Formal specification of the Cap9 kernel

Mikhail Mandrykin Ilya Shchepetkov

June 14, 2019

Contents

1 Introduction

This is an Isabelle/HOL theory that describes and proves the correctness of the Cap9 kernel specification.

2 Preliminaries

```
theory Cap9
imports
  "HOL−Word.Word"
  "HOL−Library.Adhoc_Overloading"
  "HOL−Library.DAList"
  "Word_Lib/Word_Lemmas"
begin
```

2.1 Type class instantiations

Instantiate *len* type class to extract lengths from word types avoiding repeated explicit numeric specification of the length e.g. *LENGTH*(*byte*) or *LENGTH*('*a* :: *len word*) instead of *8* or *LENGTH*('*a*), where '*a* cannot be directly extracted from a type such as '*a word*.

```
instantiation word :: (len) len begin
definition len_word[simp]: "len_of (_ :: 'a::len word itself) = LENGTH('a)"
instance by (standard, simp)
end

lemma len_word': "LENGTH('a::len word) = LENGTH('a)" by (rule len_word)
```

Instantiate *size* type class for types of the form '*a itself*. This allows us to parametrize operations by word lengths using the dummy variables of type '*a word itself*. The operations cannot be directly parametrized by numbers as there is no lifting from term numbers to type numbers due to the lack of dependent types.

```
instantiation itself :: (len) size begin
definition size_itself where [simp, code]: "size (n::'a::len itself) = LENGTH('a)"
instance ..
end

declare unat_word_ariths[simp] word_size[simp] is_up_def[simp] wsst_TYs(1,2)[simp]
```

2.2 Word width

We introduce definition of the least number of bits to hold the current value of a word. This is needed because in our specification we often word with *UCAST*('*a* → '*b*)'ed values (right aligned subranges of bits), largely again due to the lack of dependent types (or true type-level functions), e.g. the it's hard to specify that the length of $a \bowtie b$ (where $\bowtie$ stands for concatenation) is the sum of the length of a and b, since length is a type parameter and there's no equivalent of sum on the type level. So we instead fix the length of $a \bowtie b$ to be the maximum possible one (say, 32 bytes) and then use conditions of the form *width* $a \le s$ to specify that the actual "size" of a is s.

```
definition "width w ≡ LEAST n. unat w < 2 ^ n" for w :: "'a::len word"

lemma widthI[intro]: "⟦⋀ u. u < n ⟹ 2 ^ u ≤ unat w; unat w < 2 ^ n⟧ ⟹ width w = n"
  unfolding width_def Least_def
  using not_le
  apply (intro the_equality, blast)
  by (meson nat_less_le)

lemma width_wf[simp]: "∃! n. (∀ u < n. 2 ^ u ≤ unat w) ∧ unat w < 2 ^ n"
  (is "?Ex1 (unat w)")
proof (induction ("unat w"))
  case 0
  show "?Ex1 0" by (intro ex1I[of _ 0], auto)
next
  case (Suc x)
  then obtain n where x:"(∀ u<n. 2 ^ u ≤ x) ∧ x < 2 ^ n " by auto
```

```
  show "?Ex1 (Suc x)"
  proof (cases "Suc x < 2 ^ n")
    case True
    thus "?Ex1 (Suc x)"
      using x
      apply (intro ex1I[of _ "n"], auto)
      by (meson Suc_lessD leD linorder_neqE_nat)
  next
    case False
    thus "?Ex1 (Suc x)"
      using x
      apply (intro ex1I[of _ "Suc n"], auto simp add: less_Suc_eq)
      apply (intro antisym)
       apply (metis One_nat_def Suc_lessI Suc_n_not_le_n leI numeral_2_eq_2 power_increasing_iff)
      by (metis Suc_lessD le_antisym not_le not_less_eq_eq)
  qed
qed

lemma width_iff[iff]: "(width w = n) = ((∀ u < n. 2 ^ u ≤ unat w) ∧ unat w < 2 ^ n)"
  using width_wf widthI by metis

lemma width_le_size: "width x ≤ size x"
proof–
  {
    assume "size x < width x"
    hence "2 ^ size x ≤ unat x" using width_iff by metis
    hence "2 ^ size x ≤ uint x" unfolding unat_def by simp
  }
  thus ?thesis using uint_range_size[of x] by (force simp del:word_size)
qed

lemma width_le_size'[simp]: "size x ≤ n ⟹ width x ≤ n" by (insert width_le_size[of x], simp)

lemma nth_width_high[simp]: "width x ≤ i ⟹ ¬ x !! i"
proof (cases "i < size x")
  case False
  thus ?thesis by (simp add: test_bit_bin')
next
  case True
  hence "(x < 2 ^ i) = (unat x < 2 ^ i)"
    unfolding unat_def
    using word_2p_lem by fastforce
  moreover assume "width x ≤ i"
  then obtain n where "unat x < 2 ^ n" and "n ≤ i" using width_iff by metis
  hence "unat x < 2 ^ i"
    by (meson le_less_trans nat_power_less_imp_less not_less zero_less_numeral)
  ultimately show ?thesis using bang_is_le by force
qed

lemma width_zero[iff]: "(width x = 0) = (x = 0)"
proof
  show "width x = 0 ⟹ x = 0" using nth_width_high[of x] word_eq_iff[of x 0] nth_0 by (metis le0)
  show "x = 0 ⟹ width x = 0" by simp
qed

lemma width_zero'[simp]: "width 0 = 0" by simp

lemma width_one[simp]: "width 1 = 1" by simp

lemma high_zeros_less: "(∀ i ≥ u. ¬ x !! i) ⟹ unat x < 2 ^ u"
```

```
  (is "?high ⟹ _") for x :: "'a::len word"
proof-
  assume ?high
  have size:"size (mask u :: 'a word) = size x" by simp
  {
    fix i
    from ‹?high› have "(x AND mask u) !! i = x !! i"
      using nth_mask[of u i] size test_bit_size[of x i]
      by (subst word_ao_nth) (elim allE[of _ i], auto)
  }
  with ‹?high› have "x AND mask u = x" using word_eq_iff by blast
  thus ?thesis unfolding unat_def using mask_eq_iff by auto
qed

lemma nth_width_msb[simp]: "x ≠ 0 ⟹ x !! (width x - 1)"
proof (rule ccontr)
  fix x :: "'a word"
  assume "x ≠ 0"
  hence width:"width x > 0" using width_zero by fastforce
  assume "¬ x !! (width x - 1)"
  with width have "∀ i ≥ width x - 1. ¬ x !! i"
    using nth_width_high[of x] antisym_conv2 by fastforce
  hence "unat x < 2 ^ (width x - 1)" using high_zeros_less[of "width x - 1" x] by simp
  moreover from width have "unat x ≥ 2 ^ (width x - 1)" using width_iff[of x "width x"] by simp
  ultimately show False by simp
qed

lemma width_iff': "((∀ i > u. ¬ x !! i) ∧ x !! u) = (width x = Suc u)"
proof (rule; (elim conjE | intro conjI))
  assume "x !! u" and "∀ i > u. ¬ x !! i"
  show "width x = Suc u"
  proof (rule antisym)
    from ‹x !! u› show "width x ≥ Suc u" using not_less nth_width_high by force
    from ‹x !! u› have "x ≠ 0" by auto
    with ‹∀ i > u. ¬ x !! i› have "width x - 1 ≤ u" using not_less nth_width_msb by metis
    thus "width x ≤ Suc u" by simp
  qed
next
  assume "width x = Suc u"
  show "∀i>u. ¬ x !! i" by (simp add:‹width x = Suc u›)
  from ‹width x = Suc u› show "x !! u" using nth_width_msb width_zero
    by (metis diff_Suc_1 old.nat.distinct(2))
qed

lemma width_word_log2: "x ≠ 0 ⟹ width x = Suc (word_log2 x)"
  using word_log2_nth_same word_log2_nth_not_set width_iff' test_bit_size
  by metis

lemma width_ucast[OF refl, simp]: "uc = ucast ⟹ is_up uc ⟹ width (uc x) = width x"
  by (metis uint_up_ucast unat_def width_def)

lemma width_ucast'[OF refl, simp]:
  "uc = ucast ⟹ width x ≤ size (uc x) ⟹ width (uc x) = width x"
proof-
  have "unat x < 2 ^ width x" unfolding width_def by (rule LeastI_ex, auto)
  moreover assume "width x ≤ size (uc x)"
  ultimately have "unat x < 2 ^ size (uc x)" by (simp add: less_le_trans)
  moreover assume "uc = ucast"
  ultimately have "unat x = unat (uc x)" by (metis unat_ucast mod_less word_size)
  thus ?thesis unfolding width_def by simp
```

```
qed

lemma width_lshift[simp]:
  "⟦x ≠ 0; n ≤ size x - width x⟧ ⟹ width (x << n) = width x + n"
  (is "⟦_; ?nbound⟧ ⟹ _")
proof-
  assume "x ≠ 0"
  hence 0:"width x = Suc (width x - 1)" using width_zero by (metis Suc_pred' neq0_conv)
  from ‹x ≠ 0› have 1:"width x > 0" by (auto intro:gr_zeroI)
  assume ?nbound
  {
    fix i
    from ‹?nbound› have "i ≥ size x ⟹ ¬ x !! (i - n)" by (auto simp add:le_diff_conv2)
    hence "(x << n) !! i = (n ≤ i ∧ x !! (i - n))" using nth_shiftl'[of x n i] by auto
  } note corr = this
  hence "∀ i > width x + n - 1. ¬ (x << n) !! i" by auto
  moreover from corr have "(x << n) !! (width x + n - 1)"
    using width_iff'[of "width x - 1" x] 1
    by auto
  ultimately have "width (x << n) = Suc (width x + n - 1)" using width_iff' by auto
  thus ?thesis using 0 by simp
qed

lemma width_lshift'[simp]: "n ≤ size x - width x ⟹ width (x << n) ≤ width x + n"
  using width_zero width_lshift shiftl_0 by (metis eq_iff le0)

lemma width_or[simp]: "width (x OR y) = max (width x) (width y)"
proof-
  {
    fix a b
    assume "width x = Suc a" and "width y = Suc b"
    hence "width (x OR y) = Suc (max a b)"
      using width_iff' word_ao_nth[of x y] max_less_iff_conj[of "a" "b"]
      by (metis (no_types) max_def)
  } note succs = this
  thus ?thesis
  proof (cases "width x = 0 ∨ width y = 0")
    case True
    thus ?thesis using width_zero word_log_esimps(3,9) by (metis max_0L max_0R)
  next
    case False
    with succs show ?thesis by (metis max_Suc_Suc not0_implies_Suc)
  qed
qed
```

2.3 Right zero-padding

Here's the first time we use *width*. If x is a value of size n right-aligned in a word of size $s = size\ x$ (note there's nowhere to keep the value n, since the size of x is some $s \geq n$, so we require it to be provided explicitly), then *rpad* n x will move the value x to the left. For the operation to be correct (no losing of significant higher bits) we need the precondition $width\ x \leq n$ in all the lemmas, hence the need for *width*.

```
definition rpad where "rpad n x ≡ x << size x - n"

lemma rpad_low[simp]: "⟦width x ≤ n; i < size x - n⟧ ⟹ ¬ (rpad n x) !! i"
  unfolding rpad_def by (simp add:nth_shiftl)

lemma rpad_high[simp]:
  "⟦width x ≤ n; n ≤ size x; size x - n ≤ i⟧ ⟹ (rpad n x) !! i = x !! (i + n - size x)"
```

```
  (is "⟦?xbound; ?nbound; i ≥ ?ibound⟧ ⟹ ?goal i")
proof-
  fix i
  assume ?xbound ?nbound and "i ≥ ?ibound"
  moreover from ‹?nbound› have "i + n − size x = i − ?ibound" by simp
  moreover from ‹?xbound› have "x !! (i + n − size x) ⟹ i < size x" by − (rule ccontr, simp)
  ultimately show "?goal i" unfolding rpad_def by (subst nth_shiftl', metis)
qed

lemma rpad_inj: "⟦width x ≤ n; width y ≤ n; n ≤ size x⟧ ⟹ rpad n x = rpad n y ⟹ x = y"
  (is "⟦?xbound; ?ybound; ?nbound; _⟧ ⟹ _")
  unfolding inj_def word_eq_iff
proof (intro allI impI)
  fix i
  let ?i' = "i + size x − n"
  assume ?xbound ?ybound ?nbound
  assume "∀j < LENGTH('a). rpad n x !! j = rpad n y !! j"
  hence "⋀ j. rpad n x !! j = rpad n y !! j" using test_bit_bin by blast
  from this[of ?i'] and ‹?xbound› ‹?ybound› ‹?nbound› show "x !! i = y !! i" by simp
qed
```

2.4 Spanning concatenation

```
abbreviation ucastl ("'(ucast')_ _" [1000, 100] 100) where
  "(ucast)_l a ≡ ucast a :: 'b word" for l :: "'b::len0 itself"

notation (input) ucastl ("'(ucast')_ _" [1000, 100] 100)

definition pad_join :: "'a::len word ⇒ nat ⇒ 'c::len itself ⇒ 'b::len word ⇒ 'c word"
  ("_ _◇_ _" [60, 1000, 1000, 61] 60) where
  "x _n◇_l y ≡ rpad n (ucast x) OR ucast y"

notation (input) pad_join ("_ _◇_ _" [60, 1000, 1000, 61] 60)

lemma pad_join_high:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n; size l − n ≤ i⟧
   ⟹ (a _n◇_l b) !! i = a !! (i + n − size l)"
  unfolding pad_join_def
  using nth_ucast nth_width_high by fastforce

lemma pad_join_high'[simp]:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n⟧ ⟹ a !! i = (a _n◇_l b) !! (i + size l − n)"
  using pad_join_high[of a n l b "i + size l − n"] by simp

lemma pad_join_mid[simp]:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n; width b ≤ i; i < size l − n⟧
   ⟹ ¬ (a _n◇_l b) !! i"
  unfolding pad_join_def by auto

lemma pad_join_low[simp]:
  "⟦width a ≤ n; n ≤ size l; width b ≤ size l − n; i < width b⟧ ⟹ (a _n◇_l b) !! i = b !! i"
  unfolding pad_join_def by (auto simp add: nth_ucast)

lemma pad_join_inj:
  assumes eq:"a _n◇_l b = c _n◇_l d"
  assumes a:"width a ≤ n" and c:"width c ≤ n"
  assumes n: "n ≤ size l"
  assumes b:"width b ≤ size l − n"
  assumes d:"width d ≤ size l − n"
  shows   "a = c" and "b = d"
proof-
```

```
  from eq have eq':"⋀j. (a n◇l b) !! j = (c n◇l d) !! j"
    using test_bit_bin unfolding word_eq_iff by auto
  moreover from a n b
  have "⋀ i. a !! i = (a n◇l b) !! (i + size l − n)" by simp
  moreover from c n d
  have "⋀ i. c !! i = (c n◇l d) !! (i + size l − n)" by simp
  ultimately show "a = c" unfolding word_eq_iff by auto

  {
    fix i
    from a n b have "i < width b ⟹ b !! i = (a n◇l b) !! i" by simp
    moreover from c n d have "i < width d ⟹ d !! i = (c n◇l d) !! i" by simp
    moreover have "i ≥ width b ⟹ ¬ b !! i" and "i ≥ width d ⟹ ¬ d !! i" by auto
    ultimately have "b !! i = d !! i"
      using eq'[of i] b d
        pad_join_mid[of a n l b i, OF a n b]
        pad_join_mid[of c n l d i, OF c n d]
      by (meson leI less_le_trans)
  }
  thus "b = d" unfolding word_eq_iff by simp
qed

lemma pad_join_inj'[dest!]:
 "⟦a n◇l b = c n◇l d;
   width a ≤ n; width c ≤ n; n ≤ size l;
   width b ≤ size l − n;
   width d ≤ size l − n⟧ ⟹ a = c ∧ b = d"
  apply (rule conjI)
  subgoal by (frule (4) pad_join_inj(1))
  by (frule (4) pad_join_inj(2))

lemma pad_join_and[simp]:
  assumes "width x ≤ n" "n ≤ m" "width a ≤ m" "m ≤ size l" "width b ≤ size l − m"
  shows   "(a m◇l b) AND rpad n x = rpad m a AND rpad n x"
  unfolding word_eq_iff
proof ((subst word_ao_nth)+, intro allI impI)
  from assms have 0:"n ≤ size x" by simp
  from assms have 1:"m ≤ size a" by simp
  fix i
  assume "i < LENGTH('a)"
  from assms show "((a m◇l b) !! i ∧ rpad n x !! i) = (rpad m a !! i ∧ rpad n x !! i)"
    using rpad_low[of x n i, OF assms(1)] rpad_high[of x n i, OF assms(1) 0]
          rpad_low[of a m i, OF assms(3)] rpad_high[of a m i, OF assms(3) 1]
          pad_join_high[of a m l b i, OF assms(3,4,5)]
          size_itself_def[of l] word_size[of x] word_size[of a]
    by (metis add.commute add_lessD1 le_Suc_ex le_diff_conv not_le)
qed
```

2.5 Deal with partially undefined results

```
definition restrict :: "'a::len word ⇒ nat set ⇒ 'a word" (infixl "↾" 60) where
  "restrict x s ≡ BITS i. i ∈ s ∧ x !! i"

lemma nth_restrict[iff]: "(x ↾ s) !! n = (n ∈ s ∧ x !! n)"
  unfolding restrict_def
  by (simp add: bang_conj_lt test_bit.eq_norm)

lemma restrict_inj2:
  assumes eq:"f x₁ y₁ OR v₁ ↾ s = f x₂ y₂ OR v₂ ↾ s"
  assumes fi:"⋀ x y i. i ∈ s ⟹ ¬ f x y !! i"
  assumes inj:"⋀ x₁ y₁ x₂ y₂. f x₁ y₁ = f x₂ y₂ ⟹ x₁ = x₂ ∧ y₁ = y₂"
```

```
  shows   "x1 = x2 ∧ y1 = y2"
proof-
  from eq and fi have "f x1 y1 = f x2 y2" unfolding word_eq_iff by auto
  with inj show ?thesis .
qed

lemmas restrict_inj_pad_join[dest] = restrict_inj2[of "λ x y. x _◊_ y"]
```

2.6 Plain concatenation

```
definition join :: "'a::len word ⇒ 'c::len itself ⇒ nat ⇒ 'b::len word ⇒ 'c word"
  ("_ _⋈_ _" [62,1000,1000,61] 61) where
  "(a l⋈n b) ≡ (ucast a << n) OR (ucast b)"

notation (input) join ("_ _⋈_ _" [62,1000,1000,61] 61)

lemma width_join:
  "⟦width a + n ≤ size l; width b ≤ n⟧ ⟹ width (a l⋈n b) ≤ width a + n"
  (is "⟦?abound; ?bbound⟧ ⟹ _")
proof-
  assume ?abound and ?bbound
  moreover hence "width b ≤ size l" by simp
  ultimately show ?thesis
    using width_lshift'[of n "(ucast)l a"]
    unfolding join_def
    by simp
qed

lemma width_join'[simp]:
  "⟦width a + n ≤ size l; width b ≤ n; width a + n ≤ q⟧ ⟹ width (a l⋈n b) ≤ q"
  by (drule (1) width_join, simp)

lemma join_high[simp]:
  "⟦width a + n ≤ size l; width b ≤ n; width a + n ≤ i⟧ ⟹ ¬ (a l⋈n b) !! i"
  by (drule (1) width_join, simp)

lemma join_mid:
  "⟦width a + n ≤ size l; width b ≤ n; n ≤ i; i < width a + n⟧ ⟹ (a l⋈n b) !! i = a !! (i - n)"
  apply (subgoal_tac "i < size ((ucast)l a) ∧ size ((ucast)l a) = size l")
  unfolding join_def
  using word_ao_nth nth_ucast nth_width_high nth_shiftl'
   apply (metis less_imp_diff_less order_trans word_size)
  by simp

lemma join_mid'[simp]:
  "⟦width a + n ≤ size l; width b ≤ n⟧ ⟹ a !! i = (a l⋈n b) !! (i + n)"
  using join_mid[of a n l b "i + n"] nth_width_high[of a i] join_high[of a n l b "i + n"]
  by force

lemma join_low[simp]:
  "⟦width a + n ≤ size l; width b ≤ n; i < n⟧ ⟹ (a l⋈n b) !! i = b !! i"
  unfolding join_def
  by (simp add: nth_shiftl nth_ucast)

lemma join_inj:
  assumes eq:"a l⋈n b = c l⋈n d"
  assumes "width a + n ≤ size l" and "width b ≤ n"
  assumes "width c + n ≤ size l" and "width d ≤ n"
  shows   "a = c" and "b = d"
proof-
  from assms show "a = c" unfolding word_eq_iff using join_mid' eq by metis
```

```
  from assms show "b = d" unfolding word_eq_iff using join_low nth_width_high
    by (metis eq less_le_trans not_le)
qed

lemma join_inj'[dest!]:
  "⟦a _l⋈_n b = c _l⋈_n d;
    width a + n ≤ size l; width b ≤ n;
    width c + n ≤ size l; width d ≤ n⟧ ⟹ a = c ∧ b = d"
  apply (rule conjI)
  subgoal by (frule (4) join_inj(1))
  by (frule (4) join_inj(2))

lemma join_and:
  assumes "width x ≤ n" "n ≤ size l" "k ≤ size l" "m ≤ k"
          "n ≤ k − m" "width a ≤ k − m" "width a + m ≤ k" "width b ≤ m"
  shows   "rpad k (a _l⋈_m b) AND rpad n x = rpad (k − m) a AND rpad n x"
  unfolding word_eq_iff
proof ((subst word_ao_nth)+, intro allI impI)
  from assms have 0:"n ≤ size x" by simp
  from assms have 1:"k − m ≤ size a" by simp
  from assms have 2:"width (a _l⋈_m b) ≤ k" by simp
  from assms have 3:"k ≤ size (a _l⋈_m b)" by simp
  from assms have 4:"width a + m ≤ size l" by simp
  fix i
  assume "i < LENGTH('a)"
  moreover with assms have "i + k − size (a _l⋈_m b) − m = i + (k − m) − size a" by simp
  moreover from assms have "i + k − size (a _l⋈_m b) < m ⟹ i < size x − n" by simp
  moreover from assms have
    "⟦i ≥ size l − k; m ≤ i + k − size (a _l⋈_m b)⟧ ⟹ size a − (k − m) ≤ i" by simp
  moreover from assms have "width a + m ≤ i + k − size (a _l⋈_m b) ⟹ ¬ rpad (k − m) a !! i"
    by (simp add: nth_shiftl' rpad_def)
  moreover from assms have "¬ i ≥ size l − k ⟹ i < size x − n" by simp
  ultimately show "(rpad k (a _l⋈_m b) !! i ∧ rpad n x !! i) =
                   (rpad (k − m) a !! i ∧ rpad n x !! i)"
    using assms
          rpad_high[of x n i, OF assms(1) 0] rpad_low[of x n i, OF assms(1)]
          rpad_high[of a "k − m" i, OF assms(6) 1] rpad_low[of a "k − m" i, OF assms(6)]
          rpad_high[of "a _l⋈_m b" k i, OF 2 3] rpad_low[of "a _l⋈_m b" k i, OF 2]
          join_high[of a m l b "i + k − size (a _l⋈_m b)", OF 4 assms(8)]
          join_mid[of a m l b "i + k − size (a _l⋈_m b)", OF 4 assms(8)]
          join_low[of a m l b "i + k − size (a _l⋈_m b)", OF 4 assms(8)]
          size_itself_def[of l] word_size[of x] word_size[of a] word_size[of "a _l⋈_m b"]
    by (metis not_le)
qed

lemma join_and'[simp]:
   "⟦width x ≤ n; n ≤ size l; k ≤ size l; m ≤ k;
     n ≤ k − m; width a ≤ k − m; width a + m ≤ k; width b ≤ m⟧ ⟹
    rpad k (a _l⋈_m b) AND rpad n x = rpad (k − m) (ucast a) AND rpad n x"
  using join_and[of x n l k m "ucast a" b] unfolding join_def
  by (simp add: ucast_id)
```

3 Data formats

This section contains definitions of various data formats used in the specification.

3.1 Common notation

Before we proceed some common notation that would be used later will be established.

3.1.1 Machine words

Procedure keys are represented as 24-byte (192 bits) machine words.

```
type_synonym word24 = "192 word" — 24 bytes
type_synonym key = word24
```

Byte is 8-bit machine word.

```
type_synonym byte = "8 word"
```

32-byte machine words that are used to model keys and values of the storage.

```
type_synonym word32 = "256 word" — 32 bytes
```

Storage is a function that takes a 32-byte word (key) and returns another 32-byte word (value).

```
type_synonym storage = "word32 ⇒ word32"
```

3.1.2 Concatenation operations

Specialize previously defined general concatenation operations for the fixed result size of 32 bytes. Thus we avoid lots of redundant type annotations for every intermediate result (note that these intermediate types cannot be inferred automatically (in a purely Hindley-Milner setting as in Isabelle), because this would require type-level functions/dependent types).

```
abbreviation "len (_ :: 'a::len word itself) ≡ TYPE('a)"

no_notation join ("_ _⋈_ _" [62,1000,1000,61] 61)
no_notation (input) join ("_ _⋈_ _" [62,1000,1000,61] 61)

abbreviation join32 ("_ ⋈_ _" [62,1000,61] 61) where
  "a ⋈n b ≡ join a (len TYPE(word32)) (n ∗ 8) b"
abbreviation (output) join32_out ("_ ⋈_ _" [62,1000,61] 61) where
  "join32_out a n b ≡ join a (TYPE(256)) n b"
notation (input) join32 ("_ ⋈_ _" [62,1000,61] 61)

no_notation pad_join ("_ _◊_ _" [60,1000,1000,61] 60)
no_notation (input) pad_join ("_ _◊_ _" [60,1000,1000,61] 60)

abbreviation pad_join32 ("_ _◊ _" [60,1000,61] 60) where
  "a n◊ b ≡ pad_join a (n ∗ 8) (len TYPE(word32)) b"
abbreviation (output) pad_join32_out ("_ _◊ _" [60,1000,61] 60) where
  "pad_join32_out a n b ≡ pad_join a n (TYPE(256)) b"
notation (input) pad_join32 ("_ _◊ _" [60,1000,61] 60)
```

Override treatment of hexidecimal numeric constants to make them monomorphic words of fixed length, mimicking the notation used in the informal specification (e.g. $1::'a$) is always a word 1 byte long and is not, say, the natural number one). Otherwise, again, lots of redundant type annotations would arise.

```
parse_ast_translation ‹
  let
    open Ast
    fun mk_numeral t = mk_appl (Constant @{syntax_const _Numeral}) t
    fun mk_word_numeral num t =
      if String.isPrefix 0x num then
        mk_appl (Constant @{syntax_const _constrain})
          [mk_numeral t,
           mk_appl (Constant @{type_syntax word})
             [mk_appl (Constant @{syntax_const _NumeralType})
             [Variable (4 ∗ (size num − 2) |> string_of_int)]]]
      else
        mk_numeral t
```

```
  fun numeral_ast_tr ctxt (t as [Appl [Constant @{syntax_const _constrain},
                                       Constant num,
                                       _]])
                                              = mk_word_numeral num t
    | numeral_ast_tr ctxt (t as [Constant num]) = mk_word_numeral num t
    | numeral_ast_tr _ t                        = mk_numeral t
    | numeral_ast_tr _ t                        = raise AST (@{syntax_const _Numeral}, t)
 in
   [(@{syntax_const _Numeral}, numeral_ast_tr)]
 end
›
```

3.2 Datatypes

Introduce generic notation for mapping of various entities into high-level and low-level representations. A high-level representation of an entity e would be written as $\lceil e \rceil$ and a low-level as $\lfloor e \rfloor$ accordingly. Using a high-level representation it is easier to express and proof some properties and invariants, but some of them can be expressed only using a low-level representation.

We use adhoc overloading to use the same notation for various types of entities (indices, offsets, addresses, capabilities etc.).

```
no_notation floor ("⌊_⌋")

consts rep :: "'a ⇒ 'b" ("⌊_⌋")

no_notation ceiling ("⌈_⌉")

consts abs :: "'a ⇒ 'b" ("⌈_⌉")
```

3.2.1 Deterministic inverse functions

```
definition "maybe_inv f y ≡ if y ∈ range f then Some (the_inv f y) else None"

lemma maybe_inv_inj[intro]: "inj f ⟹ maybe_inv f (f x) = Some x"
  unfolding maybe_inv_def
  by (auto simp add:inj_def the_inv_f_f)

lemma maybe_inv_inj'[dest]: "⟦inj f; maybe_inv f y = Some x⟧ ⟹ f x = y"
  unfolding maybe_inv_def
  by (auto intro:f_the_inv_into_f simp add:inj_def split:if_splits)

locale invertible =
  fixes rep :: "'a ⇒ 'b" ("⌊_⌋")
  assumes inj:"inj rep"
begin
definition inv :: "'b ⇒ 'a option" where "inv ≡ maybe_inv rep"

lemmas inv_inj[folded inv_def, simp] = maybe_inv_inj[OF inj]

lemmas inv_inj'[folded inv_def, simp] = maybe_inv_inj'[OF inj]
end

definition "range2 f ≡ {y. ∃x₁ ∈ UNIV. ∃ x₂ ∈ UNIV. y = f x₁ x₂}"

definition "the_inv2 f ≡ λ x. THE y. ∃ y'. f y y' = x"

definition "maybe_inv2 f y ≡ if y ∈ range2 f then Some (the_inv2 f y) else None"

definition "inj2 f ≡ ∀ x₁ x₂ y₁ y₂. f x₁ y₁ = f x₂ y₂ ⟶ x₁ = x₂"
```

lemma *inj2I*: "$(\bigwedge x_1\ x_2\ y_1\ y_2.\ f\ x_1\ y_1 = f\ x_2\ y_2 \Longrightarrow x_1 = x_2) \Longrightarrow inj2\ f$" **unfolding** *inj2_def*
  by *blast*

lemma *maybe_inv2_inj*[*intro*]: "*inj2 f* ⟹ *maybe_inv2 f* (*f x y*) = *Some x*"
  unfolding *maybe_inv2_def the_inv2_def inj2_def range2_def*
  by (*simp split*:*if_splits*, *blast*)

lemma *maybe_inv2_inj'*[*dest*]:
  "⟦*inj2 f*; *maybe_inv2 f y* = *Some x*⟧ ⟹ ∃ *y'*. *f x y'* = *y*"
  unfolding *maybe_inv2_def the_inv2_def range2_def inj2_def*
  by (*force split*:*if_splits intro*:*theI*)

locale *invertible2* =
  fixes *rep* :: "*'a* ⇒ *'b* ⇒ *'c*" ("⌊_⌋")
  assumes *inj*:"*inj2 rep*"
begin
definition *inv2* :: "*'c* ⇒ *'a option*" **where** "*inv2* ≡ *maybe_inv2 rep*"

lemmas *inv2_inj*[*folded inv2_def*, *simp*] = *maybe_inv2_inj*[*OF inj*]

lemmas *inv2_inj'*[*folded inv_def*, *simp*] = *maybe_inv2_inj'*[*OF inj*]
end

3.2.2 Capability

Introduce capability type. Note that we don't include *Null* capability into it. *Null* is only handled specially inside the call delegation, otherwise it only complicates the proofs with side additional cases. There will be separate type *call* defined as *capability option* to respect the fact that in some places it can indeed be *Null*.

datatype *capability* =
    Call
  | *Reg*
  | *Del*
  | *Entry*
  | *Write*
  | *Log*
  | *Send*

Capability representation would be its assigned number.

In general, in the following we strive to make all encoding functions injective without any preconditions. All the necessary invariants are built into the type definitions.

definition *cap_type_rep* :: "*capability* ⇒ *byte*" **where**
  "*cap_type_rep c* ≡ *case c of*
      Call ⇒ *0x03*
    | *Reg* ⇒ *0x04*
    | *Del* ⇒ *0x05*
    | *Entry* ⇒ *0x06*
    | *Write* ⇒ *0x07*
    | *Log* ⇒ *0x08*
    | *Send* ⇒ *0x09*"

adhoc_overloading *rep cap_type_rep*

Capability representation range from *3* to *9* since *Null* is not included and *2* does not exist.

lemma *cap_type_rep_rng*[*simp*]: "⌊*c*⌋ ∈ {*0x03..0x09*}" **for** *c* :: *capability*
  unfolding *cap_type_rep_def* **by** (*simp split*:*capability.split*)

Capability representation is injective.

lemma *cap_type_rep_inj*[*simp*]: "$\lfloor c_1 \rfloor = \lfloor c_2 \rfloor \Longrightarrow c_1 = c_2$" **for** c_1 c_2 :: *capability*
  unfolding *cap_type_rep_def*
  by (*simp split*:*capability.splits*)

4 bits is sufficient to store a capability number.

lemma *width_cap_type*: "*width* $\lfloor c \rfloor \leq 4$" **for** *c* :: *capability*
proof (*rule ccontr*, *drule not_le_imp_less*)
  assume "$4 <$ *width* $\lfloor c \rfloor$"
  moreover hence "$\lfloor c \rfloor$!! (*width* $\lfloor c \rfloor - 1$)" **using** *nth_width_msb* **by** *force*
  ultimately obtain *n* **where** "$\lfloor c \rfloor$!! *n*" **and** "$n \geq 4$" **by** (*metis le_step_down_nat nat_less_le*)
  thus *False* **unfolding** *cap_type_rep_def* **by** (*simp split*:*capability.splits*)
qed

So, any number greater than or equal to *4* will be enough.

lemma *width_cap_type'*[*simp*]: "$4 \leq n \Longrightarrow$ *width* $\lfloor c \rfloor \leq n$" **for** *c* :: *capability*
  using *width_cap_type*[*of c*] **by** *simp*

Capability representation can't be zero.

lemma *cap_type_nonzero*[*simp*]: "$\lfloor c \rfloor \neq 0$" **for** *c* :: *capability*
  unfolding *cap_type_rep_def* **by** (*simp split*:*capability.splits*)

3.2.3 Capability index

Introduce capability index type that is a natural number in range from 0 to 254.

typedef *capability_index* = "$\{i$ :: *nat*. $i < 2$ ^ *LENGTH*(*byte*) $- 1\}$"
  morphisms *cap_index_rep' cap_index*
  by (*intro exI*[*of* _ "*0*"], *simp*)

adhoc_overloading *rep cap_index_rep'*

adhoc_overloading *abs cap_index*

Capability index representation is a byte. Zero byte is reserved, so capability index representation starts with 1.

definition "*cap_index_rep* $i \equiv$ *of_nat* ($\lfloor i \rfloor + 1$) :: *byte*" **for** *i* :: *capability_index*

adhoc_overloading *rep cap_index_rep*

A single byte is sufficient to store the least number of bits of capability index representation.

lemma *width_cap_index*: "*width* $\lfloor i \rfloor \leq$ *LENGTH*(*byte*)" **for** *i* :: *capability_index* **by** *simp*

lemma *width_cap_index'*[*simp*]: "*LENGTH*(*byte*) $\leq n \Longrightarrow$ *width* $\lfloor i \rfloor \leq n$"
  for *i* :: *capability_index* **by** *simp*

Capability index representation can't be zero byte.

lemma *cap_index_nonzero*[*simp*]: "$\lfloor i \rfloor \neq$ *0x00*" **for** *i* :: *capability_index*
  unfolding *cap_index_rep_def* **using** *cap_index_rep'*[*of i*] *of_nat_neq_0*[*of* "*Suc* $\lfloor i \rfloor$"]
  by *force*

Capability index representation is injective.

lemma *cap_index_inj*[*simp*]: "($\lfloor i_1 \rfloor$ :: *byte*) $= \lfloor i_2 \rfloor \Longrightarrow i_1 = i_2$" **for** i_1 i_2 :: *capability_index*
  unfolding *cap_index_rep_def*
  using *cap_index_rep'*[*of* i_1] *cap_index_rep'*[*of* i_2] *word_of_nat_inj*[*of* "$\lfloor i_1 \rfloor$" "$\lfloor i_2 \rfloor$"]
    cap_index_rep'_inject
  by *force*

lemmas *cap_index_invertible*[*intro*] = *invertible.intro*[*OF injI*, *OF cap_index_inj*]

```
interpretation cap_index_inv: invertible cap_index_rep ..

adhoc_overloading abs cap_index_inv.inv
```

3.2.4 Capability offset

The following datatype specifies data offsets for addresses in the procedure heap.

```
type_synonym capability_offset = byte

datatype data_offset =
  Addr
  | Index
  | Ncaps capability
  | Cap capability capability_index capability_offset
```

Machine word representation of data offsets. Using these offsets the following data can be obtained:

- *Addr*: procedure Ethereum address;
- *Index*: procedure index;
- *Ncaps ty*: the number of capabilities of type *ty*;
- *Cap ty i off*: capability of type *ty*, with index *ty* and offset *off* into that capability.

```
definition data_offset_rep :: "data_offset ⇒ word32" where
 "data_offset_rep off ≡ case off of
    Addr          ⇒ 0x00  ⋈₂ 0x00  ⋈₁  0x00
  | Index         ⇒ 0x00  ⋈₂ 0x00  ⋈₁  0x01
  | Ncaps ty      ⇒ ⌊ty⌋  ⋈₂ 0x00  ⋈₁  0x00
  | Cap ty i off  ⇒ ⌊ty⌋  ⋈₂ ⌊i⌋   ⋈₁  off"

adhoc_overloading rep data_offset_rep
```

Data offset representation is injective.

```
lemma data_offset_inj[simp]:
  "⌊d₁⌋ = ⌊d₂⌋ ⟹ d₁ = d₂" for d₁ d₂ :: data_offset
  unfolding data_offset_rep_def
  by (auto split:data_offset.splits)
```

Least number of bytes to hold the current value of a data offset is *3*.

```
lemma width_data_offset: "width ⌊d⌋ ≤ 3 ∗ LENGTH(byte)" for d :: data_offset
  unfolding data_offset_rep_def
  by (simp split:data_offset.splits)

lemma width_data_offset'[simp]: "3 ∗ LENGTH(byte) ≤ n ⟹ width ⌊d⌋ ≤ n" for d :: data_offset
  using width_data_offset[of d] by simp
```

3.2.5 Kernel storage address

Type definition for procedure indices. A procedure index is represented as a natural number that is smaller then $2^{192} - 1$. It can be zero here, to simplify its future use as an array index, but its low-level representation will start from *1*.

```
typedef key_index = "{i :: nat. i < 2 ^ LENGTH(key) − 1}" morphisms key_index_rep' key_index
  by (rule exI[of _ "0"], simp)

adhoc_overloading rep key_index_rep'

adhoc_overloading abs key_index
```

Introduce address datatype that describes possible addresses in the kernel storage.

```
datatype address =
   Heap_proc key data_offset
  | Nprocs
  | Proc_key key_index
  | Kernel
  | Curr_proc
  | Entry_proc

definition "key_index_rep i ≡ of_nat (⌊i⌋ + 1) :: key" for i :: key_index

adhoc_overloading rep key_index_rep

lemma key_index_nonzero[simp]: "⌊i⌋ ≠ (0 :: key)" for i :: key_index
  unfolding key_index_rep_def using key_index_rep'[of i]
  by (intro of_nat_neq_0, simp_all)

lemma key_index_inj[simp]: "(⌊i₁⌋ :: key) = ⌊i₂⌋ ⟹ i₁ = i₂" for i :: key_index
  unfolding key_index_rep_def using key_index_rep'[of i₁] key_index_rep'[of i₂]
  by (simp add:key_index_rep'_inject of_nat_inj)

abbreviation "kern_prefix ≡ 0xffffffff"
```

Machine word representation of the kernel storage layout, which consists of the following addresses:

- *Heap_proc k offs*: procedure heap of key k and data offset *offs*;
- *Nprocs*: number of procedures;
- *Proc_key i*: a procedure with index i in the procedure list;
- *Kernel*: kernel Ethereum address;
- *Curr_proc*: current procedure;
- *Entry_proc*: entry procedure.

```
definition addr_rep :: "address ⇒ word32" where
  "addr_rep a ≡ case a of
    Heap_proc k offs ⇒ kern_prefix ⋈₁ 0x00 ₅◊ k          ⋈₃ ⌊offs⌋
  | Nprocs           ⇒ kern_prefix ⋈₁ 0x01 ₅◊ (0 :: key) ⋈₃ 0x000000
  | Proc_key i       ⇒ kern_prefix ⋈₁ 0x01 ₅◊ ⌊i⌋        ⋈₃ 0x000000
  | Kernel           ⇒ kern_prefix ⋈₁ 0x02 ₅◊ (0 :: key) ⋈₃ 0x000000
  | Curr_proc        ⇒ kern_prefix ⋈₁ 0x03 ₅◊ (0 :: key) ⋈₃ 0x000000
  | Entry_proc       ⇒ kern_prefix ⋈₁ 0x04 ₅◊ (0 :: key) ⋈₃ 0x000000"

adhoc_overloading rep addr_rep
```

Kernel storage address representation is injective.

```
lemma addr_inj[simp]: "⌊a₁⌋ = ⌊a₂⌋ ⟹ a₁ = a₂" for a₁ a₂ :: address
  unfolding addr_rep_def
  by (split address.splits) (force split:address.splits)+

lemmas addr_invertible[intro] = invertible.intro[OF injI, OF addr_inj]

interpretation addr_inv: invertible addr_rep ..

adhoc_overloading abs addr_inv.inv

abbreviation "prefix_bound ≡ rpad (size kern_prefix) (ucast kern_prefix :: word32)"
```

lemma *prefix_bound*: "$unat\ prefix\_bound < 2\ \hat{}\ LENGTH(word32)$" **unfolding** *rpad_def* **by** *simp*

lemma *prefix_bound'*[*simplified*, *simp*]: "$x \leq unat\ prefix\_bound \Longrightarrow x < 2\ \hat{}\ LENGTH(word32)$"
  using *prefix_bound* **by** *simp*

lemma *addr_prefix*[*simp*, *intro*]: "$limited\_and\ prefix\_bound\ \lfloor a \rfloor$" **for** $a :: address$
  unfolding *limited_and_def addr_rep_def*
  by (*subst word_bw_comms*) (*auto split*:*address.split simp del*:*ucast_bintr*)

3.3 Capability formats

We define capability format generally as a *locale*. It has two parameters: first one is a *subset* function (denoted as $\subseteq_c$), and second one is a *set_of* function, which maps a capability to its high-level representation that is expressed as a set. We have an assumption that *Capability A* is a subset of *Capability B* if and only if their high-level representations are also subsets of each other. We call it the well-definedness assumption (denoted as wd) and using it we can prove abstractly that such generic capability format satisfies the properties of reflexivity and transitivity.

Then sing this locale we can prove that capability formats of all available system calls satisfy the properties of reflexivity and transitivity simply by formalizing corresponding *subset* and *set_of* functions and then proving the well-definedness assumption. This process is called locale interpretation.

no_notation *abs* ("$\lceil \_ \rceil$")

locale *cap_sub* =
  fixes *set_of* :: "$'a \Rightarrow 'b\ set$" ("$\lceil \_ \rceil$")
  fixes *sub* :: "$'a \Rightarrow 'a \Rightarrow bool$" ("$(\_/ \subseteq_c \_)$" [*51*, *51*] *50*)
  assumes *wd*:"$a \subseteq_c b = (\lceil a \rceil \subseteq \lceil b \rceil)$" **begin**

lemma *sub_refl*: "$a \subseteq_c a$" **using** *wd* **by** *auto*

lemma *sub_trans*: "$[\![a \subseteq_c b;\ b \subseteq_c c]\!] \Longrightarrow a \subseteq_c c$" **using** *wd* **by** *blast*
end

notation *abs* ("$\lceil \_ \rceil$")

consts *sub* :: "$'a \Rightarrow 'a \Rightarrow bool$" ("$(\_/ \subseteq_c \_)$" [*51*, *51*] *50*)

3.3.1 Call, Register and Delete capabilities

Call, Register and Delete capabilities have the same format, so we combine them together here. The capability format defines a range of procedure keys that the capability allows one to call. This is defined as a base procedure key and a prefix.

Prefix is defined as a natural number, whose length is bounded by a maximum length of a procedure key.

typedef *prefix_size* = "$\{n :: nat.\ n \leq LENGTH(key)\}$"
  morphisms *prefix_size_rep' prefix_size*
  by *auto*

adhoc_overloading *rep prefix_size_rep'*

Low-level representation of a prefix is a 8-bit machine word (or simply a byte).

definition "$prefix\_size\_rep\ s \equiv of\_nat\ \lfloor s \rfloor :: byte$" **for** $s :: prefix\_size$

adhoc_overloading *rep prefix_size_rep*

Prefix representation is injective.

lemma *prefix_size_inj*[*simp*]: "$(\lfloor s_1 \rfloor :: byte) = \lfloor s_2 \rfloor \Longrightarrow s_1 = s_2$" **for** $s_1\ s_2 :: prefix\_size$

```
  unfolding prefix_size_rep_def using prefix_size_rep'[of s1] prefix_size_rep'[of s2]
  by (simp add:prefix_size_rep'_inject of_nat_inj)

lemma prefix_size_rep_less[simp]: "LENGTH(key) ≤ n ⟹ ⌊s⌋ ≤ (n :: nat)" for s :: prefix_size
  using prefix_size_rep'[of s] by simp
```

Capabilities that have the same format based on prefixes we call "prefixed". Type of prefixed capabilities is defined as a direct product of prefixes and procedure keys.

```
type_synonym prefixed_capability = "prefix_size × key"
```

High-level representation of a prefixed capability is a set of all procedure keys whose first s number of bits (specified by the prefix) are the same as the first s number of bits of the base procedure key k.

```
definition
  "set_of_pref_cap sk ≡ let (s, k) = sk in {k' :: key. take ⌊s⌋ (to_bl k') = take ⌊s⌋ (to_bl k)}"
  for sk :: prefixed_capability

adhoc_overloading abs set_of_pref_cap
```

A prefixed capability A is a subset of a prefixed capability B if:

- the prefix size of A is equal to or greater than the prefix size of B;
- the first s bits (specified by the prefix size of B) of the base procedure of A is equal to the first s bits of the base procedure of B.

```
definition "pref_cap_sub A B ≡
  let (s_A, k_A) = A; (s_B, k_B) = B in
  (⌊s_A⌋ :: nat) ≥ ⌊s_B⌋ ∧ take ⌊s_B⌋ (to_bl k_A) = take ⌊s_B⌋ (to_bl k_B)"
  for A B :: prefixed_capability

adhoc_overloading sub pref_cap_sub

lemma nth_take_i[dest]: "⟦take n a = take n b; i < n⟧ ⟹ a ! i = b ! i"
  by (metis nth_take)

lemma take_less_diff:
  fixes l' l'' :: "'a list"
  assumes ex:"⋀ u :: 'a. ∃ u'. u' ≠ u"
  assumes "n < m"
  assumes "length l' = length l''"
  assumes "n ≤ length l'"
  assumes "m ≤ length l'"
  obtains l where
      "length l = length l'"
  and "take n l = take n l'"
  and "take m l ≠ take m l''"
proof-
  let ?x = "l'' ! n"
  from ex obtain y where neq:"y ≠ ?x" by auto
  let ?l = "take n l' @ y # drop (n + 1) l'"
  from assms have 0:"n = length (take n l') + 0" by simp
  from assms have "take n ?l = take n l'" by simp
  moreover from assms and neq have "take m ?l ≠ take m l''"
    using 0 nth_take_i nth_append_length
    by (metis add.right_neutral)
  moreover have "length ?l = length l'" using assms by auto
  ultimately show ?thesis using that by blast
qed
```

Prove the well-definedness assumption for the prefixed capability format.

```
lemma pref_cap_sub_iff[iff]: "a ⊆c b = (⌈a⌉ ⊆ ⌈b⌉)" for a b :: prefixed_capability
proof
  show "a ⊆c b ⟹ ⌈a⌉ ⊆ ⌈b⌉"
    unfolding pref_cap_sub_def set_of_pref_cap_def
    by (force intro:nth_take_lemma)
  {
    fix n m :: prefix_size
    fix x y :: key
    assume "⌊n⌋ < (⌊m⌋ :: nat)"
    then obtain z where
      "length z = size x"
      "take ⌊n⌋ z = take ⌊n⌋ (to_bl x)" and "take ⌊m⌋ z ≠ take ⌊m⌋ (to_bl y)"
      using take_less_diff[of "⌊n⌋" "⌊m⌋" "to_bl x" "to_bl y"]
      by auto
    moreover hence "to_bl (of_bl z :: key) = z" by (intro word_bl.Abs_inverse[of z], simp)
    ultimately
    have "∃ u :: key.
          take ⌊n⌋ (to_bl u) = take ⌊n⌋ (to_bl x) ∧ take ⌊m⌋ (to_bl u) ≠ take ⌊m⌋ (to_bl y)"
      by metis
  }
  thus "⌈a⌉ ⊆ ⌈b⌉ ⟹ a ⊆c b"
    unfolding pref_cap_sub_def set_of_pref_cap_def subset_eq
    apply (auto split:prod.split)
    by (erule contrapos_pp[of "∀ x. _ x"], simp)
qed

lemmas pref_cap_subsets[intro] = cap_sub.intro[OF pref_cap_sub_iff]
```

Locale interpretation to prove the reflexivity and transitivity properties of a subset function of the prefixed capability format.

```
interpretation pref_cap_sub: cap_sub set_of_pref_cap pref_cap_sub ..
```

Low-level 32-byte machine word representation of the prefixed capability format:

- first byte is the prefix;
- next seven bytes are undefined;
- 24 bytes of the base procedure key.

```
definition "pref_cap_rep sk r ≡
  let (s, k) = sk in ⌊s⌋ 1◇ k OR r ↾ {LENGTH(key)..<LENGTH(word32) − LENGTH(byte)}"
  for sk :: prefixed_capability

adhoc_overloading rep pref_cap_rep
```

Low-level representation is injective.

```
lemma pref_cap_rep_inj_helper_inj[simp]: "⌊s₁⌋ 1◇ k₁ = ⌊s₂⌋ 1◇ k₂ ⟹ s₁ = s₂ ∧ k₁ = k₂"
  for s₁ s₂ :: prefix_size and k₁ k₂ :: key
  by auto

lemma pref_cap_rep_inj_helper_zero[simplified, simp]:
  "n ∈ {LENGTH(key)..<LENGTH(word32) − LENGTH(byte)} ⟹ ¬ (⌊s⌋ 1◇ k) !! n"
  for s :: prefix_size and k :: key
  by simp

lemma pref_cap_rep_inj[simp]: "⌊c₁⌋ r₁ = ⌊c₂⌋ r₂ ⟹ c₁ = c₂" for c₁ c₂ :: prefixed_capability
  unfolding pref_cap_rep_def
  by (auto split:prod.splits)
```

```
lemmas pref_cap_invertible[intro] = invertible2.intro[OF inj2I, OF pref_cap_rep_inj]

interpretation pref_cap_inv: invertible2 pref_cap_rep ..

adhoc_overloading abs pref_cap_inv.inv2
```

3.3.2 Write capability

The write capability format includes 2 values: the first is the base address where we can write to storage. The second is the number of additional addresses we can write to.

Note that write capability must not allow to write to the kernel storage.

```
typedef write_capability = "{(a :: word32, n). n < unat prefix_bound − unat a}"
  morphisms write_cap_rep' write_cap
  unfolding rpad_def
  by (intro exI[of _ "(0, 0)"], simp)

adhoc_overloading rep write_cap_rep'
```

A write capability is correctly bounded by the lowest kernel storage address.

```
lemma write_cap_additional_bound[simplified, simp]:
  "snd ⌊w⌋ < unat prefix_bound" for w :: write_capability
  using write_cap_rep'[of w]
  by (auto split:prod.split)

lemma write_cap_additional_bound'[simplified, simp]:
  "unat prefix_bound ≤ n ⟹ ⌊w⌋ = (a, b) ⟹ b < n"
  using write_cap_additional_bound[of w] by simp

lemma write_cap_bound: "unat (fst ⌊w⌋) + snd ⌊w⌋ < unat prefix_bound"
  using write_cap_rep'[of w]
  by (simp split:prod.splits)

lemma write_cap_bound'[simplified, simp]: "⌊w⌋ = (a, b) ⟹ unat a + b < unat prefix_bound"
  using write_cap_bound[of w] by simp
```

There is no possible overflow in adding the number of additional addresses to the base write address.

```
lemma write_cap_no_overflow: "fst ⌊w⌋ ≤ fst ⌊w⌋ + of_nat (snd ⌊w⌋)" for w :: write_capability
  by (simp add:word_le_nat_alt unat_of_nat_eq less_imp_le)

lemma write_cap_no_overflow'[simp]: "⌊w⌋ = (a, b) ⟹ a ≤ a + of_nat b"
  for w :: write_capability
  using write_cap_no_overflow[of w] by simp

lemma nth_kern_prefix: "kern_prefix !! i = (i < size kern_prefix)"
proof−
  fix i
  {
    fix c :: nat
    assume "i < c"
    then consider "i = c − 1" | "i < c − 1 ∧ c ≥ 1"
      by fastforce
  } note elim = this
  have "i < size kern_prefix ⟹ kern_prefix !! i"
    by (subst test_bit_bl, (erule elim, simp_all)+)
  moreover have "i ≥ size kern_prefix ⟹ ¬ kern_prefix !! i" by simp
  ultimately show "kern_prefix !! i = (i < size kern_prefix)" by auto
qed

lemma nth_prefix_bound[iff]:
```

"prefix_bound !! i = (i ∈ {LENGTH(word32) − size (kern_prefix)..<LENGTH(word32)})"
  (is "_ = (i ∈ {?l..<?r})")
proof−
  have 0:"is_up (ucast :: 32 word ⇒ word32)" by simp
  have 1:"width (ucast kern_prefix :: word32) ≤ size kern_prefix"
    using width_ucast[of kern_prefix, OF 0] by (simp del:width_iff)
  fix i
  show "prefix_bound !! i = (i ∈ {?l..<?r})"
    using rpad_high
      [of "(ucast)$_{(len\ TYPE(word32))}$ kern_prefix" "size (kern_prefix)" i, OF 1, simplified]
      rpad_low
      [of "(ucast)$_{(len\ TYPE(word32))}$ kern_prefix" "size (kern_prefix)" i, OF 1, simplified]
      nth_kern_prefix[of "i − ?l", simplified] nth_ucast[of kern_prefix i, simplified]
      test_bit_size[of prefix_bound i, simplified]
  by (simp (no_asm_simp)) linarith
qed

lemma write_cap_high[dest]:
  "unat a < unat prefix_bound ⟹
  ∃ i ∈ {LENGTH(word32) − size (kern_prefix)..<LENGTH(word32)}. ¬ a !! i"
  (is "_ ⟹ ∃ i ∈ {?l..<?r}. _")
  for a :: word32
proof (rule ccontr, simp del:word_size len_word ucast_bintr)
  {
    fix i
    have "(ucast kern_prefix :: word32) !! i = (i < size kern_prefix)"
      using nth_kern_prefix[of i] nth_ucast[of kern_prefix i] by auto
    moreover assume "i + ?l < ?r ⟹ a !! (i + ?l)"
    ultimately have "(a >> ?l) !! i = (ucast kern_prefix :: word32) !! i"
      using nth_shiftr[of a ?l i] by fastforce
  }
  moreover assume "∀i∈{?l..<?r}. a !! i"
  ultimately have "a >> ?l = ucast kern_prefix" unfolding word_eq_iff using nth_ucast by auto
  moreover have "unat (a >> ?l) = unat a div 2 ^ ?l" using shiftr_div_2n′ by blast
  moreover have "unat (ucast kern_prefix :: word32) = unat kern_prefix"
    by (rule unat_ucast_upcast, simp)
  ultimately have "unat a div 2 ^ ?l = unat kern_prefix" by simp
  hence "unat a ≥ unat kern_prefix ∗ 2 ^ ?l" by simp
  hence "unat a ≥ unat prefix_bound" unfolding rpad_def by simp
  also assume "unat a < unat prefix_bound"
  finally show False ..
qed

High-level representation of a write capability is a set of all addresses to which the capability allows to write.

definition "set_of_write_cap w ≡ let (a, n) = ⌊w⌋ in {a .. a + of_nat n}" **for** w :: write_capability

adhoc_overloading abs set_of_write_cap

A write capability A is a subset of a write capability B if:

- the lowest writable address (which is the base address) of B is less than or equal to the lowest writable address of A;
- the highest writable address (which is base address plus the number of additional keys) of A is less than or equal to the highest writable address of B.

definition "write_cap_sub A B ≡
  let (a_A, n_A) = ⌊A⌋ in let (a_B, n_B) = ⌊B⌋ in a_B ≤ a_A ∧ a_A + of_nat n_A ≤ a_B + of_nat n_B"
  for A B :: write_capability

adhoc_overloading *sub write_cap_sub*

Prove the well-definedness assumption for the write capability format.

lemma *write_cap_sub_iff*[*iff*]: "$a \subseteq_c b = (\lceil a \rceil \subseteq \lceil b \rceil)$" **for** *a b* :: *write_capability*
  unfolding *write_cap_sub_def set_of_write_cap_def*
  by (*auto split*:*prod.splits*)

lemmas *write_cap_subsets*[*intro*] = *cap_sub.intro*[*OF write_cap_sub_iff*]

Locale interpretation to prove the reflexivity and transitivity properties of a subset function of the write capability format.

interpretation *write_cap_sub*: *cap_sub set_of_write_cap write_cap_sub* **..**

Low-level representation of the write capability format is a 32-byte machine word list of two elements:

- the base address;
- the number of additional addresses (also as a machine word).

definition "*write_cap_rep* $w \equiv$ *let* $(a, n) = \lfloor w \rfloor$ *in* $(a,$ *of_nat* n :: *word32*$)$"

adhoc_overloading *rep write_cap_rep*

Low-level representation is injective.

lemma *write_cap_inj*[*simp*]: "$(\lfloor w_1 \rfloor$ :: *word32* $\times$ *word32*$) = \lfloor w_2 \rfloor \Longrightarrow w_1 = w_2$"
  for w_1 w_2 :: *write_capability*
  unfolding *write_cap_rep_def*
  by (*auto*
    split:*prod.splits iff* :*write_cap_rep'_inject*[*symmetric*]
    intro!:*word_of_nat_inj simp add*:*rpad_def*)

lemmas *write_cap_invertible*[*intro*] = *invertible.intro*[*OF injI*, *OF write_cap_inj*]

interpretation *write_cap_inv*: *invertible write_cap_rep* **..**

adhoc_overloading *abs write_cap_inv.inv*

lemma *write_cap_prefix*[*dest*]: "$a \in \lceil w \rceil \Longrightarrow \neg$ *limited_and prefix_bound a*" **for** *w* :: *write_capability*
proof
  assume "$a \in \lceil w \rceil$"
  hence "*unat a* < *unat prefix_bound*"
    unfolding *set_of_write_cap_def*
    apply (*simp split*:*prod.splits*)
    using *write_cap_bound'*[*of w*] *word_less_nat_alt word_of_nat_less* **by** *fastforce*
  then obtain *n* **where** "*n*∈{*LENGTH*(*256 word*) − *size kern_prefix*..<*LENGTH*(*256 word*)}" **and** "¬ *a* !! *n*"
    using *write_cap_high*[*of a*] **by** *auto*
  moreover assume "*limited_and prefix_bound a*"
  ultimately show *False*
    unfolding *limited_and_def word_eq_iff*
    by (*subst* (*asm*) *nth_prefix_bound*, *auto*)
qed

lemma *write_cap_safe*[*simp*]: "$a \in \lceil w \rceil \Longrightarrow a \neq \lfloor a' \rfloor$" **for** *w* :: *write_capability* **and** a' :: *address*
  by *auto*

3.3.3 Log capability

The log capability format includes between 0 and 4 values for log topics and 1 value that specifies the number of enforced topics. We model it as a 32-byte machine word list whose length is between 0 and 4.

```
typedef log_capability = "{ws :: word32 list. length ws ≤ 4}"
  morphisms log_cap_rep' log_capability
  by (intro exI[of _ "[]"], simp)

adhoc_overloading rep log_cap_rep'
```

High-level representation of a log capability is a set of all possible log capabilities whose list prefix in the same and equals to the given log capability.

```
definition "set_of_log_cap l ≡ {xs . prefix ⌊l⌋ xs}" for l :: log_capability

adhoc_overloading abs set_of_log_cap
```

A log capability A is a subset of a log capability B if for each log topic of B the topic is either undefined or equal to that of A. But here we specify that A is a subset of B if B is a list prefix for A. Below we prove that this conditions are equivalent.

```
definition "log_cap_sub A B ≡ prefix ⌊B⌋ ⌊A⌋" for A B :: log_capability

adhoc_overloading sub log_cap_sub
```

Prove the well-definedness assumption for the log capability format.

```
lemma log_cap_sub_iff[iff]: "a ⊆c b = (⌈a⌉ ⊆ ⌈b⌉)" for a b :: log_capability
  unfolding log_cap_sub_def set_of_log_cap_def
  by force

lemmas log_cap_subsets[intro] = cap_sub.intro[OF log_cap_sub_iff]
```

Locale interpretation to prove the reflexivity and transitivity properties of a subset function of the log capability format.

```
interpretation log_cap_sub: cap_sub set_of_log_cap log_cap_sub ..
```

Proof that that the log capability subset is defined according to the specification.

```
lemma "a ⊆c b = (∀ i < length ⌊b⌋ . ⌊a⌋ ! i = ⌊b⌋ ! i ∧ i < length ⌊a⌋)"
  (is "_ = ?R") for a b :: log_capability
  unfolding log_cap_sub_def prefix_def
proof
  let ?L = "∃ zs. ⌊a⌋ = ⌊b⌋ @ zs"
  {
    assume ?L
    moreover hence "length ⌊b⌋ ≤ length ⌊a⌋" by auto
    ultimately show "?L ⟹ ?R"
      by (auto simp add:nth_append)
  next
    assume ?R
    moreover hence len:"length ⌊b⌋ ≤ length ⌊a⌋"
      using le_def by blast
    moreover from ‹?R› have "⌊a⌋ = take (length ⌊b⌋) ⌊a⌋ @ drop (length ⌊b⌋) ⌊a⌋ "
      by simp
    moreover from ‹?R› len have "take (length ⌊b⌋) ⌊a⌋ = ⌊b⌋"
      by (metis nth_take_lemma order_refl take_all)
    ultimately show "?R ⟹ ?L" by (intro exI[of _ "drop (length ⌊b⌋) ⌊a⌋"], arith)
  }
qed
```

Low-level representation of the log capability format is a 32-byte machine word list that includes between 1 and 5 values. First value is the number of enforced topics and the rest are possible values for log topics.

definition "$log\_cap\_rep\ l \equiv (of\_nat\ (length\ \lfloor l \rfloor) :: word32)\ \#\ \lfloor l \rfloor$"

no_adhoc_overloading *rep log_cap_rep′*

adhoc_overloading *rep log_cap_rep*

Low-level representation is injective.

lemma *log_cap_rep_inj*[*simp*]: "$(\lfloor l_1 \rfloor :: word32\ list) = \lfloor l_2 \rfloor \Longrightarrow l_1 = l_2$" **for** $l_1\ l_2 :: log\_capability$
  unfolding *log_cap_rep_def* **using** *log_cap_rep′_inject* **by** *auto*

lemmas *log_cap_rep_invertible*[*intro*] = *invertible.intro*[*OF injI*, *OF log_cap_rep_inj*]

interpretation *log_cap_inv*: *invertible log_cap_rep* **..**

adhoc_overloading *abs log_cap_inv.inv*

lemma *log_cap_rep_length*[*simp*]: "$length\ \lfloor l \rfloor = length\ (log\_cap\_rep'\ l) + 1$"
  unfolding *log_cap_rep_def* **by** *simp*

3.3.4 External call capability

We model the external call capability format using a record with two fields: *allow_addr* and *may_send*, with the following semantic:

- if the field *allow_addr* has value, then only the Ethereum address specified by it can be called, otherwise any address can be called. This models the *CallAny* flag and the *EthAddress* together;
- if the value of the field *may_send* is true, the any quantity of Ether can be sent, otherwise no Ether can be sent. It models the *SendValue* flag.

type_synonym *ethereum_address* = "*160 word*" — 20 bytes

record *external_call_capability* =
  allow_addr :: "*ethereum_address option*"
  may_send :: *bool*

High-level representation of a external call capability is a set of all possible pairs of account addresses and Ether amount that can be sent using this capability.

definition "$set\_of\_ext\_cap\ e \equiv$
  $\{(a, v)\ .\ case\_option\ True\ ((=)\ a)\ (allow\_addr\ e) \wedge (\neg\ may\_send\ e \longrightarrow v = (0 :: word32))\ \}$"

adhoc_overloading *abs set_of_ext_cap*

An external call capability A is a subset of an external call capability B if and only if:

- if A allows to call any Ethereum address, then B also must allow to call any address;
- if A allows to call only specified Ethereum address, then B either must allow to call any address, or it must allow to only call the same address as A;
- if A may send Ether, then B also must be able to send Ether.

abbreviation "$allow\_any\ e \equiv Option.is\_none\ (allow\_addr\ e)$"

abbreviation "$the\_addr\ e \equiv the\ (allow\_addr\ e)$"

definition "$ext\_cap\_sub\ A\ B \equiv$
    $(allow\_any\ A \longrightarrow allow\_any\ B)$
  $\wedge\ ((\neg\ allow\_any\ A \longrightarrow allow\_any\ B) \vee (the\_addr\ A = the\_addr\ B))$

```
  ∧ (may_send A ⟶ may_send B)"
  for A B :: external_call_capability

adhoc_overloading sub ext_cap_sub
```

Prove the well-definedness assumption for the external call capability format.

```
lemma ext_cap_sub_iff[iff]: "a ⊆c b = (⌈a⌉ ⊆ ⌈b⌉)" for a b :: external_call_capability
proof−
  {
    fix v' :: word32
    have "∃ v. v ≠ v'" by (intro exI[of _ "v' − 1"], simp)
  } note [intro] = this
  {
    fix a' :: ethereum_address
    have "∃ a. a ≠ a'" by (intro exI[of _ "a' − 1"], simp)
  } note [intro] = this
  show ?thesis
  unfolding set_of_ext_cap_def ext_cap_sub_def
  by (cases "allow_addr a";
      cases "allow_addr b";
      cases "may_send a";
      cases "may_send b",
      auto iff:subset_iff)
qed

lemmas ext_cap_subsets[intro] = cap_sub.intro[OF ext_cap_sub_iff]
```

Locale interpretation to prove the reflexivity and transitivity properties of a subset function of the external call capability format.

```
interpretation ext_cap_sub: cap_sub set_of_ext_cap ext_cap_sub ..
```

Helper functions to define low-level representation.

```
definition "ext_cap_val e ≡
  (of_bl ([allow_any e, may_send e]
        @ replicate 6 False) :: byte) 1◊ case_option 0 id (allow_addr e)"

definition "ext_cap_frame e ≡
  {if allow_any e then 0 else LENGTH(ethereum_address)..<LENGTH(word32) − LENGTH(byte)}"
```

Low-level 32-byte machine word representation of the external call capability format:

- first bit is the CallAny flag;
- second bit is the SendValue flag;
- 6 undefined bits;
- 11 undefined bytes;
- 20 bytes of the Ethereum address.

```
definition "ext_cap_rep e r ≡  ext_cap_val e OR r ↾ ext_cap_frame e"
  for e :: external_call_capability

adhoc_overloading rep ext_cap_rep
```

Low-level representation is injective.

```
lemma ext_cap_rep_helper_inj[dest]: "ext_cap_val e₁ = ext_cap_val e₂ ⟹ e₁ = e₂"
  for e₁ e₂ :: external_call_capability
  unfolding ext_cap_val_def
```

```
  by (cases "allow_any e₁"; cases "allow_any e₂")
     (auto simp del:of_bl_True of_bl_False dest:word_bl.Abs_eqD split:option.splits)

lemma ext_cap_rep_helper_zero[simp]: "n ∈ ext_cap_frame e ⟹ ¬ ext_cap_val e !! n"
  unfolding ext_cap_frame_def ext_cap_val_def
  by (auto simp del:of_bl_True split:option.split)

lemma ext_cap_rep_inj[simp]: "⌊e₁⌋ r₁ = ⌊e₂⌋ r₂ ⟹ e₁ = e₂" for e₁ e₂ :: external_call_capability
proof (erule rev_mp; cases "allow_any e₁"; cases "allow_any e₂")
  let ?goal = "⌊e₁⌋ r₁ = ⌊e₂⌋ r₂ ⟶ e₁ = e₂"
  {
    {
      fix P e
      have "allow_any e ⟹ (⋀s. P ⦇ allow_addr = None, may_send = s ⦈) ⟹ P e"
        by (cases e, simp add:Option.is_none_def)
    } note[elim!] = this
    note [dest] =
      restrict_inj2[of "λ s (_ :: unit). ext_cap_val ⦇ allow_addr = None, may_send = s ⦈"]
    assume "allow_any e₁" and "allow_any e₂"
    thus ?goal unfolding ext_cap_rep_def by (auto simp add:ext_cap_frame_def)
  next
    {
      fix P e
      have "¬ allow_any e ⟹ (⋀a s. P ⦇ allow_addr = Some a, may_send = s ⦈) ⟹ P e"
        by (cases e, auto simp add:Option.is_none_def)
    } note [elim!] = this
    note [dest] = restrict_inj2[of "λ a s. ext_cap_val ⦇ allow_addr = Some a, may_send = s ⦈"]
    assume "¬ allow_any e₁" and "¬ allow_any e₂"
    thus ?goal unfolding ext_cap_rep_def by (auto simp add:ext_cap_frame_def)
  next
    let ?neq = "allow_any e₁ ≠ allow_any e₂"
    {
      presume ?neq
      moreover hence "msb (ext_cap_val e₁) ≠ msb (ext_cap_val e₂)"
        unfolding ext_cap_val_def msb_nth
        by (auto simp del:of_bl_True of_bl_False simp add:pad_join_high iff:test_bit_of_bl)
      ultimately show ?goal
        unfolding ext_cap_rep_def ext_cap_frame_def word_eq_iff msb_nth word_or_nth nth_restrict
        by simp (meson less_irrefl numeral_less_iff semiring_norm(76) semiring_norm(81))
      thus ?goal .
    next
      assume "allow_any e₁" and "¬ allow_any e₂"
      thus ?neq by simp
    next
      assume "¬ allow_any e₁" and "allow_any e₂"
      thus ?neq by simp
    }
  }
qed

lemmas ext_cap_invertible[intro] = invertible2.intro[OF inj2I, OF ext_cap_rep_inj]

interpretation ext_cap_inv: invertible2 ext_cap_rep ..

adhoc_overloading abs ext_cap_inv.inv2
```

4 Kernel state

4.1 Procedure data

```
typedef 'a capability_list = "{l :: 'a list. length l < 2 ^ LENGTH(byte) − 1}"
  morphisms cap_list_rep cap_list
  by (intro exI[of _ "[]"], simp)

adhoc_overloading rep cap_list_rep

record procedure =
  eth_addr   :: ethereum_address
  call_caps  :: "prefixed_capability capability_list"
  reg_caps   :: "prefixed_capability capability_list"
  del_caps   :: "prefixed_capability capability_list"
  entry_cap  :: bool
  write_caps :: "write_capability capability_list"
  log_caps   :: "log_capability capability_list"
  ext_caps   :: "external_call_capability capability_list"

lemmas alist_simps = size_alist_def alist.Alist_inverse alist.impl_of_inverse

declare alist_simps[simp]

definition "caps_rep (k :: key) p r ty (i :: capability_index) (off :: capability_offset) ≡
  let addr = ⌊Heap_proc k (Cap ty i off)⌋ in
  case ty of
    Call  ⇒ if ⌊i⌋ < length ⌊call_caps p⌋ ∧ off = 0
            then ⌊⌊call_caps p⌋ ! ⌊i⌋⌋ (r addr)
            else r addr

  | Reg   ⇒ if ⌊i⌋ < length ⌊reg_caps p⌋ ∧ off = 0
            then ⌊⌊reg_caps p⌋ ! ⌊i⌋⌋ (r addr)
            else r addr
  | Del   ⇒ if ⌊i⌋ < length ⌊del_caps p⌋ ∧ off = 0
            then ⌊⌊del_caps p⌋ ! ⌊i⌋⌋ (r addr)
            else r addr
  | Entry ⇒ r addr
  | Write ⇒ if ⌊i⌋ < length ⌊write_caps p⌋
            then
              if off = 0x00      then fst (⌊⌊write_caps p⌋ ! ⌊i⌋⌋ :: _ × word32)
              else if off = 0x01 then snd ⌊⌊write_caps p⌋ ! ⌊i⌋⌋
              else                    r addr
            else                      r addr
  | Log   ⇒ if ⌊i⌋ < length ⌊log_caps p⌋
            then
              if unat off < length ⌊⌊log_caps p⌋ ! ⌊i⌋⌋ then ⌊⌊log_caps p⌋ ! ⌊i⌋⌋ ! unat off
              else                                        r addr
            else                                          r addr
  | Send  ⇒ if ⌊i⌋ < length ⌊ext_caps p⌋ ∧ off = 0
            then ⌊⌊ext_caps p⌋ ! ⌊i⌋⌋ (r addr)
            else r addr"

lemma caps_rep_inj[dest]:
  assumes "caps_rep k₁ p₁ r₁ = caps_rep k₂ p₂ r₂"
  shows   "length ⌊call_caps p₁⌋ = length ⌊call_caps p₂⌋   ⟹ call_caps p₁ = call_caps p₂"
    and   "length ⌊reg_caps p₁⌋ = length ⌊reg_caps p₂⌋     ⟹ reg_caps p₁ = reg_caps p₂"
    and   "length ⌊del_caps p₁⌋ = length ⌊del_caps p₂⌋     ⟹ del_caps p₁ = del_caps p₂"
    and   "length ⌊write_caps p₁⌋ = length ⌊write_caps p₂⌋ ⟹ write_caps p₁ = write_caps p₂"
    and   "length ⌊log_caps p₁⌋ = length ⌊log_caps p₂⌋     ⟹ log_caps p₁ = log_caps p₂"
    and   "length ⌊ext_caps p₁⌋ = length ⌊ext_caps p₂⌋     ⟹ ext_caps p₁ = ext_caps p₂"
proof−
  from assms have eq:"⋀ ty i off. caps_rep k₁ p₁ r₁ ty i off = caps_rep k₂ p₂ r₂ ty i off"
    by simp
```

```
note Let_def[simp] if_splits[split] nth_equalityI[intro] cap_list_rep_inject[symmetric, iff]
{
  fix i :: nat
  let ?addr_1 = "⌊Heap_proc k_1 (Cap Call ⌈i⌉ 0)⌋"
  and ?addr_2 = "⌊Heap_proc k_2 (Cap Call ⌈i⌉ 0)⌋"
  assume idx:"i < length ⌊call_caps p_1⌋"
  hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
    using capability_list.cap_list_rep[of "call_caps p_1"] by simp
  assume "length ⌊call_caps p_1⌋ = length ⌊call_caps p_2⌋"
  with idx eq[of Call "⌈i⌉" 0]
  have "⌊⌊call_caps p_1⌋ ! i⌋ (r_1 ?addr_1) = ⌊⌊call_caps p_2⌋ ! i⌋ (r_2 ?addr_2)"
    unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
}
thus "length ⌊call_caps p_1⌋ = length ⌊call_caps p_2⌋ ⟹ call_caps p_1 = call_caps p_2"
  by force

{
  fix i :: nat
  let ?addr_1 = "⌊Heap_proc k_1 (Cap Reg ⌈i⌉ 0)⌋"
  and ?addr_2 = "⌊Heap_proc k_2 (Cap Reg ⌈i⌉ 0)⌋"
  assume idx:"i < length ⌊reg_caps p_1⌋"
  hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
    using capability_list.cap_list_rep[of "reg_caps p_1"] by simp
  assume "length ⌊reg_caps p_1⌋ = length ⌊reg_caps p_2⌋"
  with idx eq[of Reg "⌈i⌉" 0]
  have "⌊⌊reg_caps p_1⌋ ! i⌋ (r_1 ?addr_1) = ⌊⌊reg_caps p_2⌋ ! i⌋ (r_2 ?addr_2)"
    unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
}
thus "length ⌊reg_caps p_1⌋ = length ⌊reg_caps p_2⌋ ⟹ reg_caps p_1 = reg_caps p_2"
  by force

{
  fix i :: nat
  let ?addr_1 = "⌊Heap_proc k_1 (Cap Del ⌈i⌉ 0)⌋"
  and ?addr_2 = "⌊Heap_proc k_2 (Cap Del ⌈i⌉ 0)⌋"
  assume idx:"i < length ⌊del_caps p_1⌋"
  hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
    using capability_list.cap_list_rep[of "del_caps p_1"] by simp
  assume "length ⌊del_caps p_1⌋ = length ⌊del_caps p_2⌋"
  with idx eq[of Del "⌈i⌉" 0]
  have "⌊⌊del_caps p_1⌋ ! i⌋ (r_1 ?addr_1) = ⌊⌊del_caps p_2⌋ ! i⌋ (r_2 ?addr_2)"
    unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
}
thus "length ⌊del_caps p_1⌋ = length ⌊del_caps p_2⌋ ⟹ del_caps p_1 = del_caps p_2"
  by force

{
  fix i :: nat
  let ?addr_1 = "⌊Heap_proc k_1 (Cap Send ⌈i⌉ 0)⌋"
  and ?addr_2 = "⌊Heap_proc k_2 (Cap Send ⌈i⌉ 0)⌋"
  assume idx:"i < length ⌊ext_caps p_1⌋"
  hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
    using capability_list.cap_list_rep[of "ext_caps p_1"] by simp
  assume "length ⌊ext_caps p_1⌋ = length ⌊ext_caps p_2⌋"
  with idx eq[of Send "⌈i⌉" 0]
  have "⌊⌊ext_caps p_1⌋ ! i⌋ (r_1 ?addr_1) = ⌊⌊ext_caps p_2⌋ ! i⌋ (r_2 ?addr_2)"
    unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
}
thus "length ⌊ext_caps p_1⌋ = length ⌊ext_caps p_2⌋ ⟹ ext_caps p_1 = ext_caps p_2"
  by force
```

```
  {
    fix i :: nat
    let ?addr1 = "⌊Heap_proc k1 (Cap Write ⌈i⌉ 0)⌋"
    and ?addr2 = "⌊Heap_proc k2 (Cap Write ⌈i⌉ 0)⌋"
    assume idx:"i < length ⌊write_caps p1⌋"
    hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) - 1}"
      using capability_list.cap_list_rep[of "write_caps p1"] by simp
    assume "length ⌊write_caps p1⌋ = length ⌊write_caps p2⌋"
    with idx eq[of Write "⌈i⌉" "0x00"] eq[of Write "⌈i⌉" "0x01"]
    have "(⌊⌊write_caps p1⌋ ! i⌋ :: word32 × word32) = ⌊⌊write_caps p2⌋ ! i⌋"
      unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0] prod_eqI)
  }
  thus "length ⌊write_caps p1⌋ = length ⌊write_caps p2⌋ ⟹ write_caps p1 = write_caps p2"
    by force

  {
    fix i :: nat
    let ?addr1 = "⌊Heap_proc k1 (Cap Log ⌈i⌉ 0)⌋"
    and ?addr2 = "⌊Heap_proc k2 (Cap Log ⌈i⌉ 0)⌋"
    assume idx:"i < length ⌊log_caps p1⌋"
    hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) - 1}"
      using capability_list.cap_list_rep[of "log_caps p1"] by simp
    {
      fix l
      from log_cap_rep'[of l]
      have "unat (of_nat (length (log_cap_rep' l)) :: word32) = length (log_cap_rep' l)"
        by (simp add:unat_of_nat_eq)
    }
    moreover assume len:"length ⌊log_caps p1⌋ = length ⌊log_caps p2⌋"
    ultimately have rep_len:"length ⌊⌊log_caps p1⌋ ! i⌋ = length ⌊⌊log_caps p2⌋ ! i⌋"
      using idx eq[of Log "⌈i⌉" 0]
      unfolding caps_rep_def log_cap_rep_def
      by (auto simp add:cap_index_inverse[OF 0], metis)
    {
      fix off
      assume off:"off < length ⌊⌊log_caps p1⌋ ! i⌋"
      hence "unat (of_nat off :: byte) = off"
        using log_cap_rep'[of "⌊log_caps p1⌋ ! i"] by (simp add:unat_of_nat_eq)
      with idx off eq[of Log "⌈i⌉" "of_nat off"] len rep_len
      have "⌊⌊log_caps p1⌋ ! i⌋ ! off = ⌊⌊log_caps p2⌋ ! i⌋ ! off"
        unfolding caps_rep_def
        by (auto simp add:cap_index_inverse[OF 0])
    }
    with len rep_len have "⌊⌊log_caps p1⌋ ! i⌋ = ⌊⌊log_caps p2⌋ ! i⌋" by auto
  }
  thus "length ⌊log_caps p1⌋ = length ⌊log_caps p2⌋ ⟹ log_caps p1 = log_caps p2"
    by force
qed

definition "proc_rep k (i :: key_index) (p :: procedure) r (off :: data_offset) ≡
  let addr = ⌊off⌋ in
  let ncaps = λ n. ucast (of_nat n :: byte) OR r addr ↾ {LENGTH(byte)..<LENGTH(word32)} in
  case off of
    Addr        ⇒ ucast (eth_addr p) OR r addr ↾ {LENGTH(ethereum_address) ..<LENGTH(word32)}
  | Index       ⇒ ucast ⌊i⌋ OR r addr ↾ {LENGTH(key) ..<LENGTH(word32)}
  | Ncaps Call  ⇒ ncaps (length ⌊call_caps p⌋)
  | Ncaps Reg   ⇒ ncaps (length ⌊reg_caps p⌋)
  | Ncaps Del   ⇒ ncaps (length ⌊del_caps p⌋)
  | Ncaps Entry ⇒ ncaps (of_bool (entry_cap p))
```

```
  | Ncaps Write  ⇒ ncaps (length ⌊write_caps p⌋)
  | Ncaps Log    ⇒ ncaps (length ⌊log_caps p⌋)
  | Ncaps Send   ⇒ ncaps (length ⌊ext_caps p⌋)
  | Cap ty i off ⇒ caps_rep k p r ty i off"

lemma restrict_ucast_inj[simplified, dest!]:
  "⟦ucast x₁ OR y₁ ↾ {l ..<LENGTH(word32)} = ucast x₂ OR y₂ ↾ {l ..<LENGTH(word32)};
   l = LENGTH('b); LENGTH('b) < LENGTH(word32)⟧ ⟹ x₁ = x₂"
  for x₁ x₂ :: "'b::len word" and y₁ y₂ :: word32
    by (auto dest!:restrict_inj2[of "λ x (_ :: unit). ucast x"] intro:ucast_up_inj)

lemma proc_rep_inj[dest]:
  assumes "proc_rep k₁ i₁ p₁ r₁ = proc_rep k₂ i₂ p₂ r₂"
  shows   "p₁ = p₂" and "i₁ = i₂"
proof (rule procedure.equality)
  from assms have eq:"⋀ off. proc_rep k₁ i₁ p₁ r₁ off = proc_rep k₂ i₂ p₂ r₂ off" by simp

  from eq[of Addr] show "eth_addr p₁ = eth_addr p₂"
    unfolding proc_rep_def by auto
  from eq[of Index] show "i₁ = i₂" unfolding proc_rep_def by auto

  {
    fix l :: "'b capability_list"
    from cap_list_rep[of l]
    have "unat (of_nat (length ⌊l⌋) :: byte) = length ⌊l⌋" by (simp add:unat_of_nat_eq)
  }
  hence [dest]:"⋀ l₁ :: 'b capability_list. ⋀ l₂ :: 'b capability_list.
          (of_nat (length ⌊l₁⌋) :: byte) = of_nat (length ⌊l₂⌋) ⟹ length ⌊l₁⌋ = length ⌊l₂⌋"
    by metis

  from eq[of "Cap _ _ _"] have caps:"caps_rep k₁ p₁ r₁ = caps_rep k₂ p₂ r₂"
    unfolding proc_rep_def by force

  from eq[of "Ncaps Call"] have "length ⌊call_caps p₁⌋ = length ⌊call_caps p₂⌋"
    unfolding proc_rep_def by auto
  with caps show "call_caps p₁ = call_caps p₂" ..

  from eq[of "Ncaps Reg"] have "length ⌊reg_caps p₁⌋ = length ⌊reg_caps p₂⌋"
    unfolding proc_rep_def by auto
  with caps show "reg_caps p₁ = reg_caps p₂" ..

  from eq[of "Ncaps Del"] have "length ⌊del_caps p₁⌋ = length ⌊del_caps p₂⌋"
    unfolding proc_rep_def by auto
  with caps show "del_caps p₁ = del_caps p₂" ..

  from eq[of "Ncaps Write"] have "length ⌊write_caps p₁⌋ = length ⌊write_caps p₂⌋"
    unfolding proc_rep_def by auto
  with caps show "write_caps p₁ = write_caps p₂" ..

  from eq[of "Ncaps Log"] have "length ⌊log_caps p₁⌋ = length ⌊log_caps p₂⌋"
    unfolding proc_rep_def by auto
  with caps show "log_caps p₁ = log_caps p₂" ..

  from eq[of "Ncaps Send"] have "length ⌊ext_caps p₁⌋ = length ⌊ext_caps p₂⌋"
    unfolding proc_rep_def by auto
  with caps show "ext_caps p₁ = ext_caps p₂" ..

  from eq[of "Ncaps Entry"] show "entry_cap p₁ = entry_cap p₂"
    unfolding proc_rep_def by (auto del:iffI) (simp split:if_splits add:of_bool_def)
qed simp
```

4.2 Kernel storage layout

Maximum number of procedures registered in the kernel:

```
abbreviation "max_nprocs ≡ 2 ^ LENGTH(key) − 1 :: nat"

typedef procedure_list = "{l :: (key, procedure) alist. size l ≤ max_nprocs}"
  morphisms proc_list_rep proc_list
  by (intro exI[of _ "Alist []"], simp)

adhoc_overloading rep proc_list_rep

adhoc_overloading rep DAList.impl_of

record kernel =
  curr_proc  :: ethereum_address
  entry_proc :: ethereum_address
  proc_list  :: procedure_list
```

Here we introduce some useful abbreviations that will simplify the expression of the kernel state properties.

Number of the procedures:

```
abbreviation "nprocs σ ≡ size ⌊proc_list σ⌋"
```

Set of procedure indexes:

```
definition "proc_ids σ ≡ {0..<nprocs σ}"

abbreviation "procs σ ≡ DAList.lookup ⌊proc_list σ⌋"

definition "has_key k σ ≡ k ∈ dom (procs σ)"
```

Procedure by its key:

```
definition "proc σ k ≡ the (procs σ k)"

abbreviation "proc_key σ i ≡ fst (⌊⌊proc_list σ⌋⌋ ! i)"
```

Index of procedure:

```
definition "proc_id σ k ≡ ⌈length (takeWhile ((≠) k ∘ fst) ⌊⌊proc_list σ⌋⌋)⌉ :: key_index"

lemma proc_id_alt[simp]:
  "has_key k σ ⟹ ⌊proc_id σ k⌋ ∈ proc_ids σ"
  "has_key k σ ⟹ ⌊⌊proc_list σ⌋⌋ ! ⌊proc_id σ k⌋ = (k, proc σ k)"
proof−
  assume "has_key k σ"
  hence 0:"(k, proc σ k) ∈ set ⌊⌊kernel.proc_list σ⌋⌋"
    unfolding has_key_def proc_def DAList.lookup_def
    by auto
  hence "length (takeWhile ((≠) k ∘ fst) ⌊⌊proc_list σ⌋⌋) ∈ proc_ids σ"
    unfolding has_key_def proc_id_def proc_ids_def
    using length_takeWhile_less[of "⌊⌊proc_list σ⌋⌋ :: (key × procedure) list" "(≠) k ∘ fst"]
    by force
  moreover hence [simp]:"⌊⌈length (takeWhile ((≠) k ∘ fst) ⌊⌊proc_list σ⌋⌋)⌉ :: key_index⌋ =
                        length (takeWhile ((≠) k ∘ fst) ⌊⌊proc_list σ⌋⌋)"
    unfolding proc_ids_def
    using key_index_inverse proc_list_rep[of "proc_list σ"]
    by auto
  ultimately show 1:"⌊proc_id σ k⌋ ∈ proc_ids σ" unfolding proc_ids_def proc_id_def by simp

  from 0 have "∃! i. i < length ⌊⌊proc_list σ⌋⌋ ∧ ⌊⌊proc_list σ⌋⌋ ! i = (k, proc σ k)"
```

```
    using distinct_map by (auto intro!:distinct_Ex1)
  moreover
  {
    fix p i j
    assume 0:"i < length ⌊⌊proc_list σ⌋⌋" and 1:"j < length ⌊⌊proc_list σ⌋⌋"
    moreover assume "⌊⌊proc_list σ⌋⌋ ! i = (k, p)" and "fst (⌊⌊proc_list σ⌋⌋ ! j) = k"
    ultimately have "snd (⌊⌊proc_list σ⌋⌋ ! j) = p"
      using impl_of_distinct nth_mem distinct_map[of fst] unfolding inj_on_def
      by (metis fst_conv snd_conv)
  }
  ultimately have "∀ i < length ⌊⌊proc_list σ⌋⌋.
                   fst (⌊⌊proc_list σ⌋⌋ ! i) = k ⟶ snd (⌊⌊proc_list σ⌋⌋ ! i) = proc σ k"
    by auto
  with 1 show "⌊⌊proc_list σ⌋⌋ ! ⌊proc_id σ k⌋ = (k, proc σ k)"
    unfolding proc_id_def proc_def proc_ids_def DAList.lookup_def
    using nth_length_takeWhile[of "(≠) k ∘ fst" "⌊⌊proc_list σ⌋⌋ :: (key × procedure) list"]
    by (auto intro:prod_eqI)
qed

definition "kernel_rep (σ :: kernel) r a ≡
  case ⌈a⌉ of
    None              ⇒ r a
  | Some addr          ⇒ (case addr of
      Nprocs           ⇒ ucast (of_nat (nprocs σ) :: key) OR r a ↾ {LENGTH(key) ..<LENGTH(word32)}
    | Proc_key i       ⇒ ucast (proc_key σ ⌊i⌋) OR r a ↾ {LENGTH(key) ..<LENGTH(word32)}
    | Kernel           ⇒ 0
    | Curr_proc        ⇒ ucast (curr_proc σ) OR r a ↾ {LENGTH(ethereum_address) ..<LENGTH(word32)}
    | Entry_proc       ⇒ ucast (entry_proc σ) OR r a ↾ {LENGTH(ethereum_address) ..<LENGTH(word32)}
    | Heap_proc k off  ⇒ if has_key k σ
                           then proc_rep k (proc_id σ k) (proc σ k) r off
                           else r a)"

adhoc_overloading rep kernel_rep

lemma proc_list_eqI[intro]:
  assumes "nprocs σ1 = nprocs σ2"
      and "⋀ i. i < nprocs σ1 ⟹ proc_key σ1 i = proc_key σ2 i"
      and "⋀ k. ⟦has_key k σ1; has_key k σ2⟧ ⟹ proc σ1 k = proc σ2 k"
    shows "proc_list σ1 = proc_list σ2"
  unfolding has_key_def DAList.lookup_def proc_def
proof–
  from assms have "∀ i < nprocs σ1.
                  snd (⌊⌊kernel.proc_list σ1⌋⌋ ! i) = snd (⌊⌊kernel.proc_list σ2⌋⌋ ! i)"
    unfolding has_key_def DAList.lookup_def proc_def
    apply (auto iff:fun_eq_iff)
    using
      Some_eq_map_of_iff[of "⌊⌊proc_list σ1⌋⌋"] Some_eq_map_of_iff[of "⌊⌊proc_list σ2⌋⌋"]
      nth_mem[of _ "⌊⌊proc_list σ1⌋⌋"]          nth_mem[of _ "⌊⌊proc_list σ2⌋⌋"]
      impl_of_distinct[of "⌊proc_list σ1⌋"]     impl_of_distinct[of "⌊proc_list σ2⌋"]
    by (metis domIff option.sel option.simps(3) surjective_pairing)
  with assms show ?thesis
    by (auto intro!:nth_equalityI prod_eqI
          iff:proc_list_rep_inject[symmetric] impl_of_inject[symmetric] fun_eq_iff)
qed

lemma kernel_rep_inj[simp]: "⌊σ1⌋ r1 = ⌊σ2⌋ r2 ⟹ σ1 = σ2" for σ1 σ2 :: kernel
proof (rule kernel.equality)
  assume "⌊σ1⌋ r1 = ⌊σ2⌋ r2"
  hence eq:"⋀ a. ⌊σ1⌋ r1 a = ⌊σ2⌋ r2 a" by simp
```

```
  from eq[of "⌊Curr_proc⌋"] show "curr_proc σ1 = curr_proc σ2"
    unfolding kernel_rep_def by auto

  from eq[of "⌊Entry_proc⌋"] show "entry_proc σ1 = entry_proc σ2"
    unfolding kernel_rep_def by auto

  from eq[of "⌊Nprocs⌋"] have "nprocs σ1 = nprocs σ2"
    unfolding kernel_rep_def
    using proc_list_rep[of "proc_list σ1"] proc_list_rep[of "proc_list σ2"]
    by (auto iff:of_nat_inj[symmetric])
  moreover {
    fix i
    assume "i < nprocs σ1"
    with eq[of "⌊Proc_key ⌈i⌉⌋"] have "proc_key σ1 i = proc_key σ2 i"
      unfolding kernel_rep_def
      using proc_list_rep[of "proc_list σ1"]
      by (auto simp add:key_index_inject simp add: key_index_inverse)
  }
  moreover {
    fix k
    assume "has_key k σ1" and " has_key k σ2"
    with eq[of "⌊Heap_proc k _⌋"] have "proc σ1 k = proc σ2 k"
      unfolding kernel_rep_def
      by (auto iff:fun_eq_iff[symmetric])
  }
  ultimately show "proc_list σ1 = proc_list σ2" ..
qed simp

lemmas kernel_invertible[intro] = invertible2.intro[OF inj2I, OF kernel_rep_inj]

interpretation kernel_inv: invertible2 kernel_rep ..

adhoc_overloading abs kernel_inv.inv2

lemma kernel_update_neq[simp]: "¬ limited_and prefix_bound a ⟹ ⌊σ⌋ r a = r a"
proof-
  assume "¬ limited_and prefix_bound a"
  hence "(⌈a⌉ :: address option) = None"
    using addr_prefix by - (rule ccontr, auto dest:addr_inv.inv_inj')
  thus ?thesis unfolding kernel_rep_def by auto
qed
```

5 Call formats

```
primrec split :: "'a::len word list ⇒ 'b::len word list list" where
  "split []       = []" |
  "split (x # xs) = word_rsplit x # split xs"

lemma cat_split[simp]: "map word_rcat (split x) = x"
  unfolding split_def
  by (induct x, simp_all add:word_rcat_rsplit)

lemma split_inj[simp]: "split x = split y ⟹ x = y"
  by (frule arg_cong[where f="map word_rcat"]) (subst (asm) cat_split)+
```

5.1 Deterministic inverse function

```
definition "maybe_inv2_tf z f l ≡
  if ∃ n. takefill z n l ∈ range2 f
  then Some (the_inv2 f (takefill z (SOME n. takefill z n l ∈ range2 f) l))
```

```
  else None"

lemma takefill_implies_prefix:
  assumes "x = takefill u n y"
  obtains (Prefix) "prefix x y" | (Postfix) "prefix y x"
proof (cases "length x ≤ length y")
  case True
  with assms have "prefix x y" unfolding takefill_alt by (simp add: take_is_prefix)
  with that show ?thesis by simp
next
  case False
  with assms have "prefix y x" unfolding takefill_alt by simp
  with that show ?thesis by simp
qed

lemma takefill_prefix_inj:
  "⟦⋀ x y. ⟦P x; P y; prefix x y⟧ ⟹ x = y; P x; P y; x = takefill u n y⟧ ⟹ x = y"
  by (elim takefill_implies_prefix) auto

definition "inj2_tf f ≡ ∀ x₁ y₁ x₂ y₂. prefix (f x₁ y₁) (f x₂ y₂) ⟶ x₁ = x₂"

lemma inj2_tfI: "(⋀ x₁ y₁ x₂ y₂. prefix (f x₁ y₁) (f x₂ y₂) ⟹ x₁ = x₂) ⟹ inj2_tf f"
  unfolding inj2_tf_def
  by blast

lemma exI2[intro]: "P x y ⟹ ∃ x y. P x y" by auto

lemma maybe_inv2_tf_inj[intro]:
  "⟦inj2_tf f; ⋀ x y y'. length (f x y) = length (f x y')⟧ ⟹ maybe_inv2_tf z f (f x y) = Some x"
  unfolding maybe_inv2_tf_def range2_def the_inv2_def inj2_tf_def
  apply (auto split:if_splits)
   apply (subst some1_equality[rotated], erule exI2)
     apply (metis length_takefill takefill_implies_prefix)
  apply (smt length_takefill takefill_implies_prefix the_equality)
  by (meson takefill_same)

lemma maybe_inv2_tf_inj':
  "⟦inj2_tf f; ⋀ x y y'. length (f x y) = length (f x y')⟧ ⟹
    maybe_inv2_tf z f v = Some x ⟹ ∃ y n. f x y = takefill z n v"
  unfolding maybe_inv2_tf_def range2_def the_inv2_def inj2_tf_def
  apply (simp split:if_splits)
  apply (subst (asm) some1_equality[rotated], erule exI2)
   apply (metis length_takefill nat_less_le not_less take_prefix take_takefill)
  by (smt prefix_order.eq_iff the1_equality)

locale invertible2_tf =
  fixes rep :: "'a ⇒ 'b ⇒ 'c::zero list" ("⌊_⌋")
  assumes inj:"inj2_tf rep"
      and len_inv:"⋀ x y y'. length (rep x y) = length (rep x y')"
begin
definition inv2_tf :: "'c list ⇒ 'a option" where "inv2_tf ≡ maybe_inv2_tf 0 rep"

lemmas inv2_tf_inj[folded inv2_tf_def, simp] = maybe_inv2_tf_inj[where z=0, OF inj len_inv]

lemmas inv2_tf_inj'[folded inv2_tf_def, simp] = maybe_inv2_tf_inj'[where z=0, OF inj len_inv]
end
```

5.2 Register system call

```
definition "wf_cap c l ≡
  case (c, l) of
```

```
    (Call,  [c])      ⇒ (⌈c⌉ :: prefixed_capability option) ≠ None
  | (Reg,   [c])      ⇒ (⌈c⌉ :: prefixed_capability option) ≠ None
  | (Del,   [c])      ⇒ (⌈c⌉ :: prefixed_capability option) ≠ None
  | (Entry, [])       ⇒ True
  | (Write, [c1, c2]) ⇒ (⌈(c1, c2)⌉ :: write_capability option) ≠ None
  | (Log,   c)        ⇒ (⌈c⌉ :: log_capability option) ≠ None
  | (Send,  [c])      ⇒ (⌈c⌉ :: external_call_capability option) ≠ None
  | _                 ⇒ False"

definition "overwrite_cap c l r ≡
  case (c, l) of
    (Call,  [c])       ⇒ [⌊the ⌈c⌉ :: prefixed_capability⌋ (r ! 0)]
  | (Reg,   [c])       ⇒ [⌊the ⌈c⌉ :: prefixed_capability⌋ (r ! 0)]
  | (Del,   [c])       ⇒ [⌊the ⌈c⌉ :: prefixed_capability⌋ (r ! 0)]
  | (Entry, [])        ⇒ []
  | (Write, [c1, c2])  ⇒ let (c1, c2) = ⌊the ⌈(c1, c2)⌉ :: write_capability⌋ in [c1, c2]
                         — for mere consistency, no actual need in this, can be just [c1, c2]
  | (Log,   c)         ⇒ ⌊the ⌈c⌉ :: log_capability⌋
  | (Send,  [c])       ⇒ [⌊the ⌈c⌉ :: external_call_capability⌋ (r ! 0)]"

typedef capability_data =
  "{ l :: ((capability × capability_index) × word32 list) list.
     ∀ ((c, _), l) ∈ set l. wf_cap c l }"
  morphisms cap_data_rep cap_data
  by (intro exI[of _ "[]"], simp)

adhoc_overloading rep cap_data_rep

record reg_call =
  proc_key :: key
  eth_addr :: ethereum_address
  cap_data :: capability_data

no_adhoc_overloading rep cap_index_rep

no_adhoc_overloading abs cap_index_inv.inv

definition "cap_index_rep0 i ≡ of_nat ⌊i⌋ :: byte" for i :: capability_index

adhoc_overloading rep cap_index_rep0

lemma width_cap_index0: "width ⌊i⌋ ≤ LENGTH(byte)" for i :: capability_index by simp

lemma width_cap_index0'[simp]: "LENGTH(byte) ≤ n ⟹ width ⌊i⌋ ≤ n"
  for i :: capability_index by simp

lemma cap_index_inj0[simp]: "(⌊i1⌋ :: byte) = ⌊i2⌋ ⟹ i1 = i2" for i1 i2 :: capability_index
  unfolding cap_index_rep0_def
  using cap_index_rep'[of i1] cap_index_rep'[of i2] word_of_nat_inj[of "⌊i1⌋" "⌊i2⌋"]
        cap_index_rep'_inject
  by force

lemmas cap_index0_invertible[intro] = invertible.intro[OF injI, OF cap_index_inj0]

interpretation cap_index_inv0: invertible cap_index_rep0 ..

adhoc_overloading abs cap_index_inv0.inv

definition "reg_call_rep d r ≡
   [ucast (proc_key d) OR (r ! 0) ↾ {LENGTH(key) ..<LENGTH(word32)},
```

```
    ucast (eth_addr d) OR (r ! 1) ↾ {LENGTH(key) ..<LENGTH(word32)}] @
    snd
     (fold
       (λ ((c, i), l) (j, d).
         (j + 3 + length l,
          d @
          [ucast (of_nat (3 + length l) :: byte) OR (r ! j) ↾ {LENGTH(byte) ..<LENGTH(word32)},
           ucast ⌊c⌋ OR (r ! (j + 1)) ↾ {LENGTH(byte) ..<LENGTH(word32)},
           ucast ⌊i⌋ OR (r ! (j + 2)) ↾ {LENGTH(byte) ..<LENGTH(word32)}]
          @ overwrite_cap c l (drop (j + 3) r)))
       ⌊cap_data d⌋
       (2, []))"

datatype result =
    Success storage
  | Revert

abbreviation "SYSCALL_NOEXIST ≡ 0xaa"

abbreviation "SYSCALL_BADCAP ≡ 0x33"

definition "cap_type_opt_rep c ≡ case c of Some c ⇒ ⌊c⌋ | None ⇒ 0x00"
  for c :: "capability option"

adhoc_overloading rep cap_type_opt_rep

lemma cap_type_opt_rep_inj[intro]: "inj cap_type_opt_rep" unfolding cap_type_opt_rep_def inj_def
  by (auto split:option.split)

lemmas cap_type_opt_invertible[intro] = invertible.intro[OF cap_type_opt_rep_inj]

interpretation cap_type_opt_inv: invertible cap_type_opt_rep ..

adhoc_overloading abs cap_type_opt_inv.inv

definition call :: "capability_index ⇒ byte list ⇒ storage ⇒ result × byte list" where
  "call _ _ s ≡ (Success s, [])"

definition register :: "capability_index ⇒ byte list ⇒ storage ⇒ result × byte list" where
  "register _ _ s ≡ (Success s, [])"

definition delete :: "capability_index ⇒ byte list ⇒ storage ⇒ result × byte list" where
  "delete _ _ s ≡ (Success s, [])"

definition set_entry :: "capability_index ⇒ byte list ⇒ storage ⇒ result × byte list" where
  "set_entry _ _ s ≡ (Success s, [])"

definition write_addr :: "capability_index ⇒ byte list ⇒ storage ⇒ result × byte list" where
  "write_addr _ _ s ≡ (Success s, [])"

definition log :: "capability_index ⇒ byte list ⇒ storage ⇒ result × byte list" where
  "log _ _ s ≡ (Success s, [])"

definition external :: "capability_index ⇒ byte list ⇒ storage ⇒ result × byte list" where
  "external _ _ s ≡ (Success s, [])"

definition execute :: "byte list ⇒ storage ⇒ result × byte list" where
  "execute c s ≡ case takefill 0x00 2 c of ct # ci # c ⇒
    (case ⌈ct⌉ of
      None          ⇒ (Revert, [SYSCALL_NOEXIST])
```

$|$ *Some None* $\Rightarrow$ (*Success s*, [])
$|$ *Some* (*Some ct*) $\Rightarrow$ (*case* $\lceil ci \rceil$ *of*
None $\Rightarrow$ (*Revert*, [*SYSCALL_BADCAP*]) — Capability index out of bounds
$|$ *Some ci* $\Rightarrow$ (*case ct of*
Call $\Rightarrow$ *call ci c s*
$|$ *Reg* $\Rightarrow$ *register ci c s*
$|$ *Del* $\Rightarrow$ *delete ci c s*
$|$ *Entry* $\Rightarrow$ *set_entry ci c s*
$|$ *Write* $\Rightarrow$ *write_addr ci c s*
$|$ *Log* $\Rightarrow$ *log ci c s*
$|$ *Send* $\Rightarrow$ *external ci c s*)))"

end