
    P m ⟹ *mem*1.*map_of* (*snd* (*State_Monad.run_state* (*update*2 *k′ v*) *m*)) $\subseteq_m$ *mem*1.*map_of m*
begin

lemma *map_of_le_pair*:
  pair.map_of m $\subseteq_m$ *map_of*1 *m* ++ *map_of*2 *m*
  if *inv_pair m*
  using *that*
  unfolding *pair.map_of_def map_of*1*_def map_of*2*_def*
  unfolding *lookup_pair_def inv_pair_def map_of_def map_le_def dom_def map_add_def*
  unfolding *State_Monad.bind_def*
  by (*auto* 4 4
    simp: *mem*2.*map_of_def mem*1.*map_of_def Let_def*
    dest: *get_state split*: *prod.split_asm if_split_asm*
)

lemma *pair_le_map_of*:
*map_of*1 *m* ++ *map_of*2 *m* $\subseteq_m$ *pair.map_of m*
if *inv_pair m*
using *that*
unfolding *pair.map_of_def map_of1_def map_of2_def*
unfolding *lookup_pair_def inv_pair_def map_of_def map_le_def dom_def map_add_def*
unfolding *State_Monad.bind_def*
by (*auto*
simp: *mem2.map_of_def mem1.map_of_def State_Monad.run_state_return Let_def*
dest: *get_state split*: *prod.splits if_split_asm option.split*
)

lemma *map_of_eq_pair*:
*map_of*1 *m* ++ *map_of*2 *m* = *pair.map_of m*
if *inv_pair m*
using *that*
unfolding *pair.map_of_def map_of1_def map_of2_def*
unfolding *lookup_pair_def inv_pair_def map_of_def map_le_def dom_def map_add_def*
unfolding *State_Monad.bind_def*
by (*auto* 4 4
simp: *mem2.map_of_def mem1.map_of_def State_Monad.run_state_return Let_def*
dest: *get_state split*: *prod.splits option.split*
)

lemma *inv_pair_neq*[*simp*]:
False **if** *inv_pair m fst* (*State_Monad.run_state get_k*1 *m*) = *fst* (*State_Monad.run_state get_k*2 *m*)
using *that* **unfolding** *inv_pair_def* **by** *auto*

lemma *inv_pair_P_D*:
P m **if** *inv_pair m*
using *that* **unfolding** *inv_pair_def* **by** (*auto simp*: *Let_def*)

lemma *inv_pair_domD*[*intro*]:
dom (*map_of*1 *m*) ∩ *dom* (*map_of*2 *m*) = {} **if** *inv_pair m*
using *that* **unfolding** *inv_pair_def map_of1_def map_of2_def* **by** (*auto split*: *if_split_asm*)

lemma *move12_correct1*:

*map_of*1 *heap'* $\subseteq_m$ *Map.empty* **if** *State_Monad.run_state* (*move*12 *k*1) *heap* = (*x*, *heap'*) *P heap*
using *move*12_*correct*[*OF that*(2,1)] **unfolding** *map_of*1_*def* **by** (*auto simp*: *move*12_*keys map_le_def*)

lemma *move*12_*correct*2:
*map_of*2 *heap'* $\subseteq_m$ *map_of*1 *heap* **if** *State_Monad.run_state* (*move*12 *k*1) *heap* = (*x*, *heap'*) *P heap*
using *move*12_*correct*(2)[*OF that*(2,1)] *that* **unfolding** *map_of*1_*def* *map_of*2_*def*
by (*auto simp*: *move*12_*keys map_le_def*)

lemma *dom_empty*[*simp*]:
dom (*map_of*1 *heap'*) = {} **if** *State_Monad.run_state* (*move*12 *k*1) *heap* = (*x*, *heap'*) *P heap*
using *move*12_*correct*1[*OF that*] **by** (*auto dest*: *map_le_implies_dom_le*)

lemma *inv_pair_lookup*1:
inv_pair m' **if** *State_Monad.run_state* (*lookup*1 *k*) *m* = (*v*, *m'*) *inv_pair* *m*
using *that lookup_inv*[*of k*] *inv_pair_P_D*[*OF* ‹*inv_pair m*›] **unfolding** *inv_pair_def*
by (*auto* 4 4
simp: *Let_def lookup_keys*
dest: *lift_p_P lookup_correct*[*of _ k*, *THEN map_le_implies_dom_le*]
)

lemma *inv_pair_lookup*2:
inv_pair m' **if** *State_Monad.run_state* (*lookup*2 *k*) *m* = (*v*, *m'*) *inv_pair* *m*
using *that lookup_inv*[*of k*] *inv_pair_P_D*[*OF* ‹*inv_pair m*›] **unfolding** *inv_pair_def*
by (*auto* 4 4
simp: *Let_def lookup_keys*
dest: *lift_p_P lookup_correct*[*of _ k*, *THEN map_le_implies_dom_le*]
)

lemma *inv_pair_update*1:
inv_pair m'
if *State_Monad.run_state* (*update*1 (*key*2 *k*) *v*) *m* = (*v'*, *m'*) *inv_pair m* *fst* (*State_Monad.run_state get_k*1 *m*) = *key*1 *k*
using *that update_inv*[*of key*2 *k v*] *inv_pair_P_D*[*OF* ‹*inv_pair m*›] **unfolding** *inv_pair_def*
apply (*auto*

simp: *Let_def update_keys*
dest: *lift_p_P update_correct*[*of _ key*2 *k v*, *THEN map_le_implies_dom_le*]
)
apply (*frule update_correct*[*of _ key*2 *k v*, *THEN map_le_implies_dom_le*]; *auto* 13 2; *fail*)
apply (*frule update_correct*[*of _ key*2 *k v*, *THEN map_le_implies_dom_le*]; *auto* 13 2; *fail*)
done

lemma *inv_pair_update*2:
inv_pair m'
if *State_Monad.run_state* (*update*2 (*key*2 *k*) *v*) *m* = (*v'*, *m'*) *inv_pair m* *fst* (*State_Monad.run_state get_k*2 *m*) = *key*1 *k*
using *that update_inv*[*of key*2 *k v*] *inv_pair_P_D*[*OF* ‹*inv_pair m*›] **unfolding** *inv_pair_def*
apply (*auto*
simp: *Let_def update_keys*
dest: *lift_p_P update_correct*[*of _ key*2 *k v*, *THEN map_le_implies_dom_le*]
)
apply (*frule update_correct*[*of _ key*2 *k v*, *THEN map_le_implies_dom_le*]; *auto* 13 2; *fail*)
apply (*frule update_correct*[*of _ key*2 *k v*, *THEN map_le_implies_dom_le*]; *auto* 13 2; *fail*)
done

lemma *inv_pair_move*12:
inv_pair m'
if *State_Monad.run_state* (*move*12 *k*) *m* = (*v'*, *m'*) *inv_pair m fst* (*State_Monad.run_state get_k*1 *m*) ≠ *k*
using *that move*12*_inv*[*of k*] *inv_pair_P_D*[*OF* ‹*inv_pair m*›] **unfolding** *inv_pair_def*
apply (*auto*
simp: *Let_def move*12*_keys*
dest: *lift_p_P move*12*_correct*[*of _ k*, *THEN map_le_implies_dom_le*]
)
apply (*blast dest*: *move*12*_correct*[*of _ k*, *THEN map_le_implies_dom_le*])
done

lemma *mem_correct_pair*:
mem_correct lookup_pair update_pair inv_pair
if *injective*: ∀ *k k'*. *key*1 *k* = *key*1 *k'* ∧ *key*2 *k* = *key*2 *k'* ⟶ *k* = *k'*
proof (*standard*, *goal_cases*)
case (1 *k*) — Lookup invariant
show *?case*

```
    unfolding lookup_pair_def Let_def
    by (auto 4 4
        intro!: lift_pI
        dest: get_state inv_pair_lookup1 inv_pair_lookup2
        simp: State_Monad.bind_def State_Monad.run_state_return
        split: if_split_asm prod.split_asm
        )
next
  case (2 k v) — Update invariant
  show ?case
    unfolding update_pair_def Let_def
    apply (auto 4 4
        intro!: lift_pI intro: inv_pair_update1 inv_pair_update2
        dest: get_state
      simp: State_Monad.bind_def get_state State_Monad.run_state_return
        split: if_split_asm prod.split_asm
        )+
    apply (elim inv_pair_update1 inv_pair_move12)
      apply (((subst get_state, assumption)+)?, auto intro: move12_keys
dest: get_state; fail)+
    done
next
  case (3 m k)
  {
    let ?m = snd (State_Monad.run_state (lookup2 (key2 k)) m)
    have map_of1 ?m ⊆m map_of1 m
        by (smt 3 domIff inv_pair_P_D local.lookup_keys lookup_correct
map_le_def map_of1_def surjective_pairing)
    moreover have map_of2 ?m ⊆m map_of2 m
        by (smt 3 domIff inv_pair_P_D local.lookup_keys lookup_correct
map_le_def map_of2_def surjective_pairing)
    moreover have dom (map_of1 ?m) ∩ dom (map_of2 m) = {}
    using 3 ‹map_of1 ?m ⊆m map_of1 m› inv_pair_domD map_le_implies_dom_le
by fastforce
    moreover have inv_pair ?m
      using 3 inv_pair_lookup2 surjective_pairing by metis
    ultimately have pair.map_of ?m ⊆m pair.map_of m
      apply (subst map_of_eq_pair[symmetric])
       defer
       apply (subst map_of_eq_pair[symmetric])
      by (auto intro: 3 map_add_mono)
  }
  moreover
  {
```

```
    let ?m = snd (State_Monad.run_state (lookup1 (key2 k)) m)
    have map_of1 ?m ⊆m map_of1 m
        by (smt 3 domIff inv_pair_P_D local.lookup_keys lookup_correct
map_le_def map_of1_def surjective_pairing)
    moreover have map_of2 ?m ⊆m map_of2 m
        by (smt 3 domIff inv_pair_P_D local.lookup_keys lookup_correct
map_le_def map_of2_def surjective_pairing)
    moreover have dom (map_of1 ?m) ∩ dom (map_of2 m) = {}
     using 3 ‹map_of1 ?m ⊆m map_of1 m› inv_pair_domD map_le_implies_dom_le
by fastforce
    moreover have inv_pair ?m
      using 3 inv_pair_lookup1 surjective_pairing by metis
    ultimately have pair.map_of ?m ⊆m pair.map_of m
      apply (subst map_of_eq_pair[symmetric])
       defer
       apply (subst map_of_eq_pair[symmetric])
      by (auto intro: 3 map_add_mono)
  }
  ultimately show ?case
    by (auto
        split:if_split prod.split
      simp: Let_def lookup_pair_def State_Monad.bind_def State_Monad.run_state_return
dest: get_state intro: map_le_refl
        )
next
  case prems: (4 m k v)
  let ?m1 = snd (State_Monad.run_state (update1 (key2 k) v) m)
  let ?m2 = snd (State_Monad.run_state (update2 (key2 k) v) m)
  from prems have disjoint: dom (map_of1 m) ∩ dom (map_of2 m) = {}
    by (simp add: inv_pair_domD)
  show ?case
    apply (auto
        intro: map_le_refl dest: get_state
        split: prod.split
      simp: Let_def update_pair_def State_Monad.bind_def State_Monad.run_state_return
        )
  proof goal_cases
    case (1 m')
    then have m' = m
      by (rule get_state)
    from 1 prems have map_of1 ?m1 ⊆m (map_of1 m)(k ↦ v)
      by (smt inv_pair_P_D map_le_def map_of1_def surjective_pairing
domIff
          fst_conv fun_upd_apply injective update_correct update_keys
```

)
moreover from *prems* **have** *map_of2 ?m1* $\subseteq_m$ *map_of2 m*
by (*smt domIff inv_pair_P_D update_correct update_keys map_le_def map_of2_def surjective_pairing*)
moreover from *prems* **have** *dom* (*map_of1 ?m1*) ∩ *dom* (*map_of2 m*) = {}
by (*smt inv_pair_P_D*[*OF* ‹*inv_pair m*›] *domIff Int_emptyI eq_snd_iff inv_pair_neq*
map_of1_def map_of2_def update_keys(1)
)
moreover from 1 *prems* **have** $k \notin$ *dom* (*map_of2 m*)
using *inv_pair_neq map_of2_def* **by** *fastforce*
moreover from 1 *prems* **have** *inv_pair ?m1*
using *inv_pair_update1 fst_conv surjective_pairing* **by** *metis*
ultimately show *pair.map_of* (*snd* (*State_Monad.run_state* (*update1* (*key2 k*) *v*) *m'*)) $\subseteq_m$ (*pair.map_of m*)(*k* ↦ *v*)
unfolding ‹*m'* = *m*› **using** *disjoint*
apply (*subst map_of_eq_pair*[*symmetric*])
defer
apply (*subst map_of_eq_pair*[*symmetric*], *rule prems*)
apply (*subst map_add_upd2*[*symmetric*])
by (*auto intro*: *map_add_mono*)
next
case (2 *k1 m' m''*)
then have *m'* = *m m''* = *m*
by (*auto dest*: *get_state*)
from 2 *prems* **have** *map_of2 ?m2* $\subseteq_m$ (*map_of2 m*)(*k* ↦ *v*)
unfolding ‹*m'* = *m*› ‹*m''* = *m*›
by (*smt inv_pair_P_D map_le_def map_of2_def surjective_pairing domIff*
fst_conv fun_upd_apply injective update_correct update_keys
)
moreover from *prems* **have** *map_of1 ?m2* $\subseteq_m$ *map_of1 m*
by (*smt domIff inv_pair_P_D update_correct update_keys map_le_def map_of1_def surjective_pairing*)
moreover from 2 **have** *dom* (*map_of1 ?m2*) ∩ *dom* ((*map_of2 m*)(*k* ↦ *v*)) = {}
unfolding ‹*m'* = *m*›
by (*smt domIff* ‹*map_of1 ?m2* $\subseteq_m$ *map_of1 m*› *disjoint_iff_not_equal fst_conv fun_upd_apply*
map_le_def map_of1_def map_of2_def
)
moreover from 2 *prems* **have** *inv_pair ?m2*
unfolding ‹*m'* = *m*›

using *inv_pair_update2 fst_conv surjective_pairing* **by** *metis*
ultimately show *pair.map_of* (*snd* (*State_Monad.run_state* (*update2* (*key2 k*) *v*) m'')) $\subseteq_m$ (*pair.map_of m*)($k \mapsto v$)
unfolding ‹$m' = m$› ‹$m'' = m$›
apply (*subst map_of_eq_pair*[*symmetric*])
defer
apply (*subst map_of_eq_pair*[*symmetric*], *rule prems*)
apply (*subst map_add_upd*[*symmetric*])
by (*rule map_add_mono*)
next
case (3 *k1 m1 k2 m2 m3*)
then have $m1 = m$ $m2 = m$
by (*auto dest*: *get_state*)
let *?m3* = *snd* (*State_Monad.run_state* (*update1* (*key2 k*) *v*) *m3*)
from 3 *prems* **have** *map_of1 ?m3* $\subseteq_m$ (*map_of2 m*)($k \mapsto v$)
unfolding ‹$m2 = m$›
by (*smt inv_pair_P_D map_le_def map_of1_def surjective_pairing domIff*
fst_conv fun_upd_apply injective
inv_pair_move12 move12_correct move12_keys update_correct update_keys
)
moreover have *map_of2 ?m3* $\subseteq_m$ *map_of1 m*
proof −
from *prems* 3 **have** *P m P m3*
unfolding ‹$m1 = m$› ‹$m2 = m$›
using *inv_pair_P_D*[*OF prems*] **by** (*auto elim*: *lift_p_P*[*OF move12_inv*])
from 3(3)[*unfolded* ‹$m2 = m$›] **have** *mem2.map_of ?m3* $\subseteq_m$ *mem1.map_of m*
by − (*erule map_le_trans*[*OF update_correct*(3)[*OF* ‹*P m3*›] *move12_correct*(2)[*OF* ‹*P m*›]])
with 3 *prems* **show** *?thesis*
unfolding ‹$m1 = m$› ‹$m2 = m$› *map_le_def map_of2_def*
apply *auto*
apply (*frule move12_keys*(2), *simp*)
by (*metis*
domI inv_pair_def map_of1_def surjective_pairing
inv_pair_move12 move12_keys(2) *update_keys*(2)
)
qed
moreover from *prems* 3 **have** *dom* (*map_of1 ?m3*) ∩ *dom* (*map_of1 m*) = {}
unfolding ‹$m1 = m$› ‹$m2 = m$›

by (*smt inv_pair_P_D disjoint_iff_not_equal map_of*1*_def surjective_pairing domIff*
*fst_conv inv_pair_move*12 *move*12*_keys update_keys*
)
moreover from 3 **have** $k \notin$ *dom* (*map_of*1 *m*)
by (*simp add*: *domIff map_of*1*_def*)
moreover from 3 *prems* **have** *inv_pair ?m*3
unfolding ‹*m*2 = *m*›
by (*metis inv_pair_move*12 *inv_pair_update*1 *move*12*_keys*(1) *fst_conv surjective_pairing*)
ultimately show *?case*
unfolding ‹*m*1 = *m*› ‹*m*2 = *m*› **using** *disjoint*
apply (*subst map_of_eq_pair*[*symmetric*])
defer
apply (*subst map_of_eq_pair*[*symmetric*])
apply (*rule prems*)
apply (*subst* (2) *map_add_comm*)
defer
apply (*subst map_add_upd*2[*symmetric*])
apply (*auto intro*: *map_add_mono*)
done
qed
qed

lemma *emptyI*:
assumes *inv_pair m mem*1*.map_of m* $\subseteq_m$ *Map.empty mem*2*.map_of m* $\subseteq_m$ *Map.empty*
shows *pair.map_of m* $\subseteq_m$ *Map.empty*
using *assms* **by** (*auto simp*: *map_of*1*_def map_of*2*_def map_le_def map_of_eq_pair*[*symmetric*])

end

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

context *mem_correct_empty*
begin

context
fixes *key* :: '*a* ⇒ '*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 ⇒ *let k′* = *key k in*
if k′ = *k2 then case State_Monad.run_state* (*lookup k*) *m2 of* (*v*, *m*) ⇒ (*v*, *Pair_Storage k1 k2 m1 m*)
else if k′ = *k1 then case State_Monad.run_state* (*lookup k*) *m1 of* (*v*, *m*) ⇒ (*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 (λ *mem*.
(
case mem of Pair_Storage k1 k2 m1 m2 ⇒ *let k′* = *key k in*
if k′ = *k1 then case State_Monad.run_state* (*update k v*) *m1 of* (_, *m*) ⇒ ((), *Pair_Storage k1 k2 m m2*)
else if k′ = *k2 then case State_Monad.run_state* (*update k v*) *m2 of* (_, *m*) ⇒ ((),*Pair_Storage k1 k2 m1 m*)
else case State_Monad.run_state (*update k v*) *empty of* (_, *m*) ⇒ ((), *Pair_Storage k′ k1 m m1*)
)
)

interpretation *pair*: *state_mem_defs lookup_pair update_pair* **.**

definition
inv_pair p = (*case p of Pair_Storage k1 k2 m1 m2* ⇒
key ‘ *dom* (*map_of m1*) ⊆ {*k1*} ∧ *key* ‘ *dom* (*map_of m2*) ⊆ {*k2*} ∧ *k1* ≠ *k2* ∧ *P m1* ∧ *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* *k*1 *k*2 *m*1 *m*2)
using *that*
unfolding *pair.map_of_def*
unfolding *lookup_pair_def inv_pair_def map_of_def map_le_def dom_def map_add_def*
apply *auto*
apply (*auto* 4 6 *split*: *prod.split_asm if_split_asm option.split simp*: *Let_def*)
done

lemma *pair_le_map_of*:
*map_of m*1 ++ *map_of m*2 $\subseteq_m$ *pair.map_of* (*Pair_Storage k*1 *k*2 *m*1 *m*2)
if *inv_pair* (*Pair_Storage k*1 *k*2 *m*1 *m*2)
using *that*
unfolding *pair.map_of_def*
unfolding *lookup_pair_def inv_pair_def map_of_def map_le_def dom_def map_add_def*
by (*auto* 4 5 *split*: *prod.split_asm if_split_asm option.split simp*: *Let_def*)

lemma *map_of_eq_pair*:
*map_of m*1 ++ *map_of m*2 = *pair.map_of* (*Pair_Storage k*1 *k*2 *m*1 *m*2)
if *inv_pair* (*Pair_Storage k*1 *k*2 *m*1 *m*2)
using *that*
unfolding *pair.map_of_def*
unfolding *lookup_pair_def inv_pair_def map_of_def map_le_def dom_def map_add_def*
by (*auto* 4 7 *split*: *prod.split_asm if_split_asm option.split simp*: *Let_def*)

lemma *inv_pair_neq*[*simp*, *dest*]:
False **if** *inv_pair* (*Pair_Storage k k x y*)
using *that* **unfolding** *inv_pair_def* **by** *auto*

lemma *inv_pair_P_D*1:
*P m*1 **if** *inv_pair* (*Pair_Storage k*1 *k*2 *m*1 *m*2)
using *that* **unfolding** *inv_pair_def* **by** *auto*

lemma *inv_pair_P_D*2:
*P m*2 **if** *inv_pair* (*Pair_Storage k*1 *k*2 *m*1 *m*2)
using *that* **unfolding** *inv_pair_def* **by** *auto*

lemma *inv_pair_domD*[*intro*]:
dom (*map_of m*1) ∩ *dom* (*map_of m*2) = {} **if** *inv_pair* (*Pair_Storage k*1 *k*2 *m*1 *m*2)

```
  using that unfolding inv_pair_def by fastforce

lemma mem_correct_pair:
  mem_correct lookup_pair update_pair inv_pair
proof (standard, goal_cases)
  case (1 k) — Lookup invariant
  with lookup_inv[of k] show ?case
    unfolding lookup_pair_def Let_def
   by (auto intro!: lift_pI split: pair_storage.split_asm if_split_asm prod.split_asm)
       (auto dest: lift_p_P simp: inv_pair_def,
         (force dest!: lookup_correct[of _ k] map_le_implies_dom_le)+
       )
next
  case (2 k v) — Update invariant
   with update_inv[of k v] update_correct[OF P_empty, of k v] P_empty
show ?case
    unfolding update_pair_def Let_def
   by (auto intro!: lift_pI split: pair_storage.split_asm if_split_asm prod.split_asm)
       (auto dest: lift_p_P simp: inv_pair_def,
      (force dest: lift_p_P dest!: update_correct[of _ k v] map_le_implies_dom_le)+
       )
next
  case (3 m k)
  {
    fix m1 v1 m1' m2 v2 m2' k1 k2
    assume assms:
    State_Monad.run_state (lookup k) m1 = (v1, m1') State_Monad.run_state
(lookup k) m2 = (v2, m2')
      inv_pair (Pair_Storage k1 k2 m1 m2)
    from assms have P m1 P m2
      by (auto intro: inv_pair_P_D1 inv_pair_P_D2)
    have [intro]: map_of m1' ⊆m map_of m1 map_of m2' ⊆m map_of m2
      using lookup_correct[OF ‹P m1›, of k] lookup_correct[OF ‹P m2›, of
k] assms by auto
    from inv_pair_domD[OF assms(3)] have 1: dom (map_of m1') ∩ dom
(map_of m2) = {}
     by (metis (no_types) ‹map_of m1' ⊆m map_of m1› disjoint_iff_not_equal
domIff map_le_def)
    have inv1: inv_pair (Pair_Storage (key k) k2 m1' m2) if k2 ≠ key k k1
= key k
      using that ‹P m1› ‹P m2› unfolding inv_pair_def
      apply clarsimp
      apply safe
      subgoal for x' y
```

using *assms*(1,3) *lookup_correct*[*OF* ‹*P m*1›, *of k*, *THEN map_le_implies_dom_le*]
unfolding *inv_pair_def* **by** *auto*
subgoal for *x′ y*
using *assms*(3) **unfolding** *inv_pair_def* **by** *fastforce*
using *lookup_inv*[*of k*] *assms* **unfolding** *lift_p_def* **by** *force*
have *inv*2: *inv_pair* (*Pair_Storage k*1 (*key k*) *m*1 *m*2′) **if** *k*2 = *key k k*1 ≠ *key k*
using *that* ‹*P m*1› ‹*P m*2› **unfolding** *inv_pair_def*
apply *clarsimp*
apply *safe*
subgoal for *x′ y*
using *assms*(3) **unfolding** *inv_pair_def* **by** *fastforce*
subgoal for *x x′ y*
using *assms*(2,3) *lookup_correct*[*OF* ‹*P m*2›, *of k*, *THEN map_le_implies_dom_le*]
unfolding *inv_pair_def* **by** *fastforce*
using *lookup_inv*[*of k*] *assms* **unfolding** *lift_p_def* **by** *force*
have *A*:
pair.map_of (*Pair_Storage* (*key k*) *k*2 *m*1′ *m*2) $\subseteq_m$ *pair.map_of* (*Pair_Storage* (*key k*) *k*2 *m*1 *m*2)
if *k*2 ≠ *key k k*1 = *key k*
using *inv*1 *assms*(3) 1
by (*auto intro*: *map_add_mono map_le_refl simp*: *that map_of_eq_pair*[*symmetric*])
have *B*:
pair.map_of (*Pair_Storage k*1 (*key k*) *m*1 *m*2′) $\subseteq_m$ *pair.map_of* (*Pair_Storage k*1 (*key k*) *m*1 *m*2)
if *k*2 = *key k k*1 ≠ *key k*
using *inv*2 *assms*(3) *that*
by (*auto intro*: *map_add_mono map_le_refl simp*: *map_of_eq_pair*[*symmetric*] *dest*: *inv_pair_domD*)
note *A B*
}
with ‹*inv_pair m*› **show** *?case*
by (*auto split*: *pair_storage.split if_split prod.split simp*: *Let_def lookup_pair_def*)
next
case (4 *m k v*)
{
fix *m*1 *v*1 *m*1′ *m*2 *v*2 *m*2′ *m*3 *k*1 *k*2
assume *assms*:
State_Monad.run_state (*update k v*) *m*1 = ((), *m*1′) *State_Monad.run_state* (*update k v*) *m*2 = ((), *m*2′)
State_Monad.run_state (*update k v*) *empty* = ((), *m*3)
inv_pair (*Pair_Storage k*1 *k*2 *m*1 *m*2)
from *assms* **have** *P m*1 *P m*2
by (*auto intro*: *inv_pair_P_D*1 *inv_pair_P_D*2)

from *assms*(3) *P_empty update_inv*[*of k v*] **have** *P m*3
unfolding *lift_p_def* **by** *auto*
have [*intro*]: *map_of m*1′ $\subseteq_m$ (*map_of m*1)(*k* ↦ *v*) *map_of m*2′ $\subseteq_m$ (*map_of m*2)(*k* ↦ *v*)
using *update_correct*[*OF* ‹*P m*1›, *of k v*] *update_correct*[*OF* ‹*P m*2›, *of k v*] *assms* **by** *auto*
have *map_of m*3 $\subseteq_m$ (*map_of empty*)(*k* ↦ *v*)
using *assms*(3) *update_correct*[*OF P_empty*, *of k v*] **by** *auto*
also have ... $\subseteq_m$ (*map_of m*2)(*k* ↦ *v*)
using *empty_correct* **by** (*auto elim*: *map_le_trans intro*!: *map_le_upd*)
finally have *map_of m*3 $\subseteq_m$ (*map_of m*2)(*k* ↦ *v*) **.**
have 1: *dom* (*map_of m*1) ∩ *dom* ((*map_of m*2)(*k* ↦ *v*)) = {} **if** *k*1 ≠ *key k*
using *assms*(4) *that* **by** (*force simp*: *inv_pair_def*)
have 2: *dom* (*map_of m*3) ∩ *dom* (*map_of m*1) = {} **if** *k*1 ≠ *key k*
using ‹*local.map_of m*3 $\subseteq_m$ (*map_of empty*)(*k* ↦ *v*)› *assms*(4) *that*
by (*fastforce dest*!: *map_le_implies_dom_le simp*: *inv_pair_def*)
have *inv*: *inv_pair* (*Pair_Storage* (*key k*) *k*1 *m*3 *m*1) **if** *k*2 ≠ *key k k*1 ≠ *key k*
using *that* ‹*P m*1› ‹*P m*2› ‹*P m*3› **unfolding** *inv_pair_def*
apply *clarsimp*
apply *safe*
subgoal for *x x*′ *y*
using *assms*(3) *update_correct*[*OF P_empty*, *of k v*, *THEN map_le_implies_dom_le*] *empty_correct*
by (*auto dest*: *map_le_implies_dom_le*)
subgoal for *x x*′ *y*
using *assms*(4) **unfolding** *inv_pair_def* **by** *fastforce*
done
have *A*:
pair.map_of (*Pair_Storage* (*key k*) *k*1 *m*3 *m*1) $\subseteq_m$ (*pair.map_of* (*Pair_Storage k*1 *k*2 *m*1 *m*2))(*k* ↦ *v*)
if *k*2 ≠ *key k k*1 ≠ *key k*
using *inv assms*(4) ‹*map_of m*3 $\subseteq_m$ (*map_of m*2)(*k* ↦ *v*)› 1
apply (*simp add*: *that map_of_eq_pair*[*symmetric*])
apply (*subst map_add_upd*[*symmetric*], *subst Map.map_add_comm*, *rule* 2, *rule that*)
by (*rule map_add_mono*; *auto*)
have *inv*1: *inv_pair* (*Pair_Storage* (*key k*) *k*2 *m*1′ *m*2) **if** *k*2 ≠ *key k k*1 = *key k*
using *that* ‹*P m*1› ‹*P m*2› **unfolding** *inv_pair_def*
apply *clarsimp*
apply *safe*
subgoal for *x*′ *y*

using *assms*(1,4) *update_correct*[*OF* ‹*P m*1›, *of k v*, *THEN map_le_implies_dom_le*]
unfolding *inv_pair_def* **by** *auto*
subgoal for *x′ y*
using *assms*(4) **unfolding** *inv_pair_def* **by** *fastforce*
using *update_inv*[*of k v*] *assms* **unfolding** *lift_p_def* **by** *force*
have *inv*2: *inv_pair* (*Pair_Storage k*1 (*key k*) *m*1 *m*2′) **if** *k*2 = *key k k*1 ≠ *key k*
using *that* ‹*P m*1› ‹*P m*2› **unfolding** *inv_pair_def*
apply *clarsimp*
apply *safe*
subgoal for *x′ y*
using *assms*(4) **unfolding** *inv_pair_def* **by** *fastforce*
subgoal for *x x′ y*
using *assms*(2,4) *update_correct*[*OF* ‹*P m*2›, *of k v*, *THEN map_le_implies_dom_le*]
unfolding *inv_pair_def* **by** *fastforce*
using *update_inv*[*of k v*] *assms* **unfolding** *lift_p_def* **by** *force*
have *C*:
pair.map_of (*Pair_Storage* (*key k*) *k*2 *m*1′ *m*2) ⊆$_m$
(*pair.map_of* (*Pair_Storage* (*key k*) *k*2 *m*1 *m*2))(*k* ↦ *v*)
if *k*2 ≠ *key k k*1 = *key k*
using *inv*1[*OF that*] *assms*(4) ‹*inv_pair m*›
by (*simp add*: *that map_of_eq_pair*[*symmetric*])
(*subst map_add_upd*2[*symmetric*]; *force simp*: *inv_pair_def intro*: *map_add_mono map_le_refl*)
have *B*:
pair.map_of (*Pair_Storage k*1 (*key k*) *m*1 *m*2′) ⊆$_m$
(*pair.map_of* (*Pair_Storage k*1 (*key k*) *m*1 *m*2))(*k* ↦ *v*)
if *k*2 = *key k k*1 ≠ *key k*
using *inv*2[*OF that*] *assms*(4)
by (*simp add*: *that map_of_eq_pair*[*symmetric*])
(*subst map_add_upd*[*symmetric*]; *rule map_add_mono*; *force simp*: *inv_pair_def*)
note *A B C*
}
with ‹*inv_pair m*› **show** *?case*
by (*auto split*: *pair_storage.split if_split prod.split simp*: *Let_def update_pair_def*)
qed

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
    ⟶ 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 −
  have a ∗ b0 + b < (Suc a) ∗ b0
    using b by auto
  also have . . . ≤ a0 ∗ b0
    using a[THEN Suc_leI, THEN mult_le_mono1] .
  finally show ?thesis .
qed

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 ∧ b = d
proof (rule conjI)
  { fix a b c d :: nat
    assume ac: a < c and b: b < b0
    have a ∗ b0 + b < (Suc a) ∗ b0
      using b by auto
    also have . . . ≤ c ∗ b0
      using ac[THEN Suc_leI, THEN mult_le_mono1] .
    also have . . . ≤ c ∗ b0 + d
      by auto
    finally have a ∗ b0 + b ≠ c ∗ b0 + d
      by auto
  } note ac = this

  { assume a ≠ c
```

then consider (*le*) $a < c$ | (*ge*) $a > c$
by *fastforce*
hence *False* **proof** *cases*
case *le* **show** *?thesis* **using** *ac*[*OF le assms*(1)] *assms*(3) **..**
next
case *ge* **show** *?thesis* **using** *ac*[*OF ge assms*(2)] *assms*(3)[*symmetric*]
..
qed
}

then show $a = c$
by *auto*

with *assms*(3) **show** $b = d$
by *auto*
qed

locale *prod_order_def* =
*order*0: *ord less_eq*0 *less*0 +
*order*1: *ord less_eq*1 *less*1
for *less_eq*0 *less*0 *less_eq*1 *less*1
begin

fun *less* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow$ *bool* **where**
less (*a*,*b*) (*c*,*d*) $\longleftrightarrow$ *less*0 *a c* $\wedge$ *less*1 *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_eq*0 *less*0 *less_eq*1 *less*1 +
*order*0: *order less_eq*0 *less*0 +
*order*1: *order less_eq*1 *less*1
for *less_eq*0 *less*0 *less_eq*1 *less*1
begin

sublocale *order less_eq less*
proof qed *fastforce*+

end

locale *option_order* =
  *order*0: *order less_eq*0 *less*0
  for *less_eq*0 *less*0
begin

fun *less_eq_option* :: *'a option* ⇒ *'a option* ⇒ *bool* **where**
  less_eq_option None _ ⟷ *True*
| *less_eq_option* (*Some _*) *None* ⟷ *False*
| *less_eq_option* (*Some a*) (*Some b*) ⟷ *less_eq*0 *a b*

fun *less_option* :: *'a option* ⇒ *'a option* ⇒ *bool* **where**
  less_option ao bo ⟷ *less_eq_option ao bo* ∧ *ao* ≠ *bo*

sublocale *order less_eq_option less_option*
  apply *standard*
  subgoal for *x y* **by** (*cases x*; *cases y*) *auto*
  subgoal for *x* **by** (*cases x*) *auto*
  subgoal for *x y z* **by** (*cases x*; *cases y*; *cases z*) *auto*
  subgoal for *x y* **by** (*cases x*; *cases y*) *auto*
  done

end

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

definition *in_bound* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a bound* ⇒ *'a* ⇒ *bool* **where**
  in_bound less_eq less bound x ≡ *case bound of Bound l r* ⇒ *less_eq l x* ∧ *less x r* **for** *less_eq less*

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

locale *index_locale* = *index_locale_def* + *idx_ord*: *order* +
  assumes *idx_valid*: *in_bound less_eq less bound x* ⟹ *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*⟧ ⟹ *x* = *y*

locale *prod_index_def* =
  *index*0: *index_locale_def less_eq*0 *less*0 *idx*0 *size*0 +
  *index*1: *index_locale_def less_eq*1 *less*1 *idx*1 *size*1

for *less_eq0 less0 idx0 size0 less_eq1 less1 idx1 size1*
begin

fun *idx* :: ('*a* × '*b*) *bound* ⇒ '*a* × '*b* ⇒ *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* × '*b*) *bound* ⇒ *nat* **where**
size (*Bound* (*l0*, *r0*) (*l1*, *r1*)) = *size0* (*Bound l0 l1*) ∗ *size1* (*Bound r0 r1*)

end

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

sublocale *prod_order less_eq0 less0 less_eq1 less1* **..**

sublocale *index_locale less_eq less idx size* **proof**
{ fix *ab* :: '*a* × '*b* **and** *bound* :: ('*a* × '*b*) *bound*
assume *bound*: *in_bound less_eq less bound ab*

obtain *a b l0 r0 l1 r1* **where** *defs*:*ab* = (*a*, *b*) *bound* = *Bound* (*l0*, *r0*) (*l1*, *r1*)
by (*cases ab*; *cases bound*) *auto*

with *bound* **have** *a*: *in_bound less_eq0 less0* (*Bound l0 l1*) *a* **and** *b*: *in_bound less_eq1 less1* (*Bound r0 r1*) *b*
unfolding *in_bound_def* **by** *auto*

have *idx* (*Bound* (*l0*, *r0*) (*l1*, *r1*)) (*a*, *b*) < *size* (*Bound* (*l0*, *r0*) (*l1*, *r1*))
using *index_mono*[*OF index0*.*idx_valid*[*OF a*] *index1*.*idx_valid*[*OF b*]] **by** *auto*

thus *idx bound ab* < *size bound*
unfolding *defs* **.**
}

{ fix *ab cd* :: '*a* × '*b* **and** *bound* :: ('*a* × '*b*) *bound*
assume *bound*: *in_bound less_eq less bound ab in_bound less_eq less bound cd*

```
    and idx_eq: idx bound ab = idx bound cd

  obtain a b c d l0 r0 l1 r1 where
    defs: ab = (a, b) cd = (c, d) bound = Bound (l0, l1) (r0, r1)
    by (cases ab; cases cd; cases bound) auto

  from defs bound have
        a: in_bound less_eq0 less0 (Bound l0 r0) a
    and b: in_bound less_eq1 less1 (Bound l1 r1) b
    and c: in_bound less_eq0 less0 (Bound l0 r0) c
    and d: in_bound less_eq1 less1 (Bound l1 r1) d
    unfolding in_bound_def by auto

  from index_eq_iff[OF index1.idx_valid[OF b] index1.idx_valid[OF d]
idx_eq[unfolded defs, simplified]]
   have ac: idx0 (Bound l0 r0) a = idx0 (Bound l0 r0) c and bd: idx1
(Bound l1 r1) b = idx1 (Bound l1 r1) d by auto
  show ab = cd
    unfolding defs using index0.idx_inj[OF a c ac] index1.idx_inj[OF b
d bd] by auto
 }
qed
end

locale option_index =
 index0: index_locale less_eq0 less0 idx0 size0
 for less_eq0 less0 idx0 size0
begin

fun idx :: 'a option bound ⇒ 'a option ⇒ nat where
 idx (Bound (Some l) (Some r)) (Some a) = idx0 (Bound l r) a
| idx _ _ = undefined


end

locale nat_index_def = ord (≤) :: nat ⇒ nat ⇒ bool (<)
begin

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

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

sublocale *index_locale* (≤) (<) *idx size*
proof qed (*auto simp*: *in_bound_def split*: *bound.splits*)

end

locale *nat_index* = *nat_index_def* + *order* (≤) :: *nat* ⇒ *nat* ⇒ *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 qed (*auto simp*: *in_bound_def split*: *bound.splits*)

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*
  using *is_locale* **.**

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

unfolding *injective_def*
unfolding *checked_idx_def*
using *idx_inj* **by** (*fastforce split*: *if_splits*)
end

instantiation *nat* :: *index*
begin

interpretation *nat_index* **..**
thm *index_locale_axioms*

definition [*simp*]: *less_eq_nat* ≡ (≤) :: *nat* ⇒ *nat* ⇒ *bool*
definition [*simp*]: *less_nat* ≡ (<) :: *nat* ⇒ *nat* ⇒ *bool*
definition [*simp*]: *idx_nat* ≡ *idx*
definition *size_nat* **where** [*simp*]: *size_nat* ≡ *size*

instance by (*standard*, *simp*, *fact index_locale_axioms*)

end

instantiation *int* :: *index*
begin

interpretation *int_index* **..**
thm *index_locale_axioms*

definition [*simp*]: *less_eq_int* ≡ (≤) :: *int* ⇒ *int* ⇒ *bool*
definition [*simp*]: *less_int* ≡ (<) :: *int* ⇒ *int* ⇒ *bool*
definition [*simp*]: *idx_int* ≡ *idx*
definition [*simp*]: *size_int* ≡ *size*

lemmas *size_int* = *size.simps*

instance by (*standard*, *simp*, *fact index_locale_axioms*)
end

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

interpretation *prod_index*
less_eq::′*a* ⇒ ′*a* ⇒ *bool less idx size*
less_eq::′*b* ⇒ ′*b* ⇒ *bool less idx size*
by (*rule prod_index.intro*; *fact is_locale*)
thm *index_locale_axioms*

definition [*simp*]: *less_eq_prod* ≡ *less_eq*
definition [*simp*]: *less_prod* ≡ *less*
definition [*simp*]: *idx_prod* ≡ *idx*
definition [*simp*]: *size_prod* ≡ *size* **for** *size_prod*

lemmas *size_prod* = *size.simps*

instance by (*standard*, *simp*, *fact index_locale_axioms*)

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)
by (*simp add*: *bounded_index.size_def*,*unfold size_int_def*[*symmetric*] *size_prod*,*simp add*: *size_int*)

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*
using *that* **unfolding** *map_le_def* **by** *auto*

lemma *result_of_return*[*simp*]:
result_of (*Heap_Monad.return x*) *h* = *x*
by (*simp add*: *execute_simps*)

lemma *get_result_of_lookup*:
result_of (!*r*) *heap* = *x* **if** *Ref.get heap r* = *x*
using *that* **by** (*auto simp*: *execute_simps*)

context
fixes *size* :: *nat*
and *to_index* :: ('*k2* :: *heap*) ⇒ *nat*
begin

definition
mem_empty = (*Array.new size* (*None* :: ('*v* :: *heap*) *option*))

lemma *success_empty*[*intro*]:
success mem_empty heap
unfolding *mem_empty_def* **by** (*auto intro*: *success_intros*)

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*
unfolding *mem_empty_def* **by** (*auto simp*: *execute_simps Array.length_alloc*)

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*
apply (*subst execute_nth*(1))
apply (*simp add*: *length_mem_empty that*)
apply (*simp add*: *execute_simps mem_empty_def Array.get_alloc that*)
done

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*
apply *standard*
subgoal *lookup_inv*
unfolding *State_Heap.lift_p_def mem_lookup_def* **by** (*simp add*: *Let_def execute_simps*)
subgoal *update_inv*
unfolding *State_Heap.lift_p_def mem_update_def* **by** (*simp add*: *Let_def execute_simps*)
subgoal for *k heap*
unfolding *heap_mem_defs.map_of_heap_def map_le_def mem_lookup_def*
by (*auto simp*: *execute_simps Let_def split*: *if_split_asm*)
subgoal for *heap k*
unfolding *heap_mem_defs.map_of_heap_def map_le_def mem_lookup_def mem_update_def*
apply (*auto simp*: *execute_simps Let_def length_def split*: *if_split_asm*)
apply (*subst* (*asm*) *nth_list_update_neq*)
using *injective*[*unfolded injective_def*] **apply** *auto*

done
done

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*
apply *standard*
subgoal
apply (*rule map_emptyI*)
unfolding *map_of_heap_def mem_lookup_def* **by** (*auto simp*: *Let_def nth_mem_empty*)
subgoal
by (*simp add*: *length_mem_empty*)
done

lemmas *array_heap_emptyI* = *heap_correct_empty_axioms*

context
fixes *dp* :: *'k2* $\Rightarrow$ *'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*
by *standard*

lemmas *array_consistentI* = *dp_consistency_heap_empty_axioms*

end

end

end

end

lemma *execute_bind_success'*:
assumes *success f h execute* (*f* $\gg=$ *g*) *h* = *Some* (*y*, *h''*)

obtains $x\ h'$ **where** $execute\ f\ h = Some\ (x,\ h')\ execute\ (g\ x)\ h' = Some\ (y,\ h'')$
using *assms* **by** (*auto simp*: *execute_simps elim*: *successE*)

lemma *success_bind_I*:
assumes *success f h*
and $\bigwedge x\ h'.\ execute\ f\ h = Some\ (x,\ h') \Longrightarrow success\ (g\ x)\ h'$
shows $success\ (f \gg= g)\ h$
by (*rule successE*[*OF assms*(1)]) (*auto elim*: *assms*(2) *intro*: *success_bind_executeI*)

definition
alloc_pair a b ≡ *do* {
*r*1 ← *ref a*;
*r*2 ← *ref b*;
return (*r*1, *r*2)
}

lemma *alloc_pair_alloc*:
*Ref.get heap′ r*1 = *a Ref.get heap′ r*2 = *b*
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap′*)
using *that* **unfolding** *alloc_pair_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success′*[*OF success_refI*])
(*metis Ref.get_alloc fst_conv get_alloc_neq next_present present_alloc_neq snd_conv*)+

lemma *alloc_pairD*1:
r =!= *r*1 ∧ *r* =!= *r*2 ∧ *Ref.present heap′ r*
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap′*) *Ref.present heap r*
using *that* **unfolding** *alloc_pair_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success′*[*OF success_refI*])
(*metis next_fresh noteq_I Ref.present_alloc snd_conv*)+

lemma *alloc_pairD*2:
*r*1 =!= *r*2 ∧ *Ref.present heap′ r*2 ∧ *Ref.present heap′ r*1
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap′*)
using *that* **unfolding** *alloc_pair_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success′*[*OF success_refI*])
(*metis next_fresh next_present noteq_I Ref.present_alloc snd_conv*)+

lemma *alloc_pairD*3:
Array.present heap′ r
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap′*) *Array.present heap r*

using *that* **unfolding** *alloc_pair_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success*′[*OF success_refI*])
(*metis array_present_alloc snd_conv*)

lemma *alloc_pairD4*:
Ref.get heap′ *r* = *x*
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap*′)
Ref.get heap r = *x Ref.present heap r*
using *that* **unfolding** *alloc_pair_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success*′[*OF success_refI*])
(*metis Ref.not_present_alloc Ref.present_alloc get_alloc_neq noteq_I snd_conv*)

lemma *alloc_pair_array_get*:
Array.get heap′ *r* = *x*
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap*′) *Array.get heap r* = *x*
using *that* **unfolding** *alloc_pair_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success*′[*OF success_refI*])
(*metis array_get_alloc snd_conv*)

lemma *alloc_pair_array_length*:
Array.length heap′ *r* = *Array.length heap r*
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap*′)
using *that* **unfolding** *alloc_pair_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success*′[*OF success_refI*])
(*metis Ref.length_alloc snd_conv*)

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* ((*r*1, *r*2), *heap*′)
using *alloc_pair_array_get*[*OF that*(1) *HOL.refl*, *of r*] *alloc_pair_array_length*[*OF that*(1), *of r*]
by (*cases* (λ*h*. *i* < *Array.length h r*) *heap*; *simp add*: *execute_simps Array.nth_def*)

lemma *succes_alloc_pair*[*intro*]:
success (*alloc_pair a b*) *heap*
unfolding *alloc_pair_def* **by** (*auto intro*: *success_intros success_bind_I*)

definition
*init_state_inner k*1 *k*2 *m*1 *m*2 ≡ *do* {
(*k_ref*1, *k_ref*2) ← *alloc_pair k*1 *k*2;
(*m_ref*1, *m_ref*2) ← *alloc_pair m*1 *m*2;

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*
using *assms* **unfolding** *init_state_inner_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success'*[*OF succes_alloc_pair*])
(*auto intro*: *alloc_pair_alloc dest*: *alloc_pairD2 elim*: *alloc_pairD4*)

lemma *init_state_inner_distinct*:
assumes
execute (*init_state_inner k1 k2 m1 m2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
m_ref1 =!= *m_ref2* ∧ *m_ref1* =!= *k_ref1* ∧ *m_ref1* =!= *k_ref2* ∧ *m_ref2* =!= *k_ref1*
∧ *m_ref2* =!= *k_ref2* ∧ *k_ref1* =!= *k_ref2*
using *assms* **unfolding** *init_state_inner_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success'*[*OF succes_alloc_pair*])
(*blast dest*: *alloc_pairD1 alloc_pairD2 intro*: *noteq_sym*)+

lemma *init_state_inner_present*:
assumes
execute (*init_state_inner k1 k2 m1 m2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
Ref.present heap' k_ref1 *Ref.present heap' k_ref2*
Ref.present heap' m_ref1 *Ref.present heap' m_ref2*
using *assms* **unfolding** *init_state_inner_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success'*[*OF succes_alloc_pair*])
(*blast dest*: *alloc_pairD1 alloc_pairD2*)+

lemma *inite_state_inner_present'*:
assumes
execute (*init_state_inner k1 k2 m1 m2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
Array.present heap a
shows

Array.present heap′ a
using *assms* **unfolding** *init_state_inner_def*
by (*auto simp*: *execute_simps elim*!: *execute_bind_success′*[*OF succes_alloc_pair*] *alloc_pairD*3)

lemma *succes_init_state_inner*[*intro*]:
success (*init_state_inner k*1 *k*2 *m*1 *m*2) *heap*
unfolding *init_state_inner_def* **by** (*auto* 4 3 *intro*: *success_intros success_bind_I*)

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′*)
using *that* **unfolding** *init_state_inner_def*
by (*auto simp*: *execute_simps alloc_pair_nth elim*!: *execute_bind_success′*[*OF succes_alloc_pair*])

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*
unfolding *init_state_def* **by** (*auto intro*: *success_intros success_bind_I*)

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
using *assms* **unfolding** *init_state_def inv_distinct_def*
by (*elim execute_bind_success′*[*OF success_empty*] *init_state_inner_distinct*)

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
using *assms* **unfolding** *init_state_def*
by (*auto*
simp: *execute_simps elim*!: *execute_bind_success*′[*OF success_empty*]
dest: *init_state_inner_present*
)

lemma *empty_present*:
Array.present h′ *x* **if** *execute mem_empty heap* = *Some* (*x*, *h*′)
using *that* **unfolding** *mem_empty_def*
by (*auto simp*: *execute_simps*) (*metis Array.present_alloc fst_conv snd_conv*)

lemma *empty_present*′:
Array.present h′ *a* **if** *execute mem_empty heap* = *Some* (*x*, *h*′) *Array.present heap a*
using *that* **unfolding** *mem_empty_def*
by (*auto simp*: *execute_simps Array.present_def Array.alloc_def Array.set_def Let_def*)

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)
using *assms* **unfolding** *init_state_def*
by (*auto* 4 3
simp: *execute_simps init_state_inner_alloc elim*!: *execute_bind_success*′[*OF success_empty*]
dest: *inite_state_inner_present*′ *empty_present empty_present*′
)

lemma *init_state_neq*:
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.get heap′ m_ref1 =!!= Ref.get heap′ m_ref2
using *assms* **unfolding** *init_state_def*
by (*auto* 4 3
simp: *execute_simps init_state_inner_alloc elim*!: *execute_bind_success*′[*OF success_empty*]
dest: *inite_state_inner_present*′ *empty_present empty_present*′
)
(*metis empty_present execute_new fst_conv mem_empty_def option.inject present_alloc_noteq*)

lemma *present_alloc_get*:
Array.get heap′ *a* = *Array.get heap a*
if *Array.alloc xs heap* = (*a*′, *heap*′) *Array.present heap a*
using *that* **by** (*auto simp*: *Array.alloc_def Array.present_def Array.get_def Let_def Array.set_def*)

lemma *init_state_length*:
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.length heap′ (*Ref.get heap*′ *m_ref*1) = *size*
Array.length heap′ (*Ref.get heap*′ *m_ref*2) = *size*
using *assms* **unfolding** *init_state_def*
apply (*auto*
simp: *execute_simps init_state_inner_alloc elim*!: *execute_bind_success*′[*OF success_empty*]
dest: *inite_state_inner_present*′ *empty_present empty_present*′
)
apply (*auto*
simp: *execute_simps init_state_inner_def alloc_pair_def mem_empty_def Array.length_def*
elim!: *execute_bind_success*′[*OF success_refI*]
)
apply (*metis*
Array.alloc_def Array.get_set_eq Array.present_alloc array_get_alloc fst_conv length_replicate
present_alloc_get snd_conv
)+
done

context
fixes *key*1 :: ′*k* ⇒ (′*k*1 :: *heap*) **and** *key*2 :: ′*k* ⇒ ′*k*2
and *m_ref*1 *m_ref*2 :: (′*v* :: *heap*) *option array ref*

and *k_ref*1 *k_ref*2 :: (′*k*1 :: *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
      m1 = Ref.get heap m_ref1;
      m2 = Ref.get heap m_ref2
    in Array.length heap m1 = size ∧ Array.length heap m2 = size
      ∧ Ref.present heap k_ref1 ∧ Ref.present heap k_ref2
      ∧ Ref.present heap m_ref1 ∧ Ref.present heap m_ref2
      ∧ Array.present heap m1 ∧ Array.present heap m2
      ∧ m1 =!!= m2
  )
```

definition
inv_pair heap ≡ *inv_pair_weak heap* ∧ *inv_distinct k_ref1 k_ref2 m_ref1 m_ref2*

lemma *init_state_inv*:
assumes
execute (*init_state k1 k2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows *inv_pair_weak heap'*
using *assms* **unfolding** *inv_pair_weak_def Let_def*
by (*auto intro*:
init_state_present init_state_present2 init_state_neq init_state_length
init_state_distinct

)

lemma *inv_pair_lengthD*1:
Array.length heap (*Ref.get heap m_ref*1) = *size* **if** *inv_pair_weak heap*
using *that* **unfolding** *inv_pair_weak_def* **by** (*auto simp*: *Let_def*)

lemma *inv_pair_lengthD*2:
Array.length heap (*Ref.get heap m_ref*2) = *size* **if** *inv_pair_weak heap*
using *that* **unfolding** *inv_pair_weak_def* **by** (*auto simp*: *Let_def*)

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*
using *that* **unfolding** *inv_pair_weak_def* **by** (*auto simp*: *Let_def*)

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*
using *that* **unfolding** *inv_pair_weak_def* **by** (*auto simp*: *Let_def*)

lemma *inv_pair_not_eqD*:
*Ref.get heap m_ref*1 =!!= *Ref.get heap m_ref*2 **if** *inv_pair_weak heap*
using *that* **unfolding** *inv_pair_weak_def* **by** (*auto simp*: *Let_def*)

definition *lookup*1 *k* ≡ *state_of* (*do* {*m* ← !*m_ref*1; *mem_lookup m k*})

definition *lookup*2 *k* ≡ *state_of* (*do* {*m* ← !*m_ref*2; *mem_lookup m k*})

definition *update*1 *k v* ≡ *state_of* (*do* {*m* ← !*m_ref*1; *mem_update m k v*})

definition *update*2 *k v* ≡ *state_of* (*do* {*m* ← !*m_ref*2; *mem_update m k v*})

definition *move*12 *k* ≡ *state_of* (*do* {
*k*1 ← !*k_ref*1;
m ← *mem_empty*;
*m*1 ← !*m_ref*1;
*k_ref*2 := *k*1;
*k_ref*1 := *k*;
*m_ref*2 := *m*1;
*m_ref*1 := *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)
  unfolding state_of_def by simp

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
  using inv_distinct unfolding inv_distinct_def by auto

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
  using that unfolding Array.alloc_def Array.present_def
  apply (simp add: Let_def)
  apply (subst Array.get_set_neq)
  subgoal
    by (simp add: Array.noteq_def)
  subgoal
    unfolding Array.get_def by simp
  done

lemma [simp]:
  Ref.get (snd (Array.alloc xs heap)) r = Ref.get heap r if Ref.present heap
r
  using that unfolding Array.alloc_def Ref.present_def
  by (simp add: Let_def Ref.get_def Array.set_def)
```

lemma *alloc_present*:
Array.present (*snd* (*Array.alloc xs heap*)) *a* **if** *Array.present heap a*
using *that* **unfolding** *Array.present_def Array.alloc_def* **by** (*simp add*: *Let_def Array.set_def*)

lemma *alloc_present′*:
Ref.present (*snd* (*Array.alloc xs heap*)) *r* **if** *Ref.present heap r*
using *that* **unfolding** *Ref.present_def Array.alloc_def* **by** (*simp add*: *Let_def Array.set_def*)

lemma *length_get_upd*[*simp*]:
length (*Array.get* (*Array.update a i x heap*) *r*) = *length* (*Array.get heap r*)
unfolding *Array.get_def Array.update_def Array.set_def* **by** *simp*

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*
supply [*simp*] =
mem_empty_def state_mem_defs.map_of_def map_le_def
*move*12*_def update*1*_def update*2*_def lookup*1*_def lookup*2*_def get_k*1*_def get_k*2*_def*
mem_update_def mem_lookup_def
execute_bind_success[*OF success_newI*] *execute_simps Let_def Array.get_alloc length_def*
*inv_pair_presentD inv_pair_presentD*2
*Memory_Heap.lookup*1*_def Memory_Heap.lookup*2*_def Memory_Heap.mem_lookup_def*
apply *standard*
apply (*solve*1; *fail*)+
subgoal
apply (*rule lift_pI*)
unfolding *inv_pair_weak_def*
apply (*auto simp*:
intro: *alloc_present alloc_present′*
elim: *present_alloc_noteq*[*THEN Array.noteq_sym*]
)
done
apply (*rule lift_pI*, *unfold inv_pair_weak_def*, *auto split*: *if_split_asm*; *fail*)+
apply (*solve*1; *fail*)+
subgoal
using *injective*[*unfolded injective_def*] **by** − (*solve*1, *subst* (*asm*) *nth_list_update_neq*,

auto)
  subgoal
   using *injective*[*unfolded injective_def*] **by** − (*solve*1, *subst* (*asm*) *nth_list_update_neq*, *auto*)
   apply (*solve*1; *fail*)+
  done

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_k*2′ ≡ !*k_ref*2

definition *update*1′ *k v* ≡ *do* {*m* ← !*m_ref*1; *mem_update m k v*}

definition *update*2′ *k v* ≡ *do* {*m* ← !*m_ref*2; *mem_update m k v*}

definition *move*12′ *k* ≡ *do* {
    *k*1 ← !*k_ref*1;
    m ← *mem_empty*;
    *m*1 ← !*m_ref*1;
    *k_ref*2 := *k*1;
    *k_ref*1 := *k*;
    *m_ref*2 := *m*1;
    *m_ref*1 := *m*
  }

interpretation *heap_mem_defs inv_pair_weak lookup_pair update_pair*
.

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
  using *that* **unfolding** *rel_state_def*
  by (*auto split*: *option.split intro*: *lift_p_P*′′ *simp*: *success_def*)

lemma *inv_pair_iff*:

```
  inv_pair_weak = inv_pair
  unfolding inv_pair_def using inv_distinct by simp

lemma lift_p_inv_pairI:
  State_Heap.lift_p inv_pair m if State_Heap.lift_p inv_pair_weak m
  using that unfolding inv_pair_iff by simp

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
  using that
  unfolding lift_p_def DP_CRelVS.lift_p_def
  by (auto simp: success_def split: option.split)

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)
  using that by (blast intro: rel_state_ofI lift_p_success)

context
  includes lifting_syntax
begin

lemma [transfer_rule]:
  ((=) ===> rel_state (=)) move12 move12′
  unfolding move12_def move12′_def
  apply (intro rel_funI)
  apply simp
  apply (rule rel_state_ofI2)
  subgoal
    by (auto
        simp: mem_empty_def inv_pair_lengthD1 execute_simps Let_def
        intro: success_intros intro!: success_bind_I
       )
  subgoal
    using pair.move12_inv unfolding move12_def .
  done

lemma [transfer_rule]:
  ((=) ===> rel_state (rel_option (=))) lookup1 mem_lookup1
  unfolding lookup1_def mem_lookup1_def
  apply (intro rel_funI)
```

```
  apply (simp add: option.rel_eq)
  apply (rule rel_state_ofI2)
  subgoal
    by (auto 4 4
        simp: mem_lookup_def inv_pair_lengthD1 execute_simps Let_def
        intro: success_bind_executeI success_returnI Array.success_nthI
       )
  subgoal
    using pair.lookup_inv(1) unfolding lookup1_def .
  done

lemma [transfer_rule]:
  ((=) ===> rel_state (rel_option (=))) lookup2 mem_lookup2
  unfolding lookup2_def mem_lookup2_def
  apply (intro rel_funI)
  apply (simp add: option.rel_eq)
  apply (rule rel_state_ofI2)
  subgoal
    by (auto 4 3
        simp: mem_lookup_def inv_pair_lengthD2 execute_simps Let_def
        intro: success_intros intro!: success_bind_I
       )
  subgoal
    using pair.lookup_inv(2) unfolding lookup2_def .
  done

lemma [transfer_rule]:
  rel_state (=) get_k1 get_k1'
  unfolding get_k1_def get_k1'_def
  apply (rule rel_state_ofI2)
  subgoal
    by (auto intro: success_lookupI)
  subgoal
   unfolding get_k1_def[symmetric] by (auto dest: pair.get_state(1) intro:
lift_pI)
  done

lemma [transfer_rule]:
  rel_state (=) get_k2 get_k2'
  unfolding get_k2_def get_k2'_def
  apply (rule rel_state_ofI2)
  subgoal
    by (auto intro: success_lookupI)
  subgoal
```

```
   unfolding get_k2_def[symmetric] by (auto dest: pair.get_state(2) intro:
lift_pI)
  done

lemma [transfer_rule]:
  ((=) ===> (=) ===> rel_state (=)) update1 update1'
  unfolding update1_def update1'_def
  apply (intro rel_funI)
  apply simp
  apply (rule rel_state_ofI2)
  subgoal
    by (auto 4 3
        simp: mem_update_def inv_pair_lengthD1 execute_simps Let_def
        intro: success_intros intro!: success_bind_I
      )
  subgoal
    using pair.update_inv(1) unfolding update1_def .
  done

lemma [transfer_rule]:
  ((=) ===> (=) ===> rel_state (=)) update2 update2'
  unfolding update2_def update2'_def
  apply (intro rel_funI)
  apply simp
  apply (rule rel_state_ofI2)
  subgoal
    by (auto 4 3
        simp: mem_update_def inv_pair_lengthD2 execute_simps Let_def
        intro: success_intros intro!: success_bind_I
      )
  subgoal
    using pair.update_inv(2) unfolding update2_def .
  done

lemma [transfer_rule]:
  ((=) ===> rel_state (rel_option (=))) lookup1 mem_lookup1
  unfolding lookup1_def mem_lookup1_def
  apply (intro rel_funI)
  apply (simp add: option.rel_eq)
  apply (rule rel_state_ofI2)
  subgoal
    by (auto 4 3
        simp: mem_lookup_def inv_pair_lengthD1 execute_simps Let_def
        intro: success_intros intro!: success_bind_I
```

```
    )
  subgoal
    using pair.lookup_inv(1) unfolding lookup1_def .
  done

lemma rel_state_lookup:
  ((=) ===> rel_state (=)) pair.lookup_pair lookup_pair
  unfolding pair.lookup_pair_def lookup_pair_def
  unfolding
    mem_lookup1_def[symmetric] mem_lookup2_def[symmetric]
    get_k2_def[symmetric] get_k2'_def[symmetric]
    get_k1_def[symmetric] get_k1'_def[symmetric]
  by transfer_prover

lemma rel_state_update:
  ((=) ===> (=) ===> rel_state (=)) pair.update_pair update_pair
  unfolding pair.update_pair_def update_pair_def
  unfolding move12'_def[symmetric]
  unfolding
    update1'_def[symmetric] update2'_def[symmetric]
    get_k2_def[symmetric] get_k2'_def[symmetric]
    get_k1_def[symmetric] get_k1'_def[symmetric]
  by transfer_prover

interpretation mem: heap_mem_defs pair.inv_pair lookup_pair update_pair
.

lemma inv_pairD:
  inv_pair_weak heap if pair.inv_pair heap
  using that unfolding pair.inv_pair_def by (auto simp: Let_def)

lemma mem_rel_state_ofI:
  mem.rel_state (=) m' m if
  rel_state (=) m' m
  ⋀ heap. pair.inv_pair heap ⟹
    (case State_Monad.run_state m' heap of (_, heap) ⇒ inv_pair_weak
heap ⟶ pair.inv_pair heap)
proof -
  show ?thesis
    apply (rule mem.rel_state_intro)
    subgoal for heap v heap'
      by (auto elim: rel_state_elim[OF that(1)] dest!: inv_pairD)
    subgoal premises prems for heap v heap'
    proof -
```

from *prems that*(1) **have** *inv_pair_weak heap'*
  by (*fastforce elim*: *rel_state_elim dest*: *inv_pairD*)
with *prems* **show** *?thesis*
  by (*auto dest*: *that*(2))
qed
done
qed

lemma *mem_rel_state_ofI'*:
mem.rel_state (=) *m' m* **if**
rel_state (=) *m' m*
DP_CRelVS.lift_p pair.inv_pair m'
using *that* **by** (*auto elim*: *DP_CRelVS.lift_p_P intro*: *mem_rel_state_ofI*)

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*
by (*rule mem_correct_pair*[*OF keys*])

lemma *rel_state_lookup'*:
((=) ===> *mem.rel_state* (=)) *pair.lookup_pair lookup_pair*
apply (*intro rel_funI*)
apply *simp*
apply (*rule mem_rel_state_ofI'*)
using *rel_state_lookup* **apply** (*rule rel_funD*) **apply** (*rule refl*)
apply (*rule lookup_inv*)
done

lemma *rel_state_update'*:
((=) ===> (=) ===> *mem.rel_state* (=)) *pair.update_pair update_pair*
apply (*intro rel_funI*)
apply *simp*
apply (*rule mem_rel_state_ofI'*)
subgoal for *x y a b*
  using *rel_state_update* **by** (*blast dest*: *rel_funD*)
by (*rule update_inv*)

interpretation *heap_correct pair.inv_pair update_pair lookup_pair*
by (*rule mem.mem_correct_heap_correct*[*OF _ rel_state_lookup' rel_state_update'*])
standard

lemmas *heap_correct_pairI* = *heap_correct_axioms*

lemma *mem_rel_state_resultD*:
result_of m heap = *fst* (*run_state m′ heap*) **if** *mem.rel_state* (=) *m′ m*
pair.inv_pair heap
by (*metis* (*mono_tags*, *lifting*) *mem.rel_state_elim option.sel that*)

lemma *map_of_heap_eq*:
mem.map_of_heap heap = *pair.pair.map_of heap* **if** *pair.inv_pair heap*
unfolding *mem.map_of_heap_def pair.pair.map_of_def*
using *that* **by** (*simp add*: *mem_rel_state_resultD*[*OF rel_state_lookup′*[*THEN rel_funD*]])

context
fixes *k*1 *k*2 *heap heap′*
assumes *init*: *execute* (*init_state k*1 *k*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap′*)
begin

lemma *init_state_empty*1:
*pair.mem*1*.map_of heap′ k* = *None*
using *init*
unfolding *pair.mem*1*.map_of_def lookup*1*_def mem_lookup_def init_state_def*
by (*auto*
simp: *init_state_inner_nth init_state_inner_alloc*(3) *execute_simps*
Let_def
elim!: *execute_bind_success′*[*OF success_empty*])
(*metis*
Array.present_alloc Memory_Heap.length_mem_empty execute_new
execute_nth(1) *fst_conv*
length_def mem_empty_def nth_mem_empty option.sel present_alloc_get
snd_conv
)

lemma *init_state_empty*2:
*pair.mem*2*.map_of heap′ k* = *None*
using *init*
unfolding *pair.mem*2*.map_of_def lookup*2*_def mem_lookup_def init_state_def*
by (*auto*
simp: *execute_simps init_state_inner_nth init_state_inner_alloc*(4)
Let_def
elim!: *execute_bind_success′*[*OF success_empty*]
)
(*metis fst_conv nth_mem_empty option.sel snd_conv*)

lemma
  shows *init_state_k*1: *result_of* (!*k_ref*1) *heap′* = *k*1
    and *init_state_k*2: *result_of* (!*k_ref*2) *heap′* = *k*2
  using *init init_state_inner_alloc*
  by (*auto simp*: *execute_simps init_state_def elim*!: *execute_bind_success′*[*OF success_empty*])

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

lemma *init_state_inv′*:
  pair.inv_pair heap′
  unfolding *pair.inv_pair_def*
  apply (*auto simp*: *Let_def*)
  subgoal
    using *init_state_empty*1 **by** *simp*
  subgoal
    using *init_state_empty*2 **by** *simp*
  subgoal
    using *neq init* **by** (*simp add*: *get_k*1*_def get_k*2*_def init_state_k*1 *init_state_k*2)
  subgoal
    by (*rule init_state_inv*[*OF init*])
  done

lemma *init_state_empty*:
  pair.pair.map_of heap′ $\subseteq_m$ *Map.empty*
  using *neq* **by** (*intro pair.emptyI init_state_inv′ map_emptyI init_state_empty*1 *init_state_empty*2)

interpretation *heap_correct_empty pair.inv_pair update_pair lookup_pair heap′*
  apply (*rule heap_correct_empty.intro*)
  apply (*rule heap_correct_pairI*)
  apply *standard*
  subgoal
    by (*subst map_of_heap_eq*; *intro init_state_inv′ init_state_empty*)
  subgoal
    by (*rule init_state_inv′*)
  done

lemmas *heap_correct_empty_pairI* = *heap_correct_empty_axioms*

context
  fixes *dp* :: *'k* ⇒ *'v*
begin

interpretation *dp_consistency_heap_empty*
  pair.inv_pair update_pair lookup_pair dp heap'
  by *standard*

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_file ‹../transform/Transform_Misc.ML›
ML_file ‹../transform/Transform_Const.ML›
ML_file ‹../transform/Transform_Data.ML›
ML_file ‹../transform/Transform_Tactic.ML›
ML_file ‹../transform/Transform_Term.ML›
ML_file ‹../transform/Transform.ML›
ML_file ‹../transform/Transform_Parser.ML›

ML ‹
val _ =
  Outer_Syntax.local_theory @{command_keyword memoize_fun}
   (Transform_Parser.dp_fun_part1_parser >> Transform_DP.dp_fun_part1_cmd)

val _ =
  Outer_Syntax.local_theory @{command_keyword monadifies}
   (Transform_Parser.dp_fun_part2_parser >> Transform_DP.dp_fun_part2_cmd)

val _ =
 Outer_Syntax.local_theory_to_proof @{command_keyword memoize_correct}

    (Scan.succeed Transform_DP.dp_correct_cmd)
›

method_setup memoize_prover = ‹
  Scan.succeed (fn ctxt => SIMPLE_METHOD' (
    Transform_Data.get_last_cmd_info ctxt
    |> Transform_Tactic.solve_consistentDP_tac ctxt))
›

method_setup memoize_prover_init = ‹
  Scan.succeed (fn ctxt => SIMPLE_METHOD' (
    Transform_Data.get_last_cmd_info ctxt
    |> Transform_Tactic.prepare_consistentDP_tac ctxt))
›

method_setup memoize_prover_case_init = ‹
  Scan.succeed (fn ctxt => SIMPLE_METHOD' (
    Transform_Data.get_last_cmd_info ctxt
    |> Transform_Tactic.prepare_case_tac ctxt))
›

method_setup memoize_prover_match_step = ‹
```

```
  Scan.succeed (fn ctxt => SIMPLE_METHOD' (
    Transform_Data.get_last_cmd_info ctxt
    |> Transform_Tactic.step_tac ctxt))
›

method_setup memoize_unfold_defs = ‹
  Scan.option (Scan.lift (Args.parens Args.name) -- Args.term) >>
    (fn tm_opt => fn ctxt => SIMPLE_METHOD'
      (Transform_Data.get_or_last_cmd_info ctxt tm_opt
        |> Transform_Tactic.dp_unfold_defs_tac ctxt))
›

method_setup memoize_combinator_init = ‹
  Scan.option (Scan.lift (Args.parens Args.name) -- Args.term) >>
    (fn tm_opt => fn ctxt => SIMPLE_METHOD'
      (Transform_Data.get_or_last_cmd_info ctxt tm_opt
        |> Transform_Tactic.prepare_combinator_tac ctxt))
›

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* ())
unfolding *adm_wf_def* **by** *auto*

lemma *wellfounded*:
wf {(*nxt x*, *x*) | *x*. *cnt x*} (**is** *wf ?S*)
proof −
from *terminating* **have** *acyclic ?S*
by (*auto intro*: *acyclicI_order*[**where** *f* = *sizef*])
moreover have *finite ?S*
using [[*simproc add*: *finite_Collect*]] *terminating*(1) **by** *auto*
ultimately show *?thesis*
by − (*rule finite_acyclic_wf*)
qed

lemma *iter_state_unfold*:
iter_state f x = (*if cnt x then do* {*f x*; *iter_state f* (*nxt x*)} *else State_Monad.return* ())
unfolding *iter_state_def* **by** (*simp add*: *wfrec_fixpoint*[*OF wellfounded admissible*])

lemma *iterator_to_list_unfold*:
iterator_to_list x = (*if cnt x then x* # *iterator_to_list* (*nxt x*) *else* [])
unfolding *iterator_to_list_def* **by** (*simp add*: *adm_wf_def wfrec_fixpoint*[*OF wellfounded*])

lemma *iter_state_iterate_state*:
iter_state f x = *iterate_state f* (*iterator_to_list x*)
apply (*induction iterator_to_list x arbitrary*: *x*)

```
   apply (simp add: iterator_to_list_unfold split: if_split_asm)
   apply (simp add: iter_state_unfold)
  apply (subst (asm) (3) iterator_to_list_unfold)
  apply (simp split: if_split_asm)
  apply (auto simp: iterator_to_list_unfold iter_state_unfold)
  done

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 (induction xs)
  case Nil
  then show ?case
     by (simp; rule crel_vs_return_ext[unfolded Transfer.Rel_def]; simp;
fail)
next
  case (Cons x xs)
  have unit_expand: () = (λ a f. f a) () (λ _. ()) ..
  from Cons show ?case
    by simp
      (rule
      bind_transfer[unfolded rel_fun_def, rule_format, unfolded unit_expand]
        that[unfolded rel_fun_def, rule_format] HOL.refl
      )+
qed

lemma crel_vs_bind_ignore:
  crel_vs R a (do {d; b}) if crel_vs R a b crel_vs S c d
proof -
  have unit_expand: a = (λ a f. f a) () (λ _. a) ..
  show ?thesis
    by (subst unit_expand)
          (rule bind_transfer[unfolded rel_fun_def, rule_format, unfolded
unit_expand] that)+
qed
```

```
lemma crel_vs_iterate_and_compute:
  assumes ((=) ===>T R) g f
  shows crel_vs R (g x) (do {iterate_state f xs; f x})
  by (rule
        crel_vs_bind_ignore crel_vs_iterate_state HOL.refl
        assms[unfolded rel_fun_def, rule_format] assms
     )+

end

end

locale dp_consistency_iterator =
  dp_consistency lookup update + iterator cnt nxt sizef
  for lookup :: 'a ⇒ ('b, 'c option) state and update
    and cnt :: 'a ⇒ 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})
 unfolding iter_state_iterate_state using crel_vs_iterate_and_compute[OF
assms] .

lemma consistentDP_iter_and_compute:
  assumes consistentDP f
  shows crel_vs (=) (dp x) (do {iter_state f y; f x})
 using assms unfolding consistentDP_def by (rule crel_vs_iter_and_compute)

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
  using consistentDP_iter_and_compute[OF that, of x y]
  by (auto elim!: crel_vs_elim intro: P_empty cmem_empty)

lemma cmem_result:
  cmem (snd (run_state (do {iter_state f y; f x}) empty)) if consistentDP
f
  using consistentDP_iter_and_compute[OF that, of x y]
```

by (*auto elim*!: *crel_vs_elim intro*: *P_empty cmem_empty*)

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*
by (*meson*
dp_consistency_empty.axioms(1) *dp_consistency_iterator_def*
dp_consistency_iterator_empty_def that
)

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

lemma *table_iterator_up*:
iterator
(λ (x, y). $x \leq n \wedge y \leq m$)
(λ (x, y). *if* $y < m$ *then* (x, $y + 1$) *else* ($x + 1$, 0))
(λ (x, y). $x * (m + 1) + y$)
by *standard auto*

lemma *table_iterator_down*:
iterator
(λ (x, y). $x \leq n \wedge y \leq m \wedge x > 0$)
(λ (x, y). *if* $y > 0$ *then* (x, $y - 1$) *else* ($x - 1$, m))
(λ (x, y). $(n - x) * (m + 1) + (m - y)$)
using [[*simproc add*: *finite_Collect*]] **by** *standard* (*auto simp*: *Suc_diff_le*)

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*) | *x*. *cnt x*}
(λ *rec x*. *if cnt x then do* {*f x*; *rec* (*nxt x*)} *else return* ())

lemma (**in** *iterator*) *iter_heap_unfold*:
iter_heap f x = (*if cnt x then do* {*f x*; *iter_heap f* (*nxt x*)} *else return* ())
unfolding *iter_heap_def*
by (*simp add*: *wfrec_fixpoint*[*OF iterator.wellfounded*,*OF iterator.intro*,*OF terminating*] *adm_wf_def*)

locale *dp_consistency_iterator_heap* =
dp_consistency_heap P update lookup dp + *iterator cnt nxt sizef*
for *lookup* :: *'a* ⇒ (*'c option*) *Heap* **and** *update* **and** *P dp*
and *cnt* :: *'a* ⇒ *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*
using *wellfounded*
proof *induction*
case (*less x*)
have *unit_expand*: () = (λ *a f*. *f a*) () (λ _. ()) **..**
from *less* **show** *?case*
by (*subst iter_heap_unfold*)
(*auto intro*:
bind_transfer[*unfolded rel_fun_def*, *rule_format*, *unfolded unit_expand*]
crel_vs_return_ext[*unfolded Transfer.Rel_def*] *that*[*unfolded rel_fun_def*, *rule_format*]
)
qed

lemma *crel_vs_bind_ignore*:
crel_vs R a (*do* {*d*; *b*}) **if** *crel_vs R a b crel_vs S c d*
proof −
have *unit_expand*: *a* = (λ *a f*. *f a*) () (λ _. *a*) **..**
show *?thesis*
by (*subst unit_expand*)

```
        (rule bind_transfer[unfolded rel_fun_def, rule_format, unfolded
unit_expand] that)+
qed

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})
  by (rule
      crel_vs_bind_ignore crel_vs_iterate_state HOL.refl
      assms[unfolded rel_fun_def, rule_format] assms
    )+

lemma consistent_DP_iter_and_compute:
  assumes consistentDP f
  shows consistentDP (λ x. do {iter_heap f y; f x})
  apply (rule consistentDP_intro)
  using assms unfolding consistentDP_def Rel_def
  by (rule crel_vs_iter_and_compute)

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
unfolding *Heap_Monad_Ext.if*$_T$*_def*
unfolding *return_bind*
using *if_cong*[*OF assms*] **.**

lemma *return_app_return_cong*:
assumes $f\ x = g\ y$
shows $\langle f \rangle$. $\langle x \rangle$ = $\langle g \rangle$. $\langle y \rangle$
unfolding *Heap_Monad_Ext.return_app_return_meta assms* **..**

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 ((*R*1 ===>$_T$ *R*2) ===>$_T$ (*R*0 ===>$_T$ *R*1) ===>$_T$ (*R*0 ===>$_T$ *R*2)) *comp comp*$_T$
apply *memoize_combinator_init*
subgoal premises *IH* [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
done

memoize_fun *map*$_T$: *map* **monadifies** (*heap*) *list.map*
lemma (**in** *dp_consistency_heap*) *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*$_T$
apply *memoize_combinator_init*
apply (*erule list_all2_induct*)
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
done

memoize_fun *fold*$_T$: *fold* **monadifies** (*heap*) *fold.simps*

lemma (**in** *dp_consistency_heap*) $fold_T$*_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* $fold_T$
apply *memoize_combinator_init*
apply (*erule list_all*2*_induct*)
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
done

context includes *heap_monad_syntax* **begin**

thm *map_cong*
lemma *mapT_cong*:
assumes *xs* = *ys* ⋀*x*. *x*∈*set ys* ⟹ *f x* = *g x*
shows map_T . ⟨*f*⟩ . ⟨*xs*⟩ = map_T . ⟨*g*⟩ . ⟨*ys*⟩
unfolding map_T*_def*
unfolding *assms*(1)
using *assms*(2) **by** (*induction ys*) (*auto simp*: *Heap_Monad_Ext.return_app_return_meta*)

thm *fold_cong*
lemma *foldT_cong*:
assumes *xs* = *ys* ⋀*x*. *x*∈*set ys* ⟹ *f x* = *g x*
shows $fold_T$. ⟨*f*⟩ . ⟨*xs*⟩ = $fold_T$. ⟨*g*⟩ . ⟨*ys*⟩
unfolding $fold_T$*_def*
unfolding *assms*(1)
using *assms*(2) **by** (*induction ys*) (*auto simp*: *Heap_Monad_Ext.return_app_return_meta*)

lemma *abs_unit_cong*:

assumes *x* = *y*
shows (λ_::*unit*. *x*) = (λ_. *y*)
using *assms* **..**

lemma *arg_cong*4:
f a b c d = *f a′ b′ c′ d′* **if** *a* = *a′* *b* = *b′* *c* = *c′* *d* = *d′*
by (*simp add*: *that*)

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 ⟹ Rel (crel_vs R) x x_T ¬c ⟹ Rel (crel_vs R)
y y_T
  shows Rel (crel_vs R) (if (Wrap b) then x else y) (Heap_Monad_Ext.if_T
⟨c⟩ x_T y_T)
   using assms unfolding Heap_Monad_Ext.if_T_def bind_left_identity
Rel_def Wrap_def
  by (auto split: if_split)

lemma map_T_cong2:
  assumes
    is_equality R
    Rel R xs ys
    ⋀x. x∈set ys ⟹ 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 . ⟨f_T'⟩ . ⟨ys⟩)
  unfolding map_T_def
  unfolding Heap_Monad_Ext.return_app_return_meta
  unfolding assms(2)[unfolded Rel_def assms(1)[unfolded is_equality_def]]
  using assms(3)
  unfolding Rel_def Wrap_def App_def
  apply (induction ys)
  subgoal premises by (memoize_unfold_defs (heap) map) transfer_prover
  subgoal premises prems for a ys
  apply (memoize_unfold_defs (heap) map)
   apply (unfold Heap_Monad_Ext.return_app_return_meta Wrap_App_Wrap)
    supply [transfer_rule] =
      prems(2)[OF list.set_intros(1)]
      prems(1)[OF prems(2)[OF list.set_intros(2)], simplified]
    by transfer_prover
  done
```

```
lemma foldT_cong2:
  assumes
    is_equality R
    Rel R xs ys
    ⋀x. x∈set ys ⟹ Rel (crel_vs (S ===> crel_vs S)) (f x) (fT' x)
  shows
    Rel (crel_vs (S ===> crel_vs S)) (fold f xs) (foldT . ⟨fT'⟩ . ⟨ys⟩)
  unfolding foldT_def
  unfolding Heap_Monad_Ext.return_app_return_meta
  unfolding assms(2)[unfolded Rel_def assms(1)[unfolded is_equality_def]]
  using assms(3)
  unfolding Rel_def
  apply (induction ys)
  subgoal premises by (memoize_unfold_defs (heap) fold) transfer_prover
  subgoal premises prems for a ys
    apply (memoize_unfold_defs (heap) fold)
    apply (unfold Heap_Monad_Ext.return_app_return_meta Wrap_App_Wrap)
    supply [transfer_rule] =
      prems(2)[OF list.set_intros(1)]
      prems(1)[OF prems(2)[OF list.set_intros(2)], simplified]
    by transfer_prover
  done

lemma refl2:
  is_equality R ⟹ Rel R x x
  unfolding is_equality_def Rel_def by simp

lemma rel_fun2:
  assumes is_equality R0 ⋀x. Rel R1 (f x) (g x)
  shows Rel (rel_fun R0 R1) f g
  using assms unfolding is_equality_def Rel_def by auto

lemma crel_vs_return_app_return:
  assumes Rel R (f x) (g x)
  shows Rel R (App (Wrap f) (Wrap x)) (⟨g⟩ . ⟨x⟩)
  using assms unfolding Heap_Monad_Ext.return_app_return_meta Wrap_App_Wrap
.

thm option.case_cong[no_vars]
lemma option_case_cong':
Rel (=) option' option ⟹
(option = None ⟹ Rel R f1 g1) ⟹
(⋀x2. option = Some x2 ⟹ Rel R (f2 x2) (g2 x2)) ⟹
Rel R (case option' of None ⇒ f1 | Some x2 ⇒ f2 x2)
```

(*case option of None* ⇒ *g*1 | *Some x*2 ⇒ *g*2 *x*2)
  unfolding *Rel_def* **by** (*auto split*: *option.split*)

thm *prod.case_cong*[*no_vars*]
lemma *prod_case_cong'*: **fixes** *prod prod'* **shows**
Rel (=) *prod prod'* ⟹
(⋀*x*1 *x*2. *prod'* = (*x*1, *x*2) ⟹ *Rel R* (*f x*1 *x*2) (*g x*1 *x*2)) ⟹
Rel R (*case prod of* (*x*1, *x*2) ⇒ *f x*1 *x*2)
(*case prod' of* (*x*1, *x*2) ⇒ *g x*1 *x*2)
  unfolding *Rel_def* **by** (*auto split*: *prod.splits*)

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*_cong*2
  $fold_T$*_cong*2
  if_T*_cong*2

lemmas [*dp_match_rule*] =
  crel_vs_return
  crel_vs_fun_app
  *refl*2
  *rel_fun*2

end
end

2.7.1 More Heap

lemma *execute_heap_ofD*:
  heap_of c h = *h'* **if** *execute c h* = *Some* (*v*, *h'*)
  using *that* **by** *auto*

lemma *execute_result_ofD*:
  result_of c h = *v* **if** *execute c h* = *Some* (*v*, *h'*)
  using *that* **by** *auto*

locale *heap_correct_init_defs* =

fixes *P* :: *'m* ⇒ *heap* ⇒ *bool*
and *lookup* :: *'m* ⇒ *'k* ⇒ *'v option Heap*
and *update* :: *'m* ⇒ *'k* ⇒ *'v* ⇒ *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*: ⋀ *m*. *lift_p* (*P m*) (*lookup m k*)
assumes *update_inv*: ⋀ *m*. *lift_p* (*P m*) (*update m k v*)

locale *heap_correct_init* =
heap_correct_init_inv +
assumes *lookup_correct*:
⋀ *a*. *P a m* ⟹ *map_of_heap' a* (*snd* (*the* (*execute* (*lookup a k*) *m*)))
$\subseteq_m$ (*map_of_heap' a m*)
and *update_correct*:
⋀ *a*. *P a m* ⟹
map_of_heap' a (*snd* (*the* (*execute* (*update a k v*) *m*))) $\subseteq_m$ (*map_of_heap'*
a m)(*k* ↦ *v*)
begin

end

locale *dp_consistency_heap_init* = *heap_correct_init* _ *lookup* **for** *lookup*
:: *'m* ⇒ *'k* ⇒ *'v option Heap* +
fixes *dp* :: *'k* ⇒ *'v*
fixes *init* :: *'m Heap*
assumes *success*: *success init Heap.empty*
assumes *empty_correct*:
⋀ *empty heap*. *execute init Heap.empty* = *Some* (*empty*, *heap*) ⟹
map_of_heap' empty heap $\subseteq_m$ *Map.empty*
and *P_empty*: ⋀ *empty heap*. *execute init Heap.empty* = *Some* (*empty*,
heap) ⟹ *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*
apply *standard*
apply (*rule lookup_inv*[*of init_mem*])
apply (*rule update_inv*[*of init_mem*])
subgoal
unfolding *heap_mem_defs.map_of_heap_def*
by (*rule lookup_correct*[*of init_mem*, *unfolded map_of_heap′_def*])
subgoal
unfolding *heap_mem_defs.map_of_heap_def*
by (*rule update_correct*[*of init_mem*, *unfolded map_of_heap′_def*])
done

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*
apply *standard*
subgoal
apply (*rule successE*[*OF success*])
apply (*frule empty_correct*)
unfolding *heap_mem_defs.map_of_heap_def init_mem_def map_of_heap′_def*
by *simp*
subgoal
apply (*rule successE*[*OF success*])
apply (*frule P_empty*)
unfolding *init_mem_def*
by *simp*
done

lemma *memoized_empty*:
$dp\ x = result\_of\ (init \ggg= (\lambda mem.\ dp_T\ mem\ x))\ Heap.empty$
if $consistentDP\ (dp_T\ (result\_of\ init\ Heap.empty))$
by (*simp add*: *execute_bind_success consistent.memoized*[*OF that*(1)] *success*)

end

locale *dp_consistency_heap_init′* = *heap_correct_init_ lookup* **for** *lookup* :: *′m ⇒ ′k ⇒ ′v option Heap* +
fixes *dp* :: *′k ⇒ ′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*
apply *standard*
apply (*rule lookup_inv*[*of init_mem*])
apply (*rule update_inv*[*of init_mem*])
subgoal
unfolding *heap_mem_defs.map_of_heap_def*
by (*rule lookup_correct*[*of init_mem*, *unfolded map_of_heap′_def*])
subgoal
unfolding *heap_mem_defs.map_of_heap_def*
by (*rule update_correct*[*of init_mem*, *unfolded map_of_heap′_def*])
done

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*
apply *standard*
subgoal
apply (*rule successE*[*OF success*])
apply (*frule empty_correct*)
unfolding *heap_mem_defs.map_of_heap_def init_mem_def map_of_heap′_def*
by *simp*
subgoal
apply (*rule successE*[*OF success*])
apply (*frule P_empty*)
unfolding *init_mem_def*
by *simp*
done

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

by (*simp add*: *execute_bind_success consistent.memoized*[*OF that*(1)] *success*)

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*
using *success* **by** (*auto* 4 3 *intro*: *consistent successE*)

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*))
by (*simp add*: *execute_bind_success memoized*[*OF that*(1)] *success*)

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*
unfolding *mem_is_init*
using *success* **by** (*auto* 4 3 *intro*: *consistent successE*)

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*))
by (*simp add*: *execute_bind_success memoized*[*OF that*(1)] *success*)

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*
apply (*rule dp_consistency_new'.intro*)
subgoal
by (*rule mem_is_init*)

```
  subgoal
    by (rule success_empty)
  subgoal for empty heap
    using length_mem_empty by (metis fst_conv option.sel snd_conv)
  subgoal
    apply (frule execute_heap_ofD[symmetric])
    apply (frule execute_result_ofD[symmetric])
    apply simp
    apply (rule array_consistentI[OF injective HOL.refl])
    done
  done

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
  apply (rule dp_consistency_new.intro)
  subgoal
    by (rule success_empty)
  subgoal for empty heap
    using length_mem_empty by (metis fst_conv option.sel snd_conv)
  subgoal
    apply (frule execute_heap_ofD[symmetric])
    apply (frule execute_result_ofD[symmetric])
    apply simp
    apply (rule array_consistentI[OF injective HOL.refl])
    done
  done

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
  apply standard
  subgoal lookup_inv
     unfolding lift_p_def mem_lookup_def by (simp add: Let_def exe-
cute_simps)
  subgoal update_inv
      unfolding State_Heap.lift_p_def mem_update_def by (simp add:
Let_def execute_simps)
  subgoal for k heap
   unfolding heap_correct_init_defs.map_of_heap'_def map_le_def mem_lookup_def
    by (auto simp: execute_simps Let_def split: if_split_asm)
  subgoal for heap k
   unfolding heap_correct_init_defs.map_of_heap'_def map_le_def mem_lookup_def
mem_update_def
   apply (auto simp: execute_simps Let_def length_def split: if_split_asm)
    apply (subst (asm) nth_list_update_neq)
    using injective[unfolded injective_def] apply auto
    done
  subgoal
    by (rule success_empty)
  subgoal for empty' heap
   unfolding heap_correct_init_defs.map_of_heap'_def mem_lookup_def
    by (auto intro!: map_emptyI simp: Let_def ) (metis fst_conv option.sel
snd_conv nth_mem_empty)
  subgoal for empty' heap
   unfolding heap_correct_init_defs.map_of_heap'_def mem_lookup_def
map_le_def
    using length_mem_empty by (metis fst_conv option.sel snd_conv)
  done

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
  apply (rule dp_consistency_new'.intro)
  subgoal
    by (rule mem_is_init)
  subgoal
    by (rule succes_init_state)
  subgoal for empty heap
    unfolding inv_pair'_def
    apply safe
    apply (rule init_state_inv')
        apply (rule injective)
       apply (erule init_state_distinct)
      apply (rule keys_injective)
```

apply *assumption*
apply (*rule keys_neq*)
done
apply *safe*
unfolding *inv_pair′_def*
apply *simp*
apply (*rule consistent_empty_pairI*)
apply (*rule injective*)
apply (*erule init_state_distinct*)
apply (*rule keys_injective*)
apply *assumption*
apply (*rule keys_neq*)
done

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 key1 key2 m_ref1 m_ref2 k_ref1 k_ref2)
..

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*
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
  apply (rule dp_consistency_new.intro)
  subgoal
    by (rule succes_init_state)
  subgoal for empty heap
    unfolding inv_pair'_def
    apply safe
    apply (rule init_state_inv')
        apply (rule injective)
       apply (erule init_state_distinct)
      apply (rule keys_injective)
     apply assumption
    apply (rule keys_neq)
    done
  apply safe
  unfolding inv_pair'_def
  apply simp
  apply (rule consistent_empty_pairI)
      apply (rule injective)
     apply (erule init_state_distinct)
    apply (rule keys_injective)
   apply assumption
  apply (rule keys_neq)
  done

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
  unfolding *State_Monad_Ext.if*$_T$*_def*
  unfolding *bind_left_identity*
  using *if_cong*[*OF assms*] **.**

lemma *return_app_return_cong*:
  assumes $f\ x = g\ y$
  shows $\langle f \rangle$. $\langle x \rangle$ = $\langle g \rangle$. $\langle y \rangle$
  unfolding *State_Monad_Ext.return_app_return_meta assms* **..**

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 $((R1 ===>_T R2) ===>_T (R0 ===>_T R1) ===>_T (R0 ===>_T R2))$ *comp comp*$_T$
  apply *memoize_combinator_init*
  subgoal premises *IH* [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*

done

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
apply *memoize_combinator_init*
apply (*erule list_all*2*_induct*)
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
done

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* $fold_T$
apply *memoize_combinator_init*
apply (*erule list_all*2*_induct*)
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
subgoal premises [*transfer_rule*] **by** *memoize_unfold_defs transfer_prover*
done

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 . ⟨*f*⟩ . ⟨*xs*⟩ = map_T . ⟨*g*⟩ . ⟨*ys*⟩
unfolding map_T*_def*
unfolding *assms*(1)
using *assms*(2) **by** (*induction ys*) (*auto simp*: *State_Monad_Ext.return_app_return_meta*)

thm *fold_cong*
lemma *foldT_cong*:
assumes $xs = ys$ $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows $fold_T$. ⟨*f*⟩ . ⟨*xs*⟩ = $fold_T$. ⟨*g*⟩ . ⟨*ys*⟩
unfolding $fold_T$*_def*
unfolding *assms*(1)
using *assms*(2) **by** (*induction ys*) (*auto simp*: *State_Monad_Ext.return_app_return_meta*)

lemma *abs_unit_cong*:

assumes $x = y$
shows (λ_::*unit*. *x*) = (λ_. *y*)

using *assms* **..**

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* ⟹ *Rel* (*crel_vs R*) *x* x_T ¬*c* ⟹ *Rel* (*crel_vs R*) *y* y_T
shows *Rel* (*crel_vs R*) (*if* (*Wrap b*) *then x else y*) (*State_Monad_Ext.*if_T ⟨*c*⟩ x_T y_T)
using *assms* **unfolding** *State_Monad_Ext.*if_T*_def* *bind_left_identity* *Rel_def Wrap_def*
by (*auto split*: *if_split*)

lemma map_T*_cong2*:
assumes
is_equality R
Rel R xs ys
⋀*x*. *x*∈*set ys* ⟹ *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 . ⟨f_T′⟩ . ⟨*ys*⟩)
unfolding map_T*_def*
unfolding *State_Monad_Ext.return_app_return_meta*
unfolding *assms*(2)[*unfolded Rel_def assms*(1)[*unfolded is_equality_def*]]
using *assms*(3)
unfolding *Rel_def Wrap_def App_def*
apply (*induction ys*)
subgoal premises by (*memoize_unfold_defs* (*state*) *map*) *transfer_prover*
subgoal premises *prems* **for** *a ys*
apply (*memoize_unfold_defs* (*state*) *map*)
apply (*unfold State_Monad_Ext.return_app_return_meta Wrap_App_Wrap*)
supply [*transfer_rule*] =
prems(2)[*OF list.set_intros*(1)]

prems(1)[*OF prems*(2)[*OF list.set_intros*(2)], *simplified*]
by *transfer_prover*
done

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
$Rel\ (crel\_vs\ (S ===> crel\_vs\ S))\ (fold\ f\ xs)\ (fold_T\ .\ \langle f_T' \rangle\ .\ \langle ys \rangle)$
unfolding $fold_T$*_def*
unfolding *State_Monad_Ext.return_app_return_meta*
unfolding *assms*(2)[*unfolded Rel_def assms*(1)[*unfolded is_equality_def*]]
using *assms*(3)
unfolding *Rel_def*
apply (*induction ys*)
subgoal premises by (*memoize_unfold_defs* (*state*) *fold*) *transfer_prover*
subgoal premises *prems* **for** *a ys*
apply (*memoize_unfold_defs* (*state*) *fold*)
apply (*unfold State_Monad_Ext.return_app_return_meta Wrap_App_Wrap*)
supply [*transfer_rule*] =
prems(2)[*OF list.set_intros*(1)]
prems(1)[*OF prems*(2)[*OF list.set_intros*(2)], *simplified*]
by *transfer_prover*
done

lemma *refl2*:
$is\_equality\ R \Longrightarrow Rel\ R\ x\ x$
unfolding *is_equality_def Rel_def* **by** *simp*

lemma *rel_fun2*:
assumes $is\_equality\ R0\ \bigwedge x.\ Rel\ R1\ (f\ x)\ (g\ x)$
shows *Rel* (*rel_fun R0 R1*) *f g*
using *assms* **unfolding** *is_equality_def Rel_def* **by** *auto*

lemma *crel_vs_return_app_return*:
assumes *Rel R* (*f x*) (*g x*)
shows $Rel\ R\ (App\ (Wrap\ f)\ (Wrap\ x))\ (\langle g \rangle\ .\ \langle x \rangle)$
using *assms* **unfolding** *State_Monad_Ext.return_app_return_meta Wrap_App_Wrap*
.

thm *option.case_cong*[*no_vars*]
lemma *option_case_cong'*:

Rel (=) *option′ option* ⟹
(*option* = *None* ⟹ *Rel R f1 g1*) ⟹
(⋀*x2*. *option* = *Some x2* ⟹ *Rel R* (*f2 x2*) (*g2 x2*)) ⟹
Rel R (*case option′ of None* ⇒ *f1* | *Some x2* ⇒ *f2 x2*)
(*case option of None* ⇒ *g1* | *Some x2* ⇒ *g2 x2*)
  unfolding *Rel_def* **by** (*auto split*: *option.split*)

thm *prod.case_cong*[*no_vars*]
lemma *prod_case_cong′*: **fixes** *prod prod′* **shows**
Rel (=) *prod prod′* ⟹
(⋀*x1 x2*. *prod′* = (*x1*, *x2*) ⟹ *Rel R* (*f x1 x2*) (*g x1 x2*)) ⟹
Rel R (*case prod of* (*x1*, *x2*) ⇒ *f x1 x2*)
(*case prod′ of* (*x1*, *x2*) ⇒ *g x1 x2*)
  unfolding *Rel_def* **by** (*auto split*: *prod.splits*)

thm *nat.case_cong*[*no_vars*]
lemma *nat_case_cong′*: **fixes** *nat nat′* **shows**
Rel (=) *nat nat′* ⟹
(*nat′* = 0 ⟹ *Rel R f1 g1*) ⟹
(⋀*x2*. *nat′* = *Suc x2* ⟹ *Rel R* (*f2 x2*) (*g2 x2*)) ⟹
Rel R (*case nat of* 0 ⇒ *f1* | *Suc x2* ⇒ *f2 x2*) (*case nat′ of* 0 ⇒ *g1* | *Suc x2* ⇒ *g2 x2*)
  unfolding *Rel_def* **by** (*auto split*: *nat.splits*)

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* ⇒ *'a* **where**
  min_list (*x* # *xs*) = (*case xs of* [] ⇒ *x* | _ ⇒ *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*
  by (*induction xs arbitrary*: *a*; *auto*; *metis min.commute min.assoc*)

lemma *min_list_fold*:
  min_list (*x* # *xs*) = *fold min xs x* **for** *x* :: *'a* :: *linorder*
  by (*induction xs arbitrary*: *x*; *auto simp*: *fold_min_commute*[*symmetric*]; *metis min.commute*)

lemma *induct_list*012:
  ⟦*P* []; ⋀*x*. *P* [*x*]; ⋀*x y zs*. *P* (*y* # *zs*) ⟹ *P* (*x* # *y* # *zs*)⟧ ⟹ *P xs*
by *induction_schema* (*pat_completeness*, *lexicographic_order*)

lemma *min_list_Min*: *xs* ≠ [] ⟹ *min_list xs* = *Min* (*set xs*)

by (*induction xs rule*: *induct_list*012)(*auto*)

Extended Data Type **lemma** *Pinf_add_right*[*simp*]:
$\infty + x = \infty$
by (*cases x*; *simp*)

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*)
unfolding *trace_def* **by** *simp*

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)
unfolding *checkmem_trace_def checkmem_def trace_alt_def* **..**

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_file ‹*../util/Ground_Function.ML*›

ML ‹
```
fun ground_function_cmd ((termination, binding), thm_refs) lthy =
  let
    val def_thms = Attrib.eval_thms lthy thm_refs
  in
```

Ground_Function.mk_fun (*termination* <> *NONE*) *def_thms binding lthy*
end

val ground_function_parser =
Scan.option (*Parse.*$$$ (|−− *Parse.reserved prove_termination* −−| *Parse.*$$$))
−− (*Parse.binding* −−| *Parse.*$$$:) (∗ *scope*, *e.g.*, bf_T ∗)
−− *Parse.thms*1

val _ =
Outer_Syntax.local_theory @{*command_keyword ground_function*}
Define a new ground constant from an existing function definition
(*ground_function_parser* >> *ground_function_cmd*)
›

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 $i \leq n$
obtains $i < n$ | $i = n$
using *assms* **by** (*cases* $i = n$) *auto*

context *dp_consistency_iterator*
begin

lemma *crel_vs_iterate_state*:

crel_vs (=) () (*iter_state f x*) **if** ((=) ===>$_T$ *R*) *g f*
by (*metis crel_vs_iterate_state iter_state_iterate_state that*)

lemma *consistent_crel_vs_iterate_state*:
crel_vs (=) () (*iter_state f x*) **if** *consistentDP f*
using *consistentDP_def crel_vs_iterate_state that* **by** *simp*

end

instance *extended* :: (*countable*) *countable*
proof *standard*
obtain *to_nat* :: *'a* ⇒ *nat* **where** *inj to_nat*
by *auto*
let *?f* = λ *x*. *case x of Fin n* ⇒ *to_nat n* + 2 | *Pinf* ⇒ 0 | *Minf* ⇒ 1
from ‹*inj* _ › **have** *inj ?f*
by (*auto simp*: *inj_def split*: *extended.split*)
then show ∃ *to_nat* :: *'a extended* ⇒ *nat*. *inj to_nat*
by *auto*
qed

instance *extended* :: (*heap*) *heap* **..**

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

definition
Inf A = (
if A = {} ∨ *A* = {∞} *then* ∞
else if −∞ ∈ *A* ∨ ¬ *bdd_below* (*Fin* −‘ *A*) *then* −∞
else Fin (*Inf* (*Fin* −‘ *A*)))

definition
Sup A = (
if A = {} ∨ *A* = {−∞} *then* −∞
else if ∞ ∈ *A* ∨ ¬ *bdd_above* (*Fin* −‘ *A*) *then* ∞
else Fin (*Sup* (*Fin* −‘ *A*)))

instance
proof *standard*
have [*dest*]: *Inf* (*Fin* −‘ *A*) ≤ *x* **if** *Fin x* ∈ *A bdd_below* (*Fin* −‘ *A*) **for** *A* **and** *x* :: *'a*
using *that* **by** (*intro cInf_lower*) *auto*
have ∗: *False* **if** ¬ *z* ≤ *Inf* (*Fin* −‘ *A*) ⋀*x*. *x* ∈ *A* ⟹ *Fin z* ≤ *x Fin x* ∈ *A* **for** *A* **and** *x z* :: *'a*

using *cInf_greatest*[*of Fin* −' *A z*] *that vimage_eq* **by** *force*
show *Inf A* ≤ *x* **if** *x* ∈ *A* **for** *x* :: *'a extended* **and** *A*
using *that* **unfolding** *Inf_extended_def* **by** (*cases x*) *auto*
show *z* ≤ *Inf A* **if** ⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x* **for** *z* :: *'a extended* **and** *A*
using *that*
unfolding *Inf_extended_def*
apply (*clarsimp*; *safe*)
apply *force*
apply *force*
subgoal
by (*cases z*; *force simp*: *bdd_below_def*)
subgoal
by (*cases z*; *force simp*: *bdd_below_def*)
subgoal for *x y*
by (*cases z*; *cases y*) (*auto elim*: ∗)
subgoal for *x y*
by (*cases z*; *cases y*; *simp*; *metis* ∗ *less_eq_extended.elims*(2))
done
have [*dest*]: *x* ≤ *Sup* (*Fin* −' *A*) **if** *Fin x* ∈ *A bdd_above* (*Fin* −' *A*) **for** *A* **and** *x* :: *'a*
using *that* **by** (*intro cSup_upper*) *auto*
have ∗: *False* **if** ¬ *Sup* (*Fin* −' *A*) ≤ *z* ⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *Fin z Fin x* ∈ *A* **for** *A* **and** *x z* :: *'a*
using *cSup_least*[*of Fin* −' *A z*] *that vimage_eq* **by** *force*
show *x* ≤ *Sup A* **if** *x* ∈ *A* **for** *x* :: *'a extended* **and** *A*
using *that* **unfolding** *Sup_extended_def* **by** (*cases x*) *auto*
show *Sup A* ≤ *z* **if** ⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z* **for** *z* :: *'a extended* **and** *A*
using *that*
unfolding *Sup_extended_def*
apply (*clarsimp*; *safe*)
apply *force*
apply *force*
subgoal
by (*cases z*; *force*)
subgoal
by (*cases z*; *force*)
subgoal for *x y*
by (*cases z*; *cases y*) (*auto elim*: ∗)
subgoal for *x y*
by (*cases z*; *cases y*; *simp*; *metis* ∗ *extended.exhaust*)
done
show *Inf* {} = (*top*::*'a extended*)
unfolding *Inf_extended_def top_extended_def* **by** *simp*
show *Sup* {} = (*bot*::*'a extended*)

unfolding *Sup_extended_def bot_extended_def* **by** *simp*
qed

end

instance *extended* :: ({*conditionally_complete_lattice*,*linorder*}) *complete_linorder*
..

lemma *Minf_eq_zero*[*simp*]: $-\infty = 0 \longleftrightarrow$ *False* **and** *Pinf_eq_zero*[*simp*]: $\infty = 0 \longleftrightarrow$ *False*
unfolding *zero_extended_def* **by** *auto*

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 −
from *assms* **obtain** *x y* **where** $X \subseteq \{..y\}$ $x \in X$
by (*auto simp*: *bdd_above_def*)
then have ∗: *finite* $(X \cap \{x..y\})$ $X \cap \{x..y\} \neq \{\}$ **and** $x \leq y$
by (*auto simp*: *subset_eq*)
have $\exists!x{\in}X.\ (\forall y{\in}X.\ y \leq x)$
proof
{ fix *z* **assume** $z \in X$
have $z \leq$ *Max* $(X \cap \{x..y\})$
proof *cases*
assume $x \leq z$ **with** ‹$z \in X$› ‹$X \subseteq \{..y\}$› ∗(1) **show** *?thesis*
by (*auto intro*!: *Max_ge*)
next
assume $\neg\ x \leq z$
then have $z < x$ **by** *simp*
also have $x \leq$ *Max* $(X \cap \{x..y\})$
using ‹$x \in X$› ∗(1) ‹$x \leq y$› **by** (*intro Max_ge*) *auto*
finally show *?thesis* **by** *simp*
qed }
note *le* = *this*
with *Max_in*[*OF* ∗] **show** *ex*: *Max* $(X \cap \{x..y\}) \in X \wedge (\forall z{\in}X.\ z \leq$ *Max* $(X \cap \{x..y\}))$ **by** *auto*

fix *z* **assume** ∗: $z \in X \wedge (\forall y{\in}X.\ y \leq z)$
with *le* **have** $z \leq$ *Max* $(X \cap \{x..y\})$
by *auto*
moreover have *Max* $(X \cap \{x..y\}) \leq z$

using $*$ *ex* **by** *auto*
ultimately show $z = Max\ (X \cap \{x..y\})$
by *auto*
qed
then show $Sup\ X \in X \wedge (\forall y{\in}X.\ y \leq Sup\ X)$
unfolding *Sup_int_def* **by** (*rule theI'*)
qed

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

lemma *Inf_int_in*:
fixes *S* :: *int set*
assumes $S \neq \{\}$ *bdd_below S*
shows $Inf\ S \in S$
using *assms* **unfolding** *Inf_int_def* **by** (*smt Sup_int_in bdd_above_uminus image_iff image_is_empty*)

lemma *finite_setcompr_eq_image*: *finite* $\{f\ x\ |x.\ P\ x\} \longleftrightarrow$ *finite* $(f\ `\ \{x.\ P\ x\})$
by (*simp add*: *setcompr_eq_image*)

lemma *finite_lists_length_le*1: *finite* $\{xs.\ length\ xs \leq i \wedge set\ xs \subseteq \{0..(n{::}nat)\}\}$ **for** *i*
by (*auto intro*: *finite_subset*[*OF _ finite_lists_length_le*[*OF finite_atLeastAtMost*]])

lemma *finite_lists_length_le*2: *finite* $\{xs.\ length\ xs + 1 \leq i \wedge set\ xs \subseteq \{0..(n{::}nat)\}\}$ **for** *i*
by (*auto intro*: *finite_subset*[*OF _ finite_lists_length_le*1[*of i*]])

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*)
by (*simp add*: *State_Monad.bind_def State_Monad.get_def*)

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 −

from *assms* **have** $\neg$ *distinct xs*
by (*metis card_mono distinct_card not_le*)
then show *?thesis*
by (*metis append.assoc append_Cons not_distinct_conv_prefix split_list that*)
qed

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'
using *assms* **by** (*auto simp*: *Cons_eq_append_conv append_eq_Cons_conv append_eq_append_conv2*)

lemma *le_add_same_cancel1*:
$a + b \geq a \longleftrightarrow b \geq 0$ **if** $a < \infty$ $-\infty < a$ **for** a b :: *int extended*
using *that* **by** (*cases a*; *cases b*) (*auto simp add*: *zero_extended_def*)

lemma *add_gt_minfI*:
assumes $-\infty < a$ $-\infty < b$
shows $-\infty < a + b$
using *assms* **by** (*cases a*; *cases b*) *auto*

lemma *add_lt_infI*:
assumes $a < \infty$ $b < \infty$
shows $a + b < \infty$
using *assms* **by** (*cases a*; *cases b*) *auto*

lemma *sum_list_not_infI*:
sum_list xs $< \infty$ **if** $\forall$ $x \in$ *set xs*. $x < \infty$ **for** *xs* :: *int extended list*
using *that*
apply (*induction xs*)
apply (*simp add*: *zero_extended_def*)+
by (*smt less_extended_simps*(2) *plus_extended.elims*)

lemma *sum_list_not_minfI*:
sum_list xs $> -\infty$ **if** $\forall$ $x \in$ *set xs*. $x > -\infty$ **for** *xs* :: *int extended list*
using *that* **by** (*induction xs*) (*auto intro*: *add_gt_minfI simp*: *zero_extended_def*)

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* ⇒ *nat* ⇒ *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* ⇒ *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*]) | *xs*. *length xs* + 1 ≤ *i* ∧ *set xs* ⊆ {0..*n*}} ∪
      {*if t* = *v then* 0 *else* ∞}
)
)

lemma *weight_alt_def'*:
  weight (*s* # *xs*) + *w* = *snd* (*fold* (λ*j* (*i*, *x*). (*j*, *W i j* + *x*)) *xs* (*s*, *w*))
 by (*induction xs arbitrary*: *s w*; *simp*; *smt add.commute add.left_commute*)

lemma *weight_alt_def*:
  weight (*s* # *xs*) = *snd* (*fold* (λ*j* (*i*, *x*). (*j*, *W i j* + *x*)) *xs* (*s*, 0))
  by (*rule weight_alt_def'*[*of s xs* 0, *simplified*])

lemma *weight_append*:
  weight (*xs* @ *a* # *ys*) = *weight* (*xs* @ [*a*]) + *weight* (*a* # *ys*)
  by (*induction xs rule*: *weight.induct*; *simp add*: *add.assoc*)

lemma *OPT_0*:
  OPT 0 *v* = (*if t* = *v then* 0 *else* ∞)
  unfolding *OPT_def* **by** *simp*

3.2.3 Functional Correctness

lemma *OPT_cases*:

obtains (*path*) *xs* **where** *OPT i v* = *weight* (*v* # *xs* @ [*t*]) *length xs* + 1 ≤ *i set xs* ⊆ {0..*n*}
| (*sink*) *v* = *t OPT i v* = 0
| (*unreachable*) *v* ≠ *t OPT i v* = ∞
unfolding *OPT_def*
using *Min_in*[*of* {*weight* (*v* # *xs* @ [*t*]) |*xs*. *length xs* + 1 ≤ *i* ∧ *set xs* ⊆ {0..*n*}}
∪ {*if t* = *v then* 0 *else* ∞}]
by (*auto simp*: *finite_lists_length_le*2[*simplified*] *split*: *if_split_asm*)

lemma *OPT_Suc*:
OPT (*Suc i*) *v* = *min* (*OPT i v*) (*Min* {*OPT i w* + *W v w* | *w*. *w* ≤ *n*}) (**is** *?lhs* = *?rhs*)
if *t* ≤ *n*
proof −
have *OPT i w* + *W v w* ≥ *OPT* (*Suc i*) *v* **if** *w* ≤ *n* **for** *w*
using *OPT_cases*[*of i w*]
proof *cases*
case (*path xs*)
with ‹*w* ≤ *n*› **show** *?thesis*
by (*subst OPT_def*) (*auto intro*!: *Min_le exI*[**where** *x* = *w* # *xs*] *simp*: *add.commute*)
next
case *sink*
then show *?thesis*
by (*subst OPT_def*) (*auto intro*!: *Min_le exI*[**where** *x* = []])
next
case *unreachable*
then show *?thesis*
by *simp*
qed
then have *Min* {*OPT i w* + *W v w* |*w*. *w* ≤ *n*} ≥ *OPT* (*Suc i*) *v*
by (*auto intro*!: *Min.boundedI*)
moreover have *OPT i v* ≥ *OPT* (*Suc i*) *v*
unfolding *OPT_def* **by** (*rule Min_antimono*) *auto*
ultimately have *?lhs* ≤ *?rhs*
by *simp*

from *OPT_cases*[*of Suc i v*] **have** *?lhs* ≥ *?rhs*
proof *cases*
case (*path xs*)
note [*simp*] = *path*(1)
from *path* **consider**
(*zero*) *i* = 0 *length xs* = 0 | (*new*) *i* > 0 *length xs* = *i* | (*old*) *length xs*

```
< i
      by (cases length xs = i) auto
    then show ?thesis
    proof cases
      case zero
      with path have OPT (Suc i) v = W v t
        by simp
      also have W v t = OPT i t + W v t
        unfolding OPT_def using ‹i = 0› by auto
      also have ... ≥ Min {OPT i w + W v w |w. w ≤ n}
        using ‹t ≤ n› by (auto intro: Min_le)
      finally show ?thesis
        by (rule min.coboundedI2)
    next
      case new
      with ‹_ = i› obtain u ys where [simp]: xs = u # ys
        by (cases xs) auto
      from path have OPT i u ≤ weight (u # ys @ [t])
        unfolding OPT_def by (intro Min_le) auto
      from path have Min {OPT i w + W v w |w. w ≤ n} ≤ W v u + OPT
i u
        by (intro Min_le) (auto simp: add.commute)
      also from ‹OPT i u ≤ _› have ... ≤ OPT (Suc i) v
        by (simp add: add_left_mono)
      finally show ?thesis
        by (rule min.coboundedI2)
    next
      case old
      with path have OPT i v ≤ OPT (Suc i) v
        by (auto 4 3 intro: Min_le simp: OPT_def)
      then show ?thesis
        by (rule min.coboundedI1)
    qed
  next
    case unreachable
    then show ?thesis
      by simp
  next
    case sink
    then have OPT i v ≤ OPT (Suc i) v
      unfolding OPT_def by simp
    then show ?thesis
      by (rule min.coboundedI1)
  qed
```

with ‹*?lhs* ≤ *?rhs*› **show** *?thesis*
by (*rule order.antisym*)
qed

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* ≤ *n*›
proof (*induction i arbitrary*: *j*)
case 0
then show *?case*
by (*simp add*: *OPT_0*)
next
case (*Suc i*)
have ∗:
{*bf i w* + *W j w* |*w*. *w* ≤ *n*} = *set* (*map* (λ*w*. *W j w* + *bf i w*) [0..<*Suc n*])
by (*fastforce simp*: *add.commute image_def*)
from *Suc* ‹*t* ≤ *n*› **show** *?case*
by (*simp add*: *OPT_Suc del*: *upt_Suc*, *subst Min.set_eq_fold*[*symmetric*], *auto simp*: ∗)
qed

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
by *memoize_prover*
print_theorems

lemmas [*code*] = bf_m.*memoized_correct*

interpretation *iterator*
  λ (*x*, *y*). *x* ≤ *n* ∧ *y* ≤ *n*
  λ (*x*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0)
  λ (*x*, *y*). *x* ∗ (*n* + 1) + *y*
  by (*rule table_iterator_up*)

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* ← *State_Monad.get*; *State_Monad.set* (*Mapping.update k v*
m)}
  λ (*x*, *y*). *x* ≤ *n* ∧ *y* ≤ *n*
  λ (*x*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0)
  λ (*x*, *y*). *x* ∗ (*n* + 1) + *y*
  Mapping.empty **..**

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

lemma *iter_bf_unfold*[*code*]:
  iter_bf = (λ (*i*, *j*).
    (*if i* ≤ *n* ∧ *j* ≤ *n*
     then do {
            bf_m' *i j*;
            iter_bf (*if j* < *n then* (*i*, *j* + 1) *else* (*i* + 1, 0))
          }
     else State_Monad.return ()))
 unfolding *iter_bf_def* **by** (*rule ext*) (*safe*, *clarsimp simp*: *iter_state_unfold*)

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' (λ*i*. bf_m' *n i*) [0..<*n*+1];
      ys ← *State_Main*.map_T' (λ*i*. 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*$_T$. ⟨λ*i*. *bf*$_m$′ (*n* + 1) *i*⟩ . ⟨[0..<*n*+1]⟩)⟩)
. (*State_Main.map*$_T$. ⟨λ*i*. *bf*$_m$′ *n i*⟩ . ⟨[0..<*n*+1]⟩)
}
unfolding
State_Monad_Ext.fun_app_lifted_def bellman_ford_def State_Main.map$_T$*_def bind_left_identity*
.

end

3.2.5 Imperative Memoization

context
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*
begin

lemma [*intro*]:
dp_consistency_heap_array_pair′ (*n* + 1) *fst snd id* 1 0 *mem*
by (*standard*; *simp add*: *mem_is_init injective_def*)

interpretation *iterator*
λ (*x*, *y*). *x* ≤ *n* ∧ *y* ≤ *n*
λ (*x*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0)
λ (*x*, *y*). *x* ∗ (*n* + 1) + *y*
by (*rule table_iterator_up*)

lemma [*intro*]:
dp_consistency_heap_array_pair_iterator (*n* + 1) *fst snd id* 1 0 *mem*
(λ (*x*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0))
(λ (*x*, *y*). *x* ∗ (*n* + 1) + *y*)

(λ (x, y). x ≤ n ∧ y ≤ n)
by (*standard*; *simp add*: *mem_is_init injective_def*)

memoize_fun bf_h: *bf*
with_memory (**default_proof**) *dp_consistency_heap_array_pair_iterator*
where *size* = n + 1
and *key*1 = *fst* :: *nat* × *nat* ⇒ *nat* **and** *key*2 = *snd* :: *nat* × *nat* ⇒ *nat*
and *k*1 = 1 :: *nat* **and** *k*2 = 0 :: *nat*
and *to_index* = *id* :: *nat* ⇒ *nat*
and *mem* = *mem*
and *cnt* = λ (x, y). x ≤ n ∧ y ≤ n
and *nxt* = λ (x :: *nat*, y). *if* y < n *then* (x, y + 1) *else* (x + 1, 0)
and *sizef* = λ (x, y). x ∗ (n + 1) + y
monadifies (*heap*) *bf.simps*

memoize_correct
by *memoize_prover*

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* ⊆ {0..n}} ∪
{*if* t = v *then* 0 *else* ∞}
)
)

definition
is_path xs ≡ *weight* (*xs* @ [t]) < ∞

definition
has_negative_cycle ≡
∃ *xs a ys*. *set* (a # *xs* @ *ys*) ⊆ {0..n} ∧ *weight* (a # *xs* @ [a]) < 0 ∧ *is_path* (a # *ys*)

definition
reaches a ≡ ∃ *xs*. *is_path* (a # *xs*) ∧ a ≤ n ∧ *set xs* ⊆ {0..n}

lemma *fold_sum_aux'*:
assumes ∀ *u* ∈ *set* (*a* # *xs*). ∀ *v* ∈ *set* (*xs* @ [*b*]). *f v* + *W u v* ≥ *f u*
shows *sum_list* (*map f* (*a* # *xs*)) ≤ *sum_list* (*map f* (*xs* @ [*b*])) + *weight* (*a* # *xs* @ [*b*])
using *assms*
by (*induction xs arbitrary*: *a*; *simp*)
(*smt ab_semigroup_add_class.add_ac*(1) *add.left_commute add_mono*)

lemma *fold_sum_aux*:
assumes ∀ *u* ∈ *set* (*a* # *xs*). ∀ *v* ∈ *set* (*a* # *xs*). *f v* + *W u v* ≥ *f u*
shows *sum_list* (*map f* (*a* # *xs* @ [*a*])) ≤ *sum_list* (*map f* (*a* # *xs* @ [*a*])) + *weight* (*a* # *xs* @ [*a*])
using *fold_sum_aux'*[*of a xs a f*] *assms*
by *auto* (*metis* (*no_types*, *opaque_lifting*) *add.assoc add.commute add_left_mono*)

context
begin

private definition *is_path2 xs* ≡ *weight xs* < ∞

private lemma *is_path2_remove_cycle*:
assumes *is_path2* (*as* @ *a* # *bs* @ *a* # *cs*)
shows *is_path2* (*as* @ *a* # *cs*)
proof −
have *weight* (*as* @ *a* # *bs* @ *a* # *cs*) =
weight (*as* @ [*a*]) + *weight* (*a* # *bs* @ [*a*]) + *weight* (*a* # *cs*)
by (*metis Bellman_Ford.weight_append append_Cons append_assoc*)
with *assms* **have** *weight* (*as* @ [*a*]) < ∞ *weight* (*a* # *cs*) < ∞
unfolding *is_path2_def*
by (*simp*, *metis Pinf_add_right antisym less_extended_simps*(4) *not_less add.commute*)+
then show *?thesis*
unfolding *is_path2_def* **by** (*subst weight_append*) (*rule add_lt_infI*)
qed

private lemma *is_path_eq*:
is_path xs ⟷ *is_path2* (*xs* @ [*t*])
unfolding *is_path_def is_path2_def* **..**

lemma *is_path_remove_cycle*:
assumes *is_path* (*as* @ *a* # *bs* @ *a* # *cs*)
shows *is_path* (*as* @ *a* # *cs*)
using *assms* **unfolding** *is_path_eq* **by** (*simp add*: *is_path2_remove_cycle*)

lemma *is_path_remove_cycle2*:
  assumes *is_path* (*as* @ *t* # *cs*)
  shows *is_path as*
 using *assms* **unfolding** *is_path_eq* **by** (*simp add*: *is_path2_remove_cycle*)

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*
proof (*cases length xs* < *n*)
  case *True*
  with *assms* **show** *?thesis*
    by (*auto intro*: *that*)
next
  case *False*
  then have *length xs* $\geq$ *n*
    by *auto*
  with *assms*(1,3) **show** *?thesis*
  proof (*induction length xs arbitrary*: *xs rule*: *less_induct*)
    case *less*
    then have *length* (*i* # *xs* @ [*t*]) > *card* ({0..*n*})
      by *auto*
    moreover from *less.prems* ‹$i \leq n$› ‹$t \leq n$› **have** *set* (*i* # *xs* @ [*t*]) $\subseteq$ {0..*n*}
      by *auto*
    ultimately obtain *a as bs cs* **where** ∗: *i* # *xs* @ [*t*] = *as* @ *a* # *bs* @ *a* # *cs*
      by (*elim list_pidgeonhole*) *auto*
    obtain *ys* **where** *ys*: *is_path* (*i* # *ys*) *length ys* < *length xs set* (*i* # *ys*) $\subseteq$ {0..*n*}
      apply *atomize_elim*
      using ∗
    proof (*cases rule*: *path_eq_cycleE*)
      case *Nil_Nil*
      with ‹$t \neq i$› **show** ∃ *ys*. *is_path* (*i* # *ys*) ∧ *length ys* < *length xs* ∧ *set* (*i* # *ys*) $\subseteq$ {0..*n*}
        by *auto*
    next
      case (*Nil_Cons cs'*)
      then show ∃ *ys*. *is_path* (*i* # *ys*) ∧ *length ys* < *length xs* ∧ *set* (*i* # *ys*) $\subseteq$ {0..*n*}
      using ‹*set* (*i* # *xs* @ [*t*]) $\subseteq$ {0..*n*}› ‹*is_path* (*i* # *xs*)› *is_path_remove_cycle*[*of*

[]]
by − (*rule exI*[**where** $x = cs'$], *simp*)
next
case (*Cons_Nil* as')
then show $\exists$ *ys*. *is_path* (i # *ys*) ∧ *length ys* < *length xs* ∧ *set* (i #
ys) ⊆ {0..n}
using ‹*set* (i # *xs* @ [t]) ⊆ {0..n}› ‹*is_path* (i # *xs*)›
by − (*rule exI*[**where** $x = as'$], *auto intro*: *is_path_remove_cycle*2)
next
case (*Cons_Cons* as' cs')
then show $\exists$ *ys*. *is_path* (i # *ys*) ∧ *length ys* < *length xs* ∧ *set* (i #
ys) ⊆ {0..n}
using ‹*set* (i # *xs* @ [t]) ⊆ {0..n}› ‹*is_path* (i # *xs*)› *is_path_remove_cycle*[*of*
i # as']
by − (*rule exI*[**where** $x = as'$ @ a # cs'], *auto*)
qed
then show *?thesis*
by (*cases* $n \le$ *length ys*) (*auto intro*: *that less*)
qed
qed

lemma *reaches_non_inf_path*:
assumes *reaches* i $i \le n$ $t \le n$
shows *OPT* n $i < \infty$
proof (*cases* $t = i$)
case *True*
with ‹$i \le n$› ‹$t \le n$› **have** *OPT* n $i \le 0$
unfolding *OPT_def*
by (*auto intro*: *Min_le simp*: *finite_lists_length_le*2[*simplified*])
then show *?thesis*
using *less_linear* **by** (*fastforce simp*: *zero_extended_def*)
next
case *False*
from *assms*(1) **obtain** *xs* **where** *is_path* (i # *xs*) $i \le n$ *set xs* ⊆ {0..n}
unfolding *reaches_def* **by** *safe*
then obtain *xs* **where** *xs*: *is_path* (i # *xs*) $i \le n$ *set xs* ⊆ {0..n} *length*
xs < n
using ‹$t \ne i$› ‹$t \le n$› **by** (*auto intro*: *is_path_shorten*)
then have *weight* (i # *xs* @ [t]) < ∞
unfolding *is_path_def* **by** *auto*
with *xs*(2−) **show** *?thesis*
unfolding *OPT_def*
by (*elim order.strict_trans*1[*rotated*])
(*auto simp*: *setcompr_eq_image finite_lists_length_le*2[*simplified*])

qed

lemma *OPT_sink_le_0*:
OPT i t ≤ 0
unfolding *OPT_def* **by** (*auto simp*: *finite_lists_length_le2*[*simplified*])

lemma *is_path_appendD*:
assumes *is_path* (*as* @ *a* # *bs*)
shows *is_path* (*a* # *bs*)
using *assms weight_append*[*of as a bs* @ [*t*]] **unfolding** *is_path_def*
by *simp* (*metis Pinf_add_right add.commute less_extended_simps*(4) *not_less_iff_gr_or_eq*)

lemma *has_negative_cycleI*:
assumes *set* (*a* # *xs* @ *ys*) ⊆ {0..*n*} *weight* (*a* # *xs* @ [*a*]) < 0 *is_path* (*a* # *ys*)
shows *has_negative_cycle*
using *assms* **unfolding** *has_negative_cycle_def* **by** *auto*

lemma *OPT_cases2*:
obtains (*path*) *xs* **where**
v ≠ *t* *OPT i v* ≠ ∞ *OPT i v* = *weight* (*v* # *xs* @ [*t*]) *length xs* + 1 ≤ *i* *set xs* ⊆ {0..*n*}
| (*unreachable*) *v* ≠ *t* *OPT i v* = ∞
| (*sink*) *v* = *t* *OPT i v* ≤ 0
unfolding *OPT_def*
using *Min_in*[*of* {*weight* (*v* # *xs* @ [*t*]) |*xs*. *length xs* + 1 ≤ *i* ∧ *set xs* ⊆ {0..*n*}}
∪ {*if t* = *v then* 0 *else* ∞}]
by (*cases v* = *t*; *force simp*: *finite_lists_length_le2*[*simplified*] *split*: *if_split_asm*)

lemma *shortest_le_OPT*:
assumes *v* ≤ *n*
shows *shortest v* ≤ *OPT i v*
unfolding *OPT_def shortest_def*
apply (*subst Min_Inf*)
apply (*simp add*: *setcompr_eq_image finite_lists_length_le2*[*simplified*]; *fail*)+
apply (*rule Inf_superset_mono*)
apply *auto*
done

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* $\neq []$
using *that* **using** *W_wellformed* ‹$t \leq n$›
by (*induction xs rule*: *induct_list*012) (*auto intro*: *add_gt_minfI simp*: *zero_extended_def*)

lemma *OPT_not_minfI*:
OPT n i $> -\infty$ **if** $i \leq n$
proof −
have *OPT n i* $\in$
{*weight* (*i* # *xs* @ [*t*]) |*xs*. *length xs* + 1 $\leq$ *n* $\wedge$ *set xs* $\subseteq \{0..n\}$} $\cup$ {*if t* = *i then* 0 *else* ∞}
unfolding *OPT_def*
by (*rule Min_in*) (*auto simp*: *setcompr_eq_image finite_lists_length_le*2[*simplified*])
with *that* ‹$t \leq n$› **show** *?thesis*
by (*auto* 4 3 *intro*!: *weight_not_minfI simp*: *zero_extended_def*)
qed

theorem *detects_cycle*:
assumes *has_negative_cycle*
shows $\exists\, i \leq n.$ *OPT* $(n + 1)$ *i* $<$ *OPT n i*
proof −
from *assms* ‹$t \leq n$› **obtain** *xs a ys* **where** *cycle*:
$a \leq n$ *set xs* $\subseteq \{0..n\}$ *set ys* $\subseteq \{0..n\}$
weight (*a* # *xs* @ [*a*]) < 0 *is_path* (*a* # *ys*)
unfolding *has_negative_cycle_def* **by** *clarsimp*
then have *reaches a*
unfolding *reaches_def* **by** *auto*
have *reaches*: *reaches x* **if** $x \in$ *set xs* **for** *x*
proof −
from *that* **obtain** *as bs* **where** *xs* = *as* @ *x* # *bs*
by *atomize_elim* (*rule split_list*)
with *cycle* **have** *weight* (*x* # *bs* @ [*a*]) $< \infty$
using *weight_append*[*of a* # *as x bs* @ [*a*]]
by *simp* (*metis Pinf_add_right Pinf_le add.commute less_eq_extended.simps*(2) *not_less*)

moreover from ‹*reaches a*› **obtain** *cs* **where** *local.weight* (*a* # *cs* @ [*t*]) $< \infty$ *set cs* $\subseteq \{0..n\}$
unfolding *reaches_def is_path_def* **by** *auto*

ultimately show *?thesis*
unfolding *reaches_def is_path_def*
using ‹$a \le n$› *weight_append*[*of x # bs a cs* @ [*t*]] *cycle*(2) ‹*xs* = _›
by − (*rule exI*[**where** *x* = *bs* @ [*a*] @ *cs*], *auto intro*: *add_lt_infI*)
qed
let *?S* = *sum_list* (*map* (*OPT n*) (*a* # *xs* @ [*a*]))
obtain *u v* **where** $u \le n$ $v \le n$ *OPT n v* + *W u v* < *OPT n u*
proof (*atomize_elim*, *rule ccontr*)
assume ∄ *u v*. $u \le n \wedge v \le n \wedge$ *OPT n v* + *W u v* < *OPT n u*
then have *?S* ≤ *?S* + *weight* (*a* # *xs* @ [*a*])
using *cycle*(1−3) **by** (*subst fold_sum_aux*; *fastforce simp*: *subset_eq*)
moreover have *?S* > −∞
using *cycle*(1−4) **by** (*intro sum_list_not_minfI*, *auto intro*!: *OPT_not_minfI*)
moreover have *?S* < ∞
using *reaches* ‹$t \le n$› *cycle*(1,2)
by (*intro sum_list_not_infI*) (*auto intro*: *reaches_non_inf_path* ‹*reaches a*› *simp*: *subset_eq*)
ultimately have *weight* (*a* # *xs* @ [*a*]) ≥ 0
by (*simp add*: *le_add_same_cancel*1)
with ‹*weight* _ < 0› **show** *False*
by *simp*
qed
then show *?thesis*
by −
(*rule exI*[**where** *x* = *u*],
auto 4 4 *intro*: *Min.coboundedI min.strict_coboundedI*2 *elim*: *order.strict_trans*1[*rotated*]
simp: *OPT_Suc*[*OF* ‹$t \le n$›])
qed

corollary *bf_detects_cycle*:
assumes *has_negative_cycle*
shows ∃ $i \le n$. *bf* (*n* + 1) *i* < *bf n i*
using *detects_cycle*[*OF assms*] **unfolding** *bf_correct*[*OF* ‹$t \le n$›] **.**

lemma *shortest_cases*:
assumes $v \le n$
obtains (*path*) *xs* **where** *shortest v* = *weight* (*v* # *xs* @ [*t*]) *set xs* ⊆ {0..*n*}
| (*sink*) *v* = *t* *shortest v* = 0
| (*unreachable*) *v* ≠ *t* *shortest v* = ∞
| (*negative_cycle*) *shortest v* = −∞ ∀ *x*. ∃ *xs*. *set xs* ⊆ {0..*n*} ∧ *weight* (*v* # *xs* @ [*t*]) < *Fin x*
proof −

```
  let ?S = {weight (v # xs @ [t]) | xs. set xs ⊆ {0..n}} ∪ {if t = v then 0
else ∞}
  have ?S ≠ {}
    by auto
  have Minf_lowest: False if  −∞ < a −∞ = a for a :: int extended
    using that by auto
  show ?thesis
  proof (cases shortest v)
    case (Fin x)
    then have −∞ ∉ ?S bdd_below (Fin −‘ ?S) ?S ≠ {∞} x = Inf (Fin
−‘ ?S)
     unfolding shortest_def Inf_extended_def by (auto split: if_split_asm)
    from this(1−3) have x ∈ Fin −‘ ?S
      unfolding ‹x = _›
      by (intro Inf_int_in, auto simp: zero_extended_def)
      (smt empty_iff extended.exhaust insertI2 mem_Collect_eq vimage_eq)
    with ‹shortest v = _› show ?thesis
      unfolding vimage_eq by (auto split: if_split_asm intro: that)
  next
    case Pinf
    with ‹?S ≠ {}› have t ≠ v
     unfolding shortest_def Inf_extended_def by (auto split: if_split_asm)
    with ‹_ = ∞› show ?thesis
      by (auto intro: that)
  next
    case Minf
    then have ?S ≠ {} ?S ≠ {∞} −∞ ∈ ?S ∨ ¬ bdd_below (Fin −‘ ?S)
     unfolding shortest_def Inf_extended_def by (auto split: if_split_asm)
    from this(3) have ∀ x. ∃ xs. set xs ⊆ {0..n} ∧ weight (v # xs @ [t]) <
Fin x
    proof
      assume −∞ ∈ ?S
      with weight_not_minfI have False
      using ‹v ≤ n› ‹t ≤ n› by (auto split: if_split_asm elim: Minf_lowest[rotated])
      then show ?thesis ..
    next
      assume ¬ bdd_below (Fin −‘ ?S)
      show ?thesis
      proof
        fix x :: int
        let ?m = min x (−1)
        from ‹¬ bdd_below _› obtain m where Fin m ∈ ?S m < ?m
          unfolding bdd_below_def by − (simp, drule spec[of _ ?m], force)
        then show ∃ xs. set xs ⊆ {0..n} ∧ weight (v # xs @ [t]) < Fin x
```

```
          by (auto split: if_split_asm simp: zero_extended_def) (metis
less_extended_simps(1))+
      qed
    qed
    with ‹shortest v = _› show ?thesis
      by (auto intro: that)
  qed
qed

lemma simple_paths:
  assumes ¬ has_negative_cycle weight (v # xs @ [t]) < ∞ set xs ⊆ {0..n}
v ≤ n
  obtains ys where
    weight (v # ys @ [t]) ≤ weight (v # xs @ [t]) set ys ⊆ {0..n} length ys
< n | v = t
  using assms(2−)
proof (atomize_elim, induction length xs arbitrary: xs rule: less_induct)
  case (less ys)
  note ys = less.prems(1,2)
  note IH = less.hyps
  have path: is_path (v # ys)
    using is_path_def not_less_iff_gr_or_eq ys(1) by fastforce
  show ?case
  proof (cases length ys ≥ n)
    case True
    with ys ‹v ≤ n› ‹t ≤ n› obtain a as bs cs where v # ys @ [t] = as @
a # bs @ a # cs
      by − (rule list_pidgeonhole[of v # ys @ [t] {0..n}], auto)
    then show ?thesis
    proof (cases rule: path_eq_cycleE)
      case Nil_Nil
      then show ?thesis
        by simp
    next
      case (Nil_Cons cs′)
      then have ∗: weight (v # ys @ [t]) = weight (a # bs @ [a]) + weight
(a # cs′ @ [t])
        by (simp add: weight_append[of a # bs a cs′ @ [t], simplified])
      show ?thesis
      proof (cases weight (a # bs @ [a]) < 0)
        case True
        with Nil_Cons ‹set ys ⊆ _› path show ?thesis
          using assms(1) by (force intro: has_negative_cycleI[of a bs ys])
      next
```

case *False*
then have *weight* (*a* # *bs* @ [*a*]) ≥ 0
by *auto*
with ∗ *ys* **have** *weight* (*a* # *cs′* @ [*t*]) ≤ *weight* (*v* # *ys* @ [*t*])
using *add_mono not_le* **by** *fastforce*
with *Nil_Cons* ‹*length ys* ≥ *n*› *ys* **show** *?thesis*
using *IH*[*of cs′*] **by** *simp* (*meson le_less_trans order_trans*)
qed
next
case (*Cons_Nil as′*)
with *ys* **have** ∗: *weight* (*v* # *ys* @ [*t*]) = *weight* (*v* # *as′* @ [*t*]) + *weight* (*a* # *bs* @ [*a*])
using *weight_append*[*of v* # *as′ t bs* @ [*t*]] **by** *simp*
show *?thesis*
proof (*cases weight* (*a* # *bs* @ [*a*]) < 0)
case *True*
with *Cons_Nil* ‹*set ys* ⊆ _› *path assms*(1) **show** *?thesis*
using *is_path_appendD*[*of v* # *as′*] **by** (*force intro*: *has_negative_cycleI*[*of a bs bs*])
next
case *False*
then have *weight* (*a* # *bs* @ [*a*]) ≥ 0
by *auto*
with ∗ *ys*(1) **have** *weight* (*v* # *as′* @ [*t*]) ≤ *weight* (*v* # *ys* @ [*t*])
using *add_left_mono* **by** *fastforce*
with *Cons_Nil* ‹*length ys* ≥ *n*› ‹*v* ≤ *n*› *ys* **show** *?thesis*
using *IH*[*of as′*] **by** *simp* (*meson le_less_trans order_trans*)
qed
next
case (*Cons_Cons as′ cs′*)
with *ys* **have** ∗:
weight (*v* # *ys* @ [*t*]) = *weight* (*v* # *as′* @ *a* # *cs′* @ [*t*]) + *weight* (*a* # *bs* @ [*a*])
using
weight_append[*of v* # *as′ a bs* @ *a* # *cs′* @ [*t*]]
weight_append[*of a* # *bs a cs′* @ [*t*]]
weight_append[*of v* # *as′ a cs′* @ [*t*]]
by (*simp add*: *algebra_simps*)
show *?thesis*
proof (*cases weight* (*a* # *bs* @ [*a*]) < 0)
case *True*
with *Cons_Cons* ‹*set ys* ⊆ _› *path assms*(1) **show** *?thesis*
using *is_path_appendD*[*of v* # *as′*]
by (*force intro*: *has_negative_cycleI*[*of a bs bs* @ *a* # *cs′*])

```
      next
        case False
        then have weight (a # bs @ [a]) ≥ 0
          by auto
        with * ys have weight (v # as' @ a # cs' @ [t]) ≤ weight (v # ys
@ [t])
          using add_left_mono by fastforce
        with Cons_Cons ‹v ≤ n› ys show ?thesis
          using is_path_remove_cycle2 IH[of as' @ a # cs']
          by simp (meson le_less_trans order_trans)
      qed
    qed
  next
    case False
    with ‹set ys ⊆ _› show ?thesis
      by auto
  qed
qed

theorem shorter_than_OPT_n_has_negative_cycle:
  assumes shortest v < OPT n v v ≤ n
  shows has_negative_cycle
proof −
  from assms obtain ys where ys:
    weight (v # ys @ [t]) < OPT n v set ys ⊆ {0..n}
     apply (cases rule: OPT_cases2[of v n]; cases rule: shortest_cases[OF
‹v ≤ n›]; simp)
      apply (metis uminus_extended.cases)
    using less_extended_simps(2) less_trans apply blast
   apply (metis less_eq_extended.elims(2) less_extended_def zero_extended_def)
    done
  show ?thesis
  proof (cases v = t)
    case True
    with ys ‹t ≤ n› show ?thesis
     using OPT_sink_le_0[of n] unfolding has_negative_cycle_def is_path_def
      using less_extended_def by force
  next
    case False
    show ?thesis
    proof (rule ccontr)
      assume ¬ has_negative_cycle
      with False False ys ‹v ≤ n› obtain xs where
        weight (v # xs @ [t]) ≤ weight (v # ys @ [t]) set xs ⊆ {0..n} length
```

```
xs < n
        using less_extended_def by (fastforce elim!: simple_paths[of v ys])
      then have OPT n v ≤ weight (v # xs @ [t])
        unfolding OPT_def by (intro Min_le) auto
      with ‹_ ≤ weight (v # ys @ [t])› ‹weight (v # ys @ [t]) < OPT n v›
show False
        by simp
    qed
  qed
qed

corollary detects_cycle_has_negative_cycle:
  assumes OPT (n + 1) v < OPT n v v ≤ n
  shows has_negative_cycle
  using assms shortest_le_OPT[of v n + 1] shorter_than_OPT_n_has_negative_cycle[of
v] by auto

corollary bellman_ford_detects_cycle:
  has_negative_cycle ⟷ (∃ v ≤ n. OPT (n + 1) v < OPT n v)
  using detects_cycle_has_negative_cycle detects_cycle by blast

corollary bellman_ford_shortest_paths:
  assumes ¬ has_negative_cycle
  shows ∀ v ≤ n. bf n v = shortest v
proof −
  have OPT n v ≤ shortest v if v ≤ n for v
    using that assms shorter_than_OPT_n_has_negative_cycle[of v] by
force
  then show ?thesis
    unfolding bf_correct[OF ‹t ≤ n›, symmetric]
    by (safe, rule order.antisym) (auto elim: shortest_le_OPT)
qed

lemma OPT_mono:
  OPT m v ≤ OPT n v if ‹v ≤ n› ‹n ≤ m›
  using that unfolding OPT_def by (intro Min_antimono) auto

corollary bf_fix:
  assumes ¬ has_negative_cycle m ≥ n
  shows ∀ v ≤ n. bf m v = bf n v
proof (intro allI impI)
  fix v assume v ≤ n
  from ‹v ≤ n› ‹n ≤ m› have shortest v ≤ OPT m v
    by (simp add: shortest_le_OPT)
```

moreover from ‹$v \le n$› ‹$n \le m$› **have** *OPT m v* ≤ *OPT n v*
by (*rule OPT_mono*)
moreover from ‹$v \le n$› *assms* **have** *OPT n v* ≤ *shortest v*
using *shorter_than_OPT_n_has_negative_cycle*[*of v*] **by** *force*
ultimately show *bf m v* = *bf n v*
unfolding *bf_correct*[*OF* ‹$t \le n$›, *symmetric*] **by** *simp*
qed

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*
let *?l* = *if has_negative_cycle then None else Some* (*map shortest* [0..<*n* + 1])
let *?r* = (λ*xs*. (λ*ys*. (*if xs* = *ys then Some xs else None*))
$ (*map* $ ⟪*bf* (*n* + 1)⟫ $ ⟪[0..<*n* + 1]⟫)) $ (*map* $ ⟪*bf n*⟫ $ ⟪[0..<*n* + 1]⟫)
note *crel_*bf_m*'* = bf_m.*crel*[*unfolded* bf_m.*consistentDP_def*, *THEN rel_funD*, *of* (*m*, *x*) (*m*, *y*) **for** *m x y*, *unfolded prod.case*]
have *?l* = *?r*
supply [*simp del*] = *bf_simps*
supply [*simp add*] = *bf_fix*[*rule_format*, *symmetric*] *bellman_ford_shortest_paths*[*rule_format*, *symmetric*]
unfolding *Wrap_def App_def* **using** *bf_detects_cycle* **by** (*fastforce elim*: *nat_le_cases*)
— Slightly transform the goal, then apply parametric reasoning like usual.
show *?thesis*
— Roughly
unfolding *bellman_ford_alt_def* ‹*?l* = *?r*› — Obtain parametric form.
apply (*rule* bf_m.*crel_vs_bind_ignore*[*rotated*]) — Drop bind.
apply (*rule bottom_up.consistent_crel_vs_iterate_state*[*OF* bf_m.*crel*, *folded iter_bf_def*])
apply (*subst Transfer.Rel_def*[*symmetric*]) — Setup typical goal for automated reasoner.
— We need to reason manually because we are not in the context where bf_m was defined.
— This is roughly what *memoize_prover_match_step*/*Transform_Tactic.step_tac* does.
apply (*tactic* ‹*Transform_Tactic.solve_relator_tac* ***context*** 1›
| *rule HOL.refl*
| *rule* bf_m.*dp_match_rule*
| *rule* bf_m.*crel_vs_return_ext*

```
      | (subst Rel_def, rule crel_bf_m')
      | tactic ‹Transform_Tactic.transfer_raw_tac context 1›)+
  done
qed

theorem bellman_ford_correct:
  fst (run_state bellman_ford Mapping.empty) =
  (if has_negative_cycle then None else Some (map shortest [0..<n+1]))
 using bf_m.cmem_empty bellman_ford_correct'[unfolded bf_m.crel_vs_def,
rule_format, of Mapping.empty]
  unfolding bf_m.crel_vs_def by auto

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 −
  have bf_h'_impl n w t mem i j = bf_h' n w t mem i j for i j
    by (induction rule: bf_h'.induct[OF mem_is_init];
        simp add: bf_h'.simps[OF mem_is_init]; solve_cong simp
      )
  then show ?thesis
    by auto
qed

definition
  iter_bf_heap n w t mem = iterator_defs.iter_heap
      (λ(x, y). x ≤ n ∧ y ≤ n)
      (λ(x, y). if y < n then (x, y + 1) else (x + 1, 0))
      (λ(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* ≤ *n* ∧ *j* ≤ *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 ()))
 unfolding *iter_bf_heap_def* **by** (*rule ext*) (*safe*, *simp add*: *iter_heap_unfold*)

definition
  bf_impl n w t i j = *do* {
    mem ← (*init_state* (*n* + 1) (1::*nat*) (0::*nat*) ::
       (*nat ref* × *nat ref* × *int extended option array ref* × *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*
  using *memoized_empty*[*OF HOL.refl*, *of n w t* (*i*, *j*)]
  by (*simp add*:
      execute_bind_success[*OF succes_init_state*] *bf_impl_def* bf_h'*_impl_def iter_bf_heap_def*
)

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* ∞ (*Fin o snd*) (*List.find* (λ *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

ML ‹*Test.test_bf* ()›

lemma *bottom_up_alt*[*code*]:
bf n W t i j =
  fst (*run_state*
   (*iter_bf n W t* (0, 0) ⤜ (λ_. bf_m*' n W t i j*))
   Mapping.empty)
using *bf_bottom_up* **by** *auto*

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*) ⤜ (λ_. 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
by *eval*+

— Regular test cases.
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]
by *eval*+

— Test detection of negative cycles.
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) *Map-*

```
ping.empty) = None
  by eval+

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 ⇒ 'v
begin

interpretation idx: bounded_index bound .

sublocale dp_consistency_heap
  where P=λheap. 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
  apply (rule dp_consistency_heap.intro)
  apply (rule mem_heap_correct)
  apply (rule idx.checked_idx_injective)
  done

context
  fixes empty
  assumes empty: map_of_heap empty ⊆m Map.empty
      and len: Array.length empty mem = idx.size
begin

interpretation consistent: dp_consistency_heap_empty
  where P=λheap. 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
  by (standard; rule len empty)

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*
by (*auto intro*!: *map_emptyI simp*:
that length_mem_empty Let_def nth_mem_empty mem_lookup_def heap_mem_defs.map_of_heap_def
)

lemma *memoized_empty*:
dp x = *result_of* ((*mem_empty idx.size* :: 'v option array Heap) ≫= (λ*mem*. dp_T *mem x*)) *Heap.empty*
if *consistentDP* (dp_T *mem*) *mem* = *result_of* (*mem_empty idx.size*) *Heap.empty*
apply (*subst execute_bind_success*)
defer
apply (*subst memoizedI*[*OF* _ _ *that*(1)])
using *mem_empty_empty*[*OF that*(2)] **by** (*auto simp*: *that*(2) *length_mem_empty*)

lemma *init_success*:
success ((*mem_empty idx.size* :: 'v option array Heap) ≫= (λ*mem*. dp_T *mem x*)) *Heap.empty*
if *consistentDP* (dp_T *mem*) *mem* = *result_of* (*mem_empty idx.size*) *Heap.empty*
apply (*rule success_bind_I*[*OF success_empty*])
apply (*frule execute_result_ofD*)
apply (*drule execute_heap_ofD*)
using *mem_empty_empty that* **by** (*auto simp*: *length_mem_empty intro*: *successI*)

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* ⇒ *nat*
begin

context
  fixes *v* :: *nat* ⇒ *nat*
begin

fun *knapsack* :: *nat* ⇒ *nat* ⇒ *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* {∑ *i* ∈ *S*. *v i* | *S*. *S* ⊆ {1..*n*} ∧ (∑ *i* ∈ *S*. *w i*) ≤ *W*}

lemma *OPT_0*:
  OPT 0 *W* = 0
  unfolding *OPT_def* **by** *simp*

3.3.2 Functional Correctness

lemma *Max_add_left*:
  (*x* :: *nat*) + *Max S* = *Max* (((+) *x*) ‘ *S*) (**is** *?A* = *?B*) **if** *finite S S* ≠ {}
proof −
  have *?A* ≤ *?B*
    using *that* **by** (*force intro*: *Min.boundedI*)
  moreover have *?B* ≤ *?A*
    using *that* **by** (*force intro*: *Min.boundedI*)
  ultimately show *?thesis*
    by *simp*
qed

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 −
have *OPT_in*: *OPT n W* ∈ {∑ *i* ∈ *S*. *v i* | *S*. *S* ⊆ {1..*n*} ∧ (∑ *i* ∈ *S*. *w i*) ≤ *W*} **for** *n W*
unfolding *OPT_def* **by** − (*rule Max_in*; *force*)
from *OPT_in*[*of Suc i W*] **obtain** *S* **where** *S*:
S ⊆ {1..*Suc i*} *sum w S* ≤ *W* **and** [*simp*]: *OPT* (*Suc i*) *W* = *sum v S*
by *auto*

have *OPT i W* ≤ *OPT* (*Suc i*) *W*
unfolding *OPT_def* **by** (*force intro*: *Max_mono*)
moreover have *v* (*Suc i*) + *OPT i* (*W* − *w* (*Suc i*)) ≤ *OPT* (*Suc i*) *W* **if** *w* (*Suc i*) ≤ *W*
proof −
have ∗:
v (*Suc i*) + *sum v S* = *sum v* (*S* ∪ {*Suc i*}) ∧ (*S* ∪ {*Suc i*}) ⊆ {1..*Suc i*}
∧ *sum w* (*S* ∪ {*Suc i*}) ≤ *W* **if** *S* ⊆ {1..*i*} *sum w S* ≤ *W* − *w* (*Suc i*) **for** *S*
using *that* ‹*w* (*Suc i*) ≤ *W*›
by (*subst sum.insert_if* | *auto intro*: *finite_subset*[*OF _ finite_atLeastAtMost*])+
show *?thesis*
unfolding *OPT_def*
by (*subst Max_add_left*;
fastforce intro: *Max_mono finite_subset*[*OF _ finite_atLeastAtMost*] *dest*: ∗
)
qed
ultimately have *?lhs* ≥ *?rhs*
by *auto*

from *S* **have** ∗: *sum v S* ≤ *OPT i W* **if** *Suc i* ∉ *S*
using *that* **unfolding** *OPT_def* **by** (*auto simp*: *atLeastAtMostSuc_conv intro*!: *Max_ge*)

have *sum v S* ≤ *OPT i W* **if** *W* < *w* (*Suc i*)
proof (*rule* ∗, *rule ccontr*, *simp*)
assume *Suc i* ∈ *S*
then have *sum w S* ≥ *w* (*Suc i*)
using *S*(1) **by** (*subst sum.remove*) (*auto intro*: *finite_subset*[*OF _ finite_atLeastAtMost*])
with ‹*W* < _› ‹_ ≤ *W*› **show** *False*
by *simp*

qed
moreover have
OPT (*Suc i*) *W* ≤ *max*(*v* (*Suc i*) + *OPT i* (*W* − *w* (*Suc i*))) (*OPT i* *W*) **if** *w* (*Suc i*) ≤ *W*
proof (*cases Suc i* ∈ *S*)
case *True*
then have [*simp*]:
sum v S = *v* (*Suc i*) + *sum v* (*S* − {*Suc i*}) *sum w S* = *w* (*Suc i*) + *sum w* (*S* − {*Suc i*})
using *S*(1) **by** (*auto intro*: *finite_subset*[*OF _ finite_atLeastAtMost*] *sum.remove*)
have *OPT i* (*W* − *w* (*Suc i*)) ≥ *sum v* (*S* − {*Suc i*})
unfolding *OPT_def* **using** *S* **by** (*fastforce intro*!: *Max_ge*)
then show *?thesis*
by *simp*
next
case *False*
then show *?thesis*
by (*auto dest*: ∗)
qed
ultimately have *?lhs* ≤ *?rhs*
by *auto*
with ‹*?lhs* ≥ *?rhs*› **show** *?thesis*
by *simp*
qed

theorem *knapsack_correct*:
OPT n W = *knapsack n W*
by (*induction n arbitrary*: *W*; *auto simp*: *OPT_0 OPT_Suc*)

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
by *memoize_prover*
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
by *memoize_prover*

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*
unfolding $knapsack_h$*_def* **using** *memoized_empty'*[*of _ n W*] **by** (*simp add*: *index_size_defs*)

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*
  by (*induction n arbitrary*: *W*; *simp*)

lemma *su_correct*:
  Max {∑ *i* ∈ *S*. *w i* | *S*. *S* ⊆ {1..*n*} ∧ (∑ *i* ∈ *S*. *w i*) ≤ *W*} = *su n W*
  unfolding *su_knapsack knapsack_correct*[*symmetric*] *OPT_def* **..**

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
  by *memoize_prover*
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 ‹*Test.knapsack_test* ()›

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* ∩ {*l. length l* = *n*}) **if** *finite A*
  using *that*
proof (*induction n*)

case 0 **thus** *?case*
by *auto*
next
case (*Suc n*)
have *lists A* ∩ { *l. length l* = *Suc n* } = (λ(*a*,*l*). *a*#*l*) ‘ (*A* × (*lists A* ∩ {*l. length l* = *n*}))
by (*auto simp*: *length_Suc_conv*)
moreover from *Suc* **have** *finite* ...
by *auto*
ultimately show *?case*
by *simp*
qed

lemma *disjE1*:
$A \vee B \Longrightarrow (A \Longrightarrow P) \Longrightarrow (\neg\, A \Longrightarrow B \Longrightarrow P) \Longrightarrow P$
by *metis*

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 ⟹ *valid* (*B* # *xs*) |
valid xs ⟹ *n* ≥ *m* ⟹ *valid* (*replicate n R* @ *xs*)

Definition of the fill-count function

definition *F n* = *card* {*l. length l* = *n* ∧ *valid l*}

3.4.3 Combinatorial Identities

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

inductive *valid′* **where**
valid′ [] |
n ≥ *m* ⟹ *valid′* (*replicate n R*) |
valid′ xs ⟹ *valid′* (*B* # *xs*) |

valid′ xs ⟹ *n* ≥ *m* ⟹ *valid′* (*replicate n R* @ *B* # *xs*)

lemma *valid_valid′*:
valid l ⟹ *valid′ l*
by (*induction rule*: *valid.induct*)
(*auto* 4 4 *intro*: *valid′.intros elim*: *valid′.cases*
simp: *replicate_add*[*symmetric*] *append_assoc*[*symmetric*]
)

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

lemma *valid′_valid*:
valid′ l ⟹ *valid l*
by (*induction rule*: *valid′.induct*) (*auto intro*: *valid.intros valid_red*)

lemma *valid_eq_valid′*:
valid′ l = *valid l*
using *valid_valid′ valid′_valid* **by** *metis*

Additional Facts on Replicate

lemma *replicate_iff*:
(∀ *i*<*length l*. *l* ! *i* = *R*) ⟷ (∃ *n*. *l* = *replicate n R*)
by *auto* (*metis* (*full_types*) *in_set_conv_nth replicate_eqI*)

lemma *replicate_iff2*:
(∀ *i*<*n*. *l* ! *i* = *R*) ⟷ (∃ *l′*. *l* = *replicate n R* @ *l′*) **if** *n* < *length l*
using *that* **by** (*auto simp*: *list_eq_iff_nth_eq nth_append intro*: *exI*[**where** *x* = *drop n l*])

lemma *replicate_Cons_eq*:
replicate n x = *y* # *ys* ⟷ (∃ *n′*. *n* = *Suc n′* ∧ *x* = *y* ∧ *replicate n′ x* = *ys*)
by (*cases n*) *auto*

Main Case Analysis on @*term valid*

lemma *valid_split*:
valid l ⟷
l = [] ∨
(*l*!0 = *B* ∧ *valid* (*tl l*)) ∨
length l ≥ *m* ∧ (∀ *i* < *length l*. *l* ! *i* = *R*) ∨
(∃ *j* < *length l*. *j* ≥ *m* ∧ (∀ *i* < *j*. *l* ! *i* = *R*) ∧ *l* ! *j* = *B* ∧ *valid* (*drop* (*j* + 1) *l*))
unfolding *valid_eq_valid′*[*symmetric*]
apply *standard*

subgoal
by (*erule valid'.cases*) (*auto simp*: *nth_append nth_Cons split*: *nat.splits*)
subgoal
apply (*auto intro*: *valid'.intros simp*: *replicate_iff elim*!: *disjE*1)
apply (*fastforce intro*: *valid'.intros simp*: *neq_Nil_conv*)
apply (*subst* (*asm*) *replicate_iff*2; *fastforce intro*: *valid'.intros simp*: *neq_Nil_conv nth_append*)+
done
done

Base cases

lemma *valid_line_just_B*:
valid (*replicate n B*)
by (*induction n*) (*auto intro*: *valid.intros*)

lemma *F_base_0_aux*:
$\{l.\ l = [] \wedge valid\ l\} = \{[]\}$
by (*auto intro*: *valid.intros*)

lemma *F_base_0*: $F\ 0 = 1$
by (*auto simp*: *F_base_0_aux F_def*)

lemma *F_base_aux*: $\{l.\ length\ l{=}n \wedge valid\ l\} = \{replicate\ n\ B\}$ **if** $n > 0$ $n < m$
using *that*
proof (*induction n*)
case 0
then show *?case*
by *simp*
next
case (*Suc n*)
show *?case*
proof (*cases* $n = 0$)
case *True*
with *Suc.prems* **show** *?thesis*
by (*auto intro*: *valid.intros elim*: *valid.cases*)
next
case *False*
with *Suc.prems* **show** *?thesis*
apply *safe*
using *Suc.IH*
apply −
apply (*erule valid.cases*)
apply (*auto intro*: *valid.intros elim*: *valid.cases*)

```
    done
  qed
qed

lemma F_base_1:
  F n = 1 if n > 0 n < m
  using that unfolding F_def by (simp add: F_base_aux)

lemma valid_m_Rs [simp]:
  valid (replicate m R)
  using valid_red[of m, simplified] by simp

lemma F_base_aux_2: {l. length l=m ∧ valid l} = {replicate m R, repli-
cate m B}
  apply (auto simp: valid_line_just_B)
  apply (erule Counting_Tiles.valid.cases)
    apply auto
  subgoal for xs
    using F_base_aux[of length xs] by (cases xs = []) auto
  done

lemma F_base_2:
  F m = 2 if 0 < m
  using that unfolding F_def by (simp add: F_base_aux_2)
```

The recursion case

```
lemma finite_valid_length:
  finite {l. length l = n ∧ valid l} (is finite ?S)
proof -
  have ?S ⊆ lists {R, B} ∩ {l. length l = n}
    by (auto intro: color.exhaust)
  moreover have finite ...
    by (auto intro: lists_of_len_fin1)
  ultimately show ?thesis
    by (rule finite_subset)
qed

lemma valid_line_aux:
  {l. length l = n ∧ valid l} ≠ {} (is ?S ≠ {})
  using valid_line_just_B[of n] by force

lemma replicate_unequal_aux:
  replicate x R @ B # l ≠ replicate y R @ B # l' (is ?l ≠ ?r) if ‹x < y›
for l l'
```

proof −
  have *?l* ! *x* = *B* *?r* ! *x* = *R*
    using *that* **by** (*auto simp*: *nth_append*)
  then show *?thesis*
    by *auto*
qed

lemma *valid_prepend_B_iff*:
  valid (*B* # *xs*) ⟷ *valid xs* **if** *m* > 0
  using *that*
  by (*auto* 4 3 *intro*: *valid.intros elim*: *valid.cases simp*: *Cons_replicate_eq* *Cons_eq_append_conv*)

lemma *F_rec*: *F n* = *F* (*n*−1) + 1 + (∑ *i*=*m*..<*n*. *F* (*n*−*i*−1)) **if** ‹*n*>*m*› *m* > 0
proof −
  have {*l*. *length l* = *n* ∧ *valid l*}
        = {*l*. *length l* = *n* ∧ *valid* (*tl l*) ∧ *l*!0=*B*}
        ∪ {*l*. *length l* = *n* ∧
            (∃ *i*. *i* < *n* ∧ *i* ≥ *m* ∧ (∀ *k* < *i*. *l*!*k* = *R*) ∧ *l*!*i* = *B* ∧ *valid* (*drop* (*i* + 1) *l*))}
        ∪ {*l*. *length l* = *n* ∧ (∀ *i*<*n*. *l*!*i*=*R*)}
         (**is** *?A* = *?B* ∪ *?D* ∪ *?C*)
    using ‹*n* > *m*› **by** (*subst valid_split*) *auto*

  let *?B*1 = ((#) *B*) ‘ {*l*. *length l* = *n* − *Suc* 0 ∧ *valid l*}
  from ‹*n* > *m*› **have** *?B* = *?B*1
    apply *safe*
    subgoal for *l*
      by (*cases l*) (*auto simp*: *valid_prepend_B_iff*)
    by *auto*
  have 1: *card ?B*1 = *F* (*n*−1)
    unfolding *F_def* **by** (*auto intro*: *card_image*)

  have *?C* = {*replicate n R*}
    by (*auto simp*: *nth_equalityI*)
  have 2: *card* {*replicate n R*} = 1
    by *auto*

  let *?D*1=(⋃ *i* ∈ {*m*..<*n*}. (λ *l*. *replicate i R* @ *B* # *l*)‘ {*l*. *length l* = *n* − *i* − 1 ∧ *valid l*})
  have *?D* =
      (⋃ *i* ∈ {*m*..<*n*}. {*l*. *length l* = *n* ∧ (∀ *k* < *i*. *l*!*k* = *R*) ∧ *l*!*i* = *B* ∧ *valid* (*drop* (*i* + 1) *l*)})

by *auto*
have {*l. length l* = *n* ∧ (∀ *k* < *i. l*!*k* = *R*) ∧ *l*!*i* = *B* ∧ *valid* (*drop* (*i* + 1) *l*)}
= (λ *l. replicate i R* @ *B* # *l*)' {*l. length l* = *n* − *i* − 1 ∧ *valid* *l*}
if *i* < *n* **for** *i*
apply *safe*
subgoal for *l*
apply (*rule image_eqI*[**where** *x* = *drop* (*i* + 1) *l*])
apply (*rule nth_equalityI*)
using *that*
apply (*simp_all split*: *nat.split add*: *nth_Cons nth_append*)
using *add_diff_inverse_nat* **apply** *fastforce*
done
using *that* **by** (*simp add*: *nth_append*; *fail*)+

then have *D_eq*: *?D* = *?D*1
unfolding ‹*?D* = _› **by** *auto*

have *inj*: *inj_on* (λ*l. replicate x R* @ *B* # *l*) {*l. length l* = *n* − *Suc x* ∧ *valid l*} **for** *x*
unfolding *inj_on_def* **by** *auto*

have ∗:
(λ*l. replicate x R* @ *B* # *l*) ' {*l. length l* = *n* − *Suc x* ∧ *valid l*} ∩
(λ*l. replicate y R* @ *B* # *l*) ' {*l. length l* = *n* − *Suc y* ∧ *valid l*} = {}
if *m* ≤ *x x* < *y y* < *n* **for** *x y*
using *that replicate_unequal_aux*[*OF* ‹*x* < *y*›] **by** *auto*

have 3: *card ?D*1 = ($\sum$ *i*=*m*..<*n*. *F* (*n*−*i*−1))
proof (*subst card_Union_disjoint*, *goal_cases*)
case 1
show *?case*
unfolding *pairwise_def disjnt_def*
proof (*clarsimp*, *goal_cases*)
case *prems*: (1 *x y*)
from *prems* **show** *?case*
apply −
apply (*rule linorder_cases*[*of x y*])
apply (*rule* ∗; *assumption*)
apply (*simp*; *fail*)
apply (*subst Int_commute*; *rule* ∗; *assumption*)
done

```
    qed
  next
    case 3
    show ?case
    proof (subst sum.reindex, unfold inj_on_def, clarsimp, goal_cases)
      case prems: (1 x y)
      with *[of y x] *[of x y] valid_line_aux[of n - Suc x] show ?case
        by - (rule linorder_cases[of x y], auto)
    next
      case 2
      then show ?case
        by (simp add: F_def card_image[OF inj])
    qed
  qed (auto intro: finite_subset[OF _ finite_valid_length])

  show ?thesis
    apply (subst F_def)
    unfolding ‹?A = _› ‹?B = _› ‹?C = _› D_eq
    apply (subst card_Un_disjoint)

       apply (blast intro: finite_subset[OF _ finite_valid_length])+

    subgoal
      using Cons_replicate_eq[of B _ n R] replicate_unequal_aux by fast-
force
    apply (subst card_Un_disjoint)

       apply (blast intro: finite_subset[OF _ finite_valid_length])+

      unfolding 1 2 3 using ‹m > 0› by (auto simp: Cons_replicate_eq
Cons_eq_append_conv)
qed
```

3.4.4 Computing the Fill-Count Function

```
fun lcount :: nat ⇒ nat where
  lcount n = (
    if n < m then 1
    else if n = m then 2
    else lcount (n - 1) + 1 + (∑ i ← [m..<n]. lcount (n - i - 1))
  )

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

lemma *lcount_correct*:
  lcount n = *F n* **if** *m* > 0
proof (*induction n rule*: *less_induct*)
  case (*less n*)
  from ‹*m* > 0› **show** *?case*
    apply (*cases n* = 0)
    subgoal
      by (*simp add*: *lcount.simps F_base_*0)
    by (*subst lcount.simps*)
    (*simp add*: *less.IH F_base_*1 *F_base_*2 *F_rec interv_sum_list_conv_sum_set_nat*)
qed

3.4.5 Memoization

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

memoize_correct
  by *memoize_prover*

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 −
 have *lcount* 50 168 > 1000000
  by *eval*
 moreover have ∀ *n* ∈ {0..<168}. *lcount* 50 *n* < 1000000
  by *eval*
 ultimately show *?thesis*
  by − (*rule Least_equality*; *rule ccontr*; *force simp*: *not_le lcount_correct*)
qed

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*:
 $w = u@v \longleftrightarrow (\exists\, k \in \{0..length\ w\}.\ u = take\ k\ w \wedge v = drop\ k\ w)$
by (*metis* (*full_types*) *append_eq_conv_conj append_take_drop_id atLeastAtMost_iff le*0 *le_add*1 *length_append*)

lemma *append_iff_take_drop*1: $u \neq [] \Longrightarrow v \neq [] \Longrightarrow$
 $w = u@v \longleftrightarrow (\exists\, k \in \{1..length\ w - 1\}.\ u = take\ k\ w \wedge v = drop\ k\ w)$
by(*auto simp*: *append_iff_take_drop*)

3.5.2 Definitions

```
datatype ('n, 't) rhs = NN 'n 'n | T 't

type_synonym ('n, 't) prods = ('n × ('n, 't) rhs) list

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

inductive yield :: 'n ⇒ 't list ⇒ bool where
(A, T a) ∈ set P ⟹ yield A [a] |
⟦ (A, NN B C) ∈ set P; yield B u; yield C v ⟧ ⟹ yield A (u@v)

lemma yield_not_Nil: yield A w ⟹ w ≠ []
by (induction rule: yield.induct) auto

lemma yield_eq1:
  yield A [a] ⟷ (A, T a) ∈ set P (is ?L = ?R)
proof
  assume ?L thus ?R
    by(induction A [a] arbitrary: a rule: yield.induct)
      (auto simp add: yield_not_Nil append_eq_Cons_conv)
qed (simp add: yield.intros)

lemma yield_eq2: assumes length w > 1
shows yield A w ⟷ (∃ B u C v. yield B u ∧ yield C v ∧ w = u@v ∧ (A,
NN B C) ∈ set P)
       (is ?L = ?R)
proof
  assume ?L from this assms show ?R
    by(induction rule: yield.induct) (auto)
next
  assume ?R with assms show ?L
    by (auto simp add: yield.intros)
qed
```

3.5.3 CYK on Lists

```
fun cyk :: 't list ⇒ 'n list where
cyk [] = [] |
cyk [a] = [A . (A, T a') <− P, a'= a] |
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* ≥ 2 ⟹ *set*(*cyk w*) =
(⋃ *k* ∈ {1..*length w* − 1}. ⋃ *B* ∈ *set*(*cyk* (*take k w*)). ⋃ *C* ∈ *set*(*cyk* (*drop k w*)). {*A*. (*A*, *NN B C*) ∈ *set P*})
apply(*cases w*)
 apply *simp*
subgoal for _ *w′*
apply(*case_tac w′*)
 apply *auto*
    apply *force*
   apply *force*
  apply *force*
 using *le_Suc_eq le_simps*(3) **apply** *auto*[1]
by (*metis drop_Suc_Cons le_Suc_eq le_antisym not_le take_Suc_Cons*)
done

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

lemma *cyk_correct*: *set*(*cyk w*) = {*N*. *yield N w*}
proof (*induction w rule*: *cyk.induct*)
  case 1 **thus** *?case* **by** (*auto dest*: *yield_not_Nil*)
next
  case 2 **thus** *?case* **by** (*auto simp add*: *yield_eq1*)
next
  case (3 *v vb vc*)
  let *?w* = *v* # *vb* # *vc*
  have *set*(*cyk ?w*) = (⋃ *k*∈{1..*length ?w*−1}. {*N*. ∃ *A B*. (*N*, *NN A B*) ∈ *set P* ∧
            yield A (*take k ?w*) ∧ *yield B* (*drop k ?w*)})
    by(*auto simp add*:3.*IH simp del*:*upt_Suc*)
  also have ... = {*N*. ∃ *A B*. (*N*, *NN A B*) ∈ *set P* ∧
            (∃ *u v*. *yield A u* ∧ *yield B v* ∧ *?w* = *u*@*v*)}
    by(*fastforce simp add*: *append_iff_take_drop1 yield_not_Nil*)
  also have ... = {*N*. *yield N ?w*} **using** *yield_eq2*[*of ?w*] **by**(*auto*)
  finally show *?case* **.**
qed

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: (⋃xb∈set P. {A. (A, NN B C) = xb}) = {A. (A, NN B C) ∈ set P}
by auto

lemma cyk2_eq_cyk: i+n ≤ length w ⟹ set(cyk2 w (i,n)) = set(cyk (take n (drop i w)))
proof(induction w (i,n) arbitrary: i n rule: cyk2.induct)
  case 1 **show** ?case **by**(simp)
next
  case 2 **show** ?case **using** 2.prems
    by(auto simp: hd_drop_conv_nth take_Suc)
next
  case (3 w i m)
  show ?case **using** 3.prems
   by(simp add: 3(1,2) min.absorb1 min.absorb2 drop_take atLeastLessThanSuc_atLeastAtMost set_aux
        del:upt_Suc cong: SUP_cong_simp)
      (simp add: add.commute)
qed

definition CYK S w = (S ∈ set(cyk2 w (0, length w)))

theorem CYK_correct: CYK S w = yield S w
by(simp add: CYK_def cyk2_eq_cyk cyk_correct)

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))) =

$(\bigcup k \in \{1..Suc\ n\}.\ \bigcup B \in set(cyk\_ix\ (i,k)).\ \bigcup C \in set(cyk\_ix\ (i+k,n+2-k)).$
$\{A.\ (A,\ NN\ B\ C) \in set\ P\})$
by(*simp add*: *atLeastLessThanSuc_atLeastAtMost set_aux del*: *upt_Suc*)

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

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

lemma *slice_append_iff_take_drop*1: $u \neq [] \Longrightarrow v \neq [] \Longrightarrow$
$slice\ w\ i\ j = u\ @\ v \longleftrightarrow (\exists k.\ 1 \le k \wedge k \le j-i-1 \wedge slice\ w\ i\ (i + k) = u$
$\wedge\ slice\ w\ (i + k)\ j = v)$
by(*subst append_iff_take_drop*1) (*auto simp*: *take_map drop_map Bex_def*)

lemma *cyk_ix_correct*:
$set(cyk\_ix\ (i,n)) = \{N.\ yield\ N\ (slice\ w\ i\ (i+n))\}$
proof (*induction* (*i*,*n*) *arbitrary*: *i n rule*: *cyk_ix.induct*)
case 1 **thus** *?case* **by** (*auto simp*: *dest*: *yield_not_Nil*)
next
case 2 **thus** *?case* **by** (*auto simp add*: *yield_eq*1)
next
case (3 *i m*)
let $?n = Suc(Suc\ m)$ **let** $?w = slice\ w\ i\ (i+?n)$
have $set(cyk\_ix\ (i,?n)) = (\bigcup k{\in}\{1..Suc\ m\}.\ \{N.\ \exists A\ B.\ (N,\ NN\ A\ B) \in$
$set\ P\ \wedge$
$yield\ A\ (slice\ w\ i\ (i+k)) \wedge yield\ B\ (slice\ w\ (i+k)\ (i+?n))\})$
by(*auto simp add*: 3 *cyk_ix_simp*2 *simp del*: *upt_Suc*)
also have ... $= \{N.\ \exists A\ B.\ (N,\ NN\ A\ B) \in set\ P\ \wedge$
$(\exists u\ v.\ yield\ A\ u \wedge yield\ B\ v \wedge slice\ w\ i\ (i+?n) = u@v)\}$
by(*fastforce simp del*: *upt_Suc simp*: *slice_append_iff_take_drop*1 *yield_not_Nil*
cong: *conj_cong*)
also have ... $= \{N.\ yield\ N\ ?w\}$ **using** *yield_eq*2[*of ?w*] **by**(*auto*)
finally show *?case* **.**
qed

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
by *memoize_prover*
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
  by *memoize_prover*

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*
  using *init_success*[*of _ cyk_ix*$_h$*'* (*i*, *j*), *OF cyk_ix*$_h$*.crel*]
  by (*simp add*: *cyk_ix_impl_def index_size_defs*)

lemma *min_wpl_heap*:
  cyk_ix (*i*, *j*) = *result_of* (*cyk_ix_impl i j*) *Heap.empty*
  unfolding *cyk_ix_impl_def*
  using *memoized_empty*[*of _ cyk_ix*$_h$*'* (*i*, *j*), *OF cyk_ix*$_h$*.crel*]
  by (*simp add*: *index_size_defs*)

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*)
by(*simp add*: *CYK_ix_def cyk_ix_correct*)

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*)
unfolding *CYK_ix_impl_def*
by (*simp add*: *execute_bind_success*[*OF cyk_ix_impl_success*]
min_wpl_heap[*symmetric*] *CYK_ix_correct CYK_ix_def*[*symmetric*]
)

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 (λ*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* (λ*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* (λ*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 ‹*List.map* (*fn f* => *f* ()) *Test.test*›

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* ⇒ '*b*::*order*) ⇒ '*a list* ⇒ '*a* **where**
argmin f [*a*] = *a* |

argmin f (*a*#*as*) = (*let m* = *argmin f as in if f a* ≤ *f m then a else m*)

fun *argmin2* :: (*'a* ⇒ *'b*::*order*) ⇒ *'a list* ⇒ *'a* ∗ *'b* **where**
argmin2 f [*a*] = (*a*, *f a*) |
argmin2 f (*a*#*as*) = (*let fa* = *f a*; (*am*,*m*) = *argmin2 f as in if fa* ≤ *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* ⇒ *'a list* ⇒ *'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 ≡ *length* [*e* <− *es*. *e* ≠ *Copy*]

3.6.3 Minimum Edit Sequence

fun *min_eds* :: *'a list* ⇒ *'a list* ⇒ *'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''*]
by *eval*

lemma *min_eds_correct*: *edit* (*min_eds xs ys*) *xs* = *ys*
by (*induction xs ys rule*: *min_eds.induct*) *auto*

```
lemma min_eds_same: min_eds xs xs = replicate (length xs) Copy
by (induction xs) auto

lemma min_eds_eq_Nil_iff: min_eds xs ys = [] ⟷ xs = [] ∧ ys = []
by (induction xs ys rule: min_eds.induct) auto

lemma min_eds_Nil: min_eds [] ys = map Ins ys
by (induction ys) auto

lemma min_eds_Nil2: min_eds xs [] = replicate (length xs) Del
by (induction xs) auto

lemma if_edit_Nil2: edit es ([]::'a list) = ys ⟹ length ys ≤ cost es
apply(induction es []::'a list arbitrary: ys rule: edit.induct)
apply auto
done

lemma if_edit_eq_Nil: edit es xs = [] ⟹ length xs ≤ cost es
by (induction es xs rule: edit.induct) auto

lemma min_eds_minimal: edit es xs = ys ⟹ cost(min_eds xs ys) ≤ cost
es
proof(induction xs ys arbitrary: es rule: min_eds.induct)
  case 1 thus ?case by simp
next
  case 2 thus ?case by (auto simp add: min_eds_Nil dest: if_edit_Nil2)
next
  case 3
  thus ?case by(auto simp add: min_eds_Nil2 dest: if_edit_eq_Nil)
next
  case 4
  show ?case
  proof (cases es)
   case Nil then show ?thesis using 4.prems by (auto simp: min_eds_same)
  next
    case [simp]: (Cons e es')
    show ?thesis
    proof (cases e)
      case Copy
      thus ?thesis using 4.prems 4.IH(3)[of es'] by simp
    next
      case (Repl a)
      thus ?thesis using 4.prems 4.IH(3)[of es']
        using [[simp_depth_limit=1]] by simp
```

```
    next
      case (Ins a)
      thus ?thesis using 4.prems 4.IH(1)[of es']
        using [[simp_depth_limit=1]] by auto
    next
      case Del
      thus ?thesis using 4.prems 4.IH(2)[of es']
        using [[simp_depth_limit=1]] by auto
    qed
  qed
qed
```

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)
apply(induction xs ys rule: min_ed.induct)
apply (auto split!: if_splits)
done
```

```
lemma min_ed ''madagascar'' ''bananas'' = 6
by eval
```

Exercise: Optimization of the Copy case

```
fun min_eds2 :: 'a list ⇒ 'a list ⇒ 'a ed list where
min_eds2 [] [] = [] |
min_eds2 [] (y#ys) = Ins y # min_eds2 [] ys |
min_eds2 (x#xs) [] = Del # min_eds2 xs [] |
min_eds2 (x#xs) (y#ys) =
  (if x=y then Copy # min_eds2 xs ys
   else argmin cost
     [Ins y # min_eds2 (x#xs) ys, Del # min_eds2 xs (y#ys), Repl y #
min_eds2 xs ys])
```

```
value min_eds2 ''madagascar'' ''bananas''
```

```
lemma cost_Copy_Del: cost(min_eds xs ys) ≤ cost (min_eds xs (x#ys))
```

+ 1
apply(*induction xs ys rule*: *min_eds.induct*)
apply(*auto simp del*: *filter_True filter_False split*!: *if_splits*)
done

lemma *cost_Copy_Ins*: *cost*(*min_eds xs ys*) ≤ *cost* (*min_eds* (*x#xs*) *ys*) + 1
apply(*induction xs ys rule*: *min_eds.induct*)
apply(*auto simp del*: *filter_True filter_False split*!: *if_splits*)
done

lemma *cost*(*min_eds*2 *xs ys*) = *cost*(*min_eds xs ys*)
proof(*induction xs ys rule*: *min_eds*2.*induct*)
  case (4 *x xs y ys*) **thus** *?case*
    apply (*auto split*!: *if_split*)
    apply (*metis* (*mono_tags*, *lifting*) *Suc_eq_plus*1 *Suc_leI cost_Copy_Del cost_Copy_Ins le_imp_less_Suc le_neq_implies_less not_less*)
     apply (*metis Suc_eq_plus*1 *cost_Copy_Del le_antisym*)
    by (*metis Suc_eq_plus*1 *cost_Copy_Ins le_antisym*)
qed *simp_all*

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

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)})
by *pat_completeness auto*
termination by(*relation measure*(λ(*i*,*j*). (*m* − *i*) + (*n* − *j*))) *auto*

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*)
apply(*induction* (*i*,*j*) *arbitrary*: *i j rule*: *min_ed_ix′.induct*[*of length xs length ys*])
apply(*subst min_ed_ix′.simps*)
apply(*simp add*: *Cons_nth_drop_Suc*[*symmetric*])
done

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)])
by *pat_completeness auto*
termination by(*relation measure*(λ(*i*,*j*). (*m* − *i*) + (*n* − *j*))) *auto*

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
by *memoize_prover*
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* ≤ *m* ∧ *y* ≤ *n* ∧ *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*)
  by (*rule table_iterator_down*)

lemma [*intro*]:
  dp_consistency_heap_array_pair' (*n* + 1) *fst snd id m* (*m* + 1) *mem*
  by (*standard*; *simp add*: *mem_is_init injective_def*)

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*))
    (λ (*x*, *y*). *x* ≤ *m* ∧ *y* ≤ *n* ∧ *x* > 0)

  by (*standard*; *simp add*: *mem_is_init injective_def*)

memoize_fun $min\_ed\_ix_h$: *min_ed_ix*
 with_memory (**default_proof**) *dp_consistency_heap_array_pair_iterator*
  where *size* = *n* + 1
    and *key*1=*fst* :: *nat* × *nat* ⇒ *nat* **and** *key*2=*snd* :: *nat* × *nat* ⇒ *nat*
    and *k*1=*m* :: *nat* **and** *k*2=*m* + 1 :: *nat*
    and *to_index* = *id* :: *nat* ⇒ *nat*
    and *mem* = *mem*
    and *cnt* = λ (*x*, *y*). *x* ≤ *m* ∧ *y* ≤ *n* ∧ *x* > 0
    and *nxt* = λ (*x*::*nat*, *y*). *if* *y* > 0 *then* (*x*, *y* − 1) *else* (*x* − 1, *n*)
    and *sizef* = λ (*x*, *y*). (*m* − *x*) ∗ (*n* + 1) + (*n* − *y*)
monadifies (*heap*) *min_ed_ix.simps*

memoize_correct

by *memoize_prover*

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_Nil*1: *min_ed* [] *ys* = *length ys*
by (*induction ys*) *auto*

lemma *min_ed_Nil*2: *min_ed xs* [] = *length xs*
by (*induction xs*) *auto*

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*)
apply(*induction* (*i*,*j*) *arbitrary*: *i j rule*: *min_ed_ix*.*induct*[*of m n*])
apply(*simp add*: *min_ed_ix*.*simps upt_conv_Cons min_ed_Nil*1 *min_ed_Nil*2 *Suc_diff_Suc*)
done

Functional Test Cases

definition *min_ed_list xs ys* = *min_ed_ix* (λ*i*. *xs*!*i*) (λ*i*. *ys*!*i*) (*length xs*) (*length ys*) (0,0)

lemma *min_ed_list* ''*madagascar*'' ''*bananas*'' = 6
by *eval*

definition *min_ed_ia xs ys* = (*let a* = *IArray xs*; *b* = *IArray ys*
in min_ed_ix (λ*i*. *a*!!*i*) (λ*i*. *b*!!*i*) (*length xs*) (*length ys*) (0,0))

lemma *min_ed_ia* ''*madagascar*'' ''*bananas*'' = 6
by *eval*

Extracting an Executable Constant for the Imperative Implementation

ground_function *min_ed_ix*$_h$'*_impl*: *min_ed_ix*$_h$'.*simps*

termination
by(*relation measure*(λ(*xs*, *ys*, *m*, *n*, *mem*, *i*, *j*). (*m* − *i*) + (*n* − *j*))) *auto*

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* × *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*
shows *min_ed_ix*$_h$*'_impl xs ys m n mem* = *min_ed_ix*$_h$*' xs ys m n mem*
proof −
have *min_ed_ix*$_h$*'_impl xs ys m n mem* (*i*, *j*) = *min_ed_ix*$_h$*' xs ys m n mem* (*i*, *j*) **for** *i j*
apply (*induction rule*: *min_ed_ix*$_h$*'.induct*[*OF mem_is_init*])
apply (*subst min_ed_ix*$_h$*'_impl.simps*)
apply (*subst min_ed_ix*$_h$*'.simps*[*OF mem_is_init*])
apply (*solve_cong simp*)
done
then show *?thesis*
by *auto*
qed

definition
iter_min_ed_ix xs ys m n mem = *iterator_defs.iter_heap*
(λ (*x*, *y*). *x* ≤ *m* ∧ *y* ≤ *n* ∧ *x* > 0)
(λ (*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 = (λ (*i*, *j*).
(*if i* > 0 ∧ *i* ≤ *m* ∧ *j* ≤ *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 ()))
unfolding *iter_min_ed_ix_def* **by** (*rule ext*) (*safe*, *simp add*: *iter_heap_unfold*)

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*
  using *memoized_empty*[*OF HOL.refl*, *of xs ys m n* (*i*, *j*) λ _. (*m*, *n*)]
  by (*simp add*:
      execute_bind_success[*OF succes_init_state*] *min_ed_ix_impl_def* *min_ed_ix_h'_impl_def*
      iter_min_ed_ix_def
)

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 ‹*Test.test_case* ()›

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

Note that *arg_min_list* is similar but returns the leftmost element.

lemma *argmin_forall*: *xs* ≠ [] ⟹ (⋀*x*. *x*∈*set xs* ⟹ *P x*) ⟹ *P* (*argmin f xs*)
by(*induction xs*) (*auto simp*: *Let_def*)

lemma *argmin_Min*: *xs* ≠ [] ⟹ *f* (*argmin f xs*) = *Min* (*f* ‘ *set xs*)
by(*induction xs*) (*auto simp*: *min_def intro*!: *antisym*)

3.7.2 Misc

lemma *upto_join*: ⟦ *i* ≤ *j*; *j* ≤ *k* ⟧ ⟹ [*i*..*j*−1] @ *j* # [*j*+1..*k*] = [*i*..*k*]
using *upto_rec*1 *upto_split*1 **by** *auto*

lemma *atLeastAtMost_split*:
{*i*..*j*} = {*i*..*k*} ∪ {*k*+1..*j*} **if** *i* ≤ *k k* ≤ *j* **for** *i j k* :: *int*
using *that* **by** *auto*

lemma *atLeastAtMost_split_insert*:
{*i*..*k*} = *insert k* {*i*..*k*−1} **if** *k* ≥ *i* **for** *i* :: *int*
using *that* **by** *auto*

3.7.3 Definitions

context
fixes *W* :: *int* ⇒ *int* ⇒ *nat*
begin

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

function min_wpl :: int ⇒ int ⇒ nat where
min_wpl i j =
  (if i > j then 0
   else min_list (map (λk. min_wpl i (k−1) + min_wpl (k+1) j + W i j)
[i..j]))
  by auto
termination by (relation measure (λ(i,j) . nat(j−i+1))) auto
declare min_wpl.simps[simp del]

function opt_bst :: int ⇒ int ⇒ int tree where
opt_bst i j =
  (if i > j then Leaf else argmin (wpl i j) [⟨opt_bst i (k−1), k, opt_bst (k+1)
j⟩. k ← [i..j]])
  by auto
termination by (relation measure (λ(i,j) . nat(j−i+1))) auto
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
  by memoize_prover

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*$_h$ ≡ λ *i j*. *Heap_Monad.bind* (*mem_empty* (*n* ∗ *n*)) (λ *mem*. *min_wpl*$_T$′ *mem i j*)

lemma *min_wpl_heap*:
  min_wpl i j = *result_of* (*min_wpl*$_h$ *i j*) *Heap.empty*
  unfolding *min_wpl*$_h$*_def*
  using *memoized_empty*[*of* _ λ *m*. λ (*a*, *b*). *min_wpl*$_T$′ *m a b* (*i*, *j*), *OF min_wpl*$_T$*.crel*]
  by (*simp add*: *index_size_defs*)

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
  by *memoize_prover*
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
  by *memoize_prover*
print_theorems
lemmas [*code*] = *opt_bst*$_m$*.memoized_correct*

end

3.7.5 Correctness Proof

```
lemma min_wpl_minimal:
  inorder t = [i..j] ⟹ min_wpl i j ≤ wpl i j t
proof(induction i j t rule: wpl.induct)
  case (1 i j)
  then show ?case by (simp add: min_wpl.simps)
next
  case (2 i j l k r)
  then show ?case
  proof cases
    assume i > j thus ?thesis by(simp add: min_wpl.simps)
  next
    assume [arith]: ¬ i > j
    have kk_ij: k∈set[i..j] using 2
        by (metis set_inorder tree.set_intros(2))

    let ?M = ((λk. min_wpl i (k−1) + min_wpl (k+1) j + W i j) ` {i..j})
    let ?w = min_wpl i (k−1) + min_wpl (k+1) j + W i j

    have aux_min:Min ?M ≤ ?w
    proof (rule Min_le)
      show finite ?M by simp
      show ?w ∈ ?M using kk_ij by auto
    qed

    haveinorder ⟨l,k,r⟩ = inorder l @k#inorder r by auto
    from this have C:[i..j] = inorder l @ k#inorder r using 2 by auto
    have D: [i..j] = [i..k−1]@k#[k+1..j] using kk_ij upto_rec1 upto_split1
      by (metis atLeastAtMost_iff set_upto)

    have l_inorder: inorder l = [i..k−1]
      by (smt C D append_Cons_eq_iff atLeastAtMost_iff set_upto)
    have r_inorder: inorder r = [k+1..j]
      by (smt C D append_Cons_eq_iff atLeastAtMost_iff set_upto)

   have min_wpl i j = Min ?M by (simp add: min_wpl.simps min_list_Min)
    also have ... ≤ ?w by (rule aux_min)
    also have ... ≤ wpl i (k−1) l + wpl (k+1) j r + W i j using l_inorder
r_inorder 2.IH by simp
    also have ... = wpl i j ⟨l,k,r⟩ by simp
    finally show ?thesis .
  qed
qed
```

lemma *opt_bst_correct*: *inorder* (*opt_bst i j*) = [*i*..*j*]
by (*induction i j rule*: *opt_bst.induct*)
(*clarsimp simp*: *opt_bst.simps upto_join* | *rule argmin_forall*)+

lemma *wpl_opt_bst*: *wpl i j* (*opt_bst i j*) = *min_wpl i j*
proof(*induction i j rule*: *min_wpl.induct*)
case (1 *i j*)
show *?case*
proof *cases*
assume *i* > *j* **thus** *?thesis* **by**(*simp add*: *min_wpl.simps opt_bst.simps*)
next
assume ∗[*arith*]: ¬ *i* > *j*
let *?ts* = [⟨*opt_bst i* (*k*−1), *k*, *opt_bst* (*k*+1) *j*⟩. *k* <− [*i*..*j*]]
let *?M* = ((λ*k*. *min_wpl i* (*k*−1) + *min_wpl* (*k*+1) *j* + *W i j*) ‘ {*i*..*j*})
have *?ts* ≠ [] **by** (*auto simp add*: *upto.simps*)
have *wpl i j* (*opt_bst i j*) = *wpl i j* (*argmin* (*wpl i j*) *?ts*) **by** (*simp add*: *opt_bst.simps*)
also have . . . = *Min* (*wpl i j* ‘ (*set ?ts*)) **by** (*rule argmin_Min*[*OF* ‹*?ts* ≠ []›])
also have . . . = *Min ?M*
proof (*rule arg_cong*[**where** *f*=*Min*])
show *wpl i j* ‘ (*set ?ts*) = *?M*
by (*fastforce simp*: *Bex_def image_iff* 1[*OF* ∗])
qed
also have . . . = *min_wpl i j* **by** (*simp add*: *min_wpl.simps min_list_Min*)
finally show *?thesis* **.**
qed
qed

lemma *opt_bst_is_optimal*:
inorder t = [*i*..*j*] =⇒ *wpl i j* (*opt_bst i j*) ≤ *wpl i j t*
by (*simp add*: *min_wpl_minimal wpl_opt_bst*)

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)
  unfolding *W_def* **by** (*simp add*: *atLeastAtMost_split_insert*)

— The weight function correctly implements costs.
lemma *inorder_wpl_correct*:
  inorder t = [*i..j*] ⟹ *wpl W i j t* = *cost t*
proof (*induction t arbitrary*: *i j*)
case *Leaf*
  show *?case*
    by *simp*
next
  case (*Node l k r*)
  from ‹*inorder* ⟨*l*, *k*, *r*⟩ = [*i..j*]› **have** ∗: *i* ≤ *k k* ≤ *j*
    by − (*simp*, *metis atLeastAtMost_iff in_set_conv_decomp set_upto*)+
   moreover from ‹*i* ≤ *k*› ‹*k* ≤ *j*› **have** *inorder l* = [*i..k*−1] *inorder r* = [*k*+1..*j*]
    using ‹*inorder* ⟨*l*, *k*, *r*⟩ = [*i..j*]›[*symmetric*] **by** (*simp add*: *upto_split*3 *append_Cons_eq_iff*)+
  ultimately show *?case*
    by (*simp add*: *Node.IH*, *subst W_def*, *subst atLeastAtMost_split*)
     (*simp add*: *sum.union_disjoint atLeastAtMost_split_insert flip*: *set_inorder*)+
qed

The optimal binary search tree has minimal cost among all binary search trees.

lemma *opt_bst_has_optimal_cost*:

inorder t = [*i..j*] ⟹ *cost* (*opt_bst W i j*) ≤ *cost t*
using *inorder_wpl_correct opt_bst_is_optimal opt_bst_correct* **by** *metis*

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*] ⟹ *min_wpl W i j* ≤ *cost t*
using *inorder_wpl_correct min_wpl_minimal* **by** *metis*

lemma *min_wpl_tree*:
cost (*opt_bst W i j*) = *min_wpl W i j*
using *wpl_opt_bst opt_bst_correct inorder_wpl_correct* **by** *metis*

An alternative view of costs. **fun** *depth* :: *'a* ⇒ *'a tree* ⇒ *nat extended* **where**
depth x Leaf = ∞
| *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* | *the_fin* _ = *undefined*

definition *cost'* :: *int tree* ⇒ *nat* **where**
cost' t = *sum* (λ*x*. *the_fin* (*depth x t*) ∗ *p x*) (*set_tree t*)

lemma [*simp*]:
the_fin 1 = 1
by (*simp add*: *one_extended_def*)

lemma *set_tree_depth*:
assumes *x* ∉ *set_tree t*
shows *depth x t* = ∞
using *assms* **by** (*induction t*) *auto*

lemma *depth_inf_iff*:
depth x t = ∞ ⟷ *x* ∉ *set_tree t*
apply (*induction t*)
apply (*auto simp*: *one_extended_def*)
subgoal for *t*1 *k t*2

```
    by (cases depth x t1; cases depth x t2) auto
  subgoal for t1 k t2
    by (cases depth x t1; cases depth x t2) auto
  subgoal for t1 k t2
    by (cases depth x t1; cases depth x t2) auto
  subgoal for t1 k t2
    by (cases depth x t1; cases depth x t2) auto
  done

lemma depth_not_neg_inf[simp]:
  depth x t = −∞ ⟷ False
  apply (induction t)
   apply (auto simp: one_extended_def)
  subgoal for t1 k t2
    by (cases depth x t1; cases depth x t2) auto
  done

lemma depth_FinD:
  assumes x ∈ set_tree t
  obtains d where depth x t = Fin d
  using assms by (cases depth x t) (auto simp: depth_inf_iff)

lemma cost'_Leaf[simp]:
  cost' Leaf = 0
  unfolding cost'_def by simp

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)
  unfolding cost'_def
  apply simp
  apply (subst sum.union_disjoint)
     apply (simp; fail)+
  apply (subst sum.cong[OF HOL.refl, where h = λx. (the_fin (depth x l)
+ 1) ∗ p x])
  subgoal for k
   using set_tree_depth by (force simp: one_extended_def elim: depth_FinD)
  apply (subst (2) sum.cong[OF HOL.refl, where h = λx. (the_fin (depth
x r) + 1) ∗ p x])
  subgoal
   using set_tree_depth by (force simp: one_extended_def elim: depth_FinD)
  apply (simp add: sum.distrib)
  done
```

— The two variants coincide

lemma *weight_correct*:
  distinct (*inorder t*) $\Longrightarrow$ *cost′ t* = *cost t*
  by (*induction t*; *simp add*: *cost′_Node*)

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*)
  by *auto*

termination
  by (*relation measure* (λ(*i*::*int*, *j*::*int*). *nat* (*j* − *i* + 1))) *auto*

lemma *W_fun_correct*:
  W_fun i j = *W i j*
  by (*induction rule*: *W_fun.induct*) (*simp add*: *W_def atLeastAtMost_split_insert*)

memoize_fun W_m: *W_fun*
  with_memory *dp_consistency_mapping*
  monadifies (*state*) *W_fun.simps*

memoize_correct
  by *memoize_prover*

definition
  compute_W n = *snd* (*run_state* (*State_Main.map*$_T$*′* (λ*i*. W_m*′ i n*) [0..*n*]) *Mapping.empty*)

notation W_m*.crel_vs* (‹*crel*›)

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*
proof −
  include *state_monad_syntax* **and** *app_syntax* **and** *lifting_syntax*
  let *?p* = *State_Main.map*$_T$*′* (λ*i*. W_m*′ i n*) [0..*n*]
  let *?q* = *map* (λ*i*. *W i n*) [0..*n*]
  have *?q* = *map* $ ⟪(λ*i*. *W_fun i n*)⟫ $ ⟪[0..*n*]⟫
    unfolding *Wrap_def App_def W_fun_correct* **..**

have *?p* = *State_Main.map*$_T$. ⟨λ*i*. W_m′ *i n*⟩ . ⟨[0..*n*]⟩
unfolding *State_Monad_Ext.fun_app_lifted_def State_Main.map*$_T$*_def bind_left_identity* **..**
— Not forgetting to write *list_all2* (=) instead of (=) was the tricky part.
have W_m*.crel_vs* (*list_all2* (=)) *?q ?p*
unfolding ‹*?p* = _› ‹*?q* = _›
apply (*subst Transfer.Rel_def*[*symmetric*])
apply *memoize_prover_match_step*+
apply (*subst Rel_def*, *rule* W_m*_crel*, *rule HOL.refl*)
done
then have W_m*.cmem* (*compute_W n*)
unfolding *compute_W_def* **by** (*elim* W_m*.crel_vs_elim*[*OF* _ W_m*.cmem_empty*]; *simp del*: W_m′*.simps*)
with *assms* **show** *?thesis*
unfolding *W_fun_correct*[*symmetric*] **by** (*elim* W_m*.cmem_elim*) (*simp*)+
qed

definition
min_wpl′ *i j* ≡
let
M = *compute_W j*;
W = (λ*i j*. *case Mapping.lookup M* (*i*, *j*) *of None* ⇒ *W i j* | *Some x* ⇒ *x*)
in min_wpl W i j

lemma *W_compute*: *W i j* = (*case Mapping.lookup* (*compute_W n*) (*i*, *j*) *of None* ⇒ *W i j* | *Some x* ⇒ *x*)
by (*auto dest*: *compute_W_correct split*: *option.split*)

lemma *min_wpl*′*_correct*:
min_wpl′ *i j* = *min_wpl W i j*
using *W_compute* **unfolding** *min_wpl*′*_def* **by** *simp*

definition
opt_bst′ *i j* ≡
let
M = *compute_W j*;
W = (λ*i j*. *case Mapping.lookup M* (*i*, *j*) *of None* ⇒ *W i j* | *Some x* ⇒ *x*)
in opt_bst W i j

lemma *opt_bst*′*_correct*:
opt_bst′ *i j* = *opt_bst W i j*
using *W_compute* **unfolding** *opt_bst*′*_def* **by** *simp*

end

3.7.8 Test Case

Functional Implementations

lemma *min_wpl* ($\lambda i\ j.\ nat(i{+}j)$) 0 4 = 10
  by *eval*

lemma *opt_bst* ($\lambda i\ j.\ nat(i{+}j)$) 0 4 = ⟨⟨⟨⟨⟨⟨⟩, 0, ⟨⟩⟩, 1, ⟨⟩⟩, 2, ⟨⟩⟩, 3, ⟨⟩⟩, 4, ⟨⟩⟩
  by *eval*

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 =
  ⟨
    ⟨
      ⟨⟨⟩, 1::*int*, ⟨⟩⟩,
      2,
      ⟨⟨⟩, 3, ⟨⟩⟩
    ⟩,
    4,
    ⟨⟨⟩, 5, ⟨⟩⟩
  ⟩

lemma *opt_bst′* (*list_to_p ex_p_*1) 1 5 = *opt_tree_*1
  by *eval*

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 ‹*Test.min_wpl_test* ()›

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 −
  have *?S* ⊆ {*xs*. *set xs* ⊆ *set ys* ∧ *length xs* ≤ *length ys*}
    by (*auto elim*: *list_emb_set intro*: *list_emb_length*)
  moreover have *finite* ...
    by (*intro finite_lists_length_le finite_set*)
  ultimately show *?thesis*
    by (*rule finite_subset*)
qed

lemma *subseq_singleton_right*:
  subseq xs [*x*] = (*xs* = [*x*] ∨ *xs* = [])
  by (*cases xs*; *simp add*: *subseq_append_le_same_iff*[*of* _ [], *simplified*])

lemma *subseq_append_single_right*:
  subseq xs (*ys* @ [*x*]) = ((∃ *xs'*. *subseq xs' ys* ∧ *xs* = *xs'* @ [*x*]) ∨ *subseq xs ys*)
  by (*auto simp*: *subseq_append_iff subseq_singleton_right*)

lemma *Max_nat_plus*:
  Max (((+) *n*) ‘ *S*) = (*n* :: *nat*) + *Max S* **if** *finite S S* ≠ {}
  using *that* **by** (*auto intro*!: *Max_ge Max_in Max_eqI*)

3.8.2 Definitions

context

fixes A B :: $'a$ *list*
begin

fun *lcs* :: *nat* ⇒ *nat* ⇒ *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* | *xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take j B*)}

lemma *finite_OPT*:
finite {*xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take j B*)} (**is** *finite ?S*)
proof −
have *?S* ⊆ {*xs*. *subseq xs* (*take i A*)}
by *auto*
moreover have *finite* ...
by (*rule finite_subseq*)
ultimately show *?thesis*
by (*rule finite_subset*)
qed

3.8.3 Correctness Proof

lemma *non_empty_OPT*:
{*xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take j B*)} ≠ {}
by *auto*

lemma *OPT_0_left*:
OPT 0 *j* = 0
unfolding *OPT_def* **by** (*simp add*: *subseq_append_le_same_iff*[*of* _ [], *simplified*])

lemma *OPT_0_right*:
OPT i 0 = 0
unfolding *OPT_def* **by** (*simp add*: *subseq_append_le_same_iff*[*of* _ [], *simplified*])

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*
proof −
let *?S* = {*length xs* |*xs*. *subseq xs* (*take* (*i* + 1) *A*) ∧ *subseq xs* (*take* (*j* +

1) *B*)}
  let *?R* = {*length xs* + 1 |*xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take j B*)}
  have *?S* = {*length xs* | *xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take j B*)}
    ∪ {*length xs* | *xs*. ∃ *ys*. *subseq ys* (*take i A*) ∧ *subseq ys* (*take j B*) ∧ *xs* = *ys* @ [*B*!*i*]}

    using *that*
    apply (*simp add*: *take_Suc_conv_app_nth*)
    apply (*simp add*: *subseq_append_single_right*)
    apply *auto*
    apply (*metis length_append_singleton list_emb_prefix subseq_append*)+
    done
  moreover have ... = {*length xs* | *xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take j B*)}
    ∪ {*length xs* + 1 | *xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take j B*)}
    by *force*
  moreover have *Max* ... = *Max ?R*
    using *finite_OPT* **by** − (*rule Max_eq_if*, *auto*)
  ultimately show *?l* = *?r*
    unfolding *OPT_def*
    using *finite_OPT non_empty_OPT*
    by (*subst Max_nat_plus*[*symmetric*]) (*auto simp*: *image_def intro*: *arg_cong*[**where** *f* = *Max*])
qed

lemma *OPT_rec*2:
  OPT (*i* + 1) (*j* + 1) = *max* (*OPT i* (*j* + 1)) (*OPT* (*i* + 1) *j*) (**is** *?l* = *?r*)
  if *A*!*i* ≠ *B*!*j* *i* < *length A j* < *length B*
proof −
  have {*length xs* |*xs*. *subseq xs* (*take* (*i* + 1) *A*) ∧ *subseq xs* (*take* (*j* + 1) *B*)}
    = {*length xs* |*xs*. *subseq xs* (*take i A*) ∧ *subseq xs* (*take* (*j* + 1) *B*)}
    ∪ {*length xs* |*xs*. *subseq xs* (*take* (*i* + 1) *A*) ∧ *subseq xs* (*take j B*)}
   using *that* **by** (*auto simp*: *subseq_append_single_right take_Suc_conv_app_nth*)
  with *finite_OPT non_empty_OPT* **show** *?l* = *?r*
    unfolding *OPT_def* **by** (*simp*) (*rule Max_Un*, *auto*)
qed

lemma *lcs_correct′*:
  OPT i j = *lcs i j* **if** *i* ≤ *length A j* ≤ *length B*
  using *that OPT_rec*1 *OPT_rec*2 **by** (*induction i j rule*: *lcs.induct*; *simp add*: *OPT_0_left OPT_0_right*)

theorem *lcs_correct*:
Max {*length xs* | *xs*. *subseq xs A* ∧ *subseq xs B*} = *lcs* (*length A*) (*length B*)
by (*simp add*: *OPT_def lcs_correct'*[*symmetric*])

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*
by (*induction i j rule*: *lcs_ia.induct*; *simp*; *simp add*: *that*)

memoize_fun lcs_m: *lcs_ia* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *lcs_ia.simps*

memoize_correct
by *memoize_prover*

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*
unfolding lcs_a*_def* **by** (*simp add*: *lcs_lcs_ia*)

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.