

# Monadification, Memoization and Dynamic Programming

Simon Wimmer      Shuwei Hu      Tobias Nipkow

Technical University of Munich

February 13, 2022

## 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	4
<b>1</b>	<b>Monadification</b>	<b>5</b>
1.1	Monads	5
1.2	Parametricity of the State Monad	5
1.3	Miscellaneous Parametricity Theorems	11
1.4	Heap Monad	12
1.5	Relation Between the State and the Heap Monad	13
1.6	Parametricity of the Heap Monad	20
<b>2</b>	<b>Memoization</b>	<b>25</b>
2.1	Memory Implementations for the State Monad	25
2.1.1	Tracing Memory	28
2.2	Pair Memory	30
2.3	Indexing	48
2.4	Heap Memory Implementations	56
2.5	Tool Setup	78

2.6	Bottom-Up Computation . . . . .	80
2.7	Setup for the Heap Monad . . . . .	86
2.7.1	More Heap . . . . .	91
2.7.2	Code Setup . . . . .	102
2.8	Setup for the State Monad . . . . .	102
2.8.1	Code Setup . . . . .	107
<b>3</b>	<b>Examples</b>	<b>107</b>
3.1	Misc . . . . .	107
3.2	The Bellman-Ford Algorithm . . . . .	110
3.2.1	Misc . . . . .	110
3.2.2	Single-Sink Shortest Path Problem . . . . .	116
3.2.3	Functional Correctness . . . . .	116
3.2.4	Functional Memoization . . . . .	119
3.2.5	Imperative Memoization . . . . .	121
3.2.6	Detecting Negative Cycles . . . . .	122
3.2.7	Extracting an Executable Constant for the Imperative Implementation . . . . .	135
3.2.8	Test Cases . . . . .	136
3.3	The Knapsack Problem . . . . .	139
3.3.1	Definitions . . . . .	140
3.3.2	Functional Correctness . . . . .	140
3.3.3	Functional Memoization . . . . .	142
3.3.4	Imperative Memoization . . . . .	143
3.3.5	Memoization . . . . .	144
3.3.6	Regression Test . . . . .	144
3.4	A Counting Problem . . . . .	145
3.4.1	Misc . . . . .	145
3.4.2	Problem Specification . . . . .	146
3.4.3	Combinatorial Identities . . . . .	146
3.4.4	Computing the Fill-Count Function . . . . .	152
3.4.5	Memoization . . . . .	153
3.4.6	Problem solutions . . . . .	153
3.5	The CYK Algorithm . . . . .	154
3.5.1	Misc . . . . .	154
3.5.2	Definitions . . . . .	155
3.5.3	CYK on Lists . . . . .	155
3.5.4	CYK on Lists and Index . . . . .	156
3.5.5	CYK With Index Function . . . . .	157
3.5.6	Correctness Proof . . . . .	157
3.5.7	Functional Memoization . . . . .	158
3.5.8	Imperative Memoization . . . . .	159
3.5.9	Functional Test Case . . . . .	160
3.5.10	Imperative Test Case . . . . .	161

3.6	Minimum Edit Distance . . . . .	161
3.6.1	Misc . . . . .	161
3.6.2	Edit Distance . . . . .	162
3.6.3	Minimum Edit Sequence . . . . .	162
3.6.4	Computing the Minimum Edit Distance . . . . .	164
3.6.5	Indexing . . . . .	165
3.6.6	Functional Memoization . . . . .	166
3.6.7	Imperative Memoization . . . . .	167
3.6.8	Test Cases . . . . .	168
3.7	Optimal Binary Search Trees . . . . .	170
3.7.1	Function <i>argmin</i> . . . . .	171
3.7.2	Misc . . . . .	171
3.7.3	Definitions . . . . .	171
3.7.4	Functional Memoization . . . . .	172
3.7.5	Correctness Proof . . . . .	174
3.7.6	Access Frequencies . . . . .	175
3.7.7	Memoizing Weights . . . . .	179
3.7.8	Test Case . . . . .	181
3.8	Longest Common Subsequence . . . . .	182
3.8.1	Misc . . . . .	182
3.8.2	Definitions . . . . .	182
3.8.3	Correctness Proof . . . . .	183
3.8.4	Functional Memoization . . . . .	185
3.8.5	Test Case . . . . .	185

## 0.1 State Monad

**theory** *State\_Monad\_Ext*

**imports** *HOL-Library.State\_Monad*  
**begin**

**definition** *fun\_app\_lifted* ::  $('M, 'a \Rightarrow ('M, 'b) \text{ state}) \text{ state} \Rightarrow ('M, 'a) \text{ state} \Rightarrow ('M, 'b) \text{ state}$  **where**

$\text{fun\_app\_lifted } f_T \ x_T \equiv \text{do } \{ f \leftarrow f_T; x \leftarrow x_T; f \ x \}$

**bundle** *state\_monad\_syntax* **begin**

**notation** *fun\_app\_lifted* (**infixl** . 999)

**type\_synonym**  $('a, 'M, 'b)$  *fun\_lifted* =  $'a \Rightarrow ('M, 'b) \text{ state} (\_ ==\_ \Longrightarrow \_)$   
[3,1000,2] 2)

**type\_synonym**  $('a, 'b)$  *dpfun* =  $'a == ('a \rightarrow 'b) \Longrightarrow 'b$  (**infixr**  $\Rightarrow_T$  2)

**type\_notation** *state* ([ $\_ | \_$ ])

**notation** *State\_Monad.return* ( $\langle \_ \rangle$ )

**notation** (*ASCII*) *State\_Monad.return* ( $(\# \_ \#)$ )

**notation** *Transfer.Rel* (*Rel*)

**end**

**context includes** *state\_monad\_syntax* **begin**

**qualified lemma** *return\_app\_return*:

$\langle f \rangle . \langle x \rangle = f \ x$

**unfolding** *fun\_app\_lifted\_def bind\_left\_identity* ..

**qualified lemma** *return\_app\_return\_meta*:

$\langle f \rangle . \langle x \rangle \equiv f \ x$

**unfolding** *return\_app\_return* .

**qualified definition** *if\_T* ::  $('M, \text{bool}) \text{ state} \Rightarrow ('M, 'a) \text{ state} \Rightarrow ('M, 'a) \text{ state} \Rightarrow ('M, 'a) \text{ state}$  **where**

$\text{if}_T \ b_T \ x_T \ y_T \equiv \text{do } \{ b \leftarrow b_T; \text{if } b \text{ then } x_T \text{ else } y_T \}$

**end**

**end**

# 1 Monadification

## 1.1 Monads

**theory** *Pure\_Monad*

**imports** *Main*

**begin**

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

$Wrap\ x \equiv x$

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

$App\ f \equiv f$

**lemma** *Wrap\_App\_Wrap*:

$App\ (Wrap\ f)\ (Wrap\ x) \equiv f\ x$

**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* ::  $('s \Rightarrow bool) \Rightarrow ('s, 'a)\ state \Rightarrow bool$  **where**

$lift\_p\ P\ f =$

$(\forall\ heap.\ P\ heap \longrightarrow (case\ State\_Monad.run\_state\ f\ heap\ of\ (\_,\ heap) \Rightarrow P\ heap))$

**context**

**fixes**  $P\ f\ heap$

**assumes** *lift*:  $lift\_p\ P\ f$  **and**  $P: P\ heap$

**begin**

**lemma** *run\_state\_cases*:

$case\ State\_Monad.run\_state\ f\ heap\ of\ (\_,\ heap) \Rightarrow P\ heap$

**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  $\Rightarrow$  ('mem, 'result option) state  
  **and** *update* :: 'param  $\Rightarrow$  'result  $\Rightarrow$  ('mem, unit) state  
**begin**

**definition** *checkmem* :: 'param  $\Rightarrow$  ('mem, 'result) state  $\Rightarrow$  ('mem, 'result) state **where**  
  *checkmem* param calc  $\equiv$  do {  
    *x*  $\leftarrow$  *lookup* param;  
    case *x* of  
      Some *x*  $\Rightarrow$  State\_Monad.return *x*  
  | None  $\Rightarrow$  do {  
    *x*  $\leftarrow$  calc;  
    *update* param *x*;  
    State\_Monad.return *x*  
  }  
}

**abbreviation** *checkmem\_eq* ::  
  ('param  $\Rightarrow$  ('mem, 'result) state)  $\Rightarrow$  'param  $\Rightarrow$  ('mem, 'result) state  $\Rightarrow$  bool  
  ( $\_ \$ \_ = \text{CHECKMEM} = \_ [1000,51] 51$ ) **where**  
  ( $dp_T \$ \text{param} = \text{CHECKMEM} = \text{calc}$ )  $\equiv$  ( $dp_T \text{param} = \text{checkmem param calc}$ )  
**term** 0

**definition** *map\_of* **where**  
  *map\_of* heap *k* = *fst* (*run\_state* (*lookup* *k*) heap)

**definition** *checkmem'* :: 'param  $\Rightarrow$  (unit  $\Rightarrow$  ('mem, 'result) state)  $\Rightarrow$  ('mem, 'result) state **where**  
  *checkmem'* param calc  $\equiv$  do {  
    *x*  $\leftarrow$  *lookup* param;  
    case *x* of  
      Some *x*  $\Rightarrow$  State\_Monad.return *x*  
  | None  $\Rightarrow$  do {  
    *x*  $\leftarrow$  calc ();  
    *update* param *x*;  
    State\_Monad.return *x*  
  }  
}

**lemma** *checkmem\_checkmem'*:  
*checkmem' param* ( $\lambda \_ . calc$ ) = *checkmem param calc*  
**unfolding** *checkmem'\_def checkmem\_def* ..

**lemma** *checkmem\_eq\_alt*:  
*checkmem\_eq dp param calc* = (*dp param* = *checkmem' param* ( $\lambda \_ . calc$ ))  
**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*:  $P\ m \implies \text{map\_of}\ (\text{snd}\ (\text{State\_Monad.run\_state}\ (\text{lookup}\ k)\ m)) \subseteq_m\ (\text{map\_of}\ m)$   
**and**  
*update\_correct*:  $P\ m \implies \text{map\_of}\ (\text{snd}\ (\text{State\_Monad.run\_state}\ (\text{update}\ k\ v)\ m)) \subseteq_m\ (\text{map\_of}\ m)(k \mapsto v)$

**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 state\_monad\_syntax*  
**begin**

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

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

**abbreviation** *rel\_fun\_lifted* :: (*'a*  $\Rightarrow$  *'c*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*'b*  $\Rightarrow$  *'d*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'c*  $\implies$  *'d*)  $\Rightarrow$  *bool* (**infixr**  $\implies_T$  55) **where**

*rel\_fun\_lifted*  $R R' \equiv R \implies \text{crel\_vs } R'$   
**term** 0

**definition** *consistentDP* :: ('param == 'mem  $\implies$  'result)  $\implies$  bool **where**  
*consistentDP*  $\equiv ((=) \implies \text{crel\_vs } (=)) \text{ dp}$   
**term** 0

**private lemma** *cmem\_intro*:  
**assumes**  $\bigwedge \text{param } v M'. \text{State\_Monad.run\_state } (\text{lookup param}) M =$   
 $(\text{Some } v, M') \implies v = \text{dp param}$   
**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*  $M \text{State\_Monad.run\_state } (\text{lookup param}) M = (\text{Some } v, M')$   
**obtains**  $\text{dp param} = v$   
**using** *assms unfolding cmem\_def dom\_def map\_of\_def* **by** *auto (metis fst\_conv option.inject)*  
**term** 0

**lemma** *crel\_vs\_intro*:  
**assumes**  $\bigwedge M v' M'. \llbracket \text{cmem } M; P M; \text{State\_Monad.run\_state } v_T M =$   
 $(v', M') \rrbracket \implies R v v' \wedge \text{cmem } M' \wedge P M'$   
**shows** *crel\_vs*  $R v v_T$   
**using** *assms unfolding crel\_vs\_def* **by** *blast*  
**term** 0

**lemma** *crel\_vs\_elim*:  
**assumes** *crel\_vs*  $R v v_T \text{cmem } M P M$   
**obtains**  $v' M'$  **where**  $\text{State\_Monad.run\_state } v_T M = (v', M') R v v'$   
 $\text{cmem } M' P M'$   
**using** *assms unfolding crel\_vs\_def* **by** *blast*  
**term** 0

**lemma** *consistentDP\_intro*:  
**assumes**  $\bigwedge \text{param}. \text{Transfer.Rel } (\text{crel\_vs } (=)) (\text{dp param}) (\text{dp}_T \text{ param})$



**shows** *consistentDP dp<sub>T</sub>*  
**using** *assms unfolding consistentDP\_def Rel\_def* **by** *blast*

**lemma** *crel\_vs\_return*:  
 $\llbracket \text{Transfer.Rel } R \ x \ y \rrbracket \implies \text{Transfer.Rel } (\text{crel\_vs } R) (\text{Wrap } x) (\text{State\_Monad.return } y)$   
**unfolding** *State\\_Monad.return\_def Wrap\_def Rel\_def* **by** (*fastforce intro: crel\_vs\_intro*)  
**term** 0

**lemma** *crel\_vs\_return\_ext*:  
 $\llbracket \text{Transfer.Rel } R \ x \ y \rrbracket \implies \text{Transfer.Rel } (\text{crel\_vs } R) \ x \ (\text{State\_Monad.return } y)$   
**by** (*fact crel\_vs\_return[unfolded Wrap\_def]*)  
**term** 0

**private lemma** *cmem\_upd*:  
 $\text{cmem } M' \text{ if } \text{cmem } M \ P \ M \ \text{State\_Monad.run\_state } (\text{update param } (dp \text{ param})) \ M = (v, M')$   
**using** *update\_correct[of M param dp param]* **that** **unfolding** *cmem\_def map\_le\_def* **by** *simp force*

**private lemma** *P\_upd*:  
 $P \ M' \text{ if } P \ M \ \text{State\_Monad.run\_state } (\text{update param } (dp \text{ param})) \ M = (v, M')$   
**by** (*meson lift\_p\_P that update\_inv*)

**private lemma** *crel\_vs\_get*:  
 $\llbracket \bigwedge M. \text{cmem } M \implies \text{crel\_vs } R \ v \ (sf \ M) \rrbracket \implies \text{crel\_vs } R \ v \ (\text{State\_Monad.get } s \gg sf)$   
**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*:  
 $\llbracket \text{crel\_vs } R \ v \ sf; \text{cmem } M; P \ M \rrbracket \implies \text{crel\_vs } R \ v \ (\text{State\_Monad.set } M \gg sf)$   
**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*:  
 $\llbracket \text{crel\_vs } (=) \ v \ s; \text{crel\_vs } R \ (f \ v) \ (sf \ v) \rrbracket \implies \text{crel\_vs } R \ (f \ v) \ (s \gg sf)$

**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 R0*  $\implies$  (*R0*  $\implies_T$  *R1*)  $\implies$  *crel\_vs R1*) ( $\lambda v f. f v$ ) ( $\gg$ )

**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* ( $\lambda v v'. \text{case } v' \text{ of None} \Rightarrow \text{True} \mid \text{Some } v' \Rightarrow v = v' \wedge v = dp \text{ param}$ ) (*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*:

$\llbracket is\_equality R; Transfer.Rel (crel\_vs R) (dp param) s \rrbracket$

$\implies Transfer.Rel (crel\_vs R) (dp param) (checkmem param s)$

**unfolding** *checkmem\_def Rel\_def is\_equality\_def*

**by** (*rule bind\_transfer[unfolding rel\_fun\_def, rule\_format, OF crel\_vs\_lookup]*)

(*auto 4 3 intro: crel\_vs\_lookup crel\_vs\_update crel\_vs\_return[unfolding Rel\_def Wrap\_def]* *crel\_vs\_bind\_eq*

*split: option.split\_asm*

)

**lemma** *crel\_vs\_checkmem\_tupled*:

**assumes** *v = dp param*

**shows**  $\llbracket is\_equality R; Transfer.Rel (crel\_vs R) v s \rrbracket$

$\implies Transfer.Rel (crel\_vs R) v (checkmem param s)$

**unfolding** *assms* **by** (*fact crel\_vs\_checkmem*)

```

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

lemma fun_app_lifted_transfer[transfer_rule]:
  ( $crel\_vs (R0 \text{ ===>}_T R1) \text{ ===> } crel\_vs R0 \text{ ===> } crel\_vs R1$ ) App (.)
  unfolding App_def fun_app_lifted_def by transfer_prover

lemma crel_vs_fun_app:
   $\llbracket Transfer.Rel (crel\_vs R0) x x_T; Transfer.Rel (crel\_vs (R0 \text{ ===>}_T R1))$ 
 $f f_T \rrbracket \implies Transfer.Rel (crel\_vs R1) (App f x) (f_T . x_T)$ 
  unfolding Rel_def using fun_app_lifted_transfer[THEN rel_funD, THEN
rel_funD] .

lemma if_T_transfer[transfer_rule]:
  ( $crel\_vs (=) \text{ ===> } crel\_vs R \text{ ===> } crel\_vs R \text{ ===> } crel\_vs R$ ) 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 ( $R1 \text{ ===> } S1$ )  $f g$  ( $R2 \text{ ===> } S2$ )  $g h$ 
  shows ( $R1 \text{ OO } R2 \text{ ===> } S1 \text{ OO } S2$ )  $f h$ 
  using assms by (auto intro!: rel_funI dest!: rel_funD)

lemma rel_fun_comp1:
  assumes ( $R1 \text{ ===> } S1$ )  $f g$  ( $R2 \text{ ===> } S2$ )  $g h$   $R' = R1 \text{ OO } R2$ 
  shows ( $R' \text{ ===> } S1 \text{ OO } S2$ )  $f h$ 
  using assms rel_fun_comp by metis

lemma rel_fun_comp2:

```

**assumes**  $(R1 \implies S1) f g (R2 \implies S2) g h S' = S1 \text{ OO } S2$   
**shows**  $(R1 \text{ OO } R2 \implies S') f h$   
**using** *assms rel\_fun\_comp by metis*

**lemma** *rel\_fun\_relcomp*:

$((R1 \implies S1) \text{ OO } (R2 \implies S2)) a b \implies ((R1 \text{ OO } R2) \implies (S1 \text{ OO } S2)) a b$

**unfolding** *OO\_def rel\_fun\_def* **by** *blast*

**lemma** *rel\_fun\_comp1'*:

**assumes**  $(R1 \implies S1) f g (R2 \implies S2) g h \wedge a b. R' a b \implies (R1 \text{ OO } R2) a b$

**shows**  $(R' \implies S1 \text{ OO } S2) f h$

**by** (*auto intro: assms rel\_fun\_mono[OF rel\_fun\_comp1]*)

**lemma** *rel\_fun\_comp2'*:

**assumes**  $(R1 \implies S1) f g (R2 \implies S2) g h \wedge a b. (S1 \text{ OO } S2) a b \implies S' a b$

**shows**  $(R1 \text{ OO } R2 \implies S') f h$

**by** (*auto intro: assms rel\_fun\_mono[OF rel\_fun\_comp1]*)

**end**

**end**

## 1.4 Heap Monad

**theory** *Heap\_Monad\_Ext*

**imports** *HOL-Imperative\_HOL.Imperative\_HOL*

**begin**

**definition** *fun\_app\_lifted* ::  $('a \Rightarrow 'b \text{ Heap}) \text{ Heap} \Rightarrow 'a \text{ Heap} \Rightarrow 'b \text{ Heap}$   
**where**

$\text{fun\_app\_lifted } f_T x_T \equiv \text{do } \{ f \leftarrow f_T; x \leftarrow x_T; f x \}$

**bundle** *heap\_monad\_syntax* **begin**

**notation** *fun\_app\_lifted* (**infixl** . 999)

**type\_synonym**  $('a, 'b) \text{ fun\_lifted} = 'a \Rightarrow 'b \text{ Heap} (\_ \implies \_ [3,2] 2)$

**type\_notation** *Heap* ( $[\_]$ )

**notation** *Heap\_Monad.return* ( $\langle \_ \rangle$ )

**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<sub>T</sub>* :: *bool Heap*  $\Rightarrow$  *'a Heap*  $\Rightarrow$  *'a Heap*  $\Rightarrow$  *'a Heap*

**where**

*if<sub>T</sub>* *b<sub>T</sub>* *x<sub>T</sub>* *y<sub>T</sub>*  $\equiv$  *do* { *b*  $\leftarrow$  *b<sub>T</sub>*; *if* *b* *then* *x<sub>T</sub>* *else* *y<sub>T</sub>* }

**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*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a Heap*  $\Rightarrow$  *bool* **where**

*lift\_p* *P* *f* =

$(\forall$  *heap*. *P* *heap*  $\longrightarrow$  (*case* *execute* *f* *heap* *of* *None*  $\Rightarrow$  *False* | *Some* (*\_*, *heap*)  $\Rightarrow$  *P* *heap*))

**context**

**fixes** *P* *f* *heap*

**assumes** *lift*: *lift\_p* *P* *f* **and** *P*: *P* *heap*

**begin**

**lemma** *execute\_cases*:

*case* *execute* *f* *heap* *of* *None*  $\Rightarrow$  *False* | *Some* (*\_*, *heap*)  $\Rightarrow$  *P* *heap*

**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* ≡  
 $\forall$  *heap. P heap*  $\longrightarrow$   
*(case State\_Monad.run\_state f heap of (v1, heap1) ⇒ case execute g heap of*  
*Some (v2, heap2) ⇒ R v1 v2 ∧ heap1 = heap2 ∧ P heap2 | None ⇒*

*False*)

**definition** *lookup'*  $k \equiv \text{State } (\lambda \text{ heap. the (execute (lookup } k) \text{ heap}))$

**definition** *update'*  $k \ v \equiv \text{State } (\lambda \text{ heap. the (execute (update } k \ v) \text{ heap}))$

**definition** *heap\_get* = *Heap\_Monad.Heap*  $(\lambda \text{ heap. Some (heap, heap))$

**definition** *checkmem* ::  $'k \Rightarrow 'v \text{ Heap} \Rightarrow 'v \text{ Heap}$  **where**

```
checkmem param calc  $\equiv$   
  Heap_Monad.bind (lookup param)  $(\lambda x.$   
    case x of  
      Some x  $\Rightarrow$  return x  
    | None  $\Rightarrow$  Heap_Monad.bind calc  $(\lambda x.$   
      Heap_Monad.bind (update param x)  $(\lambda \_.$   
        return x  
      )  
    )  
  )  
)
```

**definition** *checkmem'* ::  $'k \Rightarrow (\text{unit} \Rightarrow 'v \text{ Heap}) \Rightarrow 'v \text{ Heap}$  **where**

```
checkmem' param calc  $\equiv$   
  Heap_Monad.bind (lookup param)  $(\lambda x.$   
    case x of  
      Some x  $\Rightarrow$  return x  
    | None  $\Rightarrow$  Heap_Monad.bind (calc ())  $(\lambda x.$   
      Heap_Monad.bind (update param x)  $(\lambda \_.$   
        return x  
      )  
    )  
  )  
)
```

**lemma** *checkmem\_checkmem'*:

```
checkmem' param  $(\lambda \_.$  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
     $\bigwedge \text{heap } v \text{ heap}'. P \text{ heap} \implies \text{State\_Monad.run\_state } f \text{ heap} = (v, \text{heap}')$ 
     $\implies \exists v'. R v v' \wedge \text{execute } g \text{ heap} = \text{Some } (v', \text{heap}')$ 
     $\bigwedge \text{heap } v \text{ heap}'. P \text{ heap} \implies \text{State\_Monad.run\_state } f \text{ heap} = (v, \text{heap}')$ 
 $\implies P \text{ heap}'$ 
  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 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*:

$P\ m \implies \text{map\_of\_heap}\ (\text{snd}\ (\text{the}\ (\text{execute}\ (\text{lookup}\ k)\ m))) \subseteq_m$   
 $(\text{map\_of\_heap}\ m)$

**and** *update\_correct*:

$P\ m \implies \text{map\_of\_heap}\ (\text{snd}\ (\text{the}\ (\text{execute}\ (\text{update}\ k\ v)\ m))) \subseteq_m$   
 $(\text{map\_of\_heap}\ m)(k \mapsto v)$

**begin**

**lemma** *lookup'\_correct*:

$\text{state\_mem\_defs.map\_of}\ \text{lookup}'\ (\text{snd}\ (\text{State\_Monad.run\_state}\ (\text{lookup}'\ k)\ m)) \subseteq_m$   $\text{state\_mem\_defs.map\_of}\ \text{lookup}'\ m$  **if**  $P\ m$   
**using**  $\langle P\ m \rangle$  **unfolding**  $\text{state\_mem\_defs.map\_of\_def}\ \text{map\_le\_def}\ \text{lookup}'\_def$   
**by** *simp* (*metis* (*mono\_tags*, *lifting*) *domIff* *lookup\_correct* *map\_le\_def* *map\_of\_heap\_def*)

**lemma** *update'\_correct*:

$\text{state\_mem\_defs.map\_of}\ \text{lookup}'\ (\text{snd}\ (\text{State\_Monad.run\_state}\ (\text{update}'\ k\ v)\ m)) \subseteq_m$   $\text{state\_mem\_defs.map\_of}\ \text{lookup}'\ m(k \mapsto v)$   
**if**  $P\ m$   
**unfolding**  $\text{state\_mem\_defs.map\_of\_def}\ \text{map\_le\_def}\ \text{lookup}'\_def\ \text{update}'\_def$   
**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*:

$DP\_CRelVS.lift\_p\ P\ (\text{lookup}'\ k)$   
**unfolding**  $DP\_CRelVS.lift\_p\_def\ \text{lookup}'\_def$  **by** (*auto elim: lift\_p\_P'[OF lookup\_inv]*)

**lemma** *update'\_inv*:

$DP\_CRelVS.lift\_p\ P\ (\text{update}'\ k\ v)$   
**unfolding**  $DP\_CRelVS.lift\_p\_def\ \text{update}'\_def$  **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 lookups updates P
    and lookup: ((=) ==> rel_state (=)) lookups lookup
    and update: ((=) ==> (=) ==> rel_state (=)) updates update
  shows heap_correct P update lookup
proof –
  interpret mem: mem_correct lookups updates P
    by (rule correct)
  have [simp]: the (execute (lookup k) m) = run_state (lookups 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 (updates 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 (lookups 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  $\langle P \ m \rangle$ , of k]  $\langle P \ m \rangle$  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  $\Rightarrow$  bool and dp ::  $'k \Rightarrow 'v$  and lookup ::  $'k \Rightarrow 'v$  option
  Heap
  and lookup_st update update_st +
  assumes
    rel_state_lookup: rel_fun (=) (rel_state (=)) lookup_st lookup
    and
    rel_state_update: rel_fun (=) (rel_fun (=) (rel_state (=))) update_st
  update
begin

context
  includes lifting_syntax heap_monad_syntax
begin

definition crel_vs R v f  $\equiv$ 
   $\forall$  heap. P heap  $\wedge$  Q heap  $\wedge$  state_dp_consistency.cmem heap  $\longrightarrow$ 
  (case execute f heap of
    None  $\Rightarrow$  False |

```

Some ( $v', \text{heap}'$ )  $\Rightarrow$   $P \text{ heap}' \wedge Q \text{ heap}' \wedge R v v' \wedge \text{state\_dp\_consistency.cmem heap}'$   
 $)$

**abbreviation**  $\text{rel\_fun\_lifted} :: ('a \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('c ==H\Longrightarrow 'd) \Rightarrow \text{bool}$  (**infixr**  $===>_T$  55) **where**  
 $\text{rel\_fun\_lifted } R R' \equiv R ===> \text{crel\_vs } R'$

**definition**  $\text{consistentDP} :: ('k \Rightarrow 'v \text{ Heap}) \Rightarrow \text{bool}$  **where**  
 $\text{consistentDP} \equiv ((=) ===> \text{crel\_vs } (=)) \text{ dp}$

**lemma**  $\text{consistentDP\_intro}$ :  
**assumes**  $\bigwedge \text{param. Transfer.Rel } (\text{crel\_vs } (=)) (\text{dp param}) (\text{dp}_T \text{ param})$   
**shows**  $\text{consistentDP dp}_T$   
**using**  $\text{assms unfolding consistentDP\_def Rel\_def by blast}$

**lemma**  $\text{crel\_vs\_execute\_None}$ :  
**False if**  $\text{crel\_vs } R a b \text{ execute } b \text{ heap} = \text{None } P \text{ heap } Q \text{ heap state\_dp\_consistency.cmem heap}$   
**using**  $\text{that unfolding crel\_vs\_def by auto}$

**lemma**  $\text{crel\_vs\_execute\_Some}$ :  
**assumes**  $\text{crel\_vs } R a b P \text{ heap } Q \text{ heap state\_dp\_consistency.cmem heap}$   
**obtains**  $x \text{ heap}'$  **where**  $\text{execute } b \text{ heap} = \text{Some } (x, \text{heap}') P \text{ heap}' Q \text{ heap}'$   
**using**  $\text{assms unfolding crel\_vs\_def by (cases execute } b \text{ heap) auto}$

**lemma**  $\text{crel\_vs\_executeD}$ :  
**assumes**  $\text{crel\_vs } R a b P \text{ heap } Q \text{ heap state\_dp\_consistency.cmem heap}$   
**obtains**  $x \text{ heap}'$  **where**  
 $\text{execute } b \text{ heap} = \text{Some } (x, \text{heap}') P \text{ heap}' Q \text{ heap}' \text{ state\_dp\_consistency.cmem heap}' R a x$   
**using**  $\text{assms unfolding crel\_vs\_def by (cases execute } b \text{ heap) auto}$

**lemma**  $\text{crel\_vs\_success}$ :  
**assumes**  $\text{crel\_vs } R a b P \text{ heap } Q \text{ heap state\_dp\_consistency.cmem heap}$   
**shows**  $\text{success } b \text{ heap}$   
**using**  $\text{assms unfolding success\_def by (auto elim: crel\_vs\_executeD)}$

**lemma**  $\text{crel\_vsI}$ :  $\text{crel\_vs } R a b$  **if**  $(\text{state\_dp\_consistency.crel\_vs } R \text{ OO rel\_state } (=)) a b$   
**using**  $\text{that by (auto 4 3 elim: state\_dp\_consistency.crel\_vs\_elim rel\_state\_elim simp: crel\_vs\_def)}$

```

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_comp1 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:
  [[Transfer.Rel R x y]] ==> 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 R0 ==> (R0 ==> crel_vs R1) ==> crel_vs R1) (λv f. f v) (≫)
  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 => True | Some v' => v = v' ∧ 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* ( $\lambda x. x = v$ )) *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*:

$\llbracket \text{crel\_vs } (=) \text{ } v \text{ } s; \text{ crel\_vs } R \text{ } (f \text{ } v) \text{ } (sf \text{ } v) \rrbracket \implies \text{crel\_vs } R \text{ } (f \text{ } v) \text{ } (s \gg= sf)$   
**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**  $\llbracket \text{is\_equality } R; \text{ Transfer.Rel } ( \text{crel\_vs } R ) \text{ } v \text{ } s \rrbracket$   
 $\implies \text{Transfer.Rel } ( \text{crel\_vs } R ) \text{ } v \text{ } ( \text{checkmem param } s )$   
**unfolding** *assms* **by** (*fact crel\_vs\_checkmem*)

**lemma** *transfer\_fun\_app\_lifted*[*transfer\_rule*]:

(*crel\_vs* (*R0*  $\implies$  *crel\_vs R1*)  $\implies$  *crel\_vs R0*  $\implies$  *crel\_vs R1*)  
*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*:

$\llbracket \text{Transfer.Rel } ( \text{crel\_vs } R0 ) \text{ } x \text{ } x_T; \text{ Transfer.Rel } ( \text{crel\_vs } ( R0 \implies_T R1 ) )$   
 $f \text{ } f_T \rrbracket \implies \text{Transfer.Rel } ( \text{crel\_vs } R1 ) \text{ } ( \text{App } f \text{ } x ) \text{ } ( f_T \cdot x_T )$   
**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  $\Rightarrow$  '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 (dpT x) empty if consistentDP dpT
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 (dpT x) empty)) if consistentDP
dpT
using surjective_pairing memoization_correct(1)[OF that]
memoized_success[OF that, unfolded success_def]
by (cases execute (dpT x) empty; auto)

```

**lemma** cmem\_result:

```

state_dp_consistency.cmem (snd (the (Heap_Monad.execute (dpT x) empty)))
if consistentDP dpT
using surjective_pairing memoization_correct(2)[OF that(1)]
memoized_success[OF that, unfolded success_def]
by (cases execute (dpT 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  $\implies$  run_state f heap = (x, heap')
 $\implies$  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*  
 ( $\lambda k. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.return } (\text{Mapping.lookup } m \ k)\}$ )  
 ( $\lambda k \ v. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.set } (\text{Mapping.update } k \ v \ m)\}$ )  
 ( $\lambda \_ . \text{True}$ )  
**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*:  $\text{map\_of } \text{empty} \subseteq_m \text{Map.empty}$  **and** *P\_empty*:  
*P empty*

**lemma** (**in** *mem\_correct\_empty*) *dom\_empty*[*simp*]:  
 $\text{dom } (\text{map\_of } \text{empty}) = \{\}$   
**using** *empty\_correct* **by** (*auto dest: map\_le\_implies\_dom\_le*)

**lemma** *mem\_correct\_empty\_default*:  
*mem\_correct\_empty*  
 ( $\lambda k. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.return } (m \ k)\}$ )  
 ( $\lambda k \ v. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.set } (m(k \mapsto v))\}$ )  
 ( $\lambda \_ . \text{True}$ )  
*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*  
 ( $\lambda k. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.return } (\text{Mapping.lookup } m \ k)\}$ )  
 ( $\lambda k \ v. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.set } (\text{Mapping.update } k \ v \ m)\}$ )  
 ( $\lambda \_ . \text{True}$ )  
*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  $\Rightarrow$  'result
begin

```

```

sublocale dp_consistency_empty
   $\lambda$  k. do {(m::'param  $\rightarrow$  'result)  $\leftarrow$  State_Monad.get; State_Monad.return
    (m k)}
   $\lambda$  k v. do {m  $\leftarrow$  State_Monad.get; State_Monad.set (m(k $\mapsto$ v))}
   $\lambda$  (_::'param  $\rightarrow$  '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  $\Rightarrow$  'result
begin

  sublocale dp_consistency_empty
    ( $\lambda$  k. do {(m::('param,'result) mapping)  $\leftarrow$  State_Monad.get; State_Monad.return
      (Mapping.lookup m k)})
    ( $\lambda$  k v. do {m  $\leftarrow$  State_Monad.get; State_Monad.set (Mapping.update
      k v m)})
    ( $\lambda$  _::('param,'result) mapping. True) dp Mapping.empty
  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 ( $\lambda$  (log, m). case State_Monad.run_state (lookup k) m of
    (None, m)  $\Rightarrow$  (None, (("Missed", k) # log, m)) |
    (Some v, m)  $\Rightarrow$  (Some v, (("Found", k) # log, m))
  )

```

#### definition

```

update_trace k v =
  State ( $\lambda$  (log, m). case State_Monad.run_state (update k v) m of
    (_, m)  $\Rightarrow$  ((), (("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 ( $\lambda$  _. 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 ← 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
  mem1.map_of m (key2 k) else None)

```

**definition**

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

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



$P\ m \implies \text{State\_Monad.run\_state (update1 } k' v) m = (x, m') \implies$   
 $\text{fst (State\_Monad.run\_state get\_k2 } m') = \text{fst (State\_Monad.run\_state}$   
 $\text{get\_k2 } m)$

$P\ m \implies \text{State\_Monad.run\_state (update2 } k' v) m = (x, m') \implies$   
 $\text{fst (State\_Monad.run\_state get\_k1 } m') = \text{fst (State\_Monad.run\_state}$   
 $\text{get\_k1 } m)$

$P\ m \implies \text{State\_Monad.run\_state (update2 } k' v) m = (x, m') \implies$   
 $\text{fst (State\_Monad.run\_state get\_k2 } m') = \text{fst (State\_Monad.run\_state}$   
 $\text{get\_k2 } m)$

**assumes**

*lookup\_correct:*

$P\ m \implies \text{mem1.map\_of (snd (State\_Monad.run\_state (lookup1 } k')$   
 $m)) \subseteq_m (\text{mem1.map\_of } m)$

$P\ m \implies \text{mem2.map\_of (snd (State\_Monad.run\_state (lookup1 } k')$   
 $m)) \subseteq_m (\text{mem2.map\_of } m)$

$P\ m \implies \text{mem1.map\_of (snd (State\_Monad.run\_state (lookup2 } k')$   
 $m)) \subseteq_m (\text{mem1.map\_of } m)$

$P\ m \implies \text{mem2.map\_of (snd (State\_Monad.run\_state (lookup2 } k')$   
 $m)) \subseteq_m (\text{mem2.map\_of } m)$

**assumes**

*update\_correct:*

$P\ m \implies \text{mem1.map\_of (snd (State\_Monad.run\_state (update1 } k' v)$   
 $m)) \subseteq_m (\text{mem1.map\_of } m)(k' \mapsto v)$

$P\ m \implies \text{mem2.map\_of (snd (State\_Monad.run\_state (update2 } k' v)$   
 $m)) \subseteq_m (\text{mem2.map\_of } m)(k' \mapsto v)$

$P\ m \implies \text{mem2.map\_of (snd (State\_Monad.run\_state (update1 } k' v)$   
 $m)) \subseteq_m \text{mem2.map\_of } m$

$P\ m \implies \text{mem1.map\_of (snd (State\_Monad.run\_state (update2 } k' v)$   
 $m)) \subseteq_m \text{mem1.map\_of } m$

**begin**

**lemma** *map\_of\_le\_pair:*

*pair.map\_of*  $m \subseteq_m \text{map\_of1 } m ++ \text{map\_of2 } m$

**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 Let\_def*

*dest: get\_state split: prod.split\_asm if\_split\_asm*

)

**lemma** *pair\_le\_map\_of*:  
 $map\_of1\ m\ ++\ map\_of2\ m\ \subseteq_m\ pair.map\_of\ m$   
**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*:  
 $map\_of1\ m\ ++\ map\_of2\ m\ =\ pair.map\_of\ m$   
**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\_k1* *m*) = *fst* (*State\_Monad.run\_state* *get\_k2* *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]*:  
 $dom\ (map\_of1\ m) \cap dom\ (map\_of2\ m) = \{\}$  **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*:  
 $map\_of1\ heap' \subseteq_m\ Map.empty$  **if** *State\_Monad.run\_state* (*move12* *k1*)

$heap = (x, heap')$   $P$   $heap$   
**using**  $move12\_correct[OF\ that(2,1)]$  **unfolding**  $map\_of1\_def$  **by** ( $auto$   
 $simp: move12\_keys\ map\_le\_def$ )

**lemma**  $move12\_correct2$ :  
 $map\_of2\ heap' \subseteq_m\ map\_of1\ heap$  **if**  $State\_Monad.run\_state\ (move12\ k1)$   
 $heap = (x, heap')$   $P$   $heap$   
**using**  $move12\_correct(2)[OF\ that(2,1)]$  **that** **unfolding**  $map\_of1\_def$   
 $map\_of2\_def$   
**by** ( $auto\ simp: move12\_keys\ map\_le\_def$ )

**lemma**  $dom\_empty[simp]$ :  
 $dom\ (map\_of1\ heap') = \{\}$  **if**  $State\_Monad.run\_state\ (move12\ k1)$   $heap$   
 $= (x, heap')$   $P$   $heap$   
**using**  $move12\_correct1[OF\ that]$  **by** ( $auto\ dest: map\_le\_implies\_dom\_le$ )

**lemma**  $inv\_pair\_lookup1$ :  
 $inv\_pair\ m'$  **if**  $State\_Monad.run\_state\ (lookup1\ k)\ m = (v, m')$   $inv\_pair$   
 $m$   
**using**  $that\ lookup\_inv[of\ k]$   $inv\_pair\_P\_D[OF\ \langle inv\_pair\ m \rangle]$  **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**  $inv\_pair\_lookup2$ :  
 $inv\_pair\ m'$  **if**  $State\_Monad.run\_state\ (lookup2\ k)\ m = (v, m')$   $inv\_pair$   
 $m$   
**using**  $that\ lookup\_inv[of\ k]$   $inv\_pair\_P\_D[OF\ \langle inv\_pair\ m \rangle]$  **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**  $inv\_pair\_update1$ :  
 $inv\_pair\ m'$   
**if**  $State\_Monad.run\_state\ (update1\ (key2\ k)\ v)\ m = (v', m')$   $inv\_pair\ m$   
 $fst\ (State\_Monad.run\_state\ get\_k1\ m) = key1\ k$   
**using**  $that\ update\_inv[of\ key2\ k\ v]$   $inv\_pair\_P\_D[OF\ \langle inv\_pair\ m \rangle]$  **un-**  
**folding**  $inv\_pair\_def$   
**apply** ( $auto$   
 $simp: Let\_def\ update\_keys$ )

```

    dest: lift_p_P update_correct[of __ key2 k v, THEN map_le_implies_dom_le]
  )
  apply (frule update_correct[of __ key2 k v, THEN map_le_implies_dom_le];
auto 13 2; fail)
  apply (frule update_correct[of __ key2 k v, THEN map_le_implies_dom_le];
auto 13 2; fail)
  done

```

**lemma** *inv\_pair\_update2*:

```

  inv_pair m'
  if State_Monad.run_state (update2 (key2 k) v) m = (v', m') inv_pair m
fst (State_Monad.run_state get_k2 m) = key1 k
  using that update_inv[of key2 k v] inv_pair_P_D[OF <inv_pair m>] un-
folding inv_pair_def
  apply (auto
    simp: Let_def update_keys
    dest: lift_p_P update_correct[of __ key2 k v, THEN map_le_implies_dom_le]
  )
  apply (frule update_correct[of __ key2 k v, THEN map_le_implies_dom_le];
auto 13 2; fail)
  apply (frule update_correct[of __ key2 k v, THEN map_le_implies_dom_le];
auto 13 2; fail)
  done

```

**lemma** *inv\_pair\_move12*:

```

  inv_pair m'
  if State_Monad.run_state (move12 k) m = (v', m') inv_pair m fst (State_Monad.run_state
get_k1 m) ≠ k
  using that move12_inv[of k] inv_pair_P_D[OF <inv_pair m>] unfolding
inv_pair_def
  apply (auto
    simp: Let_def move12_keys
    dest: lift_p_P move12_correct[of __ k, THEN map_le_implies_dom_le]
  )
  apply (blast dest: move12_correct[of __ k, THEN map_le_implies_dom_le])
  done

```

**lemma** *mem\_correct\_pair*:

```

  mem_correct lookup_pair update_pair inv_pair
  if injective:  $\forall k k'. \text{key1 } k = \text{key1 } k' \wedge \text{key2 } k = \text{key2 } k' \longrightarrow k = k'$ 
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  $\subseteq_m$  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  $\subseteq_m$  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)  $\cap$  dom (map_of2 m) = {}
    using 3  $\langle$ map_of1 ?m  $\subseteq_m$  map_of1 m $\rangle$  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  $\subseteq_m$  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  $\subseteq_m$  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  $\subseteq_m$  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)  $\cap$  dom (map_of2 m) = {}
    using 3  $\langle$ map_of1 ?m  $\subseteq_m$  map_of1 m $\rangle$  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  $\subseteq_m$  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)  $\cap$  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  $\subseteq_m$  map_of1 m(k  $\mapsto$  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 ∉ 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  $\langle m' = m \rangle \langle m'' = m \rangle$ 
  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  $\langle m2 = m \rangle$ 
  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  $\langle m1 = m \rangle \langle m2 = m \rangle$ 
  using inv_pair_P_D[OF prems] by (auto elim: lift_p_P[OF
move12_inv])
  from 3(3)[unfolded  $\langle m2 = m \rangle$ ] have mem2.map_of ?m3  $\subseteq_m$  mem1.map_of
m
  by - (erule map_le_trans[OF update_correct(3)[OF  $\langle P m3 \rangle$ ] move12_correct(2)[OF
 $\langle P m \rangle$ ]])
  with 3 prems show ?thesis
  unfolding  $\langle m1 = m \rangle \langle m2 = m \rangle$  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)  $\cap$  dom (map_of1
m) = {}
  unfolding  $\langle m1 = m \rangle \langle m2 = m \rangle$ 
  by (smt inv_pair_P_D disjoint_iff_not_equal map_of1_def surjec-

```



```

tive_pairing domIff
  fst_conv inv_pair_move12 move12_keys update_keys
)
moreover from 3 have  $k \notin \text{dom} (\text{map\_of1 } m)$ 
  by (simp add: domIff map_of1_def)
moreover from 3 prems have inv_pair ?m3
  unfolding  $\langle m2 = m \rangle$ 
  by (metis inv_pair_move12 inv_pair_update1 move12_keys(1) fst_conv
surjective_pairing)
  ultimately show ?case
  unfolding  $\langle m1 = m \rangle \langle m2 = m \rangle$  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_upd2[symmetric])
  apply (auto intro: map_add_mono)
done
qed
qed

```

```

lemma emptyI:
  assumes inv_pair m mem1.map_of m  $\subseteq_m$  Map.empty mem2.map_of m
 $\subseteq_m$  Map.empty
  shows pair.map_of m  $\subseteq_m$  Map.empty
  using assms by (auto simp: map_of1_def map_of2_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  $\Rightarrow$  'k
begin

```

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

**definition**

```

lookup_pair k =
  State (λ mem.
    (
      case mem of Pair_Storage k1 k2 m1 m2 ⇒ 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) ⊆m (map_of m1 ++ map_of
m2)

```

**if** inv\_pair (Pair\_Storage k1 k2 m1 m2)

**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*:  
 $map\_of\ m1\ ++\ map\_of\ m2\ \subseteq_m\ pair.map\_of\ (Pair\_Storage\ k1\ k2\ m1\ m2)$   
**if** *inv\_pair (Pair\_Storage k1 k2 m1 m2)*  
**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*:  
 $map\_of\ m1\ ++\ map\_of\ m2\ =\ pair.map\_of\ (Pair\_Storage\ k1\ k2\ m1\ m2)$   
**if** *inv\_pair (Pair\_Storage k1 k2 m1 m2)*  
**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\_D1*:  
 $P\ m1$  **if** *inv\_pair (Pair\_Storage k1 k2 m1 m2)*  
**using** *that* **unfolding** *inv\_pair\_def* **by** *auto*

**lemma** *inv\_pair\_P\_D2*:  
 $P\ m2$  **if** *inv\_pair (Pair\_Storage k1 k2 m1 m2)*  
**using** *that* **unfolding** *inv\_pair\_def* **by** *auto*

**lemma** *inv\_pair\_domD[intro]*:  
 $dom\ (map\_of\ m1)\ \cap\ dom\ (map\_of\ m2)\ =\ \{\}$  **if** *inv\_pair (Pair\_Storage k1 k2 m1 m2)*  
**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'  $\subseteq_m$  map_of m1 map_of m2'  $\subseteq_m$  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')  $\cap$  dom
(map_of m2) = {}
      by (metis (no_types) ⟨map_of m1'  $\subseteq_m$  map_of m1⟩ disjoint_iff_not_equal
domIff map_le_def)
    have inv1: inv_pair (Pair_Storage (key k) k2 m1' m2) if k2  $\neq$  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 m1⟩, 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 inv2: inv_pair (Pair_Storage k1 (key k) m1 m2') if  $k2 = \text{key } k \text{ } k1$ 
 $\neq \text{key } k$ 
  using that  $\langle P \text{ } m1 \rangle \langle P \text{ } m2 \rangle$  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  $\langle P \text{ } m2 \rangle$ , 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) k2 m1' m2)  $\subseteq_m$  pair.map_of
(Pair_Storage (key k) k2 m1 m2)
    if  $k2 \neq \text{key } k \text{ } k1 = \text{key } k$ 
    using inv1 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 k1 (key k) m1 m2')  $\subseteq_m$  pair.map_of
(Pair_Storage k1 (key k) m1 m2)
    if  $k2 = \text{key } k \text{ } k1 \neq \text{key } k$ 
    using inv2 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  $\langle \text{inv\_pair } m \rangle$  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 m1 v1 m1' m2 v2 m2' m3 k1 k2
  assume assms:
    State_Monad.run_state (update k v) m1 =  $((), m1')$  State_Monad.run_state
(update k v) m2 =  $((), m2')$ 
    State_Monad.run_state (update k v) empty =  $((), m3)$ 
    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)
  from assms(3) P_empty update_inv[of k v] have P m3
  unfolding lift_p_def by auto

```

```

have [intro]: map_of m1'  $\subseteq_m$  map_of m1(k  $\mapsto$  v) map_of m2'  $\subseteq_m$ 
map_of m2(k  $\mapsto$  v)
  using update_correct[OF  $\langle P \ m1 \rangle$ , of k v] update_correct[OF  $\langle P \ m2 \rangle$ ,
of k v] assms by auto
have map_of m3  $\subseteq_m$  map_of empty(k  $\mapsto$  v)
  using assms(3) update_correct[OF P_empty, of k v] by auto
also have ...  $\subseteq_m$  map_of m2(k  $\mapsto$  v)
  using empty_correct by (auto elim: map_le_trans intro!: map_le_upd)
finally have map_of m3  $\subseteq_m$  map_of m2(k  $\mapsto$  v) .
have 1: dom (map_of m1)  $\cap$  dom (map_of m2(k  $\mapsto$  v)) = {} if k1  $\neq$ 
key k
  using assms(4) that by (force simp: inv_pair_def)
have 2: dom (map_of m3)  $\cap$  dom (map_of m1) = {} if k1  $\neq$  key k
  using  $\langle$ local.map_of m3  $\subseteq_m$  local.map_of empty(k  $\mapsto$  v) $\rangle$  assms(4)
that
  by (fastforce dest!: map_le_implies_dom_le simp: inv_pair_def)
have inv: inv_pair (Pair_Storage (key k) k1 m3 m1) if k2  $\neq$  key k k1
 $\neq$  key k
  using that  $\langle P \ m1 \rangle \langle P \ m2 \rangle \langle P \ m3 \rangle$  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) k1 m3 m1)  $\subseteq_m$  pair.map_of
(Pair_Storage k1 k2 m1 m2)(k  $\mapsto$  v)
  if k2  $\neq$  key k k1  $\neq$  key k
  using inv assms(4)  $\langle$ map_of m3  $\subseteq_m$  map_of m2(k  $\mapsto$  v) $\rangle$  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 inv1: inv_pair (Pair_Storage (key k) k2 m1' m2) if k2  $\neq$  key k k1
= key k
  using that  $\langle P \ m1 \rangle \langle P \ m2 \rangle$  unfolding inv_pair_def
  apply clarsimp
  apply safe
  subgoal for x' y
  using assms(1,4) update_correct[OF  $\langle P \ m1 \rangle$ , 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 inv2: inv_pair (Pair_Storage  $k_1$  (key  $k$ )  $m_1$   $m_2'$ ) if  $k_2 = \text{key } k \ k_1$ 
 $\neq \text{key } k$ 
    using that  $\langle P \ m_1 \rangle \langle P \ m_2 \rangle$  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  $\langle P \ m_2 \rangle$ , 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  $k$ )  $k_2$   $m_1'$   $m_2$ )  $\subseteq_m$ 
    pair.map_of (Pair_Storage (key  $k$ )  $k_2$   $m_1$   $m_2$ )( $k \mapsto v$ )
    if  $k_2 \neq \text{key } k \ k_1 = \text{key } k$ 
    using inv1[OF that] assms(4)  $\langle \text{inv\_pair } m \rangle$ 
    by (simp add: that map_of_eq_pair[symmetric])
    (subst map_add_upd2[symmetric]; force simp: inv_pair_def intro:
map_add_mono map_le_refl)
  have B:
    pair.map_of (Pair_Storage  $k_1$  (key  $k$ )  $m_1$   $m_2'$ )  $\subseteq_m$ 
    pair.map_of (Pair_Storage  $k_1$  (key  $k$ )  $m_1$   $m_2$ )( $k \mapsto v$ )
    if  $k_2 = \text{key } k \ k_1 \neq \text{key } k$ 
    using inv2[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  $\langle \text{inv\_pair } m \rangle$  show ?case
  by (auto split: pair_storage.split if_split prod.split simp: Let_def update_pair_def)
qed

end

end

end

end

```

## 2.3 Indexing

```
theory Indexing
  imports Main
begin
```

```
definition injective :: nat  $\Rightarrow$  ('k  $\Rightarrow$  nat)  $\Rightarrow$  bool where
  injective size to_index  $\equiv \forall a b.$ 
    to_index a = to_index b
   $\wedge$  to_index a < size
   $\wedge$  to_index b < size
   $\longrightarrow a = b$ 
for size to_index
```

```
lemma index_mono:
  fixes a b a0 b0 :: nat
  assumes a: a < a0 and b: b < b0
  shows a * b0 + b < a0 * b0
proof -
  have a * b0 + b < (Suc a) * b0
    using b by auto
  also have ...  $\leq$  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  $\wedge$  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 ...  $\leq$  c * b0
      using ac[THEN Suc_leI, THEN mult_le_mono1] .
    also have ...  $\leq$  c * b0 + d
      by auto
    finally have a * b0 + b  $\neq$  c * b0 + d
      by auto
  } note ac = this

  { assume a  $\neq$  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 =
  order0: ord less_eq0 less0 +
  order1: ord less_eq1 less1
  for less_eq0 less0 less_eq1 less1
begin

fun less :: 'a × 'b ⇒ 'a × 'b ⇒ bool where
  less (a,b) (c,d) ⇔ less0 a c ∧ less1 b d

fun less_eq :: 'a × 'b ⇒ 'a × 'b ⇒ bool where
  less_eq ab cd ⇔ less ab cd ∨ ab = cd

end

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

sublocale order less_eq less
proof qed fastforce+

end

```

```

locale option_order =
  order0: order less_eq0 less0
  for less_eq0 less0
begin

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

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

sublocale order less_eq_option less_option
  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  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a bound
 $\Rightarrow$  'a  $\Rightarrow$  bool where
  in_bound less_eq less bound x  $\equiv$  case bound of Bound l r  $\Rightarrow$  less_eq l x
 $\wedge$  less x r for less_eq less

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

locale index_locale = index_locale_def + idx_ord: order +
  assumes idx_valid: in_bound less_eq less bound x  $\implies$  idx bound x <
size bound
  and idx_inj :  $\llbracket$ in_bound less_eq less bound x; in_bound less_eq less
bound y; idx bound x = idx bound y $\rrbracket \implies x = y$ 

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

```

```

for less_eq0 less0 idx0 size0 less_eq1 less1 idx1 size1
begin

fun idx :: ('a × '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  $\Rightarrow$  'a option  $\Rightarrow$  nat where
  idx (Bound (Some l) (Some r)) (Some a) = idx0 (Bound l r) a
  | idx _ _ = undefined

end

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

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

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

```

```

sublocale index_locale ( $\leq$ ) ( $<$ ) idx size
proof qed (auto simp: in_bound_def split: bound.splits)

end

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

locale int_index_def = ord ( $\leq$ ) :: int  $\Rightarrow$  int  $\Rightarrow$  bool ( $<$ )
begin

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

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

sublocale index_locale ( $\leq$ ) ( $<$ ) idx size
proof qed (auto simp: in_bound_def split: bound.splits)

end

locale int_index = int_index_def + order ( $\leq$ ) :: int  $\Rightarrow$  int  $\Rightarrow$  bool ( $<$ )

class index =
  fixes less_eq less :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  and idx :: 'a bound  $\Rightarrow$  'a  $\Rightarrow$  nat
  and size :: 'a bound  $\Rightarrow$  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  $\equiv$  index_class.size bound for size

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

instance by (standard, simp, fact index_locale_axioms)

end

instantiation int :: index
begin

interpretation int_index ..
thm index_locale_axioms

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

lemmas size_int = size.simps

instance by (standard, simp, fact index_locale_axioms)
end

instantiation prod :: (index, index) index
begin

interpretation prod_index
  less_eq::'a  $\Rightarrow$  'a  $\Rightarrow$  bool less idx size
  less_eq::'b  $\Rightarrow$  'b  $\Rightarrow$  bool less idx size
  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 (l1, l2) (u1, u2)) = nat (u1 - l1) * nat (u2 -
  l2)
  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  $\equiv$  fst (the (execute c h))
abbreviation heap_of c h  $\equiv$  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)  $\Rightarrow$  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 λheap. 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
  λheap. 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 ⇒ 'v
begin

interpretation dp_consistency_heap_empty
  λheap. 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 ≫= 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**  $\bigwedge x h'.\ execute\ f\ h = Some\ (x, h') \implies success\ (g\ x)\ h'$   
**shows**  $success\ (f \ggg g)\ h$   
**by** (*rule successE[OF assms(1)] (auto elim: assms(2) intro: success\_bind\_executeI)*)

**definition**

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

**lemma** *alloc\_pair\_alloc*:  
 $Ref.get\ heap'\ r1 = a\ Ref.get\ heap'\ r2 = b$   
**if**  $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1, r2), heap')$   
**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\_pairD1*:  
 $r \neq r1 \wedge r \neq r2 \wedge Ref.present\ heap'\ r$   
**if**  $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1, r2), heap')$   $Ref.present\ heap\ r$   
**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\_pairD2*:  
 $r1 \neq r2 \wedge Ref.present\ heap'\ r2 \wedge Ref.present\ heap'\ r1$   
**if**  $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1, r2), heap')$   
**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\_pairD3*:  
 $Array.present\ heap'\ r$   
**if**  $execute\ (alloc\_pair\ a\ b)\ heap = Some\ ((r1, r2), heap')$   $Array.present\ heap\ r$

**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 ((r1, r2), 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 ((r1, r2), heap')*  
*= 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 ((r1, r2), 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 ((r1, r2), heap')*  
**using** *alloc\_pair\_array\_get[OF that(1) HOL.refl, of r] alloc\_pair\_array\_length[OF*  
*that(1), of r]*  
**by** (*cases (λh. i < Array.length h r) heap; simp add: execute\_simps Ar-*  
*ray.nth\_def)*

**lemma** *success\_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 k1 k2 m1 m2 ≡ do {*  
*(k\_ref1, k\_ref2) ← alloc\_pair k1 k2;*  
*(m\_ref1, m\_ref2) ← alloc\_pair m1 m2;*

```

    return (k_ref1, k_ref2, m_ref1, m_ref2)
}

```

**lemma** *init\_state\_inner\_alloc*:

```

assumes
  execute (init_state_inner k1 k2 m1 m2) heap = Some ((k_ref1, k_ref2,
m_ref1, m_ref2), heap')
shows
  Ref.get heap' k_ref1 = k1 Ref.get heap' k_ref2 = k2
  Ref.get heap' m_ref1 = m1 Ref.get heap' m_ref2 = m2
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\_pairD3*)

**lemma** *success\_init\_state\_inner[intro]:*  
*success (init\_state\_inner k1 k2 m1 m2) 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 k1 k2 m1 m2) heap = Some ((r1, r2), 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 k1 k2*  $\equiv$  *do* {  
*m1*  $\leftarrow$  *mem\_empty*;  
*m2*  $\leftarrow$  *mem\_empty*;  
*init\_state\_inner k1 k2 m1 m2*  
}

**lemma** *success\_init\_state[intro]:*  
*success (init\_state k1 k2) heap*  
**unfolding** *init\_state\_def* **by** (*auto intro: success\_intros success\_bind\_I*)

**definition**

*inv\_distinct k\_ref1 k\_ref2 m\_ref1 m\_ref2*  $\equiv$   
*m\_ref1*  $\neq$  *m\_ref2*  $\wedge$  *m\_ref1*  $\neq$  *k\_ref1*  $\wedge$  *m\_ref1*  $\neq$  *k\_ref2*  $\wedge$   
*m\_ref2*  $\neq$  *k\_ref1*  
 $\wedge$  *m\_ref2*  $\neq$  *k\_ref2*  $\wedge$  *k\_ref1*  $\neq$  *k\_ref2*

**lemma** *init\_state\_distinct:*

**assumes**  
*execute (init\_state k1 k2) heap = Some ((k\_ref1, k\_ref2, m\_ref1, m\_ref2), heap')*  
**shows**  
*inv\_distinct k\_ref1 k\_ref2 m\_ref1 m\_ref2*  
**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 k1 k2) 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\_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\_present2*:

**assumes**

*execute (init\_state k1 k2) heap = Some ((k\_ref1, k\_ref2, m\_ref1, m\_ref2), heap')*

**shows**

*Array.present heap' (Ref.get heap' m\_ref1) Array.present heap' (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'*

)

**lemma** *init\_state\_neq*:

**assumes**

*execute (init\_state k1 k2) heap = Some ((k\_ref1, k\_ref2, m\_ref1, m\_ref2), heap')*

**shows**

```

    Ref.get heap' m_ref1 == Ref.get heap' m_ref2
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_newfst_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 k1 k2) heap = Some ((k_ref1, k_ref2, m_ref1,
m_ref2), heap')
    shows
        Array.length heap' (Ref.get heap' m_ref1) = size
        Array.length heap' (Ref.get heap' m_ref2) = size
    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_ref1]
        )
    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 key1 :: 'k ⇒ ('k1 :: heap) and key2 :: 'k ⇒ 'k2
    and m_ref1 m_ref2 :: ('v :: heap) option array ref

```



**and**  $k\_ref1\ k\_ref2 :: ('k1 :: heap)\ ref$   
**begin**

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

**definition**

```
lookup_pair k = do {
  let k' = key1 k;
  k2 ← !k_ref2;
  if k' = k2 then
    do {
      m2 ← !m_ref2;
      mem_lookup m2 (key2 k)
    }
  else
    do {
      k1 ← !k_ref1;
      if k' = k1 then
        do {
          m1 ← !m_ref1;
          mem_lookup m1 (key2 k)
        }
      else
        return None
    }
}
```

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

**definition**

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

```

else do {
  do {
    k1 ← !k_ref1;
    m ← mem_empty;
    m1 ← !m_ref1;
    k_ref2 := k1;
    k_ref1 := k';
    m_ref2 := m1;
    m_ref1 := m
  }
;
m ← !m_ref1;
mem_update m (key2 k) v
}
}
}

```

**definition**

```

inv_pair_weak heap = (
  let
    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**

$inv\_pair\ heap \equiv inv\_pair\_weak\ heap \wedge inv\_distinct\ k\_ref1\ k\_ref2\ m\_ref1\ m\_ref2$

**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\_lengthD1*:

*Array.length heap (Ref.get heap m\_ref1) = size* **if** *inv\_pair\_weak heap*  
**using that unfolding** *inv\_pair\_weak\_def* **by** (*auto simp: Let\_def*)

**lemma** *inv\_pair\_lengthD2*:

*Array.length heap (Ref.get heap m\_ref2) = 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\_ref1) Array.present heap (Ref.get heap m\_ref2)*

**if** *inv\_pair\_weak heap*

**using that unfolding** *inv\_pair\_weak\_def* **by** (*auto simp: Let\_def*)

**lemma** *inv\_pair\_presentD2*:

*Ref.present heap m\_ref1 Ref.present heap m\_ref2*

*Ref.present heap k\_ref1 Ref.present heap k\_ref2*

**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\_ref1 !== Ref.get heap m\_ref2* **if** *inv\_pair\_weak heap*  
**using that unfolding** *inv\_pair\_weak\_def* **by** (*auto simp: Let\_def*)

**definition** *lookup1 k*  $\equiv$  *state\_of (do {m  $\leftarrow$  !m\_ref1; mem\_lookup m k})*

**definition** *lookup2 k*  $\equiv$  *state\_of (do {m  $\leftarrow$  !m\_ref2; mem\_lookup m k})*

**definition** *update1 k v*  $\equiv$  *state\_of (do {m  $\leftarrow$  !m\_ref1; mem\_update m k v})*

**definition** *update2 k v*  $\equiv$  *state\_of (do {m  $\leftarrow$  !m\_ref2; mem\_update m k v})*

**definition** *move12 k*  $\equiv$  *state\_of (do {*

*k1  $\leftarrow$  !k\_ref1;*

*m  $\leftarrow$  mem\_empty;*

*m1  $\leftarrow$  !m\_ref1;*

*k\_ref2 := k1;*

*k\_ref1 := k;*

*m\_ref2 := m1;*

*m\_ref1 := m*

})

**definition** *get\_k1*  $\equiv$  *state\_of* (!*k\_ref1*)

**definition** *get\_k2*  $\equiv$  *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*  $\neq$  *m\_ref2* *m\_ref1*  $\neq$  *k\_ref1* *m\_ref1*  $\neq$  *k\_ref2*

*m\_ref2*  $\neq$  *k\_ref1* *m\_ref2*  $\neq$  *k\_ref2*

*k\_ref1*  $\neq$  *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 solve1 =
  (frule inv_pair_lengthD1, frule inv_pair_lengthD2, frule inv_pair_not_eqD)?,
  auto split: if_split_asm dest: Array.noteq_sym

interpretation pair: pair_mem lookup1 lookup2 update1 update2 move12
get_k1 get_k2 inv_pair_weak
  supply [simp] =
    mem_empty_def state_mem_defs.map_of_def map_le_def
    move12_def update1_def update2_def lookup1_def lookup2_def get_k1_def
get_k2_def
    mem_update_def mem_lookup_def
    execute_bind_success[OF success_newI] execute_simps Let_def Ar-
ray.get_alloc length_def
    inv_pair_presentD inv_pair_presentD2
    Memory_Heap.lookup1_def Memory_Heap.lookup2_def Memory_Heap.mem_lookup_def
  apply standard
    apply (solve1; 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 (solve1; fail)+

subgoal
  using injective[unfolded injective_def] by - (solve1, subst (asm) nth_list_update_neq,

```

```

auto)
  subgoal
    using injective[unfolded injective_def] by - (solve1, subst (asm) nth_list_update_neq,
auto)
    apply (solve1; fail)+
  done

```

**lemmas** *mem\_correct\_pair = pair.mem\_correct\_pair*

**definition**

*mem\_lookup1 k = do {m ← !m\_ref1; mem\_lookup m k}*

**definition**

*mem\_lookup2 k = do {m ← !m\_ref2; mem\_lookup m k}*

**definition** *get\_k1' ≡ !k\_ref1*

**definition** *get\_k2' ≡ !k\_ref2*

**definition** *update1' k v ≡ do {m ← !m\_ref1; mem\_update m k v}*

**definition** *update2' k v ≡ do {m ← !m\_ref2; mem\_update m k v}*

**definition** *move12' k ≡ do {*

```

  k1 ← !k_ref1;
  m ← mem_empty;
  m1 ← !m_ref1;
  k_ref2 := k1;
  k_ref1 := k;
  m_ref2 := m1;
  m_ref1 := m
}
```

**interpretation** *heap\_mem\_defs inv\_pair\_weak lookup\_pair update\_pair*

.

**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)  $\forall$  heap. inv\_pair\_weak heap  $\longrightarrow$  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**  
 $\forall$  *heap. inv\_pair\_weak heap  $\longrightarrow$  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
   $\wedge$  heap. pair.inv_pair heap  $\implies$ 
  (case State_Monad.run_state m' heap of (_, heap)  $\implies$  inv_pair_weak
  heap  $\longrightarrow$  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:  $\forall k k'. \text{key1 } k = \text{key1 } k' \wedge \text{key2 } k = \text{key2 } k' \longrightarrow k = k'$ 
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** *k1 k2 heap heap'*  
**assumes** *init: execute (init\_state k1 k2) heap = Some ((k\_ref1, k\_ref2,*  
*m\_ref1, m\_ref2), heap^)*  
**begin**

**lemma** *init\_state\_empty1*:

*pair.mem1.map\_of\_heap' k = None*  
**using** *init*  
**unfolding** *pair.mem1.map\_of\_def lookup1\_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\_empty2*:

*pair.mem2.map\_of\_heap' k = None*  
**using** *init*  
**unfolding** *pair.mem2.map\_of\_def lookup2\_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\_k1: result\_of (!k\_ref1) heap' = k1*  
**and** *init\_state\_k2: result\_of (!k\_ref2) heap' = k2*  
**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 ≠ k2*  
**begin**

**lemma** *init\_state\_inv'*:  
*pair.inv\_pair heap'*  
**unfolding** *pair.inv\_pair\_def*  
**apply** (*auto simp: Let\_def*)  
**subgoal**  
**using** *init\_state\_empty1* **by** *simp*  
**subgoal**  
**using** *init\_state\_empty2* **by** *simp*  
**subgoal**  
**using** *neq init* **by** (*simp add: get\_k1\_def get\_k2\_def init\_state\_k1 init\_state\_k2*)  
**subgoal**  
**by** (*rule init\_state\_inv[OF init]*)  
**done**

**lemma** *init\_state\_empty*:  
*pair.pair.map\_of heap' ⊆<sub>m</sub> Map.empty*  
**using** *neq* **by** (*intro pair.emptyI init\_state\_inv' map\_emptyI init\_state\_empty1 init\_state\_empty2*)

**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  $dp :: 'k \Rightarrow 'v$ 
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

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 next :: 'a => 'a
begin

definition
  iter_state f ≡
  wfrec
  {(next x, x) | x. cnt x}
  (λ rec x. if cnt x then do {f x; rec (next 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 sizeof :: 'a ⇒ nat
  assumes terminating:
    finite {x. cnt x} ∀ x. cnt x → sizeof x < sizeof (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 = sizeof])
  moreover have finite ?S
    using [[simplproc 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
  next sizef empty
  if dp_consistency_empty lookup update P empty
  iterator cnt next 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 ≤ n ∧ y ≤ 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 ≤ n ∧ y ≤ m ∧ 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**  $((=) == => \text{crel\_vs } R) \ g \ f$

**shows**  $\text{crel\_vs } R \ (g \ x) \ (\text{do } \{ \text{iter\_heap } f \ y; \ f \ x \})$

**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*  $(\lambda \ x. \ \text{do } \{ \text{iter\_heap } f \ y; \ f \ x \})$

**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  $b = c$   $c \implies x = u$   $\neg c \implies y = v$ 
  shows  $\text{Heap\_Monad\_Ext.if}_T \langle b \rangle x y = \text{Heap\_Monad\_Ext.if}_T \langle c \rangle u v$ 
  unfolding Heap_Monad_Ext.ifT_def
  unfolding return_bind
  using if_cong[OF assms] .

lemma return_app_return_cong:
  assumes  $f x = g y$ 
  shows  $\langle f \rangle \cdot \langle x \rangle = \langle g \rangle \cdot \langle y \rangle$ 
  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]:
   $\text{crel\_vs } ((R1 \implies_T R2) \implies_T (R0 \implies_T R1) \implies_T (R0 \implies_T R2)) \text{ comp comp}_T$ 
  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]:
   $\text{crel\_vs } ((R0 \implies_T R1) \implies_T \text{list\_all2 } R0 \implies_T \text{list\_all2 } R1)$ 
  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) foldT_transfer[transfer_rule]:
  crel_vs ((R0  $\implies_T$  R1  $\implies_T$  R1)  $\implies_T$  list_all2 R0  $\implies_T$  R1
 $\implies_T$  R1) fold foldT
  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

```

```

context includes heap_monad_syntax begin

```

```

thm map_cong

```

```

lemma mapT_cong:

```

```

  assumes  $xs = ys \wedge x. x \in \text{set } ys \implies f x = g x$ 

```

```

  shows  $\text{mapT} . \langle f \rangle . \langle xs \rangle = \text{mapT} . \langle g \rangle . \langle ys \rangle$ 

```

```

  unfolding mapT_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  $xs = ys \wedge x. x \in \text{set } ys \implies f x = g x$ 

```

```

  shows  $\text{foldT} . \langle f \rangle . \langle xs \rangle = \text{foldT} . \langle g \rangle . \langle ys \rangle$ 

```

```

  unfolding foldT_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  $(\lambda \_ :: \text{unit}. x) = (\lambda \_. y)$ 

```

```

  using assms ..

```

```

lemma arg_cong4:

```

```

   $f a b c d = f a' b' c' d'$  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 heap_monad_syntax begin

named_theorems dp_match_rule

thm if_cong
lemma if_T_cong2:
  assumes Rel (=) b c c  $\implies$  Rel (crel_vs R) x x_T  $\neg$ c  $\implies$  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
     $\bigwedge x. x \in \text{set } ys \implies \text{Rel } (\text{crel\_vs } S) (f x) (f_T' x)$ 
  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** *fold<sub>T</sub>\_cong2*:

**assumes**

*is\_equality* *R*

*Rel* *R* *xs* *ys*

$\bigwedge x. x \in \text{set } ys \implies \text{Rel } (\text{crel\_vs } (S \implies \text{crel\_vs } S)) (f\ x) (f_T' x)$

**shows**

$\text{Rel } (\text{crel\_vs } (S \implies \text{crel\_vs } S)) (\text{fold } f\ xs) (\text{fold}_T . \langle f_T \rangle . \langle ys \rangle)$

**unfolding** *fold<sub>T</sub>\_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*  $\implies \text{Rel } R\ x\ x$

**unfolding** *is\_equality\_def Rel\_def* **by** *simp*

**lemma** *rel\_fun2*:

**assumes** *is\_equality* *R0*  $\bigwedge x. \text{Rel } R1\ (f\ x)\ (g\ x)$

**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** *Heap\_Monad\_Ext.return\_app\_return\_meta Wrap\_App\_Wrap*

.

**thm** *option.case\_cong[no\_vars]*

**lemma** *option\_case\_cong'*:

*Rel* (=) *option'* *option*  $\implies$

(*option* = *None*  $\implies \text{Rel } R\ f1\ g1$ )  $\implies$

( $\bigwedge x2. \text{option} = \text{Some } x2 \implies \text{Rel } R\ (f2\ x2)\ (g2\ x2)$ )  $\implies$

*Rel* *R* (*case* *option'* of *None*  $\Rightarrow f1$  | *Some* *x2*  $\Rightarrow f2\ x2$ )

(*case option of None*  $\Rightarrow$  *g1* | *Some x2*  $\Rightarrow$  *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'  $\Longrightarrow$*

( $\bigwedge x1\ x2. prod' = (x1, x2) \Longrightarrow Rel\ R\ (f\ x1\ x2)\ (g\ x1\ x2)$ )  $\Longrightarrow$

*Rel R (case prod of (x1, x2)  $\Rightarrow$  f x1 x2)*

(*case prod' of (x1, x2)  $\Rightarrow$  g x1 x2*)

**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\_cong2*  
*fold\_T\_cong2*  
*if\_T\_cong2*

**lemmas** [*dp\_match\_rule*] =  
*crel\_vs\_return*  
*crel\_vs\_fun\_app*  
*refl2*  
*rel\_fun2*

**end**

**end**

### 2.7.1 More Heap

**lemma** *execute\_heap\_ofD*:

*heap\_of c h = h' if execute c h = Some (v, h')*

**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  $P :: 'm \Rightarrow \text{heap} \Rightarrow \text{bool}$ 
and  $\text{lookup} :: 'm \Rightarrow 'k \Rightarrow 'v \text{ option Heap}$ 
and  $\text{update} :: 'm \Rightarrow 'k \Rightarrow 'v \Rightarrow \text{unit Heap}$ 
begin

definition  $\text{map\_of\_heap}'$  where
   $\text{map\_of\_heap}' m \text{ heap } k = \text{fst} (\text{the} (\text{execute} (\text{lookup } m k) \text{ heap}))$ 

end

locale  $\text{heap\_correct\_init\_inv} = \text{heap\_correct\_init\_defs} +$ 
assumes  $\text{lookup\_inv}: \bigwedge m. \text{lift\_p} (P m) (\text{lookup } m k)$ 
assumes  $\text{update\_inv}: \bigwedge m. \text{lift\_p} (P m) (\text{update } m k v)$ 

locale  $\text{heap\_correct\_init} =$ 
   $\text{heap\_correct\_init\_inv} +$ 
assumes  $\text{lookup\_correct}:$ 
   $\bigwedge a. P a m \implies \text{map\_of\_heap}' a (\text{snd} (\text{the} (\text{execute} (\text{lookup } a k) m)))$ 
 $\subseteq_m (\text{map\_of\_heap}' a m)$ 
and  $\text{update\_correct}:$ 
   $\bigwedge a. P a m \implies$ 
 $\text{map\_of\_heap}' a (\text{snd} (\text{the} (\text{execute} (\text{update } a k v) m))) \subseteq_m (\text{map\_of\_heap}'$ 
 $a m)(k \mapsto v)$ 
begin

end

locale  $\text{dp\_consistency\_heap\_init} = \text{heap\_correct\_init\_lookup}$  for  $\text{lookup}$ 
 $:: 'm \Rightarrow 'k \Rightarrow 'v \text{ option Heap} +$ 
fixes  $\text{dp} :: 'k \Rightarrow 'v$ 
fixes  $\text{init} :: 'm \text{ Heap}$ 
assumes  $\text{success}: \text{success } \text{init } \text{Heap.empty}$ 
assumes  $\text{empty\_correct}:$ 
   $\bigwedge \text{empty heap. execute } \text{init } \text{Heap.empty} = \text{Some} (\text{empty}, \text{heap}) \implies$ 
 $\text{map\_of\_heap}' \text{empty heap} \subseteq_m \text{Map.empty}$ 
and  $P\_empty: \bigwedge \text{empty heap. execute } \text{init } \text{Heap.empty} = \text{Some} (\text{empty},$ 
 $\text{heap}) \implies P \text{empty heap}$ 
begin

definition  $\text{init\_mem} = \text{result\_of } \text{init } \text{Heap.empty}$ 

sublocale  $\text{dp\_consistency\_heap}$ 
where  $P=P \text{init\_mem}$ 
and  $\text{lookup}=\text{lookup } \text{init\_mem}$ 

```

```

    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  $\gg$  ( $\lambda mem. dp_T mem x$ )) 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  $\Rightarrow$  'k  $\Rightarrow$  'v option Heap +
  fixes dp :: 'k  $\Rightarrow$  'v
  fixes init :: 'm Heap
  assumes success: success init Heap.empty
  assumes empty_correct:

```

$\bigwedge \text{empty heap. execute init Heap.empty} = \text{Some}(\text{empty}, \text{heap}) \implies$   
 $\text{map\_of\_heap}' \text{ empty heap} \subseteq_m \text{Map.empty}$   
**and**  $P\_empty: \bigwedge \text{empty heap. execute init Heap.empty} = \text{Some}(\text{empty},$   
 $\text{heap}) \implies P \text{ empty heap}$   
**begin**

**sublocale**  $dp\_consistency\_heap$   
**where**  $P=P \text{ init\_mem}$   
**and**  $lookup=lookup \text{ init\_mem}$   
**and**  $update=update \text{ init\_mem}$   
**apply**  $standard$   
**apply**  $(rule \text{ lookup\_inv}[of \text{ init\_mem}])$   
**apply**  $(rule \text{ update\_inv}[of \text{ init\_mem}])$   
**subgoal**  
**unfolding**  $heap\_mem\_defs.map\_of\_heap\_def$   
**by**  $(rule \text{ lookup\_correct}[of \text{ init\_mem}, \text{unfolded map\_of\_heap}'\_def])$   
**subgoal**  
**unfolding**  $heap\_mem\_defs.map\_of\_heap\_def$   
**by**  $(rule \text{ update\_correct}[of \text{ init\_mem}, \text{unfolded map\_of\_heap}'\_def])$   
**done**

**definition**  $init\_mem = result\_of \text{ init Heap.empty}$

**interpretation**  $consistent: dp\_consistency\_heap\_empty$   
**where**  $P=P \text{ init\_mem}$   
**and**  $lookup=lookup \text{ init\_mem}$   
**and**  $update=update \text{ init\_mem}$   
**and**  $empty= heap\_of \text{ init Heap.empty}$   
**apply**  $standard$   
**subgoal**  
**apply**  $(rule \text{ successE}[OF \text{ success}])$   
**apply**  $(frule \text{ empty\_correct})$   
**unfolding**  $heap\_mem\_defs.map\_of\_heap\_def \text{ init\_mem\_def map\_of\_heap}'\_def$   
**by**  $simp$   
**subgoal**  
**apply**  $(rule \text{ successE}[OF \text{ success}])$   
**apply**  $(frule \text{ P\_empty})$   
**unfolding**  $init\_mem\_def$   
**by**  $simp$   
**done**

**lemma**  $memoized\_empty:$

$dp \ x = result\_of \ (init \gg (\lambda mem. dp_T \ mem \ x)) \ \text{Heap.empty}$   
**if**  $consistentDP \ \text{init\_mem} \ (dp_T \ (result\_of \ \text{init} \ \text{Heap.empty}))$

**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)*  
 $\implies P$  empty heap  
**assumes** *consistent:*  
 $\bigwedge$  empty heap. execute init Heap.empty = Some (empty, heap)  
 $\implies dp\_consistency\_heap\_empty$  (*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*  $\gg$  ( $\lambda mem. dp_T$  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)
 $\implies$  P empty heap
assumes consistent:
   $\bigwedge$  empty heap. execute init Heap.empty = Some (empty, heap)
   $\implies$  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  $\gg$  ( $\lambda$ mem. dpT mem x)) Heap.empty
  if consistentDP (dpT (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 =  $\lambda$  mem heap. Array.length heap mem = size
    and lookup =  $\lambda$  mem. mem_lookup size to_index mem
    and update =  $\lambda$  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)  $\Rightarrow$  nat
    and dp :: 'k  $\Rightarrow$  'v::heap
  assumes injective: injective size to_index
begin

sublocale dp_consistency_new
  where P =  $\lambda$  mem heap. Array.length heap mem = size
    and lookup =  $\lambda$  mem. mem_lookup size to_index mem
    and update =  $\lambda$  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 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_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.mapT_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  $b = c \implies x = u \neg c \implies y = v$ 
```

```
  shows  $\text{State\_Monad\_Ext.if}_T \langle b \rangle x y = \text{State\_Monad\_Ext.if}_T \langle c \rangle u v$ 
```

```
  unfolding State_Monad_Ext.ifT_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 compT: comp monadifies (state) comp_def
```

```
lemma (in dp_consistency) compT_transfer[transfer_rule]:
```

```
  crel_vs (( $R1 \implies_T R2 \implies_T (R0 \implies_T R1) \implies_T (R0 \implies_T R2)$ )) comp compT
```

```
  apply memoize_combinator_init
```

```
  subgoal premises IH [transfer_rule] by memoize_unfold_defs transfer_prover
```

**done**

```
memoize_fun mapT: map monadifies (state) list.map  
lemma (in dp_consistency) mapT_transfer[transfer_rule]:  
  crel_vs ((R0 ==>T R1) ==>T list_all2 R0 ==>T list_all2 R1)  
map mapT  
  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 foldT: fold monadifies (state) fold.simps  
lemma (in dp_consistency) foldT_transfer[transfer_rule]:  
  crel_vs ((R0 ==>T R1 ==>T R1) ==>T list_all2 R0 ==>T R1  
==>T R1) fold foldT  
  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
```

**context includes state\_monad\_syntax begin**

```
thm map_cong  
lemma mapT_cong:  
  assumes xs = ys  $\wedge x. x \in \text{set } ys \implies f x = g x$   
  shows mapT . ⟨f⟩ . ⟨xs⟩ = mapT . ⟨g⟩ . ⟨ys⟩  
  unfolding mapT_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  $\wedge x. x \in \text{set } ys \implies f x = g x$   
  shows foldT . ⟨f⟩ . ⟨xs⟩ = foldT . ⟨g⟩ . ⟨ys⟩  
  unfolding foldT_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 state_monad_syntax begin

named_theorems dp_match_rule

thm if_cong
lemma ifT_cong2:
  assumes Rel (=) b c c  $\implies$  Rel (crel_vs R) x xT  $\neg c \implies$  Rel (crel_vs R)
y yT
  shows Rel (crel_vs R) (if (Wrap b) then x else y) (State_Monad_Ext.ifT
<c> xT yT)
  using assms unfolding State_Monad_Ext.ifT_def bind_left_identity
Rel_def Wrap_def
  by (auto split: if_split)

lemma mapT_cong2:
  assumes
    is_equality R
    Rel R xs ys
     $\bigwedge x. x \in \text{set } ys \implies \text{Rel } (\text{crel\_vs } S) (f x) (f_T' x)$ 
  shows Rel (crel_vs (list_all2 S)) (App (App map (Wrap f)) (Wrap xs))
(mapT . <fT'> . <ys>)
  unfolding mapT_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<sub>T</sub>\_cong2*:

```

assumes
  is_equality R
  Rel R xs ys
   $\bigwedge x. x \in \text{set } ys \implies \text{Rel } (\text{crel\_vs } (S \implies \text{crel\_vs } S)) (f x) (f_T' x)$ 
shows
  Rel (crel_vs (S  $\implies$  crel_vs S)) (fold f xs) (foldT .  $\langle f_T' \rangle$  .  $\langle ys \rangle$ )
unfolding foldT_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 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  $\implies$  Rel R x x
unfolding is_equality_def Rel_def by simp

```

**lemma** *rel\_fun2*:

```

assumes is_equality R0  $\bigwedge x. \text{Rel } R1 (f x) (g x)$ 
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'*:

$Rel (=) option' option \implies$   
 $(option = None \implies Rel R f1 g1) \implies$   
 $(\bigwedge x2. option = Some x2 \implies Rel R (f2 x2) (g2 x2)) \implies$   
 $Rel R (case option' of None \Rightarrow f1 \mid Some x2 \Rightarrow f2 x2)$   
 $(case option of None \Rightarrow g1 \mid Some x2 \Rightarrow g2 x2)$   
**unfolding**  $Rel\_def$  **by**  $(auto split: option.split)$

**thm**  $prod.case\_cong[no\_vars]$   
**lemma**  $prod\_case\_cong'$ : **fixes**  $prod prod'$  **shows**  
 $Rel (=) prod prod' \implies$   
 $(\bigwedge x1 x2. prod' = (x1, x2) \implies Rel R (f x1 x2) (g x1 x2)) \implies$   
 $Rel R (case prod of (x1, x2) \Rightarrow f x1 x2)$   
 $(case prod' of (x1, x2) \Rightarrow g x1 x2)$   
**unfolding**  $Rel\_def$  **by**  $(auto split: prod.splits)$

**thm**  $nat.case\_cong[no\_vars]$   
**lemma**  $nat\_case\_cong'$ : **fixes**  $nat nat'$  **shows**  
 $Rel (=) nat nat' \implies$   
 $(nat' = 0 \implies Rel R f1 g1) \implies$   
 $(\bigwedge x2. nat' = Suc x2 \implies Rel R (f2 x2) (g2 x2)) \implies$   
 $Rel R (case nat of 0 \Rightarrow f1 \mid Suc x2 \Rightarrow f2 x2) (case nat' of 0 \Rightarrow g1 \mid Suc x2$   
 $\Rightarrow g2 x2)$   
**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**  $[dp\_match\_rule] =$   
 $mapT\_cong2$   
 $foldT\_cong2$   
 $ifT\_cong2$

**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
  mapT_def
```

```
end
```

## 3 Examples

### 3.1 Misc

```
theory Example_Misc
  imports
    Main
    HOL-Library.Extended
    ../state_monad/State_Main
begin
```

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

```
lemma fold_min_commute:
  fold min xs (min a b) = min a (fold min xs b) for a :: 'a :: linorder
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_list012:
   $\llbracket P []; \bigwedge x. P [x]; \bigwedge x y zs. P (y # zs) \implies P (x # y # zs) \rrbracket \implies P xs$ 
by induction_schema (pat_completeness, lexicographic_order)
```

```
lemma min_list_Min:  $xs \neq [] \implies \text{min\_list } xs = \text{Min } (\text{set } xs)$ 
```

by (induction xs rule: induct\_list012)(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* ( $\langle\langle\_ \rangle\rangle$ )

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*  $\Rightarrow$  *unit* **where**

*writeln* = ( $\lambda$  s. ())

**code\_printing**

**constant** *writeln*  $\mapsto$  (SML) *writeln* \_

**definition** *trace* **where**

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

**lemma** *trace\_alt\_def*[simp]:

*trace* s x = ( $\lambda$  \_. x) (*writeln* s)

**unfolding** *trace\_def* **by** *simp*

**definition** (in *heap\_mem\_defs*) *checkmem\_trace* ::

(*k*  $\Rightarrow$  *String.literal*)  $\Rightarrow$  *k*  $\Rightarrow$  (*unit*  $\Rightarrow$  *'v Heap*)  $\Rightarrow$  *'v Heap*

**where**

*checkmem\_trace* *trace\_key* *param* *calc*  $\equiv$

*Heap\_Monad.bind* (*lookup* *param*) ( $\lambda$  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.thms1

  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_Natural
    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 ≤ 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  $\bigwedge x. x \in A \implies z \leq x$  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  $\neg \text{Sup (Fin -' A)} \leq z \bigwedge x. x \in A \implies x \leq \text{Fin } z \text{ Fin } x$ 
∈ 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  $\bigwedge x. x \in A \implies x \leq z$  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]*:  $-\infty = 0 \longleftrightarrow \text{False}$  **and** *Pinf\_eq\_zero[simp]*:  
 $\infty = 0 \longleftrightarrow \text{False}$

**unfolding** *zero\_extended\_def* **by** *auto*

**lemma** *Sup\_int*:

**fixes** *x* :: *int* **and** *X* :: *int set*

**assumes**  $X \neq \{\}$  *bdd\_above X*

**shows**  $\text{Sup } X \in X \wedge (\forall y \in X. y \leq \text{Sup } X)$

**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**  $z \leq \text{Max } (X \cap \{x..y\})$

**proof** *cases*

**assume**  $x \leq z$  **with**  $\langle z \in X \rangle \langle X \subseteq \{..y\} \rangle$  \*(1) **show** *?thesis*

**by** (*auto intro!: Max\_ge*)

**next**

**assume**  $\neg x \leq z$

**then have**  $z < x$  **by** *simp*

**also have**  $x \leq \text{Max } (X \cap \{x..y\})$

**using**  $\langle x \in X \rangle$  \*(1)  $\langle x \leq y \rangle$  **by** (*intro Max\_ge*) *auto*

**finally show** *?thesis* **by** *simp*

**qed** }

**note** *le = this*

**with** *Max\_in[OF \*]* **show**  $\text{ex}: \text{Max } (X \cap \{x..y\}) \in X \wedge (\forall z \in X. z \leq \text{Max } (X \cap \{x..y\}))$  **by** *auto*

**fix** *z* **assume** \*:  $z \in X \wedge (\forall y \in X. y \leq z)$

**with** *le* **have**  $z \leq \text{Max } (X \cap \{x..y\})$

**by** *auto*

**moreover have**  $\text{Max } (X \cap \{x..y\}) \leq z$

```

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

lemmas Sup_int_in = Sup_int[THEN conjunct1]

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

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

lemma finite_lists_length_le1:  $\text{finite } \{xs. \text{length } xs \leq i \wedge \text{set } xs \subseteq \{0..(n::\text{nat})\}\}$ 
  for  $i$ 
  by (auto intro: finite_subset[OF finite_lists_length_le[OF finite_atLeastAtMost]])

lemma finite_lists_length_le2:  $\text{finite } \{xs. \text{length } xs + 1 \leq i \wedge \text{set } xs \subseteq \{0..(n::\text{nat})\}\}$  for  $i$ 
  by (auto intro: finite_subset[OF finite_lists_length_le1[of i]])

lemmas [simp] =
  finite_setcompr_eq_image finite_lists_length_le2[simplified] finite_lists_length_le1

lemma get_return:
  run_state (State_Monad.bind State_Monad.get ( $\lambda m. \text{State_Monad.return } (f m)$ ))  $m = (f m, m)$ 
  by (simp add: State_Monad.bind_def State_Monad.get_def)

lemma list_pidgeonhole:
  assumes  $\text{set } xs \subseteq S$   $\text{card } S < \text{length } xs$  finite  $S$ 
  obtains  $as$   $bs$   $cs$  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\_cancell*:  
 $a + b \geq a \iff b \geq 0$  **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*  $\Rightarrow$  *nat*  $\Rightarrow$  *int extended*

**begin**

**context**

**fixes** *t* :: *nat* — Final node

**begin**

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

**fun** *weight* :: *nat list*  $\Rightarrow$  *int extended* **where**

*weight* [v] = 0

| *weight* (v # w # xs) = *W* v w + *weight* (w # xs)

**definition**

*OPT* *i* *v* = (

*Min* (

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

)

)

**lemma** *weight\_alt\_def'*:

*weight* (s # xs) + w = *snd* (*fold* ( $\lambda j$  (i, x). (j, *W* i j + x)) xs (s, w))

**by** (*induction* xs *arbitrary*: s w; *simp*; *smt* *add.commute* *add.left\_commute*)

**lemma** *weight\_alt\_def*:

*weight* (s # xs) = *snd* (*fold* ( $\lambda j$  (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  $\infty$ )

**unfolding** *OPT\_def* **by** *simp*

### 3.2.3 Functional Correctness

**lemma** *OPT\_cases*:

**obtains**  $(path) xs$  **where**  $OPT\ i\ v = weight\ (v\ \#\ xs\ @\ [t])\ length\ xs + 1$   
 $\leq i$  **set**  $xs \subseteq \{0..n\}$   
 $|$   $(sink)\ v = t\ OPT\ i\ v = 0$   
 $|$   $(unreachable)\ v \neq t\ OPT\ i\ v = \infty$   
**unfolding**  $OPT\_def$   
**using**  $Min\_in[of\ \{weight\ (v\ \#\ xs\ @\ [t])\ |xs.\ length\ xs + 1 \leq i \wedge set\ xs$   
 $\subseteq \{0..n\}\}$   
 $\cup\ \{if\ t = v\ then\ 0\ else\ \infty\}]$   
**by**  $(auto\ simp:\ finite\_lists\_length\_le2[simplified]\ split:\ if\_split\_asm)$

**lemma**  $OPT\_Suc$ :

$OPT\ (Suc\ i)\ v = min\ (OPT\ i\ v)\ (Min\ \{OPT\ i\ w + W\ v\ w\ | w.\ w \leq n\})$   
**(is**  $?lhs = ?rhs$   
**if**  $t \leq n$

**proof** –

**have**  $OPT\ i\ w + W\ v\ w \geq OPT\ (Suc\ i)\ v$  **if**  $w \leq n$  **for**  $w$   
**using**  $OPT\_cases[of\ i\ w]$   
**proof**  $cases$   
**case**  $(path\ xs)$   
**with**  $\langle w \leq n \rangle$  **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**  $Min\ \{OPT\ i\ w + W\ v\ w\ |w.\ w \leq n\} \geq OPT\ (Suc\ i)\ v$   
**by**  $(auto\ intro!:\ Min.boundedI)$   
**moreover** **have**  $OPT\ i\ v \geq OPT\ (Suc\ i)\ v$   
**unfolding**  $OPT\_def$  **by**  $(rule\ Min\_antimono)\ auto$   
**ultimately** **have**  $?lhs \leq ?rhs$   
**by**  $simp$

**from**  $OPT\_cases[of\ Suc\ i\ v]$  **have**  $?lhs \geq ?rhs$

**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 (λw. 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 bfm: bf with_memory dp_consistency_mapping monad-
ifies (state) bf.simps

```

Generated Definitions

```

context includes state_monad_syntax begin
thm bfm'.simps bfm_def
end

```

Correspondence Proof

```

memoize_correct
  by memoize_prover
print_theorems

```

**lemmas** [code] = *bf<sub>m</sub>.memoized\_correct*

**interpretation** *iterator*

$\lambda (x, y). x \leq n \wedge y \leq n$   
 $\lambda (x, y). \text{if } y < n \text{ then } (x, y + 1) \text{ else } (x + 1, 0)$   
 $\lambda (x, y). x * (n + 1) + y$   
**by** (*rule table\_iterator\_up*)

**interpretation** *bottom\_up: dp\_consistency\_iterator\_empty*

$\lambda (\_::(\text{nat} \times \text{nat}, \text{int extended}) \text{ mapping}). \text{True}$   
 $\lambda (x, y). \text{bf } x \ y$   
 $\lambda k. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.return } (\text{Mapping.lookup } m \ k :: \text{int extended option})\}$   
 $\lambda k \ v. \text{do } \{m \leftarrow \text{State\_Monad.get}; \text{State\_Monad.set } (\text{Mapping.update } k \ v \ m)\}$   
 $\lambda (x, y). x \leq n \wedge y \leq n$   
 $\lambda (x, y). \text{if } y < n \text{ then } (x, y + 1) \text{ else } (x + 1, 0)$   
 $\lambda (x, y). x * (n + 1) + y$   
*Mapping.empty ..*

**definition**

*iter\_bf* = *iter\_state* ( $\lambda (x, y). \text{bf}_m' \ x \ y$ )

**lemma** *iter\_bf\_unfold*[code]:

*iter\_bf* = ( $\lambda (i, j).$   
 (*if*  $i \leq n \wedge j \leq n$   
*then do* {  
    $\text{bf}_m' \ i \ j;$   
    $\text{iter\_bf } (\text{if } j < n \text{ then } (i, j + 1) \text{ else } (i + 1, 0))$   
 }  
*else State\_Monad.return* ()))

**unfolding** *iter\_bf\_def* **by** (*rule ext*) (*safe, clarsimp simp: iter\_state\_unfold*)

**lemmas** *bf\_memoized* = *bf<sub>m</sub>.memoized*[*OF bf<sub>m</sub>.crel*]

**lemmas** *bf\_bottom\_up* = *bottom\_up.memoized*[*OF bf<sub>m</sub>.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*  $\equiv$   
*do* {  
 $\_ \leftarrow \text{iter\_bf } (n, n);$   
 $xs \leftarrow \text{State\_Main.mapT}' (\lambda i. \text{bf}_m' \ n \ i) [0..<n+1];$   
 $ys \leftarrow \text{State\_Main.mapT}' (\lambda i. \text{bf}_m' \ (n + 1) \ i) [0..<n+1];$



```

    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);
    (⟨λxs. ⟨λys. State_Monad.return (if xs = ys then Some xs else None)⟩
    . (State_Main.map_T . ⟨λi. bf_m' (n + 1) i⟩ . ⟨[0..<n+1]⟩))
    . (State_Main.map_T . ⟨λi. 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)

```

```

( $\lambda (x, y). x \leq n \wedge y \leq n$ )
by (standard; simp add: mem_is_init injective_def)

memoize_fun bfh: bf
with_memory (default_proof) dp_consistency_heap_array_pair_iterator
where size = n + 1
  and key1 = fst :: nat × nat ⇒ nat and key2 = snd :: nat × nat ⇒ nat
  and k1 = 1 :: nat and k2 = 0 :: nat
  and to_index = id :: nat ⇒ nat
  and mem = mem
  and cnt =  $\lambda (x, y). x \leq n \wedge y \leq n$ 
  and nxt =  $\lambda (x :: nat, y). \text{if } y < n \text{ then } (x, y + 1) \text{ else } (x + 1, 0)$ 
  and sizef =  $\lambda (x, y). x * (n + 1) + y$ 
monadifies (heap) bf.simps

memoize_correct
  by memoize_prover

lemmas memoized_empty = bfh.memoized_empty[OF bfh.consistent_DP_iter_and_compute[OF
bfh.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  $\forall u \in \text{set } (a \# xs). \forall v \in \text{set } (xs @ [b]). f v + W u v \geq f u$ 
  shows  $\text{sum\_list } (\text{map } f (a \# xs)) \leq \text{sum\_list } (\text{map } f (xs @ [b])) + \text{weight } (a \# xs @ [b])$ 
  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  $\forall u \in \text{set } (a \# xs). \forall v \in \text{set } (a \# xs). f v + W u v \geq f u$ 
  shows  $\text{sum\_list } (\text{map } f (a \# xs @ [a])) \leq \text{sum\_list } (\text{map } f (a \# xs @ [a])) + \text{weight } (a \# xs @ [a])$ 
  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  $xs \equiv \text{weight } xs < \infty$ 

private lemma is_path2_remove_cycle:
  assumes is_path2 (as @ a # bs @ a # cs)
  shows is_path2 (as @ a # cs)
proof –
  have  $\text{weight } (as @ a \# bs @ a \# cs) =$ 
     $\text{weight } (as @ [a]) + \text{weight } (a \# bs @ [a]) + \text{weight } (a \# cs)$ 
  by (metis Bellman_Ford.weight_append append_Cons append_assoc)
  with assms have  $\text{weight } (as @ [a]) < \infty \text{ weight } (a \# cs) < \infty$ 
  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:
   $is\_path \ xs \longleftrightarrow is\_path2 \ (xs @ [t])$ 
  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 ≤ n set xs ⊆ {0..n} t ≤ n t ≠ i
  obtains xs where is_path (i # xs) i ≤ n set xs ⊆ {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 ≥ 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.premis ⟨i ≤ n⟩ ⟨t ≤ n⟩ have set (i # xs @ [t]) ⊆
{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) ⊆ {0..n}
  apply atomize_elim
  using *
proof (cases rule: path_eq_cycleE)
  case Nil_Nil
  with ⟨t ≠ i⟩ show ∃ ys. is_path (i # ys) ∧ length ys < length xs ∧
set (i # ys) ⊆ {0..n}
  by auto
  next
  case (Nil_Cons cs')
  then show ∃ 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

```

```

[]
  by - (rule exI[where x = cs'], simp)
next
  case (Cons_Nil as')
  then show  $\exists ys. is\_path (i \# ys) \wedge length\ ys < length\ xs \wedge set (i \# ys) \subseteq \{0..n\}$ 
    using  $\langle set (i \# xs @ [t]) \subseteq \{0..n\} \rangle \langle is\_path (i \# xs) \rangle$ 
    by - (rule exI[where x = as'], auto intro: is_path_remove_cycle2)
  next
  case (Cons_Cons as' cs')
  then show  $\exists ys. is\_path (i \# ys) \wedge length\ ys < length\ xs \wedge set (i \# ys) \subseteq \{0..n\}$ 
    using  $\langle set (i \# xs @ [t]) \subseteq \{0..n\} \rangle \langle is\_path (i \# xs) \rangle is\_path\_remove\_cycle[of\ i \# as']$ 
    by - (rule exI[where x = as' @ a # cs'], auto)
  qed
  then show ?thesis
    by (cases  $n \leq length\ ys$ ) (auto intro: that less)
  qed
qed

```

```

lemma reaches_non_inf_path:
  assumes reaches i  $i \leq n$   $t \leq n$ 
  shows  $OPT\ n\ i < \infty$ 
proof (cases  $t = i$ )
  case True
  with  $\langle i \leq n \rangle \langle t \leq n \rangle$  have  $OPT\ n\ i \leq 0$ 
    unfolding OPT_def
    by (auto intro: Min_le simp: finite_lists_length_le2[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)  $i \leq n$   $set\ xs \subseteq \{0..n\}$ 
    unfolding reaches_def by safe
  then obtain xs where xs: is_path (i # xs)  $i \leq n$   $set\ xs \subseteq \{0..n\}$   $length\ xs < n$ 
    using  $\langle t \neq i \rangle \langle t \leq n \rangle$  by (auto intro: is_path_shorten)
  then have weight (i # xs @ [t])  $< \infty$ 
    unfolding is_path_def by auto
  with xs(2-) show ?thesis
    unfolding OPT_def
    by (elim order.strict_trans1[rotated])
      (auto simp: setcompr_eq_image finite_lists_length_le2[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**  $W\_wellformed\ \langle t \leq n \rangle$   
**by** (*induction*  $xs$  *rule*: *induct\_list012*) (*auto* *intro*: *add\_gt\_minfI* *simp*:  
*zero\_extended\_def*)

**lemma** *OPT\_not\_minfI*:

$OPT\ n\ i > -\infty$  **if**  $i \leq n$

**proof** –

**have**  $OPT\ n\ i \in$   
 $\{weight\ (i\ \# \ xs\ @\ [t]) \mid xs.\ length\ xs + 1 \leq n \wedge set\ xs \subseteq \{0..n\}\} \cup \{if\ t$   
 $=\ i\ then\ 0\ else\ \infty\}$   
**unfolding** *OPT\_def*  
**by** (*rule* *Min\_in*) (*auto* *simp*: *setcompr\_eq\_image\_finite\_lists\_length\_le2[simplified]*)  
**with** *that*  $\langle t \leq n \rangle$  **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*  $\langle t \leq n \rangle$  **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**  $\langle a \leq n \rangle$  *weight\_append*[*of x # bs a cs @ [t]*] *cycle(2)*  $\langle xs = \_ \rangle$   
**by** – (*rule exI*[**where**  $x = bs @ [a] @ cs$ ], *auto intro: add\_lt\_infI*)  
**qed**  
**let**  $?S = \text{sum\_list } (\text{map } (OPT\ n) (a \# xs @ [a]))$   
**obtain**  $u\ v$  **where**  $u \leq n\ v \leq n\ OPT\ n\ v + W\ u\ v < OPT\ n\ u$   
**proof** (*atomize\_elim*, *rule ccontr*)  
**assume**  $\nexists u\ v. u \leq n \wedge v \leq n \wedge OPT\ n\ v + W\ u\ v < OPT\ n\ u$   
**then have**  $?S \leq ?S + \text{weight } (a \# xs @ [a])$   
**using** *cycle(1–3)* **by** (*subst fold\_sum\_aux*; *fastforce simp: subset\_eq*)  
**moreover have**  $?S > -\infty$   
**using** *cycle(1–4)* **by** (*intro sum\_list\_not\_minfI*, *auto intro!: OPT\_not\_minfI*)  
**moreover have**  $?S < \infty$   
**using** *reaches*  $\langle t \leq n \rangle$  *cycle(1,2)*  
**by** (*intro sum\_list\_not\_infI*) (*auto intro: reaches\_non\_inf\_path*  
 $\langle \text{reaches } a \rangle$  *simp: subset\_eq*)  
**ultimately have**  $\text{weight } (a \# xs @ [a]) \geq 0$   
**by** (*simp add: le\_add\_same\_cancel1*)  
**with**  $\langle \text{weight } \_ < 0 \rangle$  **show** *False*  
**by** *simp*  
**qed**  
**then show** *?thesis*  
**by** –  
(*rule exI*[**where**  $x = u$ ],  
*auto* 4 4 *intro: Min.coboundedI min.strict\_coboundedI2 elim: or-der.strict\_trans1*[*rotated*]  
*simp: OPT\_Suc*[*OF*  $\langle t \leq n \rangle$ ])  
**qed**

**corollary** *bf\_detects\_cycle*:  
**assumes** *has\_negative\_cycle*  
**shows**  $\exists i \leq n. bf\ (n + 1)\ i < bf\ n\ i$   
**using** *detects\_cycle*[*OF assms*] **unfolding** *bf\_correct*[*OF*  $\langle t \leq n \rangle$ ] .

**lemma** *shortest\_cases*:  
**assumes**  $v \leq n$   
**obtains** (*path*)  $xs$  **where**  $\text{shortest } v = \text{weight } (v \# xs @ [t])$  *set*  $xs \subseteq \{0..n\}$   
| (*sink*)  $v = t$   $\text{shortest } v = 0$   
| (*unreachable*)  $v \neq t$   $\text{shortest } v = \infty$   
| (*negative\_cycle*)  $\text{shortest } v = -\infty \forall x. \exists xs. \text{set } xs \subseteq \{0..n\} \wedge \text{weight } (v \# xs @ [t]) < Fin\ x$   
**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
  end
end

```

```

      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.prem(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  $\text{weight } (a \# bs @ [a]) \geq 0$ 
    by auto
  with * ys have  $\text{weight } (a \# cs' @ [t]) \leq \text{weight } (v \# ys @ [t])$ 
    using add_mono not_le by fastforce
  with Nil_Cons  $\langle \text{length } ys \geq n \rangle$  ys show ?thesis
    using IH[of cs'] by simp (meson le_less_trans order_trans)
  qed
next
  case (Cons_Nil as')
  with ys have *:  $\text{weight } (v \# ys @ [t]) = \text{weight } (v \# as' @ [t]) +$ 
 $\text{weight } (a \# bs @ [a])$ 
    using weight_append[of v # as' t bs @ [t]] by simp
  show ?thesis
  proof (cases  $\text{weight } (a \# bs @ [a]) < 0$ )
    case True
    with Cons_Nil  $\langle \text{set } ys \subseteq \_ \rangle$  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  $\text{weight } (a \# bs @ [a]) \geq 0$ 
      by auto
    with * ys(1) have  $\text{weight } (v \# as' @ [t]) \leq \text{weight } (v \# ys @ [t])$ 
      using add_left_mono by fastforce
    with Cons_Nil  $\langle \text{length } ys \geq n \rangle$   $\langle v \leq n \rangle$  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 *:
     $\text{weight } (v \# ys @ [t]) = \text{weight } (v \# as' @ a \# cs' @ [t]) + \text{weight}$ 
 $(a \# bs @ [a])$ 
    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  $\text{weight } (a \# bs @ [a]) < 0$ )
    case True
    with Cons_Cons  $\langle \text{set } ys \subseteq \_ \rangle$  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 ( $a \# bs @ [a]$ )  $\geq 0$ 
    by auto
  with * ys have weight ( $v \# as' @ a \# cs' @ [t]$ )  $\leq$  weight ( $v \# ys$ 
@ [t])
    using add_left_mono by fastforce
  with Cons_Cons  $\langle v \leq n \rangle$  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  $\langle set\ ys \subseteq \_ \rangle$  show ?thesis
    by auto
qed
qed

theorem shorter_than_OPT_n_has_negative_cycle:
  assumes shortest  $v < OPT\ n\ v\ v \leq n$ 
  shows has_negative_cycle
proof -
  from assms obtain ys where ys:
    weight ( $v \# ys @ [t]$ )  $< OPT\ n\ v$   $set\ ys \subseteq \{0..n\}$ 
  apply (cases rule: OPT_cases2[of  $v\ n$ ]; cases rule: shortest_cases[OF
 $\langle v \leq n \rangle$ ]; 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  $\langle t \leq n \rangle$  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  $\neg$  has_negative_cycle
      with False False ys  $\langle v \leq n \rangle$  obtain xs where
        weight ( $v \# xs @ [t]$ )  $\leq$  weight ( $v \# ys @ [t]$ )  $set\ xs \subseteq \{0..n\}$  length

```

$xs < n$   
**using** *less\_extended\_def* **by** (*fastforce elim!:* *simple\_paths*[of  $v$   $ys$ ])  
**then have**  $OPT\ n\ v \leq weight\ (v\ \#\ xs\ @\ [t])$   
**unfolding** *OPT\_def* **by** (*intro Min\_le*) *auto*  
**with**  $\langle \_ \leq weight\ (v\ \#\ ys\ @\ [t]) \rangle$   $\langle weight\ (v\ \#\ ys\ @\ [t]) < OPT\ n\ v \rangle$   
**show** *False*  
**by** *simp*  
**qed**  
**qed**  
**qed**

**corollary** *detects\_cycle\_has\_negative\_cycle:*  
**assumes**  $OPT\ (n + 1)\ v < OPT\ n\ v\ v \leq n$   
**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*  $\longleftrightarrow (\exists v \leq n. OPT\ (n + 1)\ v < OPT\ n\ v)$   
**using** *detects\_cycle\_has\_negative\_cycle* *detects\_cycle* **by** *blast*

**corollary** *bellman\_ford\_shortest\_paths:*  
**assumes**  $\neg has\_negative\_cycle$   
**shows**  $\forall v \leq n. bf\ n\ v = shortest\ v$   
**proof** –  
**have**  $OPT\ n\ v \leq shortest\ v$  **if**  $v \leq n$  **for**  $v$   
**using** *that* *assms shorter\_than\_OPT\_n\_has\_negative\_cycle*[of  $v$ ] **by** *force*  
**then show** *?thesis*  
**unfolding** *bf\_correct*[OF  $\langle t \leq n \rangle$ , *symmetric*]  
**by** (*safe, rule order.antisym*) (*auto elim: shortest\_le\_OPT*)  
**qed**

**lemma** *OPT\_mono:*  
 $OPT\ m\ v \leq OPT\ n\ v$  **if**  $\langle v \leq n \rangle$   $\langle n \leq m \rangle$   
**using** *that* **unfolding** *OPT\_def* **by** (*intro Min\_antimono*) *auto*

**corollary** *bf\_fix:*  
**assumes**  $\neg has\_negative\_cycle\ m \geq n$   
**shows**  $\forall v \leq n. bf\ m\ v = bf\ n\ v$   
**proof** (*intro allI impI*)  
**fix**  $v$  **assume**  $v \leq n$   
**from**  $\langle v \leq n \rangle$   $\langle n \leq m \rangle$  **have**  $shortest\ v \leq OPT\ m\ v$   
**by** (*simp add: shortest\_le\_OPT*)

**moreover from**  $\langle v \leq n \rangle \langle n \leq m \rangle$  **have**  $OPT\ m\ v \leq OPT\ n\ v$   
**by** (*rule*  $OPT\_mono$ )  
**moreover from**  $\langle v \leq n \rangle$  *assms* **have**  $OPT\ n\ v \leq shortest\ v$   
**using** *shorter\_than\_OPT\_n\_has\_negative\_cycle*[*of*  $v$ ] **by force**  
**ultimately show**  $bf\ m\ v = bf\ n\ v$   
**unfolding** *bf\_correct*[*OF*  $\langle t \leq n \rangle$ , *symmetric*] **by** *simp*  
**qed**

**lemma** *bellman\_ford\_correct'*:

*bf<sub>m</sub>.crel\_vs* (=) (*if has\_negative\_cycle* then *None* else *Some* (*map shortest*  $[0..<n+1]$ )) *bellman\_ford*

**proof** —

**include** *state\_monad\_syntax app\_syntax*

**let**  $?l = if\ has\_negative\_cycle\ then\ None\ else\ Some\ (map\ shortest\ [0..<n + 1])$

**let**  $?r = (\lambda xs.\ (\lambda ys.\ (if\ xs = ys\ then\ Some\ xs\ else\ None)))$

$\$ (map\ \$ \langle\langle bf\ (n + 1) \rangle\rangle\ \$ \langle\langle [0..<n + 1] \rangle\rangle) \$ (map\ \$ \langle\langle bf\ n \rangle\rangle\ \$ \langle\langle [0..<n + 1] \rangle\rangle)$

**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*  $\langle ?l = ?r \rangle$  — 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<sub>m</sub>* was defined.

— This is roughly what *memoize\_prover\_match\_step/Transform\_Tactic.step\_tac* does.

**apply** (*tactic*  $\langle Transform\_Tactic.solve\_relator\_tac\ context\ 1 \rangle$

| *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]:  
 $iter\_bf\_heap\ n\ w\ t\ mem = (\lambda\ (i,\ j).$   
 (if  $i \leq n \wedge j \leq n$   
 then do {  
    $bf\_h'\_impl\ n\ w\ t\ mem\ i\ j;$   
    $iter\_bf\_heap\ n\ w\ t\ mem\ (if\ j < n\ then\ (i,\ j + 1)\ else\ (i + 1,\ 0))$   
 }  
 else *Heap\_Monad.return* ()))  
**unfolding** *iter\_bf\_heap\_def* **by** (*rule ext*) (*safe, simp add: iter\_heap\_unfold*)

**definition**

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

**lemma** *bf\_impl\_correct*:

$bf\ n\ w\ t\ i\ j = result\_of\ (bf\_impl\ n\ w\ t\ i\ j)\ Heap.empty$   
**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**



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

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

**code\_reflect** *Test functions*  $test\_bf$

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

**ML**  $\langle Test.test\_bf\ () \rangle$

**lemma** *bottom\_up\_alt*[code]:

$bf\ n\ W\ t\ i\ j =$   
 $\quad fst\ (run\_state$   
 $\quad\quad (iter\_bf\ n\ W\ t\ (0, 0) \gg (\lambda\_.\ bf_m'\ n\ W\ t\ i\ j))$   
 $\quad\quad Mapping.empty)$   
**using**  $bf\_bottom\_up$  **by** *auto*

**definition**

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

— 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**

$fst\ (run\_state\ (bellman\_ford\ 3\ (graph\_of\ (IArray\ G_2\_list))\ 3)\ Mapping.empty) = None$   
 $fst\ (run\_state\ (bellman\_ford\ 4\ (graph\_of\ (IArray\ G_4\_list))\ 4)\ Mapping.empty) = None$

```

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  $\Rightarrow$  'v
begin

interpretation idx: bounded_index bound .

sublocale dp_consistency_heap
  where P= $\lambda$ heap. Array.length heap mem = idx.size
  and lookup=mem_lookup idx.size idx.checked_idx mem
  and update=mem_update idx.size idx.checked_idx mem
  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  $\subseteq_m$  Map.empty
  and len: Array.length empty mem = idx.size
begin

interpretation consistent: dp_consistency_heap_empty
  where P= $\lambda$ heap. Array.length heap mem = idx.size
  and lookup=mem_lookup idx.size idx.checked_idx mem
  and update=mem_update idx.size idx.checked_idx mem
  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)  $\gg=$ 
( $\lambda$ mem. dpT mem x)) Heap.empty
  if consistentDP (dpT 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)  $\gg=$  ( $\lambda$ mem. dpT
mem x)) Heap.empty
  if consistentDP (dpT 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**  $w :: nat \Rightarrow nat$

**begin**

**context**

**fixes**  $v :: nat \Rightarrow nat$

**begin**

**fun**  $knapsack :: nat \Rightarrow nat \Rightarrow nat$  **where**

$knapsack\ 0\ W = 0$  |

$knapsack\ (Suc\ i)\ W = (if\ W < w\ (Suc\ i)$

$then\ knapsack\ i\ W$

$else\ max\ (knapsack\ i\ W)\ (v\ (Suc\ i) + knapsack\ i\ (W - w\ (Suc\ i))))$

**no\_notation**  $fun\_app\_lifted$  (**infixl** . 999)

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

**definition**

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

**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**  $finite\ S\ S \neq \{\}$

**proof** –

**have**  $?A \leq ?B$

**using**  $that$  **by** ( $force\ intro: Min.boundedI$ )

**moreover** **have**  $?B \leq ?A$

**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  $OPT\_in: OPT n W \in \{\sum i \in S. v i \mid S. S \subseteq \{1..n\} \wedge (\sum i \in S. w i) \leq W\}$  for  $n W$ 
    unfolding  $OPT\_def$  by - (rule  $Max\_in$ ; force)
  from  $OPT\_in[of Suc i W]$  obtain  $S$  where  $S$ :
     $S \subseteq \{1..Suc i\}$   $sum w S \leq W$  and [ $simp$ ]:  $OPT (Suc i) W = sum v S$ 
    by auto

  have  $OPT i W \leq OPT (Suc i) W$ 
    unfolding  $OPT\_def$  by (force intro:  $Max\_mono$ )
  moreover have  $v (Suc i) + OPT i (W - w (Suc i)) \leq OPT (Suc i) W$ 
if  $w (Suc i) \leq W$ 
  proof -
    have *:
       $v (Suc i) + sum v S = sum v (S \cup \{Suc i\}) \wedge (S \cup \{Suc i\}) \subseteq \{1..Suc i\}$ 
     $\wedge sum w (S \cup \{Suc i\}) \leq W$  if  $S \subseteq \{1..i\}$   $sum w S \leq W - w (Suc i)$ 
    for  $S$ 
      using that  $\langle w (Suc i) \leq W \rangle$ 
      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  $\geq$  ?rhs
    by auto

  from  $S$  have *:  $sum v S \leq OPT i W$  if  $Suc i \notin S$ 
    using that unfolding  $OPT\_def$  by (auto simp:  $atLeastAtMostSuc\_conv$ 
intro!:  $Max\_ge$ )

  have  $sum v S \leq OPT i W$  if  $W < w (Suc i)$ 
  proof (rule *, rule ccontr, simp)
    assume  $Suc i \in S$ 
    then have  $sum w S \geq w (Suc i)$ 
      using  $S(1)$  by (subst  $sum.remove$ ) (auto intro:  $finite\_subset[OF\_finite\_atLeastAtMost]$ )
    with  $\langle W < \_ \rangle \langle \_ \leq W \rangle$  show False
      by simp

```

```

qed
moreover have
   $OPT (Suc i) W \leq \max(v (Suc i) + OPT i (W - w (Suc i))) (OPT i W)$ 
if  $w (Suc i) \leq W$ 
proof (cases  $Suc i \in S$ )
  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  $OPT i (W - w (Suc i)) \geq sum v (S - \{Suc i\})$ 
      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  $\leq$  ?rhs
  by auto
with  $\langle ?lhs \geq ?rhs \rangle$  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 knapsackm: knapsack with_memory dp_consistency_mapping
monadifies (state) knapsack.simps

```

Generated Definitions

```

context includes state_monad_syntax begin
thm knapsackm'.simps knapsackm_def
end

```

Correspondence Proof

```

memoize_correct
  by memoize_prover
print_theorems

```

**lemmas** [code] = *knapsack<sub>m</sub>.memoized\_correct*

### 3.3.4 Imperative Memoization

**context** fixes

*mem* :: *nat option array*

**and** *n W* :: *nat*

**begin**

**memoize\_fun** *knapsack<sub>T</sub>*: *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<sub>T</sub>' .simps knapsack<sub>T</sub>\_def*

**end**

**memoize\_correct**

**by** *memoize\_prover*

**lemmas** *memoized\_empty* = *knapsack<sub>T</sub>.memoized\_empty*

**end**

Adding Memory Initialization

**context**

**includes** *heap\_monad\_syntax*

**notes** [*simp del*] = *knapsack<sub>T</sub>' .simps*

**begin**

**definition**

*knapsack<sub>h</sub>* ≡ λ *i j*. *Heap\_Monad.bind* (*mem\_empty* (*i \* j*)) (λ *mem*.  
*knapsack<sub>T</sub>' mem i j i j*)

**lemmas** *memoized\_empty'* = *memoized\_empty*[

*of mem n W λ m. λ(i,j). knapsack<sub>T</sub>' m n W i j,*

*OF knapsack<sub>T</sub>.crel[of mem n W], of (n, W) for mem n W*

]

**lemma** *knapsack\_heap*:

*knapsack n W* = *result\_of* (*knapsack<sub>h</sub> n W*) *Heap.empty*

**unfolding** *knapsack<sub>h</sub>\_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 sum: su with_memory dp_consistency_mapping monad-  
ifies (state) su.simps
```

Generated Definitions

```
context includes state_monad_syntax begin
thm sum'.simps sum_def
end
```

Correspondence Proof

```
memoize_correct
  by memoize_prover
print_theorems
lemmas [code] = sum.memoized_correct
```

**end**

### 3.3.6 Regression Test

```
definition
  knapsack_test = (knapsackh (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_Natural
    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_fin1:
  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 ∨ B ⇒ (A ⇒ P) ⇒ (¬ A ⇒ B ⇒ P) ⇒ 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) |

```

$valid' xs \implies n \geq m \implies valid' (replicate\ n\ R\ @\ B\ \# xs)$

**lemma** *valid\_valid'*:

$valid\ l \implies valid'\ l$

**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*:

$valid'\ l \implies valid\ l$

**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*:

$(\forall i < \text{length}\ l.\ l\ !\ i = R) \iff (\exists n.\ l = replicate\ n\ R)$

**by** *auto (metis (full\_types) in\_set\_conv\_nth replicate\_eqI)*

**lemma** *replicate\_iff2*:

$(\forall i < n.\ l\ !\ i = R) \iff (\exists l'. l = replicate\ n\ R\ @\ l')\ \text{if}\ n < \text{length}\ l$

**using** *that* **by** (*auto simp: list\_eq\_iff\_nth\_eq nth\_append intro: exI[where*  
*x = drop\ n\ l]*)

**lemma** *replicate\_Cons\_eq*:

$replicate\ n\ x = y\ \# ys \iff (\exists n'. n = \text{Suc}\ n' \wedge x = y \wedge replicate\ n'\ x = ys)$

**by** (*cases\ n*) *auto*

Main Case Analysis on @term valid

**lemma** *valid\_split*:

$valid\ l \iff$

$l = [] \vee$

$(l!0 = B \wedge valid\ (tl\ l)) \vee$

$\text{length}\ l \geq m \wedge (\forall i < \text{length}\ l.\ l\ !\ i = R) \vee$

$(\exists j < \text{length}\ l.\ j \geq m \wedge (\forall i < j.\ l\ !\ i = R) \wedge l\ !\ j = B \wedge valid\ (drop\ (j + 1)\ l))$

**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!: disjE1)
    apply (fastforce intro: valid'.intros simp: neq_Nil_conv)
    apply (subst (asm) replicate_iff2; 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:
   $\{l. l = [] \wedge \text{valid } l\} = \{\}\}
  by (auto intro: valid.intros)$ 
```

```

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

```

```

lemma F_base_aux:  $\{l. \text{length } l = n \wedge \text{valid } l\} = \{\text{replicate } n\ B\}$  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*:  $\{l.\ \text{length } l = m \wedge \text{valid } l\} = \{\text{replicate } m\ R, \text{ replicate } m\ B\}$   
**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*  $\{l.\ \text{length } l = n \wedge \text{valid } l\}$  (**is** *finite* *?S*)  
**proof** –  
**have**  $?S \subseteq \text{lists } \{R, B\} \cap \{l.\ \text{length } l = n\}$   
**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*:  
 $\{l.\ \text{length } l = n \wedge \text{valid } l\} \neq \{\}$  (**is** *?S*  $\neq \{\}$ )  
**using** *valid\_line\_just\_B*[*of n*] **by** *force*

**lemma** *replicate\_unequal\_aux*:  
 $\text{replicate } x\ R\ @\ B\ \# l \neq \text{replicate } y\ R\ @\ B\ \# l'$  (**is** *?l*  $\neq$  *?r*) **if**  $\langle x < y \rangle$   
**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)  $\longleftrightarrow$  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*:  $F\ n = F\ (n-1) + 1 + (\sum_{i=m..<n}. F\ (n-i-1))$  **if**  $\langle n > m \rangle$   
 $m > 0$

**proof** –

**have**  $\{l. \text{length } l = n \wedge \text{valid } l\}$   
 $= \{l. \text{length } l = n \wedge \text{valid } (tl\ l) \wedge !l0=B\}$   
 $\cup \{l. \text{length } l = n \wedge$   
 $(\exists i. i < n \wedge i \geq m \wedge (\forall k < i. !k = R) \wedge !i = B \wedge \text{valid}$   
 $(\text{drop } (i + 1) l))\}$   
 $\cup \{l. \text{length } l = n \wedge (\forall i < n. !i=R)\}$   
**(is**  $?A = ?B \cup ?D \cup ?C$   
**using**  $\langle n > m \rangle$  **by** (*subst valid\_split*) *auto*

**let**  $?B1 = ((\#) B) ' \{l. \text{length } l = n - \text{Suc } 0 \wedge \text{valid } l\}$

**from**  $\langle n > m \rangle$  **have**  $?B = ?B1$

**apply** *safe*

**subgoal for**  $l$

**by** (*cases l*) (*auto simp: valid\_prepend\_B\_iff*)

**by** *auto*

**have** 1:  $\text{card } ?B1 = F\ (n-1)$

**unfolding** *F\_def* **by** (*auto intro: card\_image*)

**have**  $?C = \{\text{replicate } n\ R\}$

**by** (*auto simp: nth\_equalityI*)

**have** 2:  $\text{card } \{\text{replicate } n\ R\} = 1$

**by** *auto*

**let**  $?D1 = (\bigcup i \in \{m..<n\}. (\lambda l. \text{replicate } i\ R @ B \# l) ' \{l. \text{length } l = n$   
 $- i - 1 \wedge \text{valid } l\})$

**have**  $?D =$

$(\bigcup i \in \{m..<n\}. \{l. \text{length } l = n \wedge (\forall k < i. !k = R) \wedge !i = B \wedge$   
 $\text{valid } (\text{drop } (i + 1) l)\})$

```

    by auto
  have {l. length l = n ∧ (∀ k < i. !k = R) ∧ !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 = ?D1
    unfolding ⟨?D = _⟩ by auto

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

  have *:
    (λl. replicate x R @ B # l) ‘ {l. length l = n - Suc x ∧ valid l} ∩
    (λl. 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 ?D1 = (∑ 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  $\langle m > 0 \rangle$  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**  $\forall n \in \{0..<168\}. \textit{lcount} 50 n < 1000000$

**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 le0 le\_add1 length\_append*)

**lemma** *append\_iff\_take\_drop1*:  $u \neq [] \implies v \neq [] \implies$

$w = u@v \longleftrightarrow (\exists k \in \{1..length\ w - 1\}. u = take\ k\ w \wedge v = drop\ k\ w)$

**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) =

```

$[A. k <- [1..<n], B <- \text{cyk2 } w (i,k), C <- \text{cyk2 } w (i+k,n-k), (A, NN B' C') <- P, B' = B, C' = C]$

**lemma** *set\_aux*:  $(\bigcup_{xb \in \text{set } P}. \{A. (A, NN B C) = xb\}) = \{A. (A, NN B C) \in \text{set } P\}$   
**by** *auto*

**lemma** *cyk2\_eq\_cyk*:  $i+n \leq \text{length } w \implies \text{set}(\text{cyk2 } w (i,n)) = \text{set}(\text{cyk } (\text{take } n (\text{drop } i w)))$

**proof** (*induction w (i,n) arbitrary: i n rule: cyk2.induct*)

**case 1 show** *?case* **by** (*simp*)

**next**

**case 2 show** *?case* **using** *2.prem*s

**by** (*auto simp: hd\_drop\_conv\_nth take\_Suc*)

**next**

**case** (*3 w i m*)

**show** *?case* **using** *3.prem*s

**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* =  $(S \in \text{set}(\text{cyk2 } w (0, \text{length } w)))$

**theorem** *CYK\_correct*:  $\text{CYK } S w = \text{yield } S w$

**by** (*simp add: CYK\_def cyk2\_eq\_cyk cyk\_correct*)

### 3.5.5 CYK With Index Function

**context**

**fixes**  $w :: \text{nat} \Rightarrow 't$

**begin**

**fun** *cyk\_ix* ::  $\text{nat} * \text{nat} \Rightarrow 'n \text{ list}$  **where**

*cyk\_ix*  $(i,0) = []$  |

*cyk\_ix*  $(i,\text{Suc } 0) = [A . (A, T a) <- P, a = w i]$  |

*cyk\_ix*  $(i,n) =$

$[A. k <- [1..<n], B <- \text{cyk\_ix } (i,k), C <- \text{cyk\_ix } (i+k,n-k), (A, NN B' C') <- P, B' = B, C' = C]$

### 3.5.6 Correctness Proof

**lemma** *cyk\_ix\_simp2*:  $\text{set}(\text{cyk\_ix } (i,\text{Suc}(\text{Suc } n))) =$

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

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

**abbreviation** (*input*) *slice f i j*  $\equiv$  *map f [i..<j]*

**lemma** *slice\_append\_iff\_take\_drop1*:  $u \neq [] \implies v \neq [] \implies$   
 $slice\ w\ i\ j = u @ v \iff (\exists k. 1 \leq k \wedge k \leq j-i-1 \wedge slice\ w\ i\ (i+k) = u$   
 $\wedge slice\ w\ (i+k)\ j = v)$   
**by**(*subst* *append\_iff\_take\_drop1*) (*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\_eq1*)  
**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\_simp2 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\_drop1 yield\_not\_Nil*  
*cong*: *conj\_cong*)  
**also have**  $... = \{N. yield\ N\ ?w\}$  **using** *yield\_eq2*[*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**  $CYK\_ix\ S\ w\ n = (S \in set(cyk\_ix\ w\ (0,n)))$

**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**  $cyk\_list\ w = cyk\_ix\ (\lambda i. w\ !\ i)\ (0,length\ w)$

**definition**

$CYK\_ix\_impl\ S\ w\ n = do\ \{R \leftarrow cyk\_ix\_impl\ w\ n\ 0\ n; return\ (S \in set\ R)\}$

**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*  $map\ (\lambda w. cyk2\ P\ w\ (0,length\ w))\ ["baaba", "baba"]$ )

**value**

(*let*  $P = [(0::int, NN\ 1\ 2), (0, NN\ 2\ 3),$   
 $(1, NN\ 2\ 1), (1, T\ (CHR\ "a")),$   
 $(2, NN\ 3\ 3), (2, T\ (CHR\ "b")),$   
 $(3, NN\ 1\ 2), (3, T\ (CHR\ "a"))]$   
*in*  $map\ (cyk\_list\ P)\ ["baaba", "baba"]$ )

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

**value**



```

(let P = [(0::int, NN 1 2), (0, NN 2 3),
         (1, NN 2 1), (1, T (CHR "a")),
         (2, NN 3 3), (2, T (CHR "b")),
         (3, NN 1 2), (3, T (CHR "a"))]
  in map (cyk_ia P) ["baaba", "baba"])

```

### 3.5.10 Imperative Test Case

**definition** *cyk\_ia' P w = (let a = IArray w in cyk\_ix\_impl P (λi. 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\_Dist0*

**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 |

```

$\text{argmin } f (a\#as) = (\text{let } m = \text{argmin } f \text{ as in if } f a \leq f m \text{ then } a \text{ else } m)$

```
fun argmin2 :: ('a  $\Rightarrow$  'b::order)  $\Rightarrow$  'a list  $\Rightarrow$  '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  $\leq$  m then
    (a, fa) else (am,m))
```

### 3.6.2 Edit Distance

```
datatype 'a ed = Copy | Repl 'a | Ins 'a | Del
```

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

**abbreviation** cost **where**

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

### 3.6.3 Minimum Edit Sequence

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

**lemma** min\_eds "vintner" "writers" =

[Ins CHR "w", Repl CHR "r", Copy, Del, Copy, Del, Copy, Copy, Ins  
CHR "s"]

**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 = []  $\longleftrightarrow$  xs = []  $\wedge$  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  $\implies$  length ys  $\leq$  cost es
apply(induction es []::'a list arbitrary: ys rule: edit.induct)
apply auto
done

lemma if_edit_eq_Nil: edit es xs = []  $\implies$  length xs  $\leq$  cost es
by (induction es xs rule: edit.induct) auto

lemma min_eds_minimal: edit es xs = ys  $\implies$  cost(min_eds xs ys)  $\leq$  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.prem1 by (auto simp: min_eds_same)
  next
    case [simp]: (Cons e es')
    show ?thesis
    proof (cases e)
      case Copy
      thus ?thesis using 4.prem1 4.IH(3)[of es'] by simp
    next
      case (Repl a)
      thus ?thesis using 4.prem1 4.IH(3)[of es']
      using [simp_depth_limit=1] by simp

```

```

next
  case (Ins a)
  thus ?thesis using 4.prem1 4.IH(1)[of es]
    using [[simp_depth_limit=1]] by auto
next
  case Del
  thus ?thesis using 4.prem2 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_ed: min_ed xs ys = cost(min_ed 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_eds2 xs ys) = cost(min_eds xs ys)
proof(induction xs ys rule: min_eds2.induct)
  case (4 x xs y ys) thus ?case
    apply (auto split!: if_split)
    apply (metis (mono_tags, lifting) Suc_eq_plus1 Suc_leI cost_Copy_Del
cost_Copy_Ins le_imp_less_Suc le_neq_implies_less not_less)
    apply (metis Suc_eq_plus1 cost_Copy_Del le_antisym)
    by (metis Suc_eq_plus1 cost_Copy_Ins le_antisym)
qed simp_all

lemma min_eds2 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*

$\lambda (x, y). x \leq m \wedge y \leq n \wedge x > 0$

$\lambda (x, y). \text{if } y > 0 \text{ then } (x, y - 1) \text{ else } (x - 1, n)$

$\lambda (x, y). (m - x) * (n + 1) + (n - y)$

**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*

$(\lambda (x, y). \text{if } y > 0 \text{ then } (x, y - 1) \text{ else } (x - 1, n))$

$(\lambda (x, y). (m - x) * (n + 1) + (n - y))$

$(\lambda (x, y). x \leq m \wedge y \leq n \wedge x > 0)$

**by** (*standard*; *simp add: mem\_is\_init injective\_def*)

**memoize\_fun** *min\_ed\_ix<sub>h</sub>*: *min\_ed\_ix*

**with\_memory** (**default\_proof**) *dp\_consistency\_heap\_array\_pair\_iterator*

**where** *size* = *n* + 1

**and** *key1=fst* :: *nat* × *nat* ⇒ *nat* **and** *key2=snd* :: *nat* × *nat* ⇒ *nat*

**and** *k1=m* :: *nat* **and** *k2=m* + 1 :: *nat*

**and** *to\_index* = *id* :: *nat* ⇒ *nat*

**and** *mem* = *mem*

**and** *cnt* =  $\lambda (x, y). x \leq m \wedge y \leq n \wedge x > 0$

**and** *nxt* =  $\lambda (x::nat, y). \text{if } y > 0 \text{ then } (x, y - 1) \text{ else } (x - 1, n)$

**and** *sizef* =  $\lambda (x, y). (m - x) * (n + 1) + (n - y)$

**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*  $\equiv$  *map xs [i..<j]*

**lemma** *min\_ed\_Nil1*: *min\_ed [] ys* = *length ys*

by (*induction ys*) *auto*

**lemma** *min\_ed\_Nil2*: *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\_Nil1 min\_ed\_Nil2*  
*Suc\_diff\_Suc*)

**done**

Functional Test Cases

**definition** *min\_ed\_list xs ys* = *min\_ed\_ix* ( $\lambda i. xs!i$ ) ( $\lambda i. ys!i$ ) (*length xs*)  
(*length ys*) (0,0)

**lemma** *min\_ed\_list "madagascar" "bananas"* = 6

by *eval*

**definition** *min\_ed\_ia xs ys* = (*let a = IArray xs; b = IArray ys*  
  *in min\_ed\_ix* ( $\lambda i. a!!i$ ) ( $\lambda i. b!!i$ ) (*length xs*) (*length ys*) (0,0))

**lemma** *min\_ed\_ia "madagascar" "bananas"* = 6

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*( $\lambda(xs, ys, m, n, mem, i, j). (m - i) + (n - j)$ )) *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*  $\times$  *nat ref*  $\times$  *nat option array ref*  $\times$  *nat option array ref*

**assumes** *mem\_is\_init*: *mem* = *result\_of* (*init\_state* (*n* + 1) *m* (*m* + 1)) *Heap.empty*

**shows** *min\_ed\_ix\_h'\_impl xs ys m n mem* = *min\_ed\_ix\_h' xs ys m n mem*

**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*

( $\lambda (x, y). x \leq m \wedge y \leq n \wedge x > 0$ )

( $\lambda (x, y). \text{if } y > 0 \text{ then } (x, y - 1) \text{ else } (x - 1, n)$ )

(*min\_ed\_ix\_h'\_impl xs ys m n mem*)

**lemma** *iter\_min\_ed\_ix\_unfold*[*code*]:

*iter\_min\_ed\_ix xs ys m n mem* = ( $\lambda (i, j).$

(*if* *i* > 0  $\wedge$  *i*  $\leq$  *m*  $\wedge$  *j*  $\leq$  *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'_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'_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 (λi. a!!i) (λi. 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_recl upto_split1 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_wplT: 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_wplT'.simps min_wplT_def
end

memoize_correct
  by memoize_prover

lemmas memoized_empty = min_wplT.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]  $\implies$  min_wpl i j  $\leq$  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]:  $\neg$  i > j
    have kk_ij: k  $\in$  set [i..j] using 2
      by (metis set_inorder tree.set_intros(2))

    let ?M = (( $\lambda$ k. 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  $\leq$  ?w
    proof (rule Min_le)
      show finite ?M by simp
      show ?w  $\in$  ?M using kk_ij by auto
    qed

    have inorder <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 ...  $\leq$  ?w by (rule aux_min)
    also have ...  $\leq$  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 = ((λk. 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**  $p :: int \Rightarrow nat$   
**begin**

— The *weighted path path length* (or *cost*) of a tree.

**fun**  $cost :: int \Rightarrow nat$  **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**  $W\_rec$ :

$W\ i\ j = (if\ j \geq i\ then\ W\ i\ (j - 1) + p\ j\ else\ 0)$

**unfolding**  $W\_def$  **by** (*simp add: atLeastAtMost\_split\_insert*)

— The weight function correctly implements costs.

**lemma**  $inorder\_wpl\_correct$ :

$inorder\ t = [i..j] \implies wpl\ W\ i\ j\ t = cost\ t$

**proof** (*induction t arbitrary: i j*)

**case**  $Leaf$

**show**  $?case$

**by** *simp*

**next**

**case**  $(Node\ l\ k\ r)$

**from**  $\langle inorder\ \langle l, k, r \rangle = [i..j] \rangle$  **have**  $*$ :  $i \leq k \leq j$

**by** — (*simp, metis atLeastAtMost\_iff\_in\_set\_conv\_decomp\_set\_upto*) +

**moreover from**  $\langle i \leq k \rangle \langle k \leq j \rangle$  **have**  $inorder\ l = [i..k-1]$   $inorder\ r = [k+1..j]$

**using**  $\langle inorder\ \langle l, k, r \rangle = [i..j] \rangle$  [*symmetric*] **by** (*simp add: upto\_split3 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*  $t = [i..j] \implies \text{cost } (\text{opt\_bst } W \ i \ j) \leq \text{cost } t$   
**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*  $t = [i..j] \implies \text{min\_wpl } W \ i \ j \leq \text{cost } t$   
**using** *inorder\_wpl\_correct* *min\_wpl\_minimal* **by** *metis*

**lemma** *min\_wpl\_tree*:  
 $\text{cost } (\text{opt\_bst } W \ i \ j) = \text{min\_wpl } W \ i \ j$   
**using** *wpl\_opt\_bst* *opt\_bst\_correct* *inorder\_wpl\_correct* **by** *metis*

**An alternative view of costs.** **fun** *depth* :: 'a  $\Rightarrow$  'a tree  $\Rightarrow$  nat extended  
**where**

*depth*  $x$  Leaf =  $\infty$   
| *depth*  $x$  (Node  $l$   $k$   $r$ ) = (if  $x = k$  then 1 else min (*depth*  $x$   $l$ ) (*depth*  $x$   $r$ ) + 1)

**fun** *the\_fin* **where**  
*the\_fin* (Fin  $x$ ) =  $x$  | *the\_fin* \_ = *undefined*

**definition** *cost'* :: int tree  $\Rightarrow$  nat **where**  
*cost'*  $t = \text{sum } (\lambda x. \text{the\_fin } (\text{depth } x \ t) * p \ x) (\text{set\_tree } t)$

**lemma** [*simp*]:  
*the\_fin* 1 = 1  
**by** (*simp* *add: one\_extended\_def*)

**lemma** *set\_tree\_depth*:  
**assumes**  $x \notin \text{set\_tree } t$   
**shows** *depth*  $x$   $t = \infty$   
**using** *assms* **by** (*induction*  $t$ ) *auto*

**lemma** *depth\_inf\_iff*:  
 $\text{depth } x \ t = \infty \iff x \notin \text{set\_tree } t$   
**apply** (*induction*  $t$ )  
**apply** (*auto* *simp: one\_extended\_def*)  
**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
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 =  $-\infty$   $\longleftrightarrow$  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  $x \in \text{set\_tree } t$ 
  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⟩)  $\implies$ 
  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  $h = \lambda x. (\text{the\_fin } (\text{depth } x \ l) + 1) * p \ x$ ])
  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  $h = \lambda x. (\text{the\_fin } (\text{depth } x \ r) + 1) * p \ x$ ])
  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)  $\implies$  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 ( $\lambda(i::int, j::int). \text{nat } (j - i + 1)$ )*) *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<sub>m</sub>: 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<sub>T'</sub> ( $\lambda i. W_m' i n$ ) [0..n])*  
*Mapping.empty)*

**notation** *W<sub>m</sub>.crel\_vs (crel)*

**lemmas** *W<sub>m</sub>\_crel = W<sub>m</sub>.crel[unfolded W<sub>m</sub>.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 app\_syntax lifting\_syntax*  
**let** *?p = State\_Main.map<sub>T'</sub> ( $\lambda i. W_m' i n$ ) [0..n]*  
**let** *?q = map ( $\lambda i. W i n$ ) [0..n]*  
**have** *?q = map \$  $\langle\langle \lambda i. W\_fun i n \rangle\rangle$  \$  $\langle\langle [0..n] \rangle\rangle$*   
**unfolding** *Wrap\_def App\_def W\_fun\_correct ..*

```

have ?p = State_Main.mapT . ⟨λi. Wm' i n⟩ . ⟨[0..n]⟩
unfolding State_Monad_Ext.fun_app_lifted_def State_Main.mapT_def
bind_left_identity ..
— Not forgetting to write list_all2 (=) instead of (=) was the tricky part.
have Wm.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 Wm_crel, rule HOL.refl)
done
then have Wm.cmem (compute_W n)
unfolding compute_W_def by (elim Wm.crel_vs_elim[OF_ Wm.cmem_empty];
simp del: Wm'.simps)
with assms show ?thesis
unfolding W_fun_correct[symmetric] by (elim Wm.cmem_elim) (simp)+
qed

```

**definition**

```

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

```

**lemma** *W\_compute*:  $W\ i\ j = (\text{case } \text{Mapping.lookup } (\text{compute\_}W\ n)\ (i, j)\ \text{of } \text{None} \Rightarrow W\ i\ j\ | \text{Some } x \Rightarrow x)$   
**by** (auto dest: compute\_W\_correct split: option.split)

**lemma** *min\_wpl'\_correct*:  
 $\text{min\_wpl}'\ i\ j = \text{min\_wpl}\ W\ i\ j$   
**using** *W\_compute* **unfolding** *min\_wpl'\_def* **by** *simp*

**definition**

```

opt_bst' i j ≡
let
  M = compute_W j;
  W = (λi 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*:  
 $\text{opt\_bst}'\ i\ j = \text{opt\_bst}\ W\ i\ j$   
**using** *W\_compute* **unfolding** *opt\_bst'\_def* **by** *simp*

end

### 3.7.8 Test Case

Functional Implementations

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

**lemma** *opt\_bst* ( $\lambda i j. \text{nat}(i+j)$ ) 0 4 =  $\langle\langle\langle\langle\langle\rangle, 0, \langle\rangle\rangle, 1, \langle\rangle\rangle, 2, \langle\rangle\rangle, 3, \langle\rangle\rangle, 4, \langle\rangle\rangle$   
by *eval*

Using Frequencies

**definition**

*list\_to\_p xs (i::int)* = (if  $i - 1 \geq 0 \wedge \text{nat}(i - 1) < \text{length } xs$  then  $xs ! \text{nat}(i - 1)$  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* ( $\lambda i j. \text{nat}(i+j)$ ) 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*  $\subseteq$  {*xs. set xs*  $\subseteq$  *set ys*  $\wedge$  *length xs*  $\leq$  *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*]  $\vee$  *xs* = [])

**by** (*cases xs; simp add: subseq\_append\_le\_same\_iff[of \_ [], simplified]*)

**lemma** *subseq\_append\_single\_right*:

*subseq xs (ys @ [x])* = ( $(\exists$  *xs'. subseq xs' ys*  $\wedge$  *xs* = *xs' @ [x]*)  $\vee$  *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*  $\neq$  {}

**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**  $?R = \{\text{length } xs + 1 \mid xs. \text{subseq } xs \text{ (take } i \text{ } A) \wedge \text{subseq } xs \text{ (take } j \text{ } B)\}$   
**have**  $?S = \{\text{length } xs \mid xs. \text{subseq } xs \text{ (take } i \text{ } A) \wedge \text{subseq } xs \text{ (take } j \text{ } B)\}$   
 $\cup \{\text{length } xs \mid xs. \exists ys. \text{subseq } ys \text{ (take } i \text{ } A) \wedge \text{subseq } ys \text{ (take } j \text{ } B) \wedge xs$   
 $= ys @ [B!i]\}$

**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**  $\dots = \{\text{length } xs \mid xs. \text{subseq } xs \text{ (take } i \text{ } A) \wedge \text{subseq } xs$   
 $\text{(take } j \text{ } B)\}$   
 $\cup \{\text{length } xs + 1 \mid xs. \text{subseq } xs \text{ (take } i \text{ } A) \wedge \text{subseq } xs \text{ (take } j \text{ } B)\}$   
**by force**  
**moreover have**  $\text{Max } \dots = \text{Max } ?R$   
**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*  
 $f = \text{Max}]$ )  
**qed**

**lemma** *OPT\_rec2*:

$\text{OPT } (i + 1) (j + 1) = \max (\text{OPT } i (j + 1)) (\text{OPT } (i + 1) j)$  (**is**  $?l =$   
 $?r$ )

**if**  $A!i \neq B!j \ i < \text{length } A \ j < \text{length } B$

**proof** –

**have**  $\{\text{length } xs \mid xs. \text{subseq } xs \text{ (take } (i + 1) \text{ } A) \wedge \text{subseq } xs \text{ (take } (j + 1) \text{ } B)\}$   
 $B)\}$

$= \{\text{length } xs \mid xs. \text{subseq } xs \text{ (take } i \text{ } A) \wedge \text{subseq } xs \text{ (take } (j + 1) \text{ } B)\}$

$\cup \{\text{length } xs \mid xs. \text{subseq } xs \text{ (take } (i + 1) \text{ } A) \wedge \text{subseq } xs \text{ (take } j \text{ } B)\}$

**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'*:

$\text{OPT } i \ j = \text{lcs } i \ j$  **if**  $i \leq \text{length } A \ j \leq \text{length } B$

**using** *that OPT\_rec1 OPT\_rec2 by (induction i j rule: lcs.induct; simp*  
 $\text{add: OPT_0_left OPT_0_right})$



**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 lcsa "ABCDGH" "AEDFHR"

value lcsa "AGGTAB" "GXTXAYB"

end
theory All_Examples
  imports
    Bellman_Ford
    Knapsack
    Counting_Tiles
    CYK
    Min_Ed_Dist0
    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.