A Restricted Definition of the Magic Wand to Soundly Combine Fractions of a Wand

Thibault Dardinier

May 30, 2022

Abstract

Many separation logics support *fractional permissions* [1] to distinguish between read and write access to a heap location, for instance, to allow concurrent reads while enforcing exclusive writes. The concept has been generalized to fractional assertions [2, 5, 6, 3]. A^p (where A is a separation logic assertion and p a fraction between 0 and 1) represents a fraction p of A. A^p holds in a state σ iff there exists a state σ_A in which A holds and σ is obtained from σ_A by multiplying all permission amounts held by p.

While A^{p+q} can always be split into $A^p * A^q$, recombining $A^p * A^q$ into A^{p+q} is not always sound. We say that A is *combinable* iff the entailment $A^p * A^q \models A^{p+q}$ holds for any two positive fractions p and q such that $p + q \leq 1$. Combinable assertions are particularly useful to reason about concurrent programs, for instance, to combine the post-conditions of parallel branches when they terminate. Unfortunately, the magic wand assertion $A \mathrel{-\!\!*} B$, commonly used to specify properties of partial data structures, is typically *not* combinable.

In this entry, we formalize a novel, restricted definition of the magic wand, described in a paper at CAV 22 [4], which we call the *combinable wand*. We prove some key properties of the combinable wand; in particular, a combinable wand is combinable if its right-hand side is combinable.

Contents

1 State Model with Fractional Permissions

In this section, we define a concrete state model based on fractional permissions [1]. A state is a pair of a permission mask and a partial heap. A permission mask is a total map from heap locations to a rational between 0 and 1 (included), while a partial heap is a partial map from heap locations to values. We also define a partial addition on these states, and show that this state model corresponds to a separation algebra.

1.1 Non-negative rationals (permission amounts)

theory *PosRat*
imports *Main HOL.Rat*
begin

1.1.1 Definitions

typedef *prat* = { *r* :: *rat* |*r*. *r* ≥ *0*} ⟨*proof*⟩

setup-lifting *type-definition-prat*

lift-definition *pwrite* :: *prat* **is** *1* ⟨*proof*⟩
lift-definition *pnone* :: *prat* **is** *0* ⟨*proof*⟩
lift-definition *half* :: *prat* **is** *1 / 2* ⟨*proof*⟩

lift-definition *pgte* :: *prat* ⇒ *prat* ⇒ *bool* **is** (≥) ⟨*proof*⟩
lift-definition *pgt* :: *prat* ⇒ *prat* ⇒ *bool* **is** (>) ⟨*proof*⟩

lift-definition *lt* :: *prat* ⇒ *prat* ⇒ *bool* **is** (<) ⟨*proof*⟩
lift-definition *ppos* :: *prat* ⇒ *bool* **is** λ*p*. *p* > *0* ⟨*proof*⟩

lift-definition *pmult* :: *prat* ⇒ *prat* ⇒ *prat* **is** (∗) ⟨*proof*⟩
lift-definition *padd* :: *prat* ⇒ *prat* ⇒ *prat* **is** (+) ⟨*proof*⟩

lift-definition *pdiv* :: *prat* ⇒ *prat* ⇒ *prat* **is** (/) ⟨*proof*⟩

lift-definition *pmin* :: *prat* ⇒ *prat* ⇒ *prat* **is** (*min*) ⟨*proof*⟩
lift-definition *pmax* :: *prat* ⇒ *prat* ⇒ *prat* **is** (*max*) ⟨*proof*⟩

1.1.2 Lemmas

lemma *pmin-comm*:
pmin a b = pmin b a
⟨*proof*⟩

lemma *pmin-greater*:
pgte a (*pmin a b*)
⟨*proof*⟩

lemma *pmin-is*:
assumes *pgte a b*
shows *pmin a b = b*
⟨*proof*⟩

lemma *pmax-comm*:
pmax a b = pmax b a
⟨*proof*⟩

lemma *pmax-smaller*:
pgte (*pmax a b*) *a*
⟨*proof*⟩

lemma *pmax-is*:
assumes *pgte a b*
shows *pmax a b = a*
⟨*proof*⟩

lemma *pmax-is-smaller*:
assumes *pgte x a*
and *pgte x b*
shows *pgte x* (*pmax a b*)
⟨*proof*⟩

lemma *half-between-0-1*:
ppos half ∧ *pgt pwrite half*
⟨*proof*⟩

lemma *pgt-implies-pgte*:
assumes *pgt a b*
shows *pgte a b*

⟨*proof*⟩

lemma *half-plus-half*:
padd half half = *pwrite*
⟨*proof*⟩

lemma *padd-comm*:
padd a b = *padd b a*
⟨*proof*⟩

lemma *padd-asso*:
padd (*padd a b*) *c* = *padd a* (*padd b c*)
⟨*proof*⟩

lemma *p-greater-exists*:
pgte a b ⟷ (∃ *r*. *a* = *padd b r*)
⟨*proof*⟩

lemma *pgte-antisym*:
assumes *pgte a b*
and *pgte b a*
shows *a* = *b*
⟨*proof*⟩

lemma *greater-sum-both*:
assumes *pgte a* (*padd b c*)
shows ∃ *a1 a2*. *a* = *padd a1 a2* ∧ *pgte a1 b* ∧ *pgte a2 c*
⟨*proof*⟩

lemma *padd-cancellative*:
assumes *a* = *padd x b*
and *a* = *padd y b*
shows *x* = *y*
⟨*proof*⟩

lemma *not-pgte-charact*:
¬ *pgte a b* ⟷ *pgt b a*
⟨*proof*⟩

lemma *pgte-pgt*:
assumes *pgt a b*
and *pgte c d*
shows *pgt* (*padd a c*) (*padd b d*)
⟨*proof*⟩

lemma *pmult-distr*:
pmult a (*padd b c*) = *padd* (*pmult a b*) (*pmult a c*)
⟨*proof*⟩

lemma *pmult-comm*:
pmult a b = *pmult b a*
⟨*proof*⟩

lemma *pmult-special*:
pmult pwrite x = *x*
pmult pnone x = *pnone*
⟨*proof*⟩

definition *pinv* **where**
pinv p = *pdiv pwrite p*

lemma *pinv-double-half*:
assumes *ppos p*
shows *pmult half* (*pinv p*) = *pinv* (*padd p p*)
⟨*proof*⟩

lemma *ppos-mult*:
assumes *ppos a*
and *ppos b*
shows *ppos* (*pmult a b*)
⟨*proof*⟩

lemma *padd-zero*:
pnone = *padd a b* ⟷ *a* = *pnone* ∧ *b* = *pnone*
⟨*proof*⟩

lemma *ppos-add*:
assumes *ppos a*
shows *ppos* (*padd a b*)
⟨*proof*⟩

lemma *pinv-inverts*:
assumes *pgte a b*
and *ppos b*
shows *pgte* (*pinv b*) (*pinv a*)
⟨*proof*⟩

lemma *pinv-pmult-ok*:

assumes *ppos p*
shows *pmult p* (*pinv p*) = *pwrite*
⟨*proof*⟩

lemma *pinv-pwrite*:
pinv pwrite = *pwrite*
⟨*proof*⟩

lemma *pmult-ppos*:
assumes *ppos a*
and *ppos b*
shows *ppos* (*pmult a b*)
⟨*proof*⟩

lemma *ppos-inv*:
assumes *ppos p*
shows *ppos* (*pinv p*)
⟨*proof*⟩

lemma *pmin-pmax*:
assumes *pgte x* (*pmin a b*)
shows *x* = *pmin* (*pmax x a*) (*pmax x b*)
⟨*proof*⟩

definition *comp-one* **where**
comp-one p = (*SOME r*. *padd p r* = *pwrite*)

lemma *padd-comp-one*:
assumes *pgte pwrite x*
shows *padd x* (*comp-one x*) = *pwrite*
⟨*proof*⟩

lemma *ppos-eq-pnone*:
ppos p ⟷ *p* ≠ *pnone*
⟨*proof*⟩

lemma *pmult-order*:
assumes *pgte a b*
shows *pgte* (*pmult p a*) (*pmult b p*)
⟨*proof*⟩

lemma *multiply-smaller-pwrite*:
assumes *pgte pwrite a*
and *pgte pwrite b*

shows *pgte pwrite* (*pmult a b*)
⟨*proof*⟩

lemma *pmult-pdiv-cancel*:
assumes *ppos a*
shows *pmult a* (*pdiv x a*) = *x*
⟨*proof*⟩

lemma *pmult-padd*:
pmult a (*padd* (*pmult b x*) (*pmult c y*)) = *padd* (*pmult* (*pmult a b*) *x*) (*pmult* (*pmult a c*) *y*)
⟨*proof*⟩

lemma *pdiv-smaller*:
assumes *pgte a b*
and *ppos a*
shows *pgte pwrite* (*pdiv b a*)
⟨*proof*⟩

lemma *sum-coeff*:
assumes *ppos a*
and *ppos b*
shows *padd* (*pdiv a* (*padd a b*)) (*pdiv b* (*padd a b*)) = *pwrite*
⟨*proof*⟩

lemma *padd-one-ineq-sum*:
assumes *padd a b* = *pwrite*
and *pgte x aa*
and *pgte x bb*
shows *pgte x* (*padd* (*pmult a aa*) (*pmult b bb*))
⟨*proof*⟩

end

1.2 Permission masks: Maps from heap locations to permission amounts

theory *Mask*
imports *PosRat*
begin

1.2.1 Definitions

type-synonym *field* = *string*
type-synonym *address* = *nat*
type-synonym *heap-loc* = *address* × *field*

type-synonym *mask* = *heap-loc* ⇒ *prat*
type-synonym *bmask* = *heap-loc* ⇒ *bool*

definition *null* **where** *null* = *0*

definition *full-mask* :: *mask* **where**
full-mask hl = (*if fst hl* = *null then pnone else pwrite*)

definition *multiply-mask* :: *prat* ⇒ *mask* ⇒ *mask* **where**
multiply-mask p π *hl* = *pmult p* (π *hl*)

fun *empty-mask* **where**
empty-mask hl = *pnone*

fun *empty-bmask* **where**
empty-bmask hl = *False*

fun *add-acc* **where** *add-acc* π *hl p* = π(*hl* := *padd* (π *hl*) *p*)

inductive *rm-acc* **where**
π *hl* = *padd p r* ⟹ *rm-acc* π *hl p* (π(*hl* := *r*))

fun *add-masks* **where**
add-masks π′ π *hl* = *padd* (π′ *hl*) (π *hl*)

definition *greater-mask* **where**
greater-mask π′ π ⟷ (∃ *r*. π′ = *add-masks* π *r*)

fun *uni-mask* **where**
uni-mask hl p = *empty-mask*(*hl* := *p*)

fun *valid-mask* :: *mask* ⇒ *bool* **where**
valid-mask π ⟷ (∀ *hl*. *pgte pwrite* (π *hl*)) ∧ (∀ *f*. π (*null*, *f*) = *pnone*)

definition *valid-null* :: *mask* ⇒ *bool* **where**
valid-null π ⟷ (∀ *f*. π (*null*, *f*) = *pnone*)

definition *equal-on-mask* **where**
equal-on-mask π *h h*′ ⟷ (∀ *hl*. *ppos* (π *hl*) ⟶ *h hl* = *h*′ *hl*)

definition *equal-on-bmask* **where**
equal-on-bmask π *h h*′ ⟷ (∀ *hl*. π *hl* ⟶ *h hl* = *h*′ *hl*)

definition *big-add-masks* **where**
big-add-masks Π Π′ *h* = *add-masks* (Π *h*) (Π′ *h*)

definition *big-greater-mask* **where**

big-greater-mask Π Π′ ⟷ (∀ *h*. *greater-mask* (Π *h*) (Π′ *h*))

definition *greater-bmask* **where**
greater-bmask H H′ ⟷ (∀ *h*. *H′ h* ⟶ *H h*)

definition *update-dm* **where**
update-dm dm π π′ hl ⟷ (*dm hl* ∨ *pgt* (*π hl*) (*π′ hl*))

fun *pre-get-m* **where** *pre-get-m φ* = *fst φ*
fun *pre-get-h* **where** *pre-get-h φ* = *snd φ*
fun *srm-acc* **where** *srm-acc φ hl p* = (*rm-acc* (*pre-get-m φ*) *hl p*, *pre-get-h φ*)

datatype *val* = *Bool* (*the-bool*: *bool*) | *Address* (*the-address*: *address*) | *Rat* (*the-rat*: *prat*)

definition *upper-bounded* :: *mask* ⇒ *prat* ⇒ *bool* **where**
upper-bounded π p ⟷ (∀ *hl*. *pgte p* (*π hl*))

1.2.2 Lemmas

lemma *ssubsetI*:
assumes ⋀*π h*. (*π*, *h*) ∈ *A* ⟹ (*π*, *h*) ∈ *B*
shows *A* ⊆ *B*
⟨*proof*⟩

lemma *double-inclusion*:
assumes *A* ⊆ *B*
and *B* ⊆ *A*
shows *A* = *B*
⟨*proof*⟩

lemma *add-masks-comm*:
add-masks a b = *add-masks b a*
⟨*proof*⟩

lemma *add-masks-asso*:
add-masks (*add-masks a b*) *c* = *add-masks a* (*add-masks b c*)
⟨*proof*⟩

lemma *minus-empty*:
π = *add-masks π empty-mask*
⟨*proof*⟩

lemma *add-acc-uni-mask*:
add-acc π hl p = *add-masks π* (*uni-mask hl p*)
⟨*proof*⟩

lemma *add-masks-equiv-valid-null*:

valid-null (*add-masks a b*) ⟷ *valid-null a* ∧ *valid-null b*
⟨*proof*⟩

lemma *valid-maskI*:
assumes ⋀*hl*. *pgte pwrite* (*π hl*)
and ⋀*f*. *π* (*null*, *f*) = *pnone*
shows *valid-mask π*
⟨*proof*⟩

lemma *greater-mask-equiv-def*:
greater-mask π′ π ⟷ (∀ *hl*. *pgte* (*π′ hl*) (*π hl*))
(**is** *?A* ⟷ *?B*)
⟨*proof*⟩

lemma *greater-maskI*:
assumes ⋀*hl*. *pgte* (*π′ hl*) (*π hl*)
shows *greater-mask π′ π*
⟨*proof*⟩

lemma *greater-mask-properties*:
greater-mask π π
greater-mask a b ∧ *greater-mask b c* ⟹ *greater-mask a c*
greater-mask π′ π ∧ *greater-mask π π′* ⟹ *π* = *π′*
⟨*proof*⟩

lemma *greater-mask-decomp*:
assumes *greater-mask a* (*add-masks b c*)
shows ∃ *a1 a2*. *a* = *add-masks a1 a2* ∧ *greater-mask a1 b* ∧ *greater-mask a2 c*
⟨*proof*⟩

lemma *valid-empty*:
valid-mask empty-mask
⟨*proof*⟩

lemma *upper-valid-aux*:
assumes *valid-mask a*
and *a* = *add-masks b c*
shows *valid-mask b*
⟨*proof*⟩

lemma *upper-valid*:
assumes *valid-mask a*
and *a* = *add-masks b c*
shows *valid-mask b* ∧ *valid-mask c*
⟨*proof*⟩

lemma *equal-on-bmaskI*:
assumes ⋀*hl*. *π hl* ⟹ *h hl* = *h′ hl*
shows *equal-on-bmask π h h′*

⟨*proof*⟩

lemma *big-add-greater*:
big-greater-mask (*big-add-masks A B*) *B*
⟨*proof*⟩

lemma *big-greater-iff*:
big-greater-mask A B ⟹ (∃ *C*. *A* = *big-add-masks B C*)
⟨*proof*⟩

lemma *big-add-masks-asso*:
big-add-masks A (*big-add-masks B C*) = *big-add-masks* (*big-add-masks A B*) *C*
⟨*proof*⟩

lemma *big-add-mask-neutral*:
big-add-masks Π (λ-. *empty-mask*) = Π
⟨*proof*⟩

lemma *sym-equal-on-mask*:
equal-on-mask π a b ⟷ *equal-on-mask π b a*
⟨*proof*⟩

lemma *greater-mask-uni-equiv*:
greater-mask π (*uni-mask hl r*) ⟷ *pgte* (*π hl*) *r*
⟨*proof*⟩

lemma *greater-mask-uniI*:
assumes *pgte* (*π hl*) *r*
shows *greater-mask π* (*uni-mask hl r*)
⟨*proof*⟩

lemma *greater-bmask-refl*:
greater-bmask H H
⟨*proof*⟩

lemma *greater-bmask-trans*:
assumes *greater-bmask A B*
and *greater-bmask B C*
shows *greater-bmask A C*
⟨*proof*⟩

lemma *update-dm-same*:
update-dm dm π π = *dm*
⟨*proof*⟩

lemma *update-trans*:
assumes *greater-mask π π′*
and *greater-mask π′ π″*
shows *update-dm* (*update-dm dm π π′*) *π′ π″* = *update-dm dm π π″*

⟨*proof*⟩

lemma *equal-on-bmask-greater*:
assumes *equal-on-bmask* π' *h* h'
and *greater-bmask* π' π
shows *equal-on-bmask* π *h* h'
⟨*proof*⟩

lemma *update-dm-equal-bmask*:
assumes π = *add-masks* π' *m*
shows *equal-on-bmask* (*update-dm dm* π π') h' *h* ⟷ *equal-on-mask m h* h' ∧ *equal-on-bmask dm h* h'
⟨*proof*⟩

lemma *const-sum-mask-greater*:
assumes *add-masks a b* = *add-masks c d*
and *greater-mask a c*
shows *greater-mask d b*
⟨*proof*⟩

lemma *add-masks-cancellative*:
assumes *add-masks b c* = *add-masks b d*
shows *c* = *d*
⟨*proof*⟩

lemma *equal-on-maskI*:
assumes ⋀*hl*. *ppos* (π *hl*) ⟹ *h hl* = h' *hl*
shows *equal-on-mask* π *h* h'
⟨*proof*⟩

lemma *greater-equal-on-mask*:
assumes *equal-on-mask* π' *h* h'
and *greater-mask* π' π
shows *equal-on-mask* π *h* h'
⟨*proof*⟩

lemma *equal-on-mask-sum*:
equal-on-mask π*1 h* h' ∧ *equal-on-mask* π*2 h* h' ⟷ *equal-on-mask* (*add-masks* π*1* π*2*) *h* h'
⟨*proof*⟩

lemma *valid-larger-mask*:
valid-mask π ⟷ *greater-mask full-mask* π
⟨*proof*⟩

lemma *valid-mask-full-mask*:
valid-mask full-mask
⟨*proof*⟩

lemma *mult-greater*:
assumes *greater-mask a b*
shows *greater-mask* (*multiply-mask p a*) (*multiply-mask p b*)
⟨*proof*⟩

lemma *mult-write-mask*:
multiply-mask pwrite π = *π*
⟨*proof*⟩

lemma *mult-distr-masks*:
multiply-mask a (*add-masks b c*) = *add-masks* (*multiply-mask a b*) (*multiply-mask a c*)
⟨*proof*⟩

lemma *mult-add-states*:
multiply-mask (*padd a b*) *π* = *add-masks* (*multiply-mask a π*) (*multiply-mask b π*)
⟨*proof*⟩

lemma *upper-boundedI*:
assumes ⋀*hl*. *pgte p* (*π hl*)
shows *upper-bounded π p*
⟨*proof*⟩

lemma *upper-bounded-smaller*:
assumes *upper-bounded π a*
shows *upper-bounded* (*multiply-mask p π*) (*pmult p a*)
⟨*proof*⟩

lemma *upper-bounded-bigger*:
assumes *upper-bounded π a*
and *pgte b a*
shows *upper-bounded π b*
⟨*proof*⟩

lemma *valid-mult*:
assumes *valid-mask π*
and *pgte pwrite p*
shows *valid-mask* (*multiply-mask p π*)
⟨*proof*⟩

lemma *valid-sum*:
assumes *valid-mask a*
and *valid-mask b*
and *upper-bounded a ma*
and *upper-bounded b mb*
and *pgte pwrite* (*padd ma mb*)
shows *valid-mask* (*add-masks a b*)

and *upper-bounded* (*add-masks a b*) (*padd ma mb*)
⟨*proof*⟩

lemma *valid-multiply*:
assumes *valid-mask a*
and *upper-bounded a ma*
and *pgte pwrite* (*pmult ma p*)
shows *valid-mask* (*multiply-mask p a*)
⟨*proof*⟩

lemma *greater-mult*:
assumes *greater-mask a b*
shows *greater-mask* (*multiply-mask p a*) (*multiply-mask p b*)
⟨*proof*⟩

end

1.3 Partial heaps: Partial maps from heap location to values

theory *PartialHeapSA*
imports *Mask Package-logic.PackageLogic*
begin

1.3.1 Definitions

type-synonym *heap* = *heap-loc* ⇀ *val*
type-synonym *pre-state* = *mask* × *heap*

definition *valid-heap* :: *mask* ⇒ *heap* ⇒ *bool* **where**
valid-heap π *h* ⟷ (∀ *hl*. *ppos* (π *hl*) ⟶ *h hl* ≠ *None*)

fun *valid-state* :: *pre-state* ⇒ *bool* **where**
valid-state (π, *h*) ⟷ *valid-mask* π ∧ *valid-heap* π *h*

lemma *valid-stateI*:
assumes *valid-mask* π
and ⋀*hl*. *ppos* (π *hl*) ⟹ *h hl* ≠ *None*
shows *valid-state* (π, *h*)
⟨*proof*⟩

definition *empty-heap* **where** *empty-heap hl* = *None*

lemma *valid-pre-unit*:
valid-state (*empty-mask*, *empty-heap*)
⟨*proof*⟩

typedef *state* = { φ |φ. *valid-state* φ }
⟨*proof*⟩

fun *get-m* :: *state* ⇒ *mask* **where** *get-m a* = *fst* (*Rep-state a*)

fun *get-h* :: *state* ⇒ *heap* **where** *get-h a* = *snd* (*Rep-state a*)

fun *compatible-options* **where**
compatible-options (*Some a*) (*Some b*) ⟷ *a* = *b*
| *compatible-options* - - ⟷ *True*

definition *compatible-heaps* :: *heap* ⇒ *heap* ⇒ *bool* **where**
compatible-heaps h h′ ⟷ (∀ *hl*. *compatible-options* (*h hl*) (*h′ hl*))

definition *compatible* :: *pre-state* ⇒ *pre-state* ⇒ *bool* **where**
compatible φ φ′ ⟷ *compatible-heaps* (*snd φ*) (*snd φ′*) ∧ *valid-mask* (*add-masks* (*fst φ*) (*fst φ′*))

fun *add-states* :: *pre-state* ⇒ *pre-state* ⇒ *pre-state* **where**
add-states (*π*, *h*) (*π′*, *h′*) = (*add-masks π π′*, *h* ++ *h′*)

definition *larger-heap* **where**
larger-heap h′ h ⟷ (∀ *hl x*. *h hl* = *Some x* ⟶ *h′ hl* = *Some x*)

definition *unit* :: *state* **where**
unit = *Abs-state* (*empty-mask*, *empty-heap*)

definition *plus* :: *state* ⇒ *state* ⇀ *state* (**infixl** ⊕ *63*) **where**
a ⊕ *b* = (*if compatible* (*Rep-state a*) (*Rep-state b*) *then Some* (*Abs-state* (*add-states* (*Rep-state a*) (*Rep-state b*))) *else None*)

definition *core* :: *state* ⇒ *state* (|-|) **where**
core φ = *Abs-state* (*empty-mask*, *get-h φ*)

definition *stable* :: *state* ⇒ *bool* **where**
stable φ ⟷ (∀ *hl*. *ppos* (*get-m φ hl*) ⟷ *get-h φ hl* ≠ *None*)

1.3.2 Lemmas

lemma *valid-heapI*:
assumes ⋀*hl*. *ppos* (*π hl*) ⟹ *h hl* ≠ *None*
shows *valid-heap π h*
⟨*proof*⟩

lemma *valid-state-decompose*:
assumes *valid-state* (*add-masks a b*, *h*)
shows *valid-state* (*a*, *h*)
⟨*proof*⟩

lemma *compatible-heapsI*:
assumes ⋀*hl a b*. *h hl* = *Some a* ⟹ *h′ hl* = *Some b* ⟹ *a* = *b*
shows *compatible-heaps h h′*
⟨*proof*⟩

lemma *compatibleI-old*:
assumes $\bigwedge hl\ x\ y.\ snd\ \varphi\ hl = Some\ x \wedge snd\ \varphi'\ hl = Some\ y \Longrightarrow x = y$
and *valid-mask* (*add-masks* (*fst* φ) (*fst* φ'))
shows *compatible* $\varphi\ \varphi'$
⟨*proof*⟩

lemma *larger-heap-anti*:
assumes *larger-heap a b*
and *larger-heap b a*
shows $a = b$
⟨*proof*⟩

lemma *larger-heapI*:
assumes $\bigwedge hl\ x.\ h\ hl = Some\ x \Longrightarrow h'\ hl = Some\ x$
shows *larger-heap* $h'\ h$
⟨*proof*⟩

lemma *larger-heap-refl*:
larger-heap h h
⟨*proof*⟩

lemma *compatible-heaps-comm*:
assumes *compatible-heaps a b*
shows $a +\!\!+ b = b +\!\!+ a$
⟨*proof*⟩

lemma *larger-heaps-sum-ineq*:
assumes *larger-heap* $a'\ a$
and *larger-heap* $b'\ b$
and *compatible-heaps* $a'\ b'$
shows *larger-heap* $(a' +\!\!+ b')\ (a +\!\!+ b)$
⟨*proof*⟩

lemma *larger-heap-trans*:
assumes *larger-heap a b*
and *larger-heap b c*
shows *larger-heap a c*
⟨*proof*⟩

lemma *larger-heap-comp*:
assumes *larger-heap a b*
and *compatible-heaps a c*
shows *compatible-heaps b c*
⟨*proof*⟩

lemma *larger-heap-plus*:
assumes *larger-heap a b*
and *larger-heap a c*
shows *larger-heap a* $(b +\!\!+ c)$

⟨*proof*⟩

lemma *compatible-heaps-sum*:
assumes *compatible-heaps a b*
and *compatible-heaps a c*
shows *compatible-heaps a* (*b* ++ *c*)
⟨*proof*⟩

lemma *larger-compatible-sum-heaps*:
assumes *larger-heap a x*
and *larger-heap b y*
and *compatible-heaps a b*
shows *compatible-heaps x y*
⟨*proof*⟩

lemma *get-h-m*:
Rep-state x = (*get-m x*, *get-h x*) ⟨*proof*⟩

lemma *get-pre*:
get-h x = *snd* (*Rep-state x*)
get-m x = *fst* (*Rep-state x*)
⟨*proof*⟩

lemma *plus-ab-defined*:
$\varphi \oplus \varphi' \neq None \longleftrightarrow$ *compatible-heaps* (*get-h* φ) (*get-h* φ') $\wedge$ *valid-mask* (*add-masks* (*get-m* φ) (*get-m* φ'))
(**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *plus-charact*:
assumes $a \oplus b = Some\ x$
shows *get-m x* = *add-masks* (*get-m a*) (*get-m b*)
and *get-h x* = (*get-h a*) ++ (*get-h b*)
⟨*proof*⟩

lemma *commutative*:
$a \oplus b = b \oplus a$
⟨*proof*⟩

lemma *asso1*:
assumes $a \oplus b = Some\ ab \wedge b \oplus c = Some\ bc$
shows $ab \oplus c = a \oplus bc$
⟨*proof*⟩

lemma *asso2*:
assumes $a \oplus b = Some\ ab \wedge b \oplus c = None$
shows $ab \oplus c = None$
⟨*proof*⟩

lemma *core-defined*:
get-h $|\varphi|$ = *get-h* φ
get-m $|\varphi|$ = *empty-mask*
⟨*proof*⟩

lemma *state-ext*:
assumes *get-h* a = *get-h* b
and *get-m* a = *get-m* b
shows $a = b$
⟨*proof*⟩

lemma *core-is-smaller*:
Some $x = x \oplus |x|$
⟨*proof*⟩

lemma *core-is-pure*:
Some $|x| = |x| \oplus |x|$
⟨*proof*⟩

lemma *core-sum*:
assumes *Some* $c = a \oplus b$
shows *Some* $|c| = |a| \oplus |b|$
⟨*proof*⟩

lemma *core-max*:
assumes *Some* $x = x \oplus c$
shows $\exists\, r.$ *Some* $|x| = c \oplus r$
⟨*proof*⟩

lemma *positivity*:
assumes $a \oplus b$ = *Some* c
and *Some* $c = c \oplus c$
shows *Some* $a = a \oplus a$
⟨*proof*⟩

lemma *cancellative*:
assumes *Some* $a = b \oplus x$
and *Some* $a = b \oplus y$
and $|x| = |y|$
shows $x = y$
⟨*proof*⟩

lemma *unit-charact*:
get-h unit = *empty-heap*
get-m unit = *empty-mask*
⟨*proof*⟩

lemma *empty-heap-neutral*:
a ++ *empty-heap* = a

⟨*proof*⟩

lemma *unit-neutral*:
Some a = *a* ⊕ *unit*
⟨*proof*⟩

lemma *stableI*:
assumes ⋀*hl*. *ppos* (*get-m* φ *hl*) ⟷ *get-h* φ *hl* ≠ *None*
shows *stable* φ
⟨*proof*⟩

lemma *stable-unit*:
stable unit
⟨*proof*⟩

lemma *stable-sum*:
assumes *stable a*
and *stable b*
and *Some x* = *a* ⊕ *b*
shows *stable x*
⟨*proof*⟩

lemma *multiply-valid*:
assumes *pgte pwrite p*
shows *valid-state* (*multiply-mask p* (*get-m* φ), *get-h* φ)
⟨*proof*⟩

1.4 This state model corresponds to a separation algebra

global-interpretation *PartialSA*: *package-logic plus core unit stable*
defines *greater* (**infixl** ≽ *50*) = *PartialSA.greater*
and *add-set* (**infixl** ⊗ *60*) = *PartialSA.add-set*
and *defined* (**infixl** |#| *60*) = *PartialSA.defined*
and *greater-set* (**infixl** |≫| *50*) = *PartialSA.greater-set*
and *minus* (**infixl** |⊖| *60*) = *PartialSA.minus*
⟨*proof*⟩

lemma *greaterI*:
assumes *larger-heap* (*get-h a*) (*get-h b*)
and *greater-mask* (*get-m a*) (*get-m b*)
shows *a* ≽ *b*
⟨*proof*⟩

lemma *larger-implies-greater-mask-hl*:
assumes *a* ≽ *b*
shows *pgte* (*get-m a hl*) (*get-m b hl*)
⟨*proof*⟩

lemma *larger-implies-larger-heap*:
assumes $a \succeq b$
shows *larger-heap* (*get-h a*) (*get-h b*)
⟨*proof*⟩

lemma *compatibleI*:
assumes *compatible-heaps* (*get-h a*) (*get-h b*)
and *valid-mask* (*add-masks* (*get-m a*) (*get-m b*))
shows $a \mid\#\mid b$
⟨*proof*⟩

end

2 Combinable Magic Wands

Note that, in this theory, assertions are represented as semantic assertions, i.e., as the set of states in which they hold.

theory *CombinableWands*
imports *PartialHeapSA*
begin

2.1 Definitions

type-synonym *sem-assertion* = *state set*

fun *multiply* :: *prat* ⇒ *state* ⇒ *state* **where**
multiply p φ = *Abs-state* (*multiply-mask p* (*get-m φ*), *get-h φ*)

Because we work in an intuitionistic setting, a fraction of an assertion is defined using the upper-closure operator.

fun *multiply-sem-assertion* :: *prat* ⇒ *sem-assertion* ⇒ *sem-assertion* **where**
multiply-sem-assertion p P = *PartialSA.upper-closure* (*multiply p* ‘ *P*)

definition *combinable* :: *sem-assertion* ⇒ *bool* **where**
combinable P ⟷ (∀ *α β*. *ppos α* ∧ *ppos β* ∧ *pgte pwrite* (*padd α β*) ⟶ (*multiply-sem-assertion α P*) ⊗ (*multiply-sem-assertion β P*) ⊆ *multiply-sem-assertion* (*padd α β*) *P*)

definition *scaled* **where**
scaled φ = { *multiply p φ* |*p*. *ppos p* ∧ *pgte pwrite p* }

definition *comp-min-mask* :: *mask* ⇒ (*mask* ⇒ *mask*) **where**
comp-min-mask b a hl = *pmin* (*a hl*) (*comp-one* (*b hl*))

definition *scalable* **where**
scalable w a ⟷ (∀ *φ* ∈ *scaled w*. ¬ *a* |#| *φ*)

definition *R* **where**

$R\ a\ w = (\mathit{if}\ \mathit{scalable}\ w\ a\ \mathit{then}\ w\ \mathit{else}\ \mathit{Abs\text{-}state}\ (\mathit{comp\text{-}min\text{-}mask}\ (\mathit{get\text{-}m}\ a)\ (\mathit{get\text{-}m}\ w),\ \mathit{get\text{-}h}\ w))$

definition *cwand* **where**
$\mathit{cwand}\ A\ B = \{\ w\ |w.\ \forall\, a\ x.\ a \in A \wedge \mathit{Some}\ x = R\ a\ w \oplus a \longrightarrow x \in B\ \}$

definition *wand* :: *sem-assertion* ⇒ *sem-assertion* ⇒ *sem-assertion* **where**
$\mathit{wand}\ A\ B = \{\ w\ |w.\ \forall\, a\ x.\ a \in A \wedge \mathit{Some}\ x = w \oplus a \longrightarrow x \in B\ \}$

definition *intuitionistic* **where**
$\mathit{intuitionistic}\ A \longleftrightarrow (\forall\, a\ b.\ a \succeq b \wedge b \in A \longrightarrow a \in A)$

definition *binary-mask* :: *mask* ⇒ *mask* **where**
$\mathit{binary\text{-}mask}\ \pi\ l = (\mathit{if}\ \pi\ l = \mathit{pwrite}\ \mathit{then}\ \mathit{pwrite}\ \mathit{else}\ \mathit{pnone})$

definition *binary* :: *sem-assertion* ⇒ *bool* **where**
$\mathit{binary}\ A \longleftrightarrow (\forall\, \varphi \in A.\ \mathit{Abs\text{-}state}\ (\mathit{binary\text{-}mask}\ (\mathit{get\text{-}m}\ \varphi),\ \mathit{get\text{-}h}\ \varphi) \in A)$

2.2 Lemmas

lemma *wand-equiv-def*:
$\mathit{wand}\ A\ B = \{\ \varphi\ |\varphi.\ A \otimes \{\varphi\} \subseteq B\ \}$
⟨*proof*⟩

lemma *w-in-scaled*:
$w \in \mathit{scaled}\ w$
⟨*proof*⟩

lemma *non-scalable-instantiate*:
assumes ¬ *scalable w a*
shows $\exists\, p.\ \mathit{ppos}\ p \wedge \mathit{pgte}\ \mathit{pwrite}\ p \wedge a\ |\#|\ \mathit{multiply}\ p\ w$
⟨*proof*⟩

lemma *compatible-same-mask*:
assumes *valid-mask* (*add-masks a w*)
shows *w* = *comp-min-mask a w*
⟨*proof*⟩

lemma *R-smaller*:
$w \succeq R\ a\ w$
⟨*proof*⟩

lemma *R-compatible-same*:
assumes *a* |#| *w*
shows *R a w* = *w*
⟨*proof*⟩

lemma *in-cwand*:
assumes $\bigwedge a\ x.\ a \in A \wedge \mathit{Some}\ x = R\ a\ w \oplus a \Longrightarrow x \in B$

shows $w \in cwand\ A\ B$
⟨*proof*⟩

lemma *wandI*:
assumes $\bigwedge a\ x.\ a \in A \wedge Some\ x = a \oplus w \Longrightarrow x \in B$
shows $w \in wand\ A\ B$
⟨*proof*⟩

lemma *non-scalable-R-charact*:
assumes $\neg$ *scalable w a*
shows *get-m* $(R\ a\ w) =$ *comp-min-mask* $(get\text{-}m\ a)\ (get\text{-}m\ w) \wedge$ *get-h* $(R\ a\ w) =$ *get-h w*
⟨*proof*⟩

lemma *valid-bin*:
valid-state (*binary-mask* (*get-m a*), *get-h a*)
⟨*proof*⟩

lemma *in-multiply-sem*:
assumes $x \in multiply\text{-}sem\text{-}assertion\ p\ A$
shows $\exists\, a \in A.\ x \succeq multiply\ p\ a$
⟨*proof*⟩

lemma *get-h-multiply*:
assumes *pgte pwrite p*
shows *get-h* $(multiply\ p\ x) =$ *get-h x*
⟨*proof*⟩

lemma *in-multiply-refl*:
assumes $x \in A$
shows $multiply\ p\ x \in multiply\text{-}sem\text{-}assertion\ p\ A$
⟨*proof*⟩

lemma *get-m-smaller*:
assumes *pgte pwrite p*
shows *get-m* $(multiply\ p\ a)\ hl = pmult\ p\ (get\text{-}m\ a\ hl)$
⟨*proof*⟩

lemma *get-m-smaller-mask*:
assumes *pgte pwrite p*
shows *get-m* $(multiply\ p\ a) = multiply\text{-}mask\ p\ (get\text{-}m\ a)$
⟨*proof*⟩

lemma *multiply-order*:
assumes *pgte pwrite p*
and $a \succeq b$
shows $multiply\ p\ a \succeq multiply\ p\ b$
⟨*proof*⟩

lemma *multiply-twice*:
assumes *pgte pwrite a* ∧ *pgte pwrite b*
shows *multiply a* (*multiply b x*) = *multiply* (*pmult a b*) *x*
⟨*proof*⟩

lemma *valid-mask-add-comp-min*:
assumes *valid-mask a*
and *valid-mask b*
shows *valid-mask* (*add-masks* (*comp-min-mask b a*) *b*)
⟨*proof*⟩

2.3 The combinable wand is stronger than the original wand

lemma *cwand-stronger*:
cwand A B ⊆ *wand A B*
⟨*proof*⟩

2.4 The combinable wand is the same as the original wand when the left-hand side is binary

lemma *binary-same*:
assumes *binary A*
and *intuitionistic B*
shows *wand A B* ⊆ *cwand A B*
⟨*proof*⟩

2.5 The combinable wand is combinable

lemma *combinableI*:
assumes ⋀*a b*. *ppos a* ∧ *ppos b* ∧ *padd a b* = *pwrite* ⟹ (*multiply-sem-assertion a* (*cwand A B*)) ⊗ (*multiply-sem-assertion b* (*cwand A B*)) ⊆ *cwand A B*
shows *combinable* (*cwand A B*)
⟨*proof*⟩

lemma *combinable-cwand*:
assumes *combinable B*
and *intuitionistic B*
shows *combinable* (*cwand A B*)
⟨*proof*⟩

2.6 Theorems

The following theorem is crucial to use the package logic [4] to automatically compute footprints of combinable wands.

theorem *R-mono-transformer*:
PartialSA.mono-transformer (*R a*)
⟨*proof*⟩

theorem *properties-of-combinable-wands*:

assumes *intuitionistic B*
shows *combinable B* $\Longrightarrow$ *combinable* (*cwand A B*)
and *cwand A B* $\subseteq$ *wand A B*
and *binary A* $\Longrightarrow$ *cwand A B* = *wand A B*
⟨*proof*⟩

end

References

[1] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis (SAS)*, pages 55–72, 2003.

[2] J. T. Boyland. Semantics of fractional permissions with nesting. *Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):22:1–22:33, 2010.

[3] J. Brotherston, D. Costa, A. Hobor, and J. Wickerson. Reasoning over permissions regions in concurrent separation logic. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification (CAV)*, 2020.

[4] T. Dardinier, G. Parthasarathy, N. Weeks, P. Müller, and A. J. Summers. Sound Automation of Magic Wands. In *Computer Aided Verification (CAV)*, 2022. To appear.

[5] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Principles of Programming Languages (POPL)*, pages 271–282, 2011.

[6] X.-B. Le and A. Hobor. Logical reasoning for disjoint permissions. In A. Ahmed, editor, *European Symposium on Programming (ESOP)*, 2018.