

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
1	Monadification	4
1.1	Monads	4
1.2	Parametricity of the State Monad	5
1.3	Miscellaneous Parametricity Theorems	10
1.4	Heap Monad	11
1.5	Relation Between the State and the Heap Monad	12
1.6	Parametricity of the Heap Monad	18
2	Memoization	22
2.1	Memory Implementations for the State Monad	22
2.1.1	Tracing Memory	24
2.2	Pair Memory	26
2.3	Indexing	33
2.4	Heap Memory Implementations	39
2.5	Tool Setup	55

2.6	Bottom-Up Computation	56
2.7	Setup for the Heap Monad	60
2.7.1	More Heap	64
2.7.2	Code Setup	71
2.8	Setup for the State Monad	72
2.8.1	Code Setup	75
3	Examples	76
3.1	Misc	76
3.2	The Bellman-Ford Algorithm	78
3.2.1	Misc	78
3.2.2	Single-Sink Shortest Path Problem	81
3.2.3	Functional Correctness	82
3.2.4	Functional Memoization	83
3.2.5	Imperative Memoization	85
3.2.6	Detecting Negative Cycles	86
3.2.7	Extracting an Executable Constant for the Imperative Implementation	90
3.2.8	Test Cases	91
3.3	The Knapsack Problem	94
3.3.1	Definitions	94
3.3.2	Functional Correctness	94
3.3.3	Functional Memoization	95
3.3.4	Imperative Memoization	95
3.3.5	Memoization	97
3.3.6	Regression Test	97
3.4	A Counting Problem	97
3.4.1	Misc	98
3.4.2	Problem Specification	98
3.4.3	Combinatorial Identities	99
3.4.4	Computing the Fill-Count Function	101
3.4.5	Memoization	101
3.4.6	Problem solutions	102
3.5	The CYK Algorithm	102
3.5.1	Misc	103
3.5.2	Definitions	103
3.5.3	CYK on Lists	103
3.5.4	CYK on Lists and Index	104
3.5.5	CYK With Index Function	104
3.5.6	Correctness Proof	105
3.5.7	Functional Memoization	105
3.5.8	Imperative Memoization	105
3.5.9	Functional Test Case	107
3.5.10	Imperative Test Case	107

3.6	Minimum Edit Distance	108
3.6.1	Misc	108
3.6.2	Edit Distance	108
3.6.3	Minimum Edit Sequence	109
3.6.4	Computing the Minimum Edit Distance	110
3.6.5	Indexing	111
3.6.6	Functional Memoization	112
3.6.7	Imperative Memoization	112
3.6.8	Test Cases	113
3.7	Optimal Binary Search Trees	115
3.7.1	Function <i>argmin</i>	116
3.7.2	Misc	116
3.7.3	Definitions	116
3.7.4	Functional Memoization	117
3.7.5	Correctness Proof	118
3.7.6	Access Frequencies	119
3.7.7	Memoizing Weights	121
3.7.8	Test Case	123
3.8	Longest Common Subsequence	124
3.8.1	Misc	124
3.8.2	Definitions	124
3.8.3	Correctness Proof	125
3.8.4	Functional Memoization	126
3.8.5	Test Case	126

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**

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*:

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

$\langle \text{proof} \rangle$ **lemma** *return_app_return_meta*:

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

$\langle \text{proof} \rangle$ **definition** *if_T* :: $(M, \text{bool}) \text{ state} \Rightarrow (M, 'a) \text{ state} \Rightarrow (M, 'a) \text{ state} \Rightarrow (M, 'a) \text{ state}$ **where**

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 ⇒ 'a **where**

Wrap x ≡ x

definition *App* :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b **where**

App f ≡ f

lemma *Wrap_App_Wrap*:

App (*Wrap* f) (*Wrap* x) ≡ f x

⟨*proof*⟩

end

1.2 Parametricity of the State Monad

theory *DP_CRelVS*

imports *./State_Monad_Ext* *./Pure_Monad*

begin

definition *lift_p* :: ('s ⇒ bool) ⇒ ('s, 'a) state ⇒ bool **where**

lift_p P f =

(∀ heap. P heap → (case *State_Monad.run_state* f heap of (_, heap) ⇒ P heap))

context

fixes P f heap

assumes *lift*: *lift_p* P f **and** P: P heap

begin

lemma *run_state_cases*:

case *State_Monad.run_state* f heap of (_, heap) ⇒ P heap

⟨*proof*⟩

lemma *lift_p_P*:

P heap' **if** *State_Monad.run_state* f heap = (v, heap')

⟨*proof*⟩

end

locale *state_mem_defs* =

fixes *lookup* :: 'param ⇒ ('mem, 'result option) state

and *update* :: 'param ⇒ 'result ⇒ ('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
($_ \$ _ = CHECKMEM = _ [1000,51] 51$) **where**
(dp_T \$ param = CHECKMEM = calc) \equiv (dp_T param = *checkmem* param calc)
term 0

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

definition *checkmem'* :: 'param \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
<proof>

lemma *checkmem_eq_alt*:
checkmem_eq dp param calc = (dp param = *checkmem'* param ($\lambda _ .$ calc))

```

    <proof>

end

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

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

context
  includes lifting_syntax state_monad_syntax
begin

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

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

abbreviation rel_fun_lifted :: ('a  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  ('b  $\Rightarrow$  'd  $\Rightarrow$  bool)  $\Rightarrow$ 
('a  $\Rightarrow$  'b)  $\Rightarrow$  ('c  $\implies$  'd)  $\Rightarrow$  bool (infixr  $\implies_T$  55) where
  rel_fun_lifted R R'  $\equiv$  R  $\implies_T$  crel_vs R'
term 0

definition consistentDP :: ('param  $\implies$  'mem  $\implies$  'result)  $\Rightarrow$  bool where
  consistentDP  $\equiv$  ((=)  $\implies_T$  crel_vs (=)) dp
term 0

```

private lemma *cmem_intro*:

assumes $\bigwedge param\ v\ M'.\ State_Monad.run_state\ (lookup\ param)\ M = (Some\ v,\ M') \implies v = dp\ param$

shows *cmem* M

$\langle proof \rangle$

lemma *cmem_elim*:

assumes *cmem* M $State_Monad.run_state\ (lookup\ param)\ M = (Some\ v,\ M')$

obtains $dp\ param = v$

$\langle proof \rangle$

term 0

lemma *crel_vs_intro*:

assumes $\bigwedge M\ v'\ M'.\ \llbracket cmem\ M;\ P\ M;\ State_Monad.run_state\ v_T\ M = (v',\ M') \rrbracket \implies R\ v\ v' \wedge cmem\ M' \wedge P\ M'$

shows *crel_vs* $R\ v\ v_T$

$\langle proof \rangle$

term 0

lemma *crel_vs_elim*:

assumes *crel_vs* $R\ v\ v_T$ *cmem* $M\ P\ M$

obtains $v'\ M'$ **where** $State_Monad.run_state\ v_T\ M = (v',\ M')\ R\ v\ v'$
cmem $M'\ P\ M'$

$\langle proof \rangle$

term 0

lemma *consistentDP_intro*:

assumes $\bigwedge param.\ Transfer.Rel\ (crel_vs\ (=))\ (dp\ param)\ (dp_T\ param)$

shows *consistentDP* dp_T

$\langle proof \rangle$

lemma *crel_vs_return*:

$\llbracket Transfer.Rel\ R\ x\ y \rrbracket \implies Transfer.Rel\ (crel_vs\ R)\ (Wrap\ x)\ (State_Monad.return\ y)$

$\langle proof \rangle$

term 0

lemma *crel_vs_return_ext*:

$\llbracket Transfer.Rel\ R\ x\ y \rrbracket \implies Transfer.Rel\ (crel_vs\ R)\ x\ (State_Monad.return\ y)$

$y)$
 $\langle \text{proof} \rangle$
term 0

private lemma *cmem_upd:*

$\text{cmem } M' \text{ if cmem } M \text{ P } M \text{ State_Monad.run_state (update param (dp param)) } M = (v, M')$

$\langle \text{proof} \rangle$ **lemma** *P_upd:*

$P \text{ M}' \text{ if } P \text{ M } \text{ State_Monad.run_state (update param (dp param)) } M = (v, M')$

$\langle \text{proof} \rangle$ **lemma** *crel_vs_get:*

$\llbracket \bigwedge M. \text{cmem } M \implies \text{crel_vs } R \text{ v } (sf \text{ M}) \rrbracket \implies \text{crel_vs } R \text{ v } (\text{State_Monad.get} \gg \gg sf)$

$\langle \text{proof} \rangle$

term 0

private lemma *crel_vs_set:*

$\llbracket \text{crel_vs } R \text{ v } sf; \text{cmem } M; P \text{ M} \rrbracket \implies \text{crel_vs } R \text{ v } (\text{State_Monad.set } M \gg \gg sf)$

$\langle \text{proof} \rangle$

term 0

private lemma *crel_vs_bind_eq:*

$\llbracket \text{crel_vs } (=) \text{ v } s; \text{crel_vs } R \text{ (f v)} (sf \text{ v}) \rrbracket \implies \text{crel_vs } R \text{ (f v)} (s \gg \gg sf)$

$\langle \text{proof} \rangle$

term 0

lemma *bind_transfer[transfer_rule]:*

$(\text{crel_vs } R0 \text{ ===} \gg (\text{R0} \text{ ===} \gg_T \text{ R1}) \text{ ===} \gg \text{crel_vs } R1) (\lambda v f. f \text{ v}) (\gg \gg)$

$\langle \text{proof} \rangle$ **lemma** *cmem_lookup:*

$\text{cmem } M' \text{ if cmem } M \text{ P } M \text{ State_Monad.run_state (lookup param) } M = (v, M')$

$\langle \text{proof} \rangle$ **lemma** *P_lookup:*

$P \text{ M}' \text{ if } P \text{ M } \text{ State_Monad.run_state (lookup param) } M = (v, M')$

$\langle \text{proof} \rangle$

lemma *crel_vs_lookup:*

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

$\langle \text{proof} \rangle$

lemma *crel_vs_update:*

$\text{crel_vs } (=) \text{ () (update param (dp param))}$

$\langle \text{proof} \rangle$ **lemma** *crel_vs_checkmem*:
 $\llbracket \text{is_equality } R; \text{Transfer.Rel } (crel_vs\ R) (dp\ param)\ s \rrbracket$
 $\implies \text{Transfer.Rel } (crel_vs\ R) (dp\ param) (\text{checkmem } param\ s)$
 $\langle \text{proof} \rangle$

lemma *crel_vs_checkmem_tupled*:
assumes $v = dp\ param$
shows $\llbracket \text{is_equality } R; \text{Transfer.Rel } (crel_vs\ R)\ v\ s \rrbracket$
 $\implies \text{Transfer.Rel } (crel_vs\ R)\ v\ (\text{checkmem } param\ s)$
 $\langle \text{proof} \rangle$

lemma *return_transfer[transfer_rule]*:
 $(R\ ==>_T\ R)\ \text{Wrap } \text{State_Monad.return}$
 $\langle \text{proof} \rangle$

lemma *fun_app_lifted_transfer[transfer_rule]*:
 $(crel_vs\ (R0\ ==>_T\ R1)\ ==>_T\ crel_vs\ R0\ ==>_T\ crel_vs\ R1)\ \text{App } (\cdot)$
 $\langle \text{proof} \rangle$

lemma *crel_vs_fun_app*:
 $\llbracket \text{Transfer.Rel } (crel_vs\ R0)\ x\ x_T; \text{Transfer.Rel } (crel_vs\ (R0\ ==>_T\ R1))\ f\ f_T \rrbracket \implies \text{Transfer.Rel } (crel_vs\ R1)\ (\text{App } f\ x)\ (f_T\ \cdot\ x_T)$
 $\langle \text{proof} \rangle$

lemma *if_T_transfer[transfer_rule]*:
 $(crel_vs\ (=)\ ==>_T\ crel_vs\ R\ ==>_T\ crel_vs\ R\ ==>_T\ crel_vs\ R)\ \text{If } \text{State_Monad_Ext.if}_T$
 $\langle \text{proof} \rangle$
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 $(R1\ ==>_T\ S1)\ f\ g\ (R2\ ==>_T\ S2)\ g\ h$

shows $(R1 \text{ OO } R2 \implies S1 \text{ OO } S2) f h$
 $\langle proof \rangle$

lemma *rel_fun_comp1*:

assumes $(R1 \implies S1) f g (R2 \implies S2) g h R' = R1 \text{ OO } R2$
shows $(R' \implies S1 \text{ OO } S2) f h$
 $\langle proof \rangle$

lemma *rel_fun_comp2*:

assumes $(R1 \implies S1) f g (R2 \implies S2) g h S' = S1 \text{ OO } S2$
shows $(R1 \text{ OO } R2 \implies S') f h$
 $\langle proof \rangle$

lemma *rel_fun_relcomp*:

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

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 $(R' \implies S1 \text{ OO } S2) f h$
 $\langle proof \rangle$

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 $(R1 \text{ OO } R2 \implies S') f h$
 $\langle proof \rangle$

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

$fun_app_lifted f_T x_T \equiv do \{ f \leftarrow f_T; x \leftarrow x_T; f x \}$

bundle *heap_monad_syntax* **begin**

notation *fun_app_lifted* (**infixl** . 999)
type_synonym ('a, 'b) *fun_lifted* = 'a \Rightarrow 'b *Heap* ($_ \Rightarrow H \Longrightarrow _$ [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*:

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

$\langle \text{proof} \rangle$ **lemma** *return_app_return_meta*:

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

$\langle \text{proof} \rangle$ **definition** *if_T* :: *bool Heap* \Rightarrow 'a *Heap* \Rightarrow 'a *Heap* \Rightarrow 'a *Heap*

where

if_T *b_T* *x_T* *y_T* \equiv *do* { *b* \leftarrow *b_T*; *if* *b* *then* *x_T* *else* *y_T* }

end

end

1.5 Relation Between the State and the Heap Monad

theory *State_Heap*

imports

../state_monad/DP_CRelVS

HOL-Imperative_HOL.Imperative_HOL

State_Heap_Misc

Heap_Monad_Ext

begin

definition *lift_p* :: (*heap* \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
<proof>

lemma *execute_cases'*:

case execute f heap of Some ($_$, heap) \Rightarrow P heap
<proof>

lemma *lift_p_None[simp, dest]*:

False if execute f heap = None
<proof>

lemma *lift_p_P*:

case the (execute f heap) of ($_$, heap) \Rightarrow P heap
<proof>

lemma *lift_p_P'*:

P heap' if the (execute f heap) = (v, heap')
<proof>

lemma *lift_p_P''*:

P heap' if execute f heap = Some (v, heap')
<proof>

lemma *lift_p_the_Some[simp]*:

execute f heap = Some (v, heap') if the (execute f heap) = (v, heap')
<proof>

lemma *lift_p_E*:

obtains *v heap' where execute f heap = Some (v, heap')* *P heap'*
<proof>

end

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

locale *heap_mem_defs* =

fixes *P* :: *heap* \Rightarrow *bool*
and *lookup* :: *'k* \Rightarrow *'v option Heap*
and *update* :: *'k* \Rightarrow *'v* \Rightarrow *unit Heap*

begin

definition *rel_state* :: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*heap*, *'a*) *state* \Rightarrow *'b Heap* \Rightarrow *bool* **where**

$rel_state\ R\ f\ g \equiv$
 $\forall\ heap.\ P\ heap \longrightarrow$
 $(case\ State_Monad.run_state\ f\ heap\ of\ (v1,\ heap1) \Rightarrow case\ execute\ g$
 $heap\ of$
 $\quad Some\ (v2,\ heap2) \Rightarrow R\ v1\ v2 \wedge heap1 = heap2 \wedge P\ heap2 \mid None \Rightarrow$
 $False)$

definition $lookup'\ k \equiv State\ (\lambda\ heap.\ the\ (execute\ (lookup\ k)\ heap))$

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

definition $heap_get = Heap_Monad.Heap\ (\lambda\ heap.\ Some\ (heap,\ heap))$

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

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

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

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

lemma $checkmem_checkmem'$:

$checkmem'\ param\ (\lambda\ _.\ calc) = checkmem\ param\ calc$
 $\langle proof \rangle$

definition $map_of_heap\ \mathbf{where}$

$map_of_heap\ heap\ k = fst\ (the\ (execute\ (lookup\ k)\ heap))$

lemma *rel_state_elim*:

assumes *rel_state R f g P heap*

obtains *heap' v v' where*

State_Monad.run_state f heap = (v, heap') *execute g heap = Some (v', heap')* *R v v' P heap'*

<proof>

lemma *rel_state_intro*:

assumes

$\bigwedge heap\ v\ heap'.\ P\ heap \implies State_Monad.run_state\ f\ heap = (v, heap')$
 $\implies \exists v'.\ R\ v\ v' \wedge execute\ g\ heap = Some\ (v', heap')$

$\bigwedge heap\ v\ heap'.\ P\ heap \implies State_Monad.run_state\ f\ heap = (v, heap')$
 $\implies P\ heap'$

shows *rel_state R f g*

<proof>

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

<proof>

lemma *transfer_return[transfer_rule]*:

$(R\ ==\!\!\!=\!\!\!>\ rel_state\ R)\ State_Monad.return\ Heap_Monad.return$

<proof>

lemma *fun_app_lifted_transfer*:

$(rel_state\ (R\ ==\!\!\!=\!\!\!>\ rel_state\ Q)\ ==\!\!\!=\!\!\!>\ rel_state\ R\ ==\!\!\!=\!\!\!>\ rel_state\ Q)$

State_Monad_Ext.fun_app_lifted Heap_Monad_Ext.fun_app_lifted

<proof>

lemma *transfer_get[transfer_rule]*:

$rel_state\ (=)\ State_Monad.get\ heap_get$

<proof>

end

end

```

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

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

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

context
  includes lifting_syntax
begin

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

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

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

end

end

locale heap_correct =
  heap_inv +
  assumes lookup_correct:
    P m ⇒ map_of_heap (snd (the (execute (lookup k) m))) ⊆m
    (map_of_heap m)
  and update_correct:
    P m ⇒ map_of_heap (snd (the (execute (update k v) m))) ⊆m
    (map_of_heap m)(k ↦ v)

```


begin

lemma *lookup'_correct*:

state_mem_defs.map_of lookup' (snd (State_Monad.run_state (lookup' k) m)) \subseteq_m *state_mem_defs.map_of lookup' m* **if** *P m*
<proof>

lemma *update'_correct*:

state_mem_defs.map_of lookup' (snd (State_Monad.run_state (update' k v) m)) \subseteq_m *state_mem_defs.map_of lookup' m* (*k ↦ v*)
if *P m*
<proof>

lemma *lookup'_inv*:

DP_CRelVS.lift_p P (lookup' k)
<proof>

lemma *update'_inv*:

DP_CRelVS.lift_p P (update' k v)
<proof>

lemma *mem_correct_heap*: *mem_correct lookup' update' P*

<proof>

end

context *heap_mem_defs*

begin

context

includes *lifting_syntax*

begin

lemma *mem_correct_heap_correct*:

assumes *correct*: *mem_correct lookup_s update_s P*

and *lookup*: *((=) ==> rel_state (=)) lookup_s lookup*

and *update*: *((=) ==> (=) ==> rel_state (=)) update_s update*

shows *heap_correct P update lookup*

<proof>

end

end

end

1.6 Parametricity of the Heap Monad

theory *DP_CRelVH*

imports *State_Heap*

begin

locale *dp_heap* =

state_dp_consistency: *dp_consistency lookup_st update_st P dp* + *heap_mem_defs*

Q lookup update

for *P Q* :: *heap* \Rightarrow *bool* and *dp* :: *'k* \Rightarrow *'v* and *lookup* :: *'k* \Rightarrow *'v option*

Heap

and *lookup_st update update_st* +

assumes

rel_state_lookup: *rel_fun* (=) (*rel_state* (=)) *lookup_st lookup*

and

rel_state_update: *rel_fun* (=) (*rel_fun* (=) (*rel_state* (=))) *update_st*

update

begin

context

includes *lifting_syntax 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', heap') \Rightarrow *P heap'* \wedge *Q heap'* \wedge *R v v'* \wedge *state_dp_consistency.cmem*

heap'

)

abbreviation *rel_fun_lifted* :: (*'a* \Rightarrow *'c* \Rightarrow *bool*) \Rightarrow (*'b* \Rightarrow *'d* \Rightarrow *bool*) \Rightarrow

(*'a* \Rightarrow *'b*) \Rightarrow (*'c* \implies *H* \implies *'d*) \Rightarrow *bool* (**infixr** \implies_T 55) **where**

rel_fun_lifted R R' \equiv *R* \implies *crel_vs R'*

definition *consistentDP* :: (*'k* \Rightarrow *'v Heap*) \Rightarrow *bool* **where**

consistentDP \equiv ((=) \implies *crel_vs* (=)) *dp*

lemma *consistentDP_intro*:

assumes \bigwedge *param. Transfer.Rel* (*crel_vs* (=)) (*dp param*) (*dp_T param*)

shows *consistentDP dpT*
⟨proof⟩

lemma *crel_vs_execute_None*:

False **if** *crel_vs R a b execute b heap = None P heap Q heap state_dp_consistency.cmem heap*
⟨proof⟩

lemma *crel_vs_execute_Some*:

assumes *crel_vs R a b P heap Q heap state_dp_consistency.cmem heap*
obtains *x heap'* **where** *execute b heap = Some (x, heap')* *P heap' Q heap'*
⟨proof⟩

lemma *crel_vs_executeD*:

assumes *crel_vs R a b P heap Q heap state_dp_consistency.cmem heap*
obtains *x heap'* **where**
execute b heap = Some (x, heap') *P heap' Q heap' state_dp_consistency.cmem heap' R a x*
⟨proof⟩

lemma *crel_vs_success*:

assumes *crel_vs R a b P heap Q heap state_dp_consistency.cmem heap*
shows *success b heap*
⟨proof⟩

lemma *crel_vsI*: *crel_vs R a b* **if** *(state_dp_consistency.crel_vs R OO rel_state (=)) a b*
⟨proof⟩

lemma *transfer'_return[transfer_rule]*:

(R ===> crel_vs R) *Wrap return*
⟨proof⟩

lemma *crel_vs_return*:

Transfer.Rel (crel_vs R) (Wrap x) (return y) **if** *Transfer.Rel R x y*
⟨proof⟩

lemma *crel_vs_return_ext*:

$\llbracket \text{Transfer.Rel } R \ x \ y \rrbracket \implies \text{Transfer.Rel } (crel_vs \ R) \ x \ (\text{Heap_Monad.return } y)$
⟨proof⟩

term 0

lemma *bind_transfer[transfer_rule]*:

$(\text{crel_vs } R0 \text{ } \text{====>} (R0 \text{ } \text{====>} \text{crel_vs } R1) \text{ } \text{====>} \text{crel_vs } R1) (\lambda v f. f$
 $v) (\gg=)$
 $\langle \text{proof} \rangle$

lemma *crel_vs_update*:
 $\text{crel_vs } (=) () (\text{update param } (dp \text{ param}))$
 $\langle \text{proof} \rangle$

lemma *crel_vs_lookup*:
 crel_vs
 $(\lambda v v'. \text{case } v' \text{ of None } \Rightarrow \text{True} \mid \text{Some } v' \Rightarrow v = v' \wedge v = dp \text{ param})$
 $(dp \text{ param}) (\text{lookup param})$
 $\langle \text{proof} \rangle$

lemma *crel_vs_eq_eq_onp*:
 $\text{crel_vs } (eq_onp (\lambda x. x = v)) v s \text{ if } \text{crel_vs } (=) v s$
 $\langle \text{proof} \rangle$

lemma *crel_vs_bind_eq*:
 $\llbracket \text{crel_vs } (=) v s; \text{crel_vs } R (f v) (sf v) \rrbracket \Longrightarrow \text{crel_vs } R (f v) (s \gg= sf)$
 $\langle \text{proof} \rangle$

lemma *crel_vs_checkmem*:
 $\text{Transfer.Rel } (\text{crel_vs } R) (dp \text{ param}) (\text{checkmem param } s) \text{ if } \text{is_equality}$
 $R \text{ Transfer.Rel } (\text{crel_vs } R) (dp \text{ param}) s$
 $\langle \text{proof} \rangle$

lemma *crel_vs_checkmem_tupled*:
assumes $v = dp \text{ param}$
shows $\llbracket \text{is_equality } R; \text{Transfer.Rel } (\text{crel_vs } R) v s \rrbracket$
 $\Longrightarrow \text{Transfer.Rel } (\text{crel_vs } R) v (\text{checkmem param } s)$
 $\langle \text{proof} \rangle$

lemma *transfer_fun_app_lifted[transfer_rule]*:
 $(\text{crel_vs } (R0 \text{ } \text{====>} \text{crel_vs } R1) \text{ } \text{====>} \text{crel_vs } R0 \text{ } \text{====>} \text{crel_vs } R1)$
 $\text{App Heap_Monad_Ext.fun_app_lifted}$
 $\langle \text{proof} \rangle$

lemma *crel_vs_fun_app*:
 $\llbracket \text{Transfer.Rel } (\text{crel_vs } R0) x x_T; \text{Transfer.Rel } (\text{crel_vs } (R0 \text{ } \text{====>}_T R1))$
 $f f_T \rrbracket \Longrightarrow \text{Transfer.Rel } (\text{crel_vs } R1) (\text{App } f x) (f_T . x_T)$
 $\langle \text{proof} \rangle$

end

end

locale *dp_consistency_heap* = *heap_correct* +

fixes *dp* :: 'a ⇒ 'b

begin

interpretation *state_mem_correct*: *mem_correct lookup' update' P*

⟨*proof*⟩

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

⟨*proof*⟩

lemma *dp_heap*: *dp_heap P P lookup lookup' update update'*

⟨*proof*⟩

sublocale *dp_heap P P dp lookup lookup' update update'*

⟨*proof*⟩

notation *rel_fun_lifted* (**infixr** $===>_T$ 55)

end

locale *heap_correct_empty* = *heap_correct* +

fixes *empty*

assumes *empty_correct*: *map_of_heap empty* \subseteq_m *Map.empty* **and** *P_empty*:
P empty

locale *dp_consistency_heap_empty* =

dp_consistency_heap + *heap_correct_empty*

begin

lemma *cmem_empty*:

state_dp_consistency.cmem empty

⟨*proof*⟩

corollary *memoization_correct*:

dp x = v state_dp_consistency.cmem m **if**

consistentDP dp_T Heap_Monad.execute (dp_T x) empty = Some (v, m)

⟨*proof*⟩

lemma *memoized_success*:

success (dp_T x) empty **if** *consistentDP dp_T*

⟨*proof*⟩

lemma *memoized*:

```
  dp x = fst (the (Heap_Monad.execute (dp_T x) empty)) if consistentDP
dp_T
  ⟨proof⟩
```

lemma *cmem_result*:

```
  state_dp_consistency.cmem (snd (the (Heap_Monad.execute (dp_T x) empty)))
if consistentDP dp_T
  ⟨proof⟩
```

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  $\wedge$  heap x heap'. P heap  $\implies$  run_state f heap = (x, heap')
 $\implies$  P heap'
  ⟨proof⟩
```

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)
  ⟨proof⟩
```

lemma *mem_correct_rbt_mapping*:

```
  mem_correct
  (λ k. do {m ← State_Monad.get; State_Monad.return (Mapping.lookup
m k)})
  (λ k v. do {m ← State_Monad.get; State_Monad.set (Mapping.update
k v m)})
  (λ _. True)
  ⟨proof⟩
```

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

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

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

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

locale *dp_consistency_empty* =
dp_consistency + *mem_correct_empty*
begin

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

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

lemma *memoized*:

```

dp x = fst (State_Monad.run_state (dp_T x) empty) if consistentDP dp_T
⟨proof⟩

```

lemma *cmem_result*:

```

cmem (snd (State_Monad.run_state (dp_T x) empty)) if consistentDP dp_T
⟨proof⟩

```

end

locale *dp_consistency_default* =

```

fixes dp :: 'param ⇒ 'result

```

begin

sublocale *dp_consistency_empty*

```

λ k. do {(m::'param → 'result) ← State_Monad.get; State_Monad.return
(m k)}

```

```

λ k v. do {m ← State_Monad.get; State_Monad.set (m(k↦v))}

```

```

λ (_::'param → 'result). True

```

```

dp

```

```

Map.empty

```

```

⟨proof⟩

```

end

locale *dp_consistency_mapping* =

```

fixes dp :: 'param ⇒ 'result

```

begin

sublocale *dp_consistency_empty*

```

(λ k. do {(m::('param,'result) mapping) ← State_Monad.get; State_Monad.return
(Mapping.lookup m k)})

```

```

(λ k v. do {m ← State_Monad.get; State_Monad.set (Mapping.update
k v m)})

```

```

(λ _::('param,'result) mapping. True) dp Mapping.empty

```

```

⟨proof⟩

```

end

2.1.1 Tracing Memory

context *state_mem_defs*

begin

definition


```

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

```

definition

```

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

```

end

context *mem_correct*

begin

lemma *map_of_simp*:

```

state_mem_defs.map_of lookup_trace = map_of o snd
⟨proof⟩

```

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

```

⟨proof⟩

```

end

context *mem_correct_empty*

begin

lemma *mem_correct_tracing_empty*:

```

mem_correct_empty lookup_trace update_trace (P o snd) ([], empty)
⟨proof⟩

```

end

locale *dp_consistency_mapping_tracing* =

```

  fixes dp :: 'param ⇒ 'result

```

begin

interpretation *mapping*: *dp_consistency_mapping* ⟨proof⟩

sublocale *dp_consistency_empty*

```

  mapping.lookup_trace mapping.update_trace (λ _. True) o snd dp ([],

```

Mapping.empty)
 ⟨*proof*⟩

end

end

2.2 Pair Memory

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

lemma *map_add_mono*:

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

lemma *map_add_upd2*:

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

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 <proof>

sublocale mem1: state_mem_defs lookup1 update1 <proof>

sublocale mem2: state_mem_defs lookup2 update2 <proof>

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*:

$P\ m \implies State_Monad.run_state\ (update1\ k'\ v)\ m = (x, m') \implies fst\ (State_Monad.run_state\ get_k1\ m') = fst\ (State_Monad.run_state\ get_k1\ m)$

$P\ m \implies State_Monad.run_state\ (update1\ k'\ v)\ m = (x, m') \implies$

$fst\ (State_Monad.run_state\ get_k2\ m') = fst\ (State_Monad.run_state\ get_k2\ m)$

$get_k2\ m)$
 $P\ m \implies State_Monad.run_state\ (update2\ k'\ v)\ m = (x,\ m') \implies$
 $\text{fst}\ (State_Monad.run_state\ get_k1\ m') = \text{fst}\ (State_Monad.run_state$
 $get_k1\ m)$
 $P\ m \implies State_Monad.run_state\ (update2\ k'\ v)\ m = (x,\ m') \implies$
 $\text{fst}\ (State_Monad.run_state\ get_k2\ m') = \text{fst}\ (State_Monad.run_state$
 $get_k2\ m)$

assumes

lookup_correct:

$P\ m \implies mem1.map_of\ (snd\ (State_Monad.run_state\ (lookup1\ k')$
 $m)) \subseteq_m\ (mem1.map_of\ m)$

$P\ m \implies mem2.map_of\ (snd\ (State_Monad.run_state\ (lookup1\ k')$
 $m)) \subseteq_m\ (mem2.map_of\ m)$

$P\ m \implies mem1.map_of\ (snd\ (State_Monad.run_state\ (lookup2\ k')$
 $m)) \subseteq_m\ (mem1.map_of\ m)$

$P\ m \implies mem2.map_of\ (snd\ (State_Monad.run_state\ (lookup2\ k')$
 $m)) \subseteq_m\ (mem2.map_of\ m)$

assumes

update_correct:

$P\ m \implies mem1.map_of\ (snd\ (State_Monad.run_state\ (update1\ k'\ v)$
 $m)) \subseteq_m\ (mem1.map_of\ m)(k' \mapsto v)$

$P\ m \implies mem2.map_of\ (snd\ (State_Monad.run_state\ (update2\ k'\ v)$
 $m)) \subseteq_m\ (mem2.map_of\ m)(k' \mapsto v)$

$P\ m \implies mem2.map_of\ (snd\ (State_Monad.run_state\ (update1\ k'\ v)$
 $m)) \subseteq_m\ mem2.map_of\ m$

$P\ m \implies mem1.map_of\ (snd\ (State_Monad.run_state\ (update2\ k'\ v)$
 $m)) \subseteq_m\ mem1.map_of\ m$

begin

lemma *map_of_le_pair:*

$pair.map_of\ m \subseteq_m\ map_of1\ m\ ++\ map_of2\ m$

if *inv_pair* m

$\langle proof \rangle$

lemma *pair_le_map_of:*

$map_of1\ m\ ++\ map_of2\ m \subseteq_m\ pair.map_of\ m$

if *inv_pair* m

$\langle proof \rangle$

lemma *map_of_eq_pair:*

$map_of1\ m\ ++\ map_of2\ m = pair.map_of\ m$

if *inv_pair* m

$\langle proof \rangle$

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*)
⟨*proof*⟩

lemma *inv_pair_P_D*:

P *m* **if** *inv_pair* *m*
⟨*proof*⟩

lemma *inv_pair_domD[intro]*:

dom (*map_of1* *m*) ∩ *dom* (*map_of2* *m*) = {} **if** *inv_pair* *m*
⟨*proof*⟩

lemma *move12_correct1*:

map_of1 *heap'* ⊆_{*m*} *Map.empty* **if** *State_Monad.run_state* (*move12* *k1*)
heap = (*x*, *heap'*) *P* *heap*
⟨*proof*⟩

lemma *move12_correct2*:

map_of2 *heap'* ⊆_{*m*} *map_of1* *heap* **if** *State_Monad.run_state* (*move12* *k1*)
heap = (*x*, *heap'*) *P* *heap*
⟨*proof*⟩

lemma *dom_empty[simp]*:

dom (*map_of1* *heap'*) = {} **if** *State_Monad.run_state* (*move12* *k1*) *heap*
= (*x*, *heap'*) *P* *heap*
⟨*proof*⟩

lemma *inv_pair_lookup1*:

inv_pair *m'* **if** *State_Monad.run_state* (*lookup1* *k*) *m* = (*v*, *m'*) *inv_pair* *m*
⟨*proof*⟩

lemma *inv_pair_lookup2*:

inv_pair *m'* **if** *State_Monad.run_state* (*lookup2* *k*) *m* = (*v*, *m'*) *inv_pair* *m*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inv_pair_update2*:

inv_pair m'
if *State_Monad.run_state (update2 (key2 k) v) m = (v', m')* *inv_pair m*
fst (State_Monad.run_state get_k2 m) = key1 k
<proof>

lemma *inv_pair_move12*:

inv_pair m'
if *State_Monad.run_state (move12 k) m = (v', m')* *inv_pair m* *fst (State_Monad.run_state*
get_k1 m) ≠ k
<proof>

lemma *mem_correct_pair*:

mem_correct lookup_pair update_pair inv_pair
if *injective: ∀ k k'. key1 k = key1 k' ∧ key2 k = key2 k' → k = k'*
<proof>

lemma *emptyI*:

assumes *inv_pair m mem1.map_of m ⊆_m Map.empty mem2.map_of m*
⊆_m Map.empty
shows *pair.map_of m ⊆_m Map.empty*
<proof>

end

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

context *mem_correct_empty*

begin

context

fixes *key :: 'a ⇒ '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 ⟨proof⟩

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)
⟨proof⟩

```

lemma pair_le_map_of:

```

map_of m1 ++ map_of m2 ⊆m pair.map_of (Pair_Storage k1 k2 m1
m2)
if inv_pair (Pair_Storage k1 k2 m1 m2)
⟨proof⟩

```


lemma *map_of_eq_pair*:

map_of m1 ++ map_of m2 = pair.map_of (Pair_Storage k1 k2 m1 m2)

if *inv_pair (Pair_Storage k1 k2 m1 m2)*

<proof>

lemma *inv_pair_neq[simp, dest]*:

False if inv_pair (Pair_Storage k k x y)

<proof>

lemma *inv_pair_P_D1*:

P m1 if inv_pair (Pair_Storage k1 k2 m1 m2)

<proof>

lemma *inv_pair_P_D2*:

P m2 if inv_pair (Pair_Storage k1 k2 m1 m2)

<proof>

lemma *inv_pair_domD[intro]*:

dom (map_of m1) ∩ dom (map_of m2) = {} if inv_pair (Pair_Storage k1 k2 m1 m2)

<proof>

lemma *mem_correct_pair*:

mem_correct lookup_pair update_pair inv_pair

<proof>

end

end

end

2.3 Indexing

theory *Indexing*

imports *Main*

begin

definition *injective* :: *nat* \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>

lemma *index_eq_iff*:

fixes *a b c d b0* :: *nat*

assumes $b < b0$ $d < b0$ $a * b0 + b = c * b0 + d$

shows $a = c \wedge b = d$

<proof>

locale *prod_order_def* =

order0: *ord less_eq0 less0* +

order1: *ord less_eq1 less1*

for *less_eq0 less0 less_eq1 less1*

begin

fun *less* :: '*a* × '*b* ⇒ '*a* × '*b* ⇒ *bool* **where**

less (*a*,*b*) (*c*,*d*) \longleftrightarrow *less0* *a c* \wedge *less1* *b d*

fun *less_eq* :: '*a* × '*b* ⇒ '*a* × '*b* ⇒ *bool* **where**

less_eq *ab cd* \longleftrightarrow *less* *ab cd* \vee *ab = cd*

end

locale *prod_order* =

prod_order_def less_eq0 less0 less_eq1 less1 +

order0: *order less_eq0 less0* +

order1: *order less_eq1 less1*

for *less_eq0 less0 less_eq1 less1*

begin

sublocale *order less_eq less*

<proof>

end

locale *option_order* =

order0: *order less_eq0 less0*

for *less_eq0 less0*

begin

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

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

sublocale *order less_eq_option less_option*
 <proof>

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* =

idx0: *index_locale_def* *less_eq0* *less0* *idx0* *size0* +

idx1: *index_locale_def* *less_eq1* *less1* *idx1* *size1*

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

begin

fun *idx* :: ('a \times 'b) bound \Rightarrow 'a \times 'b \Rightarrow nat **where**

idx (Bound (l0, r0) (l1, r1)) (a, b) = (*idx0* (Bound l0 l1) a) * (*size1* (Bound
r0 r1)) + *idx1* (Bound r0 r1) b

fun *size* :: ('a \times 'b) bound \Rightarrow nat **where**

```

    size (Bound (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 +
  idx0: index_locale less_eq0 less0 idx0 size0 +
  idx1: 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 <proof>

sublocale index_locale less_eq less idx size <proof>
end

locale option_index =
  idx0: 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>

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
   $\langle$ proof $\rangle$ 

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
   $\langle$ proof $\rangle$ 

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
   $\langle$ proof $\rangle$ 
end

instantiation nat :: index
begin

interpretation nat_index  $\langle$ proof $\rangle$ 
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 $\langle proof \rangle$

end

instantiation $int :: index$
begin

interpretation int_index $\langle proof \rangle$
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 $\langle proof \rangle$
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 $size$
 $\langle proof \rangle$
thm $index_locale_axioms$

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

lemmas $size_prod = size.simps$

instance $\langle proof \rangle$

end

lemma *bound_int_simp*[code]:

*bounded_index.size (Bound (l1, l2) (u1, u2)) = nat (u1 - l1) * nat (u2 - l2)*

<proof>

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

lemmas [code] =

nat_index_def.size.simps

nat_index_def.idx.simps

lemmas [code] =

int_index_def.size.simps

int_index_def.idx.simps

lemmas [code] =

prod_index_def.size.simps

prod_index_def.idx.simps

lemmas [code] =

prod_order_def.less_eq.simps

prod_order_def.less.simps

lemmas *index_size_defs* =

prod_index_def.size.simps int_index_def.size.simps nat_index_def.size.simps

bounded_index.size_def

end

2.4 Heap Memory Implementations

theory *Memory_Heap*

imports *State_Heap DP_CRelVH Pair_Memory HOL-Eisbach.Eisbach*
../Indexing

begin

Move

abbreviation *result_of c h* \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** $\bigwedge x. m\ x = \text{None}$

<proof>

lemma *result_of_return[simp]*:
 result_of (Heap_Monad.return x) h = x
<proof>

lemma *get_result_of_lookup*:
 result_of (!r) heap = x if Ref.get heap r = x
<proof>

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

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

lemma *success_empty[intro]*:
 success mem_empty heap
<proof>

lemma *length_mem_empty*:
 Array.length
 (*heap_of (mem_empty* :: (*'b* :: *heap*) *option array*) *Heap*) *h*)
 (*result_of (mem_empty* :: (*'b* *option array*) *Heap*) *h*) = *size*
<proof>

lemma *nth_mem_empty*:
 result_of
 (*Array.nth (result_of (mem_empty* :: (*'b* *option array*) *Heap*) *h*) *i*)
 (*heap_of (mem_empty* :: (*'b* :: *heap*) *option array*) *Heap*) *h*) = *None*
if *i* < *size*
<proof>

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

definition
 mem_lookup *k* = (*let* *i* = *to_index* *k* *in*
 if *i* < *size* *then* *Array.nth* *mem* *i* *else* *return* *None*
)

definition

```

mem_update k v = (let i = to_index k in
  if i < size then (Array.upd i (Some v) mem  $\gg$  ( $\lambda \_.$  return ()))
  else return ()
)

```

context *assumes injective: injective size to_index*

begin

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

<proof>

lemmas *mem_heap_correct* = *heap_correct_axioms*

context

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

begin

interpretation *heap_correct_empty*

λ *heap.* *Array.length heap mem* = *size mem_update mem_lookup*

heap_of (mem_empty :: 'v option array Heap) Heap.empty

<proof>

lemmas *array_heap_emptyI* = *heap_correct_empty_axioms*

context

fixes dp :: 'k2 \Rightarrow 'v

begin

interpretation *dp_consistency_heap_empty*

λ *heap.* *Array.length heap mem* = *size mem_update mem_lookup dp*

heap_of (mem_empty :: 'v option array Heap) Heap.empty

<proof>

lemmas *array_consistentI* = *dp_consistency_heap_empty_axioms*

end

end

end

end

lemma *execute_bind_success'*:

assumes *success f h execute (f \ggg g) h = Some (y, h')*
obtains *x h' where execute f h = Some (x, h') execute (g x) h' = Some (y, h')*
<proof>

lemma *success_bind_I*:

assumes *success f h*
and $\bigwedge x h'. \text{execute } f h = \text{Some } (x, h') \implies \text{success } (g x) h'$
shows *success (f \ggg g) h*
<proof>

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')*
<proof>

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
<proof>

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')*
<proof>

lemma *alloc_pairD3*:

Array.present heap' r
if *execute (alloc_pair a b) heap = Some ((r1, r2), heap')* *Array.present*
heap r
<proof>

lemma *alloc_pairD4*:

Ref.get heap' r = x
if *execute (alloc_pair a b) heap = Some ((r1, r2), heap')*
Ref.get heap r = x Ref.present heap r
 ⟨*proof*⟩

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

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

lemma *alloc_pair_nth*:
result_of (Array.nth r i) heap' = result_of (Array.nth r i) heap
if *execute (alloc_pair a b) heap = Some ((r1, r2), heap')*
 ⟨*proof*⟩

lemma *success_alloc_pair[intro]*:
success (alloc_pair a b) heap
 ⟨*proof*⟩

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
 ⟨*proof*⟩

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

$m_ref1 \neq m_ref2 \wedge m_ref1 \neq k_ref1 \wedge m_ref1 \neq k_ref2 \wedge$
 $m_ref2 \neq k_ref1$
 $\wedge m_ref2 \neq k_ref2 \wedge k_ref1 \neq k_ref2$
(*proof*)

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
(*proof*)

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
(*proof*)

lemma *succes_init_state_inner[intro]*:

success (*init_state_inner* *k1 k2 m1 m2*) *heap*
(*proof*)

lemma *init_state_inner_nth*:

result_of (*Array.nth* *r i*) *heap'* = *result_of* (*Array.nth* *r i*) *heap*

if *execute* (*init_state_inner* *k1 k2 m1 m2*) *heap* = *Some* ((*r1*, *r2*), *heap'*)

(*proof*)

definition

init_state *k1 k2* \equiv *do* {
 m1 \leftarrow *mem_empty*;
 m2 \leftarrow *mem_empty*;
 init_state_inner *k1 k2 m1 m2*
}

lemma *succes_init_state[intro]*:

success (*init_state* *k1 k2*) *heap*

<proof>

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

$inv_distinct\ k_ref1\ k_ref2\ m_ref1\ m_ref2$
<proof>

lemma *init_state_present*:

assumes

$execute\ (init_state\ k1\ k2)\ heap = Some\ ((k_ref1,\ k_ref2,\ m_ref1,$
 $m_ref2),\ heap')$

shows

$Ref.present\ heap'\ k_ref1\ Ref.present\ heap'\ k_ref2$
 $Ref.present\ heap'\ m_ref1\ Ref.present\ heap'\ m_ref2$
<proof>

lemma *empty_present*:

$Array.present\ h'\ x\ \mathbf{if}\ execute\ mem_empty\ heap = Some\ (x,\ h')$
<proof>

lemma *empty_present'*:

$Array.present\ h'\ a\ \mathbf{if}\ execute\ mem_empty\ heap = Some\ (x,\ h')\ Array.present$
 $heap\ a$
<proof>

lemma *init_state_present2*:

assumes

$execute\ (init_state\ k1\ k2)\ heap = Some\ ((k_ref1,\ k_ref2,\ m_ref1,$
 $m_ref2),\ heap')$

shows

$Array.present\ heap'\ (Ref.get\ heap'\ m_ref1)\ Array.present\ heap'\ (Ref.get$
 $heap'\ m_ref2)$
<proof>

lemma *init_state_neq*:

assumes

execute (init_state k1 k2) heap = Some ((k_ref1, k_ref2, m_ref1, m_ref2), heap')

shows

Ref.get heap' m_ref1 == Ref.get heap' m_ref2
<proof>

lemma *present_alloc_get*:

Array.get heap' a = Array.get heap a

if *Array.alloc xs heap = (a', heap')* *Array.present heap a*
<proof>

lemma *init_state_length*:

assumes

execute (init_state k1 k2) heap = Some ((k_ref1, k_ref2, m_ref1, m_ref2), heap')

shows

Array.length heap' (Ref.get heap' m_ref1) = size
Array.length heap' (Ref.get heap' m_ref2) = size
<proof>

context

fixes *key1 :: 'k ⇒ ('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  $\wedge$  Array.length heap m2 = size
     $\wedge$  Ref.present heap k_ref1  $\wedge$  Ref.present heap k_ref2
     $\wedge$  Ref.present heap m_ref1  $\wedge$  Ref.present heap m_ref2
     $\wedge$  Array.present heap m1  $\wedge$  Array.present heap m2
     $\wedge$  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'*

\langle proof \rangle

lemma *inv_pair_lengthD1:*

Array.length heap (Ref.get heap m_ref1) = size **if** *inv_pair_weak heap*
 \langle proof \rangle

lemma *inv_pair_lengthD2:*

Array.length heap (Ref.get heap m_ref2) = size **if** *inv_pair_weak heap*
 \langle proof \rangle

lemma *inv_pair_presentD:*

Array.present heap (Ref.get heap m_ref1) \wedge Array.present heap (Ref.get heap m_ref2)

if *inv_pair_weak heap*

\langle proof \rangle

lemma *inv_pair_presentD2:*

Ref.present heap m_ref1 \wedge Ref.present heap m_ref2

Ref.present heap k_ref1 \wedge Ref.present heap k_ref2

if *inv_pair_weak heap*

\langle proof \rangle

lemma *inv_pair_not_eqD:*

Ref.get heap m_ref1 =!!= Ref.get heap m_ref2 if inv_pair_weak heap
<proof>

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)
<proof>

context *assumes injective: injective size to_index*

begin

context

assumes inv_distinct: inv_distinct k_ref1 k_ref2 m_ref1 m_ref2

begin

lemma *disjoint[simp]*:

m_ref1 =!= m_ref2 m_ref1 =!= k_ref1 m_ref1 =!= k_ref2
m_ref2 =!= k_ref1 m_ref2 =!= k_ref2
k_ref1 =!= k_ref2
<proof>

lemmas [simp] = disjoint[THEN noteq_sym]

lemma [simp]:

Array.get (snd (Array.alloc xs heap)) a = Array.get heap a **if** *Array.present heap a*
⟨proof⟩

lemma [simp]:

Ref.get (snd (Array.alloc xs heap)) r = Ref.get heap r **if** *Ref.present heap r*
⟨proof⟩

lemma *alloc_present*:

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

lemma *alloc_present'*:

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

lemma *length_get_upd*[simp]:

length (Array.get (Array.update a i x heap) r) = length (Array.get heap r)
⟨proof⟩

method *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
⟨proof⟩

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 $update1' k v \equiv do \{m \leftarrow !m_ref1; mem_update m k v\}$

definition $update2' k v \equiv do \{m \leftarrow !m_ref2; mem_update m k v\}$

definition $move12' k \equiv do \{$
 $k1 \leftarrow !k_ref1;$
 $m \leftarrow mem_empty;$
 $m1 \leftarrow !m_ref1;$
 $k_ref2 := k1;$
 $k_ref1 := k;$
 $m_ref2 := m1;$
 $m_ref1 := m$
 $\}$

interpretation $heap_mem_defs inv_pair_weak lookup_pair update_pair$
 $\langle proof \rangle$

lemma $rel_state_ofI:$
 $rel_state (=) (state_of m) m$ **if**
 $\forall heap. inv_pair_weak heap \longrightarrow success m heap$
 $lift_p inv_pair_weak m$
 $\langle proof \rangle$

lemma $inv_pair_iff:$
 $inv_pair_weak = inv_pair$
 $\langle proof \rangle$

lemma $lift_p_inv_pairI:$
 $State_Heap.lift_p inv_pair m$ **if** $State_Heap.lift_p inv_pair_weak m$
 $\langle proof \rangle$

lemma $lift_p_success:$
 $State_Heap.lift_p inv_pair_weak m$
 if $DP_CRelVS.lift_p inv_pair_weak (state_of m) \forall heap. inv_pair_weak$
 $heap \longrightarrow success m heap$
 $\langle proof \rangle$

lemma $rel_state_ofI2:$
 $rel_state (=) (state_of m) m$ **if**
 $\forall heap. inv_pair_weak heap \longrightarrow success m heap$
 $DP_CRelVS.lift_p inv_pair_weak (state_of m)$
 $\langle proof \rangle$

```

context
  includes lifting_syntax
begin

lemma [transfer_rule]:
  ((=) ==> rel_state (=)) move12 move12'
  <proof>

lemma [transfer_rule]:
  ((=) ==> rel_state (rel_option (=))) lookup1 mem_lookup1
  <proof>

lemma [transfer_rule]:
  ((=) ==> rel_state (rel_option (=))) lookup2 mem_lookup2
  <proof>

lemma [transfer_rule]:
  rel_state (=) get_k1 get_k1'
  <proof>

lemma [transfer_rule]:
  rel_state (=) get_k2 get_k2'
  <proof>

lemma [transfer_rule]:
  ((=) ==> (=) ==> rel_state (=)) update1 update1'
  <proof>

lemma [transfer_rule]:
  ((=) ==> (=) ==> rel_state (=)) update2 update2'
  <proof>

lemma [transfer_rule]:
  ((=) ==> rel_state (rel_option (=))) lookup1 mem_lookup1
  <proof>

lemma rel_state_lookup:
  ((=) ==> rel_state (=)) pair.lookup_pair lookup_pair
  <proof>

lemma rel_state_update:
  ((=) ==> (=) ==> rel_state (=)) pair.update_pair update_pair
  <proof>

```

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

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

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

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

context

assumes *keys*: $\forall k k'. \text{key1 } k = \text{key1 } k' \wedge \text{key2 } k = \text{key2 } k' \longrightarrow k = k'$

begin

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

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

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

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

lemmas *heap_correct_pairI* = *heap_correct_axioms*

lemma *mem_rel_state_resultD*:
result_of *m* *heap* = *fst* (*run_state* *m' heap*) **if** *mem.rel_state* (=) *m' m*

pair.inv_pair heap
⟨proof⟩

lemma *map_of_heap_eq*:
mem.map_of_heap heap = pair.pair.map_of heap **if** *pair.inv_pair heap*
⟨proof⟩

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
⟨proof⟩

lemma *init_state_empty2*:
pair.mem2.map_of heap' k = None
⟨proof⟩

lemma
shows *init_state_k1: result_of (!k_ref1) heap' = k1*
and *init_state_k2: result_of (!k_ref2) heap' = k2*
⟨proof⟩

context

assumes *neq: k1 ≠ k2*
begin

lemma *init_state_inv'*:
pair.inv_pair heap'
⟨proof⟩

lemma *init_state_empty*:
pair.pair.map_of heap' ⊆_m Map.empty
⟨proof⟩

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

lemmas *heap_correct_empty_pairI = heap_correct_empty_axioms*

```

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

interpretation  $dp\_consistency\_heap\_empty$ 
  pair.inv_pair update_pair lookup_pair dp heap'
  <proof>

lemmas  $consistent\_empty\_pairI = dp\_consistency\_heap\_empty\_axioms$ 

end

end

end

end

end

end

end

end

end

end

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

```

$\langle ML \rangle$

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* :: $'a \Rightarrow \text{bool}$ **and** *nxt* :: $'a \Rightarrow 'a$

begin

definition

iter_state $f \equiv$

wfrec

{(*nxt* x , x) | x . *cnt* x }

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

())

definition

iterator_to_list \equiv

wfrec {(*nxt* x , x) | x . *cnt* x } (λ *rec* x . *if cnt* x *then* $x \#$ *rec* (*nxt* x) *else*

[])

end

locale *iterator* = *iterator_defs* +

fixes *sizef* :: $'a \Rightarrow \text{nat}$

assumes *terminating*:

finite { x . *cnt* x } $\forall x$. *cnt* $x \longrightarrow \text{sizef } x < \text{sizef } (\text{nxt } x)$

begin

lemma *admissible*:

adm_wf

{(*nxt* x , x) | x . *cnt* x }

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

`()`
`<proof>`

lemma *wellfounded*:
`wf {(next x, x) | x. cnt x} (is wf ?S)`
`<proof>`

lemma *iter_state_unfold*:
`iter_state f x = (if cnt x then do {f x; iter_state f (next x)} else State_Monad.return`
`()`
`<proof>`

lemma *iterator_to_list_unfold*:
`iterator_to_list x = (if cnt x then x # iterator_to_list (next x) else [])`
`<proof>`

lemma *iter_state_iterate_state*:
`iter_state f x = iterate_state f (iterator_to_list x)`
`<proof>`

end

context *dp_consistency*
begin

context
includes *lifting_syntax*
begin

lemma *crel_vs_iterate_state*:
`crel_vs (=) () (iterate_state f xs) if ((=) ==>T R) g f`
`<proof>`

lemma *crel_vs_bind_ignore*:
`crel_vs R a (do {d; b}) if crel_vs R a b crel_vs S c d`
`<proof>`

lemma *crel_vs_iterate_and_compute*:
assumes `((=) ==>T R) g f`
shows `crel_vs R (g x) (do {iterate_state f xs; f x})`
`<proof>`

end

end

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

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

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

end

locale *dp_consistency_iterator_empty* =
 dp_consistency_iterator + dp_consistency_empty
begin

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

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

end

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

context

```

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

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

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

end

end

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

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

lemma (in iterator) iter_heap_unfold:
  iter_heap  $f\ x = (\text{if } cnt\ x \text{ then do } \{f\ x; \text{iter\_heap } f\ (nxt\ x)\} \text{ else return } ())$ 
   $\langle \text{proof} \rangle$ 

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

context
includes lifting_syntax
begin

```

term *iter_heap*

term *crel_vs*

lemma *crel_vs_iterate_state*:

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

lemma *crel_vs_bind_ignore*:

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

lemma *crel_vs_iter_and_compute*:

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

lemma *consistent_DP_iter_and_compute*:

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

end

end

end

2.7 Setup for the Heap Monad

theory *Solve_Cong*

imports *Main HOL-Eisbach.Eisbach*

begin

Method for solving trivial equalities with congruence reasoning

named_theorems *cong_rules*

method *solve_cong* **methods** *solve* =

rule *HOL.refl* |
rule *cong_rules*; *solve_cong* *solve* |
solve; *fail*

end

```

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

  context includes heap_monad_syntax begin

  thm if_cong
  lemma ifT_cong:
    assumes  $b = c \ c \implies x = u \ \neg c \implies y = v$ 
    shows Heap_Monad_Ext.if_T  $\langle b \rangle x y = \text{Heap\_Monad\_Ext.if\_T } \langle c \rangle u v$ 
     $\langle \text{proof} \rangle$ 

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

  lemmas [fundef_cong] =
    return_app_return_cong
    ifT_cong
  end

  memoize_fun comp_T: comp monadifies (heap) comp_def
  thm comp_T'.simps
  lemma (in dp_consistency_heap) shows comp_T_transfer[transfer_rule]:
    crel_vs ((R1 ==>_T R2) ==>_T (R0 ==>_T R1) ==>_T (R0 ==>_T
    R2)) comp comp_T
     $\langle \text{proof} \rangle$ 

  memoize_fun map_T: map monadifies (heap) list.map
  lemma (in dp_consistency_heap) map_T_transfer[transfer_rule]:
    crel_vs ((R0 ==>_T R1) ==>_T list_all2 R0 ==>_T list_all2 R1)
    map map_T
     $\langle \text{proof} \rangle$ 

  memoize_fun fold_T: fold monadifies (heap) fold.simps
  lemma (in dp_consistency_heap) fold_T_transfer[transfer_rule]:
    crel_vs ((R0 ==>_T R1 ==>_T R1) ==>_T list_all2 R0 ==>_T R1
    ==>_T R1) fold fold_T
     $\langle \text{proof} \rangle$ 

```

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 $\text{map}_T . \langle f \rangle . \langle xs \rangle = \text{map}_T . \langle g \rangle . \langle ys \rangle$

<proof>

thm *fold_cong*

lemma *foldT_cong*:

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

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

<proof>

lemma *abs_unit_cong*:

assumes $x = y$

shows $(\lambda_. \text{unit}. x) = (\lambda_. y)$

<proof>

lemma *arg_cong4*:

$f a b c d = f a' b' c' d'$ **if** $a = a' b = b' c = c' d = d'$

<proof>

lemmas [*fundef_cong*, *cong_rules*] =

return_app_return_cong

ifT_cong

mapT_cong

foldT_cong

abs_unit_cong

lemmas [*cong_rules*] =

arg_cong4[**where** $f = \text{heap_mem_defs.checkmem}$]

arg_cong2[**where** $f = \text{fun_app_lifted}$]

end

context *dp_consistency_heap* **begin**

context includes *lifting_syntax heap_monad_syntax* **begin**

named_theorems *dp_match_rule*

thm *if_cong*

lemma *ifT_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 $Rel (crel_vs R) (if (Wrap b) then x else y) (Heap_Monad_Ext.if_T \langle c \rangle x_T y_T)$
 $\langle proof \rangle$

lemma *map_T_cong2*:

assumes
 $is_equality R$
 $Rel R xs ys$
 $\bigwedge x. x \in set\ ys \implies Rel (crel_vs S) (f x) (f_T' x)$
shows $Rel (crel_vs (list_all2 S)) (App (App map (Wrap f)) (Wrap xs)) (map_T . \langle f_T' \rangle . \langle ys \rangle)$
 $\langle proof \rangle$

lemma *fold_T_cong2*:

assumes
 $is_equality R$
 $Rel R xs ys$
 $\bigwedge x. x \in set\ ys \implies Rel (crel_vs (S ==> crel_vs S)) (f x) (f_T' x)$
shows
 $Rel (crel_vs (S ==> crel_vs S)) (fold f xs) (fold_T . \langle f_T' \rangle . \langle ys \rangle)$
 $\langle proof \rangle$

lemma *refl2*:

$is_equality R \implies Rel R x x$
 $\langle proof \rangle$

lemma *rel_fun2*:

assumes $is_equality R0 \bigwedge x. Rel R1 (f x) (g x)$
shows $Rel (rel_fun R0 R1) f g$
 $\langle proof \rangle$

lemma *crel_vs_return_app_return*:

assumes $Rel R (f x) (g x)$
shows $Rel R (App (Wrap f) (Wrap x)) (\langle g \rangle . \langle x \rangle)$
 $\langle proof \rangle$

thm *option.case_cong[no_vars]*

lemma *option_case_cong'*:
 $Rel (=) option' option \implies$
 $(option = None \implies Rel R f1 g1) \implies$
 $(\bigwedge x2. option = Some x2 \implies Rel R (f2 x2) (g2 x2)) \implies$
 $Rel R (case option' of None \Rightarrow f1 \mid Some x2 \Rightarrow f2 x2)$

(*case option of None* \Rightarrow *g1* | *Some x2* \Rightarrow *g2 x2*)
<proof>

thm *prod.case_cong[no_vars]*

lemma *prod_case_cong'*: **fixes** *prod prod'* **shows**

Rel (=) prod prod' \Longrightarrow

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

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

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

<proof>

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

lemmas [*dp_match_rule*] =

crel_vs_return_app_return

lemmas [*dp_match_rule*] =

map_T_cong2

fold_T_cong2

if_T_cong2

lemmas [*dp_match_rule*] =

crel_vs_return

crel_vs_fun_app

refl2

rel_fun2

end

end

2.7.1 More Heap

lemma *execute_heap_ofD*:

heap_of c h = h' if execute c h = Some (v, h')

<proof>

lemma *execute_result_ofD*:

result_of c h = v if execute c h = Some (v, h')

<proof>

locale *heap_correct_init_defs* =


```

fixes  $P :: 'm \Rightarrow \text{heap} \Rightarrow \text{bool}$ 
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 $update = update\ init_mem$
 $\langle proof \rangle$

interpretation $consistent: dp_consistency_heap_empty$
where $P = P\ init_mem$
and $lookup = lookup\ init_mem$
and $update = update\ init_mem$
and $empty = heap_of\ init\ Heap.empty$
 $\langle proof \rangle$

lemma $memoized_empty:$
 $dp\ x = result_of\ (init\ \gg\ (\lambda mem. dp_T\ mem\ x))\ Heap.empty$
if $consistentDP\ (dp_T\ (result_of\ init\ Heap.empty))$
 $\langle proof \rangle$

end

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

sublocale $dp_consistency_heap$
where $P = P\ init_mem$
and $lookup = lookup\ init_mem$
and $update = update\ init_mem$
 $\langle proof \rangle$

definition $init_mem = result_of\ init\ Heap.empty$

interpretation $consistent: dp_consistency_heap_empty$
where $P = P\ init_mem$
and $lookup = lookup\ init_mem$
and $update = update\ init_mem$
and $empty = heap_of\ init\ Heap.empty$
 $\langle proof \rangle$

```

lemma memoized_empty:
  dp x = result_of (init  $\gg$  ( $\lambda$ mem. dpT mem x)) Heap.empty
  if consistentDP init_mem (dpT (result_of init Heap.empty))
  ⟨proof⟩

end

locale dp_consistency_new =
  fixes dp :: 'k  $\Rightarrow$  'v
  fixes P :: 'm  $\Rightarrow$  heap  $\Rightarrow$  bool
    and lookup :: 'm  $\Rightarrow$  'k  $\Rightarrow$  'v option Heap
    and update :: 'm  $\Rightarrow$  'k  $\Rightarrow$  'v  $\Rightarrow$  unit Heap
    and init
  assumes
    success: success init Heap.empty
  assumes
    inv_init:  $\bigwedge$  empty heap. execute init Heap.empty = Some (empty, heap)
 $\implies$  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
  ⟨proof⟩

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

end

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

```

```

and mem :: 'm
assumes mem_is_init: mem = result_of init Heap.empty
assumes
  success: success init Heap.empty
assumes
  inv_init:  $\bigwedge$  empty heap. execute init Heap.empty = Some (empty, heap)
 $\implies$  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
  <proof>

lemma memoized_empty:
  dp x = result_of (init  $\gg$  (lambda mem. dp_T mem x)) Heap.empty
if consistentDP (dp_T (result_of init Heap.empty))
  <proof>

end

locale dp_consistency_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
  <proof>

```

```

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
   $\langle$ proof $\rangle$ 

thm memoized_empty

end

locale dp_consistency_heap_array =
  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_heap_init
  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
   $\langle$ proof $\rangle$ 

end

locale dp_consistency_heap_array_pair' =
  fixes size :: nat
  fixes key1 :: 'k  $\Rightarrow$  ('k1 :: heap) and key2 :: 'k  $\Rightarrow$  'k2 :: heap
  and to_index :: 'k2  $\Rightarrow$  nat
  and dp :: 'k  $\Rightarrow$  'v::heap

```

```

and  $k1\ k2 :: 'k1$ 
and  $mem :: ('k1\ ref \times$ 
          $'k1\ ref \times$ 
          $'v\ option\ array\ ref \times$ 
          $'v\ option\ array\ ref)$ 
assumes  $mem\_is\_init: mem = result\_of\ (init\_state\ size\ k1\ k2)\ Heap.empty$ 
assumes  $injective: injective\ size\ to\_index$ 
         and  $keys\_injective: \forall k\ k'. key1\ k = key1\ k' \wedge key2\ k = key2\ k' \longrightarrow k$ 
          $= k'$ 
         and  $keys\_neq: k1 \neq k2$ 
begin

```

definition

```

 $inv\_pair' = (\lambda (k\_ref1, k\_ref2, m\_ref1, m\_ref2).$ 
   $pair\_mem\_defs.inv\_pair\ (lookup1\ size\ to\_index\ m\_ref1)$ 
   $(lookup2\ size\ to\_index\ m\_ref2)\ (get\_k1\ k\_ref1)$ 
   $(get\_k2\ k\_ref2)$ 
   $(inv\_pair\_weak\ size\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2)\ key1\ key2)$ 

```

sublocale $dp_consistency_new'$

```

where  $P = inv\_pair'$ 
and  $lookup = \lambda (k\_ref1, k\_ref2, m\_ref1, m\_ref2).$ 
   $lookup\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2$ 
and  $update = \lambda (k\_ref1, k\_ref2, m\_ref1, m\_ref2).$ 
   $update\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2$ 
and  $init = init\_state\ size\ k1\ k2$ 
 $\langle proof \rangle$ 

```

end

locale $dp_consistency_heap_array_pair_iterator =$

```

 $dp\_consistency\_heap\_array\_pair'$  where  $dp = dp + iterator$  where  $cnt$ 
 $= cnt$ 

```

```

for  $dp :: 'k \Rightarrow 'v::heap$  and  $cnt :: 'k \Rightarrow bool$ 

```

begin

sublocale $dp_consistency_iterator_heap$

```

where  $P = inv\_pair'\ mem$ 
and  $update = (case\ mem\ of$ 
   $(k\_ref1, k\_ref2, m\_ref1, m\_ref2) \Rightarrow$ 
   $update\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2)$ 
and  $lookup = (case\ mem\ of$ 
   $(k\_ref1, k\_ref2, m\_ref1, m\_ref2) \Rightarrow$ 
   $lookup\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2)$ 

```

```

    <proof>

end

locale dp_consistency_heap_array_pair =
  fixes size :: nat
  fixes key1 :: 'k  $\Rightarrow$  ('k1 :: heap) and key2 :: 'k  $\Rightarrow$  'k2 :: heap
  and to_index :: 'k2  $\Rightarrow$  nat
  and dp :: 'k  $\Rightarrow$  'v::heap
  and k1 k2 :: 'k1
  assumes injective: injective size to_index
  and keys_injective:  $\forall k k'. \text{key1 } k = \text{key1 } k' \wedge \text{key2 } k = \text{key2 } k' \longrightarrow k = k'$ 
  and keys_neq: k1  $\neq$  k2
begin

definition
  inv_pair' = ( $\lambda$  (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= $\lambda$  (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= $\lambda$  (k_ref1, k_ref2, m_ref1, m_ref2).
    update_pair size to_index key1 key2 m_ref1 m_ref2 k_ref1 k_ref2
  and init=init_state size k1 k2
  <proof>

end

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  $b = c \implies x = u \wedge c \implies y = v$ 
  shows  $\text{State\_Monad\_Ext.if}_T \langle b \rangle x y = \text{State\_Monad\_Ext.if}_T \langle c \rangle u v$ 
  <proof>

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

lemmas [fundef_cong] =
  return_app_return_cong
  ifT_cong
end

memoize_fun comp_T: comp monadifies (state) comp_def
lemma (in dp_consistency) comp_T_transfer[transfer_rule]:
  crel_vs (( $R1 \implies_T R2$ )  $\implies_T (R0 \implies_T R1) \implies_T (R0 \implies_T R2$ ))
  comp comp_T
  <proof>

memoize_fun map_T: map monadifies (state) list.map
lemma (in dp_consistency) map_T_transfer[transfer_rule]:
  crel_vs (( $R0 \implies_T R1$ )  $\implies_T \text{list\_all2 } R0 \implies_T \text{list\_all2 } R1$ )
  map map_T
  <proof>

memoize_fun fold_T: fold monadifies (state) fold_simps
lemma (in dp_consistency) fold_T_transfer[transfer_rule]:
  crel_vs (( $R0 \implies_T R1 \implies_T R1$ )  $\implies_T \text{list\_all2 } R0 \implies_T R1$ 
 $\implies_T R1$ ) fold fold_T
  <proof>
```


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 $\text{map}_T . \langle f \rangle . \langle xs \rangle = \text{map}_T . \langle g \rangle . \langle ys \rangle$

$\langle \text{proof} \rangle$

thm *fold_cong*

lemma *foldT_cong*:

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

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

$\langle \text{proof} \rangle$

lemma *abs_unit_cong*:

assumes $x = y$

shows $(\lambda_. :: \text{unit}. x) = (\lambda_. y)$

$\langle \text{proof} \rangle$

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 $\text{Rel } (=) b c c \implies \text{Rel } (\text{crel_vs } R) x x_T \neg c \implies \text{Rel } (\text{crel_vs } R) y y_T$

shows $\text{Rel } (\text{crel_vs } R) (\text{if } (\text{Wrap } b) \text{ then } x \text{ else } y) (\text{State_Monad_Ext.if}_T \langle c \rangle x_T y_T)$

$\langle \text{proof} \rangle$

lemma *mapT_cong2*:

assumes

is_equality R

$Rel\ R\ xs\ ys$
 $\bigwedge x. x \in set\ ys \implies Rel\ (crel_vs\ S)\ (f\ x)\ (f_T'\ x)$
shows $Rel\ (crel_vs\ (list_all2\ S))\ (App\ (App\ map\ (Wrap\ f))\ (Wrap\ xs))$
 $(map_T\ .\ \langle f_T' \rangle .\ \langle ys \rangle)$
 $\langle proof \rangle$

lemma *fold_T_cong2*:

assumes

is_equality R

$Rel\ R\ xs\ ys$

$\bigwedge x. x \in set\ ys \implies Rel\ (crel_vs\ (S\ ==>\ crel_vs\ S))\ (f\ x)\ (f_T'\ x)$

shows

$Rel\ (crel_vs\ (S\ ==>\ crel_vs\ S))\ (fold\ f\ xs)\ (fold_T\ .\ \langle f_T' \rangle .\ \langle ys \rangle)$

$\langle proof \rangle$

lemma *refl2*:

is_equality $R \implies Rel\ R\ x\ x$

$\langle proof \rangle$

lemma *rel_fun2*:

assumes *is_equality* $R0 \bigwedge x. Rel\ R1\ (f\ x)\ (g\ x)$

shows $Rel\ (rel_fun\ R0\ R1)\ f\ g$

$\langle proof \rangle$

lemma *crel_vs_return_app_return*:

assumes $Rel\ R\ (f\ x)\ (g\ x)$

shows $Rel\ R\ (App\ (Wrap\ f)\ (Wrap\ x))\ (\langle g \rangle .\ \langle x \rangle)$

$\langle proof \rangle$

thm *option.case_cong[no_vars]*

lemma *option_case_cong'*:

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

$(option = None \implies Rel\ R\ f1\ g1) \implies$

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

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

$(case\ option\ of\ None \Rightarrow g1\ | \ Some\ x2 \Rightarrow g2\ x2)$

$\langle proof \rangle$

thm *prod.case_cong[no_vars]*

lemma *prod_case_cong'*: **fixes** *prod prod'* **shows**

$Rel\ (=)\ prod\ prod' \implies$

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

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

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

<proof>

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 | Suc x2 \Rightarrow f2 x2) (case nat' of 0 \Rightarrow g1 | Suc x2

\Rightarrow g2 x2)

<proof>

lemmas [*dp_match_rule*] =

prod_case_cong'

option_case_cong'

nat_case_cong'

lemmas [*dp_match_rule*] =

crel_vs_return_app_return

lemmas [*dp_match_rule*] =

map_T_cong2

fold_T_cong2

if_T_cong2

lemmas [*dp_match_rule*] =

crel_vs_return

crel_vs_fun_app

refl2

rel_fun2

end

end

2.8.1 Code Setup

lemmas [*code_unfold*] =

state_mem_defs.checkmem_checkmem'[*symmetric*]

state_mem_defs.checkmem'_def

map_T_def

end

3 Examples

3.1 Misc

theory *Example_Misc*

imports

Main

HOL-Library.Extended

../state_monad/State_Main

begin

Lists fun *min_list* :: 'a::ord list \Rightarrow 'a **where**

min_list (x # xs) = (case xs of [] \Rightarrow x | _ \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

<proof>

lemma *min_list_fold*:

min_list (x # xs) = *fold min xs x* **for** x :: 'a :: linorder

<proof>

lemma *induct_list012*:

$\llbracket P []; \bigwedge x. P [x]; \bigwedge x y zs. P (y \# zs) \implies P (x \# y \# zs) \rrbracket \implies P xs$
<proof>

lemma *min_list_Min*: $xs \neq [] \implies \text{min_list } xs = \text{Min } (\text{set } xs)$

<proof>

Extended Data Type lemma *Pinf_add_right[simp]*:

$\infty + x = \infty$

<proof>

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)
  <proof>

```

```

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  $\Rightarrow$  trace (STR "Hit " + trace_key param) (return x)
      | None  $\Rightarrow$  trace (STR "Miss " + trace_key param)
        Heap_Monad.bind (calc ()) ( $\lambda$  x.
          Heap_Monad.bind (update param x) ( $\lambda$  _.
            return x
          )
        )
    )

```

```

lemma (in heap_mem_defs) checkmem_checkmem_trace:
  checkmem param calc = checkmem_trace trace_key param ( $\lambda$ _. calc)
  <proof>

```

```
definition nat_to_string :: nat  $\Rightarrow$  String.literal where  
  nat_to_string x = String.implode (show x)
```

```
definition nat_pair_to_string :: nat  $\times$  nat  $\Rightarrow$  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
```

```
 $\langle ML \rangle$ 
```

```
end
```

3.2 The Bellman-Ford Algorithm

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

3.2.1 Misc

```
lemma nat_le_cases:  
  fixes n :: nat
```

```

assumes  $i \leq n$ 
obtains  $i < n \mid i = n$ 
 $\langle proof \rangle$ 

context dp_consistency_iterator
begin

lemma crel_vs_iterate_state:
  crel_vs (=) () (iter_state  $f$   $x$ ) if ((=)  $\implies_T R$ )  $g$   $f$ 
   $\langle proof \rangle$ 

lemma consistent_crel_vs_iterate_state:
  crel_vs (=) () (iter_state  $f$   $x$ ) if consistentDP  $f$ 
   $\langle proof \rangle$ 

end

instance extended :: (countable) countable
 $\langle proof \rangle$ 

instance extended :: (heap) heap  $\langle proof \rangle$ 

instantiation extended :: (conditionally_complete_lattice) complete_lattice
begin

definition
  Inf  $A = ($ 
     $if$   $A = \{\}$   $\vee$   $A = \{\infty\}$   $then$   $\infty$ 
     $else$   $if$   $-\infty \in A \vee \neg bdd\_below$  (Fin  $- ' A$ )  $then$   $-\infty$ 
     $else$  Fin (Inf (Fin  $- ' A$ )))
   $\rangle$ 

definition
  Sup  $A = ($ 
     $if$   $A = \{\}$   $\vee$   $A = \{-\infty\}$   $then$   $-\infty$ 
     $else$   $if$   $\infty \in A \vee \neg bdd\_above$  (Fin  $- ' A$ )  $then$   $\infty$ 
     $else$  Fin (Sup (Fin  $- ' A$ )))
   $\rangle$ 

instance
 $\langle proof \rangle$ 

end

instance extended :: ( $\{conditionally\_complete\_lattice, linorder\}$ ) complete_linorder
 $\langle proof \rangle$ 

```

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

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

lemmas *Sup_int_in = Sup_int[THEN conjunct1]*

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

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

lemma *finite_lists_length_le1*: $\text{finite } \{xs. \text{length } xs \leq i \wedge \text{set } xs \subseteq \{0..(n::\text{nat})\}\}$
for i
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemmas [*simp*] =
finite_setcompr_eq_image finite_lists_length_le2[simplified] finite_lists_length_le1

lemma *get_return*:
 $\text{run_state } (\text{State_Monad.bind } \text{State_Monad.get } (\lambda m. \text{State_Monad.return } (f\ m)))\ m = (f\ m, m)$
 ⟨*proof*⟩

lemma *list_pidgeonhole*:

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

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** $as = v \# as'\ cs = cs' @ [t]\ ys = as' @ a \# bs @ a \# cs'$
 $\langle proof \rangle$

lemma *le_add_same_cancel1*:

$a + b \geq a \iff b \geq 0$ **if** $a < \infty\ -\infty < a$ **for** $a\ b :: int\ extended$
 $\langle proof \rangle$

lemma *add_gt_minfI*:

assumes $-\infty < a\ -\infty < b$
shows $-\infty < a + b$
 $\langle proof \rangle$

lemma *add_lt_infI*:

assumes $a < \infty\ b < \infty$
shows $a + b < \infty$
 $\langle proof \rangle$

lemma *sum_list_not_infI*:

$sum_list\ xs < \infty$ **if** $\forall x \in set\ xs.\ x < \infty$ **for** $xs :: int\ extended\ list$
 $\langle proof \rangle$

lemma *sum_list_not_minfI*:

$sum_list\ xs > -\infty$ **if** $\forall x \in set\ xs.\ x > -\infty$ **for** $xs :: int\ extended\ list$
 $\langle proof \rangle$

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 $weight_alt_def'$:

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

lemma $weight_alt_def$:

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

lemma $weight_append$:

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

lemma OPT_0 :

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

3.2.3 Functional Correctness

lemma OPT_cases :

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

lemma OPT_Suc :

$OPT\ (Suc\ i)\ v = min\ (OPT\ i\ v)\ (Min\ \{OPT\ i\ w + W\ v\ w\ | w.\ w \leq n\})$

```
(is ?lhs = ?rhs)
  if t ≤ n
  ⟨proof⟩
```

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

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

```
lemma bf_correct:
  OPT i j = bf i j if ⟨t ≤ n⟩
  ⟨proof⟩
```

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
  ⟨proof⟩
print_theorems
lemmas [code] = bfm.memoized_correct
```

```
interpretation iterator
  λ (x, y). x ≤ n ∧ y ≤ n
  λ (x, y). if y < n then (x, y + 1) else (x + 1, 0)
  λ (x, y). x * (n + 1) + y
  ⟨proof⟩
```

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

```

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

```

definition

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

lemma *iter_bf_unfold*[code]:

```

iter_bf = (λ (i, j).
  (if i ≤ n ∧ j ≤ n
    then do {
      bf_m' i j;
      iter_bf (if j < n then (i, j + 1) else (i + 1, 0))
    }
    else State_Monad.return ()))
⟨proof⟩

```

lemmas *bf_memoized* = *bf_m.memoized*[OF *bf_m.crel*]

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

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

definition

```

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

```

context

includes *state_monad_syntax*

begin

lemma *bellman_ford_alt_def*:

```

bellman_ford ≡
do {
  _ ← iter_bf (n, n);
  (⟨λxs. ⟨λys. State_Monad.return (if xs = ys then Some xs else None)⟩
  . (State_Main.mapT . ⟨λi. bf_m' (n + 1) i⟩ . ⟨[0..<n+1]⟩))

```

```

    . (State_Main.mapT . ⟨λi. bf_m' n i⟩      . ⟨[0..<n+1]⟩)
  }
⟨proof⟩

```

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
⟨proof⟩

```

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
⟨proof⟩

```

lemma [intro]:

```

  dp_consistency_heap_array_pair_iterator (n + 1) fst snd id 1 0 mem
  (λ (x, y). if y < n then (x, y + 1) else (x + 1, 0))
  (λ (x, y). x * (n + 1) + y)
  (λ (x, y). x ≤ n ∧ y ≤ n)
⟨proof⟩

```

memoize_fun bf_h: 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 = λ (x, y). x ≤ n ∧ y ≤ n
  and nxt = λ (x :: nat, y). if y < n then (x, y + 1) else (x + 1, 0)
  and sizef = λ (x, y). x * (n + 1) + y
monadifies (heap) bf.simps

```

memoize_correct

<proof>

lemmas *memoized_empty* = *bf_h.memoized_empty*[*OF bf_h.consistent_DP_iter_and_compute*[*OF bf_h.crel*]]

lemmas *iter_heap_unfold* = *iter_heap_unfold*

end

3.2.6 Detecting Negative Cycles

definition

shortest *v* = (
 Inf (
 {*weight* (*v* # *xs* @ [*t*]) | *xs. set xs* ⊆ {0..*n*}} ∪
 {if *t* = *v* then 0 else ∞}
)
)

definition

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

definition

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

definition

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

lemma *fold_sum_aux'*:

assumes ∃ *u* ∈ *set (a # xs)*. ∃ *v* ∈ *set (xs @ [b])*. *f v* + *W u v* ≥ *f u*

shows *sum_list* (*map f (a # xs)*) ≤ *sum_list* (*map f (xs @ [b])*) + *weight* (*a # xs @ [b]*)

<proof>

lemma *fold_sum_aux*:

assumes ∃ *u* ∈ *set (a # xs)*. ∃ *v* ∈ *set (a # xs)*. *f v* + *W u v* ≥ *f u*

shows *sum_list* (*map f (a # xs @ [a])*) ≤ *sum_list* (*map f (a # xs @ [a])*) + *weight* (*a # xs @ [a]*)

<proof>

context

begin

private definition $is_path2\ xs \equiv weight\ xs < \infty$

private lemma $is_path2_remove_cycle$:
 assumes $is_path2\ (as\ @\ a\ \#\ bs\ @\ a\ \#\ cs)$
 shows $is_path2\ (as\ @\ a\ \#\ cs)$
 $\langle proof \rangle$ **lemma** is_path_eq :
 $is_path\ xs \longleftrightarrow is_path2\ (xs\ @\ [t])$
 $\langle proof \rangle$

lemma $is_path_remove_cycle$:
 assumes $is_path\ (as\ @\ a\ \#\ bs\ @\ a\ \#\ cs)$
 shows $is_path\ (as\ @\ a\ \#\ cs)$
 $\langle proof \rangle$

lemma $is_path_remove_cycle2$:
 assumes $is_path\ (as\ @\ t\ \#\ cs)$
 shows $is_path\ as$
 $\langle proof \rangle$

end

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

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

lemma $OPT_sink_le_0$:
 $OPT\ i\ t \leq 0$
 $\langle proof \rangle$

lemma $is_path_appendD$:
 assumes $is_path\ (as\ @\ a\ \#\ bs)$
 shows $is_path\ (a\ \#\ bs)$
 $\langle proof \rangle$

lemma $has_negative_cycleI$:
 assumes $set\ (a\ \#\ xs\ @\ ys) \subseteq \{0..n\}\ weight\ (a\ \#\ xs\ @\ [a]) < 0\ is_path\ (a\ \#\ ys)$
 shows $has_negative_cycle$

<proof>

lemma *OPT_cases2*:

obtains (*path*) *xs* **where**

$v \neq t$ *OPT* *i v* $\neq \infty$ *OPT* *i v* = *weight* (*v # xs @ [t]*) *length xs* + 1 $\leq i$

set xs $\subseteq \{0..n\}$

| (*unreachable*) $v \neq t$ *OPT* *i v* = ∞

| (*sink*) $v = t$ *OPT* *i v* ≤ 0

<proof>

lemma *shortest_le_OPT*:

assumes $v \leq n$

shows *shortest* $v \leq$ *OPT* *i v*

<proof>

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* $\neq []$

<proof>

lemma *OPT_not_minfI*:

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

<proof>

theorem *detects_cycle*:

assumes *has_negative_cycle*

shows $\exists i \leq n. OPT\ (n + 1)\ i < OPT\ n\ i$

<proof>

corollary *bf_detects_cycle*:

assumes *has_negative_cycle*

shows $\exists i \leq n. bf\ (n + 1)\ i < bf\ n\ i$

<proof>

lemma *shortest_cases*:

assumes $v \leq n$

obtains (*path*) *xs* **where** *shortest* $v =$ *weight* (*v # xs @ [t]*) *set xs* $\subseteq \{0..n\}$

| (*sink*) $v = t$ *shortest* $v = 0$

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

lemma *simple_paths*:

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

theorem *shorter_than_OPT_n_has_negative_cycle*:

assumes *shortest* $v < \text{OPT } n$ $v \leq n$
shows *has_negative_cycle*
 $\langle \text{proof} \rangle$

corollary *detects_cycle_has_negative_cycle*:

assumes $\text{OPT } (n + 1) v < \text{OPT } n$ $v \leq n$
shows *has_negative_cycle*
 $\langle \text{proof} \rangle$

corollary *bellman_ford_detects_cycle*:

has_negative_cycle $\longleftrightarrow (\exists v \leq n. \text{OPT } (n + 1) v < \text{OPT } n v)$
 $\langle \text{proof} \rangle$

corollary *bellman_ford_shortest_paths*:

assumes $\neg \text{has_negative_cycle}$
shows $\forall v \leq n. \text{bf } n v = \text{shortest } v$
 $\langle \text{proof} \rangle$

lemma *OPT_mono*:

$\text{OPT } m v \leq \text{OPT } n v$ **if** $\langle v \leq n \rangle \langle n \leq m \rangle$
 $\langle \text{proof} \rangle$

corollary *bf_fix*:

assumes $\neg \text{has_negative_cycle}$ $m \geq n$
shows $\forall v \leq n. \text{bf } m v = \text{bf } n v$
 $\langle \text{proof} \rangle$

lemma *bellman_ford_correct'*:

$\text{bf}_m.\text{crel_vs } (=)$ (*if has_negative_cycle then None else Some (map shortest [0..<n+1])*) *bellman_ford*

```

⟨proof⟩
  include state_monad_syntax app_syntax
  ⟨proof⟩

```

```

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

```

end

end

end

3.2.7 Extracting an Executable Constant for the Imperative Implementation

```

ground_function (prove_termination) bf_h'_impl: bf_h'.simps

```

```

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

```

definition

```

iter_bf_heap n w t mem = iterator_defs.iter_heap
  (λ(x, y). x ≤ n ∧ y ≤ n)
  (λ(x, y). if y < n then (x, y + 1) else (x + 1, 0))
  (λ(x, y). bf_h'_impl n w t mem x y)

```

lemma iter_bf_heap_unfold[code]:

```

iter_bf_heap n w t mem = (λ (i, j).
  (if i ≤ n ∧ j ≤ n
  then do {
    bf_h'_impl n w t mem i j;
    iter_bf_heap n w t mem (if j < n then (i, j + 1) else (i + 1, 0))
  }
  else Heap_Monad.return ()))
  ⟨proof⟩

```

definition

```

bf_impl n w t i j = do {
  mem ← (init_state (n + 1) (1::nat) (0::nat) ::
    (nat ref × nat ref × int extended option array ref × int extended
option array ref) Heap);
  iter_bf_heap n w t mem (0, 0);
  bf_h'_impl n w t mem i j
}

```

lemma *bf_impl_correct*:

```

bf n w t i j = result_of (bf_impl n w t i j) Heap.empty
⟨proof⟩

```

3.2.8 Test Cases**definition**

```

G1_list = [[(1 :: nat, -6 :: int), (2,4), (3,5)], [(3,10)], [(3,2)], []]

```

definition

```

G2_list = [[(1 :: nat, -6 :: int), (2,4), (3,5)], [(3,10)], [(3,2)], [(0, -5)]]

```

definition

```

G3_list = [[(1 :: nat, -1 :: int), (2,2)], [(2,5), (3,4)], [(3,2), (4,3)], [(2,-2),
(4,2)], []]

```

definition

```

G4_list = [[(1 :: nat, -1 :: int), (2,2)], [(2,5), (3,4)], [(3,2), (4,3)], [(2,-3),
(4,2)], []]

```

definition

```

graph_of a i j = case_option ∞ (Fin o snd) (List.find (λ p. fst p = j) (a
!! i))

```

```

definition test_bf = bf_impl 3 (graph_of (IArray G1_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*⟩

lemma *bottom_up_alt*[*code*]:

```

bf n W t i j =
  fst (run_state
    (iter_bf n W t (0, 0) ≫= (λ_. bf_m' n W t i j))

```

Mapping.empty)
 ⟨*proof*⟩

definition

bf_ia n W t i j = (let *W'* = *graph_of* (*IArray* *W*) in
 fst (*run_state*
 (*iter_bf* *n W' t* (*i, j*) \gg ($\lambda_.$ *bf_m' n W' t i j*))
 Mapping.empty)
)

— Component tests.

lemma

fst (*run_state* (*bf_m' 3* (*graph_of* (*IArray* *G₁_list*))) 3 3 0) *Mapping.empty*)
 = 4

bf 3 (*graph_of* (*IArray* *G₁_list*)) 3 3 0 = 4
 ⟨*proof*⟩

lemma

fst (*run_state* (*bellman_ford 3* (*graph_of* (*IArray* *G₁_list*))) 3) *Mapping.empty*) = *Some* [4, 10, 2, 0]

fst (*run_state* (*bellman_ford 4* (*graph_of* (*IArray* *G₃_list*))) 4) *Mapping.empty*) = *Some* [4, 5, 3, 1, 0]
 ⟨*proof*⟩

lemma

fst (*run_state* (*bellman_ford 3* (*graph_of* (*IArray* *G₂_list*))) 3) *Mapping.empty*) = *None*

fst (*run_state* (*bellman_ford 4* (*graph_of* (*IArray* *G₄_list*))) 4) *Mapping.empty*) = *None*
 ⟨*proof*⟩

end

theory *Heap_Default*

imports

Heap_Main
 ../*Indexing*

begin

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

begin

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

```

sublocale dp_consistency_heap
  where  $P = \lambda \text{heap}. \text{Array.length heap mem} = \text{idx.size}$ 
    and  $\text{lookup} = \text{mem\_lookup idx.size idx.checked\_idx mem}$ 
    and  $\text{update} = \text{mem\_update idx.size idx.checked\_idx mem}$ 
   $\langle \text{proof} \rangle$ 

context
  fixes empty
  assumes  $\text{empty}: \text{map\_of\_heap empty} \subseteq_m \text{Map.empty}$ 
    and  $\text{len}: \text{Array.length empty mem} = \text{idx.size}$ 
begin

interpretation consistent: dp_consistency_heap_empty
  where  $P = \lambda \text{heap}. \text{Array.length heap mem} = \text{idx.size}$ 
    and  $\text{lookup} = \text{mem\_lookup idx.size idx.checked\_idx mem}$ 
    and  $\text{update} = \text{mem\_update idx.size idx.checked\_idx mem}$ 
   $\langle \text{proof} \rangle$ 

lemmas  $\text{memoizedI} = \text{consistent.memoized}$ 
lemmas  $\text{successI} = \text{consistent.memoized\_success}$ 

end

lemma mem_empty_empty:
   $\text{map\_of\_heap (heap\_of (mem\_empty idx.size :: 'v option array Heap)$ 
 $\text{Heap.empty})} \subseteq_m \text{Map.empty}$ 
  if  $\text{mem} = \text{result\_of (mem\_empty idx.size) Heap.empty}$ 
   $\langle \text{proof} \rangle$ 

lemma memoized_empty:
   $\text{dp } x = \text{result\_of ((mem\_empty idx.size :: 'v option array Heap) } \gg=$ 
 $(\lambda \text{mem}. \text{dp}_T \text{ mem } x)) \text{Heap.empty}$ 
  if  $\text{consistentDP (dp}_T \text{ mem) mem} = \text{result\_of (mem\_empty idx.size)}$ 
 $\text{Heap.empty}$ 
   $\langle \text{proof} \rangle$ 

lemma init_success:
   $\text{success ((mem\_empty idx.size :: 'v option array Heap) } \gg= (\lambda \text{mem}. \text{dp}_T$ 
 $\text{mem } x)) \text{Heap.empty}$ 
  if  $\text{consistentDP (dp}_T \text{ mem) mem} = \text{result\_of (mem\_empty idx.size)}$ 
 $\text{Heap.empty}$ 
   $\langle \text{proof} \rangle$ 

end

```

end

3.3 The Knapsack Problem

```
theory Knapsack
  imports
    HOL-Library.Code_Target_Numeral
    ../state_monad/State_Main
    ../heap_monad/Heap_Default
    Example_Misc
begin
```

3.3.1 Definitions

```
context
  fixes w :: nat ⇒ nat
begin
```

```
context
  fixes v :: nat ⇒ nat
begin
```

```
fun knapsack :: nat ⇒ nat ⇒ nat where
  knapsack 0 W = 0 |
  knapsack (Suc i) W = (if W < w (Suc i)
    then knapsack i W
    else max (knapsack i W) (v (Suc i) + knapsack i (W - w (Suc i))))
```

```
no_notation fun_app_lifted (infixl . 999)
```

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

definition

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

lemma *OPT_0*:

```
OPT 0 W = 0
⟨proof⟩
```

3.3.2 Functional Correctness

lemma *Max_add_left*:

$$(x :: nat) + \text{Max}\ S = \text{Max} ((+) x \text{ ' } S) \text{ (is ?A = ?B) if finite } S\ S \neq \{\}$$

<proof>

lemma *OPT_Suc*:

```
OPT (Suc i) W = (  
  if W < w (Suc i)  
  then OPT i W  
  else max(v (Suc i) + OPT i (W - w (Suc i))) (OPT i W)  
) (is ?lhs = ?rhs)  
<proof>
```

theorem *knapsack_correct*:

```
OPT n W = knapsack n W  
<proof>
```

3.3.3 Functional Memoization

```
memoize_fun 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  
<proof>  
print_theorems  
lemmas [code] = knapsackm.memoized_correct
```

3.3.4 Imperative Memoization

```
context fixes
```

```
  mem :: nat option array
```

```
  and n W :: nat
```

```
begin
```

```
memoize_fun knapsackT: 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 knapsackT'simps knapsackT_def
```

end

memoize_correct

<proof>

lemmas *memoized_empty* = *knapsack_T.memoized_empty*

end

Adding Memory Initialization

context

includes *heap_monad_syntax*

notes [*simp del*] = *knapsack_T'.simps*

begin

definition

knapsack_h $\equiv \lambda i j. \text{Heap_Monad.bind } (\text{mem_empty } (i * j)) (\lambda mem. \text{knapsack}_{T'} \text{ mem } i j i j)$

lemmas *memoized_empty'* = *memoized_empty*[
of *mem n W* $\lambda m. \lambda(i,j). \text{knapsack}_{T'} m n W i j$,
OF *knapsack_T.crel*[of *mem n W*], of (*n, W*) **for** *mem n W*
]

lemma *knapsack_heap*:

knapsack n W = *result_of (knapsack_h n W) Heap.empty*
<proof>

end

end

fun *su* :: *nat* \Rightarrow *nat* \Rightarrow *nat* **where**

su 0 *W* = 0 |

su (*Suc i*) *W* = (if *W* < *w* (*Suc i*)

then *su i W*

else *max (su i W) (w (Suc i) + su i (W - w (Suc i)))*)

lemma *su_knapsack*:

su n W = *knapsack w n W*

<proof>

lemma *su_correct*:

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

<proof>

3.3.5 Memoization

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

Generated Definitions

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

Correspondence Proof

```
memoize_correct  
<proof>  
print_theorems  
lemmas [code] = sum.memoized_correct  
  
end
```

3.3.6 Regression Test

definition

```
knapsack_test = (knapsackh ( $\lambda$  i. [2,3,4] ! (i - 1)) ( $\lambda$  i. [2,3,4] ! (i - 1))  
3 8)
```

code_reflect *Test functions knapsack_test*

<ML>

end

theory *Counting_Tiles*

imports

```
HOL-Library.Code_Target_Numeral  
HOL-Library.Product_Lexorder  
HOL-Library.RBT_Mapping  
../state_monad/State_Main  
Example_Misc
```

begin

3.4 A Counting Problem

This formalization contains verified solutions for Project Euler problems

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

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

This is the problem description for #115:

A row measuring n units in length has red blocks with a minimum length of m units placed on it, such that any two red blocks (which are allowed to be different lengths) are separated by at least one black square. Let the fill-count function, $F(m, n)$, represent the number of ways that a row can be filled.

For example, $F(3, 29) = 673135$ and $F(3, 30) = 1089155$.

That is, for $m = 3$, it can be seen that $n = 30$ is the smallest value for which the fill-count function first exceeds one million. In the same way, for $m = 10$, it can be verified that $F(10, 56) = 880711$ and $F(10, 57) = 1148904$, so $n = 57$ is the least value for which the fill-count function first exceeds one million.

For $m = 50$, find the least value of n for which the fill-count function first exceeds one million.

3.4.1 Misc

lemma *lists_of_len_fin1*:

finite (*lists* $A \cap \{l. \text{length } l = n\}$) **if** *finite* A
<proof>

lemma *disjE1*:

$A \vee B \implies (A \implies P) \implies (\neg A \implies B \implies P) \implies P$
<proof>

3.4.2 Problem Specification

Colors

datatype *color* = $R \mid B$

Direct natural definition of a valid line

context

fixes $m :: \text{nat}$

begin

inductive *valid* **where**

valid $[] \mid$
valid $xs \implies \text{valid } (B \# xs) \mid$

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

Definition of the fill-count function

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

3.4.3 Combinatorial Identities

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

inductive $valid'$ **where**

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

lemma $valid_valid'$:

$valid\ l \implies valid'\ l$
 $\langle proof \rangle$

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

lemma $valid'_valid$:

$valid'\ l \implies valid\ l$
 $\langle proof \rangle$

lemma $valid_eq_valid'$:

$valid'\ l = valid\ l$
 $\langle proof \rangle$

Additional Facts on Replicate

lemma $replicate_iff$:

$(\forall\ i < length\ l.\ l\ !\ i = R) \longleftrightarrow (\exists\ n.\ l = replicate\ n\ R)$
 $\langle proof \rangle$

lemma $replicate_iff2$:

$(\forall\ i < n.\ l\ !\ i = R) \longleftrightarrow (\exists\ l'.\ l = replicate\ n\ R\ @\ l')\ \mathbf{if}\ n < length\ l$
 $\langle proof \rangle$

lemma $replicate_Cons_eq$:

$replicate\ n\ x = y\ \#\ ys \longleftrightarrow (\exists\ n'.\ n = Suc\ n' \wedge x = y \wedge replicate\ n'\ x = ys)$
 $\langle proof \rangle$

Main Case Analysis on $@term\ valid$

lemma $valid_split$:

$valid\ l \longleftrightarrow$
 $l = [] \vee$
 $(!0 = B \wedge valid\ (tl\ l)) \vee$
 $length\ l \geq m \wedge (\forall\ i < length\ l.\ l!\ i = R) \vee$
 $(\exists\ j < length\ l.\ j \geq m \wedge (\forall\ i < j.\ l!\ i = R) \wedge l!\ j = B \wedge valid\ (drop$
 $(j + 1)\ l))$
 $\langle proof \rangle$

Base cases

lemma *valid_line_just_B*:
 $valid\ (replicate\ n\ B)$
 $\langle proof \rangle$

lemma *F_base_0_aux*:
 $\{l.\ l = [] \wedge valid\ l\} = \{[]\}$
 $\langle proof \rangle$

lemma *F_base_0*: $F\ 0 = 1$
 $\langle proof \rangle$

lemma *F_base_aux*: $\{l.\ length\ l = n \wedge valid\ l\} = \{replicate\ n\ B\}$ **if** $n > 0$
 $n < m$
 $\langle proof \rangle$

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

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

lemma *F_base_aux_2*: $\{l.\ length\ l = m \wedge valid\ l\} = \{replicate\ m\ R,\ repli-$
 $cate\ m\ B\}$
 $\langle proof \rangle$

lemma *F_base_2*:
 $F\ m = 2$ **if** $0 < m$
 $\langle proof \rangle$

The recursion case

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

lemma *valid_line_aux*:
 $\{l. \text{length } l = n \wedge \text{valid } l\} \neq \{\}$ (is ?S ≠ {})
 ⟨proof⟩

lemma *replicate_unequal_aux*:
 $\text{replicate } x \text{ } R @ B \# l \neq \text{replicate } y \text{ } R @ B \# l'$ (is ?l ≠ ?r) if $\langle x < y \rangle$
 for $l \ l'$
 ⟨proof⟩

lemma *valid_prepend_B_iff*:
 $\text{valid } (B \# xs) \longleftrightarrow \text{valid } xs$ if $m > 0$
 ⟨proof⟩

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⟩

3.4.4 Computing the Fill-Count Function

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

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

lemma *lcount_correct*:
 $lcount \ n = F \ n$ if $m > 0$
 ⟨proof⟩

3.4.5 Memoization

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

memoize_correct
 ⟨proof⟩

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

end

3.4.6 Problem solutions

Example and solution for problem #114

```
value lcount 3 7
value lcount 3 50
```

Examples for problem #115

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

Binary search for the solution of problem #115

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

We prove that 168 is the solution for problem #115

theorem

```
(LEAST n. F 50 n > 1000000) = 168
⟨proof⟩
```

end

3.5 The CYK Algorithm

theory *CYK*

imports

```
HOL-Library.IArray
HOL-Library.Code_Target_Numeral
HOL-Library.Product_Lexorder
HOL-Library.RBT_Mapping
../state_monad/State_Main
../heap_monad/Heap_Default
Example_Misc
```

begin

3.5.1 Misc

lemma *append_iff_take_drop*:

$w = u@v \longleftrightarrow (\exists k \in \{0..length\ w\}. u = take\ k\ w \wedge v = drop\ k\ w)$
<proof>

lemma *append_iff_take_drop1*: $u \neq [] \implies v \neq [] \implies$

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

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**

$(A, T\ a) \in set\ P \implies yield\ A\ [a]$ |

$\llbracket (A, NN\ B\ C) \in set\ P; yield\ B\ u; yield\ C\ v \rrbracket \implies yield\ A\ (u@v)$

lemma *yield_not_Nil*: $yield\ A\ w \implies w \neq []$

<proof>

lemma *yield_eq1*:

$yield\ A\ [a] \longleftrightarrow (A, T\ a) \in set\ P\ (is\ ?L = ?R)$

<proof>

lemma *yield_eq2*: **assumes** $length\ w > 1$

shows $yield\ A\ w \longleftrightarrow (\exists B\ u\ C\ v. yield\ B\ u \wedge yield\ C\ v \wedge w = u@v \wedge (A, NN\ B\ C) \in set\ P)$

$(is\ ?L = ?R)$

<proof>

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*]: $\text{length } w \geq 2 \implies \text{set}(\text{cyk } w) =$
 $(\bigcup k \in \{1.. \text{length } w - 1\}. \bigcup B \in \text{set}(\text{cyk } (\text{take } k \ w)). \bigcup C \in \text{set}(\text{cyk } (\text{drop}$
 $k \ w))). \{A. (A, NN B C) \in \text{set } P\}$
 $\langle \text{proof} \rangle$

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

lemma *cyk_correct*: $\text{set}(\text{cyk } w) = \{N. \text{yield } N \ w\}$
 $\langle \text{proof} \rangle$

3.5.4 CYK on Lists and Index

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

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

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 (drop *i* *w*)))
 $\langle \text{proof} \rangle$

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

theorem *CYK_correct*: *CYK* *S* *w* = *yield* *S* *w*
 $\langle \text{proof} \rangle$

3.5.5 CYK With Index Function

context

fixes *w* :: *nat* \Rightarrow '*t*

begin

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

$B' C') <- P, B' = B, C' = C]$

3.5.6 Correctness Proof

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

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

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

lemma *slice_append_iff_take_drop1*: $u \neq [] \implies v \neq [] \implies$
 $slice\ w\ i\ j = u @ v \longleftrightarrow (\exists k. 1 \leq k \wedge k \leq j-i-1 \wedge slice\ w\ i\ (i+k) = u$
 $\wedge slice\ w\ (i+k)\ j = v)$
<proof>

lemma *cyk_ix_correct*:
 $set(cyk_ix\ (i, n)) = \{N. yield\ N\ (slice\ w\ i\ (i+n))\}$
<proof>

3.5.7 Functional Memoization

memoize_fun *cyk_ix_m*: *cyk_ix with_memory dp_consistency_mapping*
monadifies *(state) cyk_ix.simps*
thm *cyk_ix_m'.simps*

memoize_correct
<proof>
print_theorems

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

3.5.8 Imperative Memoization

context
fixes *n :: nat*
begin

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

memoize_fun *cyk_ix_h*: *cyk_ix*

```

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

context includes heap_monad_syntax begin
thm cyk_ix_h'.simps cyk_ix_h_def
end

memoize_correct
  ⟨proof⟩

lemmas memoized_empty = cyk_ix_h.memoized_empty

lemmas init_success = cyk_ix_h.init_success

end

definition cyk_ix_impl i j = do {mem ← mem_empty (n * n); cyk_ix_h'
mem (i, j)}

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

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

end

end

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

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

definition cyk_list w = cyk_ix ( $\lambda i. w ! i$ ) (0,length w)

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

lemma CYK_ix_impl_correct:

```

result_of (CYK_ix_impl S w n) Heap.empty = yield S (slice w 0 n)
 ⟨proof⟩

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 (λw. 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 (λi. 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>

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

fun *argmin2* :: ('a ⇒ 'b::order) ⇒ 'a list ⇒ 'a * 'b **where**
argmin2 *f* [*a*] = (*a*, *f* *a*) |
argmin2 *f* (*a*#*as*) = (let *fa* = *f* *a*; (*am*,*m*) = *argmin2* *f* as in if *fa* ≤ *m* then
(*a*, *fa*) else (*am*,*m*))

3.6.2 Edit Distance

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

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

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

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**

$min_eds\ []\ [] = [] \mid$
 $min_eds\ []\ (y\ \# \ ys) = Ins\ y\ \# \ min_eds\ []\ ys \mid$
 $min_eds\ (x\ \# \ xs)\ [] = Del\ \# \ min_eds\ xs\ [] \mid$
 $min_eds\ (x\ \# \ xs)\ (y\ \# \ ys) =$
 $\quad argmin\ cost\ [Ins\ y\ \# \ min_eds\ (x\ \# \ xs)\ ys,\ Del\ \# \ min_eds\ xs\ (y\ \# \ ys),$
 $\quad (if\ x=y\ then\ Copy\ else\ Repl\ y)\ \# \ min_eds\ xs\ ys]$

lemma *min_eds "vintner" "writers" =*

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

$\langle proof \rangle$

lemma *min_eds_correct*: $edit\ (min_eds\ xs\ ys)\ xs = ys$

$\langle proof \rangle$

lemma *min_eds_same*: $min_eds\ xs\ xs = replicate\ (length\ xs)\ Copy$

$\langle proof \rangle$

lemma *min_eds_eq_Nil_iff*: $min_eds\ xs\ ys = [] \iff xs = [] \wedge ys = []$

$\langle proof \rangle$

lemma *min_eds_Nil*: $min_eds\ []\ ys = map\ Ins\ ys$

$\langle proof \rangle$

lemma *min_eds_Nil2*: $min_eds\ xs\ [] = replicate\ (length\ xs)\ Del$

$\langle proof \rangle$

lemma *if_edit_Nil2*: $edit\ es\ ([]::'a\ list) = ys \implies length\ ys \leq cost\ es$

$\langle proof \rangle$

lemma *if_edit_eq_Nil*: $edit\ es\ xs = [] \implies length\ xs \leq cost\ es$

$\langle proof \rangle$

lemma *min_eds_minimal*: $edit\ es\ xs = ys \implies cost\ (min_eds\ xs\ ys) \leq cost$

es
 ⟨proof⟩

3.6.4 Computing the Minimum Edit Distance

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

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

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

Exercise: Optimization of the Copy case

fun *min_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
 ⟨proof⟩

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

lemma *cost*(*min_eds2* xs ys) = *cost*(*min_eds* xs ys)
 ⟨proof⟩

lemma *min_eds2* xs ys = *min_eds* xs ys

<proof>

3.6.5 Indexing

Indexing lists

context

fixes $xs\ ys :: 'a\ list$

fixes $m\ n :: nat$

begin

function (*sequential*)

$min_ed_ix' :: nat * nat \Rightarrow nat$ **where**

$min_ed_ix' (i, j) =$

(if $i \geq m$ then

if $j \geq n$ then 0 else $1 + min_ed_ix' (i, j+1)$ else

if $j \geq n$ then $1 + min_ed_ix' (i+1, j)$

else

$Min \{1 + min_ed_ix' (i, j+1), 1 + min_ed_ix' (i+1, j),$

$(if\ xs!i = ys!j\ then\ 0\ else\ 1) + min_ed_ix' (i+1, j+1)\}$)

<proof>

termination *<proof>*

declare $min_ed_ix'.simps[simp\ del]$

end

lemma $min_ed_ix'_min_ed:$

$min_ed_ix' xs\ ys (length\ xs) (length\ ys) (i, j) = min_ed (drop\ i\ xs) (drop$

$j\ ys)$

<proof>

Indexing functions

context

fixes $xs\ ys :: nat \Rightarrow 'a$

fixes $m\ n :: nat$

begin

function (*sequential*)

$min_ed_ix :: nat \times nat \Rightarrow nat$ **where**

$min_ed_ix (i, j) =$

(if $i \geq m$ then

if $j \geq n$ then 0 else $n-j$ else

if $j \geq n$ then $m-i$

```

    else
      min_list [1 + min_ed_ix (i, j+1), 1 + min_ed_ix (i+1, j),
        (if xs i = ys j then 0 else 1) + min_ed_ix (i+1, j+1)]
  <proof>
termination <proof>

```

3.6.6 Functional Memoization

```

memoize_fun min_ed_ix_m: min_ed_ix with_memory dp_consistency_mapping
monadifies (state) min_ed_ix.simps
thm min_ed_ix_m'.simps

```

```

memoize_correct
  <proof>
print_theorems

```

```

lemmas [code] = min_ed_ix_m.memoized_correct

```

```

declare min_ed_ix.simps[simp del]

```

3.6.7 Imperative Memoization

context

```

  fixes mem :: nat ref × nat ref × nat option array ref × nat option array
  ref

```

```

  assumes mem_is_init: mem = result_of (init_state (n + 1) m (m +
  1)) Heap.empty

```

begin

interpretation iterator

```

  λ (x, y). x ≤ m ∧ y ≤ n ∧ x > 0
  λ (x, y). if y > 0 then (x, y - 1) else (x - 1, n)
  λ (x, y). (m - x) * (n + 1) + (n - y)
  <proof>

```

lemma [intro]:

```

  dp_consistency_heap_array_pair' (n + 1) fst snd id m (m + 1) mem
  <proof>

```

lemma [intro]:

```

  dp_consistency_heap_array_pair_iterator (n + 1) fst snd id m (m + 1)
  mem
  (λ (x, y). if y > 0 then (x, y - 1) else (x - 1, n))
  (λ (x, y). (m - x) * (n + 1) + (n - y))

```


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

$\langle proof \rangle$

memoize_fun $min_ed_ix_h: min_ed_ix$
with_memory (**default_proof**) $dp_consistency_heap_array_pair_iterator$
where $size = n + 1$
and $key1=fst :: nat \times nat \Rightarrow nat$ **and** $key2=snd :: nat \times nat \Rightarrow nat$
and $k1=m :: nat$ **and** $k2=m + 1 :: nat$
and $to_index = id :: nat \Rightarrow nat$
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

$\langle proof \rangle$

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 $min_ed_Nil1: min_ed\ []\ ys = length\ ys$
 $\langle proof \rangle$

lemma $min_ed_Nil2: min_ed\ xs\ [] = length\ xs$
 $\langle proof \rangle$

lemma $min_ed_ix_min_ed: min_ed_ix\ xs\ ys\ m\ n\ (i,j) = min_ed\ (slice$
 $xs\ i\ m)\ (slice\ ys\ j\ n)$
 $\langle proof \rangle$

Functional Test Cases

definition $\text{min_ed_list } xs \ ys = \text{min_ed_ix } (\lambda i. xs!i) (\lambda i. ys!i) (\text{length } xs) (\text{length } ys) (0,0)$

lemma $\text{min_ed_list } "madagascar" \ "bananas" = 6$
 $\langle \text{proof} \rangle$

definition $\text{min_ed_ia } xs \ ys = (\text{let } a = \text{IArray } xs; \ b = \text{IArray } ys$
 $\text{in } \text{min_ed_ix } (\lambda i. a!!i) (\lambda i. b!!i) (\text{length } xs) (\text{length } ys) (0,0))$

lemma $\text{min_ed_ia } "madagascar" \ "bananas" = 6$
 $\langle \text{proof} \rangle$

Extracting an Executable Constant for the Imperative Implementation

ground_function $\text{min_ed_ix}_h'_{\text{impl}}: \text{min_ed_ix}_h'.\text{simps}$
termination

$\langle \text{proof} \rangle$

lemmas $[\text{simp del}] = \text{min_ed_ix}_h'_{\text{impl}}.\text{simps } \text{min_ed_ix}_h'.\text{simps}$

lemma $\text{min_ed_ix}_h'_{\text{impl_def}}:$

includes heap_monad_syntax

fixes $m \ n :: \text{nat}$

fixes $\text{mem} :: \text{nat ref} \times \text{nat ref} \times \text{nat option array ref} \times \text{nat option array ref}$

assumes $\text{mem_is_init}: \text{mem} = \text{result_of } (\text{init_state } (n + 1) \ m \ (m + 1)) \ \text{Heap.empty}$

shows $\text{min_ed_ix}_h'_{\text{impl}} \ xs \ ys \ m \ n \ \text{mem} = \text{min_ed_ix}_h' \ xs \ ys \ m \ n \ \text{mem}$
 $\langle \text{proof} \rangle$

definition

$\text{iter_min_ed_ix } xs \ ys \ m \ n \ \text{mem} = \text{iterator_defs.iter_heap}$

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

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

$(\text{min_ed_ix}_h'_{\text{impl}} \ xs \ ys \ m \ n \ \text{mem})$

lemma $\text{iter_min_ed_ix_unfold}[\text{code}]:$

$\text{iter_min_ed_ix } xs \ ys \ m \ n \ \text{mem} = (\lambda (i, j).$

$(\text{if } i > 0 \wedge i \leq m \wedge j \leq n$

$\text{then do } \{$

$\text{min_ed_ix}_h'_{\text{impl}} \ xs \ ys \ m \ n \ \text{mem} \ (i, j);$

$\text{iter_min_ed_ix } xs \ ys \ m \ n \ \text{mem} \ (\text{if } j > 0 \ \text{then } (i, j - 1) \ \text{else } (i$

$- 1, n))$

$\}$

```

    else Heap_Monad.return ()))
⟨proof⟩

```

definition

```

min_ed_ix_impl xs ys m n i j = do {
  mem ← (init_state (n + 1) (m::nat) (m + 1) ::
    (nat ref × nat ref × nat option array ref × nat option array ref)
Heap);
  iter_min_ed_ix xs ys m n mem (m, n);
  min_ed_ix'_impl xs ys m n mem (i, j)
}

```

lemma *bf_impl_correct*:

```

min_ed_ix xs ys m n (i, j) = result_of (min_ed_ix_impl xs ys m n i j)
Heap.empty
⟨proof⟩

```

Imperative Test Case

definition

```

min_ed_ia_h xs ys = (let a = IArray xs; b = IArray ys
in min_ed_ix_impl (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⟩

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_Natural

```

```

../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)
⟨proof⟩

```

```

lemma argmin_Min: xs ≠ [] ⇒ f (argmin f xs) = Min (f ` set xs)
⟨proof⟩

```

3.7.2 Misc

```

lemma upto_join: [ i ≤ j; j ≤ k ] ⇒ [i..j-1] @ j # [j+1..k] = [i..k]
⟨proof⟩

```

```

lemma atLeastAtMost_split:
  {i..j} = {i..k} ∪ {k+1..j} if i ≤ k k ≤ j for i j k :: int
⟨proof⟩

```

```

lemma atLeastAtMost_split_insert:
  {i..k} = insert k {i..k-1} if k ≥ i for i :: int
⟨proof⟩

```

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 (λ*k*. *min_wpl* *i* (*k*-1) + *min_wpl* (*k*+1) *j* + *W* *i* *j*)
 [*i*..*j*]))
 ⟨*proof*⟩
termination ⟨*proof*⟩
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*]])
 ⟨*proof*⟩
termination ⟨*proof*⟩
declare *opt_bst.simps*[*simp del*]

3.7.4 Functional Memoization

context

fixes *n* :: *nat*

begin

context fixes

mem :: *nat option array*

begin

memoize_fun *min_wpl*_Γ: *min_wpl*

with_memory *dp_consistency_heap_default* **where** *bound* = *Bound*
 (0, 0) (*int n*, *int n*) **and** *mem*=*mem*

monadifies (*heap*) *min_wpl.simps*

context includes *heap_monad_syntax* **begin**

thm *min_wpl*_Γ'.*simps* *min_wpl*_Γ_def

end

memoize_correct

⟨*proof*⟩

lemmas *memoized_empty* = *min_wpl*_Γ.*memoized_empty*

end

context

includes *heap_monad_syntax*

notes [*simp del*] = *min_wpl_T'.simps*

begin

definition *min_wpl_h* $\equiv \lambda i j. \text{Heap_Monad.bind } (\text{mem_empty } (n * n)) (\lambda \text{mem. } \text{min_wpl_T' mem } i j)$

lemma *min_wpl_heap*:

min_wpl *i j* = *result_of* (*min_wpl_h* *i j*) *Heap.empty*

<proof>

end

end

context includes *state_monad_syntax* **begin**

memoize_fun *min_wpl_m*: *min_wpl* **with_memory** *dp_consistency_mapping*

monadifies (*state*) *min_wpl.simps*

thm *min_wpl_m'.simps*

memoize_correct

<proof>

print_theorems

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

memoize_fun *opt_bst_m*: *opt_bst* **with_memory** *dp_consistency_mapping*

monadifies (*state*) *opt_bst.simps*

thm *opt_bst_m'.simps*

memoize_correct

<proof>

print_theorems

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

end

3.7.5 Correctness Proof

lemma *min_wpl_minimal*:

inorder t = [i..j] \implies min_wpl i j \leq wpl i j t

<proof>

lemma *opt_bst_correct*: $\text{inorder } (\text{opt_bst } i\ j) = [i..j]$
<proof>

lemma *wpl_opt_bst*: $\text{wpl } i\ j\ (\text{opt_bst } i\ j) = \text{min_wpl } i\ j$
<proof>

lemma *opt_bst_is_optimal*:
 $\text{inorder } t = [i..j] \implies \text{wpl } i\ j\ (\text{opt_bst } i\ j) \leq \text{wpl } i\ j\ t$
<proof>

end

3.7.6 Access Frequencies

Usually, the problem is phrased in terms of access frequencies. We now give an interpretation of *wpl* in this view and show that we have actually computed the right thing.

context

— We are given a range $[i..j]$ of integer keys with access frequencies p . These can be thought of as a probability distribution but are not required to represent one. This model assumes that the tree will contain all keys in the range $[i..j]$. See *Optimal_BST* for a model with missing keys.

fixes $p :: \text{int} \Rightarrow \text{nat}$

begin

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

fun $\text{cost} :: \text{int tree} \Rightarrow \text{nat}$ **where**

$\text{cost Leaf} = 0$

$| \text{cost } (\text{Node } l\ k\ r) = \text{sum } p\ (\text{set_tree } l) + \text{cost } l + p\ k + \text{cost } r + \text{sum } p\ (\text{set_tree } r)$

— Deriving a weight function from p .

qualified definition W **where**

$W\ i\ j = \text{sum } p\ \{i..j\}$

— We will use this later for computing W efficiently.

lemma W_rec :

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

<proof>

lemma *inorder_wpl_correct*:

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

<proof>

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

lemma *opt_bst_has_optimal_cost:*

inorder t = [i..j] \implies cost (opt_bst W i j) \leq cost t

<proof>

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 min_wpl W i j \leq cost t

<proof>

lemma *min_wpl_tree:*

cost (opt_bst W i j) = min_wpl W i j

<proof>

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

depth x Leaf = ∞

| depth x (Node l k r) = (if x = k then 1 else min (depth x l) (depth x r) + 1)

fun *the_fin* **where**

the_fin (Fin x) = x | the_fin _ = undefined

definition *cost'* :: int tree \Rightarrow nat **where**

*cost' t = sum (λx . the_fin (depth x t) * p x) (set_tree t)*

lemma [*simp*]:

the_fin 1 = 1

<proof>

lemma *set_tree_depth:*

assumes *x \notin set_tree t*

shows *depth x t = ∞*

<proof>

lemma *depth_inf_iff*:
 $depth\ x\ t = \infty \iff x \notin set_tree\ t$
 ⟨proof⟩

lemma *depth_not_neg_inf[simp]*:
 $depth\ x\ t = -\infty \iff False$
 ⟨proof⟩

lemma *depth_FinD*:
assumes $x \in set_tree\ t$
obtains d **where** $depth\ x\ t = Fin\ d$
 ⟨proof⟩

lemma *cost'_Leaf[simp]*:
 $cost'\ Leaf = 0$
 ⟨proof⟩

lemma *cost'_Node*:
 $distinct\ (inorder\ \langle l, x, r \rangle) \implies$
 $cost'\ \langle l, x, r \rangle = sum\ p\ (set_tree\ l) + cost'\ l + p\ x + cost'\ r + sum\ p$
 $(set_tree\ r)$
 ⟨proof⟩

lemma *weight_correct*:
 $distinct\ (inorder\ t) \implies cost'\ t = cost\ t$
 ⟨proof⟩

3.7.7 Memoizing Weights

function *W_fun* **where**
 $W_fun\ i\ j = (if\ i > j\ then\ 0\ else\ W_fun\ i\ (j - 1) + p\ j)$
 ⟨proof⟩

termination
 ⟨proof⟩

lemma *W_fun_correct*:
 $W_fun\ i\ j = W\ i\ j$
 ⟨proof⟩

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

memoize_correct

<proof>

definition

$compute_W\ n = snd\ (run_state\ (State_Main.mapT'\ (\lambda i. W_m'\ i\ n)\ [0..n])\ Mapping.empty)$

notation $W_m.crel_vs\ (crel)$

lemmas $W_m.crel = W_m.crel[unfolded\ W_m.consistentDP_def,\ THEN\ rel_funD,\ of\ (m,\ x)\ (m,\ y)\ \text{for}\ m\ x\ y,\ unfolded\ prod.case]$

lemma $compute_W_correct:$

assumes $Mapping.lookup\ (compute_W\ n)\ (i,\ j) = Some\ x$

shows $W\ i\ j = x$

<proof>

include $state_monad_syntax\ app_syntax\ lifting_syntax$

<proof>

definition

$min_wpl'\ i\ j \equiv$

let

$M = compute_W\ j;$

$W = (\lambda i\ j. case\ Mapping.lookup\ M\ (i,\ j)\ of\ None \Rightarrow W\ i\ j\ |\ Some\ x \Rightarrow$

$x)$

in $min_wpl\ W\ i\ j$

lemma $W_compute:$ $W\ i\ j = (case\ Mapping.lookup\ (compute_W\ n)\ (i,\ j)\ of\ None \Rightarrow W\ i\ j\ |\ Some\ x \Rightarrow x)$

<proof>

lemma $min_wpl'_correct:$

$min_wpl'\ i\ j = min_wpl\ W\ i\ j$

<proof>

definition

$opt_bst'\ i\ j \equiv$

let

$M = compute_W\ j;$

$W = (\lambda i\ j. case\ Mapping.lookup\ M\ (i,\ j)\ of\ None \Rightarrow W\ i\ j\ |\ Some\ x \Rightarrow$

$x)$

in $opt_bst\ W\ i\ j$

lemma $opt_bst'_correct:$

$opt_bst'\ i\ j = opt_bst\ W\ i\ j$

<proof>

end

3.7.8 Test Case

Functional Implementations

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

<proof>

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

<proof>

Using Frequencies

definition

list_to_p xs (i::int) = (if i - 1 \geq 0 \wedge nat (i - 1) < length xs then xs ! nat (i - 1) else 0)

definition

ex_p_1 = [10, 30, 15, 25, 20]

definition

opt_tree_1 =
 \langle
 \langle
 $\langle\langle\rangle, 1::\text{int}, \langle\rangle\rangle,$
2,
 $\langle\langle\rangle, 3, \langle\rangle\rangle$
 $\rangle,$
4,
 $\langle\langle\rangle, 5, \langle\rangle\rangle$
 \rangle

lemma *opt_bst'* (*list_to_p ex_p_1*) 1 5 = *opt_tree_1*

<proof>

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>

end

3.8 Longest Common Subsequence

theory *Longest_Common_Subsequence*

imports

HOL-Library.Sublist
HOL-Library.IArray
HOL-Library.Code_Target_Numeral
HOL-Library.Product_Lexorder
HOL-Library.RBT_Mapping
../state_monad/State_Main

begin

3.8.1 Misc

lemma *finite_subseq*:

finite {xs. subseq xs ys} (is finite ?S)
<proof>

lemma *subseq_singleton_right*:

subseq xs [x] = (xs = [x] \vee xs = [])
<proof>

lemma *subseq_append_single_right*:

subseq xs (ys @ [x]) = ((\exists xs'. subseq xs' ys \wedge xs = xs' @ [x]) \vee subseq xs ys)
<proof>

lemma *Max_nat_plus*:

Max (((+) n) ' S) = (n :: nat) + Max S **if** *finite S S \neq {}*
<proof>

3.8.2 Definitions

context

fixes *A B :: 'a list*

begin

fun *lcs :: nat \Rightarrow nat \Rightarrow nat* **where**

lcs 0 _ = 0 |
lcs _ 0 = 0 |

$lcs (Suc\ i) (Suc\ j) = (if\ A!i = B!j\ then\ 1 + lcs\ i\ j\ else\ max\ (lcs\ i\ (j + 1))\ (lcs\ (i + 1)\ j))$

definition $OPT\ i\ j = Max\ \{length\ xs\ |\ xs.\ subseq\ xs\ (take\ i\ A) \wedge\ subseq\ xs\ (take\ j\ B)\}$

lemma $finite_OPT$:

$finite\ \{xs.\ subseq\ xs\ (take\ i\ A) \wedge\ subseq\ xs\ (take\ j\ B)\}$ (**is** $finite\ ?S$)
 $\langle proof \rangle$

3.8.3 Correctness Proof

lemma non_empty_OPT :

$\{xs.\ subseq\ xs\ (take\ i\ A) \wedge\ subseq\ xs\ (take\ j\ B)\} \neq \{\}$
 $\langle proof \rangle$

lemma OPT_0_left :

$OPT\ 0\ j = 0$
 $\langle proof \rangle$

lemma OPT_0_right :

$OPT\ i\ 0 = 0$
 $\langle proof \rangle$

lemma OPT_rec1 :

$OPT\ (i + 1)\ (j + 1) = 1 + OPT\ i\ j$ (**is** $?l = ?r$)
if $A!i = B!j\ i < length\ A\ j < length\ B$
 $\langle proof \rangle$

lemma OPT_rec2 :

$OPT\ (i + 1)\ (j + 1) = max\ (OPT\ i\ (j + 1))\ (OPT\ (i + 1)\ j)$ (**is** $?l = ?r$)
if $A!i \neq B!j\ i < length\ A\ j < length\ B$
 $\langle proof \rangle$

lemma $lcs_correct'$:

$OPT\ i\ j = lcs\ i\ j$ **if** $i \leq length\ A\ j \leq length\ B$
 $\langle proof \rangle$

theorem $lcs_correct$:

$Max\ \{length\ xs\ |\ xs.\ subseq\ xs\ A \wedge\ subseq\ xs\ B\} = lcs\ (length\ A)\ (length\ B)$
 $\langle proof \rangle$

end

3.8.4 Functional Memoization

context

fixes $A B :: 'a\ iarray$

begin

fun $lcs_ia :: nat \Rightarrow nat \Rightarrow nat$ **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** $A = IArray\ xs\ B = IArray\ ys$

$\langle proof \rangle$

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

memoize_correct

$\langle proof \rangle$

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 $lcs_a_correct$:

$lcs\ xs\ ys\ (length\ xs)\ (length\ ys) = lcs_a\ xs\ ys$

$\langle proof \rangle$

value $lcs_a\ "ABCDGH"\ "AEDFHR"$

value $lcs_a\ "AGGTAB"\ "GXTXAYB"$

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.