lib/memory-range/memory.ath

```
### Memory theory

#-------------------------------------------------------------------------

domain (Memory S)

module Memory {
  domain (Change S)
  domain (Loc S)

  declare \: (S) [(Memory S) (Change S)] -> (Memory S)
  declare \\: (S, T) [(Memory S) T] -> T
  declare at: (S) [(Memory S) (Loc S)] -> S

  define [M M1 M2 M3 a b c x y] :=
    [?M:(Memory 'S) ?M1:(Memory 'S) ?M2:(Memory 'S)
     ?M3:(Memory 'S) ?a:(Loc 'S) ?b:(Loc 'S) ?c:(Loc 'S)
     ?x:'S ?y:'S]

  define equality :=
    (forall M1 M2 . (forall a . M1 at a = M2 at a) <==> M1 = M2)

  declare <-: (S) [(Loc S) S] -> (Change S)

  module assign {
    define axioms :=
      (fun [((M \ a <- x) at b) =
            [x                  when (a = b)
             (M at b)           when (a =/= b)]])
    define [equal unequal] := axioms
  }

  define theory := (make-theory [] [equality assign.equal assign.unequal])

#-------------------------------------------------------------------------
  declare swap: (S) [(Loc S) (Loc S)] -> (Change S)

  module swap {

    define axioms :=
      (fun [((M \ (swap a b)) at c) =
            [(M at b)               when (a = c)
             (M at a)               when (b = c)
             (M at c)               when (a =/= c & b =/= c)]])
    define [equal1 equal2 unequal] := axioms

    (add-axioms theory axioms)
  }

#-------------------------------------------------------------------------
# Theorems:

define Double-assign :=
  (forall b M a x y . ((M \ a <- x) \ a <- y) at b = (M \ a <- y) at b)
define Direct-double-assign :=
  (forall M a x y . (M \ a <- x) \ a <- y = M \ a <- y)
define Self-assign :=
  (forall M a b . (M \ a <- M at a) at b = M at b)
define Direct-self-assign := (forall M a . M \ a <- M at a = M)
define Double-swap :=
  (forall c M a b .
    ((M \ (swap a b)) \ (swap b a)) at c = M at c)
define Direct-double-swap :=
  (forall M a b . (M \ (swap a b)) \ (swap b a) = M)

define theorems := [Double-assign Direct-double-assign Self-assign
                    Direct-self-assign Double-swap Direct-double-swap]
```

```
define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [at \ <- swap] := (adapt [at \ <- swap])}
      match theorem {
        (val-of Double-assign) =>
        pick-any b:(Loc 'S) M:(Memory 'S) a:(Loc 'S) x:'S y:'S
          (!two-cases
             assume (a = b)
               (!chain [(((M \ a <- x) \ a <- y) at b)
                        <-- (((M \ a <- x) \ a <- y) at a) [(a = b)]
                        --> y                      [assign.equal]
                        <-- ((M \ a <- y) at a)    [assign.equal]
                        --> ((M \ a <- y) at b)    [(a = b)]])
             assume (a =/= b)
               (!chain [(((M \ a <- x) \ a <- y) at b)
                        --> ((M \ a <- x) at b)    [assign.unequal]
                        --> (M at b)               [assign.unequal]
                        <-- ((M \ a <- y) at b)    [assign.unequal]]))
      | (val-of Direct-double-assign) =>
        pick-any M:(Memory 'S) i:(Loc 'S) x:'S y:'S
          let {DA := (!prove Double-assign);
               A := pick-any a:(Loc 'S)
                      (!chain [(((M \ i <- x) \ i <- y) at a)
                               --> ((M \ i <- y) at a) [DA]])}
          (!chain->
           [A <==> ((M \ i <- x) \ i <- y = M \ i <- y)
                                   [equality]])
      | (val-of Self-assign) =>
        pick-any M:(Memory 'S) a:(Loc 'S) b:(Loc 'S)
          let {goal := ((M \ a <- (M at a)) at b = M at b)}
          (!two-cases
            assume (a = b)
              (!chain [((M \ a <- (M at a)) at b)
                       <-- ((M \ a <- (M at a)) at a)    [(a = b)]
                       --> (M at a)                      [assign.equal]
                       --> (M at b)                      [(a = b)]])
            (!chain [(a =/= b) ==> goal [assign.unequal]]))
      | (val-of Direct-self-assign) =>
         pick-any M:(Memory 'S) i:(Loc 'S)
           let {SA := (!prove Self-assign);
                A := pick-any a:(Loc 'S)
                       (!chain [((\ M i <- (M at i)) at a)
                                --> (M at a) [SA]])}
           (!chain->
            [A ==> ((M \ i <- (M at i)) = M) [equality]])
      | (val-of Double-swap) =>
        pick-any c:(Loc 'S) M:(Memory 'S) a:(Loc 'S) b:(Loc 'S)
          (!three-cases
            assume (a = c)
              (!chain [(((M \ (swap a b)) \ (swap b a)) at c)
                       <-- (((M \ (swap a b)) \ (swap b a)) at a) [(a = c)]
                       --> ((M \ (swap a b)) at b)       [swap.equal2]
                       --> (M at a)                      [swap.equal2]
                       --> (M at c)                      [(a = c)]])
            assume (b = c)
              (!chain [(((M \ (swap a b)) \ (swap b a)) at c)
                       <-- (((M \ (swap a b)) \ (swap b a)) at b) [(b = c)]
                       --> ((M \ (swap a b)) at a)       [swap.equal1]
                       --> (M at b)                      [swap.equal1]
                       --> (M at c)                      [(b = c)]])
            assume (a =/= c & b =/= c)
              (!chain [(((M \ (swap a b)) \ (swap b a)) at c)
                       --> ((M \ (swap a b)) at c)       [swap.unequal]
                       --> (M at c)                      [swap.unequal]]))
      | (val-of Direct-double-swap) =>
        pick-any M:(Memory 'S) a:(Loc 'S) b:(Loc 'S)
          let {DS := (!prove Double-swap);
               A := pick-any c:(Loc 'S)
                      (!chain [(((M \ (swap a b)) \ (swap b a)) at c)
```

```
                               --> (M at c)    [DS]])}
          (!chain-> [A ==> (((M \ (swap a b)) \ (swap b a)) = M)
                                               [equality]])
      }

(add-theorems theory |{theorems := proofs}|)

} # Memory
```