Formal specification of the Cap9 kernel

Mikhail Mandrykin Ilya Shchepetkov

June 28, 2019

Contents

1 Introduction **2**

2 Preliminaries **2**
- 2.1 Type class instantiations . . . 2
- 2.2 Word width . . . 2
- 2.3 Right zero-padding . . . 5
- 2.4 Spanning concatenation . . . 6
- 2.5 Deal with partially undefined results . . . 7
- 2.6 Plain concatenation . . . 8

3 Data formats **10**
- 3.1 Common notation . . . 10
  - 3.1.1 Machine words . . . 10
  - 3.1.2 Concatenation operations . . . 10
- 3.2 Datatypes . . . 11
  - 3.2.1 Deterministic inverse functions . . . 11
  - 3.2.2 Capability . . . 12
  - 3.2.3 Capability index . . . 13
  - 3.2.4 Capability offset . . . 14
  - 3.2.5 Kernel storage address . . . 15
- 3.3 Capability formats . . . 16
  - 3.3.1 Call, Register and Delete capabilities . . . 17
  - 3.3.2 Write capability . . . 19
  - 3.3.3 Log capability . . . 22
  - 3.3.4 External call capability . . . 24

4 Kernel state **26**
- 4.1 Procedure data . . . 26
- 4.2 Kernel storage layout . . . 31

5 Call formats **34**
- 5.1 Deterministic inverse function . . . 34
- 5.2 Register system call . . . 35
- 5.3 Procedure call system call . . . 44
- 5.4 External call system call . . . 44
- 5.5 Log system call . . . 45
- 5.6 Delete and Set entry system calls . . . 46
- 5.7 Write system call . . . 46

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"
  "HOL−Library.Rewrite"
  "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: "∃! 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

lemma restrict_ucast_inv[simp]:
  "⟦a = LENGTH('a); b = LENGTH('b)⟧ ⟹ (ucast x OR y ↾ {a..<b}) AND mask a = ucast x"
  for x :: "'a::len word" and y :: "'b::len word"
  unfolding word_eq_iff
  by (rewrite nth_ucast word_ao_nth nth_mask nth_restrict test_bit_bin)+ auto

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, dest] = 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: "(⋀ x₁ x₂ y₁ y₂. f x₁ y₁ = f x₂ y₂ ⟹ x₁ = x₂) ⟹ 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, dest] = 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
```

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.

Capability representation would be its assigned number.

```
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[dest]: "⌊c1⌋ = ⌊c2⌋ ⟹ c1 = c2" for c1 c2 :: 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 ⌊c⌋ ≤ 4" for c :: capability
proof (rule ccontr, drule not_le_imp_less)
  assume "4 < width ⌊c⌋"
  moreover hence "⌊c⌋ !! (width ⌊c⌋ − 1)" using nth_width_msb by force
  ultimately obtain n where "⌊c⌋ !! n" and "n ≥ 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 ≤ n ⟹ width ⌊c⌋ ≤ n" for c :: capability
  using width_cap_type[of c] by simp
```

Capability representation can't be zero.

```
lemma cap_type_nonzero[simp]: "⌊c⌋ ≠ 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 ≡ of_nat (⌊i⌋ + 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 ⌊i⌋ ≤ LENGTH(byte)" for i :: capability_index by simp
```

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

Capability index representation can't be zero byte.

```
lemma cap_index_nonzero[simp]: "⌊i⌋ ≠ 0x00" for i :: capability_index
  unfolding cap_index_rep_def using cap_index_rep'[of i] of_nat_neq_0[of "Suc ⌊i⌋"]
  by force
```

Capability index representation is injective.

```
lemma cap_index_inj[dest]: "(⌊i₁⌋ :: byte) = ⌊i₂⌋ ⟹ i₁ = i₂" for i₁ i₂ :: capability_index
  unfolding cap_index_rep_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
```

Representation function is invertible.

```
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[dest]:
  "⌊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*]: "$3 * LENGTH(byte) \leq n \Longrightarrow width\ \lfloor d \rfloor \leq n$" **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* = "$\{i :: nat.\ i < 2\ \hat{}\ LENGTH(key) - 1\}$" **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*

Low-level representation of a procedure index is a machine word that starts from 1.

definition "$key\_index\_rep\ i \equiv of\_nat\ (\lfloor i \rfloor + 1) :: key$" **for** i :: *key_index*

adhoc_overloading *rep key_index_rep*

Proof that low-level representation can't be 0.

lemma *key_index_nonzero*[*simp*]: "$\lfloor i \rfloor \neq (0 :: key)$" **for** i :: *key_index*
  unfolding *key_index_rep_def* **using** *key_index_rep'*[*of i*]
  by (*intro of_nat_neq_0*, *simp_all*)

Low-level representation is injective.

lemma *key_index_inj*[*dest*]: "$(\lfloor i_1 \rfloor :: key) = \lfloor i_2 \rfloor \Longrightarrow i_1 = i_2$" **for** i :: *key_index*
  unfolding *key_index_rep_def* **using** *key_index_rep'*[*of* i_1] *key_index_rep'*[*of* i_2]
  by (*simp add:key_index_rep'_inject of_nat_inj*)

Address prefix for all addresses that belong to the kernel storage.

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

Representation function is invertible.

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

interpretation addr_inv: invertible addr_rep ..

adhoc_overloading abs addr_inv.inv
```

Lowest address of the kernel storage (0xffffffff0000...).

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

lemma prefix_bound: "unat prefix_bound < 2 ^ LENGTH(word32)" unfolding rpad_def by simp

lemma prefix_bound'[simplified, simp]: "x ≤ unat prefix_bound ⟹ x < 2 ^ LENGTH(word32)"
  using prefix_bound by simp
```

All addresses in the kernel storage are indeed start with the kernel prefix (0xffffffff).

```
lemma addr_prefix[simp, intro]: "limited_and prefix_bound ⌊a⌋" 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 using 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 ("⌈_⌉")

locale cap_sub =
  fixes set_of :: "'a ⇒ 'b set" ("⌈_⌉")
  fixes sub :: "'a ⇒ 'a ⇒ bool" ("(_/ ⊆c _)" [51, 51] 50)
  assumes wd:"a ⊆c b = (⌈a⌉ ⊆ ⌈b⌉)" begin

lemma sub_refl: "a ⊆c a" using wd by auto

lemma sub_trans: "⟦a ⊆c b; b ⊆c c⟧ ⟹ a ⊆c c" using wd by blast
end
```

```
notation abs ("⌈_⌉")

consts sub :: "'a ⇒ 'a ⇒ bool" ("(_/ ⊆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 ≤ 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 ≡ of_nat ⌊s⌋ :: byte" for s :: prefix_size

adhoc_overloading rep prefix_size_rep
```

Prefix representation is injective.

```
lemma prefix_size_inj[dest]: "(⌊s1⌋ :: byte) = ⌊s2⌋ ⟹ s1 = s2" for s1 s2 :: 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)
```

Any number that is greater or equal to a maximum length of a procedure key is greater or equal to any procedure index.

```
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
```

Auxiliary lemma: if first n elements of lists a and b are equal, and the number i is smaller than n, then the *ith* elements of both lists are also equal.

```
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 "$pref\_cap\_rep\ sk\ r \equiv$
  $let\ (s,\ k) = sk\ in\ \lfloor s \rfloor\ {}_1\Diamond\ k\ OR\ r \restriction \{LENGTH(key)..{<}LENGTH(word32) - LENGTH(byte)\}$"
  for sk :: *prefixed_capability*

adhoc_overloading *rep pref_cap_rep*

Low-level representation is injective.

lemma *pref_cap_rep_inj_helper_inj*[*dest*]: "$\lfloor s_1 \rfloor\ {}_1\Diamond\ k_1 = \lfloor s_2 \rfloor\ {}_1\Diamond\ k_2 \Longrightarrow s_1 = s_2 \wedge k_1 = k_2$"
  for $s_1\ s_2$ :: *prefix_size* **and** $k_1\ k_2$ :: *key*
  by *auto*

lemma *pref_cap_rep_inj_helper_zero*[*simplified, simp*]:
  "$n \in \{LENGTH(key)..{<}LENGTH(word32) - LENGTH(byte)\} \Longrightarrow \neg\ (\lfloor s \rfloor\ {}_1\Diamond\ k)\ !!\ n$"
  for s :: *prefix_size* **and** k :: *key*
  by *simp*

lemma *pref_cap_rep_inj*[*dest*]: "$\lfloor c_1 \rfloor\ r_1 = \lfloor c_2 \rfloor\ r_2 \Longrightarrow c_1 = c_2$" **for** $c_1\ c_2$ :: *prefixed_capability*
  unfolding *pref_cap_rep_def*
  by (*auto split*:*prod.splits*)

Representation function is invertible.

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*]:
  "$snd\ \lfloor w \rfloor < unat\ prefix\_bound$" **for** w :: *write_capability*
  using *write_cap_rep′*[*of w*]
  by (*auto split*:*prod.split*)

lemma *write_cap_additional_bound′*[*simplified, simp*]:
  "$unat\ prefix\_bound \le n \Longrightarrow \lfloor w \rfloor = (a,\ b) \Longrightarrow b < n$"

```
  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
```

Auxiliary lemma: the *ith* element of the kernel address prefix is binary *1* if and only if *i* is smaller then the size of the prefix, otherwise it is *0*.

```
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
```

The *ith* bit of the lowest kernel address is *1* if and only if *i* is smaller or equal to the size of the kernel prefix, otherwise it is *0*.

```
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
```

Addresses from write capabilities can not contain the prefix of the kernel storage.

```
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 ⊆_c b = (⌈a⌉ ⊆ ⌈b⌉)" 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 ≡ let (a, n) = ⌊w⌋ in (a, of_nat n :: word32)"

adhoc_overloading rep write_cap_rep
```

Low-level representation is injective.

```
lemma write_cap_inj[dest]: "(⌊w₁⌋ :: word32 × word32) = ⌊w₂⌋ ⟹ w₁ = w₂"
  for w₁ w₂ :: 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)
```

Representation function is invertible.

```
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
```

An address from the high-level representation of the write capability must be below the lowest kernel storage address.

```
lemma write_cap_prefix[dest]: "a ∈ ⌈w⌉ ⟹ ¬ limited_and prefix_bound a" for w :: write_capability
proof
  assume "a ∈ ⌈w⌉"
  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
```

An address from the high-level representation is different from any address from the kernel storage.

```
lemma write_cap_safe[simp]: "a ∈ ⌈w⌉ ⟹ a ≠ ⌊a'⌋" 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 ≡ (of_nat (length ⌊l⌋) :: word32) # ⌊l⌋"

no_adhoc_overloading rep log_cap_rep'

adhoc_overloading rep log_cap_rep
```

Low-level representation is injective.

```
lemma log_cap_rep_inj[dest]: "(⌊l1⌋ :: word32 list) = ⌊l2⌋ ⟹ l1 = l2" for l1 l2 :: log_capability
  unfolding log_cap_rep_def using log_cap_rep'_inject by auto
```

Representation function is invertible.

```
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*

Length of a low-level representation is correct: it is the length of the topics list plus 1 for storing the number of topics.

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 an 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*

Auxiliary abbreviation: *allow_any e* returns *True* if the field *allow_addr* of the capability *e* does not contain any value, and *False* otherwise.

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

Auxiliary abbreviation: *the_addr e* returns the value of the field *allow_addr* of the capability *e*. It can be used only if *allow_any e* is *False*.

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

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.

definition "$ext\_cap\_sub\ A\ B \equiv$
$(allow\_any\ A \longrightarrow allow\_any\ B)$
$\wedge ((\neg\ allow\_any\ A \longrightarrow allow\_any\ B) \vee (the\_addr\ A = the\_addr\ B))$
$\wedge (may\_send\ A \longrightarrow may\_send\ B)$"
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*]: "$a \subseteq_c b = (\lceil a \rceil \subseteq \lceil b \rceil)$" **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*) $_1\Diamond$ *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* e_1 = *ext_cap_val* e_2 ⟹ $e_1 = e_2$"
  for e_1 e_2 :: *external_call_capability*
  unfolding *ext_cap_val_def*
  by (*cases* "*allow_any* e_1"; *cases* "*allow_any* e_2")
    (*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[dest]: "⌊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
```

Representation function is invertible.

```
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

This section contains definition of the kernel state.

4.1 Procedure data

Introduce *'a capability_list* type that is a list of capabilities of a specific type *'a*, whose length is smaller than 255.

```
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
```

We model a procedure using a record with the following fields:

- *eth_addr* field stores the Ethereum address of the procedure;
- *entry_cap* field is *True* if the procedure is the entry procedure, and *False* otherwise;
- other fields are lists of capabilities of corresponding types assigned to the procedure.

```
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]
```

Low-level representation of the capability as it is stored in the kernel storage: given the procedure, the capability type, index and offset, it checks that all parameters are valid and correct and returns the machine word representation of the capability.

```
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"
```

Capability representation is injective.

```
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 ?addr₁ = "⌊Heap_proc k₁ (Cap Call ⌈i⌉ 0)⌋"
    and ?addr₂ = "⌊Heap_proc k₂ (Cap Call ⌈i⌉ 0)⌋"
    assume idx:"i < length ⌊call_caps p₁⌋"
    hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
      using cap_list_rep[of "call_caps p₁"] by simp
    assume "length ⌊call_caps p₁⌋ = length ⌊call_caps p₂⌋"
    with idx eq[of Call "⌈i⌉" 0]
    have "⌊⌊call_caps p₁⌋ ! i⌋ (r₁ ?addr₁) = ⌊⌊call_caps p₂⌋ ! i⌋ (r₂ ?addr₂)"
      unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
  }
  thus "length ⌊call_caps p₁⌋ = length ⌊call_caps p₂⌋ ⟹ call_caps p₁ = call_caps p₂"
    by force

  {
    fix i :: nat
    let ?addr₁ = "⌊Heap_proc k₁ (Cap Reg ⌈i⌉ 0)⌋"
    and ?addr₂ = "⌊Heap_proc k₂ (Cap Reg ⌈i⌉ 0)⌋"
    assume idx:"i < length ⌊reg_caps p₁⌋"
    hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
      using capability_list.cap_list_rep[of "reg_caps p₁"] by simp
    assume "length ⌊reg_caps p₁⌋ = length ⌊reg_caps p₂⌋"
    with idx eq[of Reg "⌈i⌉" 0]
    have "⌊⌊reg_caps p₁⌋ ! i⌋ (r₁ ?addr₁) = ⌊⌊reg_caps p₂⌋ ! i⌋ (r₂ ?addr₂)"
      unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
  }
  thus "length ⌊reg_caps p₁⌋ = length ⌊reg_caps p₂⌋ ⟹ reg_caps p₁ = reg_caps p₂"
    by force

  {
    fix i :: nat
    let ?addr₁ = "⌊Heap_proc k₁ (Cap Del ⌈i⌉ 0)⌋"
    and ?addr₂ = "⌊Heap_proc k₂ (Cap Del ⌈i⌉ 0)⌋"
    assume idx:"i < length ⌊del_caps p₁⌋"
    hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
      using cap_list_rep[of "del_caps p₁"] by simp
    assume "length ⌊del_caps p₁⌋ = length ⌊del_caps p₂⌋"
    with idx eq[of Del "⌈i⌉" 0]
    have "⌊⌊del_caps p₁⌋ ! i⌋ (r₁ ?addr₁) = ⌊⌊del_caps p₂⌋ ! i⌋ (r₂ ?addr₂)"
      unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
  }
  thus "length ⌊del_caps p₁⌋ = length ⌊del_caps p₂⌋ ⟹ del_caps p₁ = del_caps p₂"
    by force

  {
    fix i :: nat
    let ?addr₁ = "⌊Heap_proc k₁ (Cap Send ⌈i⌉ 0)⌋"
```

```
    and ?addr₂ = "⌊Heap_proc k₂ (Cap Send ⌈i⌉ 0)⌋"
    assume idx:"i < length ⌊ext_caps p₁⌋"
    hence 0:"i ∈ {i. i < 2 ^ LENGTH(8 word) − 1}"
      using capability_list.cap_list_rep[of "ext_caps p₁"] by simp
    assume "length ⌊ext_caps p₁⌋ = length ⌊ext_caps p₂⌋"
    with idx eq[of Send "⌈i⌉" 0]
    have "⌊⌊ext_caps p₁⌋ ! i⌋ (r₁ ?addr₁) = ⌊⌊ext_caps p₂⌋ ! i⌋ (r₂ ?addr₂)"
      unfolding caps_rep_def by (simp add:cap_index_inverse[OF 0])
  }
  thus "length ⌊ext_caps p₁⌋ = length ⌊ext_caps p₂⌋ ⟹ ext_caps p₁ = ext_caps p₂"
    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
```

Low-level representation of the procedure as it is stored in the kernel storage: given the procedure and the data offset it returns the machine word representation of the data that can be found by that offset.

```
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"
```

Low-level representation is injective.

```
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 is $2^{192} - 1$.

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

Introduce *procedure_list* type that is an association list of elements (a list in which each list element comprises a key and a value, and all keys are distinct), where element key is a procedure key and element value is a procedure itself.

```
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
```

We model the kernel storage as a record with three fields:

- *curr_proc* field stores the Ethereum address of the current procedure;
- *entry_proc* field stores the Ethereum address of the entry procedure;
- *proc_list* field stores the list of all registered procedures (with their data).

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

Here we introduce some useful abbreviations and definitions that will simplify the high-level expression of the kernel state properties.

nprocs returns the number of the procedures registered in the kernel. σ is a parameter that refers to the state of the kernel storage.

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

Function that returns set of all current procedure indexes.

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

procs returns map of procedure keys and corresponding procedures. This is an alternative representation of an association list *procedure_list* described above. Note that not all keys contain procedures.

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

Auxiliary function that returns true if and only if a procedure with the key k is registered in the state σ.

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

proc returns the procedure by its key. Can be used only if *has_key k σ = True*.

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

proc_key returns the procedure key by its index in the procedure list.

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

proc_id returns the procedure index in the procedure list by its key.

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

proc_id always returns the procedure index that exists in the current state. Given that index the correct corresponding procedure can be found in the procedure list.

```
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 ⌊⌊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
```

Low-level representation of the kernel storage is a 256 x 256 bits key-value store.

```
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
```

If the number of procedures in two kernel states is the same, procedure keys that can be found by the same index in two corresponding procedure lists are the same, and for each such procedure key its data is also the same in both states, then procedure lists in both states are equal.

```
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 (⌊⌊proc_list σ1⌋⌋ ! i) = snd (⌊⌊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
```

Low-level representation of the kernel storage is injective.

```
lemma kernel_rep_inj[dest]: "⌊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
```

Representation function is invertible.

```
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)
  thus ?thesis unfolding kernel_rep_def by auto
qed
```

5 Call formats

Here we describe formats of all available system calls.

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

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

lemma split_inj[dest]: "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, dest] = maybe_inv2_tf_inj'[where z=0, OF inj len_inv]
end
```

5.2 Register system call

Definition of well-formedness for capability l (represented as a 32-byte machine word list) of type c. l must be correctly formatted to be correctly decoded into the more high-level representation.

```
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"
```

If some capability l of the type c is well-formed, then the length of l (word list) is smaller or equal to 5.

```
lemma length_wf_cap[dest]: "wf_cap c l ⟹ length l ≤ 5"
  unfolding wf_cap_def using log_cap_rep'
  by (auto split:capability.splits list.splits)
```

Capabilities l_1 and l_2 of the type c are the same if their high-level representation are the same.

```
definition "same_cap c l₁ l₂ ≡
  case (c, l₁, l₂) of
    (Call,  [c₁], [c₂])          ⇒ the ⌈c₁⌉ = (the ⌈c₂⌉ :: prefixed_capability)
  | (Reg,   [c₁], [c₂])          ⇒ the ⌈c₁⌉ = (the ⌈c₂⌉ :: prefixed_capability)
  | (Del,   [c₁], [c₂])          ⇒ the ⌈c₁⌉ = (the ⌈c₂⌉ :: prefixed_capability)
  | (Entry, [],   [])            ⇒ True
  | (Write, [c1₁, c2₁], [c1₂, c2₂]) ⇒ the ⌈(c1₁, c2₁)⌉ = (the ⌈(c1₂, c2₂)⌉ :: write_capability)
  | (Log,   c₁,   c₂)            ⇒ the ⌈c₁⌉ = (the ⌈c₂⌉ :: log_capability)
  | (Send,  [c₁], [c₂])          ⇒ the ⌈c₁⌉ = (the ⌈c₂⌉ :: external_call_capability)
  | _                            ⇒ False"
```

Some capability formats have undefined bits or bytes. Here we define function that takes capability l of the type c and writes it over some 32-byte machine word list r in such a way that these undefined parts will contain corresponding parts from r.

```
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)]"
```

If some capability l of the type c is well-wormed, then the result of its writing over a 32-byte machine word list r will also be well-formed.

```
lemma overwrite_cap_wf: "wf_cap c l ⟹ wf_cap c (overwrite_cap c l r)"
  unfolding wf_cap_def overwrite_cap_def
  by (auto split:capability.splits list.splits simp add:write_cap_inv.inv_inj')

abbreviation "zero_fill l ≡ replicate (length l) 0"
```

Writing two equal capabilities over 32-byte machine word list filled with zeroes will produce the same result.

```
lemma same_cap_inj[dest]:
  "same_cap c l₁ l₂ ⟹ overwrite_cap c l₁ (zero_fill l₁) = overwrite_cap c l₂ (zero_fill l₂)"
  unfolding same_cap_def overwrite_cap_def
  by (simp split:capability.splits)
     (auto split:capability.splits list.splits)+
```

If the result of writing capability l_1 over r_1 is equal to the result of writing l_2 over r_2, and both these capabilities are well-formed, then they are the same.

```
lemma overwrite_cap_inj[dest]:
  "⟦overwrite_cap c l₁ r₁ = overwrite_cap c l₂ r₂; wf_cap c l₁; wf_cap c l₂⟧ ⟹ same_cap c l₁ l₂"
  unfolding wf_cap_def overwrite_cap_def same_cap_def
  by (simp split:capability.splits)
    (auto split:capability.splits list.splits simp add:write_cap_inv.inv_inj')
```

Writing well-formed capability over some machine word list some does not change its length.

```
lemma length_overwrite_cap[simp]: "wf_cap c l ⟹ length (overwrite_cap c l r) = length l"
  unfolding wf_cap_def overwrite_cap_def
  by (auto split:capability.splits list.split prod.split)
```

Introduce type the described capability data as sent in the Register Procedure system call. It is represented as a list of elements, each of which contains some capability type, capability index, and well-formed capability itself.

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

adhoc_overloading *rep cap_data_rep'*

adhoc_overloading *abs cap_data*

Data format of the Register Procedure system call is modeled as a record with three fields:

- *proc_key*: procedure key;
- *eth_addr*: procedure Ethereum address;
- *cap_data*: a series of capabilities, and each one is in the format specified above.

record *register_call_data* =
  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*

Redefine low-level representation of capability index. Previously it started with 1, but in the call data format it should start with 0.

definition "*cap_index_rep0* i ≡ *of_nat* $\lfloor i \rfloor$ :: *byte*" **for** i :: *capability_index*

adhoc_overloading *rep cap_index_rep0*

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

lemma *width_cap_index0*: "*width* $\lfloor i \rfloor$ ≤ *LENGTH*(*byte*)" **for** i :: *capability_index* **by** *simp*

lemma *width_cap_index0'*[*simp*]: "*LENGTH*(*byte*) ≤ n ⟹ *width* $\lfloor i \rfloor$ ≤ n"
  for i :: *capability_index* **by** *simp*

Capability index representation is injective.

lemma *cap_index_inj0*[*simp*]: "($\lfloor i_1 \rfloor$ :: *byte*) = $\lfloor i_2 \rfloor$ ⟹ i_1 = i_2" **for** i_1 i_2 :: *capability_index*
  unfolding *cap_index_rep0_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*

Representation function is invertible.

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*

Low-level representation of a single element from the capability data list. It starts with the number of 32-byte machine words associated with the capability, which is 3 + the length of the capability, and stored in a byte aligned right in the 32 bytes. Then there is the type of the capability and the index into the capability list of this type for the current procedure, both of which are also represented as bytes aligned right in the 32 bytes. And finally there is the capability itself as a 32-byte machine word list.

```
abbreviation "cap_data_rep_single r (c :: capability) (i :: capability_index) l j ≡
  [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)"
```

Auxiliary function that will be applied to each element from the capability data list to get its low-level representation.

```
definition "cap_data_rep0 r ≡
  λ ((c, i), l) (j, d). (j + 3 + length l, cap_data_rep_single r c i l j # d)"
```

Length of each element from the capability data list is correctly stored in the element itself in its head (since the element is also a list).

```
lemma length_cap_data_rep0:
  fixes d :: capability_data
  assumes "cap_data_rep0 r ((c, i), l) acc = (j, x # xs)" and "((c, i), l) ∈ set ⌊d⌋"
  shows  "length x = unat (hd x AND mask LENGTH(byte))"
proof–
  from assms(2) have "wf_cap c l" using cap_data_rep'[of d] by auto
  with assms(1) show ?thesis
    unfolding cap_data_rep0_def
    by (force split:prod.splits simp add:unat_ucast_upcast unat_of_nat_eq)
qed

lemma length_cap_data_rep0':
  "⟦[l] = snd (cap_data_rep0 r x acc); x ∈ set ⌊d⌋⟧ ⟹
    length l = unat (hd l AND mask LENGTH(byte))"
  (is "⟦?l; ?in_set⟧ ⟹ _")
  for d :: capability_data
proof–
  assume ?l and ?in_set
  obtain c i l' j
    where "cap_data_rep0 r ((c, i), l') acc = (j, l # [])"
      and "((c, i), l') ∈ set ⌊d⌋"
  proof (cases "cap_data_rep0 r x acc", cases x, cases "fst x")
    fix c i l' j ci ls
    assume "cap_data_rep0 r x acc = (j, ls)" and "x = (ci, l')" and "fst x = (c, i)"
    with that[of c i l' j] ‹?in_set› ‹?l› show ?thesis by simp
  qed
  thus ?thesis using length_cap_data_rep0 by simp
qed
```

Low-level representation of the capability data list is achieved by applying the *cap_data_rep0* function to each element of the list.

```
definition "cap_data_rep (d :: capability_data) r ≡ fold (cap_data_rep0 r) ⌊d⌋"

lemma cap_data_rep'_tail: "⌊d⌋ = x # xs ⟹ xs = ⌊⌈xs⌉⌋" for d :: capability_data
  using cap_data_rep'[of d]
  by (auto intro:cap_data_inverse[symmetric])

lemma length_snd_fold_cap_data_rep0:
  "length (snd (fold (cap_data_rep0 r) xs i)) = length xs + length (snd i)"
  unfolding cap_data_rep0_def by (induction xs arbitrary: i, simp_all split:prod.split)

lemma length_snd_cap_data_rep[simp]:
  "length (snd (cap_data_rep d r i)) = length ⌊d⌋ + length (snd i)"
  unfolding cap_data_rep_def by (simp add:length_snd_fold_cap_data_rep0)
```

First we prove injectivity of "extended" capability data representation, i.e. for capability data represented as a list of separate lists (of 32-byte words), each corresponding to a low-level representation of one capability. The outer list is paired with the total length of the representations. This directly corresponds to the result of *cap_data_rep*. However, to obtain the actual representation, we later take only the list of lists out from this result (no total length), then reverse and concatenate it. So this lemma is not enough to show the overall injectivity of the representation, but in the following we reduce overall injectivity to this intermediate result. We do this by proving that the total length is unambiguously recoverable from the resulting lists and that the resulting list of lists can be recovered from the concatenated list due to the lengths encoded in the initial 32-byte words.

```
lemma cap_data_rep_inj[dest]:
  "⟦cap_data_rep d₁ r₁ i₁ = cap_data_rep d₂ r₂ i₂; length (snd i₁) = length (snd i₂)⟧ ⟹ d₁ = d₂"
  (is "⟦?eq_rep d₁ i₁ d₂ i₂; ?eq_length i₁ i₂⟧ ⟹ _")
proof (induction "⌊d₁⌋" arbitrary:d₁ d₂ i₁ i₂)
  case Nil
  moreover hence "length (snd (cap_data_rep d₁ r₁ i₁)) = length (snd i₁)" by (simp (no_asm))
  ultimately have "⌊d₁⌋ = ⌊d₂⌋" by simp
  thus ?case by (simp add:cap_data_rep'_inject)
next
  {
    fix xs j₁ j₂ l₁ l₂
    have "fold (cap_data_rep0 r₁) xs (j₁, l₁) = fold (cap_data_rep0 r₂) xs (j₂, l₂) ⟹ l₁ = l₂"
      unfolding cap_data_rep0_def
      by (induction xs arbitrary: j₁ j₂ l₁ l₂, auto split:prod.splits)
  } note inj = this
  case (Cons x xs)
  hence "length ⌊d₂⌋ = length ⌊d₁⌋" by (metis add_right_cancel length_snd_cap_data_rep)
  with ‹x # xs = ⌊d₁⌋› obtain y ys where "⌊d₂⌋ = y # ys" by (metis length_Suc_conv)
  from ‹x # xs = ⌊d₁⌋› have d₁:"⌊d₁⌋ = x # xs" ..
  note d₂ = ‹⌊d₂⌋ = y # ys›
  from ‹?eq_rep d₁ i₁ d₂ i₂› obtain i₁' and i₂'
    where "cap_data_rep ⌈xs⌉ r₁ i₁' = cap_data_rep ⌈ys⌉ r₂ i₂'"
      and "length (snd i₁') = length (snd i₁) + 1"
      and "length (snd i₂') = length (snd i₂) + 1"
    unfolding cap_data_rep_def cap_data_rep0_def
    using cap_data_rep'_tail[OF d₂] cap_data_rep'_tail[OF d₁]
    by (auto simp add:d₁ d₂ split:prod.split)
  with ‹?eq_rep d₁ i₁ d₂ i₂› ‹?eq_length i₁ i₂› have tls:"xs = ys"
    using cap_data_rep'_tail[OF d₁] cap_data_rep'_tail[OF d₂]
    by (auto dest:Cons.hyps(1)[OF cap_data_rep'_tail[OF d₁]])
  with ‹?eq_rep d₁ i₁ d₂ i₂› d₁ d₂ have "snd (cap_data_rep0 r₁ x i₁) = snd (cap_data_rep0 r₂ y i₂)"
    unfolding cap_data_rep_def
    by auto (metis inj prod.collapse)
  moreover have "wf_cap (fst (fst x)) (snd x)" and  "wf_cap (fst (fst y)) (snd y)"
    using cap_data_rep'[of d₁] d₁ cap_data_rep'[of d₂]  d₂
    by auto
  ultimately have "x = y" unfolding cap_data_rep0_def
    apply (auto split:prod.splits
        del:cap_type_rep_inj overwrite_cap_inj
        dest!:cap_type_rep_inj overwrite_cap_inj)
    using cap_data_rep'[of d₁] d₁ cap_data_rep'[of d₂] d₂
    by auto
  with tls d₁ d₂ have "⌊d₁⌋ = ⌊d₂⌋" by simp
  thus ?case by (simp add:cap_data_rep'_inject)
qed
```

Helper lemma for induction base proofs. Since *concat* a = [] implies $\forall x \in set\ a.\ x = []$, to obtain a = [] we need this lemma.

```
lemma cap_data_rep_lengths:
  "list_all ((≠) []) l ⟹ list_all ((≠) []) (snd (cap_data_rep d r (i, l)))"
proof (induction "⌊d⌋" arbitrary:d i l)
  case Nil
```

```
  thus ?case unfolding cap_data_rep_def by simp
next
  case (Cons x xs)
  then obtain i' l' where "cap_data_rep0 r x (i, l) = (i', l')" and "list_all ((≠) []) l'"
    unfolding cap_data_rep0_def by (induction x) auto
  with Cons show ?case
    using cap_data_rep'_tail[of d, OF Cons.hyps(2)[symmetric]] Cons.hyps(1)[of "⌈xs⌉" l' i']
    unfolding cap_data_rep_def
    by (rewrite in ‹_ # _ = ⌊d⌋› in asm eq_commute) auto
qed
```

Now proving that the total length is unambiguously recoverable from the length of the resulting lists (and the initial total length in the general case).

```
lemma cap_data_rep_index[simp]:
  assumes "sum_list (map length l) ≤ i"
  shows   "fst (cap_data_rep d r (i, l)) =
        sum_list (map length (snd (cap_data_rep d r (i, l)))) + (i − sum_list (map length l))"
  using assms
proof (induction "⌊d⌋" arbitrary:d i l)
  case Nil
  thus ?case unfolding cap_data_rep_def by auto
next
  case (Cons x xs)
  from Cons(2) have wf:"wf_cap (fst (fst x)) (snd x)"
    using cap_data_rep'[of d] list.set_intros(1)[of x xs]
    by (induction x) auto
  hence 0:"length (overwrite_cap (fst (fst x)) (snd x) (drop (i + 3) r)) = length (snd x)" by simp
  let "?i'" = "fst (cap_data_rep0 r x (i, l))"
    and "?l'" = "snd (cap_data_rep0 r x (i, l))"
  from 0 have "sum_list (map length ?l') = sum_list (map length l) + length (snd x) + 3"
    unfolding cap_data_rep0_def by (auto split:prod.splits)
  hence 1:"?i' = sum_list (map length ?l') + (i − sum_list (map length l))"
    unfolding cap_data_rep0_def using Cons(3) by (simp split:prod.splits)
  from Cons(3) have 2:"sum_list (map length ?l') ≤ ?i'"
    unfolding cap_data_rep0_def using wf by (auto split:prod.splits)
  from Cons(1)[of "⌈xs⌉" ?l' ?i', OF _ 2] cap_data_rep'_tail[OF Cons(2)[symmetric]]
  show ?case unfolding cap_data_rep_def by ((subst Cons(2)[symmetric])+, simp) (insert 1, simp)
qed

lemma cap_data_rep_dest:
  assumes "snd (cap_data_rep d r (i, [])) ≠ []"
  obtains i' where
    "snd (cap_data_rep d r (i, l)) =
     hd (snd (cap_data_rep0 r (last ⌊d⌋) (i', []))) # snd (cap_data_rep ⌈butlast ⌊d⌋⌉ r (i, l))"
  using assms(1)
proof (induction "⌊d⌋" arbitrary:d i l ?thesis)
  case Nil
  thus ?case unfolding cap_data_rep_def by simp
next
  case nonemp:(Cons x xs)
  show ?case proof (cases xs)
    case Nil
    from nonemp(1,3,4) show ?thesis
      unfolding cap_data_rep_def cap_data_rep0_def using cap_data_inverse
      by (simp add:nonemp(2)[symmetric] Nil split:prod.splits)
  next
    case (Cons x' xs')
    let ?l' = "snd (cap_data_rep0 r x (i, l))"
      and ?i' = "fst (cap_data_rep0 r x (i, l))"
    from cap_data_rep'_tail[OF nonemp(2)[symmetric]] have xs:"⌊⌈xs⌉⌋ = xs" ..
```

```
    let ?repx' = "cap_data_rep0 r x' (?i', [])"
    have lenx':"length (snd ?repx') > 0" unfolding cap_data_rep0_def by (simp split:prod.split)
    from cap_data_rep'_tail[of "⌈xs⌉"] xs Cons have xs':"⌊⌈xs'⌉⌋ = xs'" by simp
    from xs' have "⋀ i l. length l ≤ length (snd (cap_data_rep ⌈xs'⌉ r (i, l)))"
    proof (induction xs')
      case Nil
      thus ?case by simp
    next
      case (Cons y ys)
      let ?i' = "fst (cap_data_rep0 r y (i, l))"
        and ?l' = "snd (cap_data_rep0 r y (i, l))"
      note 0 = cap_data_rep'_tail[OF Cons(2), symmetric]
      with Cons(1)[OF 0, of ?l' ?i'] Cons(2)
      show ?case unfolding cap_data_rep_def cap_data_rep0_def by (simp split:prod.splits)
    qed
    from this[of "snd ?repx'" "fst ?repx'"] xs xs' Cons lenx'
    have 0:"snd (cap_data_rep ⌈x' # xs'⌉ r (?i', [])) ≠ []" unfolding cap_data_rep_def by auto
    from nonemp(2) Cons last_ConsR[of xs x] have 1:"last xs = last ⌊d⌋" by simp
    from cap_data_inverse[of "butlast xs"] cap_data_rep'[of "⌈xs⌉"] xs
    have 2:"⌊⌈butlast xs⌉⌋ = butlast xs" by (auto split:prod.splits dest!:in_set_butlastD)
    from cap_data_inverse[of "butlast ⌊d⌋"] cap_data_rep'[of "d"]
    have 3:"⌊⌈butlast ⌊d⌋⌉⌋ = butlast ⌊d⌋" by (auto split:prod.splits dest!:in_set_butlastD)
    from Cons have 4:"butlast ⌊d⌋ = x # butlast xs" by (rewrite nonemp(2)[symmetric], simp)
    from nonemp(1)[of "⌈xs⌉" ?i' ?l', OF xs[symmetric]] 0 Cons obtain i'' where
      "snd (cap_data_rep ⌈xs⌉ r (?i', ?l')) =
         hd (snd (cap_data_rep0 r (last xs) (i'', []))) #
           snd (cap_data_rep ⌈butlast xs⌉ r (?i', ?l'))"
      using xs
      by auto
    with nonemp(3) xs show ?thesis unfolding cap_data_rep_def
      by (rewrite in asm nonemp(2)[symmetric]) (rewrite in asm 3, simp add: 1 2 4)
  qed
qed
```

Now we need to prove that the list of lists resulting from *cap_data_rep* can be recovered from its reversed and concatenated representation. This is quite hard to do directly, so we introduce an intermediate definition *cap_data_rep1*, prove the bijective correspondence between it and *cap_data_rep*, then prove injectivity for concatenation of *cap_data_rep1* and use it to prove that the initial list of lists is recoverable.

```
definition "cap_data_rep1 r ≡
  λ ((c, i), l) (j, d). (j + 3 + length l, d @ [cap_data_rep_single r c i l j])"

lemma cap_data_rep1_fold_pull[simp]:
  "snd (fold (cap_data_rep1 r) d (i, x # xs)) = x # snd (fold (cap_data_rep1 r) d (i, xs))"
proof (induction d arbitrary:xs i)
  case Nil
  thus ?case by simp
next
  case (Cons d ds)
  obtain xs' i' where
    "cap_data_rep1 r d (i, x # xs) = (i', x # xs @ xs')" and
    "cap_data_rep1 r d (i, xs) = (i', xs @ xs')"
    unfolding cap_data_rep1_def by (induction d) auto
  with Cons(1)[of i' "xs @ xs'"] show ?case by simp
qed
```

Proving bijective correspondence between *cap_data_rep* and *cap_data_rep1*.

```
lemma cap_data_rep_rel:
  "rev (snd (cap_data_rep d r (i, l))) = rev l @ snd (fold (cap_data_rep1 r) ⌊d⌋ (i, []))"
proof (induction "⌊d⌋" arbitrary: d i l)
```

```
  case Nil
  thus ?case unfolding cap_data_rep_def by simp
next
  case (Cons x xs)
  from cap_data_rep'_tail[OF Cons(2)[symmetric]] have xs:"⌊⌈xs⌉⌋ = xs" ..
  let ?i' = "fst (cap_data_rep0 r x (i, l))"
    and ?l' = "snd (cap_data_rep0 r x (i, l))"
  obtain i'' x' where 0:"cap_data_rep1 r x (i, []) = (i'', x' # [])"
    unfolding cap_data_rep1_def by (induction x) auto
  hence 1:"rev (snd (cap_data_rep0 r x (i, []))) = [x']"
    unfolding cap_data_rep0_def cap_data_rep1_def by (induction x) auto
  have [simp]: "fst (cap_data_rep0 r x (i, [])) = fst (cap_data_rep1 r x (i, []))"
    unfolding cap_data_rep0_def cap_data_rep1_def by (induction x) auto
  have [simp]:
    "cap_data_rep0 r x (i, l) =
    (fst (cap_data_rep0 r x (i, [])), snd (cap_data_rep0 r x (i, [])) @ l)"
    unfolding cap_data_rep0_def by (simp split:prod.split)
  from Cons(1)[of "⌈xs⌉" ?i' ?l', OF xs[symmetric]] xs
  show ?case unfolding cap_data_rep_def by (simp add: Cons(2)[symmetric] 0 1)
qed
```

Prove that we can recover result of *cap_data_rep1* from its concatenation.

```
lemma concat_cap_data_rep_inj_snd[dest]:
  fixes d₁' d₂' :: capability_data
  assumes "concat (snd (fold (cap_data_rep1 r₁) d₁ (i₁, []))) =
          concat (snd (fold (cap_data_rep1 r₂) d₂ (i₂, [])))"
  assumes "d₁ = ⌊d₁'⌋" and "d₂ = ⌊d₂'⌋"
  shows   "snd (fold (cap_data_rep1 r₁) d₁ (i₁, [])) =
          snd (fold (cap_data_rep1 r₂) d₂ (i₂, []))"
  using assms
proof (induction d₁ arbitrary: d₁' d₂ d₂' i₁ i₂)
  case Nil
  from Nil(3) have 0: "snd (fold (cap_data_rep1 r₂) d₂ (i₂, [])) =
                      rev (snd (cap_data_rep d₂' r₂ (i₂, [])))"
    by (subst rev_is_rev_conv[symmetric], simp add:cap_data_rep_rel)
  from Nil(3) have 1:"d₂ ≠ [] ⟹ set (snd (cap_data_rep d₂' r₂ (i₂, []))) ≠ {}"
    using length_snd_cap_data_rep[of d₂' r₂ "(i₂, [])"] by force
  from Nil[simplified] have "d₂ ≠ [] ⟹ False"
    using cap_data_rep_lengths[of "[]" d₂' r₂ i₂, simplified, unfolded list_all_def]
    by (subst (asm) 0) (subst (asm) set_rev, frule 1, metis equals0I)
  thus ?case by (cases d₂, simp_all)
next
  case (Cons x xs)
  obtain i₁' l₁' where
    0:"cap_data_rep1 r₁ x (i₁, []) = (i₁', l₁' # [])" and
    1:"l₁' ≠ []" and
    2:"[l₁'] = snd (cap_data_rep1 r₁ x (i₁, []))"
    unfolding cap_data_rep1_def by (induction x) auto
  have
    l:"concat (snd (fold (cap_data_rep1 r₁) (x # xs) (i₁, []))) =
      l₁' @ concat (snd (fold (cap_data_rep1 r₁) xs (i₁', [])))"
    by (simp add:0)
  from Cons(2) have "snd (fold (cap_data_rep1 r₂) d₂ (i₂, [])) ≠ []" by (auto simp add:0 1)
  hence "d₂ ≠ []" by auto
  then obtain y ys where 3:"d₂ = y # ys" by (cases d₂, auto)
  obtain i₂' l₂' where
    4:"cap_data_rep1 r₂ y (i₂, []) = (i₂', l₂' # [])" and
    5:"l₂' ≠ []" and
    6:"[l₂'] = snd (cap_data_rep1 r₂ y (i₂, []))"
    unfolding cap_data_rep1_def by (induction y) auto
```

```
  have
    r:"concat (snd (fold (cap_data_rep1 r₂) d₂ (i₂, []))) =
      l₂' @ concat (snd (fold (cap_data_rep1 r₂) ys (i₂', [])))"
    by (simp add: 3 4)

  from 2 have 7:"[l₁'] = snd (cap_data_rep0 r₁ x (i₁, []))"
    unfolding cap_data_rep0_def cap_data_rep1_def by (cases x) auto
  from Cons(3) have 8:"x ∈ set ⌊d₁'⌋" using list.set_intros(1)[of x xs] by simp
  note 9 = length_cap_data_rep0'[OF 7 8]
  from 6 have 10:"[l₂'] = snd (cap_data_rep0 r₂ y (i₂, []))"
    unfolding cap_data_rep0_def cap_data_rep1_def by (cases y) auto
  from Cons(4) 3 have 11:"y ∈ set ⌊d₂'⌋" using list.set_intros(1)[of y ys] by simp
  note 12 = length_cap_data_rep0'[OF 10 11]
  from Cons(2) l r 1 5 9 12 have 13:"l₁' = l₂'" by (metis append_eq_append_conv hd_append2)
  with Cons(2) l r
  have 14:"concat (snd (fold (cap_data_rep1 r₁) xs (i₁', []))) =
          concat (snd (fold (cap_data_rep1 r₂) ys (i₂', [])))"
    by simp

  note xs = cap_data_rep'_tail[OF Cons(3)[symmetric]]
  from cap_data_rep'_tail[of d₂'] Cons(4) 3 have ys:"ys = ⌊⌈ys⌉⌋" by blast
  note 15 = Cons(1)[OF 14 xs ys]

  from 0 3 4 13 15 show ?case by simp
qed
```

Final injectivity proof for capability data representation:

```
lemma concat_cap_data_rep_inj[simplified, dest]:
  "(concat ∘ rev ∘ snd) (cap_data_rep d₁ r₁ (i, [])) =
   (concat ∘ rev ∘ snd) (cap_data_rep d₂ r₂ (i, [])) ⟹
   cap_data_rep d₁ r₁ (i, []) = cap_data_rep d₂ r₂ (i, [])"
  (is "?prem ⟹ _")
proof
  assume ?prem
  hence
    "concat (snd (fold (cap_data_rep1 r₁) ⌊d₁⌋ (i, []))) =
     concat (snd (fold (cap_data_rep1 r₂) ⌊d₂⌋ (i, [])))"
    by (simp add:cap_data_rep_rel)
  hence "snd (fold (cap_data_rep1 r₁) ⌊d₁⌋ (i, [])) = snd (fold (cap_data_rep1 r₂) ⌊d₂⌋ (i, []))"
    by auto
  thus "snd (cap_data_rep d₁ r₁ (i, [])) = snd (cap_data_rep d₂ r₂ (i, []))"
    by (simp add:cap_data_rep_rel[where l="[]", simplified, symmetric])
  thus "fst (cap_data_rep d₁ r₁ (i, [])) = fst (cap_data_rep d₂ r₂ (i, []))"
    by simp
qed

definition "reg_call_rep (d :: register_call_data) r ≡
    [ucast (proc_key d) OR (r ! 0) ↾ {LENGTH(key) ..<LENGTH(word32)},
     ucast (eth_addr d) OR (r ! 1) ↾ {LENGTH(ethereum_address) ..<LENGTH(word32)}] @
     ((concat ∘ rev ∘ snd) (cap_data_rep (cap_data d) r (2, [])))"

adhoc_overloading rep reg_call_rep

lemma reg_call_rep_inj[dest]: "⌊d₁⌋ r₁ = ⌊d₂⌋ r₂ ⟹ d₁ = d₂" for d₁ d₂ :: register_call_data
proof (rule register_call_data.equality)
  assume eq:"⌊d₁⌋ r₁ = ⌊d₂⌋ r₂"

  from eq show "proc_key d₁ = proc_key d₂" unfolding reg_call_rep_def by auto
  from eq show "eth_addr d₁ = eth_addr d₂" unfolding reg_call_rep_def by auto
```

```
  from eq show "cap_data d1 = cap_data d2" unfolding reg_call_rep_def by auto
qed simp

lemmas reg_call_invertible[intro] = invertible2.intro[OF inj2I, OF reg_call_rep_inj]

interpretation reg_call_inv: invertible2 reg_call_rep ..

adhoc_overloading abs reg_call_inv.inv2
```

5.3 Procedure call system call

```
type_synonym procedure_call_data = "(key × byte list)"

definition "proc_call_rep (cd :: procedure_call_data) (r :: byte list) ≡
  let (k, d) = cd;
      r' = word_rcat (take (LENGTH(word32) div LENGTH(byte)) r) :: word32 in
  word_rsplit (ucast k OR r' ↾ {LENGTH(key) ..<LENGTH(word32)}) @ d"

adhoc_overloading rep proc_call_rep

lemma word_rsplit_inj[dest]: "word_rsplit a = word_rsplit b ⟹ a = b" for a::"'a::len word"
  by (auto dest:arg_cong[where f="word_rcat :: _ ⇒ 'a word"] simp add:word_rcat_rsplit)

lemma proc_call_rep_inj[dest]: "⌊d1⌋ r1 = ⌊d2⌋ r2 ⟹ d1 = d2" for d1 d2 :: procedure_call_data
proof–
  let "?key_rep k r" =
    "word_rsplit (ucast (k :: key) OR (r :: word32) ↾ {LENGTH(key) ..<LENGTH(word32)})
     :: byte list"
  assume "⌊d1⌋ r1 = ⌊d2⌋ r2"
  moreover then obtain k1 d1' and r1' :: word32 and k2 d2' and r2' :: word32 where
    "⌊d1⌋ r1 = ?key_rep k1 r1' @ d1'" "⌊d2⌋ r2 = ?key_rep k2 r2' @ d2'" and
    d1:"(k1, d1') = d1" and d2:"(k2, d2') = d2"
    unfolding proc_call_rep_def
    by (simp add: Let_def split:prod.splits, metis)
  moreover have "length (?key_rep k1 r1') = length (?key_rep k2 r2')"
    by (rule word_rsplit_len_indep)
  ultimately have "?key_rep k1 r1' = ?key_rep k2 r2'" and "d1' = d2'" by auto
  with d1 and d2 show ?thesis by auto
qed

lemmas proc_call_invertible[intro] = invertible2.intro[OF inj2I, OF proc_call_rep_inj]

interpretation proc_call_inv: invertible2 proc_call_rep ..

adhoc_overloading abs proc_call_inv.inv2
```

5.4 External call system call

```
record external_call_data =
  addr   :: ethereum_address
  amount :: word32
  data   :: "byte list"

definition "ext_call_rep (d :: external_call_data) (r :: byte list) ≡
  let r' = word_rcat (take (LENGTH(word32) div LENGTH(byte)) r) :: word32 in
  concat (split
    [ucast (addr d) OR r' ↾ {LENGTH(ethereum_address) ..<LENGTH(word32)},
     amount d])
  @ data d"

adhoc_overloading rep ext_call_rep
```

```
lemma ext_call_rep_inj[dest]: "⌊d₁⌋ r₁ = ⌊d₂⌋ r₂ ⟹ d₁ = d₂" for d₁ d₂ :: external_call_data
proof (rule external_call_data.equality)
  {
    fix a₁ b₁ a₂ b₂ :: word32 and d₁ d₂ :: "byte list"
    assume "concat (split [a₁, b₁]) @ d₁ = concat (split [a₂, b₂]) @ d₂"
    hence "a₁ = a₂" and "b₁ = b₂" by (auto simp add:word_rsplit_len_indep)
  } note dest[dest] = this
  assume eq:"⌊d₁⌋ r₁ = ⌊d₂⌋ r₂"

  from eq show "addr d₁ = addr d₂" unfolding ext_call_rep_def
    by (auto simp del:concat.simps split.simps)
  from eq show "amount d₁ = amount d₂" unfolding ext_call_rep_def by (auto simp only:Let_def)
  from eq show "data d₁ = data d₂" unfolding ext_call_rep_def
    by (auto simp add:word_rsplit_len_indep)
qed simp

lemmas external_call_invertible[intro] = invertible2.intro[OF inj2I, OF ext_call_rep_inj]

interpretation ext_call_inv: invertible2 ext_call_rep ..

adhoc_overloading abs ext_call_inv.inv2
```

5.5 Log system call

```
type_synonym log_topics = log_capability

type_synonym log_call_data = "log_topics × byte list"

definition "log_call_rep td r ≡
  let (t, d) = td;
      n = length ⌊t⌋;
      c = LENGTH(word32) div LENGTH(byte);
      r' = word_rcat (take c (drop (c * (n + 1)) r)) :: word32 in
  concat (split (⌊t⌋ @ [r'])) @ d"
  for td :: log_call_data

adhoc_overloading rep log_call_rep

lemma split_distrib[simp]:"split (a @ b) = split a @ split b" by (induct a, simp_all)

lemma split_length_indep[dest]: "length a = length b ⟹ length (split a) = length (split b)"
proof (induct a arbitrary:b, simp)
  case (Cons x xs)
  from Cons(1)[of "tl b"] Cons(2) show ?case by (cases b, simp_all)
qed

lemma split_concat_length_indep[dest]:
  "length a = length b ⟹
   length (concat (split a :: 'b::len word list list)) =
   length (concat (split b :: 'b::len word list list))"
  for a b :: "'a::len word list"
proof (induct a arbitrary:b, simp)
  case (Cons x xs)
  from Cons(1)[of "tl b"] Cons(2) show ?case by (cases b, simp_all add:word_rsplit_len_indep)
qed

lemma split_lengths:
  "i ∈ set (split (a :: 'a::len word list) :: 'b::len word list list)
   ⟹ length i = (LENGTH('a) + LENGTH('b) − 1) div LENGTH('b)"
  by (induct a, auto simp add:length_word_rsplit_exp_size')
```

```
lemma log_call_rep_inj[dest]: "⌊d₁⌋ r₁ = ⌊d₂⌋ r₂ ⟹ d₁ = d₂" for d₁ d₂ :: log_call_data
proof
  {
    fix a b :: "word32 list" and d₁ d₂
    assume "(concat (split a) :: byte list) @ d₁ = concat (split b) @ d₂"
      and "length a = length b"
    hence "a = b"
      by (intro split_inj, intro concat_injective, auto)
        (subst (asm) append_eq_append_conv, auto elim:in_set_zipE simp add:split_lengths)
  } note [dest] = this

  assume eq:"⌊d₁⌋ r₁ = ⌊d₂⌋ r₂"
  moreover hence "length ⌊fst d₁⌋ = length ⌊fst d₂⌋" unfolding log_call_rep_def log_cap_rep_def
    using log_cap_rep'[of "fst d₁"] log_cap_rep'[of "fst d₂"]
    by (auto split:prod.splits simp add:word_rsplit_len_indep of_nat_inj)
  ultimately show "fst d₁ = fst d₂" unfolding log_call_rep_def by (auto split:prod.splits)

  with eq show "snd d₁ = snd d₂" unfolding log_call_rep_def
    by (auto split:prod.splits simp add:word_rsplit_len_indep)
qed

lemmas log_call_invertible[intro] = invertible2.intro[OF inj2I, OF log_call_rep_inj]

interpretation log_call_inv: invertible2 log_call_rep ..

adhoc_overloading abs log_call_inv.inv2
```

5.6 Delete and Set entry system calls

```
type_synonym delete_call_data = key

type_synonym set_entry_call_data = key

definition "proc_key_call_rep k r = [ucast k OR r ↾ {LENGTH(key) ..<LENGTH(word32)}]"
  for k :: key and r :: word32

adhoc_overloading rep proc_key_call_rep

lemma proc_key_call_rep_inj0[dest]: "⌊d₁⌋ r₁ = ⌊d₂⌋ r₂ ⟹ d₁ = d₂" for d₁ d₂ :: key
  unfolding proc_key_call_rep_def by auto

lemma proc_key_call_rep_length[simp]: "length (⌊d⌋ r) = 1" for d :: key
  unfolding proc_key_call_rep_def by simp

lemma proc_key_call_rep_inj[dest]: "prefix (⌊d₁⌋ r₁) (⌊d₂⌋ r₂) ⟹ d₁ = d₂" for d₁ d₂ :: key
  unfolding prefix_def using proc_key_call_rep_length
  by (subst (asm) append_Nil2[symmetric]) (subst (asm) append_eq_append_conv, auto)

lemma proc_key_call_rep_indep: "length (⌊d₁⌋ r₁) = length (⌊d₂⌋ r₂)" for d₁ d₂ :: key by simp

lemmas proc_key_call_invertible[intro] =
  invertible2_tf.intro[OF inj2_tfI, OF proc_key_call_rep_inj proc_key_call_rep_indep]

interpretation proc_key_call_inv: invertible2_tf proc_key_call_rep ..

adhoc_overloading abs proc_key_call_inv.inv2_tf
```

5.7 Write system call

```
type_synonym write_call_data = "word32 × word32"
```

```
definition "write_call_rep w _ ≡ let (a, v) = w in [a, v]" for w :: write_call_data

adhoc_overloading rep write_call_rep

lemma write_call_rep_inj[dest]: "prefix (⌊d1⌋ r1) (⌊d2⌋ r2) ⟹ d1 = d2" for d1 d2 :: write_call_data
  unfolding write_call_rep_def by (simp split:prod.splits)

lemma write_call_rep_indep: "length (⌊d1⌋ r1) = length (⌊d2⌋ r2)" for d1 d2 :: write_call_data
  unfolding write_call_rep_def by (simp split:prod.split)

lemmas write_call_invertible[intro] =
  invertible2_tf.intro[OF inj2_tfI, OF write_call_rep_inj write_call_rep_indep]

interpretation write_call_inv: invertible2_tf write_call_rep ..

adhoc_overloading abs write_call_inv.inv2_tf

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

end
```