Some classical results in inductive inference of recursive functions

Frank J. Balbach

September 1, 2025

Abstract

This entry formalizes some classical concepts and results from inductive inference of recursive functions. In the basic setting a partial recursive function ("strategy") must identify ("learn") all functions from a set ("class") of recursive functions. To that end the strategy receives more and more values $f(0), f(1), f(2), \ldots$ of some function f from the given class and in turn outputs descriptions of partial recursive functions, for example, Gödel numbers. The strategy is considered successful if the sequence of outputs ("hypotheses") converges to a description of f. A class of functions learnable in this sense is called "learnable in the limit". The set of all these classes is denoted by LIM.

Other types of inference considered are finite learning (FIN), behaviorally correct learning in the limit (BC), and some variants of LIM with restrictions on the hypotheses: total learning (TOTAL), consistent learning (CONS), and class-preserving learning (CP). The main results formalized are the proper inclusions FIN $\subset$ CP $\subset$ TOTAL $\subset$ CONS $\subset$ LIM $\subset$ BC $\subset 2^{\mathcal{R}}$, where $\mathcal{R}$ is the set of all total recursive functions. Further results show that for all these inference types except CONS, strategies can be assumed to be total recursive functions; that all inference types but CP are closed under the subset relation between classes; and that no inference type is closed under the union of classes.

The above is based on a formalization of recursive functions heavily inspired by the *Universal Turing Machine* entry by Xu et al. [18], but different in that it models partial functions with codomain *nat option*. The formalization contains a construction of a universal partial recursive function, without resorting to Turing machines, introduces decidability and recursive enumerability, and proves some standard results: existence of a Kleene normal form, the *s-m-n* theorem, Rice's theorem, and assorted fixed-point theorems (recursion theorems) by Kleene, Rogers, and Smullyan.

Contents

1 Partial recursive functions **3**

1.1 Basic definitions . . . 3

1.1.1 Partial recursive functions . . . 3

1.1.2 Extensional equality . . . 8

1.1.3 Primitive recursive and total functions . . . 10

1.2 Simple functions . . . 11

1.2.1 Manipulating parameters . . . 12

1.2.2 Arithmetic and logic . . . 13

1.2.3 Comparison and conditions . . . 15

1.3 The halting problem . . . 17

1.4 Encoding tuples and lists . . . 17

1.4.1 Pairs and tuples . . . 18

1.4.2 Lists . . . 25

1.5 A universal partial recursive function . . . 36

1.5.1 A step function . . . 36

1.5.2 Encoding partial recursive functions . . . 50

1.5.3 The step function on encoded configurations . . . 53

1.5.4 The step function as a partial recursive function . . . 60

1.5.5 The universal function . . . 64

1.6 Applications of the universal function . . . 69

1.6.1 Lazy conditional evaluation . . . 70

1.6.2 Enumerating the domains of partial recursive functions . . . 70

1.6.3 Concurrent evaluation of functions . . . 76

1.7 Kleene normal form and the number of μ-operations . . . 81

1.8 The s-m-n theorem . . . 84

1.9 Fixed-point theorems . . . 91

1.9.1 Rogers's fixed-point theorem . . . 91

1.9.2 Kleene's fixed-point theorem . . . 93

1.9.3 Smullyan's double fixed-point theorem . . . 95

1.10 Decidable and recursively enumerable sets . . . 95

1.11 Rice's theorem . . . 100

1.12 Partial recursive functions as actual functions . . . 101

1.12.1 The definitions . . . 101

1.12.2 Some simple properties . . . 104

1.12.3 The Gödel numbering φ . . . 106

1.12.4 Fixed-point theorems . . . 108

2 Inductive inference of recursive functions **109**
2.1 Preliminaries . . . 110
2.1.1 The prefixes of a function . . . 110
2.1.2 NUM . . . 114
2.2 Types of inference . . . 120
2.2.1 LIM: Learning in the limit . . . 120
2.2.2 BC: Behaviorally correct learning in the limit . . . 122
2.2.3 CONS: Learning in the limit with consistent hypotheses . . . 124
2.2.4 TOTAL: Learning in the limit with total hypotheses . . . 126
2.2.5 CP: Learning in the limit with class-preserving hypotheses . . . 127
2.2.6 FIN: Finite learning . . . 128
2.3 FIN is a proper subset of CP . . . 129
2.4 NUM and FIN are incomparable . . . 137
2.5 NUM and CP are incomparable . . . 140
2.6 NUM is a proper subset of TOTAL . . . 142
2.7 CONS is a proper subset of LIM . . . 147
2.8 Lemma R . . . 157
2.8.1 Strong Lemma R for LIM, FIN, and BC . . . 157
2.8.2 Weaker Lemma R for CP and TOTAL . . . 167
2.8.3 No Lemma R for CONS . . . 168
2.9 LIM is a proper subset of BC . . . 193
2.9.1 Enumerating enough total strategies . . . 194
2.9.2 The diagonalization process . . . 195
2.9.3 The separating class . . . 211
2.9.4 The separating class is in BC . . . 214
2.10 TOTAL is a proper subset of CONS . . . 216
2.10.1 TOTAL is a subset of CONS . . . 216
2.10.2 The separating class . . . 219
2.11 $\mathcal{R}$ is not in BC . . . 241
2.12 The union of classes . . . 250

Chapter 1

Partial recursive functions

theory *Partial-Recursive*
  imports *Main HOL−Library.Nat-Bijection*
begin

This chapter lays the foundation for Chapter 2. Essentially it develops recursion theory up to the point of certain fixed-point theorems. This in turn requires standard results such as the existence of a universal function and the *s*-*m*-*n* theorem. Besides these, the chapter contains some results, mostly regarding decidability and the Kleene normal form, that are not strictly needed later. They are included as relatively low-hanging fruits to round off the chapter.

The formalization of partial recursive functions is very much inspired by the Universal Turing Machine AFP entry by Xu et al. [18]. It models partial recursive functions as algorithms whose semantics is given by an evaluation function. This works well for most of this chapter. For the next chapter, however, it is beneficial to regard partial recursive functions as "proper" partial functions. To that end, Section 1.12 introduces more conventional and convenient notation for the common special cases of unary and binary partial recursive functions.

Especially for the nontrivial proofs I consulted the classical textbook by Rogers [12], which also partially explains my preferring the traditional term "recursive" to the more modern "computable".

1.1 Basic definitions

1.1.1 Partial recursive functions

To represent partial recursive functions we start from the same datatype as Xu et al. [18], more specifically from Urban's version of the formalization. In fact the datatype *recf* and the function *arity* below have been copied verbatim from it.

datatype *recf* =
  Z
| *S*
| *Id nat nat*
| *Cn nat recf recf list*
| *Pr nat recf recf*
| *Mn nat recf*

fun *arity* :: *recf* ⇒ *nat* **where**

  arity Z = 1
| *arity S = 1*
| *arity (Id m n) = m*
| *arity (Cn n f gs) = n*
| *arity (Pr n f g) = Suc n*
| *arity (Mn n f) = n*

Already we deviate from Xu et al. in that we define a well-formedness predicate for partial recursive functions. Well-formedness essentially means arity constraints are respected when combining *recf*s.

fun *wellf* :: *recf* ⇒ *bool* **where**
  wellf Z = True
| *wellf S = True*
| *wellf (Id m n) = (n < m)*
| *wellf (Cn n f gs) =*
  (n > 0 ∧ (∀ g ∈ set gs. arity g = n ∧ wellf g) ∧ arity f = length gs ∧ wellf f)
| *wellf (Pr n f g) =*
  (arity g = Suc (Suc n) ∧ arity f = n ∧ wellf f ∧ wellf g)
| *wellf (Mn n f) = (n > 0 ∧ arity f = Suc n ∧ wellf f)*

lemma *wellf-arity-nonzero*: *wellf f ⟹ arity f > 0*
  by (*induction f rule*: *arity.induct*) *simp-all*

lemma *wellf-Pr-arity-greater-1*: *wellf (Pr n f g) ⟹ arity (Pr n f g) > 1*
  using *wellf-arity-nonzero* **by** *auto*

For the most part of this chapter this is the meaning of "*f* is an *n*-ary partial recursive function":

abbreviation *recfn* :: *nat* ⇒ *recf* ⇒ *bool* **where**
  recfn n f ≡ wellf f ∧ arity f = n

Some abbreviations for working with *nat option*:

abbreviation *divergent* :: *nat option* ⇒ *bool* (‹- ↑› [*50*] *50*) **where**
  x ↑ ≡ x = None

abbreviation *convergent* :: *nat option* ⇒ *bool* (‹- ↓› [*50*] *50*) **where**
  x ↓ ≡ x ≠ None

abbreviation *convergent-eq* :: *nat option* ⇒ *nat* ⇒ *bool* (**infix** ‹↓=› *50*) **where**
  x ↓= y ≡ x = Some y

abbreviation *convergent-neq* :: *nat option* ⇒ *nat* ⇒ *bool* (**infix** ‹↓≠› *50*) **where**
  x ↓≠ y ≡ x ↓ ∧ x ≠ Some y

In prose the terms "halt", "terminate", "converge", and "defined" will be used interchangeably; likewise for "not halt", "diverge", and "undefined". In names of lemmas, the abbreviations *converg* and *diverg* will be used consistently.

Our second major deviation from Xu et al. [18] is that we model the semantics of a *recf* by combining the value and the termination of a function into one evaluation function with codomain *nat option*, rather than separating both aspects into an evaluation function with codomain *nat* and a termination predicate.

The value of a well-formed partial recursive function applied to a correctly-sized list of arguments:

```
fun eval-wellf :: recf ⇒ nat list ⇒ nat option where
  eval-wellf Z xs ↓= 0
| eval-wellf S xs ↓= Suc (xs ! 0)
| eval-wellf (Id m n) xs ↓= xs ! n
| eval-wellf (Cn n f gs) xs =
   (if ∀ g ∈ set gs. eval-wellf g xs ↓
    then eval-wellf f (map (λg. the (eval-wellf g xs)) gs)
    else None)
| eval-wellf (Pr n f g) [] = undefined
| eval-wellf (Pr n f g) (0 # xs) = eval-wellf f xs
| eval-wellf (Pr n f g) (Suc x # xs) =
   Option.bind (eval-wellf (Pr n f g) (x # xs)) (λv. eval-wellf g (x # v # xs))
| eval-wellf (Mn n f) xs =
   (let E = λz. eval-wellf f (z # xs)
    in if ∃ z. E z ↓= 0 ∧ (∀ y<z. E y ↓)
       then Some (LEAST z. E z ↓= 0 ∧ (∀ y<z. E y ↓))
       else None)
```

We define a function value only if the *recf* is well-formed and its arity matches the number of arguments.

```
definition eval :: recf ⇒ nat list ⇒ nat option where
  recfn (length xs) f ⟹ eval f xs ≡ eval-wellf f xs

lemma eval-Z [simp]: eval Z [x] ↓= 0
  by (simp add: eval-def)

lemma eval-Z′ [simp]: length xs = 1 ⟹ eval Z xs ↓= 0
  by (simp add: eval-def)

lemma eval-S [simp]: eval S [x] ↓= Suc x
  by (simp add: eval-def)

lemma eval-S′ [simp]: length xs = 1 ⟹ eval S xs ↓= Suc (hd xs)
  using eval-def hd-conv-nth[of xs] by fastforce

lemma eval-Id [simp]:
  assumes n < m and m = length xs
  shows eval (Id m n) xs ↓= xs ! n
  using assms by (simp add: eval-def)

lemma eval-Cn [simp]:
  assumes recfn (length xs) (Cn n f gs)
  shows eval (Cn n f gs) xs =
    (if ∀ g∈set gs. eval g xs ↓
     then eval f (map (λg. the (eval g xs)) gs)
     else None)
proof −
  have eval (Cn n f gs) xs = eval-wellf (Cn n f gs) xs
    using assms eval-def by blast
  moreover have ⋀g. g ∈ set gs ⟹ eval-wellf g xs = eval g xs
    using assms eval-def by simp
  ultimately have eval (Cn n f gs) xs =
    (if ∀ g∈set gs. eval g xs ↓
     then eval-wellf f (map (λg. the (eval g xs)) gs)
     else None)
    using map-eq-conv[of λg. the (eval-wellf g xs) gs λg. the (eval g xs)]
```

by (*auto*, *metis*)
moreover have $\bigwedge$*ys. length ys* = *length gs* $\Longrightarrow$ *eval f ys* = *eval-wellf f ys*
using *assms eval-def* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

lemma *eval-Pr-0* [*simp*]:
assumes *recfn* (*Suc n*) (*Pr n f g*) **and** *n* = *length xs*
shows *eval* (*Pr n f g*) (*0* # *xs*) = *eval f xs*
using *assms* **by** (*simp add*: *eval-def*)

lemma *eval-Pr-diverg-Suc* [*simp*]:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and *n* = *length xs*
and *eval* (*Pr n f g*) (*x* # *xs*) ↑
shows *eval* (*Pr n f g*) (*Suc x* # *xs*) ↑
using *assms* **by** (*simp add*: *eval-def*)

lemma *eval-Pr-converg-Suc* [*simp*]:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and *n* = *length xs*
and *eval* (*Pr n f g*) (*x* # *xs*) ↓
shows *eval* (*Pr n f g*) (*Suc x* # *xs*) = *eval g* (*x* # *the* (*eval* (*Pr n f g*) (*x* # *xs*)) # *xs*)
using *assms eval-def* **by** *auto*

lemma *eval-Pr-diverg-add*:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and *n* = *length xs*
and *eval* (*Pr n f g*) (*x* # *xs*) ↑
shows *eval* (*Pr n f g*) ((*x* + *y*) # *xs*) ↑
using *assms* **by** (*induction y*) *auto*

lemma *eval-Pr-converg-le*:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and *n* = *length xs*
and *eval* (*Pr n f g*) (*x* # *xs*) ↓
and *y* ≤ *x*
shows *eval* (*Pr n f g*) (*y* # *xs*) ↓
using *assms eval-Pr-diverg-add le-Suc-ex* **by** *metis*

lemma *eval-Pr-Suc-converg*:
assumes *recfn* (*Suc n*) (*Pr n f g*)
and *n* = *length xs*
and *eval* (*Pr n f g*) (*Suc x* # *xs*) ↓
shows *eval g* (*x* # (*the* (*eval* (*Pr n f g*) (*x* # *xs*))) # *xs*) ↓
and *eval* (*Pr n f g*) (*Suc x* # *xs*) = *eval g* (*x* # *the* (*eval* (*Pr n f g*) (*x* # *xs*)) # *xs*)
using *eval-Pr-converg-Suc*[*of n f g xs x*, *OF assms*(*1*,*2*)]
eval-Pr-converg-le[*of n f g xs Suc x x*, *OF assms*] *assms*(*3*)
by *simp-all*

lemma *eval-Mn* [*simp*]:
assumes *recfn* (*length xs*) (*Mn n f*)
shows *eval* (*Mn n f*) *xs* =
(*if* (∃ *z*. *eval f* (*z* # *xs*) ↓= *0* ∧ (∀ *y*<*z*. *eval f* (*y* # *xs*) ↓))
then Some (*LEAST z*. *eval f* (*z* # *xs*) ↓= *0* ∧ (∀ *y*<*z*. *eval f* (*y* # *xs*) ↓))
else None)

```
  using assms eval-def by auto
```

For μ-recursion, the condition ∀ y<z. *eval-wellf f* (y # xs) ↓ inside *LEAST* in the definition of *eval-wellf* is redundant.

```
lemma eval-wellf-Mn:
  eval-wellf (Mn n f) xs =
    (if (∃ z. eval-wellf f (z # xs) ↓= 0 ∧ (∀ y<z. eval-wellf f (y # xs) ↓))
     then Some (LEAST z. eval-wellf f (z # xs) ↓= 0)
     else None)
proof −
  let ?P = λz. eval-wellf f (z # xs) ↓= 0 ∧ (∀ y<z. eval-wellf f (y # xs) ↓)
  {
    assume ∃ z. ?P z
    moreover define z where z = Least ?P
    ultimately have ?P z
      using LeastI[of ?P] by blast
    have (LEAST z. eval-wellf f (z # xs) ↓= 0) = z
    proof (rule Least-equality)
      show eval-wellf f (z # xs) ↓= 0
        using ‹?P z› by simp
      show z ≤ y if eval-wellf f (y # xs) ↓= 0 for y
      proof (rule ccontr)
        assume ¬ z ≤ y
        then have y < z by simp
        moreover from this have ?P y
          using that ‹?P z› by simp
        ultimately show False
          using that not-less-Least z-def by blast
      qed
    qed
  }
  then show ?thesis by simp
qed

lemma eval-Mn':
  assumes recfn (length xs) (Mn n f)
  shows eval (Mn n f) xs =
   (if (∃ z. eval f (z # xs) ↓= 0 ∧ (∀ y<z. eval f (y # xs) ↓))
    then Some (LEAST z. eval f (z # xs) ↓= 0)
    else None)
  using assms eval-def eval-wellf-Mn by auto
```

Proving that μ-recursion converges is easier if one does not have to deal with *LEAST* directly.

```
lemma eval-Mn-convergI:
  assumes recfn (length xs) (Mn n f)
    and eval f (z # xs) ↓= 0
    and ⋀y. y < z ⟹ eval f (y # xs) ↓≠ 0
  shows eval (Mn n f) xs ↓= z
proof −
  let ?P = λz. eval f (z # xs) ↓= 0 ∧ (∀ y<z. eval f (y # xs) ↓)
  have z = Least ?P
    using Least-equality[of ?P z] assms(2,3) not-le-imp-less by blast
  moreover have ?P z using assms(2,3) by simp
  ultimately show eval (Mn n f) xs ↓= z
```

  using *eval-Mn*[*OF assms*(*1*)] **by** *meson*
qed

Similarly, reasoning from a μ-recursive function is simplified somewhat by the next lemma.

lemma *eval-Mn-convergE*:
  assumes *recfn* (*length xs*) (*Mn n f*) **and** *eval* (*Mn n f*) *xs* ↓= *z*
  shows *z* = (*LEAST z. eval f* (*z* # *xs*) ↓= *0* ∧ (∀ *y*<*z. eval f* (*y* # *xs*) ↓))
    and *eval f* (*z* # *xs*) ↓= *0*
    and ⋀*y. y* < *z* ⟹ *eval f* (*y* # *xs*) ↓≠ *0*
proof −
  let *?P* = λ*z. eval f* (*z* # *xs*) ↓= *0* ∧ (∀ *y*<*z. eval f* (*y* # *xs*) ↓)
  show *z* = *Least ?P*
    using *assms eval-Mn*[*OF assms*(*1*)]
    by (*metis* (*no-types*, *lifting*) *option.inject option.simps*(*3*))
  moreover have ∃ *z. ?P z*
    using *assms eval-Mn*[*OF assms*(*1*)] **by** (*metis option.distinct*(*1*))
  ultimately have *?P z*
    using *LeastI*[*of ?P*] **by** *blast*
  then have *eval f* (*z* # *xs*) ↓= *0* ∧ (∀ *y*<*z. eval f* (*y* # *xs*) ↓)
    by *simp*
  then show *eval f* (*z* # *xs*) ↓= *0* **by** *simp*
  show ⋀*y. y* < *z* ⟹ *eval f* (*y* # *xs*) ↓≠ *0*
    using *not-less-Least*[*of - ?P*] ‹*z* = *Least ?P*› ‹*?P z*› *less-trans* **by** *blast*
qed

lemma *eval-Mn-diverg*:
  assumes *recfn* (*length xs*) (*Mn n f*)
  shows ¬ (∃ *z. eval f* (*z* # *xs*) ↓= *0* ∧ (∀ *y*<*z. eval f* (*y* # *xs*) ↓)) ⟷ *eval* (*Mn n f*) *xs* ↑
  using *assms eval-Mn*[*OF assms*(*1*)] **by** *simp*

1.1.2 Extensional equality

definition *exteq* :: *recf* ⇒ *recf* ⇒ *bool* (**infix** ‹≃› *55*) **where**
  f ≃ *g* ≡ *arity f* = *arity g* ∧ (∀ *xs. length xs* = *arity f* ⟶ *eval f xs* = *eval g xs*)

lemma *exteq-refl*: *f* ≃ *f*
  using *exteq-def* **by** *simp*

lemma *exteq-sym*: *f* ≃ *g* ⟹ *g* ≃ *f*
  using *exteq-def* **by** *simp*

lemma *exteq-trans*: *f* ≃ *g* ⟹ *g* ≃ *h* ⟹ *f* ≃ *h*
  using *exteq-def* **by** *simp*

lemma *exteqI*:
  assumes *arity f* = *arity g* **and** ⋀*xs. length xs* = *arity f* ⟹ *eval f xs* = *eval g xs*
  shows *f* ≃ *g*
  using *assms exteq-def* **by** *simp*

lemma *exteqI1*:
  assumes *arity f* = *1* **and** *arity g* = *1* **and** ⋀*x. eval f* [*x*] = *eval g* [*x*]
  shows *f* ≃ *g*
  using *assms exteqI* **by** (*metis One-nat-def Suc-length-conv length-0-conv*)

For every partial recursive function f there are infinitely many extensionally equal ones, for example, those that wrap *f* arbitrarily often in the identity function.

```
fun wrap-Id :: recf ⇒ nat ⇒ recf where
  wrap-Id f 0 = f
| wrap-Id f (Suc n) = Cn (arity f) (Id 1 0) [wrap-Id f n]

lemma recfn-wrap-Id: recfn a f ⟹ recfn a (wrap-Id f n)
  using wellf-arity-nonzero by (induction n) auto

lemma exteq-wrap-Id: recfn a f ⟹ f ≃ wrap-Id f n
proof (induction n)
  case 0
  then show ?case by (simp add: exteq-refl)
next
  case (Suc n)
  have wrap-Id f n ≃ wrap-Id f (Suc n)
  proof (rule exteqI)
    show arity (wrap-Id f n) = arity (wrap-Id f (Suc n))
      using Suc by (simp add: recfn-wrap-Id)
    show eval (wrap-Id f n) xs = eval (wrap-Id f (Suc n)) xs
      if length xs = arity (wrap-Id f n) for xs
    proof −
      have recfn (length xs) (Cn (arity f) (Id 1 0) [wrap-Id f n])
        using that Suc recfn-wrap-Id by (metis wrap-Id.simps(2))
      then show eval (wrap-Id f n) xs = eval (wrap-Id f (Suc n)) xs
        by auto
    qed
  qed
  then show ?case using Suc exteq-trans by fast
qed

fun depth :: recf ⇒ nat where
  depth Z = 0
| depth S = 0
| depth (Id m n) = 0
| depth (Cn n f gs) = Suc (max (depth f) (Max (set (map depth gs))))
| depth (Pr n f g) = Suc (max (depth f) (depth g))
| depth (Mn n f) = Suc (depth f)

lemma depth-wrap-Id: recfn a f ⟹ depth (wrap-Id f n) = depth f + n
  by (induction n) simp-all

lemma wrap-Id-injective:
  assumes recfn a f and wrap-Id f n_1 = wrap-Id f n_2
  shows n_1 = n_2
  using assms by (metis add-left-cancel depth-wrap-Id)

lemma exteq-infinite:
  assumes recfn a f
  shows infinite {g. recfn a g ∧ g ≃ f} (is infinite ?R)
proof −
  have inj (wrap-Id f)
    using wrap-Id-injective ‹recfn a f› by (meson inj-onI)
  then have infinite (range (wrap-Id f))
    using finite-imageD by blast
  moreover have range (wrap-Id f) ⊆ ?R
```

using *assms exteq-sym exteq-wrap-Id recfn-wrap-Id* **by** *blast*
ultimately show *?thesis* **by** (*simp add*: *infinite-super*)
qed

1.1.3 Primitive recursive and total functions

fun *Mn-free* :: *recf* ⇒ *bool* **where**
Mn-free Z = *True*
| *Mn-free S* = *True*
| *Mn-free* (*Id m n*) = *True*
| *Mn-free* (*Cn n f gs*) = ((∀ *g* ∈ *set gs*. *Mn-free g*) ∧ *Mn-free f*)
| *Mn-free* (*Pr n f g*) = (*Mn-free f* ∧ *Mn-free g*)
| *Mn-free* (*Mn n f*) = *False*

This is our notion of *n*-ary primitive recursive function:

abbreviation *prim-recfn* :: *nat* ⇒ *recf* ⇒ *bool* **where**
prim-recfn n f ≡ *recfn n f* ∧ *Mn-free f*

definition *total* :: *recf* ⇒ *bool* **where**
total f ≡ ∀ *xs*. *length xs* = *arity f* ⟶ *eval f xs* ↓

lemma *totalI* [*intro*]:
assumes ⋀*xs*. *length xs* = *arity f* ⟹ *eval f xs* ↓
shows *total f*
using *assms total-def* **by** *simp*

lemma *totalE* [*simp*]:
assumes *total f* **and** *recfn n f* **and** *length xs* = *n*
shows *eval f xs* ↓
using *assms total-def* **by** *simp*

lemma *totalI1* :
assumes *recfn 1 f* **and** ⋀*x*. *eval f* [*x*] ↓
shows *total f*
using *assms totalI*[*of f*] **by** (*metis One-nat-def length-0-conv length-Suc-conv*)

lemma *totalI2*:
assumes *recfn 2 f* **and** ⋀*x y*. *eval f* [*x*, *y*] ↓
shows *total f*
using *assms totalI*[*of f*] **by** (*smt length-0-conv length-Suc-conv numeral-2-eq-2*)

lemma *totalI3*:
assumes *recfn 3 f* **and** ⋀*x y z*. *eval f* [*x*, *y*, *z*] ↓
shows *total f*
using *assms totalI*[*of f*] **by** (*smt length-0-conv length-Suc-conv numeral-3-eq-3*)

lemma *totalI4*:
assumes *recfn 4 f* **and** ⋀*w x y z*. *eval f* [*w*, *x*, *y*, *z*] ↓
shows *total f*
proof (*rule totalI*[*of f*])
fix *xs* :: *nat list*
assume *length xs* = *arity f*
then have *length xs* = *Suc* (*Suc* (*Suc* (*Suc 0*)))
using *assms*(*1*) **by** *simp*
then obtain *w x y z* **where** *xs* = [*w*, *x*, *y*, *z*]
by (*smt Suc-length-conv length-0-conv*)

```
  then show eval f xs ↓ using assms(2) by simp
qed

lemma Mn-free-imp-total [intro]:
  assumes wellf f and Mn-free f
  shows total f
  using assms
proof (induction f rule: Mn-free.induct)
  case (5 n f g)
  have eval (Pr n f g) (x # xs) ↓ if length (x # xs) = arity (Pr n f g) for x xs
    using 5 that by (induction x) auto
  then show ?case by (metis arity.simps(5) length-Suc-conv totalI)
qed (auto simp add: total-def eval-def)

lemma prim-recfn-total: prim-recfn n f ⟹ total f
  using Mn-free-imp-total by simp

lemma eval-Pr-prim-Suc:
  assumes h = Pr n f g and prim-recfn (Suc n) h and length xs = n
  shows eval h (Suc x # xs) = eval g (x # the (eval h (x # xs)) # xs)
  using assms eval-Pr-converg-Suc prim-recfn-total by simp

lemma Cn-total:
  assumes ∀ g∈set gs. total g and total f and recfn n (Cn n f gs)
  shows total (Cn n f gs)
  using assms by (simp add: totalI)

lemma Pr-total:
  assumes total f and total g and recfn (Suc n) (Pr n f g)
  shows total (Pr n f g)
proof −
  have eval (Pr n f g) (x # xs) ↓ if length xs = n for x xs
    using that assms by (induction x) auto
  then show ?thesis
    using assms(3) totalI by (metis Suc-length-conv arity.simps(5))
qed

lemma eval-Mn-total:
  assumes recfn (length xs) (Mn n f) and total f
  shows eval (Mn n f) xs =
    (if (∃ z. eval f (z # xs) ↓= 0)
     then Some (LEAST z. eval f (z # xs) ↓= 0)
     else None)
  using assms by auto
```

1.2 Simple functions

This section, too, bears some similarity to Urban's formalization in Xu et al. [18], but is more minimalistic in scope.

As a general naming rule, instances of *recf* and functions returning such instances get names starting with *r-*. Typically, for an *r-xyz* there will be a lemma *r-xyz-recfn* or *r-xyz-prim* establishing its (primitive) recursiveness, arity, and well-formedness. Moreover there will be a lemma *r-xyz* describing its semantics, for which we will sometimes introduce an Isabelle function *xyz*.

1.2.1 Manipulating parameters

Appending dummy parameters:

definition *r-dummy* :: *nat* ⇒ *recf* ⇒ *recf* **where**
r-dummy n f ≡ *Cn* (*arity f* + *n*) *f* (*map* (λ*i*. *Id* (*arity f* + *n*) *i*) [*0*..<*arity f*])

lemma *r-dummy-prim* [*simp*]:
prim-recfn a f ⟹ *prim-recfn* (*a* + *n*) (*r-dummy n f*)
using *wellf-arity-nonzero* **by** (*auto simp add*: *r-dummy-def*)

lemma *r-dummy-recfn* [*simp*]:
recfn a f ⟹ *recfn* (*a* + *n*) (*r-dummy n f*)
using *wellf-arity-nonzero* **by** (*auto simp add*: *r-dummy-def*)

lemma *r-dummy* [*simp*]:
r-dummy n f = *Cn* (*arity f* + *n*) *f* (*map* (λ*i*. *Id* (*arity f* + *n*) *i*) [*0*..<*arity f*])
unfolding *r-dummy-def* **by** *simp*

lemma *r-dummy-append*:
assumes *recfn* (*length xs*) *f* **and** *length ys* = *n*
shows *eval* (*r-dummy n f*) (*xs* @ *ys*) = *eval f xs*
proof −
let *?r* = *r-dummy n f*
let *?gs* = *map* (λ*i*. *Id* (*arity f* + *n*) *i*) [*0*..<*arity f*]
have *length ?gs* = *arity f* **by** *simp*
moreover have *?gs* ! *i* = (*Id* (*arity f* + *n*) *i*) **if** *i* < *arity f* **for** *i*
by (*simp add*: *that*)
moreover have ∗: *eval-wellf* (*?gs* ! *i*) (*xs* @ *ys*) ↓= *xs* ! *i* **if** *i* < *arity f* **for** *i*
using *that assms* **by** (*simp add*: *nth-append*)
ultimately have *map* (λ*g*. *the* (*eval-wellf g* (*xs* @ *ys*))) *?gs* = *xs*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *length-map nth-equalityI nth-map option.sel*)
moreover have ∀ *g* ∈ *set ?gs*. *eval-wellf g* (*xs* @ *ys*) ↓
using ∗ **by** *simp*
moreover have *recfn* (*length* (*xs* @ *ys*)) *?r*
using *assms r-dummy-recfn* **by** *fastforce*
ultimately show *?thesis*
by (*auto simp add*: *assms eval-def*)
qed

Shrinking a binary function to a unary one is useful when we want to define a unary function via the *Pr* operation, which can only construct *recf*s of arity two or higher.

definition *r-shrink* :: *recf* ⇒ *recf* **where**
r-shrink f ≡ *Cn 1 f* [*Id 1 0*, *Id 1 0*]

lemma *r-shrink-prim* [*simp*]: *prim-recfn 2 f* ⟹ *prim-recfn 1* (*r-shrink f*)
by (*simp add*: *r-shrink-def*)

lemma *r-shrink-recfn* [*simp*]: *recfn 2 f* ⟹ *recfn 1* (*r-shrink f*)
by (*simp add*: *r-shrink-def*)

lemma *r-shrink* [*simp*]: *recfn 2 f* ⟹ *eval* (*r-shrink f*) [*x*] = *eval f* [*x*, *x*]
by (*simp add*: *r-shrink-def*)

definition *r-swap* :: *recf* ⇒ *recf* **where**
r-swap f ≡ *Cn 2 f* [*Id 2 1*, *Id 2 0*]

lemma *r-swap-recfn* [*simp*]: *recfn 2 f* ⟹ *recfn 2* (*r-swap f*)
  by (*simp add*: *r-swap-def*)

lemma *r-swap-prim* [*simp*]: *prim-recfn 2 f* ⟹ *prim-recfn 2* (*r-swap f*)
  by (*simp add*: *r-swap-def*)

lemma *r-swap* [*simp*]: *recfn 2 f* ⟹ *eval* (*r-swap f*) [*x*, *y*] = *eval f* [*y*, *x*]
  by (*simp add*: *r-swap-def*)

Prepending one dummy parameter:

definition *r-shift* :: *recf* ⇒ *recf* **where**
  r-shift f ≡ *Cn* (*Suc* (*arity f*)) *f* (*map* (λ*i*. *Id* (*Suc* (*arity f*)) (*Suc i*)) [*0*..<*arity f*])

lemma *r-shift-prim* [*simp*]: *prim-recfn a f* ⟹ *prim-recfn* (*Suc a*) (*r-shift f*)
  by (*simp add*: *r-shift-def*)

lemma *r-shift-recfn* [*simp*]: *recfn a f* ⟹ *recfn* (*Suc a*) (*r-shift f*)
  by (*simp add*: *r-shift-def*)

lemma *r-shift* [*simp*]:
  assumes *recfn* (*length xs*) *f*
  shows *eval* (*r-shift f*) (*x* # *xs*) = *eval f xs*
proof −
  let *?r* = *r-shift f*
  let *?gs* = *map* (λ*i*. *Id* (*Suc* (*arity f*)) (*Suc i*)) [*0*..<*arity f*]
  have *length ?gs* = *arity f* **by** *simp*
  moreover have *?gs* ! *i* = (*Id* (*Suc* (*arity f*)) (*Suc i*)) **if** *i* < *arity f* **for** *i*
    by (*simp add*: *that*)
  moreover have ∗: *eval* (*?gs* ! *i*) (*x* # *xs*) ↓= *xs* ! *i* **if** *i* < *arity f* **for** *i*
    using *assms nth-append that* **by** *simp*
  ultimately have *map* (λ*g*. *the* (*eval g* (*x* # *xs*))) *?gs* = *xs*
    by (*metis* (*no-types*, *lifting*) *assms length-map nth-equalityI nth-map option.sel*)
  moreover have ∀ *g* ∈ *set ?gs*. *eval g* (*x* # *xs*) ≠ *None*
    using ∗ **by** *simp*
  ultimately show *?thesis* **using** *r-shift-def assms* **by** *simp*
qed

1.2.2 Arithmetic and logic

The unary constants:

fun *r-const* :: *nat* ⇒ *recf* **where**
  r-const 0 = *Z*
| *r-const* (*Suc c*) = *Cn 1 S* [*r-const c*]

lemma *r-const-prim* [*simp*]: *prim-recfn 1* (*r-const c*)
  by (*induction c*) (*simp-all*)

lemma *r-const* [*simp*]: *eval* (*r-const c*) [*x*] ↓= *c*
  by (*induction c*) *simp-all*

Constants of higher arities:

definition *r-constn n c* ≡ *if n* = *0 then r-const c else r-dummy n* (*r-const c*)

lemma *r-constn-prim* [*simp*]: *prim-recfn* (*Suc n*) (*r-constn n c*)

unfolding *r-constn-def* **by** *simp*

lemma *r-constn* [*simp*]: *length xs* = *Suc n* ⟹ *eval* (*r-constn n c*) *xs* ↓= *c*
unfolding *r-constn-def*
by *simp* (*metis length-0-conv length-Suc-conv r-const*)

We introduce addition, subtraction, and multiplication, but interestingly enough we can make do without division.

definition *r-add* ≡ *Pr 1* (*Id 1 0*) (*Cn 3 S* [*Id 3 1*])

lemma *r-add-prim* [*simp*]: *prim-recfn 2 r-add*
by (*simp add*: *r-add-def*)

lemma *r-add* [*simp*]: *eval r-add* [*a*, *b*] ↓= *a* + *b*
unfolding *r-add-def* **by** (*induction a*) *simp-all*

definition *r-mul* ≡ *Pr 1 Z* (*Cn 3 r-add* [*Id 3 1*, *Id 3 2*])

lemma *r-mul-prim* [*simp*]: *prim-recfn 2 r-mul*
unfolding *r-mul-def* **by** *simp*

lemma *r-mul* [*simp*]: *eval r-mul* [*a*, *b*] ↓= *a* ∗ *b*
unfolding *r-mul-def* **by** (*induction a*) *simp-all*

definition *r-dec* ≡ *Cn 1* (*Pr 1 Z* (*Id 3 0*)) [*Id 1 0*, *Id 1 0*]

lemma *r-dec-prim* [*simp*]: *prim-recfn 1 r-dec*
by (*simp add*: *r-dec-def*)

lemma *r-dec* [*simp*]: *eval r-dec* [*a*] ↓= *a* − *1*
proof −
have *eval* (*Pr 1 Z* (*Id 3 0*)) [*x*, *y*] ↓= *x* − *1* **for** *x y*
by (*induction x*) *simp-all*
then show *?thesis* **by** (*simp add*: *r-dec-def*)
qed

definition *r-sub* ≡ *r-swap* (*Pr 1* (*Id 1 0*) (*Cn 3 r-dec* [*Id 3 1*]))

lemma *r-sub-prim* [*simp*]: *prim-recfn 2 r-sub*
unfolding *r-sub-def* **by** *simp*

lemma *r-sub* [*simp*]: *eval r-sub* [*a*, *b*] ↓= *a* − *b*
proof −
have *eval* (*Pr 1* (*Id 1 0*) (*Cn 3 r-dec* [*Id 3 1*])) [*x*, *y*] ↓= *y* − *x* **for** *x y*
by (*induction x*) *simp-all*
then show *?thesis* **unfolding** *r-sub-def* **by** *simp*
qed

definition *r-sign* ≡ *r-shrink* (*Pr 1 Z* (*r-constn 2 1*))

lemma *r-sign-prim* [*simp*]: *prim-recfn 1 r-sign*
unfolding *r-sign-def* **by** *simp*

lemma *r-sign* [*simp*]: *eval r-sign* [*x*] ↓= (*if x* = *0 then 0 else 1*)
proof −
have *eval* (*Pr 1 Z* (*r-constn 2 1*)) [*x*, *y*] ↓= (*if x* = *0 then 0 else 1*) **for** *x y*

by (*induction x*) *simp-all*
then show *?thesis* **unfolding** *r-sign-def* **by** *simp*
qed

In the logical functions, true will be represented by zero, and false will be represented by non-zero as argument and by one as result.

definition *r-not* ≡ *Cn 1 r-sub* [*r-const 1*, *r-sign*]

lemma *r-not-prim* [*simp*]: *prim-recfn 1 r-not*
unfolding *r-not-def* **by** *simp*

lemma *r-not* [*simp*]: *eval r-not* [*x*] ↓= (*if x* = *0 then 1 else 0*)
unfolding *r-not-def* **by** *simp*

definition *r-nand* ≡ *Cn 2 r-not* [*r-add*]

lemma *r-nand-prim* [*simp*]: *prim-recfn 2 r-nand*
unfolding *r-nand-def* **by** *simp*

lemma *r-nand* [*simp*]: *eval r-nand* [*x*, *y*] ↓= (*if x* = *0* ∧ *y* = *0 then 1 else 0*)
unfolding *r-nand-def* **by** *simp*

definition *r-and* ≡ *Cn 2 r-not* [*r-nand*]

lemma *r-and-prim* [*simp*]: *prim-recfn 2 r-and*
unfolding *r-and-def* **by** *simp*

lemma *r-and* [*simp*]: *eval r-and* [*x*, *y*] ↓= (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
unfolding *r-and-def* **by** *simp*

definition *r-or* ≡ *Cn 2 r-sign* [*r-mul*]

lemma *r-or-prim* [*simp*]: *prim-recfn 2 r-or*
unfolding *r-or-def* **by** *simp*

lemma *r-or* [*simp*]: *eval r-or* [*x*, *y*] ↓= (*if x* = *0* ∨ *y* = *0 then 0 else 1*)
unfolding *r-or-def* **by** *simp*

1.2.3 Comparison and conditions

definition *r-ifz* ≡
let ifzero = (*Cn 3 r-mul* [*r-dummy 2 r-not*, *Id 3 1*]);
ifnzero = (*Cn 3 r-mul* [*r-dummy 2 r-sign*, *Id 3 2*])
in Cn 3 r-add [*ifzero*, *ifnzero*]

lemma *r-ifz-prim* [*simp*]: *prim-recfn 3 r-ifz*
unfolding *r-ifz-def* **by** *simp*

lemma *r-ifz* [*simp*]: *eval r-ifz* [*cond*, *val0*, *val1*] ↓= (*if cond* = *0 then val0 else val1*)
unfolding *r-ifz-def* **by** (*simp add*: *Let-def*)

definition *r-eq* ≡ *Cn 2 r-sign* [*Cn 2 r-add* [*r-sub*, *r-swap r-sub*]]

lemma *r-eq-prim* [*simp*]: *prim-recfn 2 r-eq*
unfolding *r-eq-def* **by** *simp*

lemma *r-eq* [*simp*]: *eval r-eq* [x, y] ↓= (*if* $x = y$ *then 0 else 1*)
  unfolding *r-eq-def* **by** *simp*

definition *r-ifeq* ≡ *Cn 4 r-ifz* [*r-dummy 2 r-eq*, *Id 4 2*, *Id 4 3*]

lemma *r-ifeq-prim* [*simp*]: *prim-recfn 4 r-ifeq*
  unfolding *r-ifeq-def* **by** *simp*

lemma *r-ifeq* [*simp*]: *eval r-ifeq* [a, b, v_0, v_1] ↓= (*if* $a = b$ *then* v_0 *else* v_1)
  unfolding *r-ifeq-def* **using** *r-dummy-append*[*of r-eq* [a, b] [v_0, v_1] *2*]
  by *simp*

definition *r-neq* ≡ *Cn 2 r-not* [*r-eq*]

lemma *r-neq-prim* [*simp*]: *prim-recfn 2 r-neq*
  unfolding *r-neq-def* **by** *simp*

lemma *r-neq* [*simp*]: *eval r-neq* [x, y] ↓= (*if* $x = y$ *then 1 else 0*)
  unfolding *r-neq-def* **by** *simp*

definition *r-ifle* ≡ *Cn 4 r-ifz* [*r-dummy 2 r-sub*, *Id 4 2*, *Id 4 3*]

lemma *r-ifle-prim* [*simp*]: *prim-recfn 4 r-ifle*
  unfolding *r-ifle-def* **by** *simp*

lemma *r-ifle* [*simp*]: *eval r-ifle* [a, b, v_0, v_1] ↓= (*if* $a \leq b$ *then* v_0 *else* v_1)
  unfolding *r-ifle-def* **using** *r-dummy-append*[*of r-sub* [a, b] [v_0, v_1] *2*]
  by *simp*

definition *r-ifless* ≡ *Cn 4 r-ifle* [*Id 4 1*, *Id 4 0*, *Id 4 3*, *Id 4 2*]

lemma *r-ifless-prim* [*simp*]: *prim-recfn 4 r-ifless*
  unfolding *r-ifless-def* **by** *simp*

lemma *r-ifless* [*simp*]: *eval r-ifless* [a, b, v_0, v_1] ↓= (*if* $a < b$ *then* v_0 *else* v_1)
  unfolding *r-ifless-def* **by** *simp*

definition *r-less* ≡ *Cn 2 r-ifle* [*Id 2 1*, *Id 2 0*, *r-constn 1 1*, *r-constn 1 0*]

lemma *r-less-prim* [*simp*]: *prim-recfn 2 r-less*
  unfolding *r-less-def* **by** *simp*

lemma *r-less* [*simp*]: *eval r-less* [x, y] ↓= (*if* $x < y$ *then 0 else 1*)
  unfolding *r-less-def* **by** *simp*

definition *r-le* ≡ *Cn 2 r-ifle* [*Id 2 0*, *Id 2 1*, *r-constn 1 0*, *r-constn 1 1*]

lemma *r-le-prim* [*simp*]: *prim-recfn 2 r-le*
  unfolding *r-le-def* **by** *simp*

lemma *r-le* [*simp*]: *eval r-le* [x, y] ↓= (*if* $x \leq y$ *then 0 else 1*)
  unfolding *r-le-def* **by** *simp*

Arguments are evaluated eagerly. Therefore *r-ifz*, etc. cannot be combined with a diverging function to implement a conditionally diverging function in the naive way. The following function implements a special case needed in the next section. A general lazy version of *r-ifz* will be introduced later with the help of a universal function.

```
definition r-ifeq-else-diverg ≡
  Cn 3 r-add [Id 3 2, Mn 3 (Cn 4 r-add [Id 4 0, Cn 4 r-eq [Id 4 1, Id 4 2]])]

lemma r-ifeq-else-diverg-recfn [simp]: recfn 3 r-ifeq-else-diverg
  unfolding r-ifeq-else-diverg-def by simp

lemma r-ifeq-else-diverg [simp]:
  eval r-ifeq-else-diverg [a, b, v] = (if a = b then Some v else None)
  unfolding r-ifeq-else-diverg-def by simp
```

1.3 The halting problem

Decidability will be treated more thoroughly in Section 1.10. But the halting problem is prominent enough to deserve an early mention.

```
definition decidable :: nat set ⇒ bool where
  decidable X ≡ ∃f. recfn 1 f ∧ (∀x. eval f [x] ↓= (if x ∈ X then 1 else 0))
```

No matter how partial recursive functions are encoded as natural numbers, the set of all codes of functions halting on their own code is undecidable.

```
theorem halting-problem-undecidable:
  fixes code :: nat ⇒ recf
  assumes ⋀f. recfn 1 f ⟹ ∃i. code i = f
  shows ¬ decidable {x. eval (code x) [x] ↓} (is ¬ decidable ?K)
proof
  assume decidable ?K
  then obtain f where recfn 1 f and f: ∀x. eval f [x] ↓= (if x ∈ ?K then 1 else 0)
    using decidable-def by auto
  define g where g ≡ Cn 1 r-ifeq-else-diverg [f, Z, Z]
  then have recfn 1 g
    using ‹recfn 1 f› r-ifeq-else-diverg-recfn by simp
  with assms obtain i where i: code i = g by auto
  from g-def have eval g [x] = (if x ∉ ?K then Some 0 else None) for x
    using r-ifeq-else-diverg-recfn ‹recfn 1 f› f by simp
  then have eval g [i] ↓ ⟷ i ∉ ?K by simp
  also have ... ⟷ eval (code i) [i] ↑ by simp
  also have ... ⟷ eval g [i] ↑
    using i by simp
  finally have eval g [i] ↓ ⟷ eval g [i] ↑ .
  then show False by auto
qed
```

1.4 Encoding tuples and lists

This section is based on the Cantor encoding for pairs. Tuples are encoded by repeated application of the pairing function, lists by pairing their length with the code for a tuple. Thus tuples have a fixed length that must be known when decoding, whereas lists are dynamically sized and know their current length.

1.4.1 Pairs and tuples

The Cantor pairing function

definition *r-triangle* ≡ *r-shrink* (*Pr 1 Z* (*r-dummy 1* (*Cn 2 S* [*r-add*])))

lemma *r-triangle-prim*: *prim-recfn 1 r-triangle*
  unfolding *r-triangle-def* **by** *simp*

lemma *r-triangle*: *eval r-triangle* [*n*] ↓= *Sum* {*0..n*}
proof −
  let *?r* = *r-dummy 1* (*Cn 2 S* [*r-add*])
  have *eval ?r* [*x*, *y*, *z*] ↓= *Suc* (*x* + *y*) **for** *x y z*
    using *r-dummy-append*[*of Cn 2 S* [*r-add*] [*x*, *y*] [*z*] *1*] **by** *simp*
  then have *eval* (*Pr 1 Z ?r*) [*x*, *y*] ↓= *Sum* {*0..x*} **for** *x y*
    by (*induction x*) *simp-all*
  then show *?thesis* **unfolding** *r-triangle-def* **by** *simp*
qed

lemma *r-triangle-eq-triangle* [*simp*]: *eval r-triangle* [*n*] ↓= *triangle n*
  using *r-triangle gauss-sum-nat triangle-def* **by** *simp*

definition *r-prod-encode* ≡ *Cn 2 r-add* [*Cn 2 r-triangle* [*r-add*], *Id 2 0*]

lemma *r-prod-encode-prim* [*simp*]: *prim-recfn 2 r-prod-encode*
  unfolding *r-prod-encode-def* **using** *r-triangle-prim* **by** *simp*

lemma *r-prod-encode* [*simp*]: *eval r-prod-encode* [*m*, *n*] ↓= *prod-encode* (*m*, *n*)
  unfolding *r-prod-encode-def prod-encode-def* **using** *r-triangle-prim* **by** *simp*

These abbreviations are just two more things borrowed from Xu et al. [18].

abbreviation *pdec1 z* ≡ *fst* (*prod-decode z*)

abbreviation *pdec2 z* ≡ *snd* (*prod-decode z*)

lemma *pdec1-le*: *pdec1 i* ≤ *i*
  by (*metis le-prod-encode-1 prod.collapse prod-decode-inverse*)

lemma *pdec2-le*: *pdec2 i* ≤ *i*
  by (*metis le-prod-encode-2 prod.collapse prod-decode-inverse*)

lemma *pdec-less*: *pdec2 i* < *Suc i*
  using *pdec2-le* **by** (*simp add*: *le-imp-less-Suc*)

lemma *pdec1-zero*: *pdec1 0* = *0*
  using *pdec1-le* **by** *auto*

definition *r-maxletr* ≡
  Pr 1 Z (*Cn 3 r-ifle* [*r-dummy 2* (*Cn 1 r-triangle* [*S*]), *Id 3 2*, *Cn 3 S* [*Id 3 0*], *Id 3 1*])

lemma *r-maxletr-prim*: *prim-recfn 2 r-maxletr*
  unfolding *r-maxletr-def* **using** *r-triangle-prim* **by** *simp*

lemma *not-Suc-Greatest-not-Suc*:
  assumes ¬ *P* (*Suc x*) **and** ∃*x*. *P x*
  shows (*GREATEST y*. *y* ≤ *x* ∧ *P y*) = (*GREATEST y*. *y* ≤ *Suc x* ∧ *P y*)
  using *assms* **by** (*metis le-SucI le-Suc-eq*)

lemma *r-maxletr*: *eval r-maxletr* $[x_0, x_1] \downarrow= (GREATEST\ y.\ y \leq x_0 \wedge triangle\ y \leq x_1)$
proof −
  let *?g = Cn 3 r-ifle* [*r-dummy 2* (*Cn 1 r-triangle* [*S*]), *Id 3 2*, *Cn 3 S* [*Id 3 0*], *Id 3 1*]
  have *greatest*:
    $(if\ triangle\ (Suc\ x_0) \leq x_1\ then\ Suc\ x_0\ else\ (GREATEST\ y.\ y \leq x_0 \wedge triangle\ y \leq x_1)) =$
    $(GREATEST\ y.\ y \leq Suc\ x_0 \wedge triangle\ y \leq x_1)$
    for x_0 x_1
  proof $(cases\ triangle\ (Suc\ x_0) \leq x_1)$
    case *True*
    then show *?thesis*
      using *Greatest-equality*$[of\ \lambda y.\ y \leq Suc\ x_0 \wedge triangle\ y \leq x_1]$ **by** *fastforce*
  next
    case *False*
    then show *?thesis*
      using *not-Suc-Greatest-not-Suc*$[of\ \lambda y.\ triangle\ y \leq x_1\ x_0]$ **by** *fastforce*
  qed
  show *?thesis*
    unfolding *r-maxletr-def* **using** *r-triangle-prim*
  proof $(induction\ x_0)$
    case *0*
    then show *?case*
      using *Greatest-equality*$[of\ \lambda y.\ y \leq 0 \wedge triangle\ y \leq x_1\ 0]$ **by** *simp*
  next
    case $(Suc\ x_0)$
    then show *?case* **using** *greatest* **by** *simp*
  qed
qed

definition *r-maxlt* ≡ *r-shrink r-maxletr*

lemma *r-maxlt-prim*: *prim-recfn 1 r-maxlt*
  unfolding *r-maxlt-def* **using** *r-maxletr-prim* **by** *simp*

lemma *r-maxlt*: *eval r-maxlt* $[e] \downarrow= (GREATEST\ y.\ triangle\ y \leq e)$
proof −
  have $y \leq triangle\ y$ **for** y
    by $(induction\ y)$ *auto*
  then have $triangle\ y \leq e \Longrightarrow y \leq e$ **for** $y\ e$
    using *order-trans* **by** *blast*
  then have $(GREATEST\ y.\ y \leq e \wedge triangle\ y \leq e) = (GREATEST\ y.\ triangle\ y \leq e)$
    by *metis*
  moreover have *eval r-maxlt* $[e] \downarrow= (GREATEST\ y.\ y \leq e \wedge triangle\ y \leq e)$
    using *r-maxletr r-shrink r-maxlt-def r-maxletr-prim* **by** *fastforce*
  ultimately show *?thesis* **by** *simp*
qed

definition *pdec1'* $e \equiv e - triangle\ (GREATEST\ y.\ triangle\ y \leq e)$

definition *pdec2'* $e \equiv (GREATEST\ y.\ triangle\ y \leq e) - pdec1'\ e$

lemma *max-triangle-bound*: $triangle\ z \leq e \Longrightarrow z \leq e$
  by (*metis Suc-pred add-leD2 less-Suc-eq triangle-Suc zero-le zero-less-Suc*)

lemma *triangle-greatest-le*: $triangle\ (GREATEST\ y.\ triangle\ y \leq e) \leq e$
  using *max-triangle-bound GreatestI-nat*$[of\ \lambda y.\ triangle\ y \leq e\ 0\ e]$ **by** *simp*

lemma *prod-encode-pdec'*: *prod-encode* (*pdec1' e*, *pdec2' e*) = *e*
proof −
  let *?P* = *λy. triangle y* ≤ *e*
  let *?y* = *GREATEST y. ?P y*
  have *pdec1' e* ≤ *?y*
  proof (*rule ccontr*)
    assume ¬ *pdec1' e* ≤ *?y*
    then have *e* − *triangle ?y* > *?y*
      using *pdec1'-def* **by** *simp*
    then have *?P* (*Suc ?y*) **by** *simp*
    moreover have ∀ *z. ?P z* ⟶ *z* ≤ *e*
      using *max-triangle-bound* **by** *simp*
    ultimately have *Suc ?y* ≤ *?y*
      using *Greatest-le-nat*[*of ?P Suc ?y e*] **by** *blast*
    then show *False* **by** *simp*
  qed
  then have *pdec1' e* + *pdec2' e* = *?y*
    using *pdec1'-def pdec2'-def* **by** *simp*
  then have *prod-encode* (*pdec1' e*, *pdec2' e*) = *triangle ?y* + *pdec1' e*
    by (*simp add*: *prod-encode-def*)
  then show *?thesis* **using** *pdec1'-def triangle-greatest-le* **by** *simp*
qed

lemma *pdec'*:
  pdec1' e = *pdec1 e*
  pdec2' e = *pdec2 e*
  using *prod-encode-pdec' prod-encode-inverse* **by** (*metis fst-conv*, *metis snd-conv*)

definition *r-pdec1* ≡ *Cn 1 r-sub* [*Id 1 0*, *Cn 1 r-triangle* [*r-maxlt*]]

lemma *r-pdec1-prim* [*simp*]: *prim-recfn 1 r-pdec1*
  unfolding *r-pdec1-def* **using** *r-triangle-prim r-maxlt-prim* **by** *simp*

lemma *r-pdec1* [*simp*]: *eval r-pdec1* [*e*] ↓= *pdec1 e*
  unfolding *r-pdec1-def* **using** *r-triangle-prim r-maxlt-prim pdec' pdec1'-def*
  by (*simp add*: *r-maxlt*)

definition *r-pdec2* ≡ *Cn 1 r-sub* [*r-maxlt*, *r-pdec1*]

lemma *r-pdec2-prim* [*simp*]: *prim-recfn 1 r-pdec2*
  unfolding *r-pdec2-def* **using** *r-maxlt-prim* **by** *simp*

lemma *r-pdec2* [*simp*]: *eval r-pdec2* [*e*] ↓= *pdec2 e*
  unfolding *r-pdec2-def* **using** *r-maxlt-prim r-maxlt pdec' pdec2'-def* **by** *simp*

abbreviation *pdec12 i* ≡ *pdec1* (*pdec2 i*)
abbreviation *pdec22 i* ≡ *pdec2* (*pdec2 i*)
abbreviation *pdec122 i* ≡ *pdec1* (*pdec22 i*)
abbreviation *pdec222 i* ≡ *pdec2* (*pdec22 i*)

definition *r-pdec12* ≡ *Cn 1 r-pdec1* [*r-pdec2*]

lemma *r-pdec12-prim* [*simp*]: *prim-recfn 1 r-pdec12*
  unfolding *r-pdec12-def* **by** *simp*

lemma *r-pdec12* [*simp*]: *eval r-pdec12* [*e*] ↓= *pdec12 e*
  unfolding *r-pdec12-def* **by** *simp*

definition *r-pdec22* ≡ *Cn 1 r-pdec2* [*r-pdec2*]

lemma *r-pdec22-prim* [*simp*]: *prim-recfn 1 r-pdec22*
  unfolding *r-pdec22-def* **by** *simp*

lemma *r-pdec22* [*simp*]: *eval r-pdec22* [*e*] ↓= *pdec22 e*
  unfolding *r-pdec22-def* **by** *simp*

definition *r-pdec122* ≡ *Cn 1 r-pdec1* [*r-pdec22*]

lemma *r-pdec122-prim* [*simp*]: *prim-recfn 1 r-pdec122*
  unfolding *r-pdec122-def* **by** *simp*

lemma *r-pdec122* [*simp*]: *eval r-pdec122* [*e*] ↓= *pdec122 e*
  unfolding *r-pdec122-def* **by** *simp*

definition *r-pdec222* ≡ *Cn 1 r-pdec2* [*r-pdec22*]

lemma *r-pdec222-prim* [*simp*]: *prim-recfn 1 r-pdec222*
  unfolding *r-pdec222-def* **by** *simp*

lemma *r-pdec222* [*simp*]: *eval r-pdec222* [*e*] ↓= *pdec222 e*
  unfolding *r-pdec222-def* **by** *simp*

The Cantor tuple function

The empty tuple gets no code, whereas singletons are encoded by their only element and other tuples by recursively applying the pairing function. This yields, for every n, the function *tuple-encode* n, which is a bijection between the natural numbers and the lists of length $(n + 1)$.

fun *tuple-encode* :: *nat* ⇒ *nat list* ⇒ *nat* **where**
  tuple-encode n [] = *undefined*
| *tuple-encode 0* (*x* # *xs*) = *x*
| *tuple-encode* (*Suc n*) (*x* # *xs*) = *prod-encode* (*x*, *tuple-encode n xs*)

lemma *tuple-encode-prod-encode*: *tuple-encode 1* [*x*, *y*] = *prod-encode* (*x*, *y*)
  by *simp*

fun *tuple-decode* **where**
  tuple-decode 0 i = [*i*]
| *tuple-decode* (*Suc n*) *i* = *pdec1 i* # *tuple-decode n* (*pdec2 i*)

lemma *tuple-encode-decode* [*simp*]:
  tuple-encode (*length xs* − *1*) (*tuple-decode* (*length xs* − *1*) *i*) = *i*
proof (*induction length xs* − *1 arbitrary*: *xs i*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then have *length xs* − *1* > *0* **by** *simp*
  with *Suc* **have** ∗: *tuple-encode n* (*tuple-decode n j*) = *j* **for** *j*
    by (*metis diff-Suc-1 length-tl*)

from *Suc* **have** *tuple-decode* (*Suc n*) *i* = *pdec1 i* # *tuple-decode n* (*pdec2 i*)
  using *tuple-decode.simps*(*2*) **by** *blast*
then have *tuple-encode* (*Suc n*) (*tuple-decode* (*Suc n*) *i*) =
    tuple-encode (*Suc n*) (*pdec1 i* # *tuple-decode n* (*pdec2 i*))
  using *Suc* **by** *simp*
also have ... = *prod-encode* (*pdec1 i*, *tuple-encode n* (*tuple-decode n* (*pdec2 i*)))
  by *simp*
also have ... = *prod-encode* (*pdec1 i*, *pdec2 i*)
  using *Suc* ∗ **by** *simp*
also have ... = *i* **by** *simp*
finally have *tuple-encode* (*Suc n*) (*tuple-decode* (*Suc n*) *i*) = *i* **.**
then show *?case* **by** (*simp add*: *Suc.hyps*(*2*))
qed

lemma *tuple-encode-decode′* [*simp*]: *tuple-encode n* (*tuple-decode n i*) = *i*
  using *tuple-encode-decode* **by** (*metis Ex-list-of-length diff-Suc-1 length-Cons*)

lemma *tuple-decode-encode*:
  assumes *length xs* > *0*
  shows *tuple-decode* (*length xs* − *1*) (*tuple-encode* (*length xs* − *1*) *xs*) = *xs*
  using *assms*
proof (*induction length xs* − *1 arbitrary*: *xs*)
  case *0*
  moreover from *this* **have** *length xs* = *1* **by** *linarith*
  ultimately show *?case*
    by (*metis One-nat-def length-0-conv length-Suc-conv tuple-decode.simps*(*1*)
      tuple-encode.simps(*2*))
next
  case (*Suc n*)
  let *?t* = *tl xs*
  let *?i* = *tuple-encode* (*Suc n*) *xs*
  have *length ?t* > *0* **and** *length ?t* − *1* = *n*
    using *Suc* **by** *simp-all*
  then have *tuple-decode n* (*tuple-encode n ?t*) = *?t*
    using *Suc* **by** *blast*
  moreover have *?i* = *prod-encode* (*hd xs*, *tuple-encode n ?t*)
    using *Suc* **by** (*metis hd-Cons-tl length-greater-0-conv tuple-encode.simps*(*3*))
  moreover have *tuple-decode* (*Suc n*) *?i* = *pdec1 ?i* # *tuple-decode n* (*pdec2 ?i*)
    using *tuple-decode.simps*(*2*) **by** *blast*
  ultimately have *tuple-decode* (*Suc n*) *?i* = *xs*
    using *Suc.prems* **by** *simp*
  then show *?case* **by** (*simp add*: *Suc.hyps*(*2*))
qed

lemma *tuple-decode-encode′* [*simp*]:
  assumes *length xs* = *Suc n*
  shows *tuple-decode n* (*tuple-encode n xs*) = *xs*
  using *assms tuple-decode-encode* **by** (*metis diff-Suc-1 zero-less-Suc*)

lemma *tuple-decode-length* [*simp*]: *length* (*tuple-decode n i*) = *Suc n*
  by (*induction n arbitrary*: *i*) *simp-all*

lemma *tuple-decode-nonzero*:
  assumes *n* > *0*
  shows *tuple-decode n i* = *pdec1 i* # *tuple-decode* (*n* − *1*) (*pdec2 i*)
  using *assms* **by** (*metis One-nat-def Suc-pred tuple-decode.simps*(*2*))

The tuple encoding functions are primitive recursive.

fun *r-tuple-encode* :: *nat* ⇒ *recf* **where**
r-tuple-encode 0 = *Id 1 0*
| *r-tuple-encode* (*Suc n*) =
Cn (*Suc* (*Suc n*)) *r-prod-encode* [*Id* (*Suc* (*Suc n*)) *0*, *r-shift* (*r-tuple-encode n*)]

lemma *r-tuple-encode-prim* [*simp*]: *prim-recfn* (*Suc n*) (*r-tuple-encode n*)
by (*induction n*) *simp-all*

lemma *r-tuple-encode*:
assumes *length xs* = *Suc n*
shows *eval* (*r-tuple-encode n*) *xs* ↓= *tuple-encode n xs*
using *assms*
proof (*induction n arbitrary*: *xs*)
case *0*
then show *?case*
by (*metis One-nat-def eval-Id length-Suc-conv nth-Cons-0*
r-tuple-encode.simps(*1*) *tuple-encode.simps*(*2*) *zero-less-one*)
next
case (*Suc n*)
then obtain *y ys* **where** *y-ys*: *y* # *ys* = *xs*
by (*metis length-Suc-conv*)
with *Suc* **have** *eval* (*r-tuple-encode n*) *ys* ↓= *tuple-encode n ys*
by *auto*
with *y-ys* **have** *eval* (*r-shift* (*r-tuple-encode n*)) *xs* ↓= *tuple-encode n ys*
using *Suc.prems r-shift-prim r-tuple-encode-prim* **by** *auto*
moreover have *eval* (*Id* (*Suc* (*Suc n*)) *0*) *xs* ↓= *y*
using *y-ys Suc.prems* **by** *auto*
ultimately have *eval* (*r-tuple-encode* (*Suc n*)) *xs* ↓= *prod-encode* (*y*, *tuple-encode n ys*)
using *Suc.prems* **by** *simp*
then show *?case* **using** *y-ys* **by** *auto*
qed

Functions on encoded tuples

The function for accessing the n-th element of a tuple returns 0 for out-of-bounds access.

definition *e-tuple-nth* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
e-tuple-nth a i n ≡ *if n* ≤ *a then* (*tuple-decode a i*) ! *n else 0*

lemma *e-tuple-nth-le* [*simp*]: *n* ≤ *a* ⟹ *e-tuple-nth a i n* = (*tuple-decode a i*) ! *n*
using *e-tuple-nth-def* **by** *simp*

lemma *e-tuple-nth-gr* [*simp*]: *n* > *a* ⟹ *e-tuple-nth a i n* = *0*
using *e-tuple-nth-def* **by** *simp*

lemma *tuple-decode-pdec2*: *tuple-decode a* (*pdec2 es*) = *tl* (*tuple-decode* (*Suc a*) *es*)
by *simp*

fun *iterate* :: *nat* ⇒ (*'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a*) **where**
iterate 0 f = *id*
| *iterate* (*Suc n*) *f* = *f* ∘ (*iterate n f*)

lemma *iterate-additive*:
assumes *iterate* t_1 *f x* = *y* **and** *iterate* t_2 *f y* = *z*
shows *iterate* (t_1 + t_2) *f x* = *z*

using *assms* **by** (*induction* t_2 *arbitrary*: *z*) *auto*

lemma *iterate-additive′*: *iterate* (t_1 + t_2) *f x* = *iterate* t_2 *f* (*iterate* t_1 *f x*)
using *iterate-additive* **by** *metis*

lemma *e-tuple-nth-elementary*:
assumes *k* ≤ *a*
shows *e-tuple-nth a i k* = (*if a* = *k then* (*iterate k pdec2 i*) *else* (*pdec1* (*iterate k pdec2 i*)))
proof −
have ∗: *tuple-decode* (*a* − *k*) (*iterate k pdec2 i*) = *drop k* (*tuple-decode a i*)
using *assms*
by (*induction k*) (*simp*, *simp add*: *Suc-diff-Suc tuple-decode-pdec2 drop-Suc tl-drop*)
show *?thesis*
proof (*cases a* = *k*)
case *True*
then have *tuple-decode 0* (*iterate k pdec2 i*) = *drop k* (*tuple-decode a i*)
using *assms* ∗ **by** *simp*
moreover from *this* **have** *drop k* (*tuple-decode a i*) = [*tuple-decode a i* ! *k*]
using *assms True* **by** (*metis nth-via-drop tuple-decode.simps*(*1*))
ultimately show *?thesis* **using** *True* **by** *simp*
next
case *False*
with *assms* **have** *a* − *k* > *0* **by** *simp*
with ∗ **have** *tuple-decode* (*a* − *k*) (*iterate k pdec2 i*) = *drop k* (*tuple-decode a i*)
by *simp*
then have *pdec1* (*iterate k pdec2 i*) = *hd* (*drop k* (*tuple-decode a i*))
using *tuple-decode-nonzero* ‹*a* − *k* > *0*› **by** (*metis list.sel*(*1*))
with ‹*a* − *k* > *0*› **have** *pdec1* (*iterate k pdec2 i*) = (*tuple-decode a i*) ! *k*
by (*simp add*: *hd-drop-conv-nth*)
with *False assms* **show** *?thesis* **by** *simp*
qed
qed

definition *r-nth-inbounds* ≡
let r = *Pr 1* (*Id 1 0*) (*Cn 3 r-pdec2* [*Id 3 1*])
in Cn 3 r-ifeq
[*Id 3 0*,
Id 3 2,
Cn 3 r [*Id 3 2*, *Id 3 1*],
Cn 3 r-pdec1 [*Cn 3 r* [*Id 3 2*, *Id 3 1*]]]

lemma *r-nth-inbounds-prim*: *prim-recfn 3 r-nth-inbounds*
unfolding *r-nth-inbounds-def* **by** (*simp add*: *Let-def*)

lemma *r-nth-inbounds*:
k ≤ *a* ⟹ *eval r-nth-inbounds* [*a*, *i*, *k*] ↓= *e-tuple-nth a i k*
eval r-nth-inbounds [*a*, *i*, *k*] ↓
proof −
let *?r* = *Pr 1* (*Id 1 0*) (*Cn 3 r-pdec2* [*Id 3 1*])
let *?h* = *Cn 3 ?r* [*Id 3 2*, *Id 3 1*]
have *eval ?r* [*k*, *i*] ↓= *iterate k pdec2 i* **for** *k i*
using *r-pdec2-prim* **by** (*induction k*) (*simp-all*)
then have *eval ?h* [*a*, *i*, *k*] ↓= *iterate k pdec2 i*
using *r-pdec2-prim* **by** *simp*
then have *eval r-nth-inbounds* [*a*, *i*, *k*] ↓=
(*if a* = *k then iterate k pdec2 i else pdec1* (*iterate k pdec2 i*))

unfolding *r-nth-inbounds-def* **by** (*simp add*: *Let-def*)
then show $k \leq a \Longrightarrow$ *eval r-nth-inbounds* [*a*, *i*, *k*] ↓= *e-tuple-nth a i k*
and *eval r-nth-inbounds* [*a*, *i*, *k*] ↓
using *e-tuple-nth-elementary* **by** *simp-all*
qed

definition *r-tuple-nth* ≡
Cn 3 r-ifle [*Id 3 2*, *Id 3 0*, *r-nth-inbounds*, *r-constn 2 0*]

lemma *r-tuple-nth-prim*: *prim-recfn 3 r-tuple-nth*
unfolding *r-tuple-nth-def* **using** *r-nth-inbounds-prim* **by** *simp*

lemma *r-tuple-nth* [*simp*]: *eval r-tuple-nth* [*a*, *i*, *k*] ↓= *e-tuple-nth a i k*
unfolding *r-tuple-nth-def* **using** *r-nth-inbounds-prim r-nth-inbounds* **by** *simp*

1.4.2 Lists

Encoding and decoding

Lists are encoded by pairing the length of the list with the code for the tuple made up of the list's elements. Then all these codes are incremented in order to make room for the empty list (cf. Rogers [12, p. 71]).

fun *list-encode* :: *nat list* ⇒ *nat* **where**
list-encode [] = *0*
| *list-encode* (*x* # *xs*) = *Suc* (*prod-encode* (*length xs*, *tuple-encode* (*length xs*) (*x* # *xs*)))

lemma *list-encode-0* [*simp*]: *list-encode xs* = *0* ⟷ *xs* = []
using *list-encode.elims Partial-Recursive.list-encode.simps*(*1*) **by** *blast*

lemma *list-encode-1*: *list-encode* [*0*] = *1*
by (*simp add*: *prod-encode-def*)

fun *list-decode* :: *nat* ⇒ *nat list* **where**
list-decode 0 = []
| *list-decode* (*Suc n*) = *tuple-decode* (*pdec1 n*) (*pdec2 n*)

lemma *list-encode-decode* [*simp*]: *list-encode* (*list-decode n*) = *n*
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc k*)
then have ∗: *list-decode n* = *tuple-decode* (*pdec1 k*) (*pdec2 k*) (**is** - = *?t*)
by *simp*
then obtain *x xs* **where** *xxs*: *x* # *xs* = *?t*
by (*metis tuple-decode.elims*)
then have *list-encode ?t* = *list-encode* (*x* # *xs*) **by** *simp*
then have *1*: *list-encode ?t* = *Suc* (*prod-encode* (*length xs*, *tuple-encode* (*length xs*) (*x* # *xs*)))
by *simp*
have *2*: *length xs* = *length ?t* − *1*
using *xxs* **by** (*metis length-tl list.sel*(*3*))
then have *3*: *length xs* = *pdec1 k*
using ∗ **by** *simp*
then have *tuple-encode* (*length ?t* − *1*) *?t* = *pdec2 k*
using *2 tuple-encode-decode* **by** *metis*

then have *list-encode ?t* = *Suc* (*prod-encode* (*pdec1 k*, *pdec2 k*))
  using *1 2 3 xxs* **by** *simp*
with ∗ *Suc* **show** *?thesis* **by** *simp*
qed

lemma *list-decode-encode* [*simp*]: *list-decode* (*list-encode xs*) = *xs*
proof (*cases xs*)
  case *Nil*
  then show *?thesis* **by** *simp*
next
  case (*Cons y ys*)
  then have *list-encode xs* =
      Suc (*prod-encode* (*length ys*, *tuple-encode* (*length ys*) *xs*))
      (**is** - = *Suc ?i*)
    by *simp*
  then have *list-decode* (*Suc ?i*) = *tuple-decode* (*pdec1 ?i*) (*pdec2 ?i*) **by** *simp*
  moreover have *pdec1 ?i* = *length ys* **by** *simp*
  moreover have *pdec2 ?i* = *tuple-encode* (*length ys*) *xs* **by** *simp*
  ultimately have *list-decode* (*Suc ?i*) =
      tuple-decode (*length ys*) (*tuple-encode* (*length ys*) *xs*)
    by *simp*
  moreover have *length ys* = *length xs* − *1*
    using *Cons* **by** *simp*
  ultimately have *list-decode* (*Suc ?i*) =
      tuple-decode (*length xs* − *1*) (*tuple-encode* (*length xs* − *1*) *xs*)
    by *simp*
  then show *?thesis* **using** *Cons* **by** *simp*
qed

abbreviation *singleton-encode* :: *nat* ⇒ *nat* **where**
  singleton-encode x ≡ *list-encode* [*x*]

lemma *list-decode-singleton*: *list-decode* (*singleton-encode x*) = [*x*]
  by *simp*

definition *r-singleton-encode* ≡ *Cn 1 S* [*Cn 1 r-prod-encode* [*Z*, *Id 1 0*]]

lemma *r-singleton-encode-prim* [*simp*]: *prim-recfn 1 r-singleton-encode*
  unfolding *r-singleton-encode-def* **by** *simp*

lemma *r-singleton-encode* [*simp*]: *eval r-singleton-encode* [*x*] ↓= *singleton-encode x*
  unfolding *r-singleton-encode-def* **by** *simp*

definition *r-list-encode* :: *nat* ⇒ *recf* **where**
  r-list-encode n ≡ *Cn* (*Suc n*) *S* [*Cn* (*Suc n*) *r-prod-encode* [*r-constn n n*, *r-tuple-encode n*]]

lemma *r-list-encode-prim* [*simp*]: *prim-recfn* (*Suc n*) (*r-list-encode n*)
  unfolding *r-list-encode-def* **by** *simp*

lemma *r-list-encode*:
  assumes *length xs* = *Suc n*
  shows *eval* (*r-list-encode n*) *xs* ↓= *list-encode xs*
proof −
  have *eval* (*r-tuple-encode n*) *xs* ↓
    by (*simp add*: *assms r-tuple-encode*)
  then have *eval* (*Cn* (*Suc n*) *r-prod-encode* [*r-constn n n*, *r-tuple-encode n*]) *xs* ↓

```
    using assms by simp
  then have eval (r-list-encode n) xs =
      eval S [the (eval (Cn (Suc n) r-prod-encode [r-constn n n, r-tuple-encode n]) xs)]
    unfolding r-list-encode-def using assms r-tuple-encode by simp
  moreover from assms obtain y ys where xs = y # ys
    by (meson length-Suc-conv)
  ultimately show ?thesis
    unfolding r-list-encode-def using assms r-tuple-encode by simp
qed
```

Functions on encoded lists

The functions in this section mimic those on type *nat list*. Their names are prefixed by *e-* and the names of the corresponding *recf*s by *r-*.

```
abbreviation e-tl :: nat ⇒ nat where
  e-tl e ≡ list-encode (tl (list-decode e))
```

In order to turn *e-tl* into a partial recursive function we first represent it in a more elementary way.

```
lemma e-tl-elementary:
  e-tl e =
    (if e = 0 then 0
     else if pdec1 (e − 1) = 0 then 0
     else Suc (prod-encode (pdec1 (e − 1) − 1, pdec22 (e − 1))))
proof (cases e)
  case 0
  then show ?thesis by simp
next
  case Suc-d: (Suc d)
  then show ?thesis
  proof (cases pdec1 d)
    case 0
    then show ?thesis using Suc-d by simp
  next
    case (Suc a)
    have ∗: list-decode e = tuple-decode (pdec1 d) (pdec2 d)
      using Suc-d by simp
    with Suc obtain x xs where xxs: list-decode e = x # xs by simp
    then have ∗∗: e-tl e = list-encode xs by simp
    have list-decode (Suc (prod-encode (pdec1 (e − 1) − 1, pdec22 (e − 1)))) =
        tuple-decode (pdec1 (e − 1) − 1) (pdec22 (e − 1))
        (is ?lhs = -)
      by simp
    also have ... = tuple-decode a (pdec22 (e − 1))
      using Suc Suc-d by simp
    also have ... = tl (tuple-decode (Suc a) (pdec2 (e − 1)))
      using tuple-decode-pdec2 Suc by presburger
    also have ... = tl (tuple-decode (pdec1 (e − 1)) (pdec2 (e − 1)))
      using Suc Suc-d by auto
    also have ... = tl (list-decode e)
      using ∗ Suc-d by simp
    also have ... = xs
      using xxs by simp
    finally have ?lhs = xs .
    then have list-encode ?lhs = list-encode xs by simp
```

then have *Suc* (*prod-encode* (*pdec1* (*e* − *1*) − *1*, *pdec22* (*e* − *1*))) = *list-encode xs*
using *list-encode-decode* **by** *metis*
then show *?thesis* **using** ∗∗ *Suc-d Suc* **by** *simp*
qed
qed

definition *r-tl* ≡
let r = *Cn 1 r-pdec1* [*r-dec*]
in Cn 1 r-ifz
[*Id 1 0*,
Z,
Cn 1 r-ifz
[*r*, *Z*, *Cn 1 S* [*Cn 1 r-prod-encode* [*Cn 1 r-dec* [*r*], *Cn 1 r-pdec22* [*r-dec*]]]]]

lemma *r-tl-prim* [*simp*]: *prim-recfn 1 r-tl*
unfolding *r-tl-def* **by** (*simp add*: *Let-def*)

lemma *r-tl* [*simp*]: *eval r-tl* [*e*] ↓= *e-tl e*
unfolding *r-tl-def* **using** *e-tl-elementary* **by** (*simp add*: *Let-def*)

We define the head of the empty encoded list to be zero.

definition *e-hd* :: *nat* ⇒ *nat* **where**
e-hd e ≡ *if e* = *0 then 0 else hd* (*list-decode e*)

lemma *e-hd* [*simp*]:
assumes *list-decode e* = *x* # *xs*
shows *e-hd e* = *x*
using *e-hd-def assms* **by** *auto*

lemma *e-hd-0* [*simp*]: *e-hd 0* = *0*
using *e-hd-def* **by** *simp*

lemma *e-hd-neq-0* [*simp*]:
assumes *e* ≠ *0*
shows *e-hd e* = *hd* (*list-decode e*)
using *e-hd-def assms* **by** *simp*

definition *r-hd* ≡
Cn 1 r-ifz [*Cn 1 r-pdec1* [*r-dec*], *Cn 1 r-pdec2* [*r-dec*], *Cn 1 r-pdec12* [*r-dec*]]

lemma *r-hd-prim* [*simp*]: *prim-recfn 1 r-hd*
unfolding *r-hd-def* **by** *simp*

lemma *r-hd* [*simp*]: *eval r-hd* [*e*] ↓= *e-hd e*
proof −
have *e-hd e* = (*if pdec1* (*e* − *1*) = *0 then pdec2* (*e* − *1*) *else pdec12* (*e* − *1*))
proof (*cases e*)
case *0*
then show *?thesis* **using** *pdec1-zero pdec2-le* **by** *auto*
next
case (*Suc d*)
then show *?thesis* **by** (*cases pdec1 d*) (*simp-all add*: *pdec1-zero*)
qed
then show *?thesis* **unfolding** *r-hd-def* **by** *simp*
qed

abbreviation *e-length* :: *nat* ⇒ *nat* **where**
e-length e ≡ *length* (*list-decode e*)

lemma *e-length-0*: *e-length e* = *0* ⟹ *e* = *0*
by (*metis list-encode.simps*(*1*) *length-0-conv list-encode-decode*)

definition *r-length* ≡ *Cn 1 r-ifz* [*Id 1 0*, *Z*, *Cn 1 S* [*Cn 1 r-pdec1* [*r-dec*]]]

lemma *r-length-prim* [*simp*]: *prim-recfn 1 r-length*
unfolding *r-length-def* **by** *simp*

lemma *r-length* [*simp*]: *eval r-length* [*e*] ↓= *e-length e*
unfolding *r-length-def* **by** (*cases e*) *simp-all*

Accessing an encoded list out of bounds yields zero.

definition *e-nth* :: *nat* ⇒ *nat* ⇒ *nat* **where**
e-nth e n ≡ *if e* = *0 then 0 else e-tuple-nth* (*pdec1* (*e* − *1*)) (*pdec2* (*e* − *1*)) *n*

lemma *e-nth* [*simp*]:
e-nth e n = (*if n* < *e-length e then* (*list-decode e*) ! *n else 0*)
by (*cases e*) (*simp-all add*: *e-nth-def e-tuple-nth-def*)

lemma *e-hd-nth0*: *e-hd e* = *e-nth e 0*
by (*simp add*: *e-hd-def e-length-0 hd-conv-nth*)

definition *r-nth* ≡
Cn 2 r-ifz
[*Id 2 0*,
r-constn 1 0,
Cn 2 r-tuple-nth
[*Cn 2 r-pdec1* [*r-dummy 1 r-dec*], *Cn 2 r-pdec2* [*r-dummy 1 r-dec*], *Id 2 1*]]

lemma *r-nth-prim* [*simp*]: *prim-recfn 2 r-nth*
unfolding *r-nth-def* **using** *r-tuple-nth-prim* **by** *simp*

lemma *r-nth* [*simp*]: *eval r-nth* [*e*, *n*] ↓= *e-nth e n*
unfolding *r-nth-def e-nth-def* **using** *r-tuple-nth-prim* **by** *simp*

definition *r-rev-aux* ≡
Pr 1 r-hd (*Cn 3 r-prod-encode* [*Cn 3 r-nth* [*Id 3 2*, *Cn 3 S* [*Id 3 0*]], *Id 3 1*])

lemma *r-rev-aux-prim*: *prim-recfn 2 r-rev-aux*
unfolding *r-rev-aux-def* **by** *simp*

lemma *r-rev-aux*:
assumes *list-decode e* = *xs* **and** *length xs* > *0* **and** *i* < *length xs*
shows *eval r-rev-aux* [*i*, *e*] ↓= *tuple-encode i* (*rev* (*take* (*Suc i*) *xs*))
using *assms*(*3*)
proof (*induction i*)
case *0*
then show *?case*
unfolding *r-rev-aux-def* **using** *assms e-hd-def r-hd* **by** (*auto simp add*: *take-Suc*)
next
case (*Suc i*)
let *?g* = *Cn 3 r-prod-encode* [*Cn 3 r-nth* [*Id 3 2*, *Cn 3 S* [*Id 3 0*]], *Id 3 1*]
from *Suc* **have** *eval r-rev-aux* [*Suc i*, *e*] = *eval ?g* [*i*, *the* (*eval r-rev-aux* [*i*, *e*]), *e*]

```
    unfolding r-rev-aux-def by simp
  also have ... ↓= prod-encode (xs ! (Suc i), tuple-encode i (rev (take (Suc i) xs)))
    using Suc by (simp add: assms(1))
  finally show ?case by (simp add: Suc.prems take-Suc-conv-app-nth)
qed

corollary r-rev-aux-full:
  assumes list-decode e = xs and length xs > 0
  shows eval r-rev-aux [length xs − 1, e] ↓= tuple-encode (length xs − 1) (rev xs)
  using r-rev-aux assms by simp

lemma r-rev-aux-total: eval r-rev-aux [i, e] ↓
  using r-rev-aux-prim totalE by fastforce

definition r-rev ≡
  Cn 1 r-ifz
   [Id 1 0,
    Z,
    Cn 1 S
     [Cn 1 r-prod-encode
       [Cn 1 r-dec [r-length], Cn 1 r-rev-aux [Cn 1 r-dec [r-length], Id 1 0]]]]

lemma r-rev-prim [simp]: prim-recfn 1 r-rev
  unfolding r-rev-def using r-rev-aux-prim by simp

lemma r-rev [simp]: eval r-rev [e] ↓= list-encode (rev (list-decode e))
proof −
  let ?d = Cn 1 r-dec [r-length]
  let ?a = Cn 1 r-rev-aux [?d, Id 1 0]
  let ?p = Cn 1 r-prod-encode [?d, ?a]
  let ?s = Cn 1 S [?p]
  have eval-a: eval ?a [e] = eval r-rev-aux [e-length e − 1, e]
    using r-rev-aux-prim by simp
  then have eval ?s [e] ↓
    using r-rev-aux-prim by (simp add: r-rev-aux-total)
  then have ∗: eval r-rev [e] ↓= (if e = 0 then 0 else the (eval ?s [e]))
    using r-rev-aux-prim by (simp add: r-rev-def)
  show ?thesis
  proof (cases e = 0)
    case True
    then show ?thesis using ∗ by simp
  next
    case False
    then obtain xs where xs: xs = list-decode e length xs > 0
      using e-length-0 by auto
    then have len: length xs = e-length e by simp
    with eval-a have eval ?a [e] = eval r-rev-aux [length xs − 1, e]
      by simp
    then have eval ?a [e] ↓= tuple-encode (length xs − 1) (rev xs)
      using xs r-rev-aux-full by simp
    then have eval ?s [e] ↓=
        Suc (prod-encode (length xs − 1, tuple-encode (length xs − 1) (rev xs)))
      using len r-rev-aux-prim by simp
    then have eval ?s [e] ↓=
        Suc (prod-encode
              (length (rev xs) − 1, tuple-encode (length (rev xs) − 1) (rev xs)))
```

```
      by simp
    moreover have length (rev xs) > 0
      using xs by simp
    ultimately have eval ?s [e] ↓= list-encode (rev xs)
      by (metis list-encode.elims diff-Suc-1 length-Cons length-greater-0-conv)
    then show ?thesis using xs * by simp
  qed
qed

abbreviation e-cons :: nat ⇒ nat ⇒ nat where
  e-cons e es ≡ list-encode (e # list-decode es)

lemma e-cons-elementary:
  e-cons e es =
    (if es = 0 then Suc (prod-encode (0, e))
     else Suc (prod-encode (e-length es, prod-encode (e, pdec2 (es − 1)))))
proof (cases es = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have e-length es = Suc (pdec1 (es − 1))
    by (metis list-decode.elims diff-Suc-1 tuple-decode-length)
  moreover have es = e-tl (list-encode (e # list-decode es))
    by (metis list.sel(3) list-decode-encode list-encode-decode)
  ultimately show ?thesis
    using False e-tl-elementary
    by (metis list-decode.simps(2) diff-Suc-1 list-encode-decode prod.sel(1)
      prod-encode-inverse snd-conv tuple-decode.simps(2))
qed

definition r-cons-else ≡
  Cn 2 S
   [Cn 2 r-prod-encode
     [Cn 2 r-length
       [Id 2 1], Cn 2 r-prod-encode [Id 2 0, Cn 2 r-pdec2 [Cn 2 r-dec [Id 2 1]]]]]

lemma r-cons-else-prim: prim-recfn 2 r-cons-else
  unfolding r-cons-else-def by simp

lemma r-cons-else:
  eval r-cons-else [e, es] ↓=
    Suc (prod-encode (e-length es, prod-encode (e, pdec2 (es − 1))))
  unfolding r-cons-else-def by simp

definition r-cons ≡
  Cn 2 r-ifz
    [Id 2 1, Cn 2 S [Cn 2 r-prod-encode [r-constn 1 0, Id 2 0]], r-cons-else]

lemma r-cons-prim [simp]: prim-recfn 2 r-cons
  unfolding r-cons-def using r-cons-else-prim by simp

lemma r-cons [simp]: eval r-cons [e, es] ↓= e-cons e es
  unfolding r-cons-def using r-cons-else-prim r-cons-else e-cons-elementary by simp

abbreviation e-snoc :: nat ⇒ nat ⇒ nat where
```

```
  e-snoc es e ≡ list-encode (list-decode es @ [e])

lemma e-nth-snoc-small [simp]:
  assumes n < e-length b
  shows e-nth (e-snoc b z) n = e-nth b n
  using assms by (simp add: nth-append)

lemma e-hd-snoc [simp]:
  assumes e-length b > 0
  shows e-hd (e-snoc b x) = e-hd b
proof −
  from assms have b ≠ 0
    using less-imp-neq by force
  then have hd: e-hd b = hd (list-decode b) by simp
  have e-length (e-snoc b x) > 0 by simp
  then have e-snoc b x ≠ 0
    using not-gr-zero by fastforce
  then have e-hd (e-snoc b x) = hd (list-decode (e-snoc b x)) by simp
  with assms hd show ?thesis by simp
qed

definition r-snoc ≡ Cn 2 r-rev [Cn 2 r-cons [Id 2 1, Cn 2 r-rev [Id 2 0]]]

lemma r-snoc-prim [simp]: prim-recfn 2 r-snoc
  unfolding r-snoc-def by simp

lemma r-snoc [simp]: eval r-snoc [es, e] ↓= e-snoc es e
  unfolding r-snoc-def by simp

abbreviation e-butlast :: nat ⇒ nat where
  e-butlast e ≡ list-encode (butlast (list-decode e))

abbreviation e-take :: nat ⇒ nat ⇒ nat where
  e-take n x ≡ list-encode (take n (list-decode x))

definition r-take ≡
  Cn 2 r-ifle
   [Id 2 0, Cn 2 r-length [Id 2 1],
    Pr 1 Z (Cn 3 r-snoc [Id 3 1, Cn 3 r-nth [Id 3 2, Id 3 0]]),
    Id 2 1]

lemma r-take-prim [simp]: prim-recfn 2 r-take
  unfolding r-take-def by simp-all

lemma r-take:
  assumes x = list-encode es
  shows eval r-take [n, x] ↓= list-encode (take n es)
proof −
  let ?g = Cn 3 r-snoc [Id 3 1, Cn 3 r-nth [Id 3 2, Id 3 0]]
  let ?h = Pr 1 Z ?g
  have total ?h using Mn-free-imp-total by simp
  have m ≤ length es ⟹ eval ?h [m, x] ↓= list-encode (take m es) for m
  proof (induction m)
    case 0
    then show ?case using assms r-take-def by (simp add: r-take-def)
  next
```

case (*Suc m*)
then have *m* < *length es* **by** *simp*
then have *eval ?h* [*Suc m*, *x*] = *eval ?g* [*m*, *the* (*eval ?h* [*m*, *x*]), *x*]
using *Suc r-take-def* **by** *simp*
also have ... = *eval ?g* [*m*, *list-encode* (*take m es*), *x*]
using *Suc* **by** *simp*
also have ... ↓= *e-snoc* (*list-encode* (*take m es*)) (*es* ! *m*)
by (*simp add*: ‹*m* < *length es*› *assms*)
also have ... ↓= *list-encode* ((*take m es*) @ [*es* ! *m*])
using *list-decode-encode* **by** *simp*
also have ... ↓= *list-encode* (*take* (*Suc m*) *es*)
by (*simp add*: ‹*m* < *length es*› *take-Suc-conv-app-nth*)
finally show *?case* **.**
qed
moreover have *eval* (*Id 2 1*) [*m*, *x*] ↓= *list-encode* (*take m es*) **if** *m* > *length es* **for** *m*
using *that assms* **by** *simp*
moreover have *eval r-take* [*m*, *x*] ↓=
(*if m* ≤ *e-length x then the* (*eval ?h* [*m*, *x*]) *else the* (*eval* (*Id 2 1*) [*m*, *x*]))
for *m*
unfolding *r-take-def* **using** ‹*total ?h*› **by** *simp*
ultimately show *?thesis* **unfolding** *r-take-def* **by** *fastforce*
qed

corollary *r-take′* [*simp*]: *eval r-take* [*n*, *x*] ↓= *e-take n x*
by (*simp add*: *r-take*)

definition *r-last* ≡ *Cn 1 r-hd* [*r-rev*]

lemma *r-last-prim* [*simp*]: *prim-recfn 1 r-last*
unfolding *r-last-def* **by** *simp*

lemma *r-last* [*simp*]:
assumes *e* = *list-encode xs* **and** *length xs* > *0*
shows *eval r-last* [*e*] ↓= *last xs*
proof −
from *assms*(*2*) **have** *length* (*rev xs*) > *0* **by** *simp*
then have *list-encode* (*rev xs*) > *0*
by (*metis gr0I list.size*(*3*) *list-encode-0*)
moreover have *eval r-last* [*e*] = *eval r-hd* [*the* (*eval r-rev* [*e*])]
unfolding *r-last-def* **by** *simp*
ultimately show *?thesis* **using** *assms hd-rev* **by** *auto*
qed

definition *r-update-aux* ≡
let
f = *r-constn 2 0*;
g = *Cn 5 r-snoc*
[*Id 5 1*, *Cn 5 r-ifeq* [*Id 5 0*, *Id 5 3*, *Id 5 4*, *Cn 5 r-nth* [*Id 5 2*, *Id 5 0*]]]
in Pr 3 f g

lemma *r-update-aux-recfn*: *recfn 4 r-update-aux*
unfolding *r-update-aux-def* **by** *simp*

lemma *r-update-aux*:
assumes *n* ≤ *e-length b*
shows *eval r-update-aux* [*n*, *b*, *j*, *v*] ↓= *list-encode* ((*take n* (*list-decode b*))[*j*:=*v*])

using *assms*
proof (*induction n*)
case *0*
then show *?case* unfolding *r-update-aux-def* by *simp*
next
case (*Suc n*)
then have *n*: *n* < *e-length b*
by *simp*
let *?a* = *Cn 5 r-nth* [*Id 5 2*, *Id 5 0*]
let *?b* = *Cn 5 r-ifeq* [*Id 5 0*, *Id 5 3*, *Id 5 4*, *?a*]
define *g* where *g* ≡ *Cn 5 r-snoc* [*Id 5 1*, *?b*]
then have *g*: *eval g* [*n*, *r*, *b*, *j*, *v*] ↓= *e-snoc r* (*if n* = *j then v else e-nth b n*) for *r*
by *simp*

have *Pr 3* (*r-constn 2 0*) *g* = *r-update-aux*
using *r-update-aux-def g-def* by *simp*
then have *eval r-update-aux* [*Suc n*, *b*, *j*, *v*] =
eval g [*n*, *the* (*eval r-update-aux* [*n*, *b*, *j*, *v*]), *b*, *j*, *v*]
using *r-update-aux-recfn Suc n eval-Pr-converg-Suc*
by (*metis arity.simps*(*5*) *length-Cons list.size*(*3*) *nat-less-le*
numeral-3-eq-3 option.simps(*3*))
then have ∗: *eval r-update-aux* [*Suc n*, *b*, *j*, *v*] ↓= *e-snoc*
(*list-encode* ((*take n* (*list-decode b*))[*j*:=*v*]))
(*if n* = *j then v else e-nth b n*)
using *g Suc* by *simp*

consider (*j-eq-n*) *j* = *n* | (*j-less-n*) *j* < *n* | (*j-gt-n*) *j* > *n*
by *linarith*
then show *?case*
proof (*cases*)
case *j-eq-n*
moreover from *this* have (*take* (*Suc n*) (*list-decode b*))[*j*:=*v*] =
(*take n* (*list-decode b*))[*j*:=*v*] @ [*v*]
using *n*
by (*metis length-list-update nth-list-update-eq take-Suc-conv-app-nth take-update-swap*)
ultimately show *?thesis* using ∗ by *simp*
next
case *j-less-n*
moreover from *this* have (*take* (*Suc n*) (*list-decode b*))[*j*:=*v*] =
(*take n* (*list-decode b*))[*j*:=*v*] @ [(*list-decode b*) ! *n*]
using *n*
by (*simp add*: *le-eq-less-or-eq list-update-append min-absorb2 take-Suc-conv-app-nth*)
ultimately show *?thesis* using ∗ by *auto*
next
case *j-gt-n*
moreover from *this* have (*take* (*Suc n*) (*list-decode b*))[*j*:=*v*] =
(*take n* (*list-decode b*))[*j*:=*v*] @ [(*list-decode b*) ! *n*]
using *n take-Suc-conv-app-nth* by *auto*
ultimately show *?thesis* using ∗ by *auto*
qed
qed

abbreviation *e-update* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
e-update b j v ≡ *list-encode* ((*list-decode b*)[*j*:=*v*])

definition *r-update* ≡

Cn 3 r-update-aux [*Cn 3 r-length* [*Id 3 0*], *Id 3 0*, *Id 3 1*, *Id 3 2*]

lemma *r-update-recfn* [*simp*]: *recfn 3 r-update*
unfolding *r-update-def* **using** *r-update-aux-recfn* **by** *simp*

lemma *r-update* [*simp*]: *eval r-update* [*b*, *j*, *v*] ↓= *e-update b j v*
unfolding *r-update-def* **using** *r-update-aux r-update-aux-recfn* **by** *simp*

lemma *e-length-update* [*simp*]: *e-length* (*e-update b k v*) = *e-length b*
by *simp*

definition *e-append* :: *nat* ⇒ *nat* ⇒ *nat* **where**
e-append xs ys ≡ *list-encode* (*list-decode xs* @ *list-decode ys*)

lemma *e-length-append*: *e-length* (*e-append xs ys*) = *e-length xs* + *e-length ys*
using *e-append-def* **by** *simp*

lemma *e-nth-append-small*:
assumes *n* < *e-length xs*
shows *e-nth* (*e-append xs ys*) *n* = *e-nth xs n*
using *e-append-def assms* **by** (*simp add*: *nth-append*)

lemma *e-nth-append-big*:
assumes *n* ≥ *e-length xs*
shows *e-nth* (*e-append xs ys*) *n* = *e-nth ys* (*n* − *e-length xs*)
using *e-append-def assms e-nth* **by** (*simp add*: *less-diff-conv2 nth-append*)

definition *r-append* ≡
let
f = *Id 2 0*;
g = *Cn 4 r-snoc* [*Id 4 1*, *Cn 4 r-nth* [*Id 4 3*, *Id 4 0*]]
in Cn 2 (*Pr 2 f g*) [*Cn 2 r-length* [*Id 2 1*], *Id 2 0*, *Id 2 1*]

lemma *r-append-prim* [*simp*]: *prim-recfn 2 r-append*
unfolding *r-append-def* **by** *simp*

lemma *r-append* [*simp*]: *eval r-append* [*a*, *b*] ↓= *e-append a b*
proof −
define *g* **where** *g* = *Cn 4 r-snoc* [*Id 4 1*, *Cn 4 r-nth* [*Id 4 3*, *Id 4 0*]]
then have *g*: *eval g* [*j*, *r*, *a*, *b*] ↓= *e-snoc r* (*e-nth b j*) **for** *j r*
by *simp*
let *?h* = *Pr 2* (*Id 2 0*) *g*
have *eval ?h* [*n*, *a*, *b*] ↓= *list-encode* (*list-decode a* @ (*take n* (*list-decode b*)))
if *n* ≤ *e-length b* **for** *n*
using *that g g-def* **by** (*induction n*) (*simp-all add*: *take-Suc-conv-app-nth*)
then show *?thesis*
unfolding *r-append-def g-def e-append-def* **by** *simp*
qed

definition *e-append-zeros* :: *nat* ⇒ *nat* ⇒ *nat* **where**
e-append-zeros b z ≡ *e-append b* (*list-encode* (*replicate z 0*))

lemma *e-append-zeros-length*: *e-length* (*e-append-zeros b z*) = *e-length b* + *z*
using *e-append-def e-append-zeros-def* **by** *simp*

lemma *e-nth-append-zeros*: *e-nth* (*e-append-zeros b z*) *i* = *e-nth b i*

using *e-append-zeros-def e-nth-append-small e-nth-append-big* **by** *auto*

lemma *e-nth-append-zeros-big*:
assumes $i \geq$ *e-length b*
shows *e-nth* (*e-append-zeros b z*) $i = 0$
unfolding *e-append-zeros-def*
using *e-nth-append-big*[*of b i list-encode* (*replicate z 0*), *OF assms*(*1*)]
by *simp*

definition *r-append-zeros* ≡
r-swap (*Pr 1* (*Id 1 0*) (*Cn 3 r-snoc* [*Id 3 1*, *r-constn 2 0*]))

lemma *r-append-zeros-prim* [*simp*]: *prim-recfn 2 r-append-zeros*
unfolding *r-append-zeros-def* **by** *simp*

lemma *r-append-zeros*: *eval r-append-zeros* [*b*, *z*] ↓= *e-append-zeros b z*
proof −
let *?r* = *Pr 1* (*Id 1 0*) (*Cn 3 r-snoc* [*Id 3 1*, *r-constn 2 0*])
have *eval ?r* [*z*, *b*] ↓= *e-append-zeros b z*
using *e-append-zeros-def e-append-def*
by (*induction z*) (*simp-all add*: *replicate-append-same*)
then show *?thesis* **by** (*simp add*: *r-append-zeros-def*)
qed

end

1.5 A universal partial recursive function

theory *Universal*
imports *Partial-Recursive*
begin

The main product of this section is a universal partial recursive function, which given a code i of an n-ary partial recursive function f and an encoded list xs of n arguments, computes *eval f xs*. From this we can derive fixed-arity universal functions satisfying the usual results such as the s-m-n theorem. To represent the code i, we need a way to encode *recf*s as natural numbers (Section 1.5.2). To construct the universal function, we devise a ternary function taking i, xs, and a step bound t and simulating the execution of f on input xs for t steps. This function is useful in its own right, enabling techniques like dovetailing or "concurrent" evaluation of partial recursive functions.

The notion of a "step" is not part of the definition of (the evaluation of) partial recursive functions, but one can simulate the evaluation on an abstract machine (Section 1.5.1). This machine's configurations can be encoded as natural numbers, and this leads us to a step function *nat* ⇒ *nat* on encoded configurations (Section 1.5.3). This function in turn can be computed by a primitive recursive function, from which we develop the aforementioned ternary function of i, xs, and t (Section 1.5.4). From this we can finally derive a universal function (Section 1.5.5).

1.5.1 A step function

We simulate the stepwise execution of a partial recursive function in a fairly straightforward way reminiscent of the execution of function calls in an imperative programming language. A configuration of the abstract machine is a pair consisting of:

1. A stack of frames. A frame represents the execution of a function and is a triple $(f, xs, locals)$ of
   (a) a *recf* f being executed,
   (b) a *nat list* of arguments of f,
   (c) a *nat list* of local variables, which holds intermediate values when f is of the form *Cn*, *Pr*, or *Mn*.
2. A register of type *nat option* representing the return value of the last function call: *None* signals that in the previous step the stack was not popped and hence no value was returned, whereas *Some* v means that in the previous step a function returned v.

For computing h on input xs, the initial configuration is $([(h, xs, [])], None)$. When the computation for a frame ends, it is popped off the stack, and its return value is put in the register. The entire computation ends when the stack is empty. In such a final configuration the register contains the value of h at xs. If no final configuration is ever reached, h diverges at xs.

The execution of one step depends on the topmost (that is, active) frame. In the step when a frame $(h, xs, locals)$ is pushed onto the stack, the local variables are $locals = []$. The following happens until the frame is popped off the stack again (if it ever is):

- For the base functions $h = Z$, $h = S$, $h = Id\ m\ n$, the frame is popped off the stack right away, and the return value is placed in the register.
- For $h = Cn\ n\ f\ gs$, for each function g in gs:
  1. A new frame of the form $(g, xs, [])$ is pushed onto the stack.
  2. When (and if) this frame is eventually popped, the value in the register is *eval* g xs. This value is appended to the list *locals* of local variables.

  When all g in gs have been evaluated in this manner, f is evaluated on the local variables by pushing $(f, locals, [])$. The resulting register value is kept and the active frame for h is popped off the stack.
- For $h = Pr\ n\ f\ g$, let $xs = y\ \#\ ys$. First $(f, ys, [])$ is pushed and the return value stored in the *locals*. Then $(g, x\ \#\ v\ \#\ ys, [])$ is pushed, where x is the length of *locals* and v the most recently appended value. The return value is appended to *locals*. This is repeated until the length of *locals* reaches y. Then the most recently appended local is placed in the register, and the stack is popped.
- For $h = Mn\ n\ f$, frames $(f, x\ \#\ xs, [])$ are pushed for $x = 0, 1, 2, \ldots$ until one of them returns 0. Then this x is placed in the register and the stack is popped. Until then x is stored in *locals*. If none of these evaluations return 0, the stack never shrinks, and thus the machine never reaches a final state.

type-synonym *frame* $=$ *recf* $\times$ *nat list* $\times$ *nat list*

type-synonym *configuration* $=$ *frame list* $\times$ *nat option*

Definition of the step function

```
fun step :: configuration ⇒ configuration where
  step ([], rv) = ([], rv)
| step (((Z, -, -) # fs), rv) = (fs, Some 0)
| step (((S, xs, -) # fs), rv) = (fs, Some (Suc (hd xs)))
| step (((Id m n, xs, -) # fs), rv) = (fs, Some (xs ! n))
| step (((Cn n f gs, xs, ls) # fs), rv) =
    (if length ls = length gs
     then if rv = None
          then ((f, ls, []) # (Cn n f gs, xs, ls) # fs, None)
          else (fs, rv)
     else if rv = None
          then if length ls < length gs
               then ((gs ! (length ls), xs, []) # (Cn n f gs, xs, ls) # fs, None)
               else (fs, rv)   — cannot occur, so don't-care term
          else ((Cn n f gs, xs, ls @ [the rv]) # fs, None))
| step (((Pr n f g, xs, ls) # fs), rv) =
    (if ls = []
     then if rv = None
          then ((f, tl xs, []) # (Pr n f g, xs, ls) # fs, None)
          else ((Pr n f g, xs, [the rv]) # fs, None)
     else if length ls = Suc (hd xs)
          then (fs, Some (hd ls))
          else if rv = None
               then ((g, (length ls − 1) # hd ls # tl xs, []) # (Pr n f g, xs, ls) # fs, None)
               else ((Pr n f g, xs, (the rv) # ls) # fs, None))
| step (((Mn n f, xs, ls) # fs), rv) =
    (if ls = []
     then ((f, 0 # xs, []) # (Mn n f, xs, [0]) # fs, None)
     else if rv = Some 0
          then (fs, Some (hd ls))
          else ((f, (Suc (hd ls)) # xs, []) # (Mn n f, xs, [Suc (hd ls)]) # fs, None))

definition reachable :: configuration ⇒ configuration ⇒ bool where
  reachable x y ≡ ∃ t. iterate t step x = y

lemma step-reachable [intro]:
  assumes step x = y
  shows reachable x y
  unfolding reachable-def using assms by (metis iterate.simps(1,2) comp-id)

lemma reachable-transitive [trans]:
  assumes reachable x y and reachable y z
  shows reachable x z
  using assms iterate-additive[where ?f=step] reachable-def by metis

lemma reachable-refl: reachable x x
  unfolding reachable-def by (metis iterate.simps(1) eq-id-iff)
```

From a final configuration, that is, when the stack is empty, only final configurations are reachable.

```
lemma step-empty-stack:
  assumes fst x = []
  shows fst (step x) = []
  using assms by (metis prod.collapse step.simps(1))
```

lemma *reachable-empty-stack*:
  assumes *fst x* = [] **and** *reachable x y*
  shows *fst y* = []
proof −
  have *fst* (*iterate t step x*) = [] **for** *t*
    using *assms step-empty-stack* **by** (*induction t*) *simp-all*
  then show *?thesis*
    using *reachable-def assms*(*2*) **by** *auto*
qed

abbreviation *nonterminating* :: *configuration* ⇒ *bool* **where**
  nonterminating x ≡ ∀ *t*. *fst* (*iterate t step x*) ≠ []

lemma *reachable-nonterminating*:
  assumes *reachable x y* **and** *nonterminating y*
  shows *nonterminating x*
proof −
  from *assms*(*1*) **obtain** t_1 **where** *t1*: *iterate* t_1 *step x* = *y*
    using *reachable-def* **by** *auto*
  have *fst* (*iterate t step x*) ≠ [] **for** *t*
  proof (*cases t* ≤ t_1)
    case *True*
    then show *?thesis*
      using *t1 assms*(*2*) *reachable-def reachable-empty-stack iterate-additive′*
      by (*metis le-Suc-ex*)
  next
    case *False*
    then have *iterate t step x* = *iterate* (t_1 + (*t* − t_1)) *step x*
      by *simp*
    then have *iterate t step x* = *iterate* (*t* − t_1) *step* (*iterate* t_1 *step x*)
      by (*simp add*: *iterate-additive′*)
    then have *iterate t step x* = *iterate* (*t* − t_1) *step y*
      using *t1* **by** *simp*
    then show *fst* (*iterate t step x*) ≠ []
      using *assms*(*2*) **by** *simp*
  qed
  then show *?thesis* **..**
qed

The function *step* is underdefined, for example, when the top frame contains a non-well-formed *recf* or too few arguments. All is well, though, if every frame contains a well-formed *recf* whose arity matches the number of arguments. Such stacks will be called *valid*.

definition *valid* :: *frame list* ⇒ *bool* **where**
  valid stack ≡ ∀ *s*∈*set stack*. *recfn* (*length* (*fst* (*snd s*))) (*fst s*)

lemma *valid-frame*: *valid* (*s* # *ss*) ⟹ *valid ss* ∧ *recfn* (*length* (*fst* (*snd s*))) (*fst s*)
  using *valid-def* **by** *simp*

lemma *valid-ConsE*: *valid* ((*f*, *xs*, *locs*) # *rest*) ⟹ *valid rest* ∧ *recfn* (*length xs*) *f*
  using *valid-def* **by** *simp*

lemma *valid-ConsI*: *valid rest* ⟹ *recfn* (*length xs*) *f* ⟹ *valid* ((*f*, *xs*, *locs*) # *rest*)
  using *valid-def* **by** *simp*

Stacks in initial configurations are valid, and performing a step maintains the validity of the stack.

```
lemma step-valid: valid stack ⟹ valid (fst (step (stack, rv)))
proof (cases stack)
  case Nil
  then show ?thesis using valid-def by simp
next
  case (Cons s ss)
  assume valid: valid stack
  then have *: valid ss ∧ recfn (length (fst (snd s))) (fst s)
    using valid-frame Cons by simp
  show ?thesis
  proof (cases fst s)
    case Z
    then show ?thesis using Cons valid * by (metis fstI prod.collapse step.simps(2))
  next
    case S
    then show ?thesis using Cons valid * by (metis fst-conv prod.collapse step.simps(3))
  next
    case Id
    then show ?thesis using Cons valid * by (metis fstI prod.collapse step.simps(4))
  next
    case (Cn n f gs)
    then obtain xs ls where s = (Cn n f gs, xs, ls)
      using Cons by (metis prod.collapse)
    moreover consider
        length ls = length gs ∧ rv ↑
      | length ls = length gs ∧ rv ↓
      | length ls < length gs ∧ rv ↑
      | length ls ≠ length gs ∧ rv ↓
      | length ls > length gs ∧ rv ↑
      by linarith
    ultimately show ?thesis using valid Cons valid-def by (cases) auto
  next
    case (Pr n f g)
    then obtain xs ls where s: s = (Pr n f g, xs, ls)
      using Cons by (metis prod.collapse)
    consider
        length ls = 0 ∧ rv ↑
      | length ls = 0 ∧ rv ↓
      | length ls ≠ 0 ∧ length ls = Suc (hd xs)
      | length ls ≠ 0 ∧ length ls ≠ Suc (hd xs) ∧ rv ↑
      | length ls ≠ 0 ∧ length ls ≠ Suc (hd xs) ∧ rv ↓
      by linarith
    then show ?thesis using Cons * valid-def s by (cases) auto
  next
    case (Mn n f)
    then obtain xs ls where s: s = (Mn n f, xs, ls)
      using Cons by (metis prod.collapse)
    consider
        length ls = 0
      | length ls ≠ 0 ∧ rv ↑
      | length ls ≠ 0 ∧ rv ↓
      by linarith
    then show ?thesis using Cons * valid-def s by (cases) auto
```

qed
qed

corollary *iterate-step-valid*:
assumes *valid stack*
shows *valid* (*fst* (*iterate t step* (*stack*, *rv*)))
using *assms*
proof (*induction t*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc t*)
moreover have *iterate* (*Suc t*) *step* (*stack*, *rv*) = *step* (*iterate t step* (*stack*, *rv*))
by *simp*
ultimately show *?case* **using** *step-valid valid-def* **by** (*metis prod.collapse*)
qed

Correctness of the step function

The function *step* works correctly for a *recf f* on arguments *xs* in some configuration if (1) in case *f* converges, *step* reaches a configuration with the topmost frame popped and *eval f xs* in the register, and (2) in case *f* diverges, *step* does not reach a final configuration.

fun *correct* :: *configuration* ⇒ *bool* **where**
correct ([], *r*) = *True*
| *correct* ((*f*, *xs*, *ls*) # *rest*, *r*) =
(*if eval f xs* ↓ *then reachable* ((*f*, *xs*, *ls*) # *rest*, *r*) (*rest*, *eval f xs*)
else nonterminating ((*f*, *xs*, *ls*) # *rest*, *None*))

lemma *correct-convergI*:
assumes *eval f xs* ↓ **and** *reachable* ((*f*, *xs*, *ls*) # *rest*, *None*) (*rest*, *eval f xs*)
shows *correct* ((*f*, *xs*, *ls*) # *rest*, *None*)
using *assms* **by** *auto*

lemma *correct-convergE*:
assumes *correct* ((*f*, *xs*, *ls*) # *rest*, *None*) **and** *eval f xs* ↓
shows *reachable* ((*f*, *xs*, *ls*) # *rest*, *None*) (*rest*, *eval f xs*)
using *assms* **by** *simp*

The correctness proof for *step* is by structural induction on the *recf* in the top frame. The base cases *Z*, *S*, and *Id* are simple. For *X* = *Cn*, *Pr*, *Mn*, the lemmas named *reachable-X* show which configurations are reachable for *recf*s of shape *X*. Building on those, the lemmas named *step-X-correct* show *step*'s correctness for *X*.

lemma *reachable-Cn*:
assumes *valid* (((*Cn n f gs*), *xs*, []) # *rest*) (**is** *valid ?stack*)
and ⋀*xs rest. valid* ((*f*, *xs*, []) # *rest*) ⟹ *correct* ((*f*, *xs*, []) # *rest*, *None*)
and ⋀*g xs rest.*
g ∈ *set gs* ⟹ *valid* ((*g*, *xs*, []) # *rest*) ⟹ *correct* ((*g*, *xs*, []) # *rest*, *None*)
and ∀ *i*<*k*. *eval* (*gs* ! *i*) *xs* ↓
and *k* ≤ *length gs*
shows *reachable*
(*?stack*, *None*)
((*Cn n f gs*, *xs*, *take k* (*map* (λ*g*. *the* (*eval g xs*)) *gs*)) # *rest*, *None*)
using *assms*(*4*,*5*)

```
proof (induction k)
  case 0
  then show ?case using reachable-refl by simp
next
  case (Suc k)
  let ?ys = map (λg. the (eval g xs)) gs
  from Suc have k < length gs by simp
  have valid: recfn (length xs) (Cn n f gs) valid rest
    using assms(1) valid-ConsE[of (Cn n f gs)] by simp-all
  from Suc have reachable (?stack, None) ((Cn n f gs, xs, take k ?ys) # rest, None)
      (is - (?stack1, None))
    by simp
  also have reachable ... ((gs ! k, xs, []) # ?stack1, None)
    using step-reachable ‹k < length gs›
    by (auto simp: min-absorb2)
  also have reachable ... (?stack1, eval (gs ! k) xs)
      (is - (-, ?rv))
    using Suc.prems(1) ‹k < length gs› assms(3) valid valid-ConsI by auto
  also have reachable ... ((Cn n f gs, xs, (take (Suc k) ?ys)) # rest, None)
      (is - (?stack2, None))
  proof -
    have step (?stack1, ?rv) = ((Cn n f gs, xs, (take k ?ys) @ [the ?rv]) # rest, None)
      using Suc by auto
    also have ... = ((Cn n f gs, xs, (take (Suc k) ?ys)) # rest, None)
      by (simp add: ‹k < length gs› take-Suc-conv-app-nth)
    finally show ?thesis
      using step-reachable by auto
  qed
  finally show reachable (?stack, None) (?stack2, None) .
qed

lemma step-Cn-correct:
  assumes valid (((Cn n f gs), xs, []) # rest) (is valid ?stack)
    and ⋀xs rest. valid ((f, xs, []) # rest) ⟹ correct ((f, xs, []) # rest, None)
    and ⋀g xs rest.
      g ∈ set gs ⟹ valid ((g, xs, []) # rest) ⟹ correct ((g, xs, []) # rest, None)
  shows correct (?stack, None)
proof -
  have valid: recfn (length xs) (Cn n f gs) valid rest
    using valid-ConsE[OF assms(1)] by auto
  let ?ys = map (λg. the (eval g xs)) gs
  consider
      (diverg-f) ∀g∈set gs. eval g xs ↓ and eval f ?ys ↑
    | (diverg-gs) ∃g∈set gs. eval g xs ↑
    | (converg) eval (Cn n f gs) xs ↓
    using valid-ConsE[OF assms(1)] by fastforce
  then show ?thesis
  proof (cases)
    case diverg-f
    then have ∀i<length gs. eval (gs ! i) xs ↓ by simp
    then have reachable (?stack, None) ((Cn n f gs, xs, ?ys) # rest, None)
        (is - (?stack1, None))
      using reachable-Cn[OF assms, where ?k=length gs] by simp
    also have reachable ... ((f, ?ys, []) # ?stack1, None) (is - (?stack2, None))
      by (simp add: step-reachable)
    finally have reachable (?stack, None) (?stack2, None) .
```

```
  moreover have nonterminating (?stack2, None)
    using diverg-f(2) assms(2)[of ?ys ?stack1] valid-ConsE[OF assms(1)] valid-ConsI
    by auto
  ultimately have nonterminating (?stack, None)
    using reachable-nonterminating by simp
  moreover have eval (Cn n f gs) xs ↑
    using diverg-f(2) assms(1) eval-Cn valid-ConsE by presburger
  ultimately show ?thesis by simp
next
  case diverg-gs
  then have ex-i: ∃ i<length gs. eval (gs ! i) xs ↑
    using in-set-conv-nth[of - gs] by auto
  define k where k = (LEAST i. i < length gs ∧ eval (gs ! i) xs ↑) (is - = Least ?P)
  then have gs-k: eval (gs ! k) xs ↑
    using LeastI-ex[OF ex-i] by simp
  have ∀ i<k. eval (gs ! i) xs ↓
    using k-def not-less-Least[of - ?P] LeastI-ex[OF ex-i] by simp
  moreover from this have k < length gs
    using ex-i less-le-trans not-le by blast
  ultimately have reachable (?stack, None) ((Cn n f gs, xs, take k ?ys) # rest, None)
    using reachable-Cn[OF assms] by simp
  also have reachable ...
    ((gs ! (length (take k ?ys)), xs, []) # (Cn n f gs, xs, take k ?ys) # rest, None)
    (is - (?stack1, None))
  proof −
    have length (take k ?ys) < length gs
      by (simp add: ‹k < length gs› less-imp-le-nat min-less-iff-disj)
    then show ?thesis using step-reachable ‹k < length gs›
      by auto
  qed
  finally have reachable (?stack, None) (?stack1, None) .
  moreover have nonterminating (?stack1, None)
  proof −
    have recfn (length xs) (gs ! k)
      using ‹k < length gs› valid(1) by simp
    then have correct (?stack1, None)
      using ‹k < length gs› nth-mem valid valid-ConsI
        assms(3)[of gs ! (length (take k ?ys)) xs]
      by auto
    moreover have length (take k ?ys) = k
      by (simp add: ‹k < length gs› less-imp-le-nat min-absorb2)
    ultimately show ?thesis using gs-k by simp
  qed
  ultimately have nonterminating (?stack, None)
    using reachable-nonterminating by simp
  moreover have eval (Cn n f gs) xs ↑
    using diverg-gs valid by fastforce
  ultimately show ?thesis by simp
next
  case converg
  then have f: eval f ?ys ↓ and g: ⋀g. g ∈ set gs ⟹ eval g xs ↓
    using valid(1) by (metis eval-Cn)+
  then have ∀ i<length gs. eval (gs ! i) xs ↓
    by simp
  then have reachable (?stack, None) ((Cn n f gs, xs, take (length gs) ?ys) # rest, None)
    using reachable-Cn assms by blast
```

also have *reachable ...* ((*Cn n f gs*, *xs*, *?ys*) # *rest*, *None*) (**is** - (*?stack1*, *None*))
by (*simp add*: *reachable-refl*)
also have *reachable ...* ((*f*, *?ys*, []) # *?stack1*, *None*)
using *step-reachable* **by** *auto*
also have *reachable ...* (*?stack1*, *eval f ?ys*)
using *assms*(*2*)[*of ?ys*] *correct-convergE valid f valid-ConsI* **by** *auto*
also have *reachable* (*?stack1*, *eval f ?ys*) (*rest*, *eval f ?ys*)
using *f* **by** *auto*
finally have *reachable* (*?stack*, *None*) (*rest*, *eval f ?ys*) **.**
moreover have *eval* (*Cn n f gs*) *xs* = *eval f ?ys*
using *g valid*(*1*) **by** *auto*
ultimately show *?thesis*
using *converg correct-convergI* **by** *auto*
qed
qed

During the execution of a frame with a partial recursive function of shape *Pr n f g* and arguments *x* # *xs*, the list of local variables collects all the function values up to *x* in reversed order. We call such a list a *trace* for short.

definition *trace* :: *nat* ⇒ *recf* ⇒ *recf* ⇒ *nat list* ⇒ *nat* ⇒ *nat list* **where**
trace n f g xs x ≡ *map* (*λy. the* (*eval* (*Pr n f g*) (*y* # *xs*))) (*rev* [*0..<Suc x*])

lemma *trace-length*: *length* (*trace n f g xs x*) = *Suc x*
using *trace-def* **by** *simp*

lemma *trace-hd*: *hd* (*trace n f g xs x*) = *the* (*eval* (*Pr n f g*) (*x* # *xs*))
using *trace-def* **by** *simp*

lemma *trace-Suc*:
trace n f g xs (*Suc x*) = (*the* (*eval* (*Pr n f g*) (*Suc x* # *xs*))) # (*trace n f g xs x*)
using *trace-def* **by** *simp*

lemma *reachable-Pr*:
assumes *valid* (((*Pr n f g*), *x* # *xs*, []) # *rest*) (**is** *valid ?stack*)
and ⋀*xs rest. valid* ((*f*, *xs*, []) # *rest*) ⟹ *correct* ((*f*, *xs*, []) # *rest*, *None*)
and ⋀*xs rest. valid* ((*g*, *xs*, []) # *rest*) ⟹ *correct* ((*g*, *xs*, []) # *rest*, *None*)
and *y* ≤ *x*
and *eval* (*Pr n f g*) (*y* # *xs*) ↓
shows *reachable* (*?stack*, *None*) ((*Pr n f g*, *x* # *xs*, *trace n f g xs y*) # *rest*, *None*)
using *assms*(*4*,*5*)
proof (*induction y*)
case *0*
have *valid*: *recfn* (*length* (*x* # *xs*)) (*Pr n f g*) *valid rest*
using *valid-ConsE*[*OF assms*(*1*)] **by** *simp-all*
then have *f*: *eval f xs* ↓ **using** *0* **by** *simp*
let *?as* = *x* # *xs*
have *reachable* (*?stack*, *None*) ((*f*, *xs*, []) # ((*Pr n f g*), *?as*, []) # *rest*, *None*)
using *step-reachable* **by** *auto*
also have *reachable ...* (*?stack*, *eval f xs*)
using *assms*(*2*)[*of xs* ((*Pr n f g*), *?as*, []) # *rest*]
correct-convergE[*OF - f*] *f valid valid-ConsI*
by *simp*
also have *reachable ...* ((*Pr n f g*, *?as*, [*the* (*eval f xs*)]) # *rest*, *None*)
using *step-reachable valid*(*1*) *f* **by** *auto*
finally have *reachable* (*?stack*, *None*) ((*Pr n f g*, *?as*, [*the* (*eval f xs*)]) # *rest*, *None*) **.**
then show *?case* **using** *trace-def valid*(*1*) **by** *simp*

```
next
  case (Suc y)
  have valid: recfn (length (x # xs)) (Pr n f g) valid rest
    using valid-ConsE[OF assms(1)] by simp-all
  let ?ls = trace n f g xs y
  have lenls: length ?ls = Suc y
    using trace-length by auto
  moreover have hdls: hd ?ls = the (eval (Pr n f g) (y # xs))
    using Suc trace-hd by auto
  ultimately have g:
    eval g (y # hd ?ls # xs) ↓
    eval (Pr n f g) (Suc y # xs) = eval g (y # hd ?ls # xs)
    using eval-Pr-Suc-converg hdls valid(1) Suc by simp-all
  then have reachable (?stack, None) ((Pr n f g, x # xs, ?ls) # rest, None)
      (is - (?stack1, None))
    using Suc valid(1) by fastforce
  also have reachable ... ((g, y # hd ?ls # xs, []) # (Pr n f g, x # xs, ?ls) # rest, None)
    using Suc.prems lenls by fastforce
  also have reachable ... (?stack1, eval g (y # hd ?ls # xs))
      (is - (-, ?rv))
    using assms(3) g(1) valid valid-ConsI by auto
  also have reachable ... ((Pr n f g, x # xs, (the ?rv) # ?ls) # rest, None)
    using Suc.prems(1) g(1) lenls by auto
  finally have reachable (?stack, None) ((Pr n f g, x # xs, (the ?rv) # ?ls) # rest, None) .
  moreover have trace n f g xs (Suc y) = (the ?rv) # ?ls
    using g(2) trace-Suc by simp
  ultimately show ?case by simp
qed

lemma step-Pr-correct:
  assumes valid (((Pr n f g), xs, []) # rest) (is valid ?stack)
    and ⋀xs rest. valid ((f, xs, []) # rest) ⟹ correct ((f, xs, []) # rest, None)
    and ⋀xs rest. valid ((g, xs, []) # rest) ⟹ correct ((g, xs, []) # rest, None)
  shows correct (?stack, None)
proof -
  have valid: valid rest recfn (length xs) (Pr n f g)
    using valid-ConsE[OF assms(1)] by simp-all
  then have length xs > 0
    by auto
  then obtain y ys where y-ys: xs = y # ys
    using list.exhaust-sel by auto
  let ?t = trace n f g ys
  consider
      (converg) eval (Pr n f g) xs ↓
    | (diverg-f) eval (Pr n f g) xs ↑ and eval f ys ↑
    | (diverg) eval (Pr n f g) xs ↑ and eval f ys ↓
    by auto
  then show ?thesis
  proof (cases)
    case converg
    then have ⋀z. z ≤ y ⟹ reachable (?stack, None) (((Pr n f g), xs, ?t z) # rest, None)
      using assms valid by (simp add: eval-Pr-converg-le reachable-Pr y-ys)
    then have reachable (?stack, None) (((Pr n f g), xs, ?t y) # rest, None)
      by simp
    moreover have reachable (((Pr n f g), xs, ?t y) # rest, None) (rest, Some (hd (?t y)))
      using trace-length step-reachable y-ys by fastforce
```

```
  ultimately have reachable (?stack, None) (rest, Some (hd (?t y)))
    using reachable-transitive by blast
  then show ?thesis
    using assms(1) trace-hd converg y-ys by simp
next
  case diverg-f
  have *: step (?stack, None) = ((f, ys, []) # ((Pr n f g), xs, []) # tl ?stack, None)
      (is - = (?stack1, None))
    using assms(1,2) y-ys by simp
  then have reachable (?stack, None) (?stack1, None)
    using step-reachable by force
  moreover have nonterminating (?stack1, None)
    using assms diverg-f valid valid-ConsI * by auto
  ultimately have nonterminating (?stack, None)
    using reachable-nonterminating by blast
  then show ?thesis using diverg-f(1) assms(1) by simp
next
  case diverg
  let ?h = λz. the (eval (Pr n f g) (z # ys))
  let ?Q = λz. z < y ∧ eval (Pr n f g) (z # ys) ↓
  have ?Q 0
    using assms diverg neq0-conv y-ys valid by fastforce
  define zmax where zmax = Greatest ?Q
  then have ?Q zmax
    using ‹?Q 0› GreatestI-nat[of ?Q 0 y] by simp
  have le-zmax: ⋀z. ?Q z ⟹ z ≤ zmax
    using Greatest-le-nat[of ?Q - y] zmax-def by simp
  have len: length (?t zmax) < Suc y
    by (simp add: ‹?Q zmax› trace-length)
  have eval (Pr n f g) (y # ys) ↓ if y ≤ zmax for y
    using that zmax-def ‹?Q zmax› assms eval-Pr-converg-le[of n f g ys zmax y] valid y-ys
    by simp
  then have reachable (?stack, None) (((Pr n f g), xs, ?t y) # rest, None)
      if y ≤ zmax for y
    using that ‹?Q zmax› diverg y-ys assms reachable-Pr by simp
  then have reachable (?stack, None) (((Pr n f g), xs, ?t zmax) # rest, None)
      (is reachable - (?stack1, None))
    by simp
  also have reachable ...
      ((g, zmax # ?h zmax # tl xs, []) # (Pr n f g, xs, ?t zmax) # rest, None)
      (is - (?stack2, None))
  proof (rule step-reachable)
    have length (?t zmax) ≠ Suc (hd xs)
      using len y-ys by simp
    moreover have hd (?t zmax) = ?h zmax
      using trace-hd by auto
    moreover have length (?t zmax) = Suc zmax
      using trace-length by simp
    ultimately show step (?stack1, None) = (?stack2, None)
      by auto
  qed
  finally have reachable (?stack, None) (?stack2, None) .
  moreover have nonterminating (?stack2, None)
  proof -
    have correct (?stack2, None)
      using y-ys assms valid-ConsI valid by simp
```

```
      moreover have eval g (zmax # ?h zmax # ys) ↑
        using ‹?Q zmax› diverg le-zmax len less-Suc-eq trace-length y-ys valid
        by fastforce
      ultimately show ?thesis using y-ys by simp
    qed
    ultimately have nonterminating (?stack, None)
      using reachable-nonterminating by simp
    then show ?thesis using diverg assms(1) by simp
  qed
qed

lemma reachable-Mn:
  assumes valid ((Mn n f, xs, []) # rest) (is valid ?stack)
    and ⋀xs rest. valid ((f, xs, []) # rest) ⟹ correct ((f, xs, []) # rest, None)
    and ∀y<z. eval f (y # xs) ∉ {None, Some 0}
  shows reachable (?stack, None) ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None)
  using assms(3)
proof (induction z)
  case 0
  then have step (?stack, None) = ((f, 0 # xs, []) # (Mn n f, xs, [0]) # rest, None)
    using assms by simp
  then show ?case
    using step-reachable assms(1) by force
next
  case (Suc z)
  have valid: valid rest recfn (length xs) (Mn n f)
    using valid-ConsE[OF assms(1)] by auto
  have f: eval f (z # xs) ∉ {None, Some 0}
    using Suc by simp
  have reachable (?stack, None) ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None)
    using Suc by simp
  also have reachable ... ((Mn n f, xs, [z]) # rest, eval f (z # xs))
    using f assms(2)[of z # xs] valid correct-convergE valid-ConsI by auto
  also have reachable ... ((f, (Suc z) # xs, []) # (Mn n f, xs, [Suc z]) # rest, None)
      (is - (?stack1, None))
    using step-reachable f by force
  finally have reachable (?stack, None) (?stack1, None) .
  then show ?case by simp
qed

lemma iterate-step-empty-stack: iterate t step ([], rv) = ([], rv)
  using step-empty-stack by (induction t) simp-all

lemma reachable-iterate-step-empty-stack:
  assumes reachable cfg ([], rv)
  shows ∃t. iterate t step cfg = ([], rv) ∧ (∀t'<t. fst (iterate t' step cfg) ≠ [])
proof -
  let ?P = λt. iterate t step cfg = ([], rv)
  from assms have ∃t. ?P t
    by (simp add: reachable-def)
  moreover define tmin where tmin = Least ?P
  ultimately have ?P tmin
    using LeastI-ex[of ?P] by simp
  have fst (iterate t' step cfg) ≠ [] if t' < tmin for t'
  proof
    assume fst (iterate t' step cfg) = []
```

```
    then obtain v where v: iterate t′ step cfg = ([], v)
      by (metis prod.exhaust-sel)
    then have iterate t″ step ([], v) = ([], v) for t″
      using iterate-step-empty-stack by simp
    then have iterate (t′ + t″) step cfg = ([], v) for t″
      using v iterate-additive by fast
    moreover obtain t″ where t′ + t″ = tmin
      using ‹t′ < tmin› less-imp-add-positive by auto
    ultimately have iterate tmin step cfg = ([], v)
      by auto
    then have v = rv
      using ‹?P tmin› by simp
    then have iterate t′ step cfg = ([], rv)
      using v by simp
    moreover have ∀ t′<tmin. ¬ ?P t′
      unfolding tmin-def using not-less-Least[of - ?P] by simp
    ultimately show False
      using that by simp
  qed
  then show ?thesis using ‹?P tmin› by auto
qed

lemma step-Mn-correct:
  assumes valid ((Mn n f, xs, []) # rest) (is valid ?stack)
    and ⋀xs rest. valid ((f, xs, []) # rest) ⟹ correct ((f, xs, []) # rest, None)
  shows correct (?stack, None)
proof −
  have valid: valid rest recfn (length xs) (Mn n f)
    using valid-ConsE[OF assms(1)] by auto
  consider
      (diverg) eval (Mn n f) xs ↑ and ∀ z. eval f (z # xs) ↓
    | (diverg-f) eval (Mn n f) xs ↑ and ∃ z. eval f (z # xs) ↑
    | (converg) eval (Mn n f) xs ↓
    by fast
  then show ?thesis
  proof (cases)
    case diverg
    then have ∀ z. eval f (z # xs) ≠ Some 0
      using eval-Mn-diverg[OF valid(2)] by simp
    then have ∀ y<z. eval f (y # xs) ∉ {None, Some 0} for z
      using diverg by simp
    then have reach-z:
      ⋀z. reachable (?stack, None) ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None)
      using reachable-Mn[OF assms] diverg by simp

    define h :: nat ⇒ configuration where
      h z ≡ ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None) for z
    then have h-inj: ⋀x y. x ≠ y ⟹ h x ≠ h y and z-neq-Nil: ⋀z. fst (h z) ≠ []
      by simp-all

    have z: ∃ z0. ∀ z>z0. ¬ (∃ t′≤t. iterate t′ step (?stack, None) = h z) for t
    proof (induction t)
      case 0
      then show ?case by (metis h-inj le-zero-eq less-not-refl3)
    next
      case (Suc t)
```

```
    then show ?case
      using h-inj by (metis (no-types, opaque-lifting) le-Suc-eq less-not-refl3 less-trans)
  qed

  have nonterminating (?stack, None)
  proof (rule ccontr)
    assume ¬ nonterminating (?stack, None)
    then obtain t where t: fst (iterate t step (?stack, None)) = []
      by auto
    then obtain z_0 where ∀ z>z_0. ¬ (∃ t'≤t. iterate t' step (?stack, None) = h z)
      using z by auto
    then have not-h: ∀ t'≤t. iterate t' step (?stack, None) ≠ h (Suc z_0)
      by simp
    have ∀ t'≥t. fst (iterate t' step (?stack, None)) = []
      using t iterate-step-empty-stack iterate-additive'[of t]
      by (metis le-Suc-ex prod.exhaust-sel)
    then have ∀ t'≥t. iterate t' step (?stack, None) ≠ h (Suc z_0)
      using z-neq-Nil by auto
    then have ∀ t'. iterate t' step (?stack, None) ≠ h (Suc z_0)
      using not-h nat-le-linear by auto
    then have ¬ reachable (?stack, None) (h (Suc z_0))
      using reachable-def by simp
    then show False
      using reach-z[of Suc z_0] h-def by simp
  qed
  then show ?thesis using diverg by simp
next
  case diverg-f
  let ?P = λz. eval f (z # xs) ↑
  define zmin where zmin ≡ Least ?P
  then have ∀ y<zmin. eval f (y # xs) ∉ {None, Some 0}
    using diverg-f eval-Mn-diverg[OF valid(2)] less-trans not-less-Least[of - ?P]
    by blast
  moreover have f-zmin: eval f (zmin # xs) ↑
    using diverg-f LeastI-ex[of ?P] zmin-def by simp
  ultimately have
    reachable (?stack, None) ((f, zmin # xs, []) # (Mn n f, xs, [zmin]) # rest, None)
      (is reachable - (?stack1, None))
    using reachable-Mn[OF assms] by simp
  moreover have nonterminating (?stack1, None)
    using f-zmin assms valid diverg-f valid-ConsI by auto
  ultimately have nonterminating (?stack, None)
    using reachable-nonterminating by simp
  then show ?thesis using diverg-f by simp
next
  case converg
  then obtain z where z: eval (Mn n f) xs ↓= z by auto
  have f-z: eval f (z # xs) ↓= 0
    and f-less-z: ⋀y. y < z ⟹ eval f (y # xs) ↓≠ 0
    using eval-Mn-convergE(2,3)[OF valid(2) z] by simp-all
  then have
    reachable (?stack, None) ((f, z # xs, []) # (Mn n f, xs, [z]) # rest, None)
    using reachable-Mn[OF assms] by simp
  also have reachable ... ((Mn n f, xs, [z]) # rest, eval f (z # xs))
    using assms(2)[of z # xs] valid f-z valid-ConsI correct-convergE
    by auto
```

also have *reachable ... (rest, Some z)*
using *f-z f-less-z step-reachable* **by** *auto*
finally have *reachable (?stack, None) (rest, Some z)* **.**
then show *?thesis* **using** *z* **by** *simp*
qed
qed

theorem *step-correct*:
assumes *valid ((f, xs, []) # rest)*
shows *correct ((f, xs, []) # rest, None)*
using *assms*
proof (*induction f arbitrary: xs rest*)
case *Z*
then show *?case* **using** *valid-ConsE[of Z] step-reachable* **by** *auto*
next
case *S*
then show *?case* **using** *valid-ConsE[of S] step-reachable* **by** *auto*
next
case (*Id m n*)
then show *?case* **using** *valid-ConsE[of Id m n]* **by** *auto*
next
case *Cn*
then show *?case* **using** *step-Cn-correct* **by** *presburger*
next
case *Pr*
then show *?case* **using** *step-Pr-correct* **by** *simp*
next
case *Mn*
then show *?case* **using** *step-Mn-correct* **by** *presburger*
qed

1.5.2 Encoding partial recursive functions

In this section we define an injective, but not surjective, mapping from *recf*s to natural numbers.

abbreviation *triple-encode* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
triple-encode x y z ≡ *prod-encode (x, prod-encode (y, z))*

abbreviation *quad-encode* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
quad-encode w x y z ≡ *prod-encode (w, prod-encode (x, prod-encode (y, z)))*

fun *encode* :: *recf* ⇒ *nat* **where**
encode Z = 0
| *encode S = 1*
| *encode (Id m n) = triple-encode 2 m n*
| *encode (Cn n f gs) = quad-encode 3 n (encode f) (list-encode (map encode gs))*
| *encode (Pr n f g) = quad-encode 4 n (encode f) (encode g)*
| *encode (Mn n f) = triple-encode 5 n (encode f)*

lemma *prod-encode-gr1*: *a > 1* ⟹ *prod-encode (a, x) > 1*
using *le-prod-encode-1 less-le-trans* **by** *blast*

lemma *encode-not-Z-or-S*: *encode f = prod-encode (a, b)* ⟹ *a > 1* ⟹ *f ≠ Z ∧ f ≠ S*
by (*metis encode.simps(1) encode.simps(2) less-numeral-extra(4) not-one-less-zero prod-encode-gr1*)

lemma *encode-injective*: *encode f* = *encode g* ⟹ *f* = *g*
proof (*induction g arbitrary*: *f*)
  case *Z*
  have ⋀*a x*. *a* > *1* ⟹ *prod-encode* (*a*, *x*) > *0*
    using *prod-encode-gr1* **by** (*meson less-one less-trans*)
  then have *f* ≠ *Z* ⟹ *encode f* > *0*
    by (*cases f*) *auto*
  then have *encode f* = *0* ⟹ *f* = *Z* **by** *fastforce*
  then show *?case* **using** *Z* **by** *simp*
next
  case *S*
  have ⋀*a x*. *a* > *1* ⟹ *prod-encode* (*a*, *x*) ≠ *Suc 0*
    using *prod-encode-gr1* **by** (*metis One-nat-def less-numeral-extra*(*4*))
  then have *encode f* = *1* ⟹ *f* = *S*
    by (*cases f*) *auto*
  then show *?case* **using** *S* **by** *simp*
next
  case *Id*
  then obtain *z* **where** ∗: *encode f* = *prod-encode* (*2*, *z*) **by** *simp*
  show *?case*
    using *Id* **by** (*cases f*) (*simp-all add*: ∗ *encode-not-Z-or-S prod-encode-eq*)
next
  case *Cn*
  then obtain *z* **where** ∗: *encode f* = *prod-encode* (*3*, *z*) **by** *simp*
  show *?case*
  proof (*cases f*)
    case *Z*
    then show *?thesis* **using** ∗ *encode-not-Z-or-S* **by** *simp*
  next
    case *S*
    then show *?thesis* **using** ∗ *encode-not-Z-or-S* **by** *simp*
  next
    case *Id*
    then show *?thesis* **using** ∗ **by** (*simp add*: *prod-encode-eq*)
  next
    case *Cn*
    then show *?thesis*
      using ∗ *Cn.IH Cn.prems list-decode-encode*
      by (*smt encode.simps*(*4*) *fst-conv list.inj-map-strong prod-encode-eq snd-conv*)
  next
    case *Pr*
    then show *?thesis* **using** ∗ **by** (*simp add*: *prod-encode-eq*)
  next
    case *Mn*
    then show *?thesis* **using** ∗ **by** (*simp add*: *prod-encode-eq*)
  qed
next
  case *Pr*
  then obtain *z* **where** ∗: *encode f* = *prod-encode* (*4*, *z*) **by** *simp*
  show *?case*
    using *Pr* **by** (*cases f*) (*simp-all add*: ∗ *encode-not-Z-or-S prod-encode-eq*)
next
  case *Mn*
  then obtain *z* **where** ∗: *encode f* = *prod-encode* (*5*, *z*) **by** *simp*
  show *?case*

using *Mn* **by** (*cases f*) (*simp-all add*: ∗ *encode-not-Z-or-S prod-encode-eq*)
qed

definition *encode-kind* :: *nat* ⇒ *nat* **where**
encode-kind e ≡ *if e* = *0 then 0 else if e* = *1 then 1 else pdec1 e*

lemma *encode-kind-0*: *encode-kind* (*encode Z*) = *0*
unfolding *encode-kind-def* **by** *simp*

lemma *encode-kind-1*: *encode-kind* (*encode S*) = *1*
unfolding *encode-kind-def* **by** *simp*

lemma *encode-kind-2*: *encode-kind* (*encode* (*Id m n*)) = *2*
unfolding *encode-kind-def*
by (*metis encode.simps*(*1*−*3*) *encode-injective fst-conv prod-encode-inverse*
recf.simps(*16*) *recf.simps*(*8*))

lemma *encode-kind-3*: *encode-kind* (*encode* (*Cn n f gs*)) = *3*
unfolding *encode-kind-def*
by (*metis encode.simps*(*1*,*2*,*4*) *encode-injective fst-conv prod-encode-inverse*
recf.simps(*10*) *recf.simps*(*18*))

lemma *encode-kind-4*: *encode-kind* (*encode* (*Pr n f g*)) = *4*
unfolding *encode-kind-def*
by (*metis encode.simps*(*1*,*2*,*5*) *encode-injective fst-conv prod-encode-inverse*
recf.simps(*12*) *recf.simps*(*20*))

lemma *encode-kind-5*: *encode-kind* (*encode* (*Mn n f*)) = *5*
unfolding *encode-kind-def*
by (*metis encode.simps*(*1*,*2*,*6*) *encode-injective fst-conv prod-encode-inverse*
recf.simps(*14*) *recf.simps*(*22*))

lemmas *encode-kind-n* =
encode-kind-0 encode-kind-1 encode-kind-2 encode-kind-3 encode-kind-4 encode-kind-5

lemma *encode-kind-Cn*:
assumes *encode-kind* (*encode f*) = *3*
shows ∃ *n f′ gs*. *f* = *Cn n f′ gs*
using *assms encode-kind-n* **by** (*cases f*) *auto*

lemma *encode-kind-Pr*:
assumes *encode-kind* (*encode f*) = *4*
shows ∃ *n f′ g*. *f* = *Pr n f′ g*
using *assms encode-kind-n* **by** (*cases f*) *auto*

lemma *encode-kind-Mn*:
assumes *encode-kind* (*encode f*) = *5*
shows ∃ *n g*. *f* = *Mn n g*
using *assms encode-kind-n* **by** (*cases f*) *auto*

lemma *pdec2-encode-Id*: *pdec2* (*encode* (*Id m n*)) = *prod-encode* (*m*, *n*)
by *simp*

lemma *pdec2-encode-Pr*: *pdec2* (*encode* (*Pr n f g*)) = *triple-encode n* (*encode f*) (*encode g*)
by *simp*

1.5.3 The step function on encoded configurations

In this section we construct a function *estep* :: *nat* ⇒ *nat* that is equivalent to the function *step* :: *configuration* ⇒ *configuration* except that it applies to encoded configurations. We start by defining an encoding for configurations.

definition *encode-frame* :: *frame* ⇒ *nat* **where**
encode-frame s ≡
triple-encode (*encode* (*fst s*)) (*list-encode* (*fst* (*snd s*))) (*list-encode* (*snd* (*snd s*)))

lemma *encode-frame*:
encode-frame (*f*, *xs*, *ls*) = *triple-encode* (*encode f*) (*list-encode xs*) (*list-encode ls*)
unfolding *encode-frame-def* **by** *simp*

abbreviation *encode-option* :: *nat option* ⇒ *nat* **where**
encode-option x ≡ *if x* = *None then 0 else Suc* (*the x*)

definition *encode-config* :: *configuration* ⇒ *nat* **where**
encode-config cfg ≡
prod-encode (*list-encode* (*map encode-frame* (*fst cfg*)), *encode-option* (*snd cfg*))

lemma *encode-config*:
encode-config (*ss*, *rv*) = *prod-encode* (*list-encode* (*map encode-frame ss*), *encode-option rv*)
unfolding *encode-config-def* **by** *simp*

Various projections from encoded configurations:

definition *e2stack* **where** *e2stack e* ≡ *pdec1 e*
definition *e2rv* **where** *e2rv e* ≡ *pdec2 e*
definition *e2tail* **where** *e2tail e* ≡ *e-tl* (*e2stack e*)
definition *e2frame* **where** *e2frame e* ≡ *e-hd* (*e2stack e*)
definition *e2i* **where** *e2i e* ≡ *pdec1* (*e2frame e*)
definition *e2xs* **where** *e2xs e* ≡ *pdec12* (*e2frame e*)
definition *e2ls* **where** *e2ls e* ≡ *pdec22* (*e2frame e*)
definition *e2lenas* **where** *e2lenas e* ≡ *e-length* (*e2xs e*)
definition *e2lenls* **where** *e2lenls e* ≡ *e-length* (*e2ls e*)

lemma *e2rv-rv* [*simp*]:
e2rv (*encode-config* (*ss*, *rv*)) = (*if rv* ↑ *then 0 else Suc* (*the rv*))
unfolding *e2rv-def* **using** *encode-config* **by** *simp*

lemma *e2stack-stack* [*simp*]:
e2stack (*encode-config* (*ss*, *rv*)) = *list-encode* (*map encode-frame ss*)
unfolding *e2stack-def* **using** *encode-config* **by** *simp*

lemma *e2tail-tail* [*simp*]:
e2tail (*encode-config* (*s* # *ss*, *rv*)) = *list-encode* (*map encode-frame ss*)
unfolding *e2tail-def* **using** *encode-config* **by** *fastforce*

lemma *e2frame-frame* [*simp*]:
e2frame (*encode-config* (*s* # *ss*, *rv*)) = *encode-frame s*
unfolding *e2frame-def* **using** *encode-config* **by** *fastforce*

lemma *e2i-f* [*simp*]:
e2i (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *encode f*
unfolding *e2i-def* **using** *encode-config e2frame-frame encode-frame* **by** *force*

lemma *e2xs-xs* [*simp*]:
  e2xs (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *list-encode xs*
  using *e2xs-def e2frame-frame encode-frame* **by** *force*

lemma *e2ls-ls* [*simp*]:
  e2ls (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *list-encode ls*
  using *e2ls-def e2frame-frame encode-frame* **by** *force*

lemma *e2lenas-lenas* [*simp*]:
  e2lenas (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *length xs*
  using *e2lenas-def e2frame-frame encode-frame* **by** *simp*

lemma *e2lenls-lenls* [*simp*]:
  e2lenls (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *length ls*
  using *e2lenls-def e2frame-frame encode-frame* **by** *simp*

lemma *e2stack-0-iff-Nil*:
  assumes *e* = *encode-config* (*ss*, *rv*)
  shows *e2stack e* = *0* ⟷ *ss* = []
  using *assms*
  by (*metis list-encode.simps*(*1*) *e2stack-stack list-encode-0 map-is-Nil-conv*)

lemma *e2ls-0-iff-Nil* [*simp*]: *list-decode* (*e2ls e*) = [] ⟷ *e2ls e* = *0*
  by (*metis list-decode.simps*(*1*) *list-encode-decode*)

We now define *eterm* piecemeal by considering the more complicated cases *Cn*, *Pr*, and *Mn* separately.

```
definition estep-Cn e ≡
  if e2lenls e = e-length (pdec222 (e2i e))
  then if e2rv e = 0
       then prod-encode (e-cons (triple-encode (pdec122 (e2i e)) (e2ls e) 0) (e2stack e), 0)
       else prod-encode (e2tail e, e2rv e)
  else if e2rv e = 0
       then if e2lenls e < e-length (pdec222 (e2i e))
            then prod-encode
              (e-cons
                (triple-encode (e-nth (pdec222 (e2i e)) (e2lenls e)) (e2xs e) 0)
                (e2stack e),
               0)
            else prod-encode (e2tail e, e2rv e)
       else prod-encode
         (e-cons
           (triple-encode (e2i e) (e2xs e) (e-snoc (e2ls e) (e2rv e − 1)))
           (e2tail e),
          0)
```

lemma *estep-Cn*:
  assumes *c* = (((*Cn n f gs*, *xs*, *ls*) # *fs*), *rv*)
  shows *estep-Cn* (*encode-config c*) = *encode-config* (*step c*)
  using *encode-frame* **by** (*simp add*: *assms estep-Cn-def*, *simp add*: *encode-config assms*)

```
definition estep-Pr e ≡
  if e2ls e = 0
  then if e2rv e = 0
       then prod-encode
         (e-cons (triple-encode (pdec122 (e2i e)) (e-tl (e2xs e)) 0) (e2stack e),
```

```
          0)
        else prod-encode
          (e-cons (triple-encode (e2i e) (e2xs e) (singleton-encode (e2rv e − 1))) (e2tail e),
          0)
  else if e2lenls e = Suc (e-hd (e2xs e))
        then prod-encode (e2tail e, Suc (e-hd (e2ls e)))
        else if e2rv e = 0
             then prod-encode
               (e-cons
                 (triple-encode
                   (pdec222 (e2i e))
                   (e-cons (e2lenls e − 1) (e-cons (e-hd (e2ls e)) (e-tl (e2xs e))))
                   0)
                 (e2stack e),
                 0)
             else prod-encode
               (e-cons
                 (triple-encode (e2i e) (e2xs e) (e-cons (e2rv e − 1) (e2ls e))) (e2tail e),
                 0)

lemma estep-Pr1:
  assumes c = (((Pr n f g, xs, ls) # fs), rv)
    and ls ≠ []
    and length ls ≠ Suc (hd xs)
    and rv ≠ None
    and recfn (length xs) (Pr n f g)
  shows estep-Pr (encode-config c) = encode-config (step c)
proof −
  let ?e = encode-config c
  from assms(5) have length xs > 0 by auto
  then have eq: hd xs = e-hd (e2xs ?e)
    using assms e-hd-def by auto
  have step c = ((Pr n f g, xs, (the rv) # ls) # fs, None)
      (is step c = (?t # ?ss, None))
    using assms by simp
  then have encode-config (step c) =
      prod-encode (list-encode (map encode-frame (?t # ?ss)), 0)
    using encode-config by simp
  also have ... =
      prod-encode (e-cons (encode-frame ?t) (list-encode (map encode-frame (?ss))), 0)
    by simp
  also have ... = prod-encode (e-cons (encode-frame ?t) (e2tail ?e), 0)
    using assms(1) by simp
  also have ... = prod-encode
      (e-cons
        (triple-encode (e2i ?e) (e2xs ?e) (e-cons (e2rv ?e − 1) (e2ls ?e)))
        (e2tail ?e),
       0)
    by (simp add: assms encode-frame)
  finally show ?thesis
    using assms eq estep-Pr-def by auto
qed

lemma estep-Pr2:
  assumes c = (((Pr n f g, xs, ls) # fs), rv)
    and ls ≠ []
```

```
    and length ls ≠ Suc (hd xs)
    and rv = None
    and recfn (length xs) (Pr n f g)
  shows estep-Pr (encode-config c) = encode-config (step c)
proof –
  let ?e = encode-config c
  from assms(5) have length xs > 0 by auto
  then have eq: hd xs = e-hd (e2xs ?e)
    using assms e-hd-def by auto
  have step c = ((g, (length ls − 1) # hd ls # tl xs, []) # (Pr n f g, xs, ls) # fs, None)
      (is step c = (?t # ?ss, None))
    using assms by simp
  then have encode-config (step c) =
      prod-encode (list-encode (map encode-frame (?t # ?ss)), 0)
    using encode-config by simp
  also have ... =
      prod-encode (e-cons (encode-frame ?t) (list-encode (map encode-frame (?ss))), 0)
    by simp
  also have ... = prod-encode (e-cons (encode-frame ?t) (e2stack ?e), 0)
    using assms(1) by simp
  also have ... = prod-encode
    (e-cons
      (triple-encode
        (pdec222 (e2i ?e))
        (e-cons (e2lenls ?e − 1) (e-cons (e-hd (e2ls ?e)) (e-tl (e2xs ?e))))
        0)
      (e2stack ?e),
     0)
    using assms(1,2) encode-frame[of g (length ls − 1) # hd ls # tl xs []]
      pdec2-encode-Pr[of n f g] e2xs-xs e2i-f e2lenls-lenls e2ls-ls e-hd
    by (metis list-encode.simps(1) list.collapse list-decode-encode
      prod-encode-inverse snd-conv)
  finally show ?thesis
    using assms eq estep-Pr-def by auto
qed

lemma estep-Pr3:
  assumes c = (((Pr n f g, xs, ls) # fs), rv)
    and ls ≠ []
    and length ls = Suc (hd xs)
    and recfn (length xs) (Pr n f g)
  shows estep-Pr (encode-config c) = encode-config (step c)
proof –
  let ?e = encode-config c
  from assms(4) have length xs > 0 by auto
  then have hd xs = e-hd (e2xs ?e)
    using assms e-hd-def by auto
  then have (length ls = Suc (hd xs)) = (e2lenls ?e = Suc (e-hd (e2xs ?e)))
    using assms by simp
  then have *: estep-Pr ?e = prod-encode (e2tail ?e, Suc (e-hd (e2ls ?e)))
    using assms estep-Pr-def by auto
  have step c = (fs, Some (hd ls))
    using assms(1,2,3) by simp
  then have encode-config (step c) =
      prod-encode (list-encode (map encode-frame fs), encode-option (Some (hd ls)))
    using encode-config by simp
```

```
  also have ... =
      prod-encode (list-encode (map encode-frame fs), encode-option (Some (e-hd (e2ls ?e))))
    using assms(1,2) e-hd-def by auto
  also have ... = prod-encode (list-encode (map encode-frame fs), Suc (e-hd (e2ls ?e)))
    by simp
  also have ... = prod-encode (e2tail ?e, Suc (e-hd (e2ls ?e)))
    using assms(1) by simp
  finally have encode-config (step c) = prod-encode (e2tail ?e, Suc (e-hd (e2ls ?e))) .
  then show ?thesis
    using estep-Pr-def * by presburger
qed

lemma estep-Pr4:
  assumes c = (((Pr n f g, xs, ls) # fs), rv) and ls = []
  shows estep-Pr (encode-config c) = encode-config (step c)
  using encode-frame
  by (simp add: assms estep-Pr-def, simp add: encode-config assms)

lemma estep-Pr:
  assumes c = (((Pr n f g, xs, ls) # fs), rv)
    and recfn (length xs) (Pr n f g)
  shows estep-Pr (encode-config c) = encode-config (step c)
  using assms estep-Pr1 estep-Pr2 estep-Pr3 estep-Pr4 by auto

definition estep-Mn e ≡
  if e2ls e = 0
  then prod-encode
    (e-cons
      (triple-encode (pdec22 (e2i e)) (e-cons 0 (e2xs e)) 0)
      (e-cons
        (triple-encode (e2i e) (e2xs e) (singleton-encode 0))
        (e2tail e)),
     0)
  else if e2rv e = 1
       then prod-encode (e2tail e, Suc (e-hd (e2ls e)))
       else prod-encode
        (e-cons
          (triple-encode (pdec22 (e2i e)) (e-cons (Suc (e-hd (e2ls e))) (e2xs e)) 0)
          (e-cons
            (triple-encode (e2i e) (e2xs e) (singleton-encode (Suc (e-hd (e2ls e)))))
            (e2tail e)),
        0)

lemma estep-Mn:
  assumes c = (((Mn n f, xs, ls) # fs), rv)
  shows estep-Mn (encode-config c) = encode-config (step c)
proof −
  let ?e = encode-config c
  consider ls ≠ [] and rv ≠ Some 0 | ls ≠ [] and rv = Some 0 | ls = []
    by auto
  then show ?thesis
  proof (cases)
    case 1
    then have step-c: step c =
        ((f, (Suc (hd ls)) # xs, []) # (Mn n f, xs, [Suc (hd ls)]) # fs, None)
        (is step c = ?cfg)
```

```
    using assms by simp
  have estep-Mn ?e =
    prod-encode
      (e-cons
        (triple-encode (encode f) (e-cons (Suc (hd ls)) (list-encode xs)) 0)
        (e-cons
          (triple-encode (encode (Mn n f)) (list-encode xs) (singleton-encode (Suc (hd ls))))
          (list-encode (map encode-frame fs))),
      0)
    using 1 assms e-hd-def estep-Mn-def by auto
  also have ... = encode-config ?cfg
    using encode-config by (simp add: encode-frame)
  finally show ?thesis
    using step-c by simp
 next
  case 2
  have estep-Mn ?e = prod-encode (e2tail ?e, Suc (e-hd (e2ls ?e)))
    using 2 assms estep-Mn-def by auto
  also have ... = prod-encode (e2tail ?e, Suc (hd ls))
    using 2 assms e-hd-def by auto
  also have ... = prod-encode (list-encode (map encode-frame fs), Suc (hd ls))
    using assms by simp
  also have ... = encode-config (fs, Some (hd ls))
    using encode-config by simp
  finally show ?thesis
    using 2 assms by simp
 next
  case 3
  then show ?thesis
    using assms encode-frame by (simp add: estep-Mn-def, simp add: encode-config)
 qed
qed

definition estep e ≡
 if e2stack e = 0 then prod-encode (0, e2rv e)
 else if e2i e = 0 then prod-encode (e2tail e, 1)
 else if e2i e = 1 then prod-encode (e2tail e, Suc (Suc (e-hd (e2xs e))))
 else if encode-kind (e2i e) = 2 then
  prod-encode (e2tail e, Suc (e-nth (e2xs e) (pdec22 (e2i e))))
 else if encode-kind (e2i e) = 3 then estep-Cn e
 else if encode-kind (e2i e) = 4 then estep-Pr e
 else if encode-kind (e2i e) = 5 then estep-Mn e
 else 0

lemma estep-Z:
 assumes c = (((Z, xs, ls) # fs), rv)
 shows estep (encode-config c) = encode-config (step c)
 using encode-frame by (simp add: assms estep-def, simp add: encode-config assms)

lemma estep-S:
 assumes c = (((S, xs, ls) # fs), rv)
  and recfn (length xs) (fst (hd (fst c)))
 shows estep (encode-config c) = encode-config (step c)
proof -
 let ?e = encode-config c
 from assms have length xs > 0 by auto
```

then have *eq*: *hd xs* = *e-hd* (*e2xs ?e*)
  using *assms*(*1*) *e-hd-def* **by** *auto*
then have *estep ?e* = *prod-encode* (*e2tail ?e*, *Suc* (*Suc* (*e-hd* (*e2xs ?e*))))
  using *assms*(*1*) *estep-def* **by** *simp*
moreover have *step c* = (*fs*, *Some* (*Suc* (*hd xs*)))
  using *assms*(*1*) **by** *simp*
ultimately show *?thesis*
  using *assms*(*1*) *eq estep-def encode-config*[*of fs Some* (*Suc* (*hd xs*))] **by** *simp*
qed

lemma *estep-Id*:
assumes *c* = (((*Id m n*, *xs*, *ls*) # *fs*), *rv*)
  and *recfn* (*length xs*) (*fst* (*hd* (*fst c*)))
shows *estep* (*encode-config c*) = *encode-config* (*step c*)
proof −
let *?e* = *encode-config c*
from *assms* **have** *length xs* = *m* **and** *m* > *0* **by** *auto*
then have *eq*: *xs* ! *n* = *e-nth* (*e2xs ?e*) *n*
  using *assms e-hd-def* **by** *auto*
moreover have *encode-kind* (*e2i ?e*) = *2*
  using *assms*(*1*) *encode-kind-2* **by** *auto*
ultimately have *estep ?e* =
    prod-encode (*e2tail ?e*, *Suc* (*e-nth* (*e2xs ?e*) (*pdec22* (*e2i ?e*))))
  using *assms estep-def encode-kind-def* **by** *auto*
moreover have *step c* = (*fs*, *Some* (*xs* ! *n*))
  using *assms*(*1*) **by** *simp*
ultimately show *?thesis*
  using *assms*(*1*) *eq encode-config*[*of fs Some* (*xs* ! *n*)] **by** *simp*
qed

lemma *estep*:
assumes *valid* (*fst c*)
shows *estep* (*encode-config c*) = *encode-config* (*step c*)
proof (*cases fst c*)
case *Nil*
then show *?thesis*
  using *estep-def*
  by (*metis list-encode.simps*(*1*) *e2rv-def e2stack-stack encode-config-def*
    map-is-Nil-conv prod.collapse prod-encode-inverse snd-conv step.simps(*1*))
next
case (*Cons s fs*)
then obtain *f xs ls rv* **where** *c*: *c* = ((*f*, *xs*, *ls*) # *fs*, *rv*)
  by (*metis prod.exhaust-sel*)
with *assms valid-def* **have** *lenas*: *recfn* (*length xs*) *f* **by** *simp*
show *?thesis*
proof (*cases f*)
  case *Z*
  then show *?thesis* **using** *estep-Z c* **by** *simp*
next
  case *S*
  then show *?thesis* **using** *estep-S c lenas* **by** *simp*
next
  case *Id*
  then show *?thesis* **using** *estep-Id c lenas* **by** *simp*
next
  case *Cn*

then show *?thesis*
using *estep-Cn c*
by (*metis e2i-f e2stack-0-iff-Nil encode.simps*(*1*) *encode.simps*(*2*) *encode-kind-2 encode-kind-3 encode-kind-Cn estep-def list.distinct*(*1*) *recf.distinct*(*13*) *recf.distinct*(*19*) *recf.distinct*(*5*))
next
case *Pr*
then show *?thesis*
using *estep-Pr c lenas*
by (*metis e2i-f e2stack-0-iff-Nil encode.simps*(*1*) *encode.simps*(*2*) *encode-kind-2 encode-kind-4 encode-kind-Cn encode-kind-Pr estep-def list.distinct*(*1*) *recf.distinct*(*15*) *recf.distinct*(*21*) *recf.distinct*(*25*) *recf.distinct*(*7*))
next
case *Mn*
then show *?thesis*
using *estep-Pr c lenas*
by (*metis* (*no-types*, *lifting*) *e2i-f e2stack-0-iff-Nil encode.simps*(*1*) *encode.simps*(*2*) *encode-kind-2 encode-kind-5 encode-kind-Cn encode-kind-Mn encode-kind-Pr estep-Mn estep-def list.distinct*(*1*) *recf.distinct*(*17*) *recf.distinct*(*23*) *recf.distinct*(*27*) *recf.distinct*(*9*))
qed
qed

1.5.4 The step function as a partial recursive function

In this section we construct a primitive recursive function *r-step* computing *estep*. This will entail defining *recf*s for many functions defined in the previous section.

definition *r-e2stack* ≡ *r-pdec1*

lemma *r-e2stack-prim*: *prim-recfn 1 r-e2stack*
unfolding *r-e2stack-def* **using** *r-pdec1-prim* **by** *simp*

lemma *r-e2stack* [*simp*]: *eval r-e2stack* [*e*] ↓= *e2stack e*
unfolding *r-e2stack-def e2stack-def* **using** *r-pdec1-prim* **by** *simp*

definition *r-e2rv* ≡ *r-pdec2*

lemma *r-e2rv-prim*: *prim-recfn 1 r-e2rv*
unfolding *r-e2rv-def* **using** *r-pdec2-prim* **by** *simp*

lemma *r-e2rv* [*simp*]: *eval r-e2rv* [*e*] ↓= *e2rv e*
unfolding *r-e2rv-def e2rv-def* **using** *r-pdec2-prim* **by** *simp*

definition *r-e2tail* ≡ *Cn 1 r-tl* [*r-e2stack*]

lemma *r-e2tail-prim*: *prim-recfn 1 r-e2tail*
unfolding *r-e2tail-def* **using** *r-e2stack-prim r-tl-prim* **by** *simp*

lemma *r-e2tail* [*simp*]: *eval r-e2tail* [*e*] ↓= *e2tail e*
unfolding *r-e2tail-def e2tail-def* **using** *r-e2stack-prim r-tl-prim* **by** *simp*

definition *r-e2frame* ≡ *Cn 1 r-hd* [*r-e2stack*]

lemma *r-e2frame-prim*: *prim-recfn 1 r-e2frame*
unfolding *r-e2frame-def* **using** *r-hd-prim r-e2stack-prim* **by** *simp*

lemma *r-e2frame* [*simp*]: *eval r-e2frame* [*e*] ↓= *e2frame e*
  unfolding *r-e2frame-def e2frame-def* **using** *r-hd-prim r-e2stack-prim* **by** *simp*

definition *r-e2i* ≡ *Cn 1 r-pdec1* [*r-e2frame*]

lemma *r-e2i-prim*: *prim-recfn 1 r-e2i*
  unfolding *r-e2i-def* **using** *r-pdec12-prim r-e2frame-prim* **by** *simp*

lemma *r-e2i* [*simp*]: *eval r-e2i* [*e*] ↓= *e2i e*
  unfolding *r-e2i-def e2i-def* **using** *r-pdec12-prim r-e2frame-prim* **by** *simp*

definition *r-e2xs* ≡ *Cn 1 r-pdec12* [*r-e2frame*]

lemma *r-e2xs-prim*: *prim-recfn 1 r-e2xs*
  unfolding *r-e2xs-def* **using** *r-pdec122-prim r-e2frame-prim* **by** *simp*

lemma *r-e2xs* [*simp*]: *eval r-e2xs* [*e*] ↓= *e2xs e*
  unfolding *r-e2xs-def e2xs-def* **using** *r-pdec122-prim r-e2frame-prim* **by** *simp*

definition *r-e2ls* ≡ *Cn 1 r-pdec22* [*r-e2frame*]

lemma *r-e2ls-prim*: *prim-recfn 1 r-e2ls*
  unfolding *r-e2ls-def* **using** *r-pdec222-prim r-e2frame-prim* **by** *simp*

lemma *r-e2ls* [*simp*]: *eval r-e2ls* [*e*] ↓= *e2ls e*
  unfolding *r-e2ls-def e2ls-def* **using** *r-pdec222-prim r-e2frame-prim* **by** *simp*

definition *r-e2lenls* ≡ *Cn 1 r-length* [*r-e2ls*]

lemma *r-e2lenls-prim*: *prim-recfn 1 r-e2lenls*
  unfolding *r-e2lenls-def* **using** *r-length-prim r-e2ls-prim* **by** *simp*

lemma *r-e2lenls* [*simp*]: *eval r-e2lenls* [*e*] ↓= *e2lenls e*
  unfolding *r-e2lenls-def e2lenls-def* **using** *r-length-prim r-e2ls-prim* **by** *simp*

definition *r-kind* ≡
  Cn 1 r-ifz [*Id 1 0*, *Z*, *Cn 1 r-ifeq* [*Id 1 0*, *r-const 1*, *r-const 1*, *r-pdec1*]]

lemma *r-kind-prim*: *prim-recfn 1 r-kind*
  unfolding *r-kind-def* **by** *simp*

lemma *r-kind*: *eval r-kind* [*e*] ↓= *encode-kind e*
  unfolding *r-kind-def encode-kind-def* **by** *simp*

lemmas *helpers-for-r-step-prim* =
  r-e2i-prim
  r-e2lenls-prim
  r-e2ls-prim
  r-e2rv-prim
  r-e2xs-prim
  r-e2stack-prim
  r-e2tail-prim
  r-e2frame-prim

We define primitive recursive functions *r-step-Id*, *r-step-Cn*, *r-step-Pr*, and *r-step-Mn*.

The last three correspond to *estep-Cn*, *estep-Pr*, and *estep-Mn* from the previous section.

```
definition r-step-Id ≡
  Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-nth [r-e2xs, Cn 1 r-pdec22 [r-e2i]]]]

lemma r-step-Id:
  eval r-step-Id [e] ↓= prod-encode (e2tail e, Suc (e-nth (e2xs e) (pdec22 (e2i e))))
  unfolding r-step-Id-def using helpers-for-r-step-prim by simp

abbreviation r-triple-encode :: recf ⇒ recf ⇒ recf ⇒ recf where
  r-triple-encode x y z ≡ Cn 1 r-prod-encode [x, Cn 1 r-prod-encode [y, z]]

definition r-step-Cn ≡
  Cn 1 r-ifeq
   [r-e2lenls,
    Cn 1 r-length [Cn 1 r-pdec222 [r-e2i]],
    Cn 1 r-ifz
     [r-e2rv,
      Cn 1 r-prod-encode
       [Cn 1 r-cons [r-triple-encode (Cn 1 r-pdec122 [r-e2i]) r-e2ls Z, r-e2stack],
        Z],
      Cn 1 r-prod-encode [r-e2tail, r-e2rv]],
    Cn 1 r-ifz
     [r-e2rv,
      Cn 1 r-ifless
       [r-e2lenls,
        Cn 1 r-length [Cn 1 r-pdec222 [r-e2i]],
        Cn 1 r-prod-encode
         [Cn 1 r-cons
           [r-triple-encode (Cn 1 r-nth [Cn 1 r-pdec222 [r-e2i], r-e2lenls]) r-e2xs Z,
            r-e2stack],
          Z],
        Cn 1 r-prod-encode [r-e2tail, r-e2rv]],
      Cn 1 r-prod-encode
       [Cn 1 r-cons
         [r-triple-encode r-e2i r-e2xs (Cn 1 r-snoc [r-e2ls, Cn 1 r-dec [r-e2rv]]),
          r-e2tail],
        Z]]]

lemma r-step-Cn-prim: prim-recfn 1 r-step-Cn
  unfolding r-step-Cn-def using helpers-for-r-step-prim by simp

lemma r-step-Cn: eval r-step-Cn [e] ↓= estep-Cn e
  unfolding r-step-Cn-def estep-Cn-def using helpers-for-r-step-prim by simp

definition r-step-Pr ≡
  Cn 1 r-ifz
   [r-e2ls,
    Cn 1 r-ifz
     [r-e2rv,
      Cn 1 r-prod-encode
       [Cn 1 r-cons
         [r-triple-encode (Cn 1 r-pdec122 [r-e2i]) (Cn 1 r-tl [r-e2xs]) Z,
          r-e2stack],
        Z],
      Cn 1 r-prod-encode
```

```
      [Cn 1 r-cons
        [r-triple-encode r-e2i r-e2xs (Cn 1 r-singleton-encode [Cn 1 r-dec [r-e2rv]]),
         r-e2tail],
       Z]],
   Cn 1 r-ifeq
    [r-e2lenls,
     Cn 1 S [Cn 1 r-hd [r-e2xs]],
     Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-hd [r-e2ls]]],
     Cn 1 r-ifz
      [r-e2rv,
       Cn 1 r-prod-encode
        [Cn 1 r-cons
          [r-triple-encode
            (Cn 1 r-pdec222 [r-e2i])
            (Cn 1 r-cons
              [Cn 1 r-dec [r-e2lenls],
               Cn 1 r-cons [Cn 1 r-hd [r-e2ls],
               Cn 1 r-tl [r-e2xs]]])
            Z,
           r-e2stack],
         Z],
       Cn 1 r-prod-encode
        [Cn 1 r-cons
          [r-triple-encode r-e2i r-e2xs (Cn 1 r-cons [Cn 1 r-dec [r-e2rv], r-e2ls]),
           r-e2tail],
         Z]]]]
```

lemma *r-step-Pr-prim*: *prim-recfn 1 r-step-Pr*
  unfolding *r-step-Pr-def* **using** *helpers-for-r-step-prim* **by** *simp*

lemma *r-step-Pr*: *eval r-step-Pr [e] ↓= estep-Pr e*
  unfolding *r-step-Pr-def estep-Pr-def* **using** *helpers-for-r-step-prim* **by** *simp*

definition *r-step-Mn* ≡

```
  Cn 1 r-ifz
   [r-e2ls,
    Cn 1 r-prod-encode
     [Cn 1 r-cons
       [r-triple-encode (Cn 1 r-pdec22 [r-e2i]) (Cn 1 r-cons [Z, r-e2xs]) Z,
        Cn 1 r-cons
         [r-triple-encode r-e2i r-e2xs (Cn 1 r-singleton-encode [Z]),
          r-e2tail]],
      Z],
    Cn 1 r-ifeq
     [r-e2rv,
      r-const 1,
      Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-hd [r-e2ls]]],
      Cn 1 r-prod-encode
       [Cn 1 r-cons
         [r-triple-encode
           (Cn 1 r-pdec22 [r-e2i])
           (Cn 1 r-cons [Cn 1 S [Cn 1 r-hd [r-e2ls]], r-e2xs])
           Z,
          Cn 1 r-cons
           [r-triple-encode r-e2i r-e2xs (Cn 1 r-singleton-encode [Cn 1 S [Cn 1 r-hd [r-e2ls]]]),
            r-e2tail]],
```

```
      Z]]]

lemma r-step-Mn-prim: prim-recfn 1 r-step-Mn
  unfolding r-step-Mn-def using helpers-for-r-step-prim by simp

lemma r-step-Mn: eval r-step-Mn [e] ↓= estep-Mn e
  unfolding r-step-Mn-def estep-Mn-def using helpers-for-r-step-prim by simp

definition r-step ≡
  Cn 1 r-ifz
   [r-e2stack,
    Cn 1 r-prod-encode [Z, r-e2rv],
    Cn 1 r-ifz
     [r-e2i,
      Cn 1 r-prod-encode [r-e2tail, r-const 1],
      Cn 1 r-ifeq
       [r-e2i,
        r-const 1,
        Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 S [Cn 1 r-hd [r-e2xs]]]],
        Cn 1 r-ifeq
         [Cn 1 r-kind [r-e2i],
          r-const 2,
          Cn 1 r-prod-encode [r-e2tail, Cn 1 S [Cn 1 r-nth [r-e2xs, Cn 1 r-pdec22 [r-e2i]]]],
          Cn 1 r-ifeq
           [Cn 1 r-kind [r-e2i],
            r-const 3,
            r-step-Cn,
            Cn 1 r-ifeq
             [Cn 1 r-kind [r-e2i],
              r-const 4,
              r-step-Pr,
              Cn 1 r-ifeq
               [Cn 1 r-kind [r-e2i], r-const 5, r-step-Mn, Z]]]]]]]

lemma r-step-prim: prim-recfn 1 r-step
  unfolding r-step-def
  using r-kind-prim r-step-Mn-prim r-step-Pr-prim r-step-Cn-prim helpers-for-r-step-prim
  by simp

lemma r-step: eval r-step [e] ↓= estep e
  unfolding r-step-def estep-def
  using r-kind-prim r-step-Mn-prim r-step-Pr-prim r-step-Cn-prim helpers-for-r-step-prim
    r-kind r-step-Cn r-step-Pr r-step-Mn
  by simp

theorem r-step-equiv-step:
  assumes valid (fst c)
  shows eval r-step [encode-config c] ↓= encode-config (step c)
  using r-step estep assms by simp
```

1.5.5 The universal function

The next function computes the configuration after arbitrarily many steps.

```
definition r-leap ≡
  Pr 2
```

(*Cn 2 r-prod-encode*
  [*Cn 2 r-singleton-encode*
    [*Cn 2 r-prod-encode* [*Id 2 0*, *Cn 2 r-prod-encode* [*Id 2 1*, *r-constn 1 0*]]],
   r-constn 1 0])
(*Cn 4 r-step* [*Id 4 1*])

lemma *r-leap-prim* [*simp*]: *prim-recfn 3 r-leap*
  unfolding *r-leap-def* **using** *r-step-prim* **by** *simp*

lemma *r-leap-total*: *eval r-leap* [*t*, *i*, *x*] ↓
  using *prim-recfn-total*[*OF r-leap-prim*] **by** *simp*

lemma *r-leap*:
  assumes *i* = *encode f* **and** *recfn* (*e-length x*) *f*
  shows *eval r-leap* [*t*, *i*, *x*] ↓= *encode-config* (*iterate t step* ([(*f*, *list-decode x*, [])], *None*))
proof (*induction t*)
  case *0*
  then show *?case*
    unfolding *r-leap-def* **using** *r-step-prim assms encode-config encode-frame* **by** *simp*
next
  case (*Suc t*)
  let *?c* = ([(*f*, *list-decode x*, [])], *None*)
  let *?tc* = *iterate t step ?c*
  have *valid* (*fst ?c*)
    using *valid-def assms* **by** *simp*
  then have *valid*: *valid* (*fst ?tc*)
    using *iterate-step-valid* **by** *simp*
  have *eval r-leap* [*Suc t*, *i*, *x*] =
      eval (*Cn 4 r-step* [*Id 4 1*]) [*t*, *the* (*eval r-leap* [*t*, *i*, *x*]), *i*, *x*]
   by (*smt One-nat-def Suc-eq-plus1 eq-numeral-Suc eval-Pr-converg-Suc list.size*(*3*) *list.size*(*4*) *nat-1-add-1 pred-numeral-simps*(*3*) *r-leap-def r-leap-prim r-leap-total*)
  then have *eval r-leap* [*Suc t*, *i*, *x*] = *eval* (*Cn 4 r-step* [*Id 4 1*]) [*t*, *encode-config ?tc*, *i*, *x*]
    using *Suc* **by** *simp*
  then have *eval r-leap* [*Suc t*, *i*, *x*] = *eval r-step* [*encode-config ?tc*]
    using *r-step-prim* **by** *simp*
  then have *eval r-leap* [*Suc t*, *i*, *x*] ↓= *encode-config* (*step ?tc*)
   by (*simp add*: *r-step-equiv-step valid*)
  then show *?case* **by** *simp*
qed

lemma *step-leaves-empty-stack-empty*:
  assumes *iterate t step* ([(*f*, *list-decode x*, [])], *None*) = ([], *Some v*)
  shows *iterate* (t + t') *step* ([(*f*, *list-decode x*, [])], *None*) = ([], *Some v*)
  using *assms* **by** (*induction* t') *simp-all*

The next function is essentially a convenience wrapper around *r-leap*. It returns zero if the configuration returned by *r-leap* is non-final, and *Suc v* if the configuration is final with return value *v*.

definition *r-result* ≡
  Cn 3 r-ifz [*Cn 3 r-pdec1* [*r-leap*], *Cn 3 r-pdec2* [*r-leap*], *r-constn 2 0*]

lemma *r-result-prim* [*simp*]: *prim-recfn 3 r-result*
  unfolding *r-result-def* **using** *r-leap-prim* **by** *simp*

lemma *r-result-total*: *total r-result*
  using *r-result-prim* **by** *blast*

lemma *r-result-empty-stack-None*:
  assumes *i* = *encode f*
    and *recfn* (*e-length x*) *f*
    and *iterate t step* ([(*f*, *list-decode x*, [])], *None*) = ([], *None*)
  shows *eval r-result* [*t*, *i*, *x*] ↓= *0*
  unfolding *r-result-def*
  using *assms r-leap e2stack-0-iff-Nil e2stack-def e2stack-stack r-leap-total r-leap-prim e2rv-def e2rv-rv*
  by *simp*

lemma *r-result-empty-stack-Some*:
  assumes *i* = *encode f*
    and *recfn* (*e-length x*) *f*
    and *iterate t step* ([(*f*, *list-decode x*, [])], *None*) = ([], *Some v*)
  shows *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
  unfolding *r-result-def*
  using *assms r-leap e2stack-0-iff-Nil e2stack-def e2stack-stack r-leap-total r-leap-prim e2rv-def e2rv-rv*
  by *simp*

lemma *r-result-empty-stack-stays*:
  assumes *i* = *encode f*
    and *recfn* (*e-length x*) *f*
    and *iterate t step* ([(*f*, *list-decode x*, [])], *None*) = ([], *Some v*)
  shows *eval r-result* [*t* + *t'*, *i*, *x*] ↓= *Suc v*
  using *assms step-leaves-empty-stack-empty r-result-empty-stack-Some* **by** *simp*

lemma *r-result-nonempty-stack*:
  assumes *i* = *encode f*
    and *recfn* (*e-length x*) *f*
    and *fst* (*iterate t step* ([(*f*, *list-decode x*, [])], *None*)) ≠ []
  shows *eval r-result* [*t*, *i*, *x*] ↓= *0*
proof −
  obtain *ss rv* **where** *iterate t step* ([(*f*, *list-decode x*, [])], *None*) = (*ss*, *rv*)
    by *fastforce*
  moreover from *this assms*(*3*) **have** *ss* ≠ [] **by** *simp*
  ultimately have *eval r-leap* [*t*, *i*, *x*] ↓= *encode-config* (*ss*, *rv*)
    using *assms r-leap* **by** *simp*
  then have *eval* (*Cn 3 r-pdec1* [*r-leap*]) [*t*, *i*, *x*] ↓≠ *0*
    using ‹*ss* ≠ []› *r-leap-prim encode-config r-leap-total list-encode-0* **by** *auto*
  then show *?thesis* **unfolding** *r-result-def* **using** *r-leap-prim* **by** *auto*
qed

lemma *r-result-Suc*:
  assumes *i* = *encode f*
    and *recfn* (*e-length x*) *f*
    and *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
  shows *iterate t step* ([(*f*, *list-decode x*, [])], *None*) = ([], *Some v*)
    (**is** *?cfg* = -)
proof (*cases fst ?cfg*)
  case *Nil*
  then show *?thesis*
    using *assms r-result-empty-stack-None r-result-empty-stack-Some*
    by (*metis Zero-not-Suc nat.inject option.collapse option.inject prod.exhaust-sel*)
next

```
  case Cons
  then show ?thesis using assms r-result-nonempty-stack by simp
qed

lemma r-result-converg:
  assumes i = encode f
    and recfn (e-length x) f
    and eval f (list-decode x) ↓= v
  shows ∃ t.
    (∀ t'≥t. eval r-result [t', i, x] ↓= Suc v) ∧
    (∀ t'<t. eval r-result [t', i, x] ↓= 0)
proof –
  let ?xs = list-decode x
  let ?stack = [(f, ?xs, [])]
  have wellf f using assms(2) by simp
  moreover have length ?xs = arity f
    using assms(2) by simp
  ultimately have correct (?stack, None)
    using step-correct valid-def by simp
  with assms(3) have reachable (?stack, None) ([], Some v)
    by simp
  then obtain t where
    iterate t step (?stack, None) = ([], Some v)
    ∀ t'<t. fst (iterate t' step (?stack, None)) ≠ []
    using reachable-iterate-step-empty-stack by blast
  then have t:
    eval r-result [t, i, x] ↓= Suc v
    ∀ t'<t. eval r-result [t', i, x] ↓= 0
    using r-result-empty-stack-Some r-result-nonempty-stack assms(1,2)
    by simp-all
  then have eval r-result [t + t', i, x] ↓= Suc v for t'
    using r-result-empty-stack-stays assms r-result-Suc by simp
  then have ∀ t'≥t. eval r-result [t', i, x] ↓= Suc v
    using le-Suc-ex by blast
  with t(2) show ?thesis by auto
qed

lemma r-result-diverg:
  assumes i = encode f
    and recfn (e-length x) f
    and eval f (list-decode x) ↑
  shows eval r-result [t, i, x] ↓= 0
proof –
  let ?xs = list-decode x
  let ?stack = [(f, ?xs, [])]
  have recfn (length ?xs) f
    using assms(2) by auto
  then have correct (?stack, None)
    using step-correct valid-def by simp
  with assms(3) have nonterminating (?stack, None)
    by simp
  then show ?thesis
    using r-result-nonempty-stack assms(1,2) by simp
qed
```

Now we can define the universal partial recursive function. This function executes *r-result* for increasing time bounds, waits for it to reach a final configuration, and then extracts its result value. If no final configuration is reached, the universal function diverges.

definition *r-univ* ≡
  Cn 2 r-dec [*Cn 2 r-result* [*Mn 2* (*Cn 3 r-not* [*r-result*]), *Id 2 0*, *Id 2 1*]]

lemma *r-univ-recfn* [*simp*]: *recfn 2 r-univ*
  unfolding *r-univ-def* **by** *simp*

theorem *r-univ*:
  assumes *i* = *encode f* **and** *recfn* (*e-length x*) *f*
  shows *eval r-univ* [*i*, *x*] = *eval f* (*list-decode x*)
proof −
  let *?cond* = *Cn 3 r-not* [*r-result*]
  let *?while* = *Mn 2 ?cond*
  let *?res* = *Cn 2 r-result* [*?while*, *Id 2 0*, *Id 2 1*]
  let *?xs* = *list-decode x*
  have ∗: *eval ?cond* [*t*, *i*, *x*] ↓= (*if eval r-result* [*t*, *i*, *x*] ↓= *0 then 1 else 0*) **for** *t*
  proof −
    have *eval ?cond* [*t*, *i*, *x*] = *eval r-not* [*the* (*eval r-result* [*t*, *i*, *x*])]
      using *r-result-total* **by** *simp*
    moreover have *eval r-result* [*t*, *i*, *x*] ↓
      by (*simp add*: *r-result-total*)
    ultimately show *?thesis* **by** *auto*
  qed
  show *?thesis*
  proof (*cases eval f ?xs* ↑)
    case *True*
    then show *?thesis*
      unfolding *r-univ-def* **using** ∗ *r-result-diverg*[*OF assms*] *eval-Mn-diverg* **by** *simp*
  next
    case *False*
    then obtain *v* **where** *v*: *eval f ?xs* ↓= *v* **by** *auto*
    then obtain *t* **where** *t*:
      ∀ *t'*≥*t*. *eval r-result* [*t'*, *i*, *x*] ↓= *Suc v*
      ∀ *t'*<*t*. *eval r-result* [*t'*, *i*, *x*] ↓= *0*
      using *r-result-converg*[*OF assms*] **by** *blast*
    then have
      ∀ *t'*≥*t*. *eval ?cond* [*t'*, *i*, *x*] ↓= *0*
      ∀ *t'*<*t*. *eval ?cond* [*t'*, *i*, *x*] ↓= *1*
      using ∗ **by** *simp-all*
    then have *eval ?while* [*i*, *x*] ↓= *t*
      using *eval-Mn-convergI*[*of 2 ?cond* [*i*, *x*] *t*] **by** *simp*
    then have *eval ?res* [*i*, *x*] = *eval r-result* [*t*, *i*, *x*]
      by *simp*
    then have *eval ?res* [*i*, *x*] ↓= *Suc v*
      using *t*(*1*) **by** *simp*
    then show *?thesis*
      unfolding *r-univ-def* **using** *v* **by** *simp*
  qed
qed

theorem *r-univ'*:
  assumes *recfn* (*e-length x*) *f*
  shows *eval r-univ* [*encode f*, *x*] = *eval f* (*list-decode x*)
  using *r-univ assms* **by** *simp*

Universal functions for every arity can be built from *r-univ*.

definition *r-universal* :: *nat* ⇒ *recf* **where**
r-universal n ≡ *Cn* (*Suc n*) *r-univ* [*Id* (*Suc n*) *0*, *r-shift* (*r-list-encode* (*n* − *1*))]

lemma *r-universal-recfn* [*simp*]: *n* > *0* ⟹ *recfn* (*Suc n*) (*r-universal n*)
unfolding *r-universal-def* **by** *simp*

lemma *r-universal*:
assumes *recfn n f* **and** *length xs* = *n*
shows *eval* (*r-universal n*) (*encode f* # *xs*) = *eval f xs*
unfolding *r-universal-def* **using** *wellf-arity-nonzero assms r-list-encode r-univ′*
by *fastforce*

We will mostly be concerned with computing unary functions. Hence we introduce separate functions for this case.

definition *r-result1* ≡
Cn 3 r-result [*Id 3 0*, *Id 3 1*, *Cn 3 r-singleton-encode* [*Id 3 2*]]

lemma *r-result1-prim* [*simp*]: *prim-recfn 3 r-result1*
unfolding *r-result1-def* **by** *simp*

lemma *r-result1-total*: *total r-result1*
using *Mn-free-imp-total* **by** *simp*

lemma *r-result1* [*simp*]:
eval r-result1 [*t*, *i*, *x*] = *eval r-result* [*t*, *i*, *singleton-encode x*]
unfolding *r-result1-def* **by** *simp*

The following function will be our standard Gödel numbering of all unary partial recursive functions.

definition *r-phi* ≡ *r-universal 1*

lemma *r-phi-recfn* [*simp*]: *recfn 2 r-phi*
unfolding *r-phi-def* **by** *simp*

theorem *r-phi*:
assumes *i* = *encode f* **and** *recfn 1 f*
shows *eval r-phi* [*i*, *x*] = *eval f* [*x*]
unfolding *r-phi-def* **using** *r-universal assms* **by** *force*

corollary *r-phi′*:
assumes *recfn 1 f*
shows *eval r-phi* [*encode f*, *x*] = *eval f* [*x*]
using *assms r-phi* **by** *simp*

lemma *r-phi′′*: *eval r-phi* [*i*, *x*] = *eval r-univ* [*i*, *singleton-encode x*]
unfolding *r-universal-def r-phi-def* **using** *r-list-encode* **by** *simp*

1.6 Applications of the universal function

In this section we shall see some ways *r-univ* and *r-result* can be used.

1.6.1 Lazy conditional evaluation

With the help of *r-univ* we can now define a lazy variant of *r-ifz*, in which only one branch is evaluated.

```
definition r-lazyifzero :: nat ⇒ nat ⇒ nat ⇒ recf where
  r-lazyifzero n j₁ j₂ ≡
     Cn (Suc (Suc n)) r-univ
      [Cn (Suc (Suc n)) r-ifz [Id (Suc (Suc n)) 0, r-constn (Suc n) j₁, r-constn (Suc n) j₂],
       r-shift (r-list-encode n)]

lemma r-lazyifzero-recfn: recfn (Suc (Suc n)) (r-lazyifzero n j₁ j₂)
  using r-lazyifzero-def by simp

lemma r-lazyifzero:
  assumes length xs = Suc n
    and j₁ = encode f₁
    and j₂ = encode f₂
    and recfn (Suc n) f₁
    and recfn (Suc n) f₂
  shows eval (r-lazyifzero n j₁ j₂) (c # xs) = (if c = 0 then eval f₁ xs else eval f₂ xs)
proof −
  let ?a = r-constn (Suc n) n
  let ?b = Cn (Suc (Suc n)) r-ifz
    [Id (Suc (Suc n)) 0, r-constn (Suc n) j₁, r-constn (Suc n) j₂]
  let ?c = r-shift (r-list-encode n)
  have eval ?a (c # xs) ↓= n
    using assms(1) by simp
  moreover have eval ?b (c # xs) ↓= (if c = 0 then j₁ else j₂)
    using assms(1) by simp
  moreover have eval ?c (c # xs) ↓= list-encode xs
    using assms(1) r-list-encode r-shift by simp
  ultimately have eval (r-lazyifzero n j₁ j₂) (c # xs) =
      eval r-univ [if c = 0 then j₁ else j₂, list-encode xs]
    unfolding r-lazyifzero-def using r-lazyifzero-recfn assms(1) by simp
  then show ?thesis using assms r-univ by simp
qed

definition r-lifz :: recf ⇒ recf ⇒ recf where
  r-lifz f g ≡ r-lazyifzero (arity f − 1) (encode f) (encode g)

lemma r-lifz-recfn [simp]:
  assumes recfn n f and recfn n g
  shows recfn (Suc n) (r-lifz f g)
  using assms r-lazyifzero-recfn r-lifz-def wellf-arity-nonzero by auto

lemma r-lifz [simp]:
  assumes length xs = n and recfn n f and recfn n g
  shows eval (r-lifz f g) (c # xs) = (if c = 0 then eval f xs else eval g xs)
  using assms r-lazyifzero r-lifz-def wellf-arity-nonzero
  by (metis One-nat-def Suc-pred)
```

1.6.2 Enumerating the domains of partial recursive functions

In this section we define a binary function *enumdom* such that for all i, the domain of φ_i equals $\{enumdom(i, x) \mid enumdom(i, x)\downarrow\}$. In other words, the image of $enumdom_i$

is the domain of φ_i.

First we need some more properties of *r-leap* and *r-result*.

lemma *r-leap-Suc*: *eval r-leap* [*Suc t*, *i*, *x*] = *eval r-step* [*the* (*eval r-leap* [*t*, *i*, *x*])]
proof −
  have *eval r-leap* [*Suc t*, *i*, *x*] =
      eval (*Cn 4 r-step* [*Id 4 1*]) [*t*, *the* (*eval r-leap* [*t*, *i*, *x*]), *i*, *x*]
    using *r-leap-total eval-Pr-converg-Suc r-leap-def*
    by (*metis length-Cons list.size*(*3*) *numeral-2-eq-2 numeral-3-eq-3 r-leap-prim*)
  then show *?thesis* **using** *r-step-prim* **by** *auto*
qed

lemma *r-leap-Suc-saturating*:
  assumes *pdec1* (*the* (*eval r-leap* [*t*, *i*, *x*])) = *0*
  shows *eval r-leap* [*Suc t*, *i*, *x*] = *eval r-leap* [*t*, *i*, *x*]
proof −
  let *?e* = *eval r-leap* [*t*, *i*, *x*]
  have *eval r-step* [*the ?e*] ↓= *estep* (*the ?e*)
    using *r-step* **by** *simp*
  then have *eval r-step* [*the ?e*] ↓= *prod-encode* (*0*, *e2rv* (*the ?e*))
    using *estep-def assms* **by** (*simp add*: *e2stack-def*)
  then have *eval r-step* [*the ?e*] ↓= *prod-encode* (*pdec1* (*the ?e*), *pdec2* (*the ?e*))
    using *assms* **by** (*simp add*: *e2rv-def*)
  then have *eval r-step* [*the ?e*] ↓= *the ?e* **by** *simp*
  then show *?thesis* **using** *r-leap-total r-leap-Suc* **by** *simp*
qed

lemma *r-result-Suc-saturating*:
  assumes *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
  shows *eval r-result* [*Suc t*, *i*, *x*] ↓= *Suc v*
proof −
  let *?r* = λ*t*. *eval r-ifz* [*pdec1* (*the* (*eval r-leap* [*t*, *i*, *x*])), *pdec2* (*the* (*eval r-leap* [*t*, *i*, *x*])), *0*]
  have *?r t* ↓= *Suc v*
    using *assms* **unfolding** *r-result-def* **using** *r-leap-total r-leap-prim* **by** *simp*
  then have *pdec1* (*the* (*eval r-leap* [*t*, *i*, *x*])) = *0*
    using *option.sel* **by** *fastforce*
  then have *eval r-leap* [*Suc t*, *i*, *x*] = *eval r-leap* [*t*, *i*, *x*]
    using *r-leap-Suc-saturating* **by** *simp*
  moreover have *eval r-result* [*t*, *i*, *x*] = *?r t*
    unfolding *r-result-def* **using** *r-leap-total r-leap-prim* **by** *simp*
  moreover have *eval r-result* [*Suc t*, *i*, *x*] = *?r* (*Suc t*)
    unfolding *r-result-def* **using** *r-leap-total r-leap-prim* **by** *simp*
  ultimately have *eval r-result* [*Suc t*, *i*, *x*] = *eval r-result* [*t*, *i*, *x*]
    by *simp*
  with *assms* **show** *?thesis* **by** *simp*
qed

lemma *r-result-saturating*:
  assumes *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
  shows *eval r-result* [*t* + *d*, *i*, *x*] ↓= *Suc v*
  using *r-result-Suc-saturating assms* **by** (*induction d*) *simp-all*

lemma *r-result-converg'*:
  assumes *eval r-univ* [*i*, *x*] ↓= *v*
  shows ∃ *t*. (∀ *t'*≥*t*. *eval r-result* [*t'*, *i*, *x*] ↓= *Suc v*) ∧ (∀ *t'*<*t*. *eval r-result* [*t'*, *i*, *x*] ↓= *0*)
proof −

let *?f* = *Cn 3 r-not* [*r-result*]
let *?m* = *Mn 2 ?f*
have *recfn 2 ?m* **by** *simp*
have *eval-m*: *eval ?m* [*i*, *x*] ↓
proof
  assume *eval ?m* [*i*, *x*] ↑
  then have *eval r-univ* [*i*, *x*] ↑
    unfolding *r-univ-def* **by** *simp*
  with *assms* **show** *False* **by** *simp*
qed
then obtain *t* **where** *t*: *eval ?m* [*i*, *x*] ↓= *t*
  by *auto*
then have *f-t*: *eval ?f* [*t*, *i*, *x*] ↓= *0* **and** *f-less-t*: ⋀*y*. *y* < *t* ⟹ *eval ?f* [*y*, *i*, *x*] ↓≠ *0*
  using *eval-Mn-convergE*[*of 2 ?f* [*i*, *x*] *t*] ‹*recfn 2 ?m*›
  by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-1 length-Cons list.size*(*3*))+
have *eval-Cn2*: *eval* (*Cn 2 r-result* [*?m*, *Id 2 0*, *Id 2 1*]) [*i*, *x*] ↓
proof
  assume *eval* (*Cn 2 r-result* [*?m*, *Id 2 0*, *Id 2 1*]) [*i*, *x*] ↑
  then have *eval r-univ* [*i*, *x*] ↑
    unfolding *r-univ-def* **by** *simp*
  with *assms* **show** *False* **by** *simp*
qed
have *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
proof (*rule ccontr*)
  assume *neq-Suc*: ¬ *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
  show *False*
  proof (*cases eval r-result* [*t*, *i*, *x*] = *None*)
    case *True*
    then show *?thesis* **using** *f-t* **by** *simp*
  next
    case *False*
    then obtain *w* **where** *w*: *eval r-result* [*t*, *i*, *x*] ↓= *w w* ≠ *Suc v*
      using *neq-Suc* **by** *auto*
    moreover have *eval r-result* [*t*, *i*, *x*] ↓≠ *0*
      by (*rule ccontr*; *use f-t* **in** *auto*)
    ultimately have *w* ≠ *0* **by** *simp*
    have *eval* (*Cn 2 r-result* [*?m*, *Id 2 0*, *Id 2 1*]) [*i*, *x*] =
        eval r-result [*the* (*eval ?m* [*i*, *x*]), *i*, *x*]
      using *eval-m* **by** *simp*
    with *w t* **have** *eval* (*Cn 2 r-result* [*?m*, *Id 2 0*, *Id 2 1*]) [*i*, *x*] ↓= *w*
      by *simp*
    moreover have *eval r-univ* [*i*, *x*] =
        eval r-dec [*the* (*eval* (*Cn 2 r-result* [*?m*, *Id 2 0*, *Id 2 1*]) [*i*, *x*])]
      unfolding *r-univ-def* **using** *eval-Cn2* **by** *simp*
    ultimately have *eval r-univ* [*i*, *x*] = *eval r-dec* [*w*] **by** *simp*
    then have *eval r-univ* [*i*, *x*] ↓= *w* − *1* **by** *simp*
    with *assms* ‹*w* ≠ *0*› *w* **show** *?thesis* **by** *simp*
  qed
qed
then have ∀ *t'*≥*t*. *eval r-result* [*t'*, *i*, *x*] ↓= *Suc v*
  using *r-result-saturating le-Suc-ex* **by** *blast*
moreover have *eval r-result* [*y*, *i*, *x*] ↓= *0* **if** *y* < *t* **for** *y*
proof (*rule ccontr*)
  assume *neq0*: *eval r-result* [*y*, *i*, *x*] ≠ *Some 0*
  then show *False*
  proof (*cases eval r-result* [*y*, *i*, *x*] = *None*)

case *True*
then show *?thesis* **using** *f-less-t* ‹$y < t$› **by** *fastforce*
next
case *False*
then obtain *v* **where** *eval r-result* [*y*, *i*, *x*] ↓= *v v* ≠ *0*
using *neq0* **by** *auto*
then have *eval ?f* [*y*, *i*, *x*] ↓= *0* **by** *simp*
then show *?thesis* **using** *f-less-t* ‹$y < t$› **by** *simp*
qed
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *r-result-diverg*′:
assumes *eval r-univ* [*i*, *x*] ↑
shows *eval r-result* [*t*, *i*, *x*] ↓= *0*
proof (*rule ccontr*)
let *?f* = *Cn 3 r-not* [*r-result*]
let *?m* = *Mn 2 ?f*
assume *eval r-result* [*t*, *i*, *x*] ≠ *Some 0*
with *r-result-total* **have** *eval r-result* [*t*, *i*, *x*] ↓≠ *0* **by** *simp*
then have *eval ?f* [*t*, *i*, *x*] ↓= *0* **by** *auto*
moreover have *eval ?f* [*y*, *i*, *x*] ↓ **if** $y < t$ **for** *y*
using *r-result-total* **by** *simp*
ultimately have ∃ *z*. *eval ?f* (*z* # [*i*, *x*]) ↓= *0* ∧ (∀ *y*<*z*. *eval ?f* (*y* # [*i*, *x*]) ↓)
by *blast*
then have *eval ?m* [*i*, *x*] ↓ **by** *simp*
then have *eval r-univ* [*i*, *x*] ↓
unfolding *r-univ-def* **using** *r-result-total* **by** *simp*
with *assms* **show** *False* **by** *simp*
qed

lemma *r-result-bivalent*′:
assumes *eval r-univ* [*i*, *x*] ↓= *v*
shows *eval r-result* [*t*, *i*, *x*] ↓= *Suc v* ∨ *eval r-result* [*t*, *i*, *x*] ↓= *0*
using *r-result-converg*′[*OF assms*] *not-less* **by** *blast*

lemma *r-result-Some*′:
assumes *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
shows *eval r-univ* [*i*, *x*] ↓= *v*
proof (*rule ccontr*)
assume *not-v*: ¬ *eval r-univ* [*i*, *x*] ↓= *v*
show *False*
proof (*cases eval r-univ* [*i*, *x*] ↑)
case *True*
then show *?thesis*
using *assms r-result-diverg*′ **by** *simp*
next
case *False*
then obtain *w* **where** *w*: *eval r-univ* [*i*, *x*] ↓= *w w* ≠ *v*
using *not-v* **by** *auto*
then have *eval r-result* [*t*, *i*, *x*] ↓= *Suc w* ∨ *eval r-result* [*t*, *i*, *x*] ↓= *0*
using *r-result-bivalent*′ **by** *simp*
then show *?thesis* **using** *assms not-v w* **by** *simp*
qed
qed

lemma *r-result1-converg′*:
assumes *eval r-phi* [*i*, *x*] ↓= *v*
shows ∃ *t*.
(∀ *t′*≥*t*. *eval r-result1* [*t′*, *i*, *x*] ↓= *Suc v*) ∧
(∀ *t′*<*t*. *eval r-result1* [*t′*, *i*, *x*] ↓= *0*)
using *assms r-result1 r-result-converg′ r-phi′′* **by** *simp*

lemma *r-result1-diverg′*:
assumes *eval r-phi* [*i*, *x*] ↑
shows *eval r-result1* [*t*, *i*, *x*] ↓= *0*
using *assms r-result1 r-result-diverg′ r-phi′′* **by** *simp*

lemma *r-result1-Some′*:
assumes *eval r-result1* [*t*, *i*, *x*] ↓= *Suc v*
shows *eval r-phi* [*i*, *x*] ↓= *v*
using *assms r-result1 r-result-Some′ r-phi′′* **by** *simp*

The next function performs dovetailing in order to evaluate φ_i for every argument for arbitrarily many steps. Given i and z, the function decodes z into a pair (x, t) and outputs zero (meaning "true") iff. the computation of φ_i on input x halts after at most t steps. Fixing i and varying z will eventually compute φ_i for every argument in the domain of φ_i sufficiently long for it to converge.

definition *r-dovetail* ≡
Cn 2 r-not [*Cn 2 r-result1* [*Cn 2 r-pdec2* [*Id 2 1*], *Id 2 0*, *Cn 2 r-pdec1* [*Id 2 1*]]]

lemma *r-dovetail-prim*: *prim-recfn 2 r-dovetail*
by (*simp add*: *r-dovetail-def*)

lemma *r-dovetail*:
eval r-dovetail [*i*, *z*] ↓=
(*if the* (*eval r-result1* [*pdec2 z*, *i*, *pdec1 z*]) > *0 then 0 else 1*)
unfolding *r-dovetail-def* **using** *r-result-total* **by** *simp*

The function *enumdom* works as follows in order to enumerate exactly the domain of φ_i. Given i and y it searches for the minimum $z \geq y$ for which the dovetail function returns true. This z is decoded into (x, t) and the x is output. In this way every value output by *enumdom* is in the domain of φ_i by construction of *r-dovetail*. Conversely an x in the domain will be output for $y = (x, t)$ where t is such that φ_i halts on x within t steps.

definition *r-dovedelay* ≡
Cn 3 r-and
[*Cn 3 r-dovetail* [*Id 3 1*, *Id 3 0*],
Cn 3 r-ifle [*Id 3 2*, *Id 3 0*, *r-constn 2 0*, *r-constn 2 1*]]

lemma *r-dovedelay-prim*: *prim-recfn 3 r-dovedelay*
unfolding *r-dovedelay-def* **using** *r-dovetail-prim* **by** *simp*

lemma *r-dovedelay*:
eval r-dovedelay [*z*, *i*, *y*] ↓=
(*if the* (*eval r-result1* [*pdec2 z*, *i*, *pdec1 z*]) > *0* ∧ *y* ≤ *z then 0 else 1*)
by (*simp add*: *r-dovedelay-def r-dovetail r-dovetail-prim*)

definition *r-enumdom* ≡ *Cn 2 r-pdec1* [*Mn 2 r-dovedelay*]

lemma *r-enumdom-recfn* [*simp*]: *recfn 2 r-enumdom*
  by (*simp add*: *r-enumdom-def r-dovedelay-prim*)

lemma *r-enumdom* [*simp*]:
  eval r-enumdom [*i*, *y*] =
    (*if* ∃ *z*. *eval r-dovedelay* [*z*, *i*, *y*] ↓= *0*
     then Some (*pdec1* (*LEAST z*. *eval r-dovedelay* [*z*, *i*, *y*] ↓= *0*))
     else None)
proof −
  let *?h* = *Mn 2 r-dovedelay*
  have *total r-dovedelay*
    using *r-dovedelay-prim* **by** *blast*
  then have *eval ?h* [*i*, *y*] =
    (*if* (∃ *z*. *eval r-dovedelay* [*z*, *i*, *y*] ↓= *0*)
     then Some (*LEAST z*. *eval r-dovedelay* [*z*, *i*, *y*] ↓= *0*)
     else None)
    using *r-dovedelay-prim r-enumdom-recfn eval-Mn-convergI* **by** *simp*
  then show *?thesis*
    unfolding *r-enumdom-def* **using** *r-dovedelay-prim* **by** *simp*
qed

If *i* is the code of the empty function, *r-enumdom* has an empty domain, too.

lemma *r-enumdom-empty-domain*:
  assumes ⋀*x*. *eval r-phi* [*i*, *x*] ↑
  shows ⋀*y*. *eval r-enumdom* [*i*, *y*] ↑
  using *assms r-result1-diverg′ r-dovedelay* **by** *simp*

If *i* is the code of a function with non-empty domain, *r-enumdom* enumerates its domain.

lemma *r-enumdom-nonempty-domain*:
  assumes *eval r-phi* [*i*, x_0] ↓
  shows ⋀*y*. *eval r-enumdom* [*i*, *y*] ↓
    and ⋀*x*. *eval r-phi* [*i*, *x*] ↓ ⟷ (∃ *y*. *eval r-enumdom* [*i*, *y*] ↓= *x*)
proof −
  show *eval r-enumdom* [*i*, *y*] ↓ **for** *y*
  proof −
    obtain *t* **where** *t*: ∀ *t′*≥*t*. *the* (*eval r-result1* [*t′*, *i*, x_0]) > *0*
      using *assms r-result1-converg′* **by** *fastforce*
    let *?z* = *prod-encode* (x_0, *max t y*)
    have *y* ≤ *?z*
      using *le-prod-encode-2 max.bounded-iff* **by** *blast*
    moreover have *pdec2 ?z* ≥ *t* **by** *simp*
    ultimately have *the* (*eval r-result1* [*pdec2 ?z*, *i*, *pdec1 ?z*]) > *0*
      using *t* **by** *simp*
    with ‹*y* ≤ *?z*› *r-dovedelay* **have** *eval r-dovedelay* [*?z*, *i*, *y*] ↓= *0*
      by *presburger*
    then show *eval r-enumdom* [*i*, *y*] ↓
      using *r-enumdom* **by** *auto*
  qed
  show *eval r-phi* [*i*, *x*] ↓ = (∃ *y*. *eval r-enumdom* [*i*, *y*] ↓= *x*) **for** *x*
  proof
    show ∃ *y*. *eval r-enumdom* [*i*, *y*] ↓= *x* **if** *eval r-phi* [*i*, *x*] ↓ **for** *x*
    proof −
      from *that* **obtain** *v* **where** *eval r-phi* [*i*, *x*] ↓= *v* **by** *auto*
      then obtain *t* **where** *t*: *the* (*eval r-result1* [*t*, *i*, *x*]) > *0*
        using *r-result1-converg′ assms*

```
        by (metis Zero-not-Suc dual-order.refl option.sel zero-less-iff-neq-zero)
      let ?y = prod-encode (x, t)
      have eval r-dovedelay [?y, i, ?y] ↓= 0
        using r-dovedelay t by simp
      moreover from this have (LEAST z. eval r-dovedelay [z, i, ?y] ↓= 0) = ?y
        using gr-implies-not-zero r-dovedelay by (intro Least-equality; fastforce)
      ultimately have eval r-enumdom [i, ?y] ↓= x
        using r-enumdom by auto
      then show ?thesis by blast
    qed
    show eval r-phi [i, x] ↓ if ∃ y. eval r-enumdom [i, y] ↓= x for x
    proof −
      from that obtain y where y: eval r-enumdom [i, y] ↓= x
        by auto
      then have eval r-enumdom [i, y] ↓
        by simp
      then have
        ∃ z. eval r-dovedelay [z, i, y] ↓= 0 and
        ∗: eval r-enumdom [i, y] ↓= pdec1 (LEAST z. eval r-dovedelay [z, i, y] ↓= 0)
          (is - ↓= pdec1 ?z)
        using r-enumdom by metis+
      then have z: eval r-dovedelay [?z, i, y] ↓= 0
        by (meson wellorder-Least-lemma(1))
      have the (eval r-result1 [pdec2 ?z, i, pdec1 ?z]) > 0
      proof (rule ccontr)
        assume ¬ (the (eval r-result1 [pdec2 ?z, i, pdec1 ?z]) > 0)
        then show False
          using r-dovedelay z by simp
      qed
      then have eval r-phi [i, pdec1 ?z] ↓
        using r-result1-diverg' assms by fastforce
      then show ?thesis using y ∗ by auto
    qed
  qed
qed
```

For every φ_i with non-empty domain there is a total recursive function that enumerates the domain of φ_i.

```
lemma nonempty-domain-enumerable:
  assumes eval r-phi [i, x0] ↓
  shows ∃ g. recfn 1 g ∧ total g ∧ (∀ x. eval r-phi [i, x] ↓ ←→ (∃ y. eval g [y] ↓= x))
proof −
  define g where g ≡ Cn 1 r-enumdom [r-const i, Id 1 0]
  then have recfn 1 g by simp
  moreover from this have total g
    using totalI1[of g] g-def assms r-enumdom-nonempty-domain(1) by simp
  moreover have eval r-phi [i, x] ↓ ←→ (∃ y. eval g [y] ↓= x) for x
    unfolding g-def using r-enumdom-nonempty-domain(2)[OF assms] by simp
  ultimately show ?thesis by auto
qed
```

1.6.3 Concurrent evaluation of functions

We define a function that simulates two *recf*s "concurrently" for the same argument and returns the result of the one converging first. If both diverge, so does the simulation function.

```
definition r-both ≡
  Cn 4 r-ifz
   [Cn 4 r-result1 [Id 4 0, Id 4 1, Id 4 3],
    Cn 4 r-ifz
     [Cn 4 r-result1 [Id 4 0, Id 4 2, Id 4 3],
      Cn 4 r-prod-encode [r-constn 3 2, r-constn 3 0],
      Cn 4 r-prod-encode
       [r-constn 3 1, Cn 4 r-dec [Cn 4 r-result1 [Id 4 0, Id 4 2, Id 4 3]]]],
    Cn 4 r-prod-encode
     [r-constn 3 0, Cn 4 r-dec [Cn 4 r-result1 [Id 4 0, Id 4 1, Id 4 3]]]]

lemma r-both-prim [simp]: prim-recfn 4 r-both
  unfolding r-both-def by simp

lemma r-both:
  assumes ⋀x. eval r-phi [i, x] = eval f [x]
    and ⋀x. eval r-phi [j, x] = eval g [x]
  shows eval f [x] ↑ ∧ eval g [x] ↑ ⟹ eval r-both [t, i, j, x] ↓= prod-encode (2, 0)
    and ⟦eval r-result1 [t, i, x] ↓= 0; eval r-result1 [t, j, x] ↓= 0⟧ ⟹
      eval r-both [t, i, j, x] ↓= prod-encode (2, 0)
    and eval r-result1 [t, i, x] ↓= Suc v ⟹
      eval r-both [t, i, j, x] ↓= prod-encode (0, the (eval f [x]))
    and ⟦eval r-result1 [t, i, x] ↓= 0; eval r-result1 [t, j, x] ↓= Suc v⟧ ⟹
      eval r-both [t, i, j, x] ↓= prod-encode (1, the (eval g [x]))
proof –
  have r-result-total [simp]: eval r-result [t, k, x] ↓ for t k x
    using r-result-total by simp
  {
    assume eval f [x] ↑ ∧ eval g [x] ↑
    then have eval r-result1 [t, i, x] ↓= 0 and eval r-result1 [t, j, x] ↓= 0
      using assms r-result1-diverg' by auto
    then show eval r-both [t, i, j, x] ↓= prod-encode (2, 0)
      unfolding r-both-def by simp
  next
    assume eval r-result1 [t, i, x] ↓= 0 and eval r-result1 [t, j, x] ↓= 0
    then show eval r-both [t, i, j, x] ↓= prod-encode (2, 0)
      unfolding r-both-def by simp
  next
    assume eval r-result1 [t, i, x] ↓= Suc v
    moreover from this have eval r-result1 [t, i, x] ↓= Suc (the (eval f [x]))
      using assms r-result1-Some' by fastforce
    ultimately show eval r-both [t, i, j, x] ↓= prod-encode (0, the (eval f [x]))
      unfolding r-both-def by auto
  next
    assume eval r-result1 [t, i, x] ↓= 0 and eval r-result1 [t, j, x] ↓= Suc v
    moreover from this have eval r-result1 [t, j, x] ↓= Suc (the (eval g [x]))
      using assms r-result1-Some' by fastforce
    ultimately show eval r-both [t, i, j, x] ↓= prod-encode (1, the (eval g [x]))
      unfolding r-both-def by auto
  }
qed

definition r-parallel ≡
  Cn 3 r-both [Mn 3 (Cn 4 r-le [Cn 4 r-pdec1 [r-both], r-constn 3 1]), Id 3 0, Id 3 1, Id 3 2]
```

```
lemma r-parallel-recfn [simp]: recfn 3 r-parallel
  unfolding r-parallel-def by simp

lemma r-parallel:
  assumes ⋀x. eval r-phi [i, x] = eval f [x]
    and ⋀x. eval r-phi [j, x] = eval g [x]
  shows eval f [x] ↑ ∧ eval g [x] ↑ ⟹ eval r-parallel [i, j, x] ↑
    and eval f [x] ↓ ∧ eval g [x] ↑ ⟹
      eval r-parallel [i, j, x] ↓= prod-encode (0, the (eval f [x]))
    and eval g [x] ↓ ∧ eval f [x] ↑ ⟹
      eval r-parallel [i, j, x] ↓= prod-encode (1, the (eval g [x]))
    and eval f [x] ↓ ∧ eval g [x] ↓ ⟹
      eval r-parallel [i, j, x] ↓= prod-encode (0, the (eval f [x])) ∨
      eval r-parallel [i, j, x] ↓= prod-encode (1, the (eval g [x]))
proof –
  let ?cond = Cn 4 r-le [Cn 4 r-pdec1 [r-both], r-constn 3 1]
  define m where m = Mn 3 ?cond
  then have m: r-parallel = Cn 3 r-both [m, Id 3 0, Id 3 1, Id 3 2]
    unfolding r-parallel-def by simp
  from m-def have recfn 3 m by simp
  {
    assume eval f [x] ↑ ∧ eval g [x] ↑
    then have ∀t. eval r-both [t, i, j, x] ↓= prod-encode (2, 0)
      using assms r-both by simp
    then have eval ?cond [t, i, j, x] ↓= 1 for t
      by simp
    then have eval m [i, j, x] ↑
      unfolding m-def using eval-Mn-diverg by simp
    then have eval (Cn 3 r-both [m, Id 3 0, Id 3 1, Id 3 2]) [i, j, x] ↑
      using ‹recfn 3 m› by simp
    then show eval r-parallel [i, j, x] ↑
      using m by simp
  next
    assume eval f [x] ↓ ∧ eval g [x] ↓
    then obtain vf vg where v: eval f [x] ↓= vf eval g [x] ↓= vg
      by auto
    then obtain tf where tf:
      ∀t≥tf. eval r-result1 [t, i, x] ↓= Suc vf
      ∀t<tf. eval r-result1 [t, i, x] ↓= 0
      using r-result1-converg' assms by metis
    from v obtain tg where tg:
      ∀t≥tg. eval r-result1 [t, j, x] ↓= Suc vg
      ∀t<tg. eval r-result1 [t, j, x] ↓= 0
      using r-result1-converg' assms by metis
    show eval r-parallel [i, j, x] ↓= prod-encode (0, the (eval f [x])) ∨
      eval r-parallel [i, j, x] ↓= prod-encode (1, the (eval g [x]))
    proof (cases tf ≤ tg)
      case True
      with tg(2) have j0: ∀t<tf. eval r-result1 [t, j, x] ↓= 0
        by simp
      have *: eval r-both [tf, i, j, x] ↓= prod-encode (0, the (eval f [x]))
        using r-both(3) assms tf(1) by simp
      have eval m [i, j, x] ↓= tf
        unfolding m-def
      proof (rule eval-Mn-convergI)
```

show *recfn* (*length* [*i*, *j*, *x*]) (*Mn 3 ?cond*) **by** *simp*
have *eval* (*Cn 4 r-pdec1* [*r-both*]) [*tf*, *i*, *j*, *x*] ↓= *0*
using ∗ **by** *simp*
then show *eval ?cond* [*tf*, *i*, *j*, *x*] ↓= *0* **by** *simp*
have *eval r-both* [*t*, *i*, *j*, *x*] ↓= *prod-encode* (*2*, *0*) **if** *t* < *tf* **for** *t*
using *tf*(*2*) *r-both*(*2*) *assms that j0* **by** *simp*
then have *eval ?cond* [*t*, *i*, *j*, *x*] ↓= *1* **if** *t* < *tf* **for** *t*
using *that* **by** *simp*
then show ⋀*y*. *y* < *tf* ⟹ *eval ?cond* [*y*, *i*, *j*, *x*] ↓≠ *0* **by** *simp*
qed
moreover have *eval r-parallel* [*i*, *j*, *x*] =
eval (*Cn 3 r-both* [*m*, *Id 3 0*, *Id 3 1*, *Id 3 2*]) [*i*, *j*, *x*]
using *m* **by** *simp*
ultimately have *eval r-parallel* [*i*, *j*, *x*] = *eval r-both* [*tf*, *i*, *j*, *x*]
using ‹*recfn 3 m*› **by** *simp*
with ∗ **have** *eval r-parallel* [*i*, *j*, *x*] ↓= *prod-encode* (*0*, *the* (*eval f* [*x*]))
by *simp*
then show *?thesis* **by** *simp*
next
case *False*
with *tf*(*2*) **have** *i0*: ∀ *t*≤*tg*. *eval r-result1* [*t*, *i*, *x*] ↓= *0*
by *simp*
then have ∗: *eval r-both* [*tg*, *i*, *j*, *x*] ↓= *prod-encode* (*1*, *the* (*eval g* [*x*]))
using *assms r-both*(*4*) *tg*(*1*) **by** *auto*
have *eval m* [*i*, *j*, *x*] ↓= *tg*
unfolding *m-def*
proof (*rule eval-Mn-convergI*)
show *recfn* (*length* [*i*, *j*, *x*]) (*Mn 3 ?cond*) **by** *simp*
have *eval* (*Cn 4 r-pdec1* [*r-both*]) [*tg*, *i*, *j*, *x*] ↓= *1*
using ∗ **by** *simp*
then show *eval ?cond* [*tg*, *i*, *j*, *x*] ↓= *0* **by** *simp*
have *eval r-both* [*t*, *i*, *j*, *x*] ↓= *prod-encode* (*2*, *0*) **if** *t* < *tg* **for** *t*
using *tg*(*2*) *r-both*(*2*) *assms that i0* **by** *simp*
then have *eval ?cond* [*t*, *i*, *j*, *x*] ↓= *1* **if** *t* < *tg* **for** *t*
using *that* **by** *simp*
then show ⋀*y*. *y* < *tg* ⟹ *eval ?cond* [*y*, *i*, *j*, *x*] ↓≠ *0* **by** *simp*
qed
moreover have *eval r-parallel* [*i*, *j*, *x*] =
eval (*Cn 3 r-both* [*m*, *Id 3 0*, *Id 3 1*, *Id 3 2*]) [*i*, *j*, *x*]
using *m* **by** *simp*
ultimately have *eval r-parallel* [*i*, *j*, *x*] = *eval r-both* [*tg*, *i*, *j*, *x*]
using ‹*recfn 3 m*› **by** *simp*
with ∗ **have** *eval r-parallel* [*i*, *j*, *x*] ↓= *prod-encode* (*1*, *the* (*eval g* [*x*]))
by *simp*
then show *?thesis* **by** *simp*
qed
next
assume *eval-fg*: *eval g* [*x*] ↓ ∧ *eval f* [*x*] ↑
then have *i0*: ∀ *t*. *eval r-result1* [*t*, *i*, *x*] ↓= *0*
using *r-result1-diverg′ assms* **by** *auto*
from *eval-fg* **obtain** *v* **where** *eval g* [*x*] ↓= *v*
by *auto*
then obtain t_0 **where** *t0*:
∀ *t*≥t_0. *eval r-result1* [*t*, *j*, *x*] ↓= *Suc v*
∀ *t*<t_0. *eval r-result1* [*t*, *j*, *x*] ↓= *0*
using *r-result1-converg′ assms* **by** *metis*

then have ∗: *eval r-both* [t_0, *i*, *j*, *x*] ↓= *prod-encode* (*1*, *the* (*eval g* [*x*]))
using *r-both*(*4*) *assms i0* **by** *simp*
have *eval m* [*i*, *j*, *x*] ↓= t_0
unfolding *m-def*
proof (*rule eval-Mn-convergI*)
show *recfn* (*length* [*i*, *j*, *x*]) (*Mn 3 ?cond*) **by** *simp*
have *eval* (*Cn 4 r-pdec1* [*r-both*]) [t_0, *i*, *j*, *x*] ↓= *1*
using ∗ **by** *simp*
then show *eval ?cond* [t_0, *i*, *j*, *x*] ↓= *0* **by** *simp*
have *eval r-both* [*t*, *i*, *j*, *x*] ↓= *prod-encode* (*2*, *0*) **if** $t < t_0$ **for** *t*
using *t0*(*2*) *r-both*(*2*) *assms that i0* **by** *simp*
then have *eval ?cond* [*t*, *i*, *j*, *x*] ↓= *1* **if** $t < t_0$ **for** *t*
using *that* **by** *simp*
then show ⋀*y*. $y < t_0$ ⟹ *eval ?cond* [*y*, *i*, *j*, *x*] ↓≠ *0* **by** *simp*
qed
moreover have *eval r-parallel* [*i*, *j*, *x*] =
eval (*Cn 3 r-both* [*m*, *Id 3 0*, *Id 3 1*, *Id 3 2*]) [*i*, *j*, *x*]
using *m* **by** *simp*
ultimately have *eval r-parallel* [*i*, *j*, *x*] = *eval r-both* [t_0, *i*, *j*, *x*]
using ‹*recfn 3 m*› **by** *simp*
with ∗ **show** *eval r-parallel* [*i*, *j*, *x*] ↓= *prod-encode* (*1*, *the* (*eval g* [*x*]))
by *simp*
next
assume *eval-fg*: *eval f* [*x*] ↓ ∧ *eval g* [*x*] ↑
then have *j0*: ∀ *t*. *eval r-result1* [*t*, *j*, *x*] ↓= *0*
using *r-result1-diverg′ assms* **by** *auto*
from *eval-fg* **obtain** *v* **where** *eval f* [*x*] ↓= *v*
by *auto*
then obtain t_0 **where** *t0*:
∀ $t \geq t_0$. *eval r-result1* [*t*, *i*, *x*] ↓= *Suc v*
∀ $t < t_0$. *eval r-result1* [*t*, *i*, *x*] ↓= *0*
using *r-result1-converg′ assms* **by** *metis*
then have ∗: *eval r-both* [t_0, *i*, *j*, *x*] ↓= *prod-encode* (*0*, *the* (*eval f* [*x*]))
using *r-both*(*3*) *assms* **by** *blast*
have *eval m* [*i*, *j*, *x*] ↓= t_0
unfolding *m-def*
proof (*rule eval-Mn-convergI*)
show *recfn* (*length* [*i*, *j*, *x*]) (*Mn 3 ?cond*) **by** *simp*
have *eval* (*Cn 4 r-pdec1* [*r-both*]) [t_0, *i*, *j*, *x*] ↓= *0*
using ∗ **by** *simp*
then show *eval ?cond* [t_0, *i*, *j*, *x*] ↓= *0*
by *simp*
have *eval r-both* [*t*, *i*, *j*, *x*] ↓= *prod-encode* (*2*, *0*) **if** $t < t_0$ **for** *t*
using *t0*(*2*) *r-both*(*2*) *assms that j0* **by** *simp*
then have *eval ?cond* [*t*, *i*, *j*, *x*] ↓= *1* **if** $t < t_0$ **for** *t*
using *that* **by** *simp*
then show ⋀*y*. $y < t_0$ ⟹ *eval ?cond* [*y*, *i*, *j*, *x*] ↓≠ *0* **by** *simp*
qed
moreover have *eval r-parallel* [*i*, *j*, *x*] =
eval (*Cn 3 r-both* [*m*, *Id 3 0*, *Id 3 1*, *Id 3 2*]) [*i*, *j*, *x*]
using *m* **by** *simp*
ultimately have *eval r-parallel* [*i*, *j*, *x*] = *eval r-both* [t_0, *i*, *j*, *x*]
using ‹*recfn 3 m*› **by** *simp*
with ∗ **show** *eval r-parallel* [*i*, *j*, *x*] ↓= *prod-encode* (*0*, *the* (*eval f* [*x*]))
by *simp*
}

qed

end
theory *Standard-Results*
  imports *Universal*
begin

1.7 Kleene normal form and the number of μ-operations

Kleene's original normal form theorem [11] states that every partial recursive f can be expressed as $f(x) = u(\mu y[t(i, x, y) = 0]$ for some i, where u and t are specially crafted primitive recursive functions tied to Kleene's definition of partial recursive functions. Rogers [12, p. 29f.] relaxes the theorem by allowing u and t to be any primitive recursive functions of arity one and three, respectively. Both versions require a separate t-predicate for every arity. We will show a unified version for all arities by treating x as an encoded list of arguments.

Our universal function

r-univ ≡
Cn 2 r-dec [*Cn 2 r-result* [*Mn 2* (*Cn 3 r-not* [*r-result*]), *Id 2 0*, *Id 2 1*]]

can represent all partial recursive functions (see theorem *r-univ*). Moreover *r-result*, *r-dec*, and *r-not* are primitive recursive. As such *r-univ* could almost serve as the right-hand side $u(\mu y[t(i, x, y) = 0]$. Its only flaw is that the outer function, the composition of *r-dec* and *r-result*, is ternary rather than unary.

lemma *r-univ-almost-kleene-nf*:
  r-univ ≃
   (*let u* = *Cn 3 r-dec* [*r-result*];
       t = *Cn 3 r-not* [*r-result*]
    in Cn 2 u [*Mn 2 t*, *Id 2 0*, *Id 2 1*])
  unfolding *r-univ-def* **by** (*rule exteqI*) *simp-all*

We can remedy the wrong arity with some encoding and projecting.

definition *r-nf-t* :: *recf* **where**
  r-nf-t ≡ *Cn 3 r-and*
    [*Cn 3 r-eq* [*Cn 3 r-pdec2* [*Id 3 0*], *Cn 3 r-prod-encode* [*Id 3 1*, *Id 3 2*]],
     Cn 3 r-not
      [*Cn 3 r-result*
        [*Cn 3 r-pdec1* [*Id 3 0*],
         Cn 3 r-pdec12 [*Id 3 0*],
         Cn 3 r-pdec22 [*Id 3 0*]]]]

lemma *r-nf-t-prim*: *prim-recfn 3 r-nf-t*
  unfolding *r-nf-t-def* **by** *simp*

definition *r-nf-u* :: *recf* **where**
  r-nf-u ≡ *Cn 1 r-dec* [*Cn 1 r-result* [*r-pdec1*, *r-pdec12*, *r-pdec22*]]

lemma *r-nf-u-prim*: *prim-recfn 1 r-nf-u*
  unfolding *r-nf-u-def* **by** *simp*

lemma *r-nf-t-0*:
  assumes *eval r-result* [*pdec1 y*, *pdec12 y*, *pdec22 y*] ↓≠ *0*

and *pdec2 y* = *prod-encode* (*i*, *x*)
shows *eval r-nf-t* [*y*, *i*, *x*] ↓= *0*
unfolding *r-nf-t-def* **using** *assms* **by** *auto*

lemma *r-nf-t-1*:
assumes *eval r-result* [*pdec1 y*, *pdec12 y*, *pdec22 y*] ↓= *0* ∨ *pdec2 y* ≠ *prod-encode* (*i*, *x*)
shows *eval r-nf-t* [*y*, *i*, *x*] ↓= *1*
unfolding *r-nf-t-def* **using** *assms r-result-total* **by** *auto*

The next function is just as universal as *r-univ*, but satisfies the conditions of the Kleene normal form theorem because the outer funtion *r-nf-u* is unary.

definition *r-normal-form* ≡ *Cn 2 r-nf-u* [*Mn 2 r-nf-t*]

lemma *r-normal-form-recfn*: *recfn 2 r-normal-form*
unfolding *r-normal-form-def* **using** *r-nf-u-prim r-nf-t-prim* **by** *simp*

lemma *r-univ-exteq-r-normal-form*: *r-univ* ≃ *r-normal-form*
proof (*rule exteqI*)
show *arity*: *arity r-univ* = *arity r-normal-form*
using *r-normal-form-recfn* **by** *simp*
show *eval r-univ xs* = *eval r-normal-form xs* **if** *length xs* = *arity r-univ* **for** *xs*
proof −
have *length xs* = *2*
using *that* **by** *simp*
then obtain *i x* **where** *ix*: [*i*, *x*] = *xs*
by (*smt Suc-length-conv length-0-conv numeral-2-eq-2*)
have *eval r-univ* [*i*, *x*] = *eval r-normal-form* [*i*, *x*]
proof (*cases* ∀ *t*. *eval r-result* [*t*, *i*, *x*] ↓= *0*)
case *True*
then have *eval r-univ* [*i*, *x*] ↑
unfolding *r-univ-def* **by** *simp*
moreover have *eval r-normal-form* [*i*, *x*] ↑
proof −
have *eval r-nf-t* [*y*, *i*, *x*] ↓= *1* **for** *y*
using *True r-nf-t-1*[*of y i x*] **by** *fastforce*
then show *?thesis*
unfolding *r-normal-form-def* **using** *r-nf-u-prim r-nf-t-prim* **by** *simp*
qed
ultimately show *?thesis* **by** *simp*
next
case *False*
then have ∃ *t*. *eval r-result* [*t*, *i*, *x*] ↓≠ *0*
by (*simp add*: *r-result-total*)
then obtain *t* **where** *eval r-result* [*t*, *i*, *x*] ↓≠ *0*
by *auto*
then have *eval r-nf-t* [*triple-encode t i x*, *i*, *x*] ↓= *0*
using *r-nf-t-0* **by** *simp*
then obtain *y* **where** *y*: *eval* (*Mn 2 r-nf-t*) [*i*, *x*] ↓= *y*
using *r-nf-t-prim Mn-free-imp-total* **by** *fastforce*
then have *eval r-nf-t* [*y*, *i*, *x*] ↓= *0*
using *r-nf-t-prim Mn-free-imp-total eval-Mn-convergE*(*2*)[*of 2 r-nf-t* [*i*, *x*] *y*]
by *simp*
then have *r-result*: *eval r-result* [*pdec1 y*, *pdec12 y*, *pdec22 y*] ↓≠ *0*
and *pdec2*: *pdec2 y* = *prod-encode* (*i*, *x*)
using *r-nf-t-0*[*of y i x*] *r-nf-t-1*[*of y i x*] *r-result-total* **by** *auto*
then have *eval r-result* [*pdec1 y*, *i*, *x*] ↓≠ *0*

by *simp*
then obtain *v* **where** *v*:
eval r-univ [*pdec12 y*, *pdec22 y*] ↓= *v*
eval r-result [*pdec1 y*, *pdec12 y*, *pdec22 y*] ↓= *Suc v*
using *r-result r-result-bivalent'*[*of pdec12 y pdec22 y - pdec1 y*]
r-result-diverg'[*of pdec12 y pdec22 y pdec1 y*]
by *auto*

have *eval r-normal-form* [*i*, *x*] = *eval r-nf-u* [*y*]
unfolding *r-normal-form-def* **using** *y r-nf-t-prim r-nf-u-prim* **by** *simp*
also have ... = *eval r-dec* [*the* (*eval* (*Cn 1 r-result* [*r-pdec1*, *r-pdec12*, *r-pdec22*]) [*y*])]
unfolding *r-nf-u-def* **using** *r-result* **by** *simp*
also have ... = *eval r-dec* [*Suc v*]
using *v* **by** *simp*
also have ... ↓= *v*
by *simp*
finally have *eval r-normal-form* [*i*, *x*] ↓= *v* **.**
moreover have *eval r-univ* [*i*, *x*] ↓= *v*
using *v*(*1*) *pdec2* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
with *ix* **show** *?thesis* **by** *simp*
qed
qed

theorem *normal-form*:
assumes *recfn n f*
obtains *i* **where** ∀ *x*. *e-length x* = *n* ⟶ *eval r-normal-form* [*i*, *x*] = *eval f* (*list-decode x*)
proof −
have *eval r-normal-form* [*encode f*, *x*] = *eval f* (*list-decode x*) **if** *e-length x* = *n* **for** *x*
using *r-univ-exteq-r-normal-form assms that exteq-def r-univ'* **by** *auto*
then show *?thesis* **using** *that* **by** *auto*
qed

As a consequence of the normal form theorem every partial recursive function can be represented with exactly one application of the μ-operator.

fun *count-Mn* :: *recf* ⇒ *nat* **where**
count-Mn Z = *0*
| *count-Mn S* = *0*
| *count-Mn* (*Id m n*) = *0*
| *count-Mn* (*Cn n f gs*) = *count-Mn f* + *sum-list* (*map count-Mn gs*)
| *count-Mn* (*Pr n f g*) = *count-Mn f* + *count-Mn g*
| *count-Mn* (*Mn n f*) = *Suc* (*count-Mn f*)

lemma *count-Mn-zero-iff-prim*: *count-Mn f* = *0* ⟷ *Mn-free f*
by (*induction f*) *auto*

The normal form has only one μ-recursion.

lemma *count-Mn-normal-form*: *count-Mn r-normal-form* = *1*
unfolding *r-normal-form-def r-nf-u-def r-nf-t-def* **using** *count-Mn-zero-iff-prim* **by** *simp*

lemma *one-Mn-suffices*:
assumes *recfn n f*
shows ∃ *g*. *count-Mn g* = *1* ∧ *g* ≃ *f*
proof −

have *n > 0*
  using *assms wellf-arity-nonzero* **by** *auto*
obtain *i* **where** *i*:
  ∀ *x. e-length x = n ⟶ eval r-normal-form [i, x] = eval f (list-decode x)*
  using *normal-form[OF assms(1)]* **by** *auto*
define *g* **where** *g ≡ Cn n r-normal-form [r-constn (n − 1) i, r-list-encode (n − 1)]*
then have *recfn n g*
  using *r-normal-form-recfn ‹n > 0›* **by** *simp*
then have *g ≃ f*
  using *g-def r-list-encode i assms* **by** *(intro exteqI) simp-all*
moreover have *count-Mn g = 1*
  unfolding *g-def* **using** *count-Mn-normal-form count-Mn-zero-iff-prim* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

The previous lemma could have been obtained without *r-normal-form* directly from *r-univ*.

1.8 The *s*-*m*-*n* theorem

For all $m, n > 0$ there is an $(m+1)$-ary primitive recursive function s_n^m with

$$\varphi_p^{(m+n)}(c_1, \ldots, c_m, x_1, \ldots, x_n) = \varphi_{s_n^m(p, c_1, \ldots, c_m)}^{(n)}(x_1, \ldots, x_n)$$

for all $p, c_1, \ldots, c_m, x_1, \ldots, x_n$. Here, $\varphi^{(n)}$ is a function universal for n-ary partial recursive functions, which we will represent by *r-universal n*

The s_n^m functions compute codes of functions. We start simple: computing codes of the unary constant functions.

fun *code-const1* :: *nat ⇒ nat* **where**
  code-const1 0 = 0
| *code-const1 (Suc c) = quad-encode 3 1 1 (singleton-encode (code-const1 c))*

lemma *code-const1*: *code-const1 c = encode (r-const c)*
  by *(induction c) simp-all*

definition *r-code-const1-aux ≡*
  Cn 3 r-prod-encode
    [r-constn 2 3,
      Cn 3 r-prod-encode
        [r-constn 2 1,
          Cn 3 r-prod-encode
            [r-constn 2 1, Cn 3 r-singleton-encode [Id 3 1]]]]

lemma *r-code-const1-aux-prim*: *prim-recfn 3 r-code-const1-aux*
  by *(simp-all add: r-code-const1-aux-def)*

lemma *r-code-const1-aux*:
  eval r-code-const1-aux [i, r, c] ↓= quad-encode 3 1 1 (singleton-encode r)
  by *(simp add: r-code-const1-aux-def)*

definition *r-code-const1 ≡ r-shrink (Pr 1 Z r-code-const1-aux)*

lemma *r-code-const1-prim*: *prim-recfn 1 r-code-const1*

by (*simp-all add*: *r-code-const1-def r-code-const1-aux-prim*)

lemma *r-code-const1*: *eval r-code-const1* [*c*] ↓= *code-const1 c*
proof −
  let *?h* = *Pr 1 Z r-code-const1-aux*
  have *eval ?h* [*c*, *x*] ↓= *code-const1 c* **for** *x*
    using *r-code-const1-aux r-code-const1-def*
    by (*induction c*) (*simp-all add*: *r-code-const1-aux-prim*)
  then show *?thesis* **by** (*simp add*: *r-code-const1-def r-code-const1-aux-prim*)
qed

Functions that compute codes of higher-arity constant functions:

definition *code-constn* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  code-constn n c ≡
    if n = *1 then code-const1 c*
    else quad-encode 3 n (*code-const1 c*) (*singleton-encode* (*triple-encode 2 n 0*))

lemma *code-constn*: *code-constn* (*Suc n*) *c* = *encode* (*r-constn n c*)
  unfolding *code-constn-def* **using** *code-const1 r-constn-def*
  by (*cases n* = *0*) *simp-all*

definition *r-code-constn* :: *nat* ⇒ *recf* **where**
  r-code-constn n ≡
    if n = *1 then r-code-const1*
    else
      Cn 1 r-prod-encode
       [*r-const 3*,
        Cn 1 r-prod-encode
         [*r-const n*,
          Cn 1 r-prod-encode
           [*r-code-const1*,
            Cn 1 r-singleton-encode
             [*Cn 1 r-prod-encode*
               [*r-const 2*, *Cn 1 r-prod-encode* [*r-const n*, *Z*]]]]]]

lemma *r-code-constn-prim*: *prim-recfn 1* (*r-code-constn n*)
  by (*simp-all add*: *r-code-constn-def r-code-const1-prim*)

lemma *r-code-constn*: *eval* (*r-code-constn n*) [*c*] ↓= *code-constn n c*
  by (*auto simp add*: *r-code-constn-def r-code-const1 code-constn-def r-code-const1-prim*)

Computing codes of m-ary projections:

definition *code-id* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  code-id m n ≡ *triple-encode 2 m n*

lemma *code-id*: *encode* (*Id m n*) = *code-id m n*
  unfolding *code-id-def* **by** *simp*

The functions s^m_n are represented by the following function. The value m corresponds to the length of *cs*.

definition *smn* :: *nat* ⇒ *nat* ⇒ *nat list* ⇒ *nat* **where**
  smn n p cs ≡ *quad-encode*
    3
    n
    (*encode* (*r-universal* (*n* + *length cs*)))

```
    (list-encode (code-constn n p # map (code-constn n) cs @ map (code-id n) [0..<n]))

lemma smn:
  assumes n > 0
  shows smn n p cs = encode
   (Cn n
     (r-universal (n + length cs))
     (r-constn (n − 1) p # map (r-constn (n − 1)) cs @ (map (Id n) [0..<n])))
proof −
  let ?p = r-constn (n − 1) p
  let ?gs1 = map (r-constn (n − 1)) cs
  let ?gs2 = map (Id n) [0..<n]
  let ?gs = ?p # ?gs1 @ ?gs2
  have map encode ?gs1 = map (code-constn n) cs
    by (intro nth-equalityI; auto; metis code-constn assms Suc-pred)
  moreover have map encode ?gs2 = map (code-id n) [0..<n]
    by (rule nth-equalityI) (auto simp add: code-id-def)
  moreover have encode ?p = code-constn n p
    using assms code-constn[of n − 1 p] by simp
  ultimately have map encode ?gs =
      code-constn n p # map (code-constn n) cs @ map (code-id n) [0..<n]
    by simp
  then show ?thesis
    unfolding smn-def using assms encode.simps(4) by presburger
qed
```

The next function is to help us define *recf*s corresponding to the s_n^m functions. It maps $m+1$ arguments $p, c_1, \ldots, c_m$ to an encoded list of length $m+n+1$. The list comprises the $m+1$ codes of the n-ary constants $p, c_1, \ldots, c_m$ and the n codes for all n-ary projections.

```
definition r-smn-aux :: nat ⇒ nat ⇒ recf where
  r-smn-aux n m ≡
    Cn (Suc m)
     (r-list-encode (m + n))
     (map (λi. Cn (Suc m) (r-code-constn n) [Id (Suc m) i]) [0..<Suc m] @
      map (λi. r-constn m (code-id n i)) [0..<n])

lemma r-smn-aux-prim: n > 0 ⟹ prim-recfn (Suc m) (r-smn-aux n m)
  by (auto simp add: r-smn-aux-def r-code-constn-prim)

lemma r-smn-aux:
  assumes n > 0 and length cs = m
  shows eval (r-smn-aux n m) (p # cs) ↓=
    list-encode (map (code-constn n) (p # cs) @ map (code-id n) [0..<n])
proof −
  let ?xs = map (λi. Cn (Suc m) (r-code-constn n) [Id (Suc m) i]) [0..<Suc m]
  let ?ys = map (λi. r-constn m (code-id n i)) [0..<n]
  have len-xs: length ?xs = Suc m by simp

  have map-xs: map (λg. eval g (p # cs)) ?xs = map Some (map (code-constn n) (p # cs))
  proof (intro nth-equalityI)
    show len: length (map (λg. eval g (p # cs)) ?xs) =
        length (map Some (map (code-constn n) (p # cs)))
      by (simp add: assms(2))

    have map (λg. eval g (p # cs)) ?xs ! i = map Some (map (code-constn n) (p # cs)) ! i
        if i < Suc m for i
```

proof −
have *map* (*λg. eval g* (*p* # *cs*)) *?xs* ! *i* = (*λg. eval g* (*p* # *cs*)) (*?xs* ! *i*)
using *len-xs that* **by** (*metis nth-map*)
also have ... = *eval* (*Cn* (*Suc m*) (*r-code-constn n*) [*Id* (*Suc m*) *i*]) (*p* # *cs*)
using *that len-xs*
by (*metis* (*no-types, lifting*) *add.left-neutral length-map nth-map nth-upt*)
also have ... = *eval* (*r-code-constn n*) [*the* (*eval* (*Id* (*Suc m*) *i*) (*p* # *cs*))]
using *r-code-constn-prim assms*(*2*) *that* **by** *simp*
also have ... = *eval* (*r-code-constn n*) [(*p* # *cs*) ! *i*]
using *len that* **by** *simp*
finally have *map* (*λg. eval g* (*p* # *cs*)) *?xs* ! *i* ↓= *code-constn n* ((*p* # *cs*) ! *i*)
using *r-code-constn* **by** *simp*
then show *?thesis*
using *len-xs len that* **by** (*metis length-map nth-map*)
qed
moreover have *length* (*map* (*λg. eval g* (*p* # *cs*)) *?xs*) = *Suc m* **by** *simp*
ultimately show ⋀*i. i* < *length* (*map* (*λg. eval g* (*p* # *cs*)) *?xs*) ⟹
map (*λg. eval g* (*p* # *cs*)) *?xs* ! *i* =
map Some (*map* (*code-constn n*) (*p* # *cs*)) ! *i*
by *simp*
qed
moreover have *map* (*λg. eval g* (*p* # *cs*)) *?ys* = *map Some* (*map* (*code-id n*) [*0*..<*n*])
using *assms*(*2*) **by** (*intro nth-equalityI*; *auto*)
ultimately have *map* (*λg. eval g* (*p* # *cs*)) (*?xs* @ *?ys*) =
map Some (*map* (*code-constn n*) (*p* # *cs*) @ *map* (*code-id n*) [*0*..<*n*])
by (*metis map-append*)
moreover have *map* (*λx. the* (*eval x* (*p* # *cs*))) (*?xs* @ *?ys*) =
map the (*map* (*λx. eval x* (*p* # *cs*)) (*?xs* @ *?ys*))
by *simp*
ultimately have ∗: *map* (*λg. the* (*eval g* (*p* # *cs*))) (*?xs* @ *?ys*) =
(*map* (*code-constn n*) (*p* # *cs*) @ *map* (*code-id n*) [*0*..<*n*])
by *simp*

have ∀ *i*<*length ?xs. eval* (*?xs* ! *i*) (*p* # *cs*) = *map* (*λg. eval g* (*p* # *cs*)) *?xs* ! *i*
by (*metis nth-map*)
then have
∀ *i*<*length ?xs. eval* (*?xs* ! *i*) (*p* # *cs*) = *map Some* (*map* (*code-constn n*) (*p* # *cs*)) ! *i*
using *map-xs* **by** *simp*
then have ∀ *i*<*length ?xs. eval* (*?xs* ! *i*) (*p* # *cs*) ↓
using *assms map-xs* **by** (*metis length-map nth-map option.simps*(*3*))
then have *xs-converg*: ∀ *z*∈*set ?xs. eval z* (*p* # *cs*) ↓
by (*metis in-set-conv-nth*)

have ∀ *i*<*length ?ys. eval* (*?ys* ! *i*) (*p* # *cs*) = *map* (*λx. eval x* (*p* # *cs*)) *?ys* ! *i*
by *simp*
then have
∀ *i*<*length ?ys. eval* (*?ys* ! *i*) (*p* # *cs*) = *map Some* (*map* (*code-id n*) [*0*..<*n*]) ! *i*
using *assms*(*2*) **by** *simp*
then have ∀ *i*<*length ?ys. eval* (*?ys* ! *i*) (*p* # *cs*) ↓
by *simp*
then have ∀ *z*∈*set* (*?xs* @ *?ys*). *eval z* (*p* # *cs*) ↓
using *xs-converg* **by** *auto*
moreover have *recfn* (*length* (*p* # *cs*)) (*Cn* (*Suc m*) (*r-list-encode* (*m* + *n*)) (*?xs* @ *?ys*))
using *assms r-code-constn-prim* **by** *auto*
ultimately have *eval* (*r-smn-aux n m*) (*p* # *cs*) =
eval (*r-list-encode* (*m* + *n*)) (*map* (*λg. the* (*eval g* (*p* # *cs*))) (*?xs* @ *?ys*))

unfolding *r-smn-aux-def* **using** *assms* **by** *simp*
then have *eval* (*r-smn-aux n m*) (*p* # *cs*) =
eval (*r-list-encode* (*m* + *n*)) (*map* (*code-constn n*) (*p* # *cs*) @ *map* (*code-id n*) [*0*..<*n*])
using ∗ **by** *metis*
moreover have *length* (*?xs* @ *?ys*) = *Suc* (*m* + *n*) **by** *simp*
ultimately show *?thesis*
using *r-list-encode* ∗ *assms*(*1*) **by** (*metis* (*no-types*, *lifting*) *length-map*)
qed

For all $m, n > 0$, the *recf* corresponding to s_n^m is given by the next function.

definition *r-smn* :: *nat* ⇒ *nat* ⇒ *recf* **where**
r-smn n m ≡
Cn (*Suc m*) *r-prod-encode*
[*r-constn m 3*,
Cn (*Suc m*) *r-prod-encode*
[*r-constn m n*,
Cn (*Suc m*) *r-prod-encode*
[*r-constn m* (*encode* (*r-universal* (*n* + *m*))), *r-smn-aux n m*]]]

lemma *r-smn-prim* [*simp*]: *n* > *0* ⟹ *prim-recfn* (*Suc m*) (*r-smn n m*)
by (*simp-all add*: *r-smn-def r-smn-aux-prim*)

lemma *r-smn*:
assumes *n* > *0* **and** *length cs* = *m*
shows *eval* (*r-smn n m*) (*p* # *cs*) ↓= *smn n p cs*
using *assms r-smn-def r-smn-aux smn-def r-smn-aux-prim* **by** *simp*

lemma *map-eval-Some-the*:
assumes *map* (λ*g*. *eval g xs*) *gs* = *map Some ys*
shows *map* (λ*g*. *the* (*eval g xs*)) *gs* = *ys*
using *assms*
by (*metis* (*no-types*, *lifting*) *length-map nth-equalityI nth-map option.sel*)

The essential part of the *s-m-n* theorem: For all $m, n > 0$ the function s_n^m satisfies

$$\varphi_p^{(m+n)}(c_1, \ldots, c_m, x_1, \ldots, x_n) = \varphi_{s_n^m(p,c_1,\ldots,c_m)}^{(n)}(x_1, \ldots, x_n)$$

for all p, c_i, x_j.

lemma *smn-lemma*:
assumes *n* > *0* **and** *len-cs*: *length cs* = *m* **and** *len-xs*: *length xs* = *n*
shows *eval* (*r-universal* (*m* + *n*)) (*p* # *cs* @ *xs*) =
eval (*r-universal n*) ((*the* (*eval* (*r-smn n m*) (*p* # *cs*))) # *xs*)
proof −
let *?s* = *r-smn n m*
let *?f* = *Cn n*
(*r-universal* (*n* + *length cs*))
(*r-constn* (*n* − *1*) *p* # *map* (*r-constn* (*n* − *1*)) *cs* @ (*map* (*Id n*) [*0*..<*n*]))
have *eval ?s* (*p* # *cs*) ↓= *smn n p cs*
using *assms r-smn* **by** *simp*
then have *eval-s*: *eval ?s* (*p* # *cs*) ↓= *encode ?f*
by (*simp add*: *assms*(*1*) *smn*)

have *recfn n ?f*
using *len-cs assms* **by** *auto*
then have ∗: *eval* (*r-universal n*) ((*encode ?f*) # *xs*) = *eval ?f xs*

```
  using r-universal[of ?f n, OF - len-xs] by simp

let ?gs = r-constn (n − 1) p # map (r-constn (n − 1)) cs @ map (Id n) [0..<n]
have ∀ g∈set ?gs. eval g xs ↓
  using len-cs len-xs assms by auto
then have eval ?f xs =
    eval (r-universal (n + length cs)) (map (λg. the (eval g xs)) ?gs)
  using len-cs len-xs assms ‹recfn n ?f› by simp
then have eval ?f xs = eval (r-universal (m + n)) (map (λg. the (eval g xs)) ?gs)
  by (simp add: len-cs add.commute)
then have eval (r-universal n) ((the (eval ?s (p # cs))) # xs) =
    eval (r-universal (m + n)) (map (λg. the (eval g xs)) ?gs)
  using eval-s ∗ by simp
moreover have map (λg. the (eval g xs)) ?gs = p # cs @ xs
proof (intro nth-equalityI)
  show length (map (λg. the (eval g xs)) ?gs) = length (p # cs @ xs)
    by (simp add: len-xs)
  have len: length (map (λg. the (eval g xs)) ?gs) = Suc (m + n)
    by (simp add: len-cs)
  moreover have map (λg. the (eval g xs)) ?gs ! i = (p # cs @ xs) ! i
    if i < Suc (m + n) for i
  proof −
    from that consider i = 0 | i > 0 ∧ i < Suc m | Suc m ≤ i ∧ i < Suc (m + n)
      using not-le-imp-less by auto
    then show ?thesis
    proof (cases)
      case 1
      then show ?thesis using assms(1) len-xs by simp
    next
      case 2
      then have ?gs ! i = (map (r-constn (n − 1)) cs) ! (i − 1)
        using len-cs
        by (metis One-nat-def Suc-less-eq Suc-pred length-map
          less-numeral-extra(3) nth-Cons' nth-append)
      then have map (λg. the (eval g xs)) ?gs ! i =
          (λg. the (eval g xs)) ((map (r-constn (n − 1)) cs) ! (i − 1))
        using len by (metis length-map nth-map that)
      also have ... = the (eval ((r-constn (n − 1) (cs ! (i − 1)))) xs)
        using 2 len-cs by auto
      also have ... = cs ! (i − 1)
        using r-constn len-xs assms(1) by simp
      also have ... = (p # cs @ xs) ! i
        using 2 len-cs
        by (metis diff-Suc-1 less-Suc-eq-0-disj less-numeral-extra(3) nth-Cons' nth-append)
      finally show ?thesis .
    next
      case 3
      then have ?gs ! i = (map (Id n) [0..<n]) ! (i − Suc m)
        using len-cs
        by (simp; metis (no-types, lifting) One-nat-def Suc-less-eq add-leE
          plus-1-eq-Suc diff-diff-left length-map not-le nth-append
          ordered-cancel-comm-monoid-diff-class.add-diff-inverse)
      then have map (λg. the (eval g xs)) ?gs ! i =
          (λg. the (eval g xs)) ((map (Id n) [0..<n]) ! (i − Suc m))
        using len by (metis length-map nth-map that)
      also have ... = the (eval ((Id n (i − Suc m))) xs)
```

```
        using 3 len-cs by auto
      also have ... = xs ! (i − Suc m)
        using len-xs 3 by auto
      also have ... = (p # cs @ xs) ! i
        using len-cs len-xs 3
        by (metis diff-Suc-1 diff-diff-left less-Suc-eq-0-disj not-le nth-Cons'
          nth-append plus-1-eq-Suc)
      finally show ?thesis .
    qed
  qed
  ultimately show map (λg. the (eval g xs)) ?gs ! i = (p # cs @ xs) ! i
      if i < length (map (λg. the (eval g xs)) ?gs) for i
    using that by simp
qed
ultimately show ?thesis by simp
qed

theorem smn-theorem:
  assumes n > 0
  shows ∃ s. prim-recfn (Suc m) s ∧
    (∀ p cs xs. length cs = m ∧ length xs = n ⟶
        eval (r-universal (m + n)) (p # cs @ xs) =
        eval (r-universal n) ((the (eval s (p # cs))) # xs))
  using smn-lemma exI[of - r-smn n m] assms by simp
```

For every numbering, that is, binary partial recursive function, ψ there is a total recursive function c that translates ψ-indices into φ-indices.

```
lemma numbering-translation:
  assumes recfn 2 psi
  obtains c where
    recfn 1 c
    total c
    ∀ i x. eval psi [i, x] = eval r-phi [the (eval c [i]), x]
proof −
  let ?p = encode psi
  define c where c = Cn 1 (r-smn 1 1) [r-const ?p, Id 1 0]
  then have prim-recfn 1 c by simp
  moreover from this have total c
    by auto
  moreover have eval r-phi [the (eval c [i]), x] = eval psi [i, x] for i x
  proof −
    have eval c [i] = eval (r-smn 1 1) [?p, i]
      using c-def by simp
    then have eval (r-universal 1) [the (eval c [i]), x] =
        eval (r-universal 1) [the (eval (r-smn 1 1) [?p, i]), x]
      by simp
    also have ... = eval (r-universal (1 + 1)) (?p # [i] @ [x])
      using smn-lemma[of 1 [i] 1 [x] ?p] by simp
    also have ... = eval (r-universal 2) [?p, i, x]
      by (metis append-eq-Cons-conv nat-1-add-1)
    also have ... = eval psi [i, x]
      using r-universal[OF assms, of [i, x]] by simp
    finally have eval (r-universal 1) [the (eval c [i]), x] = eval psi [i, x] .
    then show ?thesis using r-phi-def by simp
  qed
  ultimately show ?thesis using that by auto
```

qed

1.9 Fixed-point theorems

Fixed-point theorems (also known as recursion theorems) come in many shapes. We prove the minimum we need for Chapter 2.

1.9.1 Rogers's fixed-point theorem

In this section we prove a theorem that Rogers [12] credits to Kleene, but admits that it is a special case and not the original formulation. We follow Wikipedia [17] and call it the Rogers's fixed-point theorem.

```
lemma s11-inj: inj (λx. smn 1 p [x])
proof
  fix x₁ x₂ :: nat
  assume smn 1 p [x₁] = smn 1 p [x₂]
  then have list-encode [code-constn 1 p, code-constn 1 x₁, code-id 1 0] =
      list-encode [code-constn 1 p, code-constn 1 x₂, code-id 1 0]
    using smn-def by (simp add: prod-encode-eq)
  then have [code-constn 1 p, code-constn 1 x₁, code-id 1 0] =
      [code-constn 1 p, code-constn 1 x₂, code-id 1 0]
    using list-decode-encode by metis
  then have code-constn 1 x₁ = code-constn 1 x₂ by simp
  then show x₁ = x₂
    using code-const1 code-constn code-constn-def encode-injective r-constn
    by (metis One-nat-def length-Cons list.size(3) option.simps(1))
qed

definition r-univuniv ≡ Cn 2 r-phi [Cn 2 r-phi [Id 2 0, Id 2 0], Id 2 1]

lemma r-univuniv-recfn: recfn 2 r-univuniv
  by (simp add: r-univuniv-def)

lemma r-univuniv-converg:
  assumes eval r-phi [x, x] ↓
  shows eval r-univuniv [x, y] = eval r-phi [the (eval r-phi [x, x]), y]
  unfolding r-univuniv-def using assms r-univuniv-recfn r-phi-recfn by simp
```

Strictly speaking this is a generalization of Rogers's theorem in that it shows the existence of infinitely many fixed-points. In conventional terms it says that for every total recursive f and $k \in \mathbb{N}$ there is an $n \geq k$ with $\varphi_n = \varphi_{f(n)}$.

```
theorem rogers-fixed-point-theorem:
  fixes k :: nat
  assumes recfn 1 f and total f
  shows ∃n≥k. ∀x. eval r-phi [n, x] = eval r-phi [the (eval f [n]), x]
proof –
  let ?p = encode r-univuniv
  define h where h = Cn 1 (r-smn 1 1) [r-const ?p, Id 1 0]
  then have prim-recfn 1 h
    by simp
  then have total h
    by blast
  have eval h [x] = eval (Cn 1 (r-smn 1 1) [r-const ?p, Id 1 0]) [x] for x
```

```
  unfolding h-def by simp
then have h: the (eval h [x]) = smn 1 ?p [x] for x
  by (simp add: r-smn)

have eval r-phi [the (eval h [x]), y] = eval r-univuniv [x, y] for x y
proof −
  have eval r-phi [the (eval h [x]), y] = eval r-phi [smn 1 ?p [x], y]
    using h by simp
  also have ... = eval r-phi [the (eval (r-smn 1 1) [?p, x]), y]
    by (simp add: r-smn)
  also have ... = eval (r-universal 2) [?p, x, y]
    using r-phi-def smn-lemma[of 1 [x] 1 [y] ?p]
    by (metis Cons-eq-append-conv One-nat-def Suc-1 length-Cons
      less-numeral-extra(1) list.size(3) plus-1-eq-Suc)
  finally show eval r-phi [the (eval h [x]), y] = eval r-univuniv [x, y]
    using r-universal r-univuniv-recfn by simp
qed
then have ∗: eval r-phi [the (eval h [x]), y] = eval r-phi [the (eval r-phi [x, x]), y]
    if eval r-phi [x, x] ↓ for x y
  using r-univuniv-converg that by simp

let ?fh = Cn 1 f [h]
have recfn 1 ?fh
  using ‹prim-recfn 1 h› assms by simp
then have infinite {r. recfn 1 r ∧ r ≃ ?fh}
  using exteq-infinite[of ?fh 1] by simp
then have infinite (encode ‘ {r. recfn 1 r ∧ r ≃ ?fh}) (is infinite ?E)
  using encode-injective by (meson finite-imageD inj-onI)
then have infinite ((λx. smn 1 ?p [x]) ‘ ?E)
  using s11-inj[of ?p] by (simp add: finite-image-iff inj-on-subset)
moreover have (λx. smn 1 ?p [x]) ‘ ?E = {smn 1 ?p [encode r] |r. recfn 1 r ∧ r ≃ ?fh}
  by auto
ultimately have infinite {smn 1 ?p [encode r] |r. recfn 1 r ∧ r ≃ ?fh}
  by simp
then obtain n where n ≥ k n ∈ {smn 1 ?p [encode r] |r. recfn 1 r ∧ r ≃ ?fh}
  by (meson finite-nat-set-iff-bounded-le le-cases)
then obtain r where r: recfn 1 r n = smn 1 ?p [encode r] recfn 1 r ∧ r ≃ ?fh
  by auto
then have eval-r: eval r [encode r] = eval ?fh [encode r]
  by (simp add: exteq-def)
then have eval-r′: eval r [encode r] = eval f [the (eval h [encode r])]
  using assms ‹total h› ‹prim-recfn 1 h› by simp
then have eval r [encode r] ↓
  using ‹prim-recfn 1 h› assms(1,2) by simp
then have eval r-phi [encode r, encode r] ↓
  by (simp add: ‹recfn 1 r› r-phi)
then have eval r-phi [the (eval h [encode r]), y] =
    eval r-phi [(the (eval r-phi [encode r, encode r])), y]
    for y
  using ∗ by simp
then have eval r-phi [the (eval h [encode r]), y] =
    eval r-phi [(the (eval r [encode r])), y]
    for y
  by (simp add: ‹recfn 1 r› r-phi)
moreover have n = the (eval h [encode r]) by (simp add: h r(2))
ultimately have eval r-phi [n, y] = eval r-phi [the (eval r [encode r]), y] for y
```

```
    by simp
  then have eval r-phi [n, y] = eval r-phi [the (eval ?fh [encode r]), y] for y
    using r by (simp add: eval-r)
  moreover have eval ?fh [encode r] = eval f [n]
    using eval-r eval-r' ‹n = the (eval h [encode r])› by auto
  ultimately have eval r-phi [n, y] = eval r-phi [the (eval f [n]), y] for y
    by simp
  with ‹n ≥ k› show ?thesis by auto
qed
```

1.9.2 Kleene's fixed-point theorem

The next theorem is what Rogers [12, p. 214] calls Kleene's version of what we call Rogers's fixed-point theorem. More precisely this would be Kleene's *second* fixed-point theorem, but since we do not cover the first one, we leave out the number.

```
theorem kleene-fixed-point-theorem:
  fixes k :: nat
  assumes recfn 2 psi
  shows ∃ n≥k. ∀ x. eval r-phi [n, x] = eval psi [n, x]
proof –
  from numbering-translation[OF assms] obtain c where c:
    recfn 1 c
    total c
    ∀ i x. eval psi [i, x] = eval r-phi [the (eval c [i]), x]
    by auto
  then obtain n where n ≥ k and ∀ x. eval r-phi [n, x] = eval r-phi [the (eval c [n]), x]
    using rogers-fixed-point-theorem by blast
  with c(3) have ∀ x. eval r-phi [n, x] = eval psi [n, x]
    by simp
  with ‹n ≥ k› show ?thesis by auto
qed
```

Kleene's fixed-point theorem can be generalized to arbitrary arities. But we need to generalize it only to binary functions in order to show Smullyan's double fixed-point theorem in Section 1.9.3.

```
definition r-univuniv2 ≡
  Cn 3 r-phi [Cn 3 (r-universal 2) [Id 3 0, Id 3 0, Id 3 1], Id 3 2]

lemma r-univuniv2-recfn: recfn 3 r-univuniv2
  by (simp add: r-univuniv2-def)

lemma r-univuniv2-converg:
  assumes eval (r-universal 2) [u, u, x] ↓
  shows eval r-univuniv2 [u, x, y] = eval r-phi [the (eval (r-universal 2) [u, u, x]), y]
  unfolding r-univuniv2-def using assms r-univuniv2-recfn by simp

theorem kleene-fixed-point-theorem-2:
  assumes recfn 2 f and total f
  shows ∃ n.
    recfn 1 n ∧
    total n ∧
    (∀ x y. eval r-phi [(the (eval n [x])), y] = eval r-phi [(the (eval f [the (eval n [x]), x])), y])
proof –
  let ?p = encode r-univuniv2
  let ?s = r-smn 1 2
```

```
define h where h = Cn 2 ?s [r-dummy 1 (r-const ?p), Id 2 0, Id 2 1]
then have [simp]: prim-recfn 2 h by simp
{
  fix u x y
  have eval h [u, x] = eval (Cn 2 ?s [r-dummy 1 (r-const ?p), Id 2 0, Id 2 1]) [u, x]
    using h-def by simp
  then have the (eval h [u, x]) = smn 1 ?p [u, x]
    by (simp add: r-smn)
  then have eval r-phi [the (eval h [u, x]), y] = eval r-phi [smn 1 ?p [u, x], y]
    by simp
  also have ... =
    eval r-phi
      [encode (Cn 1 (r-universal 3) (r-constn 0 ?p # r-constn 0 u # r-constn 0 x # [Id 1 0])),
       y]
    using smn[of 1 ?p [u, x]] by (simp add: numeral-3-eq-3)
  also have ... =
    eval r-phi
      [encode (Cn 1 (r-universal 3) (r-const ?p # r-const u # r-const x # [Id 1 0])), y]
      (is - = eval r-phi [encode ?f, y])
    by (simp add: r-constn-def)
  also have ... = eval ?f [y]
    using r-phi'[of ?f] by auto
  also have ... = eval (r-universal 3) [?p, u, x, y]
    using r-univuniv2-recfn r-universal r-phi by auto
  also have ... = eval r-univuniv2 [u, x, y]
    using r-universal
    by (simp add: r-universal r-univuniv2-recfn)
  finally have eval r-phi [the (eval h [u, x]), y] = eval r-univuniv2 [u, x, y] .
}
then have *: eval r-phi [the (eval h [u, x]), y] =
    eval r-phi [the (eval (r-universal 2) [u, u, x]), y]
    if eval (r-universal 2) [u, u, x] ↓ for u x y
  using r-univuniv2-converg that by simp

let ?fh = Cn 2 f [h, Id 2 1]
let ?e = encode ?fh
have recfn 2 ?fh
  using assms by simp
have total h
  by auto
then have total ?fh
  using assms Cn-total totalI2[of ?fh] by fastforce

let ?n = Cn 1 h [r-const ?e, Id 1 0]
have recfn 1 ?n
  using assms by simp
moreover have total ?n
  using ‹total h› totalI1[of ?n] by simp
moreover {
  fix x y
  have eval r-phi [(the (eval ?n [x])), y] = eval r-phi [(the (eval h [?e, x])), y]
    by simp
  also have ... = eval r-phi [the (eval (r-universal 2) [?e, ?e, x]), y]
    using * r-universal[of - 2] totalE[of ?fh 2] ‹total ?fh› ‹recfn 2 ?fh›
    by (metis length-Cons list.size(3) numeral-2-eq-2)
  also have ... = eval r-phi [the (eval f [the (eval h [?e, x]), x]), y]
```

```
    proof -
      have eval (r-universal 2) [?e, ?e, x] ↓
        using totalE[OF ‹total ?fh›] ‹recfn 2 ?fh› r-universal
        by (metis length-Cons list.size(3) numeral-2-eq-2)
      moreover have eval (r-universal 2) [?e, ?e, x] = eval ?fh [?e, x]
        by (metis ‹recfn 2 ?fh› length-Cons list.size(3) numeral-2-eq-2 r-universal)
      then show ?thesis using assms ‹total h› by simp
    qed
    also have ... = eval r-phi [(the (eval f [the (eval ?n [x]), x])), y]
      by simp
    finally have eval r-phi [(the (eval ?n [x])), y] =
      eval r-phi [(the (eval f [the (eval ?n [x]), x])), y] .
  }
  ultimately show ?thesis by blast
qed
```

1.9.3 Smullyan's double fixed-point theorem

```
theorem smullyan-double-fixed-point-theorem:
  assumes recfn 2 g and total g and recfn 2 h and total h
  shows ∃ m n.
    (∀ x. eval r-phi [m, x] = eval r-phi [the (eval g [m, n]), x]) ∧
    (∀ x. eval r-phi [n, x] = eval r-phi [the (eval h [m, n]), x])
proof -
  obtain m where
    recfn 1 m and
    total m and
    m: ∀ x y. eval r-phi [the (eval m [x]), y] =
      eval r-phi [the (eval g [the (eval m [x]), x]), y]
    using kleene-fixed-point-theorem-2[of g] assms(1,2) by auto
  define k where k = Cn 1 h [m, Id 1 0]
  then have recfn 1 k
    using ‹recfn 1 m› assms(3) by simp
  have total (Id 1 0)
    by (simp add: Mn-free-imp-total)
  then have total k
    using ‹total m› assms(4) Cn-total k-def ‹recfn 1 k› by simp
  obtain n where n: ∀ x. eval r-phi [n, x] = eval r-phi [the (eval k [n]), x]
    using rogers-fixed-point-theorem[of k] ‹recfn 1 k› ‹total k› by blast
  obtain mm where mm: eval m [n] ↓= mm
    using ‹total m› ‹recfn 1 m› by fastforce
  then have ∀ x. eval r-phi [mm, x] = eval r-phi [the (eval g [mm, n]), x]
    by (metis m option.sel)
  moreover have ∀ x. eval r-phi [n, x] = eval r-phi [the (eval h [mm, n]), x]
    using k-def assms(3) ‹total m› ‹recfn 1 m› mm n by simp
  ultimately show ?thesis by blast
qed
```

1.10 Decidable and recursively enumerable sets

We defined *decidable* already back in Section 1.3:

decidable ?X ≡ ∃ f. recfn 1 f ∧ (∀ x. eval f [x] ↓= (if x ∈ ?X then 1 else 0))

The next theorem is adapted from *halting-problem-undecidable*.

```
theorem halting-problem-phi-undecidable: ¬ decidable {x. eval r-phi [x, x] ↓}
  (is ¬ decidable ?K)
proof
  assume decidable ?K
  then obtain f where recfn 1 f and f: ∀ x. eval f [x] ↓= (if x ∈ ?K then 1 else 0)
    using decidable-def by auto
  define g where g ≡ Cn 1 r-ifeq-else-diverg [f, Z, Z]
  then have recfn 1 g
    using ‹recfn 1 f› r-ifeq-else-diverg-recfn by simp
  then obtain i where i: eval r-phi [i, x] = eval g [x] for x
    using r-phi' by auto
  from g-def have eval g [x] = (if x ∉ ?K then Some 0 else None) for x
    using r-ifeq-else-diverg-recfn ‹recfn 1 f› f by simp
  then have eval g [i] ↓ ⟷ i ∉ ?K by simp
  also have ... ⟷ eval r-phi [i, i] ↑ by simp
  also have ... ⟷ eval g [i] ↑
    using i by simp
  finally have eval g [i] ↓ ⟷ eval g [i] ↑ .
  then show False by auto
qed

lemma decidable-complement: decidable X ⟹ decidable (− X)
proof −
  assume decidable X
  then obtain f where f: recfn 1 f ∀ x. eval f [x] ↓= (if x ∈ X then 1 else 0)
    using decidable-def by auto
  define g where g = Cn 1 r-not [f]
  then have recfn 1 g
    by (simp add: f(1))
  moreover have eval g [x] ↓= (if x ∈ X then 0 else 1) for x
    by (simp add: g-def f)
  ultimately show ?thesis using decidable-def by auto
qed
```

Finite sets are decidable.

```
fun r-contains :: nat list ⇒ recf where
  r-contains [] = Z
| r-contains (x # xs) = Cn 1 r-ifeq [Id 1 0, r-const x, r-const 1, r-contains xs]

lemma r-contains-prim: prim-recfn 1 (r-contains xs)
  by (induction xs) auto

lemma r-contains: eval (r-contains xs) [x] ↓= (if x ∈ set xs then 1 else 0)
proof (induction xs arbitrary: x)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  have eval (r-contains (a # xs)) [x] = eval r-ifeq [x, a, 1, the (eval (r-contains xs) [x])]
    using r-contains-prim prim-recfn-total by simp
  also have ... ↓= (if x = a then 1 else if x ∈ set xs then 1 else 0)
    using Cons.IH by simp
  also have ... ↓= (if x = a ∨ x ∈ set xs then 1 else 0)
    by simp
  finally show ?case by simp
qed
```

lemma *finite-set-decidable*: *finite X ⟹ decidable X*
proof −
  fix *X* :: *nat set*
  assume *finite X*
  then obtain *xs* **where** *X = set xs*
    using *finite-list* **by** *auto*
  then have ∀ *x. eval (r-contains xs) [x] ↓= (if x ∈ X then 1 else 0)*
    using *r-contains* **by** *simp*
  then show *decidable X*
    using *decidable-def r-contains-prim* **by** *blast*
qed

definition *semidecidable* :: *nat set ⇒ bool* **where**
  semidecidable X ≡ (∃f. recfn 1 f ∧ (∀ x. eval f [x] = (if x ∈ X then Some 1 else None)))

The semidecidable sets are the domains of partial recursive functions.

lemma *semidecidable-iff-domain*:
  semidecidable X ⟷ (∃f. recfn 1 f ∧ (∀ x. eval f [x] ↓ ⟷ x ∈ X))
proof
  show *semidecidable X ⟹ ∃f. recfn 1 f ∧ (∀ x. (eval f [x] ↓) = (x ∈ X))*
    using *semidecidable-def* **by** *(metis option.distinct(1))*
  show *semidecidable X* **if** *∃f. recfn 1 f ∧ (∀ x. (eval f [x] ↓) = (x ∈ X))* **for** *X*
  proof −
    from *that* **obtain** *f* **where** *f*: *recfn 1 f ∀ x. (eval f [x] ↓) = (x ∈ X)*
      by *auto*
    let *?g = Cn 1 (r-const 1) [f]*
    have *recfn 1 ?g*
      using *f(1)* **by** *simp*
    moreover have *∀ x. eval ?g [x] = (if x ∈ X then Some 1 else None)*
      using *f* **by** *simp*
    ultimately show *semidecidable X*
      using *semidecidable-def* **by** *blast*
  qed
qed

lemma *decidable-imp-semidecidable*: *decidable X ⟹ semidecidable X*
proof −
  assume *decidable X*
  then obtain *f* **where** *f*: *recfn 1 f ∀ x. eval f [x] ↓= (if x ∈ X then 1 else 0)*
    using *decidable-def* **by** *auto*
  define *g* **where** *g = Cn 1 r-ifeq-else-diverg [f, r-const 1, r-const 1]*
  then have *recfn 1 g*
    by *(simp add: f(1))*
  have *eval g [x] = eval r-ifeq-else-diverg [if x ∈ X then 1 else 0, 1, 1]* **for** *x*
    by *(simp add: g-def f)*
  then have *⋀x. x ∈ X ⟹ eval g [x] ↓= 1* **and** *⋀x. x ∉ X ⟹ eval g [x] ↑*
    by *simp-all*
  then show *?thesis*
    using *‹recfn 1 g› semidecidable-def* **by** *auto*
qed

A set is recursively enumerable if it is empty or the image of a total recursive function.

definition *recursively-enumerable* :: *nat set ⇒ bool* **where**
  recursively-enumerable X ≡
    X = {} ∨ (∃f. recfn 1 f ∧ total f ∧ X = {the (eval f [x]) |x. x ∈ UNIV})

theorem *recursively-enumerable-iff-semidecidable*:
recursively-enumerable X ⟷ *semidecidable X*
proof
show *semidecidable X* **if** *recursively-enumerable X* **for** *X*
proof (*cases*)
assume *X* = {}
then show *?thesis*
using *finite-set-decidable decidable-imp-semidecidable*
recursively-enumerable-def semidecidable-def
by *blast*
next
assume *X* ≠ {}
with *that* **obtain** *f* **where** *f*: *recfn 1 f total f X* = {*the* (*eval f* [*x*]) |*x*. *x* ∈ *UNIV*}
using *recursively-enumerable-def* **by** *blast*
define *h* **where** *h* = *Cn 2 r-eq* [*Cn 2 f* [*Id 2 0*], *Id 2 1*]
then have *recfn 2 h*
using *f*(*1*) **by** *simp*
from *h-def* **have** *h*: *eval h* [*x*, *y*] ↓= *0* ⟷ *the* (*eval f* [*x*]) = *y* **for** *x y*
using *f*(*1*,*2*) **by** *simp*
from *h-def* ‹*recfn 2 h*› *totalI2 f*(*2*) **have** *total h* **by** *simp*
define *g* **where** *g* = *Mn 1 h*
then have *recfn 1 g*
using *h-def f*(*1*) **by** *simp*
then have *eval g* [*y*] =
(*if* (∃ *x*. *eval h* [*x*, *y*] ↓= *0* ∧ (∀ *x'*<*x*. *eval h* [*x'*, *y*] ↓))
then Some (*LEAST x*. *eval h* [*x*, *y*] ↓= *0*)
else None) **for** *y*
using *g-def* ‹*total h*› *f*(*2*) **by** *simp*
then have *eval g* [*y*] =
(*if* ∃ *x*. *eval h* [*x*, *y*] ↓= *0*
then Some (*LEAST x*. *eval h* [*x*, *y*] ↓= *0*)
else None) **for** *y*
using ‹*total h*› ‹*recfn 2 h*› **by** *simp*
then have *eval g* [*y*] ↓ ⟷ (∃ *x*. *eval h* [*x*, *y*] ↓= *0*) **for** *y*
by *simp*
with *h* **have** *eval g* [*y*] ↓ ⟷ (∃ *x*. *the* (*eval f* [*x*]) = *y*) **for** *y*
by *simp*
with *f*(*3*) **have** *eval g* [*y*] ↓ ⟷ *y* ∈ *X* **for** *y*
by *auto*
with ‹*recfn 1 g*› *semidecidable-iff-domain* **show** *?thesis* **by** *auto*
qed

show *recursively-enumerable X* **if** *semidecidable X* **for** *X*
proof (*cases*)
assume *X* = {}
then show *?thesis* **using** *recursively-enumerable-def* **by** *simp*
next
assume *X* ≠ {}
then obtain x_0 **where** x_0 ∈ *X* **by** *auto*
from *that semidecidable-iff-domain* **obtain** *f* **where** *f*: *recfn 1 f* ∀ *x*. *eval f* [*x*] ↓ ⟷ *x* ∈ *X*
by *auto*
let *?i* = *encode f*
have *i*: ⋀*x*. *eval f* [*x*] = *eval r-phi* [*?i*, *x*]
using *r-phi' f*(*1*) **by** *simp*
with ‹x_0 ∈ *X*› *f*(*2*) **have** *eval r-phi* [*?i*, x_0] ↓ **by** *simp*

    then obtain *g* **where** *g*: *recfn 1 g total g* $\forall x$. *eval r-phi* [*?i*, *x*] ↓ = ($\exists y$. *eval g* [*y*] ↓= *x*)
      using *f*(*1*) *nonempty-domain-enumerable* **by** *blast*
    with *f*(*2*) *i* **have** $\forall x$. $x \in X$ = ($\exists y$. *eval g* [*y*] ↓= *x*)
      by *simp*
    then have $\forall x$. $x \in X$ = ($\exists y$. *the* (*eval g* [*y*]) = *x*)
      using *totalE*[*OF g*(*2*) *g*(*1*)]
      by (*metis One-nat-def length-Cons list.size*(*3*) *option.collapse option.sel*)
    then have *X* = {*the* (*eval g* [*y*]) |*y*. *y* ∈ *UNIV*}
      by *auto*
    with *g*(*1*,*2*) **show** *?thesis* **using** *recursively-enumerable-def* **by** *auto*
  qed
qed

The next goal is to show that a set is decidable iff. it and its complement are semidecidable. For this we use the concurrent evaluation function.

lemma *semidecidable-decidable*:
  assumes *semidecidable X* **and** *semidecidable* (− *X*)
  shows *decidable X*
proof −
  obtain *f* **where** *f*: *recfn 1 f* ∧ (∀ *x*. *eval f* [*x*] ↓ ⟷ *x* ∈ *X*)
    using *assms*(*1*) *semidecidable-iff-domain* **by** *auto*
  let *?i* = *encode f*
  obtain *g* **where** *g*: *recfn 1 g* ∧ (∀ *x*. *eval g* [*x*] ↓ ⟷ *x* ∈ (− *X*))
    using *assms*(*2*) *semidecidable-iff-domain* **by** *auto*
  let *?j* = *encode g*
  define *d* **where** *d* = *Cn 1 r-pdec1* [*Cn 1 r-parallel* [*r-const ?j*, *r-const ?i*, *Id 1 0*]]
  then have *recfn 1 d*
    by (*simp add*: *d-def*)
  have ∗: ⋀*x*. *eval r-phi* [*?i*, *x*] = *eval f* [*x*] ⋀*x*. *eval r-phi* [*?j*, *x*] = *eval g* [*x*]
    using *f g r-phi′* **by** *simp-all*
  have *eval d* [*x*] ↓= *1* **if** *x* ∈ *X* **for** *x*
  proof −
    have *eval f* [*x*] ↓
      using *f that* **by** *simp*
    moreover have *eval g* [*x*] ↑
      using *g that* **by** *blast*
    ultimately have *eval r-parallel* [*?j*, *?i*, *x*] ↓= *prod-encode* (*1*, *the* (*eval f* [*x*]))
      using ∗ *r-parallel*(*3*) **by** *simp*
    with *d-def* **show** *?thesis* **by** *simp*
  qed
  moreover have *eval d* [*x*] ↓= *0* **if** *x* ∉ *X* **for** *x*
  proof −
    have *eval g* [*x*] ↓
      using *g that* **by** *simp*
    moreover have *eval f* [*x*] ↑
      using *f that* **by** *blast*
    ultimately have *eval r-parallel* [*?j*, *?i*, *x*] ↓= *prod-encode* (*0*, *the* (*eval g* [*x*]))
      using ∗ *r-parallel*(*2*) **by** *blast*
    with *d-def* **show** *?thesis* **by** *simp*
  qed
  ultimately show *?thesis*
    using *decidable-def* ‹*recfn 1 d*› **by** *auto*
qed

theorem *decidable-iff-semidecidable-complement*:
  decidable X ⟷ *semidecidable X* ∧ *semidecidable* (− *X*)

```
  using semidecidable-decidable decidable-imp-semidecidable decidable-complement
  by blast
```

1.11 Rice's theorem

```
definition index-set :: nat set ⇒ bool where
  index-set I ≡ ∀ i j. i ∈ I ∧ (∀ x. eval r-phi [i, x] = eval r-phi [j, x]) ⟶ j ∈ I

lemma index-set-closed-in:
  assumes index-set I and i ∈ I and ∀ x. eval r-phi [i, x] = eval r-phi [j, x]
  shows j ∈ I
  using index-set-def assms by simp

lemma index-set-closed-not-in:
  assumes index-set I and i ∉ I and ∀ x. eval r-phi [i, x] = eval r-phi [j, x]
  shows j ∉ I
  using index-set-def assms by metis

theorem rice-theorem:
  assumes index-set I and I ≠ UNIV and I ≠ {}
  shows ¬ decidable I
proof
  assume decidable I
  then obtain d where d: recfn 1 d ∀ i. eval d [i] ↓= (if i ∈ I then 1 else 0)
    using decidable-def by auto
  obtain j1 j2 where j1 ∉ I and j2 ∈ I
    using assms(2,3) by auto
  let ?if = Cn 2 r-ifz [Cn 2 d [Id 2 0], r-dummy 1 (r-const j2), r-dummy 1 (r-const j1)]
  define psi where psi = Cn 2 r-phi [?if, Id 2 1]
  then have recfn 2 psi
    by (simp add: d)
  have eval ?if [x, y] = Some (if x ∈ I then j1 else j2) for x y
    by (simp add: d)
  moreover have eval psi [x, y] = eval (Cn 2 r-phi [?if, Id 2 1]) [x, y] for x y
    using psi-def by simp
  ultimately have psi: eval psi [x, y] = eval r-phi [if x ∈ I then j1 else j2, y] for x y
    by (simp add: d)
  then have in-I: eval psi [x, y] = eval r-phi [j1, y] if x ∈ I for x y
    by (simp add: that)
  have not-in-I: eval psi [x, y] = eval r-phi [j2, y] if x ∉ I for x y
    by (simp add: psi that)
  obtain n where n: ∀ x. eval r-phi [n, x] = eval psi [n, x]
    using kleene-fixed-point-theorem[OF ‹recfn 2 psi›] by auto
  show False
  proof cases
    assume n ∈ I
    then have ∀ x. eval r-phi [n, x] = eval r-phi [j1, x]
      using n in-I by simp
    then have n ∉ I
      using ‹j1 ∉ I› index-set-closed-not-in[OF assms(1)] by simp
    with ‹n ∈ I› show False by simp
  next
    assume n ∉ I
    then have ∀ x. eval r-phi [n, x] = eval r-phi [j2, x]
      using n not-in-I by simp
```

then have $n \in I$
using ‹$j_2 \in I$› *index-set-closed-in*[*OF assms*(*1*)] **by** *simp*
with ‹$n \notin I$› **show** *False* **by** *simp*
qed
qed

1.12 Partial recursive functions as actual functions

A well-formed *recf* describes an algorithm. Usually, however, partial recursive functions are considered to be partial functions, that is, right-unique binary relations. This distinction did not matter much until now, because we were mostly concerned with the *existence* of partial recursive functions, which is equivalent to the existence of algorithms. Whenever it did matter, we could use the extensional equivalence ($\simeq$). In Chapter 2, however, we will deal with sets of functions and sets of sets of functions.

For illustration consider the singleton set containing only the unary zero function. It could be expressed by $\{Z\}$, but this would not contain *Cn 1* (*Id 1 0*) [*Z*], which computes the same function. The alternative representation as $\{f.\ f \simeq Z\}$ is not a singleton set. Another alternative would be to identify partial recursive functions with the equivalence classes of ($\simeq$). This would work for all arities. But since we will only need unary and binary functions, we can go for the less general but simpler alternative of regarding partial recursive functions as certain functions of types *nat* ⇒ *nat option* and *nat* ⇒ *nat* ⇒ *nat option*. With this notation we can represent the aforementioned set by {λ-. *Some 0*} and express that the function λ-. *Some 0* is total recursive.

In addition terms get shorter, for instance, *eval r-func* [*i*, *x*] becomes *func i x*.

1.12.1 The definitions

type-synonym *partial1* = *nat* ⇒ *nat option*

type-synonym *partial2* = *nat* ⇒ *nat* ⇒ *nat option*

definition *total1* :: *partial1* ⇒ *bool* **where**
total1 f ≡ ∀ *x*. *f x* ↓

definition *total2* :: *partial2* ⇒ *bool* **where**
total2 f ≡ ∀ *x y*. *f x y* ↓

lemma *total1I* [*intro*]: (⋀*x*. *f x* ↓) ⟹ *total1 f*
using *total1-def* **by** *simp*

lemma *total2I* [*intro*]: (⋀*x y*. *f x y* ↓) ⟹ *total2 f*
using *total2-def* **by** *simp*

lemma *total1E* [*dest*, *simp*]: *total1 f* ⟹ *f x* ↓
using *total1-def* **by** *simp*

lemma *total2E* [*dest*, *simp*]: *total2 f* ⟹ *f x y* ↓
using *total2-def* **by** *simp*

definition *P1* :: *partial1 set* (‹$\mathcal{P}$›) **where**
$\mathcal{P}$ ≡ {λ*x*. *eval r* [*x*] |*r*. *recfn 1 r*}

definition *P2* :: *partial2 set* (‹$\mathcal{P}^2$›) **where**
$\mathcal{P}^2 \equiv \{\lambda x\ y.\ eval\ r\ [x, y]\ |r.\ recfn\ 2\ r\}$

definition *R1* :: *partial1 set* (‹$\mathcal{R}$›) **where**
$\mathcal{R} \equiv \{\lambda x.\ eval\ r\ [x]\ |r.\ recfn\ 1\ r \wedge total\ r\}$

definition *R2* :: *partial2 set* (‹$\mathcal{R}^2$›) **where**
$\mathcal{R}^2 \equiv \{\lambda x\ y.\ eval\ r\ [x, y]\ |r.\ recfn\ 2\ r \wedge total\ r\}$

definition *Prim1* :: *partial1 set* **where**
$Prim1 \equiv \{\lambda x.\ eval\ r\ [x]\ |r.\ prim\text{-}recfn\ 1\ r\}$

definition *Prim2* :: *partial2 set* **where**
$Prim2 \equiv \{\lambda x\ y.\ eval\ r\ [x, y]\ |r.\ prim\text{-}recfn\ 2\ r\}$

lemma *R1-imp-P1* [*simp*, *elim*]: $f \in \mathcal{R} \Longrightarrow f \in \mathcal{P}$
using *R1-def P1-def* **by** *auto*

lemma *R2-imp-P2* [*simp*, *elim*]: $f \in \mathcal{R}^2 \Longrightarrow f \in \mathcal{P}^2$
using *R2-def P2-def* **by** *auto*

lemma *Prim1-imp-R1* [*simp*, *elim*]: $f \in Prim1 \Longrightarrow f \in \mathcal{R}$
unfolding *Prim1-def R1-def* **by** *auto*

lemma *Prim2-imp-R2* [*simp*, *elim*]: $f \in Prim2 \Longrightarrow f \in \mathcal{R}^2$
unfolding *Prim2-def R2-def* **by** *auto*

lemma *P1E* [*elim*]:
assumes $f \in \mathcal{P}$
obtains *r* **where** *recfn 1 r* **and** $\forall x.\ eval\ r\ [x] = f\ x$
using *assms P1-def* **by** *force*

lemma *P2E* [*elim*]:
assumes $f \in \mathcal{P}^2$
obtains *r* **where** *recfn 2 r* **and** $\forall x\ y.\ eval\ r\ [x, y] = f\ x\ y$
using *assms P2-def* **by** *force*

lemma *P1I* [*intro*]:
assumes *recfn 1 r* **and** $(\lambda x.\ eval\ r\ [x]) = f$
shows $f \in \mathcal{P}$
using *assms P1-def* **by** *auto*

lemma *P2I* [*intro*]:
assumes *recfn 2 r* **and** $\bigwedge x\ y.\ eval\ r\ [x, y] = f\ x\ y$
shows $f \in \mathcal{P}^2$
proof −
have $(\lambda x\ y.\ eval\ r\ [x, y]) = f$
using *assms*(*2*) **by** *simp*
then show *?thesis*
using *assms*(*1*) *P2-def* **by** *auto*
qed

lemma *R1I* [*intro*]:
assumes *recfn 1 r* **and** *total r* **and** $\bigwedge x.\ eval\ r\ [x] = f\ x$
shows $f \in \mathcal{R}$
unfolding *R1-def*

using *CollectI*[*of* $\lambda f. \exists r. f = (\lambda x.\ eval\ r\ [x]) \wedge recfn\ 1\ r \wedge total\ r\ f$] *assms*
by *metis*

lemma *R1E* [*elim*]:
assumes $f \in \mathcal{R}$
obtains *r* **where** *recfn 1 r* **and** *total r* **and** $f = (\lambda x.\ eval\ r\ [x])$
using *assms R1-def* **by** *auto*

lemma *R2I* [*intro*]:
assumes *recfn 2 r* **and** *total r* **and** $\bigwedge x\ y.\ eval\ r\ [x, y] = f\ x\ y$
shows $f \in \mathcal{R}^2$
unfolding *R2-def*
using *CollectI*[*of* $\lambda f. \exists r. f = (\lambda x\ y.\ eval\ r\ [x, y]) \wedge recfn\ 2\ r \wedge total\ r\ f$] *assms*
by *metis*

lemma *R1-SOME*:
assumes $f \in \mathcal{R}$
and $r = (SOME\ r'.\ recfn\ 1\ r' \wedge\ total\ r' \wedge f = (\lambda x.\ eval\ r'\ [x]))$
(**is** $r = (SOME\ r'.\ ?P\ r')$)
shows *recfn 1 r*
and $\bigwedge x.\ eval\ r\ [x] \downarrow$
and $\bigwedge x.\ f\ x = eval\ r\ [x]$
and $f = (\lambda x.\ eval\ r\ [x])$
proof −
obtain r' **where** $?P\ r'$
using *R1E*[*OF assms*(*1*)] **by** *auto*
then show *recfn 1 r* $\bigwedge b.\ eval\ r\ [b] \downarrow \bigwedge x.\ f\ x = eval\ r\ [x]$
using *someI*[*of ?P r'*] *assms*(*2*) *totalE*[*of r*] **by** (*auto*, *metis*)
then show $f = (\lambda x.\ eval\ r\ [x])$ **by** *auto*
qed

lemma *R2E* [*elim*]:
assumes $f \in \mathcal{R}^2$
obtains *r* **where** *recfn 2 r* **and** *total r* **and** $f = (\lambda x_1\ x_2.\ eval\ r\ [x_1, x_2])$
using *assms R2-def* **by** *auto*

lemma *R1-imp-total1* [*simp*]: $f \in \mathcal{R} \Longrightarrow total1\ f$
using *total1I* **by** *fastforce*

lemma *R2-imp-total2* [*simp*]: $f \in \mathcal{R}^2 \Longrightarrow total2\ f$
using *totalE* **by** *fastforce*

lemma *Prim1I* [*intro*]:
assumes *prim-recfn 1 r* **and** $\bigwedge x.\ f\ x = eval\ r\ [x]$
shows $f \in Prim1$
using *assms Prim1-def* **by** *blast*

lemma *Prim2I* [*intro*]:
assumes *prim-recfn 2 r* **and** $\bigwedge x\ y.\ f\ x\ y = eval\ r\ [x, y]$
shows $f \in Prim2$
using *assms Prim2-def* **by** *blast*

lemma *P1-total-imp-R1* [*intro*]:
assumes $f \in \mathcal{P}$ **and** *total1 f*
shows $f \in \mathcal{R}$
using *assms totalI1* **by** *force*

lemma *P2-total-imp-R2* [*intro*]:
  assumes $f \in \mathcal{P}^2$ **and** *total2 f*
  shows $f \in \mathcal{R}^2$
  using *assms totalI2* **by** *force*

1.12.2 Some simple properties

In order to show that a *partial1* or *partial2* function is in $\mathcal{P}$, $\mathcal{P}^2$, $\mathcal{R}$, $\mathcal{R}^2$, *Prim1*, or *Prim2* we will usually have to find a suitable *recf*. But for some simple or frequent cases this section provides shortcuts.

lemma *identity-in-R1*: *Some* $\in \mathcal{R}$
proof −
  have $\forall x.$ *eval* (*Id 1 0*) [x] $\downarrow= x$ **by** *simp*
  moreover have *recfn 1* (*Id 1 0*) **by** *simp*
  moreover have *total* (*Id 1 0*)
    by (*simp add*: *totalI1*)
  ultimately show *?thesis* **by** *blast*
qed

lemma *P2-proj-P1* [*simp*, *elim*]:
  assumes $\psi \in \mathcal{P}^2$
  shows $\psi\ i \in \mathcal{P}$
proof −
  from *assms* **obtain** *u* **where** *u*: *recfn 2 u* ($\lambda x_1\ x_2.$ *eval u* [x_1, x_2]) $= \psi$
    by *auto*
  define *v* **where** $v \equiv$ *Cn 1 u* [*r-const i*, *Id 1 0*]
  then have *recfn 1 v* ($\lambda x.$ *eval v* [x]) $= \psi\ i$
    using *u* **by** *auto*
  then show *?thesis* **by** *auto*
qed

lemma *R2-proj-R1* [*simp*, *elim*]:
  assumes $\psi \in \mathcal{R}^2$
  shows $\psi\ i \in \mathcal{R}$
proof −
  from *assms* **have** $\psi \in \mathcal{P}^2$ **by** *simp*
  then have $\psi\ i \in \mathcal{P}$ **by** *auto*
  moreover have *total1* ($\psi\ i$)
    using *assms* **by** (*simp add*: *total1I*)
  ultimately show *?thesis* **by** *auto*
qed

lemma *const-in-Prim1*: (λ-. *Some c*) $\in$ *Prim1*
proof −
  define *r* **where** $r =$ *r-const c*
  then have $\bigwedge x.$ *eval r* [x] $=$ *Some c* **by** *simp*
  moreover have *recfn 1 r Mn-free r*
    using *r-def* **by** *simp-all*
  ultimately show *?thesis* **by** *auto*
qed

lemma *concat-P1-P1*:
  assumes $f \in \mathcal{P}$ **and** $g \in \mathcal{P}$
  shows ($\lambda x.$ *if* $g\ x \downarrow \wedge f$ (*the* ($g\ x$)) $\downarrow$ *then Some* (*the* (f (*the* ($g\ x$)))) *else None*) $\in \mathcal{P}$

(**is** *?h* ∈ 𝒫)
proof −
obtain *rf* **where** *rf*: *recfn 1 rf* ∀ *x*. *eval rf* [*x*] = *f x*
using *assms*(*1*) **by** *auto*
obtain *rg* **where** *rg*: *recfn 1 rg* ∀ *x*. *eval rg* [*x*] = *g x*
using *assms*(*2*) **by** *auto*
let *?rh* = *Cn 1 rf* [*rg*]
have *recfn 1 ?rh*
using *rf*(*1*) *rg*(*1*) **by** *simp*
moreover have *eval ?rh* [*x*] = *?h x* **for** *x*
using *rf rg* **by** *simp*
ultimately show *?thesis* **by** *blast*
qed

lemma *P1-update-P1*:
assumes *f* ∈ 𝒫
shows *f*(*x*:=*z*) ∈ 𝒫
proof (*cases z*)
case *None*
define *re* **where** *re* ≡ *Mn 1* (*r-constn 1 1*)
from *assms* **obtain** *r* **where** *r*: *recfn 1 r* (λ*u*. *eval r* [*u*]) = *f*
by *auto*
define *r′* **where** *r′* = *Cn 1* (*r-lifz re r*) [*Cn 1 r-eq* [*Id 1 0*, *r-const x*], *Id 1 0*]
have *recfn 1 r′*
using *r*(*1*) *r′-def re-def* **by** *simp*
then have *eval r′* [*u*] = *eval* (*r-lifz re r*) [*if u* = *x then 0 else 1*, *u*] **for** *u*
using *r′-def* **by** *simp*
with *r*(*1*) **have** *eval r′* [*u*] = (*if u* = *x then None else eval r* [*u*]) **for** *u*
using *re-def re-def* **by** *simp*
with *r*(*2*) **have** *eval r′* [*u*] = (*f*(*x*:=*None*)) *u* **for** *u*
by *auto*
then have (λ*u*. *eval r′* [*u*]) = *f*(*x*:=*None*)
by *auto*
with *None* ‹*recfn 1 r′*› **show** *?thesis* **by** *auto*
next
case (*Some y*)
from *assms* **obtain** *r* **where** *r*: *recfn 1 r* (λ*u*. *eval r* [*u*]) = *f*
by *auto*
define *r′* **where**
r′ ≡ *Cn 1* (*r-lifz* (*r-const y*) *r*) [*Cn 1 r-eq* [*Id 1 0*, *r-const x*], *Id 1 0*]
have *recfn 1 r′*
using *r*(*1*) *r′-def* **by** *simp*
then have *eval r′* [*u*] = *eval* (*r-lifz* (*r-const y*) *r*) [*if u* = *x then 0 else 1*, *u*] **for** *u*
using *r′-def* **by** *simp*
with *r*(*1*) **have** *eval r′* [*u*] = (*if u* = *x then Some y else eval r* [*u*]) **for** *u*
by *simp*
with *r*(*2*) **have** *eval r′* [*u*] = (*f*(*x*:=*Some y*)) *u* **for** *u*
by *auto*
then have (λ*u*. *eval r′* [*u*]) = *f*(*x*:=*Some y*)
by *auto*
with *Some* ‹*recfn 1 r′*› **show** *?thesis* **by** *auto*
qed

lemma *swap-P2*:
assumes *f* ∈ $\mathcal{P}^2$
shows (λ*x y*. *f y x*) ∈ $\mathcal{P}^2$

proof −
  obtain *r* **where** *r*: *recfn 2 r* $\bigwedge x\ y.\ eval\ r\ [x,\ y] = f\ x\ y$
    using *assms* **by** *auto*
  then have *eval* (*r-swap r*) $[x,\ y] = f\ y\ x$ **for** *x y*
    by *simp*
  moreover have *recfn 2* (*r-swap r*)
    using *r-swap-recfn r*(*1*) **by** *simp*
  ultimately show *?thesis* **by** *auto*
qed

lemma *swap-R2*:
  assumes $f \in \mathcal{R}^2$
  shows $(\lambda x\ y.\ f\ y\ x) \in \mathcal{R}^2$
  using *swap-P2*[*of f*] *assms*
  by (*meson P2-total-imp-R2 R2-imp-P2 R2-imp-total2 total2E total2I*)

lemma *skip-P1*:
  assumes $f \in \mathcal{P}$
  shows $(\lambda x.\ f\ (x + n)) \in \mathcal{P}$
proof −
  obtain *r* **where** *r*: *recfn 1 r* $\bigwedge x.\ eval\ r\ [x] = f\ x$
    using *assms* **by** *auto*
  let *?s* = *Cn 1 r* [*Cn 1 r-add* [*Id 1 0*, *r-const n*]]
  have *recfn 1 ?s*
    using *r* **by** *simp*
  have *eval ?s* $[x]$ = *eval r* $[x + n]$ **for** *x*
    using *r* **by** *simp*
  with *r* **have** *eval ?s* $[x] = f\ (x + n)$ **for** *x*
    by *simp*
  with ‹*recfn 1 ?s*› **show** *?thesis* **by** *blast*
qed

lemma *skip-R1*:
  assumes $f \in \mathcal{R}$
  shows $(\lambda x.\ f\ (x + n)) \in \mathcal{R}$
  using *assms skip-P1 R1-imp-total1 total1-def* **by** *auto*

1.12.3 The Gödel numbering φ

While the term *Gödel numbering* is often used generically for mappings between natural numbers and mathematical concepts, the inductive inference literature uses it in a more specific sense. There it is equivalent to the notion of acceptable numbering [12]: For every numbering there is a recursive function mapping the numbering's indices to equivalent ones of a Gödel numbering.

definition *goedel-numbering* :: *partial2* ⇒ *bool* **where**
  goedel-numbering $\psi \equiv \psi \in \mathcal{P}^2 \wedge (\forall \chi{\in}\mathcal{P}^2.\ \exists c{\in}\mathcal{R}.\ \forall i.\ \chi\ i = \psi\ (the\ (c\ i)))$

lemma *goedel-numbering-P2*:
  assumes *goedel-numbering* ψ
  shows $\psi \in \mathcal{P}^2$
  using *goedel-numbering-def assms* **by** *simp*

lemma *goedel-numberingE*:
  assumes *goedel-numbering* ψ **and** $\chi \in \mathcal{P}^2$
  obtains *c* **where** $c \in \mathcal{R}$ **and** $\forall i.\ \chi\ i = \psi\ (the\ (c\ i))$

using *assms goedel-numbering-def* **by** *blast*

lemma *goedel-numbering-universal*:
assumes *goedel-numbering* ψ **and** $f \in \mathcal{P}$
shows $\exists\, i.\ \psi\ i = f$
proof –
define χ :: *partial2* **where** $\chi = (\lambda i.\ f)$
have $\chi \in \mathcal{P}^2$
proof –
obtain *rf* **where** *rf*: *recfn 1 rf* $\bigwedge x.$ *eval rf* $[x] = f\ x$
using *assms*(*2*) **by** *auto*
define *r* **where** $r = $ *Cn 2 rf* [*Id 2 1*]
then have *r*: *recfn 2 r* $\bigwedge i\ x.$ *eval r* $[i, x] =$ *eval rf* $[x]$
using *rf*(*1*) **by** *simp-all*
with *rf*(*2*) **have** $\bigwedge i\ x.$ *eval r* $[i, x] = f\ x$ **by** *simp*
with *r*(*1*) **show** *?thesis* **using** χ-*def* **by** *auto*
qed
then obtain *c* **where** $c \in \mathcal{R}$ **and** $\forall\, i.\ \chi\ i = \psi$ (*the* (*c i*))
using *goedel-numbering-def assms*(*1*) **by** *auto*
with χ-*def* **show** *?thesis* **by** *auto*
qed

Our standard Gödel numbering is based on *r-phi*:

definition *phi* :: *partial2* (‹φ›) **where**
$\varphi\ i\ x \equiv$ *eval r-phi* $[i, x]$

lemma *phi-in-P2*: $\varphi \in \mathcal{P}^2$
unfolding *phi-def* **using** *r-phi-recfn* **by** *blast*

Indices of any numbering can be translated into equivalent indices of φ, which thus is a Gödel numbering.

lemma *numbering-translation-for-phi*:
assumes $\psi \in \mathcal{P}^2$
shows $\exists\, c \in \mathcal{R}.\ \forall\, i.\ \psi\ i = \varphi$ (*the* (*c i*))
proof –
obtain *psi* **where** *psi*: *recfn 2 psi* $\bigwedge i\ x.$ *eval psi* $[i, x] = \psi\ i\ x$
using *assms* **by** *auto*
with *numbering-translation* **obtain** *b* **where**
recfn 1 b total b $\forall\, i\ x.$ *eval psi* $[i, x] =$ *eval r-phi* [*the* (*eval b* $[i]$), x]
by *blast*
moreover from *this* **obtain** *c* **where** *c*: $c \in \mathcal{R}\ \forall\, i.\ c\ i =$ *eval b* $[i]$
by *fast*
ultimately have $\psi\ i\ x = \varphi$ (*the* (*c i*)) x **for** *i x*
using *phi-def psi*(*2*) **by** *presburger*
then have $\psi\ i = \varphi$ (*the* (*c i*)) **for** *i*
by *auto*
then show *?thesis* **using** *c*(*1*) **by** *blast*
qed

corollary *goedel-numbering-phi*: *goedel-numbering* φ
unfolding *goedel-numbering-def* **using** *numbering-translation-for-phi phi-in-P2* **by** *simp*

corollary *phi-universal*:
assumes $f \in \mathcal{P}$
obtains *i* **where** $\varphi\ i = f$
using *goedel-numbering-universal*[*OF goedel-numbering-phi assms*] **by** *auto*

1.12.4 Fixed-point theorems

The fixed-point theorems look somewhat cleaner in the new notation. We will only need the following ones in the next chapter.

theorem *kleene-fixed-point*:
  fixes $k :: nat$
  assumes $\psi \in \mathcal{P}^2$
  obtains i **where** $i \geq k$ **and** $\varphi\ i = \psi\ i$
proof −
  obtain *r-psi* **where** *r-psi*: *recfn 2 r-psi* $\bigwedge i\ x.\ eval\ r\text{-}psi\ [i, x] = \psi\ i\ x$
    using *assms* **by** *auto*
  then obtain i **where** i: $i \geq k\ \forall x.\ eval\ r\text{-}phi\ [i, x] = eval\ r\text{-}psi\ [i, x]$
    using *kleene-fixed-point-theorem* **by** *blast*
  then have $\forall x.\ \varphi\ i\ x = \psi\ i\ x$
    using *phi-def r-psi* **by** *simp*
  then show *?thesis* **using** *i that* **by** *blast*
qed

theorem *smullyan-double-fixed-point*:
  assumes $g \in \mathcal{R}^2$ **and** $h \in \mathcal{R}^2$
  obtains $m\ n$ **where** $\varphi\ m = \varphi\ (the\ (g\ m\ n))$ **and** $\varphi\ n = \varphi\ (the\ (h\ m\ n))$
proof −
  obtain *rg* **where** *rg*: *recfn 2 rg total rg* $g = (\lambda x\ y.\ eval\ rg\ [x, y])$
    using *R2E*[*OF assms*(*1*)] **by** *auto*
  moreover obtain *rh* **where** *rh*: *recfn 2 rh total rh* $h = (\lambda x\ y.\ eval\ rh\ [x, y])$
    using *R2E*[*OF assms*(*2*)] **by** *auto*
  ultimately obtain $m\ n$ **where**
    $\forall x.\ eval\ r\text{-}phi\ [m, x] = eval\ r\text{-}phi\ [the\ (eval\ rg\ [m, n]), x]$
    $\forall x.\ eval\ r\text{-}phi\ [n, x] = eval\ r\text{-}phi\ [the\ (eval\ rh\ [m, n]), x]$
    using *smullyan-double-fixed-point-theorem*[*of rg rh*] **by** *blast*
  then have $\varphi\ m = \varphi\ (the\ (g\ m\ n))$ **and** $\varphi\ n = \varphi\ (the\ (h\ m\ n))$
    using *phi-def rg rh* **by** *auto*
  then show *?thesis* **using** *that* **by** *simp*
qed

end

Chapter 2

Inductive inference of recursive functions

theory *Inductive-Inference-Basics*
  imports *Standard-Results*
begin

Inductive inference originates from work by Solomonoff [13, 14] and Gold [9, 8] and comes in many variations. The common theme is to infer additional information about objects, such as formal languages or functions, from incomplete data, such as finitely many words contained in the language or argument-value pairs of the function. Oftentimes "additional information" means complete information, such that the task becomes identification of the object.

The basic setting in inductive inference of recursive functions is as follows. Let us denote, for a total function f, by f^n the code of the list $[f(0), ..., f(n)]$. Let U be a set (called *class*) of total recursive functions, and ψ a binary partial recursive function (called *hypothesis space*). A partial recursive function S (called *strategy*) is said to *learn U in the limit with respect to ψ* if for all $f \in U$,

- the value $S(f^n)$ is defined for all $n \in \mathbb{N}$,
- the sequence $S(f^0), S(f^1), \ldots$ converges to an $i \in \mathbb{N}$ with $\psi_i = f$.

Both the output $S(f^n)$ of the strategy and its interpretation as a function $\psi_{S(f^n)}$ are called *hypothesis*. The set of all classes learnable in the limit by S with respect to ψ is denoted by $\mathrm{LIM}_\psi(S)$. Moreover we set $\mathrm{LIM}_\psi = \bigcup_{S \in \mathcal{P}} \mathrm{LIM}_\psi(S)$ and $\mathrm{LIM} = \bigcup_{\psi \in \mathcal{P}^2} \mathrm{LIM}_\psi$. We call the latter set the *inference type* LIM.

Many aspects of this setting can be varied. We shall consider:

- Intermediate hypotheses: $\psi_{S(f^n)}$ can be required to be total or to be in the class U, or to coincide with f on arguments up to n, or a myriad of other conditions or combinations thereof.
- Convergence of hypotheses:
  - The strategy can be required to output not a sequence but a single hypothesis, which must be correct.
  - The strategy can be required to converge to a *function* rather than an index.

We formalize five kinds of results ($\mathcal{I}$ and $\mathcal{I}'$ stand for inference types):

- Comparison of learning power: results of the form $\mathcal{I} \subset \mathcal{I}'$, in particular showing that the inclusion is proper (Sections 2.3, 2.4, 2.5, 2.6, 2.7, 2.9, 2.10, 2.11).
- Whether $\mathcal{I}$ is closed under the subset relation: $U \in \mathcal{I} \wedge V \subseteq U \Longrightarrow V \in \mathcal{I}$.
- Whether $\mathcal{I}$ is closed under union: $U \in \mathcal{I} \wedge V \in \mathcal{I} \Longrightarrow U \cup V \in \mathcal{I}$ (Section 2.12).
- Whether every class in $\mathcal{I}$ can be learned with respect to a Gödel numbering as hypothesis space (Section 2.2).
- Whether every class in $\mathcal{I}$ can be learned by a *total* recursive strategy (Section 2.8).

The bulk of this chapter is devoted to the first category of results. Most results that we are going to formalize have been called "classical" by Jantke and Beick [10], who compare a large number of inference types. Another comparison is by Case and Smith [6]. Angluin and Smith [1] give an overview of various forms of inductive inference.

All (interesting) proofs herein are based on my lecture notes of the *Induktive Inferenz* lectures by Rolf Wiehagen from 1999/2000 and 2000/2001 at the University of Kaiserslautern. I have given references to the original proofs whenever I was able to find them. For the other proofs, as well as for those that I had to contort beyond recognition, I provide proof sketches.

2.1 Preliminaries

Throughout the chapter, in particular in proof sketches, we use the following notation.

Let $b \in \mathbb{N}^*$ be a list of numbers. We write $|b|$ for its length and b_i for the i-th element ($i = 0, \dots, |b| - 1$). Concatenation of numbers and lists works in the obvious way; for instance, jbk with $j, k \in \mathbb{N}$, $b \in \mathbb{N}^*$ refers to the list $jb_0 \dots b_{|b|-1}k$. For $0 \le i < |b|$, the term $b_{i:=v}$ denotes the list $b_0 \dots b_{i-1} v b_{i+1} \dots b_{|b|-1}$. The notation $b_{<i}$ refers to $b_0 \dots b_{i-1}$ for $0 < i \le |b|$. Moreover, v^n is short for the list consisting of n times the value $v \in \mathbb{N}$.

Unary partial functions can be regarded as infinite sequences consisting of numbers and the symbol $\uparrow$ denoting undefinedness. We abbreviate the empty function by $\uparrow^\infty$ and the constant zero function by 0^∞. A function can be written as a list concatenated with a partial function. For example, $jb \uparrow^\infty$ is the function

$$x \mapsto \begin{cases} j & \text{if } x = 0, \\ b_{x-1} & \text{if } 0 < x \le |b|, \\ \uparrow & \text{otherwise,} \end{cases}$$

and jp, where p is a function, means

$$x \mapsto \begin{cases} j & \text{if } x = 0, \\ p(x-1) & \text{otherwise.} \end{cases}$$

A *numbering* is a function $\psi \in \mathcal{P}^2$.

2.1.1 The prefixes of a function

A *prefix*, also called *initial segment*, is a list of initial values of a function.

definition *prefix* :: *partial1* $\Rightarrow$ *nat* $\Rightarrow$ *nat list* **where**

prefix f n ≡ *map* (λ*x*. *the* (*f x*)) [*0*..<*Suc n*]

lemma *length-prefix* [*simp*]: *length* (*prefix f n*) = *Suc n*
unfolding *prefix-def* **by** *simp*

lemma *prefix-nth* [*simp*]:
assumes *k* < *Suc n*
shows *prefix f n* ! *k* = *the* (*f k*)
unfolding *prefix-def* **using** *assms nth-map-upt*[*of k Suc n 0* λ*x*. *the* (*f x*)] **by** *simp*

lemma *prefixI*:
assumes *length vs* > *0* **and** ⋀*x*. *x* < *length vs* ⟹ *f x* ↓= *vs* ! *x*
shows *prefix f* (*length vs* − *1*) = *vs*
using *assms nth-equalityI*[*of prefix f* (*length vs* − *1*) *vs*] **by** *simp*

lemma *prefixI'*:
assumes *length vs* = *Suc n* **and** ⋀*x*. *x* < *Suc n* ⟹ *f x* ↓= *vs* ! *x*
shows *prefix f n* = *vs*
using *assms nth-equalityI*[*of prefix f* (*length vs* − *1*) *vs*] **by** *simp*

lemma *prefixE*:
assumes *prefix f* (*length vs* − *1*) = *vs*
and *f* ∈ $\mathcal{R}$
and *length vs* > *0*
and *x* < *length vs*
shows *f x* ↓= *vs* ! *x*
using *assms length-prefix prefix-nth*[*of x length vs* − *1 f*] **by** *simp*

lemma *prefix-eqI*:
assumes ⋀*x*. *x* ≤ *n* ⟹ *f x* = *g x*
shows *prefix f n* = *prefix g n*
using *assms prefix-def* **by** *simp*

lemma *prefix-0*: *prefix f 0* = [*the* (*f 0*)]
using *prefix-def* **by** *simp*

lemma *prefix-Suc*: *prefix f* (*Suc n*) = *prefix f n* @ [*the* (*f* (*Suc n*))]
unfolding *prefix-def* **by** *simp*

lemma *take-prefix*:
assumes *f* ∈ $\mathcal{R}$ **and** *k* ≤ *n*
shows *prefix f k* = *take* (*Suc k*) (*prefix f n*)
proof −
let *?vs* = *take* (*Suc k*) (*prefix f n*)
have *length ?vs* = *Suc k*
using *assms*(*2*) **by** *simp*
then have ⋀*x*. *x* < *length ?vs* ⟹ *f x* ↓= *?vs* ! *x*
using *assms* **by** *auto*
then show *?thesis*
using *prefixI*[**where** *?vs*=*?vs*] ‹*length ?vs* = *Suc k*› **by** *simp*
qed

Strategies receive prefixes in the form of encoded lists. The term "prefix" refers to both encoded and unencoded lists. We use the notation $f \triangleright n$ for the prefix f^n.

definition *init* :: *partial1* ⇒ *nat* ⇒ *nat* (**infix** ‹▷› *110*) **where**
f ▷ *n* ≡ *list-encode* (*prefix f n*)

lemma *init-neq-zero*: $f \triangleright n \neq 0$
unfolding *init-def prefix-def* **using** *list-encode-0* **by** *fastforce*

lemma *init-prefixE* [*elim*]: $\mathit{prefix}\ f\ n = \mathit{prefix}\ g\ n \Longrightarrow f \triangleright n = g \triangleright n$
unfolding *init-def* **by** *simp*

lemma *init-eqI*:
assumes $\bigwedge x.\ x \leq n \Longrightarrow f\ x = g\ x$
shows $f \triangleright n = g \triangleright n$
unfolding *init-def* **using** *prefix-eqI*[*OF assms*] **by** *simp*

lemma *initI*:
assumes $\mathit{e\text{-}length}\ e > 0$ **and** $\bigwedge x.\ x < \mathit{e\text{-}length}\ e \Longrightarrow f\ x \downarrow= \mathit{e\text{-}nth}\ e\ x$
shows $f \triangleright (\mathit{e\text{-}length}\ e - 1) = e$
unfolding *init-def* **using** *assms prefixI* **by** *simp*

lemma *initI'*:
assumes $\mathit{e\text{-}length}\ e = \mathit{Suc}\ n$ **and** $\bigwedge x.\ x < \mathit{Suc}\ n \Longrightarrow f\ x \downarrow= \mathit{e\text{-}nth}\ e\ x$
shows $f \triangleright n = e$
unfolding *init-def* **using** *assms prefixI'* **by** *simp*

lemma *init-iff-list-eq-upto*:
assumes $f \in \mathcal{R}$ **and** $\mathit{e\text{-}length}\ vs > 0$
shows $(\forall x{<}\mathit{e\text{-}length}\ vs.\ f\ x \downarrow= \mathit{e\text{-}nth}\ vs\ x) \longleftrightarrow \mathit{prefix}\ f\ (\mathit{e\text{-}length}\ vs - 1) = \mathit{list\text{-}decode}\ vs$
using *prefixI*[*OF assms*(*2*)] *prefixE*[*OF - assms*] **by** *auto*

lemma *length-init* [*simp*]: $\mathit{e\text{-}length}\ (f \triangleright n) = \mathit{Suc}\ n$
unfolding *init-def* **by** *simp*

lemma *init-Suc-snoc*: $f \triangleright (\mathit{Suc}\ n) = \mathit{e\text{-}snoc}\ (f \triangleright n)\ (\mathit{the}\ (f\ (\mathit{Suc}\ n)))$
unfolding *init-def* **by** (*simp add*: *prefix-Suc*)

lemma *nth-init*: $i < \mathit{Suc}\ n \Longrightarrow \mathit{e\text{-}nth}\ (f \triangleright n)\ i = \mathit{the}\ (f\ i)$
unfolding *init-def* **using** *prefix-nth* **by** *auto*

lemma *hd-init* [*simp*]: $\mathit{e\text{-}hd}\ (f \triangleright n) = \mathit{the}\ (f\ 0)$
unfolding *init-def* **using** *init-neq-zero* **by** (*simp add*: *e-hd-nth0*)

lemma *list-decode-init* [*simp*]: $\mathit{list\text{-}decode}\ (f \triangleright n) = \mathit{prefix}\ f\ n$
unfolding *init-def* **by** *simp*

lemma *init-eq-iff-eq-upto*:
assumes $g \in \mathcal{R}$ **and** $f \in \mathcal{R}$
shows $(\forall j{<}\mathit{Suc}\ n.\ g\ j = f\ j) \longleftrightarrow g \triangleright n = f \triangleright n$
using *assms initI' init-iff-list-eq-upto length-init list-decode-init*
by (*metis diff-Suc-1 zero-less-Suc*)

definition *is-init-of* :: $\mathit{nat} \Rightarrow \mathit{partial1} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\text{-}init\text{-}of}\ t\ f \equiv \forall i{<}\mathit{e\text{-}length}\ t.\ f\ i \downarrow= \mathit{e\text{-}nth}\ t\ i$

lemma *not-initial-imp-not-eq*:
assumes $\bigwedge x.\ x < \mathit{Suc}\ n \Longrightarrow f\ x \downarrow$ **and** $\neg\ (\mathit{is\text{-}init\text{-}of}\ (f \triangleright n)\ g)$
shows $f \neq g$
using *is-init-of-def assms* **by** *auto*

lemma *all-init-eq-imp-fun-eq*:
assumes $f \in \mathcal{R}$ **and** $g \in \mathcal{R}$ **and** $\bigwedge n.\ f \triangleright n = g \triangleright n$
shows $f = g$
proof
fix *n*
from *assms* **have** *prefix f n* = *prefix g n*
by (*metis init-def list-decode-encode*)
then have *the* (*f n*) = *the* (*g n*)
unfolding *init-def prefix-def* **by** *simp*
then show *f n* = *g n*
using *assms*(*1*,*2*) **by** (*meson R1-imp-total1 option.expand total1E*)
qed

corollary *neq-fun-neq-init*:
assumes $f \in \mathcal{R}$ **and** $g \in \mathcal{R}$ **and** $f \neq g$
shows $\exists\, n.\ f \triangleright n \neq g \triangleright n$
using *assms all-init-eq-imp-fun-eq* **by** *auto*

lemma *eq-init-forall-le*:
assumes $f \triangleright n = g \triangleright n$ **and** $m \leq n$
shows $f \triangleright m = g \triangleright m$
proof −
from *assms*(*1*) **have** *prefix f n* = *prefix g n*
by (*metis init-def list-decode-encode*)
then have *the* (*f k*) = *the* (*g k*) **if** $k \leq n$ **for** *k*
using *prefix-def that* **by** *auto*
then have *the* (*f k*) = *the* (*g k*) **if** $k \leq m$ **for** *k*
using *assms*(*2*) *that* **by** *simp*
then have *prefix f m* = *prefix g m*
using *prefix-def* **by** *simp*
then show *?thesis* **by** (*simp add*: *init-def*)
qed

corollary *neq-init-forall-ge*:
assumes $f \triangleright n \neq g \triangleright n$ **and** $m \geq n$
shows $f \triangleright m \neq g \triangleright m$
using *eq-init-forall-le assms* **by** *blast*

lemma *e-take-init*:
assumes $f \in \mathcal{R}$ **and** $k <$ *Suc n*
shows *e-take* (*Suc k*) ($f \triangleright n$) = $f \triangleright k$
using *assms take-prefix* **by** (*simp add*: *init-def less-Suc-eq-le*)

lemma *init-butlast-init*:
assumes *total1 f* **and** $f \triangleright n = e$ **and** $n > 0$
shows $f \triangleright (n - 1)$ = *e-butlast e*
proof −
let *?e* = *e-butlast e*
have *e-length e* = *Suc n*
using *assms*(*2*) **by** *auto*
then have *len*: *e-length ?e* = *n*
by *simp*
have $f \triangleright$ (*e-length ?e* − *1*) = *?e*
proof (*rule initI*)
show *0* < *e-length ?e*
using *assms*(*3*) *len* **by** *simp*

have $\bigwedge x.\ x < \textit{e-length } e \Longrightarrow f\ x \downarrow= \textit{e-nth } e\ x$
  using *assms*(*1*,*2*) *total1-def* ‹*e-length e* = *Suc n*› **by** *auto*
then show $\bigwedge x.\ x < \textit{e-length } ?e \Longrightarrow f\ x \downarrow= \textit{e-nth } ?e\ x$
  by (*simp add*: *butlast-conv-take*)
qed
with *len* **show** *?thesis* **by** *simp*
qed

Some definitions make use of recursive predicates, that is, 01-valued functions.

definition *RPred1* :: *partial1 set* (‹$\mathcal{R}_{01}$›) **where**
$\mathcal{R}_{01} \equiv \{f.\ f \in \mathcal{R} \wedge (\forall x.\ f\ x \downarrow= 0 \vee f\ x \downarrow= 1)\}$

lemma *RPred1-subseteq-R1*: $\mathcal{R}_{01} \subseteq \mathcal{R}$
unfolding *RPred1-def* **by** *auto*

lemma *const0-in-RPred1*: (λ-. *Some 0*) $\in \mathcal{R}_{01}$
using *RPred1-def const-in-Prim1* **by** *fast*

lemma *RPred1-altdef*: $\mathcal{R}_{01} = \{f.\ f \in \mathcal{R} \wedge (\forall x.\ \textit{the } (f\ x) \le 1)\}$
(**is** $\mathcal{R}_{01}$ = *?S*)
proof
show $\mathcal{R}_{01} \subseteq$ *?S*
proof
fix *f*
assume *f*: $f \in \mathcal{R}_{01}$
with *RPred1-def* **have** $f \in \mathcal{R}$ **by** *auto*
from *f* **have** $\forall x.\ f\ x \downarrow= 0 \vee f\ x \downarrow= 1$
  by (*simp add*: *RPred1-def*)
then have $\forall x.\ \textit{the } (f\ x) \le 1$
  by (*metis eq-refl less-Suc-eq-le zero-less-Suc option.sel*)
with ‹$f \in \mathcal{R}$› **show** $f \in$ *?S* **by** *simp*
qed
show *?S* $\subseteq \mathcal{R}_{01}$
proof
fix *f*
assume *f*: $f \in$ *?S*
then have $f \in \mathcal{R}$ **by** *simp*
then have *total*: $\bigwedge x.\ f\ x \downarrow$ **by** *auto*
from *f* **have** $\forall x.\ \textit{the } (f\ x) = 0 \vee \textit{the } (f\ x) = 1$
  by (*simp add*: *le-eq-less-or-eq*)
with *total* **have** $\forall x.\ f\ x \downarrow= 0 \vee f\ x \downarrow= 1$
  by (*metis option.collapse*)
then show $f \in \mathcal{R}_{01}$
  using ‹$f \in \mathcal{R}$› *RPred1-def* **by** *auto*
qed
qed

2.1.2 NUM

A class of recursive functions is in NUM if it can be embedded in a total numbering. Thus, for learning such classes there is always a total hypothesis space available.

definition *NUM* :: *partial1 set set* **where**
$NUM \equiv \{U.\ \exists \psi \in \mathcal{R}^2.\ \forall f \in U.\ \exists i.\ \psi\ i = f\}$

definition *NUM-wrt* :: *partial2* $\Rightarrow$ *partial1 set set* **where**

$\psi \in \mathcal{R}^2 \Longrightarrow$ *NUM-wrt* $\psi \equiv \{U.\ \forall f{\in}U.\ \exists\, i.\ \psi\ i = f\}$

lemma *NUM-I* [*intro*]:
assumes $\psi \in \mathcal{R}^2$ **and** $\bigwedge f.\ f \in U \Longrightarrow \exists\, i.\ \psi\ i = f$
shows $U \in NUM$
using *assms NUM-def* **by** *blast*

lemma *NUM-E* [*dest*]:
assumes $U \in NUM$
shows $U \subseteq \mathcal{R}$
and $\exists\, \psi{\in}\mathcal{R}^2.\ \forall f{\in}U.\ \exists\, i.\ \psi\ i = f$
using *NUM-def assms* **by** (*force*, *auto*)

lemma *NUM-closed-subseteq*:
assumes $U \in NUM$ **and** $V \subseteq U$
shows $V \in NUM$
using *assms subset-eq*[*of V U*] *NUM-I* **by** *auto*

This is the classical diagonalization proof showing that there is no total numbering containing all total recursive functions.

lemma *R1-not-in-NUM*: $\mathcal{R} \notin NUM$
proof
assume $\mathcal{R} \in NUM$
then obtain ψ **where** *num*: $\psi \in \mathcal{R}^2$ $\forall f{\in}\mathcal{R}.\ \exists\, i.\ \psi\ i = f$
by *auto*
then obtain *psi* **where** *psi*: *recfn 2 psi total psi eval psi* $[i,\ x] = \psi\ i\ x$ **for** *i x*
by *auto*
define *d* **where** $d = $ *Cn 1 S* [*Cn 1 psi* [*Id 1 0*, *Id 1 0*]]
then have *recfn 1 d*
using *psi*(*1*) **by** *simp*
moreover have *d*: *eval d* $[x] \downarrow=$ *Suc* (*the* $(\psi\ x\ x)$) **for** *x*
unfolding *d-def* **using** *num psi* **by** *simp*
ultimately have $(\lambda x.\ eval\ d\ [x]) \in \mathcal{R}$
using *R1I* **by** *blast*
then obtain *i* **where** $\psi\ i = (\lambda x.\ eval\ d\ [x])$
using *num*(*2*) **by** *auto*
then have $\psi\ i\ i = eval\ d\ [i]$ **by** *simp*
with *d* **have** $\psi\ i\ i \downarrow=$ *Suc* (*the* $(\psi\ i\ i)$) **by** *simp*
then show *False*
using *option.sel*[*of Suc* (*the* $(\psi\ i\ i)$)] **by** *simp*
qed

A hypothesis space that contains a function for every prefix will come in handy. The following is a total numbering with this property.

definition *r-prenum* $\equiv$
Cn 2 r-ifless [*Id 2 1*, *Cn 2 r-length* [*Id 2 0*], *Cn 2 r-nth* [*Id 2 0*, *Id 2 1*], *r-constn 1 0*]

lemma *r-prenum-prim* [*simp*]: *prim-recfn 2 r-prenum*
unfolding *r-prenum-def* **by** *simp-all*

lemma *r-prenum* [*simp*]:
eval r-prenum $[e,\ x] \downarrow=$ (*if* $x <$ *e-length e then e-nth e x else 0*)
by (*simp add*: *r-prenum-def*)

definition *prenum* :: *partial2* **where**

prenum e x ≡ *Some* (*if x* < *e-length e then e-nth e x else 0*)

lemma *prenum-in-R2*: *prenum* ∈ $\mathcal{R}^2$
using *prenum-def Prim2I*[*OF r-prenum-prim*, *of prenum*] **by** *simp*

lemma *prenum* [*simp*]: *prenum e x* ↓= (*if x* < *e-length e then e-nth e x else 0*)
unfolding *prenum-def* **..**

lemma *prenum-encode*:
prenum (*list-encode vs*) *x* ↓= (*if x* < *length vs then vs* ! *x else 0*)
using *prenum-def* **by** (*cases x* < *length vs*) *simp-all*

Prepending a list of numbers to a function:

definition *prepend* :: *nat list* ⇒ *partial1* ⇒ *partial1* (**infixr** ‹⊙› *64*) **where**
vs ⊙ *f* ≡ λ*x*. *if x* < *length vs then Some* (*vs* ! *x*) *else f* (*x* − *length vs*)

lemma *prepend* [*simp*]:
(*vs* ⊙ *f*) *x* = (*if x* < *length vs then Some* (*vs* ! *x*) *else f* (*x* − *length vs*))
unfolding *prepend-def* **..**

lemma *prepend-total*: *total1 f* ⟹ *total1* (*vs* ⊙ *f*)
unfolding *total1-def* **by** *simp*

lemma *prepend-at-less*:
assumes *n* < *length vs*
shows (*vs* ⊙ *f*) *n* ↓= *vs* ! *n*
using *assms* **by** *simp*

lemma *prepend-at-ge*:
assumes *n* ≥ *length vs*
shows (*vs* ⊙ *f*) *n* = *f* (*n* − *length vs*)
using *assms* **by** *simp*

lemma *prefix-prepend-less*:
assumes *n* < *length vs*
shows *prefix* (*vs* ⊙ *f*) *n* = *take* (*Suc n*) *vs*
using *assms length-prefix* **by** (*intro nth-equalityI*) *simp-all*

lemma *prepend-eqI*:
assumes ⋀*x*. *x* < *length vs* ⟹ *g x* ↓= *vs* ! *x*
and ⋀*x*. *g* (*length vs* + *x*) = *f x*
shows *g* = *vs* ⊙ *f*
proof
fix *x*
show *g x* = (*vs* ⊙ *f*) *x*
proof (*cases x* < *length vs*)
case *True*
then show *?thesis* **using** *assms* **by** *simp*
next
case *False*
then show *?thesis*
using *assms prepend* **by** (*metis add-diff-inverse-nat*)
qed
qed

fun *r-prepend* :: *nat list* ⇒ *recf* ⇒ *recf* **where**

```
  r-prepend [] r = r
| r-prepend (v # vs) r =
    Cn 1 (r-lifz (r-const v) (Cn 1 (r-prepend vs r) [r-dec])) [Id 1 0, Id 1 0]

lemma r-prepend-recfn:
  assumes recfn 1 r
  shows recfn 1 (r-prepend vs r)
  using assms by (induction vs) simp-all

lemma r-prepend:
  assumes recfn 1 r
  shows eval (r-prepend vs r) [x] =
    (if x < length vs then Some (vs ! x) else eval r [x − length vs])
proof (induction vs arbitrary: x)
  case Nil
  then show ?case using assms by simp
next
  case (Cons v vs)
  show ?case
    using assms Cons by (cases x = 0) (auto simp add: r-prepend-recfn)
qed

lemma r-prepend-total:
  assumes recfn 1 r and total r
  shows eval (r-prepend vs r) [x] ↓=
    (if x < length vs then vs ! x else the (eval r [x − length vs]))
proof (induction vs arbitrary: x)
  case Nil
  then show ?case using assms by simp
next
  case (Cons v vs)
  show ?case
    using assms Cons by (cases x = 0) (auto simp add: r-prepend-recfn)
qed

lemma prepend-in-P1:
  assumes f ∈ 𝒫
  shows vs ⊙ f ∈ 𝒫
proof −
  obtain r where r: recfn 1 r  ⋀x. eval r [x] = f x
    using assms by auto
  moreover have recfn 1 (r-prepend vs r)
    using r r-prepend-recfn by simp
  moreover have eval (r-prepend vs r) [x] = (vs ⊙ f) x for x
    using r r-prepend by simp
  ultimately show ?thesis by blast
qed

lemma prepend-in-R1:
  assumes f ∈ ℛ
  shows vs ⊙ f ∈ ℛ
proof −
  obtain r where r: recfn 1 r total r ⋀x. eval r [x] = f x
    using assms by auto
  then have total1 f
    using R1-imp-total1[OF assms] by simp
```

have *total* (*r-prepend vs r*)
  using *r r-prepend-total r-prepend-recfn totalI1*[*of r-prepend vs r*] **by** *simp*
with *r* **have** *total* (*r-prepend vs r*) **by** *simp*
moreover have *recfn 1* (*r-prepend vs r*)
  using *r r-prepend-recfn* **by** *simp*
moreover have *eval* (*r-prepend vs r*) [*x*] = (*vs* ⊙ *f*) *x* **for** *x*
  using *r r-prepend* ‹*total1 f*› *total1E* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

lemma *prepend-associative*: (*us* @ *vs*) ⊙ *f* = *us* ⊙ *vs* ⊙ *f* (**is** *?lhs* = *?rhs*)
proof
  fix *x*
  consider
      x < *length us*
    | *x* ≥ *length us* ∧ *x* < *length* (*us* @ *vs*)
    | *x* ≥ *length* (*us* @ *vs*)
    by *linarith*
  then show *?lhs x* = *?rhs x*
  proof (*cases*)
    case *1*
    then show *?thesis*
      by (*metis le-add1 length-append less-le-trans nth-append prepend-at-less*)
  next
    case *2*
    then show *?thesis*
      by (*smt add-diff-inverse-nat add-less-cancel-left length-append nth-append prepend*)
  next
    case *3*
    then show *?thesis*
      using *prepend-at-ge* **by** *auto*
  qed
qed

abbreviation *constant-divergent* :: *partial1* (‹↑$^\infty$›) **where**
  ↑$^\infty$ ≡ λ-. *None*

abbreviation *constant-zero* :: *partial1* (‹*0*$^\infty$›) **where**
  0$^\infty$ ≡ λ-. *Some 0*

lemma *almost0-in-R1*: *vs* ⊙ *0*$^\infty$ ∈ $\mathcal{R}$
  using *RPred1-subseteq-R1 const0-in-RPred1 prepend-in-R1* **by** *auto*

The class U_0 of all total recursive functions that are almost everywhere zero will be used several times to construct (counter-)examples.

definition *U0* :: *partial1 set* (‹U_0›) **where**
  U_0 ≡ {*vs* ⊙ *0*$^\infty$ |*vs*. *vs* ∈ *UNIV*}

The class U_0 contains exactly the functions in the numbering *prenum*.

lemma *U0-altdef*: U_0 = {*prenum e*| *e*. *e* ∈ *UNIV*} (**is** U_0 = *?W*)
proof
  show U_0 ⊆ *?W*
  proof
    fix *f*
    assume *f* ∈ U_0

with *U0-def* **obtain** *vs* **where** $f = vs \odot 0^\infty$
by *auto*
then have $f = prenum\ (list\text{-}encode\ vs)$
using *prenum-encode* **by** *auto*
then show $f \in ?W$ **by** *auto*
qed
show $?W \subseteq U_0$
unfolding *U0-def* **by** *fastforce*
qed

lemma *U0-in-NUM*: $U_0 \in NUM$
using *prenum-in-R2 U0-altdef* **by** (*intro NUM-I*[*of prenum*]; *force*)

Every almost-zero function can be represented by $v0^\infty$ for a list v not ending in zero.

lemma *almost0-canonical*:
assumes $f = vs \odot 0^\infty$ **and** $f \neq 0^\infty$
obtains *ws* **where** $length\ ws > 0$ **and** $last\ ws \neq 0$ **and** $f = ws \odot 0^\infty$
proof −
let $?P = \lambda k.\ k < length\ vs \wedge vs\ !\ k \neq 0$
from *assms* **have** $vs \neq []$
by *auto*
then have *ex*: $\exists k<length\ vs.\ vs\ !\ k \neq 0$
using *assms* **by** *auto*
define *m* **where** $m = Greatest\ ?P$
moreover have *le*: $\forall y.\ ?P\ y \longrightarrow y \leq length\ vs$
by *simp*
ultimately have $?P\ m$
using *ex GreatestI-ex-nat*[*of ?P length vs*] **by** *simp*
have *not-gr*: $\neg\ ?P\ k$ **if** $k > m$ **for** *k*
using *Greatest-le-nat*[*of ?P - length vs*] *m-def ex le not-less that* **by** *blast*
let $?ws = take\ (Suc\ m)\ vs$
have $vs \odot 0^\infty = ?ws \odot 0^\infty$
proof
fix *x*
show $(vs \odot 0^\infty)\ x = (?ws \odot 0^\infty)\ x$
proof (*cases* $x < Suc\ m$)
case *True*
then show *?thesis* **using** ‹*?P m*› **by** *simp*
next
case *False*
moreover from *this* **have** $(?ws \odot 0^\infty)\ x \downarrow= 0$
by *simp*
ultimately show *?thesis*
using *not-gr* **by** (*cases* $x < length\ vs$) *simp-all*
qed
qed
then have $f = ?ws \odot 0^\infty$
using *assms*(*1*) **by** *simp*
moreover have $length\ ?ws > 0$
by (*simp add*: ‹$vs \neq []$›)
moreover have $last\ ?ws \neq 0$
by (*simp add*: ‹*?P m*› *take-Suc-conv-app-nth*)
ultimately show *?thesis* **using** *that* **by** *blast*
qed

2.2 Types of inference

This section introduces all inference types that we are going to consider together with some of their simple properties. All these inference types share the following condition, which essentially says that everything must be computable:

abbreviation *environment* :: *partial2* $\Rightarrow$ (*partial1 set*) $\Rightarrow$ *partial1* $\Rightarrow$ *bool* **where**
environment ψ *U s* $\equiv \psi \in \mathcal{P}^2 \wedge U \subseteq \mathcal{R} \wedge s \in \mathcal{P} \wedge (\forall f{\in}U.\ \forall n.\ s\ (f \triangleright n) \downarrow)$

2.2.1 LIM: Learning in the limit

A strategy S learns a class U in the limit with respect to a hypothesis space $\psi \in \mathcal{P}^2$ if for all $f \in U$, the sequence $(S(f^n))_{n\in\mathbb{N}}$ converges to an i with $\psi_i = f$. Convergence for a sequence of natural numbers means that almost all elements are the same. We express this with the following notation.

abbreviation *Almost-All* :: (*nat* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* (**binder** ‹$\forall^\infty$› *10*) **where**
$\forall^\infty n.\ P\ n \equiv \exists n_0.\ \forall n{\geq}n_0.\ P\ n$

definition *learn-lim* :: *partial2* $\Rightarrow$ (*partial1 set*) $\Rightarrow$ *partial1* $\Rightarrow$ *bool* **where**
learn-lim ψ *U s* $\equiv$
environment ψ *U s* $\wedge$
$(\forall f{\in}U.\ \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i))$

lemma *learn-limE*:
assumes *learn-lim* ψ *U s*
shows *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i)$
using *assms learn-lim-def* **by** *auto*

lemma *learn-limI*:
assumes *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i)$
shows *learn-lim* ψ *U s*
using *assms learn-lim-def* **by** *auto*

definition *LIM-wrt* :: *partial2* $\Rightarrow$ *partial1 set set* **where**
LIM-wrt $\psi \equiv \{U.\ \exists s.$ *learn-lim* ψ *U s*$\}$

definition *Lim* :: *partial1 set set* (‹*LIM*›) **where**
LIM $\equiv \{U.\ \exists \psi\ s.$ *learn-lim* ψ *U s*$\}$

LIM is closed under the the subset relation.

lemma *learn-lim-closed-subseteq*:
assumes *learn-lim* ψ *U s* **and** $V \subseteq U$
shows *learn-lim* ψ *V s*
using *assms learn-lim-def* **by** *auto*

corollary *LIM-closed-subseteq*:
assumes $U \in$ *LIM* **and** $V \subseteq U$
shows $V \in$ *LIM*
using *assms learn-lim-closed-subseteq* **by** (*smt Lim-def mem-Collect-eq*)

Changing the hypothesis infinitely often precludes learning in the limit.

lemma *infinite-hyp-changes-not-Lim*:
assumes $f \in U$ **and** $\forall n.\ \exists m_1{>}n.\ \exists m_2{>}n.\ s\ (f \triangleright m_1) \neq s\ (f \triangleright m_2)$

shows $\neg$ *learn-lim* ψ *U s*
using *assms learn-lim-def* **by** (*metis less-imp-le*)

lemma *always-hyp-change-not-Lim*:
assumes $\bigwedge x.\ s\ (f \triangleright (Suc\ x)) \neq s\ (f \triangleright x)$
shows $\neg$ *learn-lim* $\psi\ \{f\}\ s$
using *assms learn-limE* **by** (*metis le-SucI order-refl singletonI*)

Guessing a wrong hypothesis infinitely often precludes learning in the limit.

lemma *infinite-hyp-wrong-not-Lim*:
assumes $f \in U$ **and** $\forall n.\ \exists m{>}n.\ \psi\ (the\ (s\ (f \triangleright m))) \neq f$
shows $\neg$ *learn-lim* ψ *U s*
using *assms learn-limE* **by** (*metis less-imp-le option.sel*)

Converging to the same hypothesis on two functions precludes learning in the limit.

lemma *same-hyp-for-two-not-Lim*:
assumes $f_1 \in U$
and $f_2 \in U$
and $f_1 \neq f_2$
and $\forall n{\geq}n_1.\ s\ (f_1 \triangleright n) = h$
and $\forall n{\geq}n_2.\ s\ (f_2 \triangleright n) = h$
shows $\neg$ *learn-lim* ψ *U s*
using *assms learn-limE* **by** (*metis le-cases option.sel*)

Every class that can be learned in the limit can be learned in the limit with respect to any Gödel numbering. We prove a generalization in which hypotheses may have to satisfy an extra condition, so we can re-use it for other inference types later.

lemma *learn-lim-extra-wrt-goedel*:
fixes *extra* :: (*partial1 set*) $\Rightarrow$ *partial1* $\Rightarrow$ *nat* $\Rightarrow$ *partial1* $\Rightarrow$ *bool*
assumes *goedel-numbering* χ
and *learn-lim* ψ *U s*
and $\bigwedge f\ n.\ f \in U \Longrightarrow extra\ U\ f\ n\ (\psi\ (the\ (s\ (f \triangleright n))))$
shows $\exists t.\ learn\text{-}lim\ \chi\ U\ t \wedge (\forall f{\in}U.\ \forall n.\ extra\ U\ f\ n\ (\chi\ (the\ (t\ (f \triangleright n)))))$
proof −
have *env*: *environment* ψ *U s*
and *lim*: *learn-lim* ψ *U s*
and *extra*: $\forall f{\in}U.\ \forall n.\ extra\ U\ f\ n\ (\psi\ (the\ (s\ (f \triangleright n))))$
using *assms learn-limE* **by** *auto*
obtain *c* **where** *c*: $c \in \mathcal{R}\ \forall i.\ \psi\ i = \chi\ (the\ (c\ i))$
using *env goedel-numberingE*[*OF assms*(*1*), *of* ψ] **by** *auto*
define *t* **where** $t \equiv$
$(\lambda x.\ if\ s\ x \downarrow \wedge\ c\ (the\ (s\ x)) \downarrow\ then\ Some\ (the\ (c\ (the\ (s\ x))))\ else\ None)$
have $t \in \mathcal{P}$
unfolding *t-def* **using** *env c concat-P1-P1*[*of c s*] **by** *auto*
have $t\ x = (if\ s\ x \downarrow\ then\ Some\ (the\ (c\ (the\ (s\ x))))\ else\ None)$ **for** *x*
using *t-def c*(*1*) *R1-imp-total1* **by** *auto*
then have *t*: $t\ (f \triangleright n) \downarrow= the\ (c\ (the\ (s\ (f \triangleright n))))$ **if** $f \in U$ **for** *f n*
using *lim learn-limE that* **by** *simp*
have *learn-lim* χ *U t*
proof (*rule learn-limI*)
show *environment* χ *U t*
using *t* **by** (*simp add*: ‹$t \in \mathcal{P}$› *env goedel-numbering-P2*[*OF assms*(*1*)])
show $\exists i.\ \chi\ i = f \wedge (\forall^\infty n.\ t\ (f \triangleright n) \downarrow= i)$ **if** $f \in U$ **for** *f*
proof −
from *lim learn-limE*(*2*) **obtain** $i\ n_0$ **where**

i: $\psi\ i = f \wedge (\forall n \geq n_0.\ s\ (f \triangleright n) \downarrow= i)$
using ‹$f \in U$› **by** *blast*
let $?j = the\ (c\ i)$
have $\chi\ ?j = f$
using $c(2)\ i$ **by** *simp*
moreover have $t\ (f \triangleright n) \downarrow= ?j$ **if** $n \geq n_0$ **for** n
by (*simp add*: ‹$f \in U$› *i t that*)
ultimately show *?thesis* **by** *auto*
qed
qed
moreover have $extra\ U\ f\ n\ (\chi\ (the\ (t\ (f \triangleright n))))$ **if** $f \in U$ **for** $f\ n$
proof −
from t **have** $the\ (t\ (f \triangleright n)) = the\ (c\ (the\ (s\ (f \triangleright n))))$
by (*simp add*: *that*)
then have $\chi\ (the\ (t\ (f \triangleright n))) = \psi\ (the\ (s\ (f \triangleright n)))$
using $c(2)$ **by** *simp*
with *extra* **show** *?thesis* **using** *that* **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *learn-lim-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-lim* $\psi\ U\ s$
shows $\exists t.$ *learn-lim* $\chi\ U\ t$
using *assms learn-lim-extra-wrt-goedel*[**where** $?extra = \lambda U\ f\ n\ h.$ *True*]
by *simp*

lemma *LIM-wrt-phi-eq-Lim*: *LIM-wrt* φ = *LIM*
using *LIM-wrt-def Lim-def learn-lim-wrt-goedel*[*OF goedel-numbering-phi*]
by *blast*

2.2.2 BC: Behaviorally correct learning in the limit

Behaviorally correct learning in the limit relaxes LIM by requiring that the strategy almost always output an index for the target function, but not necessarily the same index. In other words convergence of $(S(f^n))_{n\in\mathbb{N}}$ is replaced by convergence of $(\psi_{S(f^n)})_{n\in\mathbb{N}}$.

definition *learn-bc* :: *partial2* ⇒ (*partial1 set*) ⇒ *partial1* ⇒ *bool* **where**
learn-bc $\psi\ U\ s \equiv$
environment $\psi\ U\ s \wedge$
$(\forall f \in U.\ \forall^\infty n.\ \psi\ (the\ (s\ (f \triangleright n))) = f)$

lemma *learn-bcE*:
assumes *learn-bc* $\psi\ U\ s$
shows *environment* $\psi\ U\ s$
and $\bigwedge f.\ f \in U \Longrightarrow \forall^\infty n.\ \psi\ (the\ (s\ (f \triangleright n))) = f$
using *assms learn-bc-def* **by** *auto*

lemma *learn-bcI*:
assumes *environment* $\psi\ U\ s$
and $\bigwedge f.\ f \in U \Longrightarrow \forall^\infty n.\ \psi\ (the\ (s\ (f \triangleright n))) = f$
shows *learn-bc* $\psi\ U\ s$
using *assms learn-bc-def* **by** *auto*

definition *BC-wrt* :: *partial2* ⇒ *partial1 set set* **where**
BC-wrt $\psi \equiv \{U.\ \exists s.$ *learn-bc* $\psi\ U\ s\}$

definition *BC* :: *partial1 set set* **where**
  $BC \equiv \{U.\ \exists\psi\ s.\ \textit{learn-bc}\ \psi\ U\ s\}$

BC is a superset of LIM and closed under the subset relation.

lemma *learn-lim-imp-BC*: $\textit{learn-lim}\ \psi\ U\ s \Longrightarrow \textit{learn-bc}\ \psi\ U\ s$
  using *learn-limE learn-bcI*[*of* ψ *U s*] **by** *fastforce*

lemma *Lim-subseteq-BC*: $LIM \subseteq BC$
  using *learn-lim-imp-BC Lim-def BC-def* **by** *blast*

lemma *learn-bc-closed-subseteq*:
  assumes *learn-bc* ψ *U s* **and** $V \subseteq U$
  shows *learn-bc* ψ *V s*
  using *assms learn-bc-def* **by** *auto*

corollary *BC-closed-subseteq*:
  assumes $U \in BC$ **and** $V \subseteq U$
  shows $V \in BC$
  using *assms* **by** (*smt BC-def learn-bc-closed-subseteq mem-Collect-eq*)

Just like with LIM, guessing a wrong hypothesis infinitely often precludes BC-style learning.

lemma *infinite-hyp-wrong-not-BC*:
  assumes $f \in U$ **and** $\forall n.\ \exists m{>}n.\ \psi\ (the\ (s\ (f \triangleright m))) \neq f$
  shows $\neg$ *learn-bc* ψ *U s*
proof
  assume *learn-bc* ψ *U s*
  then obtain n_0 **where** $\forall n{\geq}n_0.\ \psi\ (the\ (s\ (f \triangleright n))) = f$
    using *learn-bcE assms*(*1*) **by** *metis*
  with *assms*(*2*) **show** *False* **using** *less-imp-le* **by** *blast*
qed

The proof that Gödel numberings suffice as hypothesis spaces for BC is similar to the one for *learn-lim-extra-wrt-goedel*. We do not need the *extra* part for BC, but we get it for free.

lemma *learn-bc-extra-wrt-goedel*:
  fixes $extra :: (partial1\ set) \Rightarrow partial1 \Rightarrow nat \Rightarrow partial1 \Rightarrow bool$
  assumes *goedel-numbering* χ
    and *learn-bc* ψ *U s*
    and $\bigwedge f\ n.\ f \in U \Longrightarrow extra\ U\ f\ n\ (\psi\ (the\ (s\ (f \triangleright n))))$
  shows $\exists t.\ \textit{learn-bc}\ \chi\ U\ t \wedge (\forall f{\in}U.\ \forall n.\ extra\ U\ f\ n\ (\chi\ (the\ (t\ (f \triangleright n)))))$
proof –
  have *env*: *environment* ψ *U s*
    and *lim*: *learn-bc* ψ *U s*
    and *extra*: $\forall f{\in}U.\ \forall n.\ extra\ U\ f\ n\ (\psi\ (the\ (s\ (f \triangleright n))))$
    using *assms learn-bc-def* **by** *auto*
  obtain *c* **where** *c*: $c \in \mathcal{R}\ \forall i.\ \psi\ i = \chi\ (the\ (c\ i))$
    using *env goedel-numberingE*[*OF assms*(*1*), *of* ψ] **by** *auto*
  define *t* **where**
    $t = (\lambda x.\ if\ s\ x \downarrow \wedge\ c\ (the\ (s\ x)) \downarrow\ then\ Some\ (the\ (c\ (the\ (s\ x))))\ else\ None)$
  have $t \in \mathcal{P}$
    unfolding *t-def* **using** *env c concat-P1-P1*[*of c s*] **by** *auto*
  have $t\ x = (if\ s\ x \downarrow\ then\ Some\ (the\ (c\ (the\ (s\ x))))\ else\ None)$ **for** *x*
    using *t-def c*(*1*) *R1-imp-total1* **by** *auto*

```
  then have t: t (f ▷ n) ↓= the (c (the (s (f ▷ n)))) if f ∈ U for f n
    using lim learn-bcE(1) that by simp
  have learn-bc χ U t
  proof (rule learn-bcI)
    show environment χ U t
      using t by (simp add: ‹t ∈ 𝒫› env goedel-numbering-P2[OF assms(1)])
    show ∀∞n. χ (the (t (f ▷ n))) = f if f ∈ U for f
    proof −
      obtain n0 where ∀n≥n0. ψ (the (s (f ▷ n))) = f
        using lim learn-bcE(2) ‹f ∈ U› by blast
      then show ?thesis using that t c(2) by auto
    qed
  qed
  moreover have extra U f n (χ (the (t (f ▷ n)))) if f ∈ U for f n
  proof −
    from t have the (t (f ▷ n)) = the (c (the (s (f ▷ n))))
      by (simp add: that)
    then have χ (the (t (f ▷ n))) = ψ (the (s (f ▷ n)))
      using c(2) by simp
    with extra show ?thesis using that by simp
  qed
  ultimately show ?thesis by auto
qed
```

```
corollary learn-bc-wrt-goedel:
  assumes goedel-numbering χ and learn-bc ψ U s
  shows ∃t. learn-bc χ U t
  using assms learn-bc-extra-wrt-goedel[where ?extra=λ- - - -. True] by simp
```

```
corollary BC-wrt-phi-eq-BC: BC-wrt φ = BC
  using learn-bc-wrt-goedel goedel-numbering-phi BC-def BC-wrt-def by blast
```

2.2.3 CONS: Learning in the limit with consistent hypotheses

A hypothesis is *consistent* if it matches all values in the prefix given to the strategy. Consistent learning in the limit requires the strategy to output only consistent hypotheses for prefixes from the class.

```
definition learn-cons :: partial2 ⇒ (partial1 set) ⇒ partial1 ⇒ bool where
  learn-cons ψ U s ≡
    learn-lim ψ U s ∧
    (∀f∈U. ∀n. ∀k≤n. ψ (the (s (f ▷ n))) k = f k)
```

```
definition CONS-wrt :: partial2 ⇒ partial1 set set where
  CONS-wrt ψ ≡ {U. ∃s. learn-cons ψ U s}
```

```
definition CONS :: partial1 set set where
  CONS ≡ {U. ∃ψ s. learn-cons ψ U s}
```

```
lemma CONS-subseteq-Lim: CONS ⊆ LIM
  using CONS-def Lim-def learn-cons-def by blast
```

```
lemma learn-consI:
  assumes environment ψ U s
    and ⋀f. f ∈ U ⟹ ∃i. ψ i = f ∧ (∀∞n. s (f ▷ n) ↓= i)
    and ⋀f n. f ∈ U ⟹ ∀k≤n. ψ (the (s (f ▷ n))) k = f k
```

shows *learn-cons* ψ *U s*
using *assms learn-lim-def learn-cons-def* **by** *simp*

If a consistent strategy converges, it automatically converges to a correct hypothesis. Thus we can remove $\psi\ i = f$ from the second assumption in the previous lemma.

lemma *learn-consI2*:
assumes *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \forall^\infty n.\ s\ (f \triangleright n) \downarrow= i$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \forall k \le n.\ \psi\ (the\ (s\ (f \triangleright n)))\ k = f\ k$
shows *learn-cons* ψ *U s*
proof (*rule learn-consI*)
show *environment* ψ *U s*
and *cons*: $\bigwedge f\ n.\ f \in U \Longrightarrow \forall k \le n.\ \psi\ (the\ (s\ (f \triangleright n)))\ k = f\ k$
using *assms* **by** *simp-all*
show $\exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i)$ **if** $f \in U$ **for** f
proof −
from *that assms*(*2*) **obtain** $i\ n_0$ **where** *i-n0*: $\forall n \ge n_0.\ s\ (f \triangleright n) \downarrow= i$
by *blast*
have $\psi\ i\ x = f\ x$ **for** x
proof (*cases* $x \le n_0$)
case *True*
then show *?thesis*
using *i-n0 cons that* **by** *fastforce*
next
case *False*
moreover have $\forall k \le x.\ \psi\ (the\ (s\ (f \triangleright x)))\ k = f\ k$
using *cons that* **by** *simp*
ultimately show *?thesis* **using** *i-n0* **by** *simp*
qed
with *i-n0* **show** *?thesis* **by** *auto*
qed
qed

lemma *learn-consE*:
assumes *learn-cons* ψ *U s*
shows *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i\ n_0.\ \psi\ i = f \wedge (\forall n \ge n_0.\ s\ (f \triangleright n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \forall k \le n.\ \psi\ (the\ (s\ (f \triangleright n)))\ k = f\ k$
using *assms learn-cons-def learn-lim-def* **by** *auto*

lemma *learn-cons-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-cons* ψ *U s*
shows $\exists t.$ *learn-cons* χ *U t*
using *learn-cons-def assms*
learn-lim-extra-wrt-goedel[**where** $?extra = \lambda U\ f\ n\ h.\ \forall k \le n.\ h\ k = f\ k$]
by *auto*

lemma *CONS-wrt-phi-eq-CONS*: *CONS-wrt* φ = *CONS*
using *CONS-wrt-def CONS-def learn-cons-wrt-goedel goedel-numbering-phi*
by *blast*

lemma *learn-cons-closed-subseteq*:
assumes *learn-cons* ψ *U s* **and** $V \subseteq U$
shows *learn-cons* ψ *V s*
using *assms learn-cons-def learn-lim-closed-subseteq* **by** *auto*

lemma *CONS-closed-subseteq*:
assumes $U \in CONS$ **and** $V \subseteq U$
shows $V \in CONS$
using *assms learn-cons-closed-subseteq* **by** (*smt CONS-def mem-Collect-eq*)

A consistent strategy cannot output the same hypothesis for two different prefixes from the class to be learned.

lemma *same-hyp-different-init-not-cons*:
assumes $f \in U$
and $g \in U$
and $f \triangleright n \neq g \triangleright n$
and $s\ (f \triangleright n) = s\ (g \triangleright n)$
shows $\neg$ *learn-cons* φ *U s*
unfolding *learn-cons-def* **by** (*auto*, *metis assms init-eqI*)

2.2.4 TOTAL: Learning in the limit with total hypotheses

Total learning in the limit requires the strategy to hypothesize only total functions for prefixes from the class.

definition *learn-total* :: *partial2* ⇒ (*partial1 set*) ⇒ *partial1* ⇒ *bool* **where**
learn-total ψ *U s* ≡
learn-lim ψ *U s* ∧
$(\forall f{\in}U.\ \forall n.\ \psi\ (the\ (s\ (f \triangleright n))) \in \mathcal{R})$

definition *TOTAL-wrt* :: *partial2* ⇒ *partial1 set set* **where**
TOTAL-wrt $\psi \equiv \{U.\ \exists s.$ *learn-total* ψ *U s*$\}$

definition *TOTAL* :: *partial1 set set* **where**
$TOTAL \equiv \{U.\ \exists \psi\ s.$ *learn-total* ψ *U s*$\}$

lemma *TOTAL-subseteq-LIM*: $TOTAL \subseteq LIM$
unfolding *TOTAL-def Lim-def* **using** *learn-total-def* **by** *auto*

lemma *learn-totalI*:
assumes *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \psi\ (the\ (s\ (f \triangleright n))) \in \mathcal{R}$
shows *learn-total* ψ *U s*
using *assms learn-lim-def learn-total-def* **by** *auto*

lemma *learn-totalE*:
assumes *learn-total* ψ *U s*
shows *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i\ n_0.\ \psi\ i = f \wedge (\forall n{\ge}n_0.\ s\ (f \triangleright n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \psi\ (the\ (s\ (f \triangleright n))) \in \mathcal{R}$
using *assms learn-lim-def learn-total-def* **by** *auto*

lemma *learn-total-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-total* ψ *U s*
shows $\exists t.$ *learn-total* χ *U t*
using *learn-total-def assms learn-lim-extra-wrt-goedel*[**where** $?extra{=}\lambda U\ f\ n\ h.\ h \in \mathcal{R}$]
by *auto*

lemma *TOTAL-wrt-phi-eq-TOTAL*: *TOTAL-wrt* φ = *TOTAL*
using *TOTAL-wrt-def TOTAL-def learn-total-wrt-goedel goedel-numbering-phi*

by *blast*

lemma *learn-total-closed-subseteq*:
assumes *learn-total* ψ *U s* **and** $V \subseteq U$
shows *learn-total* ψ *V s*
using *assms learn-total-def learn-lim-closed-subseteq* **by** *auto*

lemma *TOTAL-closed-subseteq*:
assumes $U \in TOTAL$ **and** $V \subseteq U$
shows $V \in TOTAL$
using *assms learn-total-closed-subseteq* **by** (*smt TOTAL-def mem-Collect-eq*)

2.2.5 CP: Learning in the limit with class-preserving hypotheses

Class-preserving learning in the limit requires all hypotheses for prefixes from the class to be functions from the class.

definition *learn-cp* :: *partial2* $\Rightarrow$ (*partial1 set*) $\Rightarrow$ *partial1* $\Rightarrow$ *bool* **where**
learn-cp ψ *U s* $\equiv$
learn-lim ψ *U s* $\wedge$
$(\forall f \in U.\ \forall n.\ \psi\ (the\ (s\ (f \triangleright n))) \in U)$

definition *CP-wrt* :: *partial2* $\Rightarrow$ *partial1 set set* **where**
CP-wrt $\psi \equiv \{U.\ \exists s.\ learn\text{-}cp\ \psi\ U\ s\}$

definition *CP* :: *partial1 set set* **where**
$CP \equiv \{U.\ \exists \psi\ s.\ learn\text{-}cp\ \psi\ U\ s\}$

lemma *learn-cp-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-cp* ψ *U s*
shows $\exists t.$ *learn-cp* χ *U t*
using *learn-cp-def assms learn-lim-extra-wrt-goedel*[**where** *?extra*=$\lambda U\ f\ n\ h.\ h \in U$]
by *auto*

corollary *CP-wrt-phi*: $CP = CP\text{-}wrt\ \varphi$
using *learn-cp-wrt-goedel*[*OF goedel-numbering-phi*]
by (*smt CP-def CP-wrt-def Collect-cong*)

lemma *learn-cpI*:
assumes *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \psi\ (the\ (s\ (f \triangleright n))) \in U$
shows *learn-cp* ψ *U s*
using *assms learn-cp-def learn-lim-def* **by** *auto*

lemma *learn-cpE*:
assumes *learn-cp* ψ *U s*
shows *environment* ψ *U s*
and $\bigwedge f.\ f \in U \Longrightarrow \exists i\ n_0.\ \psi\ i = f \wedge (\forall n \geq n_0.\ s\ (f \triangleright n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \psi\ (the\ (s\ (f \triangleright n))) \in U$
using *assms learn-lim-def learn-cp-def* **by** *auto*

Since classes contain only total functions, a CP strategy is also a TOTAL strategy.

lemma *learn-cp-imp-total*: *learn-cp* ψ *U s* $\Longrightarrow$ *learn-total* ψ *U s*
using *learn-cp-def learn-total-def learn-lim-def* **by** *auto*

lemma *CP-subseteq-TOTAL*: $CP \subseteq TOTAL$
  using *learn-cp-imp-total CP-def TOTAL-def* **by** *blast*

2.2.6 FIN: Finite learning

In general it is undecidable whether a LIM strategy has reached its final hypothesis. By contrast, in finite learning (also called "one-shot learning") the strategy signals when it is ready to output a hypothesis. Up until then it outputs a "don't know yet" value. This value is represented by zero and the actual hypothesis i by $i + 1$.

definition *learn-fin* :: *partial2* ⇒ *partial1 set* ⇒ *partial1* ⇒ *bool* **where**
  learn-fin ψ *U s* ≡
    environment ψ *U s* ∧
    $(\forall f \in U.\ \exists i\ n_0.\ \psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \rhd n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \rhd n) \downarrow= Suc\ i))$

definition *FIN-wrt* :: *partial2* ⇒ *partial1 set set* **where**
  FIN-wrt $\psi \equiv \{U.\ \exists s.$ *learn-fin* ψ *U s*$\}$

definition *FIN* :: *partial1 set set* **where**
  FIN $\equiv \{U.\ \exists \psi\ s.$ *learn-fin* ψ *U s*$\}$

lemma *learn-finI*:
  assumes *environment* ψ *U s*
    and $\bigwedge f.\ f \in U \Longrightarrow$
      $\exists i\ n_0.\ \psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \rhd n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \rhd n) \downarrow= Suc\ i)$
  shows *learn-fin* ψ *U s*
  using *assms learn-fin-def* **by** *auto*

lemma *learn-finE*:
  assumes *learn-fin* ψ *U s*
  shows *environment* ψ *U s*
    and $\bigwedge f.\ f \in U \Longrightarrow$
      $\exists i\ n_0.\ \psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \rhd n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \rhd n) \downarrow= Suc\ i)$
  using *assms learn-fin-def* **by** *auto*

lemma *learn-fin-closed-subseteq*:
  assumes *learn-fin* ψ *U s* **and** $V \subseteq U$
  shows *learn-fin* ψ *V s*
  using *assms learn-fin-def* **by** *auto*

lemma *learn-fin-wrt-goedel*:
  assumes *goedel-numbering* χ **and** *learn-fin* ψ *U s*
  shows $\exists t.$ *learn-fin* χ *U t*
proof −
  have *env*: *environment* ψ *U s*
    and *fin*: $\bigwedge f.\ f \in U \Longrightarrow$
      $\exists i\ n_0.\ \psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \rhd n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \rhd n) \downarrow= Suc\ i)$
    using *assms(2) learn-finE* **by** *auto*
  obtain *c* **where** *c*: $c \in \mathcal{R}\ \forall i.\ \psi\ i = \chi\ (the\ (c\ i))$
    using *env goedel-numberingE*[*OF assms*(*1*), *of* ψ] **by** *auto*
  define *t* **where** $t \equiv$
    $\lambda x.$ *if s x* ↑ *then None*
        else if s x = Some 0 then Some 0
          else Some (*Suc* (*the* (*c* (*the* (*s x*) − *1*))))
  have $t \in \mathcal{P}$
  proof −

```
    from c obtain rc where rc:
      recfn 1 rc
      total rc
      ∀ x. c x = eval rc [x]
      by auto
    from env obtain rs where rs: recfn 1 rs ∀ x. s x = eval rs [x]
      by auto
    then have eval rs [f ▷ n] ↓ if f ∈ U for f n
      using env that by simp
    define rt where rt = Cn 1 r-ifz [rs, Z, Cn 1 S [Cn 1 rc [Cn 1 r-dec [rs]]]]
    then have recfn 1 rt
      using rc(1) rs(1) by simp
    have eval rt [x] ↑ if eval rs [x] ↑ for x
      using rc(1) rs(1) rt-def that by auto
    moreover have eval rt [x] ↓= 0 if eval rs [x] ↓= 0 for x
      using rt-def that rc(1,2) rs(1) by simp
    moreover have eval rt [x] ↓= Suc (the (c (the (s x) − 1))) if eval rs [x] ↓≠ 0 for x
      using rt-def that rc rs by auto
    ultimately have eval rt [x] = t x for x
      by (simp add: rs(2) t-def)
    with ‹recfn 1 rt› show ?thesis by auto
  qed
  have t: t (f ▷ n) ↓=
      (if s (f ▷ n) = Some 0 then 0 else Suc (the (c (the (s (f ▷ n)) − 1))))
    if f ∈ U for f n
    using that env by (simp add: t-def)
  have learn-fin χ U t
  proof (rule learn-finI)
    show environment χ U t
      using t by (simp add: ‹t ∈ 𝒫› env goedel-numbering-P2[OF assms(1)])
    show ∃ i n0. χ i = f ∧ (∀ n<n0. t (f ▷ n) ↓= 0) ∧ (∀ n≥n0. t (f ▷ n) ↓= Suc i)
      if f ∈ U for f
    proof −
      from fin obtain i n0 where
        i: ψ i = f ∧ (∀ n<n0. s (f ▷ n) ↓= 0) ∧ (∀ n≥n0. s (f ▷ n) ↓= Suc i)
        using ‹f ∈ U› by blast
      let ?j = the (c i)
      have χ ?j = f
        using c(2) i by simp
      moreover have ∀ n<n0. t (f ▷ n) ↓= 0
        using t[OF that] i by simp
      moreover have t (f ▷ n) ↓= Suc ?j if n ≥ n0 for n
        using that i t[OF ‹f ∈ U›] by simp
      ultimately show ?thesis by auto
    qed
  qed
  then show ?thesis by auto
qed

end
```

2.3 FIN is a proper subset of CP

```
theory CP-FIN-NUM
  imports Inductive-Inference-Basics
```

begin

Let S be a FIN strategy for a non-empty class U. Let T be a strategy that hypothesizes an arbitrary function from U while S outputs “don’t know” and the hypothesis of S otherwise. Then T is a CP strategy for U.

```
lemma nonempty-FIN-wrt-impl-CP:
  assumes U ≠ {} and U ∈ FIN-wrt ψ
  shows U ∈ CP-wrt ψ
proof −
  obtain s where learn-fin ψ U s
    using assms(2) FIN-wrt-def by auto
  then have env: environment ψ U s and
    fin: ⋀f. f ∈ U ⟹
      ∃ i n_0. ψ i = f ∧ (∀ n<n_0. s (f ⊳ n) ↓= 0) ∧ (∀ n≥n_0. s (f ⊳ n) ↓= Suc i)
    using learn-finE by auto
  from assms(1) obtain f_0 where f_0 ∈ U
    by auto
  with fin obtain i_0 where ψ i_0 = f_0
    by blast
  define t where t x ≡
    (if s x ↑ then None else if s x ↓= 0 then Some i_0 else Some (the (s x) − 1))
    for x
  have t ∈ 𝒫
  proof −
    from env obtain rs where rs: recfn 1 rs ⋀x. eval rs [x] = s x
      by auto
    define rt where rt = Cn 1 r-ifz [rs, r-const i_0, Cn 1 r-dec [rs]]
    then have recfn 1 rt
      using rs(1) by simp
    then have eval rt [x] ↓= (if s x ↓= 0 then i_0 else (the (s x)) − 1) if s x ↓ for x
      using rs rt-def that by auto
    moreover have eval rt [x] ↑ if eval rs [x] ↑ for x
      using rs rt-def that by simp
    ultimately have eval rt [x] = t x for x
      using rs(2) t-def by simp
    with ‹recfn 1 rt› show ?thesis by auto
  qed
  have learn-cp ψ U t
  proof (rule learn-cpI)
    show environment ψ U t
      using env t-def ‹t ∈ 𝒫› by simp
    show ∃ i. ψ i = f ∧ (∀^∞n. t (f ⊳ n) ↓= i) if f ∈ U for f
    proof −
      from that fin obtain i n_0 where
        i: ψ i = f ∀ n<n_0. s (f ⊳ n) ↓= 0 ∀ n≥n_0. s (f ⊳ n) ↓= Suc i
        by blast
      moreover have ∀ n≥n_0. t (f ⊳ n) ↓= i
        using that t-def i(3) by simp
      ultimately show ?thesis by auto
    qed
    show ψ (the (t (f ⊳ n))) ∈ U if f ∈ U for f n
      using ‹ψ i_0 = f_0› ‹f_0 ∈ U› t-def fin env that
      by (metis (no-types, lifting) diff-Suc-1 not-less option.sel)
  qed
  then show ?thesis using CP-wrt-def env by auto
qed
```

lemma *FIN-wrt-impl-CP*:
  assumes $U \in$ *FIN-wrt* ψ
  shows $U \in$ *CP-wrt* ψ
proof (*cases* $U = \{\}$)
  case *True*
  then have $\psi \in \mathcal{P}^2 \Longrightarrow U \in$ *CP-wrt* ψ
    using *CP-wrt-def learn-cpI*[*of* ψ {} $\lambda x.$ *Some 0*] *const-in-Prim1* **by** *auto*
  moreover have $\psi \in \mathcal{P}^2$
    using *assms FIN-wrt-def learn-finE* **by** *auto*
  ultimately show $U \in$ *CP-wrt* ψ **by** *simp*
next
  case *False*
  with *nonempty-FIN-wrt-impl-CP assms* **show** *?thesis*
    by *simp*
qed

corollary *FIN-subseteq-CP*: *FIN* $\subseteq$ *CP*
proof
  fix *U*
  assume $U \in$ *FIN*
  then have $\exists \psi.\ U \in$ *FIN-wrt* ψ
    using *FIN-def FIN-wrt-def* **by** *auto*
  then have $\exists \psi.\ U \in$ *CP-wrt* ψ
    using *FIN-wrt-impl-CP* **by** *auto*
  then show $U \in$ *CP*
    by (*simp add*: *CP-def CP-wrt-def*)
qed

In order to show the *proper* inclusion, we show $U_0 \in CP - FIN$. A CP strategy for U_0 simply hypothesizes the function in U_0 with the longest prefix of f^n not ending in zero. For that we define a function computing the index of the rightmost non-zero value in a list, returning the length of the list if there is no such value.

definition *findr* :: *partial1* **where**
  findr e $\equiv$
    if $\exists$ *i*<*e-length e*. *e-nth e i* $\neq$ *0*
    then Some (*GREATEST i*. *i* < *e-length e* $\wedge$ *e-nth e i* $\neq$ *0*)
    else Some (*e-length e*)

lemma *findr-total*: *findr e* $\downarrow$
  unfolding *findr-def* **by** *simp*

lemma *findr-ex*:
  assumes $\exists$ *i*<*e-length e*. *e-nth e i* $\neq$ *0*
  shows *the* (*findr e*) < *e-length e*
    and *e-nth e* (*the* (*findr e*)) $\neq$ *0*
    and $\forall$ *i*. *the* (*findr e*) < *i* $\wedge$ *i* < *e-length e* $\longrightarrow$ *e-nth e i* = *0*
proof −
  let *?P* = λ*i*. *i* < *e-length e* $\wedge$ *e-nth e i* $\neq$ *0*
  from *assms* **have** $\exists$ *i*. *?P i* **by** *simp*
  then have *?P* (*Greatest ?P*)
    using *GreatestI-ex-nat*[*of ?P e-length e*] **by** *fastforce*
  moreover have $*$: *findr e* = *Some* (*Greatest ?P*)
    using *assms findr-def* **by** *simp*
  ultimately show *the* (*findr e*) < *e-length e* **and** *e-nth e* (*the* (*findr e*)) $\neq$ *0*

```
    by fastforce+
  show ∀ i. the (findr e) < i ∧ i < e-length e ⟶ e-nth e i = 0
    using ∗ Greatest-le-nat[of ?P - e-length e] by fastforce
qed

definition r-findr ≡
  let g =
    Cn 3 r-ifz
     [Cn 3 r-nth [Id 3 2, Id 3 0],
      Cn 3 r-ifeq [Id 3 0, Id 3 1, Cn 3 S [Id 3 0], Id 3 1],
      Id 3 0]
  in Cn 1 (Pr 1 Z g) [Cn 1 r-length [Id 1 0], Id 1 0]

lemma r-findr-prim [simp]: prim-recfn 1 r-findr
  unfolding r-findr-def by simp

lemma r-findr [simp]: eval r-findr [e] = findr e
proof −
  define g where g =
    Cn 3 r-ifz
     [Cn 3 r-nth [Id 3 2, Id 3 0],
      Cn 3 r-ifeq [Id 3 0, Id 3 1, Cn 3 S [Id 3 0], Id 3 1],
      Id 3 0]
  then have recfn 3 g
    by simp
  with g-def have g: eval g [j, r, e] ↓=
      (if e-nth e j ≠ 0 then j else if j = r then Suc j else r) for j r e
    by simp
  let ?h = Pr 1 Z g
  have recfn 2 ?h
    by (simp add: ‹recfn 3 g›)
  let ?P = λe j i. i < j ∧ e-nth e i ≠ 0
  let ?G = λe j. Greatest (?P e j)
  have h: eval ?h [j, e] =
    (if ∀ i<j. e-nth e i = 0 then Some j else Some (?G e j)) for j e
  proof (induction j)
    case 0
    then show ?case using ‹recfn 2 ?h› by auto
  next
    case (Suc j)
    then have eval ?h [Suc j, e] = eval g [j, the (eval ?h [j, e]), e]
      using ‹recfn 2 ?h› by auto
    then have eval ?h [Suc j, e] =
        eval g [j, if ∀ i<j. e-nth e i = 0 then j else ?G e j, e]
      using Suc by auto
    then have ∗: eval ?h [Suc j, e] ↓=
      (if e-nth e j ≠ 0 then j
       else if j = (if ∀ i<j. e-nth e i = 0 then j else ?G e j)
            then Suc j
            else (if ∀ i<j. e-nth e i = 0 then j else ?G e j))
      using g by simp
    show ?case
    proof (cases ∀ i<Suc j. e-nth e i = 0)
      case True
      then show ?thesis using ∗ by simp
    next
```

```
      case False
      then have ex: ∃ i<Suc j. e-nth e i ≠ 0
        by auto
      show ?thesis
      proof (cases e-nth e j = 0)
        case True
        then have ex′: ∃ i<j. e-nth e i ≠ 0
          using ex less-Suc-eq by fastforce
        then have (if ∀ i<j. e-nth e i = 0 then j else ?G e j) = ?G e j
          by metis
        moreover have ?G e j < j
          using ex′ GreatestI-nat[of ?P e j] less-imp-le-nat by blast
        ultimately have eval ?h [Suc j, e] ↓= ?G e j
          using ∗ True by simp
        moreover have ?G e j = ?G e (Suc j)
          using True by (metis less-SucI less-Suc-eq)
        ultimately show ?thesis using ex by metis
      next
        case False
        then have eval ?h [Suc j, e] ↓= j
          using ∗ by simp
        moreover have ?G e (Suc j) = j
          using ex False Greatest-equality[of ?P e (Suc j)] by simp
        ultimately show ?thesis using ex by simp
      qed
    qed
  qed
  let ?hh = Cn 1 ?h [Cn 1 r-length [Id 1 0], Id 1 0]
  have recfn 1 ?hh
    using ‹recfn 2 ?h› by simp
  with h have hh: eval ?hh [e] ↓=
      (if ∀ i<e-length e. e-nth e i = 0 then e-length e else ?G e (e-length e)) for e
    by auto
  then have eval ?hh [e] = findr e for e
    unfolding findr-def by auto
  moreover have total ?hh
    using hh totalI1 ‹recfn 1 ?hh› by simp
  ultimately show ?thesis
    using ‹recfn 1 ?hh› g-def r-findr-def findr-def by metis
qed

lemma U0-in-CP: U_0 ∈ CP
proof −
  define s where
    s ≡ λx. if findr x ↓= e-length x then Some 0 else Some (e-take (Suc (the (findr x))) x)
  have s ∈ 𝒫
  proof −
    define r where
      r ≡ Cn 1 r-ifeq [r-findr, r-length, Z, Cn 1 r-take [Cn 1 S [r-findr], Id 1 0]]
    then have ⋀x. eval r [x] = s x
      using s-def findr-total by fastforce
    moreover have recfn 1 r
      using r-def by simp
    ultimately show ?thesis by auto
  qed
  moreover have learn-cp prenum U_0 s
```

```
proof (rule learn-cpI)
  show environment prenum U0 s
    using ‹s ∈ 𝒫› s-def prenum-in-R2 U0-in-NUM by auto
  show ∃ i. prenum i = f ∧ (∀∞n. s (f ▹ n) ↓= i) if f ∈ U0 for f
  proof (cases f = (λ-. Some 0))
    case True
    then have s (f ▹ n) ↓= 0 for n
      using findr-def s-def by simp
    then have ∀ n≥0. s (f ▹ n) ↓= 0 by simp
    moreover have prenum 0 = f
      using True by auto
    ultimately show ?thesis by auto
  next
    case False
    then obtain ws where ws: length ws > 0 last ws ≠ 0 f = ws ⊙ 0∞
      using U0-def ‹f ∈ U0› almost0-canonical by blast
    let ?m = length ws − 1
    let ?i = list-encode ws
    have prenum ?i = f
      using ws by auto
    moreover have s (f ▹ n) ↓= ?i if n ≥ ?m for n
    proof −
      have e-nth (f ▹ n) ?m ≠ 0
        using ws that by (simp add: last-conv-nth)
      then have ∃ k<Suc n. e-nth (f ▹ n) k ≠ 0
        using le-imp-less-Suc that by blast
      moreover have
        (GREATEST k. k < e-length (f ▹ n) ∧ e-nth (f ▹ n) k ≠ 0) = ?m
      proof (rule Greatest-equality)
        show ?m < e-length (f ▹ n) ∧ e-nth (f ▹ n) ?m ≠ 0
          using ‹e-nth (f ▹ n) ?m ≠ 0› that by auto
        show ⋀y. y < e-length (f ▹ n) ∧ e-nth (f ▹ n) y ≠ 0 ⟹ y ≤ ?m
          using ws less-Suc-eq-le by fastforce
      qed
      ultimately have findr (f ▹ n) ↓= ?m
        using that findr-def by simp
      moreover have ?m < e-length (f ▹ n)
        using that by simp
      ultimately have s (f ▹ n) ↓= e-take (Suc ?m) (f ▹ n)
        using s-def by simp
      moreover have e-take (Suc ?m) (f ▹ n) = list-encode ws
      proof −
        have take (Suc ?m) (prefix f n) = prefix f ?m
          using take-prefix[of f ?m n] ws that by (simp add: almost0-in-R1)
        then have take (Suc ?m) (prefix f n) = ws
          using ws prefixI by auto
        then show ?thesis by simp
      qed
      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis by auto
  qed
  show ⋀f n. f ∈ U0 ⟹ prenum (the (s (f ▹ n))) ∈ U0
    using U0-def by fastforce
qed
ultimately show ?thesis using CP-def by blast
```

qed

As a bit of an interlude, we can now show that CP is not closed under the subset relation. This works by removing functions from U_0 in a "noncomputable" way such that a strategy cannot ensure that every intermediate hypothesis is in that new class.

lemma *CP-not-closed-subseteq*: $\exists$ *V U*. $V \subseteq U \wedge U \in CP \wedge V \notin CP$
proof –
— The numbering $g \in \mathcal{R}^2$ enumerates all functions $i0^\infty \in U_0$.
define *g* **where** $g \equiv \lambda i.\ [i] \odot\ 0^\infty$
have *g-inj*: $i = j$ **if** $g\ i = g\ j$ **for** *i j*
proof –
have $g\ i\ 0 \downarrow= i$ **and** $g\ j\ 0 \downarrow= j$
by (*simp-all add*: *g-def*)
with *that* **show** $i = j$
by (*metis option.inject*)
qed

— Define a class V. If the strategy φ_i learns g_i, it outputs a hypothesis for g_i on some shortest prefix g_i^m. Then the function $g_i^m 10^\infty$ is included in the class V; otherwise g_i is included.
define *V* **where** $V \equiv$
{*if learn-lim* φ {*g i*} (φ *i*)
then (*prefix* (*g i*) (*LEAST n*. φ (*the* (φ *i* ((*g i*) $\triangleright$ *n*))) = *g i*)) @ [*1*] $\odot$ 0^∞
else g i |
i. $i \in$ *UNIV*}
have $V \notin$ *CP-wrt* φ
proof
— Assuming $V \in CP_\varphi$, there is a CP strategy φ_i for V.
assume $V \in$ *CP-wrt* φ
then obtain *s* **where** *s*: $s \in \mathcal{P}$ *learn-cp* φ *V s*
using *CP-wrt-def learn-cpE*(*1*) **by** *auto*
then obtain *i* **where** *i*: $\varphi\ i = s$
using *phi-universal* **by** *auto*

show *False*
proof (*cases learn-lim* φ {*g i*} (φ *i*))
case *learn*: *True*
— If φ_i learns g_i, it hypothesizes g_i on some shortest prefix g_i^m. Thus it hypothesizes g_i on some prefix of $g_i^m 10^\infty \in V$, too. But g_i is not a class-preserving hypothesis because $g_i \notin V$.
let *?P* = λn. φ (*the* (φ *i* ((*g i*) $\triangleright$ *n*))) = *g i*
let *?m* = *Least ?P*
have $\exists$ *n*. *?P n*
using *i s* **by** (*meson learn infinite-hyp-wrong-not-Lim insertI1 lessI*)
then have *?P ?m*
using *LeastI-ex*[*of ?P*] **by** *simp*
define *h* **where** *h* = (*prefix* (*g i*) *?m*) @ [*1*] $\odot$ 0^∞
then have $h \in V$
using *V-def learn* **by** *auto*
have (*g i*) $\triangleright$ *?m* = *h* $\triangleright$ *?m*
proof –
have *prefix* (*g i*) *?m* = *prefix h ?m*
unfolding *h-def* **by** (*simp add*: *prefix-prepend-less*)
then show *?thesis* **by** *auto*
qed
then have φ (*the* (φ *i* (*h* $\triangleright$ *?m*))) = *g i*
using ‹*?P ?m*› **by** *simp*
moreover have *g i* $\notin$ *V*

```
    proof
      assume g i ∈ V
      then obtain j where j: g i =
        (if learn-lim φ {g j} (φ j)
         then (prefix (g j) (LEAST n. φ (the (φ j ((g j) ▷ n))) = g j)) @ [1] ⊙ 0^∞
         else g j)
        using V-def by auto
      show False
      proof (cases learn-lim φ {g j} (φ j))
        case True
        then have g i =
            (prefix (g j) (LEAST n. φ (the (φ j ((g j) ▷ n))) = g j)) @ [1] ⊙ 0^∞
            (is g i = ?vs @ [1] ⊙ 0^∞)
          using j by simp
        moreover have len: length ?vs > 0 by simp
        ultimately have g i (length ?vs) ↓= 1
          by (simp add: prepend-associative)
        moreover have g i (length ?vs) ↓= 0
          using g-def len by simp
        ultimately show ?thesis by simp
      next
        case False
        then show ?thesis
          using j g-inj learn by auto
      qed
    qed
    ultimately have φ (the (φ i (h ▷ ?m))) ∉ V by simp
    then have ¬ learn-cp φ V (φ i)
      using ‹h ∈ V› learn-cpE(3) by auto
    then show ?thesis by (simp add: i s(2))
  next
    — If φ_i does not learn g_i, then g_i ∈ V. Hence φ_i does not learn V.
    case False
    then have g i ∈ V
      using V-def by auto
    with False have ¬ learn-lim φ V (φ i)
      using learn-lim-closed-subseteq by auto
    then show ?thesis
      using s(2) i by (simp add: learn-cp-def)
  qed
qed
then have V ∉ CP
  using CP-wrt-phi by simp
moreover have V ⊆ U_0
  using V-def g-def U0-def by auto
ultimately show ?thesis using U0-in-CP by auto
qed
```

Continuing with the main result of this section, we show that U_0 cannot be learned finitely. Any FIN strategy would have to output a hypothesis for the constant zero function on some prefix. But U_0 contains infinitely many other functions starting with the same prefix, which the strategy then would not learn finitely.

```
lemma U0-not-in-FIN: U_0 ∉ FIN
proof
  assume U_0 ∈ FIN
```

then obtain ψ *s* **where** *learn-fin* ψ U_0 *s*
using *FIN-def* **by** *blast*
with *learn-finE* **have** *cp*: $\bigwedge f.\ f \in U_0 \Longrightarrow$
$\exists\, i\ n_0.\ \psi\ i = f \wedge (\forall\, n{<}n_0.\ s\ (f \triangleright n) \downarrow= 0) \wedge (\forall\, n{\geq}n_0.\ s\ (f \triangleright n) \downarrow= Suc\ i)$
by *simp-all*

define *z* **where** $z = [] \odot 0^\infty$
then have $z \in U_0$
using *U0-def* **by** *auto*
with *cp* **obtain** $i\ n_0$ **where** *i*: $\psi\ i = z$ **and** *n0*: $\forall\, n{\geq}n_0.\ s\ (z \triangleright n) \downarrow= Suc\ i$
by *blast*

define *w* **where** $w = replicate\ (Suc\ n_0)\ 0\ @\ [1] \odot 0^\infty$
then have $prefix\ w\ n_0 = replicate\ (Suc\ n_0)\ 0$
by (*simp add*: *prefix-prepend-less*)
moreover have $prefix\ z\ n_0 = replicate\ (Suc\ n_0)\ 0$
using *prefixI*[*of replicate* $(Suc\ n_0)$ *0 z*] *less-Suc-eq-0-disj* **unfolding** *z-def*
by *fastforce*
ultimately have $z \triangleright\ n_0 = w \triangleright n_0$
by (*simp add*: *init-prefixE*)
with *n0* **have** $*$: $s\ (w \triangleright n_0) \downarrow= Suc\ i$ **by** *auto*

have $w \in U_0$ **using** *w-def U0-def* **by** *auto*
with *cp* **obtain** $i'\ n_0'$ **where** *i'*: $\psi\ i' = w$
and *n0'*: $\forall\, n{<}n_0'.\ s\ (w \triangleright n) \downarrow= 0\ \forall\, n{\geq}n_0'.\ s\ (w \triangleright n) \downarrow= Suc\ i'$
by *blast*

have $i \neq i'$
proof
assume $i = i'$
then have $w = z$
using *i i'* **by** *simp*
have $w\ (Suc\ n_0) \downarrow= 1$
using *w-def prepend*[*of replicate* $(Suc\ n_0)$ $0\ @\ [1]\ 0^\infty\ Suc\ n_0$]
by (*metis length-append-singleton length-replicate lessI nth-append-length*)
moreover have $z\ (Suc\ n_0) \downarrow= 0$
using *z-def* **by** *simp*
ultimately show *False*
using ‹$w = z$› **by** *simp*
qed
then have $s\ (w \triangleright n_0) \downarrow\neq Suc\ i$
using *n0'* **by** (*cases* $n_0 < n_0'$) *simp-all*
with $*$ **show** *False* **by** *simp*
qed

theorem *FIN-subset-CP*: $FIN \subset CP$
using *U0-in-CP U0-not-in-FIN FIN-subseteq-CP* **by** *auto*

2.4 NUM and FIN are incomparable

The class V_0 of all total recursive functions f where $f(0)$ is a Gödel number of f can be learned finitely by always hypothesizing $f(0)$. The class is not in NUM and therefore serves to separate NUM and FIN.

definition *V0* :: *partial1 set* (‹V_0›) **where**

```
  V₀ = {f. f ∈ ℛ ∧ φ (the (f 0)) = f}

lemma V0-altdef: V₀ = {[i] ⊙ f| i f. f ∈ ℛ ∧ φ i = [i] ⊙ f}
  (is V₀ = ?W)
proof
  show V₀ ⊆ ?W
  proof
    fix f
    assume f ∈ V₀
    then have f ∈ ℛ
      unfolding V0-def by simp
    then obtain i where i: f 0 ↓= i by fastforce
    define g where g = (λx. f (x + 1))
    then have g ∈ ℛ
      using skip-R1[OF ‹f ∈ ℛ›] by blast
    moreover have [i] ⊙ g = f
      using g-def i by auto
    moreover have φ i = f
      using ‹f ∈ V₀› V0-def i by force
    ultimately show f ∈ ?W by auto
  qed
  show ?W ⊆ V₀
  proof
    fix g
    assume g ∈ ?W
    then have φ (the (g 0)) = g by auto
    moreover have g ∈ ℛ
      using prepend-in-R1 ‹g ∈ ?W› by auto
    ultimately show g ∈ V₀
      by (simp add: V0-def)
  qed
qed

lemma V0-in-FIN: V₀ ∈ FIN
proof -
  define s where s = (λx. Some (Suc (e-hd x)))
  have s ∈ 𝒫
  proof -
    define r where r = Cn 1 S [r-hd]
    then have recfn 1 r by simp
    moreover have eval r [x] ↓= Suc (e-hd x) for x
      unfolding r-def by simp
    ultimately show ?thesis
      using s-def by blast
  qed
  have s: s (f ▹ n) ↓= Suc (the (f 0)) for f n
    unfolding s-def by simp
  have learn-fin φ V₀ s
  proof (rule learn-finI)
    show environment φ V₀ s
      using s-def ‹s ∈ 𝒫› phi-in-P2 V0-def by auto
    show ∃ i n₀. φ i = f ∧ (∀ n<n₀. s (f ▹ n) ↓= 0) ∧ (∀ n≥n₀. s (f ▹ n) ↓= Suc i)
        if f ∈ V₀ for f
      using that V0-def s by auto
  qed
  then show ?thesis using FIN-def by auto
```

```
qed
```

To every $f \in \mathcal{R}$ a number can be prepended that is a Gödel number of the resulting function. Such a function is then in V_0.

If V_0 was in NUM, it would be embedded in a total numbering. Shifting this numbering to the left, essentially discarding the values at point 0, would yield a total numbering for $\mathcal{R}$, which contradicts *R1-not-in-NUM*. This proves $V_0 \notin NUM$.

```
lemma prepend-goedel:
  assumes f ∈ ℛ
  shows ∃ i. φ i = [i] ⊙ f
proof −
  obtain r where r: recfn 1 r total r ⋀x. eval r [x] = f x
    using assms by auto
  define r-psi where r-psi = Cn 2 r-ifz [Id 2 1, Id 2 0, Cn 2 r [Cn 2 r-dec [Id 2 1]]]
  then have recfn 2 r-psi
    using r(1) by simp
  have eval r-psi [i, x] = (if x = 0 then Some i else f (x − 1)) for i x
  proof −
    have eval (Cn 2 r [Cn 2 r-dec [Id 2 1]]) [i, x] = f (x − 1)
      using r by simp
    then have eval r-psi [i, x] = eval r-ifz [x, i, the (f (x − 1))]
      unfolding r-psi-def using ‹recfn 2 r-psi› r R1-imp-total1[OF assms] by auto
    then show ?thesis
      using assms by simp
  qed
  with ‹recfn 2 r-psi› have (λi x. if x = 0 then Some i else f (x − 1)) ∈ 𝒫²
    by auto
  with kleene-fixed-point obtain i where
    φ i = (λx. if x = 0 then Some i else f (x − 1))
    by blast
  then have φ i = [i] ⊙ f by auto
  then show ?thesis by auto
qed

lemma V0-in-FIN-minus-NUM: V₀ ∈ FIN − NUM
proof −
  have V₀ ∉ NUM
  proof
    assume V₀ ∈ NUM
    then obtain ψ where ψ: ψ ∈ ℛ² ⋀f. f ∈ V₀ ⟹ ∃ i. ψ i = f
      by auto
    define ψ' where ψ' i x = ψ i (Suc x) for i x
    have ψ' ∈ ℛ²
    proof
      from ψ(1) obtain r-psi where
        r-psi: recfn 2 r-psi total r-psi ⋀i x. eval r-psi [i, x] = ψ i x
        by blast
      define r-psi' where r-psi' = Cn 2 r-psi [Id 2 0, Cn 2 S [Id 2 1]]
      then have recfn 2 r-psi' and ⋀i x. eval r-psi' [i, x] = ψ' i x
        unfolding r-psi'-def ψ'-def using r-psi by simp-all
      then show ψ' ∈ 𝒫² by blast
      show total2 ψ'
        using ψ'-def ψ(1) by (simp add: total2I)
    qed
    have ∃ i. ψ' i = f if f ∈ ℛ for f
```

proof −
from *that* **obtain** j **where** j: $\varphi\ j = [j] \odot f$
using *prepend-goedel* **by** *auto*
then have $\varphi\ j \in V_0$
using *that V0-altdef* **by** *auto*
with ψ **obtain** i **where** $\psi\ i = \varphi\ j$ **by** *auto*
then have $\psi'\ i = f$
using ψ'*-def j* **by** (*auto simp add*: *prepend-at-ge*)
then show *?thesis* **by** *auto*
qed
with ‹$\psi' \in \mathcal{R}^2$› **have** $\mathcal{R} \in NUM$ **by** *auto*
with *R1-not-in-NUM* **show** *False* **by** *simp*
qed
then show *?thesis*
using *V0-in-FIN* **by** *auto*
qed

corollary *FIN-not-subseteq-NUM*: $\neg\ FIN \subseteq NUM$
using *V0-in-FIN-minus-NUM* **by** *auto*

2.5 NUM and CP are incomparable

There are FIN classes outside of NUM, and CP encompasses FIN. Hence there are CP classes outside of NUM, too.

theorem *CP-not-subseteq-NUM*: $\neg\ CP \subseteq NUM$
using *FIN-subseteq-CP FIN-not-subseteq-NUM* **by** *blast*

Conversely there is a subclass of U_0 that is in NUM but cannot be learned in a class-preserving way. The following proof is due to Jantke and Beick [10]. The idea is to diagonalize against all strategies, that is, all partial recursive functions.

theorem *NUM-not-subseteq-CP*: $\neg\ NUM \subseteq CP$
proof−
— Define a family of functions f_k.
define f **where** $f \equiv \lambda k.\ [k] \odot 0^\infty$
then have $f\ k \in \mathcal{R}$ **for** k
using *almost0-in-R1* **by** *auto*

— If the strategy φ_k learns f_k it hypothesizes f_k for some shortest prefix $f_k^{a_k}$. Define functions $f'_k = k0^{a_k}10^\infty$.
define a **where**
$a \equiv \lambda k.\ LEAST\ x.\ (\varphi\ (the\ ((\varphi\ k)\ ((f\ k) \triangleright x)))) = f\ k$
define f' **where** $f' \equiv \lambda k.\ (k\ \#\ (replicate\ (a\ k)\ 0)\ @\ [1]) \odot 0^\infty$
then have $f'\ k \in \mathcal{R}$ **for** k
using *almost0-in-R1* **by** *auto*

— Although f_k and f'_k differ, they share the prefix of length $a_k + 1$.
have *init-eq*: $(f'\ k) \triangleright (a\ k) = (f\ k) \triangleright (a\ k)$ **for** k
proof (*rule init-eqI*)
fix x **assume** $x \le a\ k$
then show $f'\ k\ x = f\ k\ x$
by (*cases* $x = 0$) (*simp-all add*: *nth-append* f'*-def f-def*)
qed
have $f\ k \neq f'\ k$ **for** k
proof −

have $f\ k\ (Suc\ (a\ k)) \downarrow= 0$ **using** *f-def* **by** *auto*
moreover have $f'\ k\ (Suc\ (a\ k)) \downarrow= 1$
using *f'-def prepend*[*of* $(k\ \#\ (replicate\ (a\ k)\ 0)\ @\ [1])\ 0^{\infty}\ Suc\ (a\ k)$]
by (*metis length-Cons length-append-singleton length-replicate lessI nth-Cons-Suc nth-append-length*)
ultimately show *?thesis* **by** *auto*
qed

— The separating class U contains f'_k if φ_k learns f_k; otherwise it contains f_k.
define U **where**
$U \equiv \{if\ learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)\ then\ f'\ k\ else\ f\ k\ |k.\ k \in UNIV\}$
have $U \notin CP$
proof
assume $U \in CP$
have $\exists k.\ learn\text{-}cp\ \varphi\ U\ (\varphi\ k)$
proof −
have $\exists \psi\ s.\ learn\text{-}cp\ \psi\ U\ s$
using *CP-def* ‹$U \in CP$› **by** *auto*
then obtain s **where** s: $learn\text{-}cp\ \varphi\ U\ s$
using *learn-cp-wrt-goedel*[*OF goedel-numbering-phi*] **by** *blast*
then obtain k **where** $\varphi\ k = s$
using *phi-universal learn-cp-def learn-lim-def* **by** *auto*
then show *?thesis* **using** s **by** *auto*
qed
then obtain k **where** k: $learn\text{-}cp\ \varphi\ U\ (\varphi\ k)$ **by** *auto*
then have *learn*: $learn\text{-}lim\ \varphi\ U\ (\varphi\ k)$
using *learn-cp-def* **by** *simp*
— If f_k was in U, φ_k would learn it. But then, by definition of U, f_k would not be in U. Hence $f_k \notin U$.
have $f\ k \notin U$
proof
assume $f\ k \in U$
then obtain m **where** m: $f\ k = (if\ learn\text{-}lim\ \varphi\ \{f\ m\}\ (\varphi\ m)\ then\ f'\ m\ else\ f\ m)$
using *U-def* **by** *auto*
have $f\ k\ 0 \downarrow= m$
using *f-def f'-def m* **by** *simp*
moreover have $f\ k\ 0 \downarrow= k$ **by** (*simp add*: *f-def*)
ultimately have $m = k$ **by** *simp*
with m **have** $f\ k = (if\ learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)\ then\ f'\ k\ else\ f\ k)$
by *auto*
moreover have $learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)$
using ‹$f\ k \in U$› *learn-lim-closed-subseteq*[*OF learn*] **by** *simp*
ultimately have $f\ k = f'\ k$
by *simp*
then show *False*
using ‹$f\ k \neq f'\ k$› **by** *simp*
qed
then have $f'\ k \in U$ **using** *U-def* **by** *fastforce*
then have *in-U*: $\forall n.\ \varphi\ (the\ ((\varphi\ k)\ ((f'\ k) \triangleright n))) \in U$
using *learn-cpE*(*3*)[*OF k*] **by** *simp*

— Since $f'_k \in U$, the strategy φ_k learns f_k. Then a_k is well-defined, $f'^{a_k} = f^{a_k}$, and φ_k hypothesizes f_k on f'^{a_k}, which is not a class-preserving hypothesis.
have $learn\text{-}lim\ \varphi\ \{f\ k\}\ (\varphi\ k)$ **using** *U-def* ‹$f\ k \notin U$› **by** *fastforce*
then have $\exists i\ n_0.\ \varphi\ i = f\ k \wedge (\forall n \geq n_0.\ \varphi\ k\ ((f\ k) \triangleright n) \downarrow= i)$
using *learn-limE*(*2*) **by** *simp*

then obtain $i\ n_0$ **where** $\varphi\ i = f\ k \wedge (\forall\, n{\geq}n_0.\ \varphi\ k\ ((f\ k) \triangleright n) \downarrow= i)$
by *auto*
then have $\varphi\ (\mathit{the}\ (\varphi\ k\ ((f\ k) \triangleright (a\ k)))) = f\ k$
using *a-def LeastI*$[\mathit{of}\ \lambda x.\ (\varphi\ (\mathit{the}\ ((\varphi\ k)\ ((f\ k) \triangleright x)))) = f\ k\ n_0]$
by *simp*
then have $\varphi\ (\mathit{the}\ ((\varphi\ k)\ ((f'\ k) \triangleright (a\ k)))) = f\ k$
using *init-eq* **by** *simp*
then show *False*
using ‹$f\ k \notin U$› *in-U* **by** *metis*
qed
moreover have $U \in \mathit{NUM}$
using *NUM-closed-subseteq*[*OF U0-in-NUM*, *of U*] *f-def f′-def U0-def U-def*
by *fastforce*
ultimately show *?thesis* **by** *auto*
qed

2.6 NUM is a proper subset of TOTAL

A NUM class U is embedded in a total numbering ψ. The strategy S with $S(f^n) = \min\{i \mid \forall k \leq n : \psi_i(k) = f(k)\}$ for $f \in U$ converges to the least index of f in ψ, and thus learns f in the limit. Moreover it will be a TOTAL strategy because ψ contains only total functions. This shows $\mathit{NUM} \subseteq \mathit{TOTAL}$.

First we define, for every hypothesis space ψ, a function that tries to determine for a given list e and index i whether e is a prefix of ψ_i. In other words it tries to decide whether i is a consistent hypothesis for e. "Tries" refers to the fact that the function will diverge if $\psi_i(x) \uparrow$ for any $x \leq |e|$. We start with a version that checks the list only up to a given length.

definition *r-consist-upto* :: *recf* ⇒ *recf* **where**
r-consist-upto r-psi ≡
let g = *Cn 4 r-ifeq*
[*Cn 4 r-psi* [*Id 4 2*, *Id 4 0*], *Cn 4 r-nth* [*Id 4 3*, *Id 4 0*], *Id 4 1*, *r-constn 3 1*]
in Pr 2 (*r-constn 1 0*) *g*

lemma *r-consist-upto-recfn*: *recfn 2 r-psi* ⟹ *recfn 3* (*r-consist-upto r-psi*)
using *r-consist-upto-def* **by** *simp*

lemma *r-consist-upto*:
assumes *recfn 2 r-psi*
shows ∀ *k*<*j*. *eval r-psi* [*i*, *k*] ↓ ⟹
eval (*r-consist-upto r-psi*) [*j*, *i*, *e*] =
(*if* ∀ *k*<*j*. *eval r-psi* [*i*, *k*] ↓= *e-nth e k then Some 0 else Some 1*)
and ¬ (∀ *k*<*j*. *eval r-psi* [*i*, *k*] ↓) ⟹ *eval* (*r-consist-upto r-psi*) [*j*, *i*, *e*] ↑
proof −
define *g* **where** *g* =
Cn 4 r-ifeq
[*Cn 4 r-psi* [*Id 4 2*, *Id 4 0*], *Cn 4 r-nth* [*Id 4 3*, *Id 4 0*], *Id 4 1*, *r-constn 3 1*]
then have *recfn 4 g*
using *assms* **by** *simp*
moreover have *eval* (*Cn 4 r-nth* [*Id 4 3*, *Id 4 0*]) [*j*, *r*, *i*, *e*] ↓= *e-nth e j* **for** *j r i e*
by *simp*
moreover have *eval* (*r-constn 3 1*) [*j*, *r*, *i*, *e*] ↓= *1* **for** *j r i e*
by *simp*
moreover have *eval* (*Cn 4 r-psi* [*Id 4 2*, *Id 4 0*]) [*j*, *r*, *i*, *e*] = *eval r-psi* [*i*, *j*] **for** *j r i e*

```
  using assms(1) by simp
ultimately have g: eval g [j, r, i, e] =
  (if eval r-psi [i, j] ↑ then None
   else if eval r-psi [i, j] ↓= e-nth e j then Some r else Some 1)
  for j r i e
  using ‹recfn 4 g› g-def assms by auto
have goal1: ∀k<j. eval r-psi [i, k] ↓ ⟹
  eval (r-consist-upto r-psi) [j, i, e] =
    (if ∀k<j. eval r-psi [i, k] ↓= e-nth e k then Some 0 else Some 1)
  for j i e
proof (induction j)
  case 0
  then show ?case
    using r-consist-upto-def r-consist-upto-recfn assms eval-Pr-0 by simp
next
  case (Suc j)
  then have eval (r-consist-upto r-psi) [Suc j, i, e] =
      eval g [j, the (eval (r-consist-upto r-psi) [j, i, e]), i, e]
    using assms eval-Pr-converg-Suc g-def r-consist-upto-def r-consist-upto-recfn
    by simp
  also have ... = eval g [j, if ∀k<j. eval r-psi [i, k] ↓= e-nth e k then 0 else 1, i, e]
    using Suc by auto
  also have ... ↓= (if eval r-psi [i, j] ↓= e-nth e j
      then if ∀k<j. eval r-psi [i, k] ↓= e-nth e k then 0 else 1 else 1)
    using g by (simp add: Suc.prems)
  also have ... ↓= (if ∀k<Suc j. eval r-psi [i, k] ↓= e-nth e k then 0 else 1)
    by (simp add: less-Suc-eq)
  finally show ?case by simp
qed
then show ∀k<j. eval r-psi [i, k] ↓ ⟹
  eval (r-consist-upto r-psi) [j, i, e] =
  (if ∀k<j. eval r-psi [i, k] ↓= e-nth e k then Some 0 else Some 1)
  by simp
show ¬ (∀k<j. eval r-psi [i, k] ↓) ⟹ eval (r-consist-upto r-psi) [j, i, e] ↑
proof −
  assume ¬ (∀k<j. eval r-psi [i, k] ↓)
  then have ∃k<j. eval r-psi [i, k] ↑ by simp
  let ?P = λk. k < j ∧ eval r-psi [i, k] ↑
  define kmin where kmin = Least ?P
  then have ?P kmin
    using LeastI-ex[of ?P] ‹∃k<j. eval r-psi [i, k] ↑› by auto
  from kmin-def have ⋀k. k < kmin ⟹ ¬ ?P k
    using kmin-def not-less-Least[of - ?P] by blast
  then have ∀k < kmin. eval r-psi [i, k] ↓
    using ‹?P kmin› by simp
  then have eval (r-consist-upto r-psi) [kmin, i, e] =
      (if ∀k<kmin. eval r-psi [i, k] ↓= e-nth e k then Some 0 else Some 1)
    using goal1 by simp
  moreover have eval r-psi [i, kmin] ↑
    using ‹?P kmin› by simp
  ultimately have eval (r-consist-upto r-psi) [Suc kmin, i, e] ↑
    using r-consist-upto-def g assms by simp
  moreover have j ≥ kmin
    using ‹?P kmin› by simp
  ultimately show eval (r-consist-upto r-psi) [j, i, e] ↑
    using r-consist-upto-def r-consist-upto-recfn ‹?P kmin› eval-Pr-converg-le assms
```

```
    by (metis (full-types) Suc-leI length-Cons list.size(3) numeral-2-eq-2 numeral-3-eq-3)
  qed
qed
```

The next function provides the consistency decision functions we need.

```
definition consistent :: partial2 ⇒ partial2 where
  consistent ψ i e ≡
    if ∀ k<e-length e. ψ i k ↓
    then if ∀ k<e-length e. ψ i k ↓= e-nth e k
         then Some 0 else Some 1
    else None
```

Given i and e, *consistent* ψ decides whether e is a prefix of ψ_i, provided ψ_i is defined for the length of e.

```
definition r-consistent :: recf ⇒ recf where
  r-consistent r-psi ≡
    Cn 2 (r-consist-upto r-psi) [Cn 2 r-length [Id 2 1], Id 2 0, Id 2 1]

lemma r-consistent-recfn [simp]: recfn 2 r-psi ⟹ recfn 2 (r-consistent r-psi)
  using r-consistent-def r-consist-upto-recfn by simp

lemma r-consistent-converg:
  assumes recfn 2 r-psi and ∀ k<e-length e. eval r-psi [i, k] ↓
  shows eval (r-consistent r-psi) [i, e] ↓=
    (if ∀ k<e-length e. eval r-psi [i, k] ↓= e-nth e k then 0 else 1)
proof −
  have eval (r-consistent r-psi) [i, e] = eval (r-consist-upto r-psi) [e-length e, i, e]
    using r-consistent-def r-consist-upto-recfn assms(1) by simp
  then show ?thesis using assms r-consist-upto(1) by simp
qed

lemma r-consistent-diverg:
  assumes recfn 2 r-psi and ∃ k<e-length e. eval r-psi [i, k] ↑
  shows eval (r-consistent r-psi) [i, e] ↑
  unfolding r-consistent-def
  using r-consist-upto-recfn[OF assms(1)] r-consist-upto[OF assms(1)] assms(2)
  by simp

lemma r-consistent:
  assumes recfn 2 r-psi and ∀ x y. eval r-psi [x, y] = ψ x y
  shows eval (r-consistent r-psi) [i, e] = consistent ψ i e
proof (cases ∀ k<e-length e. ψ i k ↓)
  case True
  then have ∀ k<e-length e. eval r-psi [i, k] ↓
    using assms by simp
  then show ?thesis
    unfolding consistent-def using True by (simp add: assms r-consistent-converg)
next
  case False
  then have consistent ψ i e ↑
    unfolding consistent-def by auto
  moreover have eval (r-consistent r-psi) [i, e] ↑
    using r-consistent-diverg[OF assms(1)] assms False by simp
  ultimately show ?thesis by simp
qed
```

lemma *consistent-in-P2*:
assumes $\psi \in \mathcal{P}^2$
shows *consistent* $\psi \in \mathcal{P}^2$
using *assms r-consistent P2E*[*OF assms*(*1*)] *P2I r-consistent-recfn* **by** *metis*

lemma *consistent-for-R2*:
assumes $\psi \in \mathcal{R}^2$
shows *consistent* ψ *i e* =
(*if* $\forall$ *j*<*e-length e*. ψ *i j* ↓= *e-nth e j then Some 0 else Some 1*)
using *assms* **by** (*simp add*: *consistent-def*)

lemma *consistent-init*:
assumes $\psi \in \mathcal{R}^2$ **and** $f \in \mathcal{R}$
shows *consistent* ψ *i* (*f* ▷ *n*) = (*if* ψ *i* ▷ *n* = *f* ▷ *n then Some 0 else Some 1*)
using *consistent-def*[*of - - init f n*] *assms init-eq-iff-eq-upto* **by** *simp*

lemma *consistent-in-R2*:
assumes $\psi \in \mathcal{R}^2$
shows *consistent* $\psi \in \mathcal{R}^2$
using *total2I consistent-in-P2 consistent-for-R2*[*OF assms*] *P2-total-imp-R2 R2-imp-P2 assms*
by (*metis option.simps*(*3*))

For total hypothesis spaces the next function computes the minimum hypothesis consistent with a given prefix. It diverges if no such hypothesis exists.

definition *min-cons-hyp* :: *partial2* ⇒ *partial1* **where**
min-cons-hyp ψ *e* ≡
if ∃ *i*. *consistent* ψ *i e* ↓= *0 then Some* (*LEAST i*. *consistent* ψ *i e* ↓= *0*) *else None*

lemma *min-cons-hyp-in-P1*:
assumes $\psi \in \mathcal{R}^2$
shows *min-cons-hyp* $\psi \in \mathcal{P}$
proof −
from *assms consistent-in-R2* **obtain** *rc* **where**
rc: *recfn 2 rc total rc* ⋀*i e*. *eval rc* [*i*, *e*] = *consistent* ψ *i e*
using *R2E*[*of consistent* ψ] **by** *metis*
define *r* **where** *r* = *Mn 1 rc*
then have *recfn 1 r*
using *rc*(*1*) **by** *simp*
moreover from *this* **have** *eval r* [*e*] = *min-cons-hyp* ψ *e* **for** *e*
using *r-def eval-Mn′*[*of 1 rc* [*e*]] *rc min-cons-hyp-def assms*
by (*auto simp add*: *consistent-in-R2*)
ultimately show *?thesis* **by** *auto*
qed

The function *min-cons-hyp* ψ is a strategy for learning all NUM classes embedded in ψ. It is an example of an "identification-by-enumeration" strategy.

lemma *NUM-imp-learn-total*:
assumes $\psi \in \mathcal{R}^2$ **and** *U* ∈ *NUM-wrt* ψ
shows *learn-total* ψ *U* (*min-cons-hyp* ψ)
proof (*rule learn-totalI*)
have *ex-psi-i-f*: ∃ *i*. ψ *i* = *f* **if** *f* ∈ *U* **for** *f*
using *assms that NUM-wrt-def* **by** *simp*
moreover have *consistent-eq-0*: *consistent* ψ *i* ((ψ *i*) ▷ *n*) ↓= *0* **for** *i n*
using *assms* **by** (*simp add*: *consistent-init*)

```
ultimately have ⋀f n. f ∈ U ⟹ min-cons-hyp ψ (f ▷ n) ↓
  using min-cons-hyp-def assms(1) by fastforce
then show env: environment ψ U (min-cons-hyp ψ)
  using assms NUM-wrt-def min-cons-hyp-in-P1 NUM-E(1) NUM-I by auto

show ⋀f n. f ∈ U ⟹ ψ (the (min-cons-hyp ψ (f ▷ n))) ∈ ℛ
  using assms by (simp)

show ∃i. ψ i = f ∧ (∀∞n. min-cons-hyp ψ (f ▷ n) ↓= i) if f ∈ U for f
proof −
  from that env have f ∈ ℛ by auto

  let ?P = λi. ψ i = f
  define imin where imin ≡ Least ?P
  with ex-psi-i-f that have imin: ?P imin ⋀j. ?P j ⟹ j ≥ imin
    using LeastI-ex[of ?P] Least-le[of ?P] by simp-all
  then have f-neq: ψ i ≠ f if i < imin for i
    using leD that by auto

  let ?Q = λi n. ψ i ▷ n ≠ f ▷ n
  define nu :: nat ⇒ nat where nu = (λi. SOME n. ?Q i n)
  have nu-neq: ψ i ▷ (nu i) ≠ f ▷ (nu i) if i < imin for i
  proof −
    from assms have ψ i ∈ ℛ by simp
    moreover from assms imin(1) have f ∈ ℛ by auto
    moreover have f ≠ ψ i
      using that f-neq by auto
    ultimately have ∃n. f ▷ n ≠ (ψ i) ▷ n
      using neq-fun-neq-init by simp
    then show ?Q i (nu i)
      unfolding nu-def using someI-ex[of λn. ?Q i n] by metis
  qed

  have ∃n₀. ∀n≥n₀. min-cons-hyp ψ (f ▷ n) ↓= imin
  proof (cases imin = 0)
    case True
    then have ∀n. min-cons-hyp ψ (f ▷ n) ↓= imin
      using consistent-eq-0 assms(1) imin(1) min-cons-hyp-def by auto
    then show ?thesis by simp
  next
    case False
    define n₀ where n₀ = Max (set (map nu [0..<imin])) (is - = Max ?N)
    have nu i ≤ n₀ if i < imin for i
    proof −
      have finite ?N
        using n₀-def by simp
      moreover have ?N ≠ {}
        using False n₀-def by simp
      moreover have nu i ∈ ?N
        using that by simp
      ultimately show ?thesis
        using that Max-ge n₀-def by blast
    qed
    then have ψ i ▷ n₀ ≠ f ▷ n₀ if i < imin for i
      using nu-neq neq-init-forall-ge that by blast
    then have ∗: ψ i ▷ n ≠ f ▷ n if i < imin and n ≥ n₀ for i n
```

using *nu-neq neq-init-forall-ge that* **by** *blast*

have ψ *imin* $\triangleright n = f \triangleright n$ **for** n
using *imin*(*1*) **by** *simp*
moreover have (*consistent* ψ i $(f \triangleright n) \downarrow= 0) = (\psi\ i \triangleright n = f \triangleright n)$ **for** i n
by (*simp add*: ‹$f \in \mathcal{R}$› *assms*(*1*) *consistent-init*)
ultimately have *min-cons-hyp* ψ $(f \triangleright n) \downarrow= ($*LEAST* $i.\ \psi\ i \triangleright n = f \triangleright n)$ **for** n
using *min-cons-hyp-def*[*of* ψ $f \triangleright n$] **by** *auto*
moreover have (*LEAST* $i.\ \psi\ i \triangleright n = f \triangleright n)$ = *imin* **if** $n \geq n_0$ **for** n
proof (*rule Least-equality*)
show ψ *imin* $\triangleright n = f \triangleright n$
using *imin*(*1*) **by** *simp*
show $\bigwedge y.\ \psi\ y \triangleright n = f \triangleright n \Longrightarrow$ *imin* $\leq y$
using *imin* $*$ *leI that* **by** *blast*
qed
ultimately have *min-cons-hyp* ψ $(f \triangleright n) \downarrow=$ *imin* **if** $n \geq n_0$ **for** n
using *that* **by** *blast*
then show *?thesis* **by** *auto*
qed
with *imin*(*1*) **show** *?thesis* **by** *auto*
qed
qed

corollary *NUM-subseteq-TOTAL*: *NUM* $\subseteq$ *TOTAL*
proof
fix U
assume $U \in$ *NUM*
then have $\exists \psi \in \mathcal{R}^2.\ \forall f \in U.\ \exists i.\ \psi\ i = f$ **by** *auto*
then have $\exists \psi \in \mathcal{R}^2.\ U \in$ *NUM-wrt* ψ
using *NUM-wrt-def* **by** *simp*
then have $\exists \psi\ s.$ *learn-total* ψ U s
using *NUM-imp-learn-total* **by** *auto*
then show $U \in$ *TOTAL*
using *TOTAL-def* **by** *auto*
qed

The class V_0 is in *TOTAL* $-$ *NUM*.

theorem *NUM-subset-TOTAL*: *NUM* $\subset$ *TOTAL*
using *CP-subseteq-TOTAL FIN-not-subseteq-NUM FIN-subseteq-CP NUM-subseteq-TOTAL*
by *auto*

end

2.7 CONS is a proper subset of LIM

theory *CONS-LIM*
imports *Inductive-Inference-Basics*
begin

That there are classes in *LIM* $-$ *CONS* was noted by Barzdin [4, 3] and Blum and Blum [5]. It was proven by Wiehagen [15] (see also Wiehagen and Zeugmann [16]). The proof uses this class:

definition *U-LIMCONS* :: *partial1 set* (‹$U_{LIM-CONS}$›) **where**
$U_{LIM-CONS} \equiv \{vs\ @\ [j] \odot p \mid vs\ j\ p.\ j \geq 2 \wedge p \in \mathcal{R}_{01} \wedge \varphi\ j = vs\ @\ [j] \odot p\}$

Every function in $U_{LIM-CONS}$ carries a Gödel number greater or equal two of itself, after which only zeros and ones occur. Thus, a strategy that always outputs the rightmost value greater or equal two in the given prefix will converge to this Gödel number.

The next function searches an encoded list for the rightmost element greater or equal two.

```
definition rmge2 :: partial1 where
  rmge2 e ≡
    if ∀ i<e-length e. e-nth e i < 2 then Some 0
    else Some (e-nth e (GREATEST i. i < e-length e ∧ e-nth e i ≥ 2))

lemma rmge2:
  assumes xs = list-decode e
  shows rmge2 e =
   (if ∀ i<length xs. xs ! i < 2 then Some 0
    else Some (xs ! (GREATEST i. i < length xs ∧ xs ! i ≥ 2)))
proof −
  have (i < e-length e ∧ e-nth e i ≥ 2) = (i < length xs ∧ xs ! i ≥ 2) for i
    using assms by simp
  then have (GREATEST i. i < e-length e ∧ e-nth e i ≥ 2) =
      (GREATEST i. i < length xs ∧ xs ! i ≥ 2)
    by simp
  moreover have (∀ i<length xs. xs ! i < 2) = (∀ i<e-length e. e-nth e i < 2)
    using assms by simp
  moreover have (GREATEST i. i < length xs ∧ xs ! i ≥ 2) < length xs (is Greatest ?P < -)
      if ¬ (∀ i<length xs. xs ! i < 2)
    using that GreatestI-ex-nat[of ?P] le-less-linear order.asym by blast
  ultimately show ?thesis using rmge2-def assms by auto
qed

lemma rmge2-init:
  rmge2 (f ▷ n) =
   (if ∀ i<Suc n. the (f i) < 2 then Some 0
    else Some (the (f (GREATEST i. i < Suc n ∧ the (f i) ≥ 2))))
proof −
  let ?xs = prefix f n
  have f ▷ n = list-encode ?xs by (simp add: init-def)
  moreover have (∀ i<Suc n. the (f i) < 2) = (∀ i<length ?xs. ?xs ! i < 2)
    by simp
  moreover have (GREATEST i. i < Suc n ∧ the (f i) ≥ 2) =
      (GREATEST i. i < length ?xs ∧ ?xs ! i ≥ 2)
    using length-prefix[of f n] prefix-nth[of - n f] by metis
  moreover have (GREATEST i. i < Suc n ∧ the (f i) ≥ 2) < Suc n
      if ¬ (∀ i<Suc n. the (f i) < 2)
    using that GreatestI-ex-nat[of λi. i<Suc n ∧ the (f i) ≥ 2 n] by fastforce
  ultimately show ?thesis using rmge2 by auto
qed

corollary rmge2-init-total:
  assumes total1 f
  shows rmge2 (f ▷ n) =
   (if ∀ i<Suc n. the (f i) < 2 then Some 0
    else f (GREATEST i. i < Suc n ∧ the (f i) ≥ 2))
  using assms total1-def rmge2-init by auto

lemma rmge2-in-R1: rmge2 ∈ ℛ
```

```
proof -
  define g where
    g = Cn 3 r-ifle [r-constn 2 2, Cn 3 r-nth [Id 3 2, Id 3 0], Cn 3 r-nth [Id 3 2, Id 3 0], Id 3 1]
  then have recfn 3 g by simp
  then have g: eval g [j, r, e] ↓= (if 2 ≤ e-nth e j then e-nth e j else r) for j r e
    using g-def by simp

  let ?h = Pr 1 Z g
  have recfn 2 ?h
    by (simp add: ‹recfn 3 g›)
  have h: eval ?h [j, e] =
   (if ∀i<j. e-nth e i < 2 then Some 0
    else Some (e-nth e (GREATEST i. i < j ∧ e-nth e i ≥ 2))) for j e
  proof (induction j)
    case 0
    then show ?case using ‹recfn 2 ?h› by auto
  next
    case (Suc j)
    then have eval ?h [Suc j, e] = eval g [j, the (eval ?h [j, e]), e]
      using ‹recfn 2 ?h› by auto
    then have *: eval ?h [Suc j, e] ↓=
      (if 2 ≤ e-nth e j then e-nth e j
       else if ∀i<j. e-nth e i < 2 then 0
            else (e-nth e (GREATEST i. i < j ∧ e-nth e i ≥ 2)))
      using g Suc by auto
    show ?case
    proof (cases ∀i<Suc j. e-nth e i < 2)
      case True
      then show ?thesis using * by auto
    next
      case ex: False
      show ?thesis
      proof (cases 2 ≤ e-nth e j)
        case True
        then have eval ?h [Suc j, e] ↓= e-nth e j
          using * by simp
        moreover have (GREATEST i. i < Suc j ∧ e-nth e i ≥ 2) = j
          using ex True Greatest-equality[of λi. i < Suc j ∧ e-nth e i ≥ 2]
          by simp
        ultimately show ?thesis using ex by auto
      next
        case False
        then have ∃i<j. e-nth e i ≥ 2
          using ex leI less-Suc-eq by blast
        with * have eval ?h [Suc j, e] ↓= e-nth e (GREATEST i. i < j ∧ e-nth e i ≥ 2)
          using False by (smt leD)
        moreover have (GREATEST i. i < Suc j ∧ e-nth e i ≥ 2) =
            (GREATEST i. i < j ∧ e-nth e i ≥ 2)
          using False ex by (metis less-SucI less-Suc-eq less-antisym numeral-2-eq-2)
        ultimately show ?thesis using ex by metis
      qed
    qed
  qed

  let ?hh = Cn 1 ?h [Cn 1 r-length [Id 1 0], Id 1 0]
  have recfn 1 ?hh
```

using ‹*recfn 2 ?h*› **by** *simp*
with *h* **have** *hh*: *eval ?hh* [*e*] $\downarrow=$
(*if* $\forall$ *i*<*e-length e. e-nth e i* < *2 then 0*
else e-nth e (*GREATEST i. i* < *e-length e* $\wedge$ *e-nth e i* $\geq$ *2*)) **for** *e*
by *auto*
then have *eval ?hh* [*e*] = *rmge2 e* **for** *e*
unfolding *rmge2-def* **by** *auto*
moreover have *total ?hh*
using *hh totalI1* ‹*recfn 1 ?hh*› **by** *simp*
ultimately show *?thesis* **using** ‹*recfn 1 ?hh*› **by** *blast*
qed

The first part of the main result is that $U_{LIM-CONS} \in LIM$.

lemma *U-LIMCONS-in-Lim*: $U_{LIM-CONS} \in LIM$
proof −
have $U_{LIM-CONS} \subseteq \mathcal{R}$
unfolding *U-LIMCONS-def* **using** *prepend-in-R1 RPred1-subseteq-R1* **by** *blast*
have *learn-lim* φ $U_{LIM-CONS}$ *rmge2*
proof (*rule learn-limI*)
show *environment* φ $U_{LIM-CONS}$ *rmge2*
using ‹U-*LIMCONS* $\subseteq \mathcal{R}$› *phi-in-P2 rmge2-def rmge2-in-R1* **by** *simp*
show $\exists$ *i*. φ *i* = *f* $\wedge$ ($\forall^\infty$*n. rmge2* (*f* $\triangleright$ *n*) $\downarrow=$ *i*) **if** *f* $\in U_{LIM-CONS}$ **for** *f*
proof −
from *that* **obtain** *vs j p* **where**
j: *j* $\geq$ *2*
and *p*: *p* $\in \mathcal{R}_{01}$
and *s*: φ *j* = *vs* @ [*j*] $\odot$ *p*
and *f*: *f* = *vs* @ [*j*] $\odot$ *p*
unfolding *U-LIMCONS-def* **by** *auto*
then have φ *j* = *f* **by** *simp*
from *that* **have** *total1 f*
using ‹$U_{LIM-CONS} \subseteq \mathcal{R}$› *R1-imp-total1 total1-def* **by** *auto*
define n_0 **where** n_0 = *length vs*
have *f-gr-n0*: *f n* $\downarrow=$ *0* $\vee$ *f n* $\downarrow=$ *1* **if** *n* > n_0 **for** *n*
proof −
have *f n* = *p* (*n* − n_0 − *1*)
using *that* n_0-*def f* **by** *simp*
with *RPred1-def p* **show** *?thesis* **by** *auto*
qed
have *rmge2* (*f* $\triangleright$ *n*) $\downarrow=$ *j* **if** *n* $\geq$ n_0 **for** *n*
proof −
have *n0-greatest*: (*GREATEST i. i* < *Suc n* $\wedge$ *the* (*f i*) $\geq$ *2*) = n_0
proof (*rule Greatest-equality*)
show n_0 < *Suc n* $\wedge$ *the* (*f* n_0) $\geq$ *2*
using n_0-*def f that j* **by** *simp*
show $\bigwedge$*y. y* < *Suc n* $\wedge$ *the* (*f y*) $\geq$ *2* $\Longrightarrow$ *y* $\leq$ n_0
proof −
fix *y* **assume** *y* < *Suc n* $\wedge$ *2* $\leq$ *the* (*f y*)
moreover have *p* $\in \mathcal{R}$ $\wedge$ ($\forall$ *n. p n* $\downarrow=$ *0* $\vee$ *p n* $\downarrow=$ *1*)
using *RPred1-def p* **by** *blast*
ultimately show *y* $\leq$ n_0
using *f-gr-n0*
by (*metis Suc-1 Suc-n-not-le-n Zero-neq-Suc le-less-linear le-zero-eq option.sel*)
qed
qed
have *f* n_0 $\downarrow=$ *j*

using n_0-*def f* **by** *simp*
then have ¬ (∀ *i*<*Suc n*. *the* (*f i*) < *2*)
using *j that less-Suc-eq-le* **by** *auto*
then have *rmge2* (*f* ▷ *n*) = *f* (*GREATEST i*. *i* < *Suc n* ∧ *the* (*f i*) ≥ *2*)
using *rmge2-init-total* ‹*total1 f*› **by** *auto*
with *n0-greatest* ‹*f* n_0 ↓= *j*› **show** *?thesis* **by** *simp*
qed
with ‹φ *j* = *f*› **show** *?thesis* **by** *auto*
qed
qed
then show *?thesis* **using** *Lim-def* **by** *auto*
qed

The class $U_{LIM-CONS}$ is *prefix-complete*, which means that every non-empty list is the prefix of some function in $U_{LIM-CONS}$. To show this we use an auxiliary lemma: For every $f \in \mathcal{R}$ and $k \in \mathbb{N}$ the value of f at k can be replaced by a Gödel number of the function resulting from the replacement.

lemma *goedel-at*:
fixes *m* :: *nat* **and** *k* :: *nat*
assumes *f* ∈ $\mathcal{R}$
shows ∃ *n*≥*m*. φ *n* = (λ*x*. *if x* = *k then Some n else f x*)
proof −
define *psi* :: *partial1* ⇒ *nat* ⇒ *partial2* **where**
psi = (λ*f k i x*. (*if x* = *k then Some i else f x*))
have *psi f k* ∈ $\mathcal{R}^2$
proof −
obtain *r* **where** *r*: *recfn 1 r total r eval r* [*x*] = *f x* **for** *x*
using *assms* **by** *auto*
define *r-psi* **where**
r-psi = *Cn 2 r-ifeq* [*Id 2 1*, *r-dummy 1* (*r-const k*), *Id 2 0*, *Cn 2 r* [*Id 2 1*]]
show *?thesis*
proof (*rule R2I*[*of r-psi*])
from *r-psi-def* **show** *recfn 2 r-psi*
using *r*(*1*) **by** *simp*
have *eval r-psi* [*i*, *x*] = (*if x* = *k then Some i else f x*) **for** *i x*
proof −
have *eval* (*Cn 2 r* [*Id 2 1*]) [*i*, *x*] = *f x*
using *r* **by** *simp*
then have *eval r-psi* [*i*, *x*] = *eval r-ifeq* [*x*, *k*, *i*, *the* (*f x*)]
unfolding *r-psi-def* **using** ‹*recfn 2 r-psi*› *r R1-imp-total1*[*OF assms*]
by *simp*
then show *?thesis* **using** *assms* **by** *simp*
qed
then show ⋀*x y*. *eval r-psi* [*x*, *y*] = *psi f k x y*
unfolding *psi-def* **by** *simp*
then show *total r-psi*
using *totalI2*[*of r-psi*] ‹*recfn 2 r-psi*› *assms psi-def* **by** *fastforce*
qed
qed
then obtain *n* **where** *n* ≥ *m* φ *n* = *psi f k n*
using *assms kleene-fixed-point*[*of psi f k m*] **by** *auto*
then show *?thesis* **unfolding** *psi-def* **by** *auto*
qed

lemma *U-LIMCONS-prefix-complete*:

assumes *length vs* > *0*
shows $\exists f \in U_{LIM-CONS}$. *prefix f* (*length vs* − *1*) = *vs*
proof −
let *?p* = λ-. *Some 0*
let *?f* = *vs* @ [*0*] ⊙ *?p*
have *?f* ∈ $\mathcal{R}$
using *prepend-in-R1 RPred1-subseteq-R1 const0-in-RPred1* **by** *blast*
with *goedel-at*[*of ?f 2 length vs*] **obtain** *j* **where**
j: *j* ≥ *2* φ *j* = (λ*x*. *if x* = *length vs then Some j else ?f x*) (**is** - = *?g*)
by *auto*
moreover have *g*: *?g x* = (*vs* @ [*j*] ⊙ *?p*) *x* **for** *x*
by (*simp add*: *nth-append*)
ultimately have *?g* ∈ $U_{LIM-CONS}$
unfolding *U-LIMCONS-def* **using** *const0-in-RPred1* **by** *fastforce*
moreover have *prefix ?g* (*length vs* − *1*) = *vs*
using *g assms prefixI prepend-associative* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

Roughly speaking, a strategy learning a prefix-complete class must be total because it must be defined for every prefix in the class. Technically, however, the empty list is not a prefix, and thus a strategy may diverge on input 0. We can work around this by showing that if there is a strategy learning a prefix-complete class then there is also a total strategy learning this class. We need the result only for consistent learning.

lemma *U-prefix-complete-imp-total-strategy*:
assumes ⋀*vs*. *length vs* > *0* ⟹ ∃*f*∈*U*. *prefix f* (*length vs* − *1*) = *vs*
and *learn-cons* ψ *U s*
shows ∃*t*. *total1 t* ∧ *learn-cons* ψ *U t*
proof −
define *t* **where** *t* = (λ*e*. *if e* = *0 then Some 0 else s e*)
have *s e* ↓ **if** *e* > *0* **for** *e*
proof −
from *that* **have** *list-decode e* ≠ [] (**is** *?vs* ≠ -)
using *list-encode-0 list-encode-decode* **by** (*metis less-imp-neq*)
then have *length ?vs* > *0* **by** *simp*
with *assms*(*1*) **obtain** *f* **where** *f*: *f* ∈ *U prefix f* (*length ?vs* − *1*) = *?vs*
by *auto*
with *learn-cons-def learn-limE* **have** *s* (*f* ▷ (*length ?vs* − *1*)) ↓
using *assms*(*2*) **by** *auto*
then show *s e* ↓
using *f*(*2*) *init-def* **by** *auto*
qed
then have *total1 t*
using *t-def* **by** *auto*
have *t* ∈ $\mathcal{P}$
proof −
from *assms*(*2*) **have** *s* ∈ $\mathcal{P}$
using *learn-consE* **by** *simp*
then obtain *rs* **where** *rs*: *recfn 1 rs eval rs* [*x*] = *s x* **for** *x*
by *auto*
define *rt* **where** *rt* = *Cn 1* (*r-lifz Z rs*) [*Id 1 0*, *Id 1 0*]
then have *recfn 1 rt*
using *rs* **by** *auto*
moreover have *eval rt* [*x*] = *t x* **for** *x*
using *rs rt-def t-def* **by** *simp*

ultimately show *?thesis* **by** *blast*
qed
have $s\ (f \triangleright n) = t\ (f \triangleright n)$ **if** $f \in U$ **for** *f n*
unfolding *t-def* **by** (*simp add*: *init-neq-zero*)
then have *learn-cons* ψ *U t*
using ‹$t \in \mathcal{P}$› *assms(2) learn-consE*[*of* ψ *U s*] *learn-consI*[*of* ψ *U t*] **by** *simp*
with ‹*total1 t*› **show** *?thesis* **by** *auto*
qed

The proof of $U_{LIM-CONS} \notin CONS$ is by contradiction. Assume there is a consistent learning strategy S. By the previous lemma S can be assumed to be total. Moreover it outputs a consistent hypothesis for every prefix. Thus for every $e \in \mathbb{N}^+$, $S(e) \neq S(e0)$ or $S(e) \neq S(e1)$ because $S(e)$ cannot be consistent with both $e0$ and $e1$. We use this property of S to construct a function in $U_{LIM-CONS}$ for which S fails as a learning strategy. To this end we define a numbering $\psi \in \mathcal{R}^2$ with $\psi_i(0) = i$ and

$$\psi_i(x+1) = \begin{cases} 0 & \text{if } S(\psi_i^x 0) \neq S(\psi_i^x), \\ 1 & \text{otherwise.} \end{cases}$$

This numbering is recursive because S is total. The "otherwise" case is equivalent to $S(\psi_i^x 1) \neq S(\psi_i^x)$ because $S(\psi_i^x)$ cannot be consistent with both $\psi_i^x 0$ and $\psi_i^x 1$. Therefore every prefix ψ_i^x is extended in such a way that S changes its hypothesis. Hence S does not learn ψ_i in the limit. Kleene's fixed-point theorem ensures that for some $j \geq 2$, $\varphi_j = \psi_j$. This ψ_j is the sought function in $U_{LIM-CONS}$.
The following locale formalizes the construction of ψ for a total strategy S.

locale *cons-lim* =
fixes *s* :: *partial1*
assumes *s-in-R1*: $s \in \mathcal{R}$
begin

A *recf* computing the strategy:

definition *r-s* :: *recf* **where**
r-s ≡ *SOME r-s. recfn 1 r-s* ∧ *total r-s* ∧ *s* = (λ*x. eval r-s* [*x*])

lemma *r-s-recfn* [*simp*]: *recfn 1 r-s*
and *r-s-total* [*simp*]: ⋀*x. eval r-s* [*x*] ↓
and *eval-r-s*: *s* = (λ*x. eval r-s* [*x*])
using *r-s-def R1-SOME*[*OF s-in-R1*, *of r-s*] **by** *simp-all*

The next function represents the prefixes of ψ_i.

fun *prefixes* :: *nat* ⇒ *nat* ⇒ *nat list* **where**
prefixes i 0 = [*i*]
| *prefixes i* (*Suc x*) = (*prefixes i x*) @
[*if s* (*e-snoc* (*list-encode* (*prefixes i x*)) *0*) = *s* (*list-encode* (*prefixes i x*))
then 1 else 0]

definition *r-prefixes-aux* ≡
Cn 3 r-ifeq
[*Cn 3 r-s* [*Cn 3 r-snoc* [*Id 3 1*, *r-constn 2 0*]],
Cn 3 r-s [*Id 3 1*],
Cn 3 r-snoc [*Id 3 1*, *r-constn 2 1*],
Cn 3 r-snoc [*Id 3 1*, *r-constn 2 0*]]

lemma *r-prefixes-aux-recfn*: *recfn 3 r-prefixes-aux*

unfolding *r-prefixes-aux-def* **by** *simp*

lemma *r-prefixes-aux*:
eval r-prefixes-aux [*j*, *v*, *i*] ↓=
e-snoc v (*if eval r-s* [*e-snoc v 0*] = *eval r-s* [*v*] *then 1 else 0*)
unfolding *r-prefixes-aux-def* **by** *auto*

definition *r-prefixes* ≡ *r-swap* (*Pr 1 r-singleton-encode r-prefixes-aux*)

lemma *r-prefixes-recfn*: *recfn 2 r-prefixes*
unfolding *r-prefixes-def r-prefixes-aux-def* **by** *simp*

lemma *r-prefixes*: *eval r-prefixes* [*i*, *n*] ↓= *list-encode* (*prefixes i n*)
proof −
let *?h* = *Pr 1 r-singleton-encode r-prefixes-aux*
have *eval ?h* [*n*, *i*] ↓= *list-encode* (*prefixes i n*)
proof (*induction n*)
case *0*
then show *?case*
using *r-prefixes-def r-prefixes-aux-recfn r-singleton-encode* **by** *simp*
next
case (*Suc n*)
then show *?case*
using *r-prefixes-aux-recfn r-prefixes-aux eval-r-s*
by *auto metis+*
qed
moreover have *eval ?h* [*n*, *i*] = *eval r-prefixes* [*i*, *n*] **for** *i n*
unfolding *r-prefixes-def* **by** (*simp add*: *r-prefixes-aux-recfn*)
ultimately show *?thesis* **by** *simp*
qed

lemma *prefixes-neq-nil*: *length* (*prefixes i x*) > *0*
by (*induction x*) *auto*

The actual numbering can then be defined via *prefixes*.

definition *psi* :: *partial2* (‹ψ›) **where**
ψ *i x* ≡ *Some* (*last* (*prefixes i x*))

lemma *psi-in-R2*: $\psi \in \mathcal{R}^2$
proof
define *r-psi* **where** *r-psi* ≡ *Cn 2 r-last* [*r-prefixes*]
have *recfn 2 r-psi*
unfolding *r-psi-def* **by** (*simp add*: *r-prefixes-recfn*)
then have *eval r-psi* [*i*, *n*] ↓= *last* (*prefixes i n*) **for** *n i*
unfolding *r-psi-def* **using** *r-prefixes r-prefixes-recfn prefixes-neq-nil* **by** *simp*
then have (λ*i x*. *Some* (*last* (*prefixes i x*))) $\in \mathcal{P}^2$
using ‹*recfn 2 r-psi*› *P2I*[*of r-psi*] **by** *simp*
with *psi-def* **show** $\psi \in \mathcal{P}^2$ **by** *presburger*
moreover show *total2 psi*
unfolding *psi-def* **by** *auto*
qed

lemma *psi-0-or-1*:
assumes *n* > *0*
shows ψ *i n* ↓= *0* ∨ ψ *i n* ↓= *1*
proof −

from *assms* **obtain** *m* **where** *n* = *Suc m*
using *gr0-implies-Suc* **by** *blast*
then have *last* (*prefixes i* (*Suc m*)) = *0* ∨ *last* (*prefixes i* (*Suc m*)) = *1*
by *simp*
then show *?thesis* **using** ‹*n* = *Suc m*› *psi-def* **by** *simp*
qed

The function *prefixes* does indeed provide the prefixes for ψ.

lemma *psi-init*: (ψ *i*) ▷ *x* = *list-encode* (*prefixes i x*)
proof −
have *prefix* (ψ *i*) *x* = *prefixes i x*
unfolding *psi-def*
by (*induction x*) (*simp-all add*: *prefix-0 prefix-Suc*)
with *init-def* **show** *?thesis* **by** *simp*
qed

One of the functions ψ_i is in $U_{LIM-CONS}$.

lemma *ex-psi-in-U*: ∃ *j*. ψ *j* ∈ $U_{LIM-CONS}$
proof −
obtain *j* **where** *j*: *j* ≥ *2* ψ *j* = φ *j*
using *kleene-fixed-point*[*of* ψ] *psi-in-R2 R2-imp-P2* **by** *metis*
then have ψ *j* ∈ $\mathcal{P}$ **by** (*simp add*: *phi-in-P2*)
define *p* **where** *p* = (λ*x*. ψ *j* (*x* + *1*))
have *p* ∈ $\mathcal{R}_{01}$
proof −
from *p-def* ‹ψ *j* ∈ $\mathcal{P}$› *skip-P1* **have** *p* ∈ $\mathcal{P}$ **by** *blast*
from *psi-in-R2* **have** *total1* (ψ *j*) **by** *simp*
with *p-def* **have** *total1 p*
by (*simp add*: *total1-def*)
with *psi-0-or-1* **have** *p n* ↓= *0* ∨ *p n* ↓= *1* **for** *n*
using *psi-def p-def* **by** *simp*
then show *?thesis*
by (*simp add*: *RPred1-def P1-total-imp-R1* ‹*p* ∈ $\mathcal{P}$› ‹*total1 p*›)
qed
moreover have ψ *j* = [*j*] ⊙ *p*
proof
fix *x*
show ψ *j x* = ([*j*] ⊙ *p*) *x*
proof (*cases x* = *0*)
case *True*
then show *?thesis* **using** *psi-def psi-def prepend-at-less* **by** *simp*
next
case *False*
then show *?thesis* **using** *p-def* **by** *simp*
qed
qed
ultimately have ψ *j* ∈ $U_{LIM-CONS}$
using *j U-LIMCONS-def* **by** (*metis* (*mono-tags*, *lifting*) *append-Nil mem-Collect-eq*)
then show *?thesis* **by** *auto*
qed

The strategy fails to learn $U_{LIM-CONS}$ because it changes its hypothesis all the time on functions $\psi_j \in V_0$.

lemma *U-LIMCONS-not-learn-cons*: ¬ *learn-cons* φ $U_{LIM-CONS}$ *s*
proof

assume *learn*: *learn-cons* φ $U_{LIM-CONS}$ *s*
have *s* (*list-encode* (*vs* @ [*0*])) ≠ *s* (*list-encode* (*vs* @ [*1*])) **for** *vs*
proof −
obtain f_0 **where** *f0*: $f_0 \in U_{LIM-CONS}$ *prefix* f_0 (*length vs*) = *vs* @ [*0*]
using *U-LIMCONS-prefix-complete*[*of vs* @ [*0*]] **by** *auto*
obtain f_1 **where** *f1*: $f_1 \in U_{LIM-CONS}$ *prefix* f_1 (*length vs*) = *vs* @ [*1*]
using *U-LIMCONS-prefix-complete*[*of vs* @ [*1*]] **by** *auto*
have f_0 (*length vs*) ≠ f_1 (*length vs*)
using *f0 f1* **by** (*metis lessI nth-append-length prefix-nth zero-neq-one*)
moreover have φ (*the* (*s* (f_0 ▷ *length vs*))) (*length vs*) = f_0 (*length vs*)
using *learn-consE*(*3*)[*of* φ *U-LIMCONS s*, *OF learn*, *of* f_0 *length vs*, *OF f0*(*1*)]
by *simp*
moreover have φ (*the* (*s* (f_1 ▷ *length vs*))) (*length vs*) = f_1 (*length vs*)
using *learn-consE*(*3*)[*of* φ *U-LIMCONS s*, *OF learn*, *of* f_1 *length vs*, *OF f1*(*1*)]
by *simp*
ultimately have *the* (*s* (f_0 ▷ *length vs*)) ≠ *the* (*s* (f_1 ▷ *length vs*))
by *auto*
then have *s* (f_0 ▷ *length vs*) ≠ *s* (f_1 ▷ *length vs*)
by *auto*
with *f0*(*2*) *f1*(*2*) **show** *?thesis* **by** (*simp add*: *init-def*)
qed
then have *s* (*list-encode* (*vs* @ [*0*])) ≠ *s* (*list-encode vs*) ∨
s (*list-encode* (*vs* @ [*1*])) ≠ *s* (*list-encode vs*)
for *vs*
by *metis*
then have *s* (*list-encode* (*prefixes i* (*Suc x*))) ≠ *s* (*list-encode* (*prefixes i x*)) **for** *i x*
by *simp*
then have ¬ *learn-lim* φ {ψ *i*} *s* **for** *i*
using *psi-def psi-init always-hyp-change-not-Lim* **by** *simp*
then have ¬ *learn-lim* φ *U-LIMCONS s*
using *ex-psi-in-U learn-lim-closed-subseteq* **by** *blast*
then show *False*
using *learn learn-cons-def* **by** *simp*
qed

end

With the locale we can now show the second part of the main result:

lemma *U-LIMCONS-not-in-CONS*: $U_{LIM-CONS} \notin$ *CONS*
proof
assume $U_{LIM-CONS} \in$ *CONS*
then have $U_{LIM-CONS} \in$ *CONS-wrt* φ
by (*simp add*: *CONS-wrt-phi-eq-CONS*)
then obtain *almost-s* **where** *learn-cons* φ $U_{LIM-CONS}$ *almost-s*
using *CONS-wrt-def* **by** *auto*
then obtain *s* **where** *s*: *total1 s learn-cons* φ $U_{LIM-CONS}$ *s*
using *U-LIMCONS-prefix-complete U-prefix-complete-imp-total-strategy* **by** *blast*
then have $s \in \mathcal{R}$
using *learn-consE*(*1*) *P1-total-imp-R1* **by** *blast*
with *cons-lim-def* **interpret** *cons-lim s* **by** *simp*
show *False*
using *s*(*2*) *U-LIMCONS-not-learn-cons* **by** *simp*
qed

The main result of this section:

theorem *CONS-subset-Lim*: *CONS* ⊂ *LIM*

using *U-LIMCONS-in-Lim U-LIMCONS-not-in-CONS CONS-subseteq-Lim* **by** *auto*

end

2.8 Lemma R

theory *Lemma-R*
imports *Inductive-Inference-Basics*
begin

A common technique for constructing a class that cannot be learned is diagonalization against all strategies (see, for instance, Section 2.9). Similarly, the typical way of proving that a class cannot be learned is by assuming there is a strategy and deriving a contradiction. Both techniques are easier to carry out if one has to consider only *total* recursive strategies. This is not possible in general, since after all the definitions of the inference types admit strictly partial strategies. However, for many inference types one can show that for every strategy there is a total strategy with at least the same "learning power". Results to that effect are called Lemma R.

Lemma R comes in different strengths depending on how general the construction of the total recursive strategy is. CONS is the only inference type considered here for which not even a weak form of Lemma R holds.

2.8.1 Strong Lemma R for LIM, FIN, and BC

In its strong form Lemma R says that for any strategy S, there is a total strategy T that learns all classes S learns regardless of hypothesis space. The strategy T can be derived from S by a delayed simulation of S. More precisely, for input f^n, T simulates S for prefixes $f^0, f^1, \ldots, f^n$ for at most n steps. If S halts on none of the prefixes, T outputs an arbitrary hypothesis. Otherwise let $k \leq n$ be maximal such that S halts on f^k in at most n steps. Then T outputs $S(f^k)$.

We reformulate some lemmas for *r-result1* to make it easier to use them with φ.

lemma *r-result1-converg-phi*:
assumes $\varphi\ i\ x \downarrow= v$
shows $\exists\, t.$
$(\forall\, t' \geq t.\ eval\ r\text{-}result1\ [t', i, x] \downarrow= Suc\ v) \wedge$
$(\forall\, t' < t.\ eval\ r\text{-}result1\ [t', i, x] \downarrow= 0)$
using *assms r-result1-converg′ phi-def* **by** *simp-all*

lemma *r-result1-bivalent′*:
assumes $eval\ r\text{-}phi\ [i, x] \downarrow= v$
shows $eval\ r\text{-}result1\ [t, i, x] \downarrow= Suc\ v \vee eval\ r\text{-}result1\ [t, i, x] \downarrow= 0$
using *assms r-result1 r-result-bivalent′ r-phi′′* **by** *simp*

lemma *r-result1-bivalent-phi*:
assumes $\varphi\ i\ x \downarrow= v$
shows $eval\ r\text{-}result1\ [t, i, x] \downarrow= Suc\ v \vee eval\ r\text{-}result1\ [t, i, x] \downarrow= 0$
using *assms r-result1-bivalent′ phi-def* **by** *simp-all*

lemma *r-result1-diverg-phi*:
assumes $\varphi\ i\ x \uparrow$
shows $eval\ r\text{-}result1\ [t, i, x] \downarrow= 0$

using *assms phi-def r-result1-diverg'* **by** *simp*

lemma *r-result1-some-phi*:
assumes *eval r-result1* $[t, i, x] \downarrow= Suc\ v$
shows $\varphi\ i\ x \downarrow= v$
using *assms phi-def r-result1-Some'* **by** *simp*

lemma *r-result1-saturating'*:
assumes *eval r-result1* $[t, i, x] \downarrow= Suc\ v$
shows *eval r-result1* $[t + d, i, x] \downarrow= Suc\ v$
using *assms r-result1 r-result-saturating r-phi''* **by** *simp*

lemma *r-result1-saturating-the*:
assumes *the* (*eval r-result1* $[t, i, x]) > 0$ **and** $t' \geq t$
shows *the* (*eval r-result1* $[t', i, x]) > 0$
proof −
from *assms*(*1*) **obtain** *v* **where** *eval r-result1* $[t, i, x] \downarrow= Suc\ v$
using *r-result1-bivalent-phi r-result1-diverg-phi*
by (*metis inc-induct le-0-eq not-less-zero option.discI option.expand option.sel*)
with *assms* **have** *eval r-result1* $[t', i, x] \downarrow= Suc\ v$
using *r-result1-saturating' le-Suc-ex* **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *Greatest-bounded-Suc*:
fixes $P :: nat \Rightarrow nat$
shows (*if* $P\ n > 0$ *then Suc n*
else if $\exists j<n.\ P\ j > 0$ *then Suc* (*GREATEST* $j.\ j < n \wedge P\ j > 0$) *else 0*) =
(*if* $\exists j<Suc\ n.\ P\ j > 0$ *then Suc* (*GREATEST* $j.\ j < Suc\ n \wedge P\ j > 0$) *else 0*)
(**is** *?lhs = ?rhs*)
proof (*cases* $\exists j<Suc\ n.\ P\ j > 0$)
case *1*: *True*
show *?thesis*
proof (*cases* $P\ n > 0$)
case *True*
then have (*GREATEST* $j.\ j < Suc\ n \wedge P\ j > 0) = n$
using *Greatest-equality*[*of* $\lambda j.\ j < Suc\ n \wedge P\ j > 0$] **by** *simp*
moreover have *?rhs = Suc* (*GREATEST* $j.\ j < Suc\ n \wedge P\ j > 0$)
using *1* **by** *simp*
ultimately have *?rhs = Suc n* **by** *simp*
then show *?thesis* **using** *True* **by** *simp*
next
case *False*
then have *?lhs = Suc* (*GREATEST* $j.\ j < n \wedge P\ j > 0$)
using *1* **by** (*metis less-SucE*)
moreover have *?rhs = Suc* (*GREATEST* $j.\ j < Suc\ n \wedge P\ j > 0$)
using *1* **by** *simp*
moreover have (*GREATEST* $j.\ j < n \wedge P\ j > 0$) =
(*GREATEST* $j.\ j < Suc\ n \wedge P\ j > 0$)
using *1 False* **by** (*metis less-SucI less-Suc-eq*)
ultimately show *?thesis* **by** *simp*
qed
next
case *False*
then show *?thesis* **by** *auto*
qed

For n, i, x, the next function simulates φ_i on all non-empty prefixes of at most length n of the list x for at most n steps. It returns the length of the longest such prefix for which φ_i halts, or zero if φ_i does not halt for any prefix.

```
definition r-delay-aux ≡
  Pr 2 (r-constn 1 0)
    (Cn 4 r-ifz
      [Cn 4 r-result1
        [Cn 4 r-length [Id 4 3], Id 4 2,
         Cn 4 r-take [Cn 4 S [Id 4 0], Id 4 3]],
       Id 4 1, Cn 4 S [Id 4 0]])

lemma r-delay-aux-prim: prim-recfn 3 r-delay-aux
  unfolding r-delay-aux-def by simp-all

lemma r-delay-aux-total: total r-delay-aux
  using prim-recfn-total[OF r-delay-aux-prim] .

lemma r-delay-aux:
  assumes n ≤ e-length x
  shows eval r-delay-aux [n, i, x] ↓=
   (if ∃j<n. the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0
    then Suc (GREATEST j.
                j < n ∧
                the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0)
    else 0)
proof −
  define z where z ≡
    Cn 4 r-result1
      [Cn 4 r-length [Id 4 3], Id 4 2, Cn 4 r-take [Cn 4 S [Id 4 0], Id 4 3]]
  then have z-recfn: recfn 4 z by simp
  have z: eval z [j, r, i, x] = eval r-result1 [e-length x, i, e-take (Suc j) x]
      if j < e-length x for j r i x
    unfolding z-def using that by simp

  define g where g ≡ Cn 4 r-ifz [z, Id 4 1, Cn 4 S [Id 4 0]]
  then have g: eval g [j, r, i, x] ↓=
      (if the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0 then Suc j else r)
      if j < e-length x for j r i x
    using that z prim-recfn-total z-recfn by simp

  show ?thesis
    using assms
  proof (induction n)
    case 0
    moreover have eval r-delay-aux [0, i, x] ↓= 0
      using eval-Pr-0 r-delay-aux-def r-delay-aux-prim r-constn
      by (simp add: r-delay-aux-def)
    ultimately show ?case by simp
  next
    case (Suc n)
    let ?P = λj. the (eval r-result1 [e-length x, i, e-take (Suc j) x])
    have eval r-delay-aux [n, i, x] ↓
      using Suc by simp
    moreover have eval r-delay-aux [Suc n, i, x] =
        eval (Pr 2 (r-constn 1 0) g) [Suc n, i, x]
```

```
      unfolding r-delay-aux-def g-def z-def by simp
    ultimately have eval r-delay-aux [Suc n, i, x] =
        eval g [n, the (eval r-delay-aux [n, i, x]), i, x]
      using r-delay-aux-prim Suc eval-Pr-converg-Suc
      by (simp add: r-delay-aux-def g-def z-def numeral-3-eq-3)
    then have eval r-delay-aux [Suc n, i, x] ↓=
        (if ?P n > 0 then Suc n
         else if ∃j<n. ?P j > 0 then Suc (GREATEST j. j < n ∧ ?P j > 0) else 0)
      using g Suc by simp
    then have eval r-delay-aux [Suc n, i, x] ↓=
        (if ∃j<Suc n. ?P j > 0 then Suc (GREATEST j. j < Suc n ∧ ?P j > 0) else 0)
      using Greatest-bounded-Suc[where ?P=?P] by simp
    then show ?case by simp
  qed
qed
```

The next function simulates φ_i on all non-empty prefixes of a list x of length n for at most n steps and outputs the length of the longest prefix for which φ_i halts, or zero if φ_i does not halt for any such prefix.

```
definition r-delay ≡ Cn 2 r-delay-aux [Cn 2 r-length [Id 2 1], Id 2 0, Id 2 1]

lemma r-delay-recfn [simp]: recfn 2 r-delay
  unfolding r-delay-def by (simp add: r-delay-aux-prim)

lemma r-delay:
  eval r-delay [i, x] ↓=
    (if ∃j<e-length x. the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0
     then Suc (GREATEST j.
        j < e-length x ∧ the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0)
     else 0)
  unfolding r-delay-def using r-delay-aux r-delay-aux-prim by simp

definition delay i x ≡ Some
 (if ∃j<e-length x. the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0
  then Suc (GREATEST j.
    j < e-length x ∧ the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0)
  else 0)

lemma delay-in-R2: delay ∈ ℛ²
  using r-delay totalI2 R2I delay-def r-delay-recfn
  by (metis (no-types, lifting) numeral-2-eq-2 option.simps(3))

lemma delay-le-length: the (delay i x) ≤ e-length x
proof (cases ∃j<e-length x. the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0)
  case True
  let ?P = λj. j < e-length x ∧ the (eval r-result1 [e-length x, i, e-take (Suc j) x]) > 0
  from True have ∃j. ?P j by simp
  moreover have ⋀y. ?P y ⟹ y ≤ e-length x by simp
  ultimately have ?P (Greatest ?P)
    using GreatestI-ex-nat[where ?P=?P] by blast
  then have Greatest ?P < e-length x by simp
  moreover have delay i x ↓= Suc (Greatest ?P)
    using delay-def True by simp
  ultimately show ?thesis by auto
next
  case False
```

then show *?thesis* **using** *delay-def* **by** *auto*
qed

lemma *e-take-delay-init*:
assumes $f \in \mathcal{R}$ **and** *the* (*delay i* ($f \triangleright n$)) > *0*
shows *e-take* (*the* (*delay i* ($f \triangleright n$))) ($f \triangleright n$) = $f \triangleright$ (*the* (*delay i* ($f \triangleright n$)) − *1*)
using *assms e-take-init*[*of f - n*] *length-init*[*of f n*] *delay-le-length*[*of i f* $\triangleright$ *n*]
by (*metis One-nat-def Suc-le-lessD Suc-pred*)

lemma *delay-gr0-converg*:
assumes *the* (*delay i x*) > *0*
shows φ *i* (*e-take* (*the* (*delay i x*)) *x*) ↓
proof −
let *?P* = λ*j*. *j* < *e-length x* ∧ *the* (*eval r-result1* [*e-length x*, *i*, *e-take* (*Suc j*) *x*]) > *0*
have ∃ *j*. *?P j*
proof (*rule ccontr*)
assume ¬ (∃ *j*. *?P j*)
then have *delay i x* ↓= *0*
using *delay-def* **by** *simp*
with *assms* **show** *False* **by** *simp*
qed
then have *d*: *the* (*delay i x*) = *Suc* (*Greatest ?P*)
using *delay-def* **by** *simp*
moreover have ⋀*y*. *?P y* ⟹ *y* ≤ *e-length x* **by** *simp*
ultimately have *?P* (*Greatest ?P*)
using ‹∃ *j*. *?P j*› *GreatestI-ex-nat*[**where** *?P*=*?P*] **by** *blast*
then have *the* (*eval r-result1* [*e-length x*, *i*, *e-take* (*Suc* (*Greatest ?P*)) *x*]) > *0*
by *simp*
then have *the* (*eval r-result1* [*e-length x*, *i*, *e-take* (*the* (*delay i x*)) *x*]) > *0*
using *d* **by** *simp*
then show *?thesis* **using** *r-result1-diverg-phi* **by** *fastforce*
qed

lemma *delay-unbounded*:
fixes *n* :: *nat*
assumes $f \in \mathcal{R}$ **and** ∀ *n*. φ *i* ($f \triangleright n$) ↓
shows ∃ *m*. *the* (*delay i* ($f \triangleright m$)) > *n*
proof −
from *assms* **have** ∃ *t*. *the* (*eval r-result1* [*t*, *i*, $f \triangleright n$]) > *0*
using *r-result1-converg-phi*
by (*metis le-refl option.exhaust-sel option.sel zero-less-Suc*)
then obtain *t* **where** *t*: *the* (*eval r-result1* [*t*, *i*, $f \triangleright n$]) > *0*
by *auto*
let *?m* = *max n t*
have *Suc ?m* ≥ *t* **by** *simp*
have *m*: *the* (*eval r-result1* [*Suc ?m*, *i*, $f \triangleright n$]) > *0*
proof −
let *?w* = *eval r-result1* [*t*, *i*, $f \triangleright n$]
obtain *v* **where** *v*: *?w* ↓= *Suc v*
using *t assms*(*2*) *r-result1-bivalent-phi* **by** *fastforce*
have *eval r-result1* [*Suc ?m*, *i*, $f \triangleright n$] = *?w*
using *v t r-result1-saturating'* ‹*Suc ?m* ≥ *t*› *le-Suc-ex* **by** *fastforce*
then show *?thesis* **using** *t* **by** *simp*
qed
let *?x* = $f \triangleright$ *?m*
have *the* (*delay i ?x*) > *n*

```
  proof −
    let ?P = λj. j < e-length ?x ∧ the (eval r-result1 [e-length ?x, i, e-take (Suc j) ?x]) > 0
    have e-length ?x = Suc ?m by simp
    moreover have e-take (Suc n) ?x = f ▹ n
      using assms(1) e-take-init by auto
    ultimately have ?P n
      using m by simp
    have ⋀y. ?P y ⟹ y ≤ e-length ?x by simp
    with ‹?P n› have n ≤ (Greatest ?P)
      using Greatest-le-nat[of ?P n e-length ?x] by simp
    moreover have the (delay i ?x) = Suc (Greatest ?P)
      using delay-def ‹?P n› by auto
    ultimately show ?thesis by simp
  qed
  then show ?thesis by auto
qed

lemma delay-monotone:
  assumes f ∈ ℛ and n₁ ≤ n₂
  shows the (delay i (f ▹ n₁)) ≤ the (delay i (f ▹ n₂))
    (is the (delay i ?x1) ≤ the (delay i ?x2))
proof (cases the (delay i (f ▹ n₁)) = 0)
  case True
  then show ?thesis by simp
next
  case False
  let ?P1 = λj. j < e-length ?x1 ∧ the (eval r-result1 [e-length ?x1, i, e-take (Suc j) ?x1]) > 0
  let ?P2 = λj. j < e-length ?x2 ∧ the (eval r-result1 [e-length ?x2, i, e-take (Suc j) ?x2]) > 0
  from False have d1: the (delay i ?x1) = Suc (Greatest ?P1) ∃j. ?P1 j
    using delay-def option.collapse by fastforce+
  moreover have ⋀y. ?P1 y ⟹ y ≤ e-length ?x1 by simp
  ultimately have ∗: ?P1 (Greatest ?P1) using GreatestI-ex-nat[of ?P1] by blast
  let ?j = Greatest ?P1
  from ∗ have ?j < e-length ?x1 by auto
  then have 1: e-take (Suc ?j) ?x1 = e-take (Suc ?j) ?x2
    using assms e-take-init by auto
  from ∗ have 2: ?j < e-length ?x2 using assms(2) by auto
  with 1 ∗ have the (eval r-result1 [e-length ?x1, i, e-take (Suc ?j) ?x2]) > 0
    by simp
  moreover have e-length ?x1 ≤ e-length ?x2
    using assms(2) by auto
  ultimately have the (eval r-result1 [e-length ?x2, i, e-take (Suc ?j) ?x2]) > 0
    using r-result1-saturating-the by simp
  with 2 have ?P2 ?j by simp
  then have d2: the (delay i ?x2) = Suc (Greatest ?P2)
    using delay-def by auto
  have ⋀y. ?P2 y ⟹ y ≤ e-length ?x2 by simp
  with ‹?P2 ?j› have ?j ≤ (Greatest ?P2) using Greatest-le-nat[of ?P2] by blast
  with d1 d2 show ?thesis by simp
qed

lemma delay-unbounded-monotone:
  fixes n :: nat
  assumes f ∈ ℛ and ∀n. φ i (f ▹ n) ↓
  shows ∃m₀. ∀m≥m₀. the (delay i (f ▹ m)) > n
proof −
```

from *assms delay-unbounded* **obtain** m_0 **where** *the* (*delay i* ($f \triangleright m_0$)) > *n*
  by *blast*
then have $\forall\, m \geq m_0$. *the* (*delay i* ($f \triangleright m$)) > *n*
  using *assms*(*1*) *delay-monotone order.strict-trans2* **by** *blast*
then show *?thesis* **by** *auto*
qed

Now we can define a function that simulates an arbitrary strategy φ_i in a delayed way. The parameter d is the default hypothesis for when φ_i does not halt within the time bound for any prefix.

definition *r-totalizer* :: *nat* ⇒ *recf* **where**
r-totalizer d ≡
Cn 2
(*r-lifz*
(*r-constn 1 d*)
(*Cn 2 r-phi*
[*Id 2 0*, *Cn 2 r-take* [*Cn 2 r-delay* [*Id 2 0*, *Id 2 1*], *Id 2 1*]]))
[*Cn 2 r-delay* [*Id 2 0*, *Id 2 1*], *Id 2 0*, *Id 2 1*]

lemma *r-totalizer-recfn*: *recfn 2* (*r-totalizer d*)
unfolding *r-totalizer-def* **by** *simp*

lemma *r-totalizer*:
eval (*r-totalizer d*) [*i*, *x*] =
(*if the* (*delay i x*) = *0 then Some d else* φ *i* (*e-take* (*the* (*delay i x*)) *x*))
proof −
let *?i* = *Cn 2 r-delay* [*Id 2 0*, *Id 2 1*]
have *eval ?i* [*i*, *x*] = *eval r-delay* [*i*, *x*] **for** *i x*
using *r-delay-recfn* **by** *simp*
then have *i*: *eval ?i* [*i*, *x*] = *delay i x* **for** *i x*
using *r-delay* **by** (*simp add*: *delay-def*)
let *?t* = *r-constn 1 d*
have *t*: *eval ?t* [*i*, *x*] ↓= *d* **for** *i x* **by** *simp*
let *?e1* = *Cn 2 r-take* [*?i*, *Id 2 1*]
let *?e* = *Cn 2 r-phi* [*Id 2 0*, *?e1*]
have *eval ?e1* [*i*, *x*] = *eval r-take* [*the* (*delay i x*), *x*] **for** *i x*
using *r-delay i delay-def* **by** *simp*
then have *eval ?e1* [*i*, *x*] ↓= *e-take* (*the* (*delay i x*)) *x* **for** *i x*
using *delay-le-length* **by** *simp*
then have *e*: *eval ?e* [*i*, *x*] = φ *i* (*e-take* (*the* (*delay i x*)) *x*)
using *phi-def* **by** *simp*
let *?z* = *r-lifz ?t ?e*
have *recfn-te*: *recfn 2 ?t recfn 2 ?e*
by *simp-all*
then have *eval* (*r-totalizer d*) [*i*, *x*] = *eval* (*r-lifz ?t ?e*) [*the* (*delay i x*), *i*, *x*]
for *i x*
unfolding *r-totalizer-def* **using** *i r-totalizer-recfn delay-def* **by** *simp*
then have *eval* (*r-totalizer d*) [*i*, *x*] =
(*if the* (*delay i x*) = *0 then eval ?t* [*i*, *x*] *else eval ?e* [*i*, *x*])
for *i x*
using *recfn-te* **by** *simp*
then show *?thesis* **using** *t e* **by** *simp*
qed

lemma *r-totalizer-total*: *total* (*r-totalizer d*)
proof (*rule totalI2*)

```
  show recfn 2 (r-totalizer d) using r-totalizer-recfn by simp
  show ⋀x y. eval (r-totalizer d) [x, y] ↓
    using r-totalizer delay-gr0-converg by simp
qed

definition totalizer :: nat ⇒ partial2 where
  totalizer d i x ≡
     if the (delay i x) = 0 then Some d else φ i (e-take (the (delay i x)) x)

lemma totalizer-init:
  assumes f ∈ ℛ
  shows totalizer d i (f ▹ n) =
    (if the (delay i (f ▹ n)) = 0 then Some d
     else φ i (f ▹ (the (delay i (f ▹ n)) − 1)))
  using assms e-take-delay-init by (simp add: totalizer-def)

lemma totalizer-in-R2: totalizer d ∈ ℛ^2
  using totalizer-def r-totalizer r-totalizer-total R2I r-totalizer-recfn
  by metis
```

For LIM, *totalizer* works with every default hypothesis d.

```
lemma lemma-R-for-Lim:
  assumes learn-lim ψ U (φ i)
  shows learn-lim ψ U (totalizer d i)
proof (rule learn-limI)
  show env: environment ψ U (totalizer d i)
    using assms learn-limE(1) totalizer-in-R2 by auto
  show ∃j. ψ j = f ∧ (∀^∞n. totalizer d i (f ▹ n) ↓= j) if f ∈ U for f
  proof −
    have f ∈ ℛ
      using assms env that by auto
    from assms learn-limE obtain j n_0 where
      j: ψ j = f and
      n0: ∀n≥n_0. (φ i) (f ▹ n) ↓= j
      using ‹f ∈ U› by metis
    obtain m_0 where m0: ∀m≥m_0. the (delay i (f ▹ m)) > n_0
      using delay-unbounded-monotone ‹f ∈ ℛ› ‹f ∈ U› assms learn-limE(1)
      by blast
    then have ∀m≥m_0. totalizer d i (f ▹ m) = φ i (e-take (the (delay i (f ▹ m))) (f ▹ m))
      using totalizer-def by auto
    then have ∀m≥m_0. totalizer d i (f ▹ m) = φ i (f ▹ (the (delay i (f ▹ m)) − 1))
      using e-take-delay-init m0 ‹f ∈ ℛ› by auto
    with m0 n0 have ∀m≥m_0. totalizer d i (f ▹ m) ↓= j
      by auto
    with j show ?thesis by auto
  qed
qed
```

The effective version of Lemma R for LIM states that there is a total recursive function computing Gödel numbers of total strategies from those of arbitrary strategies.

```
lemma lemma-R-for-Lim-effective:
  ∃g∈ℛ. ∀i.
     φ (the (g i)) ∈ ℛ ∧
     (∀U ψ. learn-lim ψ U (φ i) ⟶ learn-lim ψ U (φ (the (g i))))
proof −
```

have *totalizer 0* $\in \mathcal{P}^2$ **using** *totalizer-in-R2* **by** *auto*
then obtain *g* **where** *g*: $g \in \mathcal{R}$ $\forall i.$ (*totalizer 0*) $i = \varphi$ (*the* (*g i*))
using *numbering-translation-for-phi* **by** *blast*
with *totalizer-in-R2* **have** $\forall i.$ φ (*the* (*g i*)) $\in \mathcal{R}$
by (*metis R2-proj-R1*)
moreover from *g*(*2*) *lemma-R-for-Lim*[**where** *?d=0*] **have**
$\forall$ *i U* ψ. *learn-lim* ψ *U* (φ *i*) $\longrightarrow$ *learn-lim* ψ *U* (φ (*the* (*g i*)))
by *simp*
ultimately show *?thesis* **using** *g*(*1*) **by** *blast*
qed

In order for us to use the previous lemma, we need a function that performs the actual computation:

definition *r-limr* $\equiv$
SOME g.
recfn 1 g $\wedge$
total g $\wedge$
($\forall i.$ φ (*the* (*eval g* [*i*])) $\in \mathcal{R}$ $\wedge$
($\forall$ *U* ψ. *learn-lim* ψ *U* (φ *i*) $\longrightarrow$ *learn-lim* ψ *U* (φ (*the* (*eval g* [*i*])))))

lemma *r-limr-recfn*: *recfn 1 r-limr*
and *r-limr-total*: *total r-limr*
and *r-limr*:
φ (*the* (*eval r-limr* [*i*])) $\in \mathcal{R}$
learn-lim ψ *U* (φ *i*) $\Longrightarrow$ *learn-lim* ψ *U* (φ (*the* (*eval r-limr* [*i*])))
proof −
let *?P* = λg.
$g \in \mathcal{R}$ $\wedge$
($\forall i.$ φ (*the* (*g i*)) $\in \mathcal{R}$ $\wedge$ ($\forall$ *U* ψ. *learn-lim* ψ *U* (φ *i*) $\longrightarrow$ *learn-lim* ψ *U* (φ (*the* (*g i*)))))
let *?Q* = λg.
recfn 1 g $\wedge$
total g $\wedge$
($\forall i.$ φ (*the* (*eval g* [*i*])) $\in \mathcal{R}$ $\wedge$
($\forall$ *U* ψ. *learn-lim* ψ *U* (φ *i*) $\longrightarrow$ *learn-lim* ψ *U* (φ (*the* (*eval g* [*i*])))))
have $\exists g$. *?P g* **using** *lemma-R-for-Lim-effective* **by** *auto*
then obtain *g* **where** *?P g* **by** *auto*
then obtain *g'* **where** *g'*: *recfn 1 g' total g'* $\forall i.$ *eval g'* [*i*] = *g i*
by *blast*
with ‹*?P g*› **have** *?Q g'* **by** *simp*
with *r-limr-def someI-ex*[*of ?Q*] **show**
recfn 1 r-limr
total r-limr
φ (*the* (*eval r-limr* [*i*])) $\in \mathcal{R}$
learn-lim ψ *U* (φ *i*) $\Longrightarrow$ *learn-lim* ψ *U* (φ (*the* (*eval r-limr* [*i*])))
by *auto*
qed

For BC, too, *totalizer* works with every default hypothesis *d*.

lemma *lemma-R-for-BC*:
assumes *learn-bc* ψ *U* (φ *i*)
shows *learn-bc* ψ *U* (*totalizer d i*)
proof (*rule learn-bcI*)
show *env*: *environment* ψ *U* (*totalizer d i*)
using *assms learn-bcE*(*1*) *totalizer-in-R2* **by** *auto*
show $\exists n_0. \forall n{\ge}n_0.$ ψ (*the* (*totalizer d i* ($f \triangleright n$))) = *f* **if** $f \in U$ **for** *f*

proof –
have $f \in \mathcal{R}$
using *assms env that* **by** *auto*
obtain n_0 **where** *n0*: $\forall n \geq n_0.\ \psi\ (the\ ((\varphi\ i)\ (f \triangleright n))) = f$
using *assms learn-bcE* ‹$f \in U$› **by** *metis*
obtain m_0 **where** *m0*: $\forall m \geq m_0.\ the\ (delay\ i\ (f \triangleright m)) > n_0$
using *delay-unbounded-monotone* ‹$f \in \mathcal{R}$› ‹$f \in U$› *assms learn-bcE(1)*
by *blast*
then have $\forall m \geq m_0.\ totalizer\ d\ i\ (f \triangleright m) = \varphi\ i\ (e\text{-}take\ (the\ (delay\ i\ (f \triangleright m)))\ (f \triangleright m))$
using *totalizer-def* **by** *auto*
then have $\forall m \geq m_0.\ totalizer\ d\ i\ (f \triangleright m) = \varphi\ i\ (f \triangleright (the\ (delay\ i\ (f \triangleright m)) - 1))$
using *e-take-delay-init m0* ‹$f \in \mathcal{R}$› **by** *auto*
with *m0 n0* **have** $\forall m \geq m_0.\ \psi\ (the\ (totalizer\ d\ i\ (f \triangleright m))) = f$
by *auto*
then show *?thesis* **by** *auto*
qed
qed

corollary *lemma-R-for-BC-simple*:
assumes *learn-bc* ψ *U s*
shows $\exists s' \in \mathcal{R}.$ *learn-bc* ψ *U* s'
using *assms lemma-R-for-BC totalizer-in-R2 learn-bcE*
by (*metis R2-proj-R1 learn-bcE(1) phi-universal*)

For FIN the default hypothesis of *totalizer* must be zero, signalling "don't know yet".

lemma *lemma-R-for-FIN*:
assumes *learn-fin* ψ *U* $(\varphi\ i)$
shows *learn-fin* ψ *U* $(totalizer\ 0\ i)$
proof (*rule learn-finI*)
show *env*: *environment* ψ *U* $(totalizer\ 0\ i)$
using *assms learn-finE(1) totalizer-in-R2* **by** *auto*
show $\exists j\ n_0.\ \psi\ j = f\ \wedge$
$(\forall n < n_0.\ totalizer\ 0\ i\ (f \triangleright n) \downarrow= 0)\ \wedge$
$(\forall n \geq n_0.\ totalizer\ 0\ i\ (f \triangleright n) \downarrow= Suc\ j)$
if $f \in U$ **for** f
proof –
have $f \in \mathcal{R}$
using *assms env that* **by** *auto*
from *assms learn-finE*[*of* ψ *U* φ *i*] **obtain** j **where**
j: $\psi\ j = f$ **and**
ex-n0: $\exists n_0.\ (\forall n < n_0.\ (\varphi\ i)\ (f \triangleright n) \downarrow= 0) \wedge (\forall n \geq n_0.\ (\varphi\ i)\ (f \triangleright n) \downarrow= Suc\ j)$
using ‹$f \in U$› **by** *blast*
let $?Q = \lambda n_0.\ (\forall n < n_0.\ (\varphi\ i)\ (f \triangleright n) \downarrow= 0) \wedge (\forall n \geq n_0.\ (\varphi\ i)\ (f \triangleright n) \downarrow= Suc\ j)$
define n_0 **where** $n_0 = Least\ ?Q$
with *ex-n0* **have** *n0*: $?Q\ n_0\ \forall n < n_0.\ \neg\ ?Q\ n$
using *LeastI-ex*[*of ?Q*] *not-less-Least*[*of - ?Q*] **by** *blast+*
define m_0 **where** $m_0 = (LEAST\ m_0.\ \forall m \geq m_0.\ the\ (delay\ i\ (f \triangleright m)) > n_0)$
(**is** $m_0 = Least\ ?P$)
moreover have $\exists m_0.\ \forall m \geq m_0.\ the\ (delay\ i\ (f \triangleright m)) > n_0$
using *delay-unbounded-monotone* ‹$f \in \mathcal{R}$› ‹$f \in U$› *assms learn-finE(1)*
by *simp*
ultimately have *m0*: $?P\ m_0\ \forall m < m_0.\ \neg\ ?P\ m$
using *LeastI-ex*[*of ?P*] *not-less-Least*[*of - ?P*] **by** *blast+*
then have $\forall m \geq m_0.\ totalizer\ 0\ i\ (f \triangleright m) = \varphi\ i\ (e\text{-}take\ (the\ (delay\ i\ (f \triangleright m)))\ (f \triangleright m))$
using *totalizer-def* **by** *auto*
then have $\forall m \geq m_0.\ totalizer\ 0\ i\ (f \triangleright m) = \varphi\ i\ (f \triangleright (the\ (delay\ i\ (f \triangleright m)) - 1))$

```
    using e-take-delay-init m0 ‹f∈ℛ› by auto
  with m0 n0 have ∀ m≥m₀. totalizer 0 i (f ▹ m) ↓= Suc j
    by auto
  moreover have totalizer 0 i (f ▹ m) ↓= 0 if m < m₀ for m
  proof (cases the (delay i (f ▹ m)) = 0)
    case True
    then show ?thesis by (simp add: totalizer-def)
  next
    case False
    then have the (delay i (f ▹ m)) ≤ n₀
      using m0 that ‹f ∈ ℛ› delay-monotone by (meson leI order.strict-trans2)
    then show ?thesis
      using ‹f ∈ ℛ› n0(1) totalizer-init by (simp add: Suc-le-lessD)
  qed
  ultimately show ?thesis using j by auto
 qed
qed
```

2.8.2 Weaker Lemma R for CP and TOTAL

For TOTAL the default hypothesis used by *totalizer* depends on the hypothesis space, because it must refer to a total function in that space. Consequently the total strategy depends on the hypothesis space, which makes this form of Lemma R weaker than the ones in the previous section.

```
lemma lemma-R-for-TOTAL:
 fixes ψ :: partial2
 shows ∃ d. ∀ U. ∀ i. learn-total ψ U (φ i) ⟶ learn-total ψ U (totalizer d i)
proof (cases ∃ d. ψ d ∈ ℛ)
 case True
 then obtain d where ψ d ∈ ℛ by auto
 have learn-total ψ U (totalizer d i) if learn-total ψ U (φ i) for U i
 proof (rule learn-totalI)
  show env: environment ψ U (totalizer d i)
    using that learn-totalE(1) totalizer-in-R2 by auto
  show ⋀f. f ∈ U ⟹ ∃ j. ψ j = f ∧ (∀∞n. totalizer d i (f ▹ n) ↓= j)
    using that learn-total-def lemma-R-for-Lim[where ?d=d] learn-limE(2) by metis
  show ψ (the (totalizer d i (f ▹ n))) ∈ ℛ if f ∈ U for f n
  proof (cases the (delay i (f ▹ n)) = 0)
    case True
    then show ?thesis using totalizer-def ‹ψ d ∈ ℛ› by simp
  next
    case False
    have f ∈ ℛ
      using that env by auto
    then show ?thesis
      using False that ‹learn-total ψ U (φ i)› totalizer-init learn-totalE(3)
      by simp
  qed
 qed
 then show ?thesis by auto
next
 case False
 then show ?thesis using learn-total-def lemma-R-for-Lim by auto
qed
```

```
corollary lemma-R-for-TOTAL-simple:
  assumes learn-total ψ U s
  shows ∃ s'∈ℛ. learn-total ψ U s'
  using assms lemma-R-for-TOTAL totalizer-in-R2
  by (metis R2-proj-R1 learn-totalE(1) phi-universal)
```

For CP the default hypothesis used by *totalizer* depends on both the hypothesis space and the class. Therefore the total strategy depends on both the the hypothesis space and the class, which makes Lemma R for CP even weaker than the one for TOTAL.

```
lemma lemma-R-for-CP:
  fixes ψ :: partial2 and U :: partial1 set
  assumes learn-cp ψ U (φ i)
  shows ∃ d. learn-cp ψ U (totalizer d i)
proof (cases U = {})
  case True
  then show ?thesis using assms learn-cp-def lemma-R-for-Lim by auto
next
  case False
  then obtain f where f ∈ U by auto
  from ‹f ∈ U› obtain d where ψ d = f
    using learn-cpE(2)[OF assms] by auto
  with ‹f ∈ U› have ψ d ∈ U by simp
  have learn-cp ψ U (totalizer d i)
  proof (rule learn-cpI)
    show env: environment ψ U (totalizer d i)
      using assms learn-cpE(1) totalizer-in-R2 by auto
    show ⋀f. f ∈ U ⟹ ∃ j. ψ j = f ∧ (∀∞n. totalizer d i (f ▹ n) ↓= j)
      using assms learn-cp-def lemma-R-for-Lim[where ?d=d] learn-limE(2) by metis
    show ψ (the (totalizer d i (f ▹ n))) ∈ U if f ∈ U for f n
    proof (cases the (delay i (f ▹ n)) = 0)
      case True
      then show ?thesis using totalizer-def ‹ψ d ∈ U› by simp
    next
      case False
      then show ?thesis
        using that env assms totalizer-init learn-cpE(3) by auto
    qed
  qed
  then show ?thesis by auto
qed
```

2.8.3 No Lemma R for CONS

This section demonstrates that the class V_{01} of all total recursive functions f where $f(0)$ or $f(1)$ is a Gödel number of f can be consistently learned in the limit, but not by a total strategy. This implies that Lemma R does not hold for CONS.

```
definition V01 :: partial1 set (‹V₀₁›) where
  V₀₁ = {f. f ∈ ℛ ∧ (φ (the (f 0)) = f ∨ φ (the (f 1)) = f)}
```

No total CONS strategy for V_{01}

In order to show that no total strategy can learn V_{01} we construct, for each total strategy S, one or two functions in V_{01} such that S fails for at least one of them. At the core of this construction is a process that given a total recursive strategy S and numbers $z, i, j \in \mathbb{N}$ builds a function f as follows: Set $f(0) = i$ and $f(1) = j$. For $x \geq 1$:

(a) Check whether S changes its hypothesis when f^x is extended by 0, that is, if $S(f^x) \neq S(f^x0)$. If so, set $f(x+1) = 0$.

(b) Otherwise check if S changes its hypothesis when f^x is extended by 1, that is, if $S(f^x) \neq S(f^x1)$. If so, set $f(x+1) = 1$.

(c) If neither happens, set $f(x+1) = z$.

In other words, as long as we can force S to change its hypothesis by extending the function by 0 or 1, we do just that. Now there are two cases:

Case 1. For all $x \geq 1$ either (a) or (b) occurs; then S changes its hypothesis on f all the time and thus does not learn f in the limit (not to mention consistently). The value of z makes no difference in this case.

Case 2. For some minimal x, (c) occurs, that is, there is an f^x such that $h := S(f^x) = S(f^x0) = S(f^x1)$. But the hypothesis h cannot be consistent with both prefixes f^x0 and f^x1. Running the process once with $z = 0$ and once with $z = 1$ yields two functions starting with f^x0 and f^x1, respectively, such that S outputs the same hypothesis, h, on both prefixes and thus cannot be consistent for both functions.

This process is computable because S is total. The construction does not work if we only assume S to be a CONS strategy for V_{01}, because we need to be able to apply S to prefixes not in V_{01}.

The parameters i and j provide flexibility to find functions built by the above process that are actually in V_{01}. To this end we will use Smullyan's double fixed-point theorem.

context
  fixes *s* :: *partial1*
  assumes *s-in-R1* [*simp*, *intro*]: $s \in \mathcal{R}$
begin

The function *prefixes* constructs prefixes according to the aforementioned process.

```
fun prefixes :: nat ⇒ nat ⇒ nat ⇒ nat ⇒ nat list where
  prefixes z i j 0 = [i]
| prefixes z i j (Suc x) = prefixes z i j x @
    [if x = 0 then j
     else if s (list-encode (prefixes z i j x @ [0])) ≠ s (list-encode (prefixes z i j x))
          then 0
          else if s (list-encode (prefixes z i j x @ [1])) ≠ s (list-encode (prefixes z i j x))
               then 1
               else z]
```

```
lemma prefixes-length: length (prefixes z i j x) = Suc x
  by (induction x) simp-all
```

The functions *adverse z i j* are the functions constructed by *prefixes*.

```
definition adverse :: nat ⇒ nat ⇒ nat ⇒ nat ⇒ nat option where
  adverse z i j x ≡ Some (last (prefixes z i j x))
```

```
lemma init-adverse-eq-prefixes: (adverse z i j) ▷ n = list-encode (prefixes z i j n)
```

```
proof -
  have prefix (adverse z i j) n = prefixes z i j n
  proof (induction n)
    case 0
    then show ?case using adverse-def prefixes-length prefixI' by fastforce
  next
    case (Suc n)
    then show ?case using adverse-def by (simp add: prefix-Suc)
  qed
  then show ?thesis by (simp add: init-def)
qed

lemma adverse-at-01:
  adverse z i j 0 ↓= i
  adverse z i j 1 ↓= j
  by (auto simp add: adverse-def)
```

Had we introduced ternary partial recursive functions, the *adverse z* functions would be among them.

```
lemma adverse-in-R3: ∃ r. recfn 3 r ∧ total r ∧ (λi j x. eval r [i, j, x]) = adverse z
proof -
  obtain rs where rs: recfn 1 rs total rs (λx. eval rs [x]) = s
    using R1E by auto
  have s-total: ⋀x. s x ↓ by simp

  define f where f = Cn 2 r-singleton-encode [Id 2 0]
  then have recfn 2 f by simp
  have f: ⋀i j. eval f [i, j] ↓= list-encode [i]
    unfolding f-def by simp

  define ch1 where ch1 = Cn 4 r-ifeq
    [Cn 4 rs [Cn 4 r-snoc [Id 4 1, r-constn 3 1]],
     Cn 4 rs [Id 4 1],
     r-dummy 3 (r-const z),
     r-constn 3 1]
  then have ch1: recfn 4 ch1 total ch1
    using Cn-total prim-recfn-total rs by auto

  define ch0 where ch0 = Cn 4 r-ifeq
    [Cn 4 rs [Cn 4 r-snoc [Id 4 1, r-constn 3 0]],
     Cn 4 rs [Id 4 1],
     ch1,
     r-constn 3 0]
  then have ch0-total: total ch0 recfn 4 ch0
    using Cn-total prim-recfn-total rs ch1 by auto

  have eval ch1 [l, v, i, j] ↓= (if s (e-snoc v 1) = s v then z else 1) for l v i j
  proof -
    have eval ch1 [l, v, i, j] = eval r-ifeq [the (s (e-snoc v 1)), the (s v), z, 1]
      unfolding ch1-def using rs by auto
    then show ?thesis by (simp add: s-total option.expand)
  qed
  moreover have eval ch0 [l, v, i, j] ↓=
    (if s (e-snoc v 0) = s v then the (eval ch1 [l, v, i, j]) else 0) for l v i j
  proof -
    have eval ch0 [l, v, i, j] =
```

```
    eval r-ifeq [the (s (e-snoc v 0)), the (s v), the (eval ch1 [l, v, i, j]), 0]
   unfolding ch0-def using rs ch1 by auto
  then show ?thesis by (simp add: s-total option.expand)
qed
ultimately have ch0: ⋀l v i j. eval ch0 [l, v, i, j] ↓=
  (if s (e-snoc v 0) ≠ s v then 0
   else if s (e-snoc v 1) ≠ s v then 1 else z)
  by simp

define app where app = Cn 4 r-ifz [Id 4 0, Id 4 3, ch0]
then have recfn 4 app total app
  using ch0-total totalI4 by auto
have eval app [l, v, i, j] ↓= (if l = 0 then j else the (eval ch0 [l, v, i, j])) for l v i j
  unfolding app-def using ch0-total by simp
with ch0 have app: ⋀l v i j. eval app [l, v, i, j] ↓=
  (if l = 0 then j
   else if s (e-snoc v 0) ≠ s v then 0
   else if s (e-snoc v 1) ≠ s v then 1 else z)
  by simp

define g where g = Cn 4 r-snoc [Id 4 1, app]
with app have g: ⋀l v i j. eval g [l, v, i, j] ↓= e-snoc v
  (if l = 0 then j
   else if s (e-snoc v 0) ≠ s v then 0
   else if s (e-snoc v 1) ≠ s v then 1 else z)
  using ‹recfn 4 app› by auto
from g-def have recfn 4 g total g
  using ‹recfn 4 app› ‹total app› Cn-total Mn-free-imp-total by auto

define b where b = Pr 2 f g
then have recfn 3 b
  using ‹recfn 2 f› ‹recfn 4 g› by simp
have b: eval b [x, i, j] ↓= list-encode (prefixes z i j x) for x i j
proof (induction x)
  case 0
  then show ?case
    unfolding b-def using f ‹recfn 2 f› ‹recfn 4 g› by simp
next
  case (Suc x)
  then have eval b [Suc x, i, j] = eval g [x, the (eval b [x, i, j]), i, j]
    using b-def ‹recfn 3 b› by simp
  also have ... ↓=
   (let v = list-encode (prefixes z i j x)
    in e-snoc v
      (if x = 0 then j
       else if s (e-snoc v 0) ≠ s v then 0
            else if s (e-snoc v 1) ≠ s v then 1 else z))
    using g Suc by simp
  also have ... ↓=
   (let v = list-encode (prefixes z i j x)
    in e-snoc v
      (if x = 0 then j
       else if s (list-encode (prefixes z i j x @ [0])) ≠ s v then 0
            else if s (list-encode (prefixes z i j x @ [1])) ≠ s v then 1 else z))
    using list-decode-encode by presburger
  finally show ?case by simp
```

```
  qed

  define b′ where b′ = Cn 3 b [Id 3 2, Id 3 0, Id 3 1]
  then have recfn 3 b′
    using ‹recfn 3 b› by simp
  with b have b′: ⋀i j x. eval b′ [i, j, x] ↓= list-encode (prefixes z i j x)
    using b′-def by simp

  define r where r = Cn 3 r-last [b′]
  then have recfn 3 r
    using ‹recfn 3 b′› by simp
  with b′ have ⋀i j x. eval r [i, j, x] ↓= last (prefixes z i j x)
    using r-def prefixes-length by auto
  moreover from this have total r
    using totalI3 ‹recfn 3 r› by simp
  ultimately have (λi j x. eval r [i, j, x]) = adverse z
    unfolding adverse-def by simp
  with ‹recfn 3 r› ‹total r› show ?thesis by auto
qed

lemma adverse-in-R1: adverse z i j ∈ ℛ
proof –
  from adverse-in-R3 obtain r where
    r: recfn 3 r total r (λi j x. eval r [i, j, x]) = adverse z
    by blast
  define rij where rij = Cn 1 r [r-const i, r-const j, Id 1 0]
  then have recfn 1 rij total rij
    using r(1,2) Cn-total Mn-free-imp-total by auto
  from rij-def have ⋀x. eval rij [x] = eval r [i, j, x]
    using r(1) by auto
  with r(3) have ⋀x. eval rij [x] = adverse z i j x
    by metis
  with ‹recfn 1 rij› ‹total rij› show ?thesis by auto
qed
```

Next we show that for every z there are i, j such that *adverse* $z\ i\ j \in V_{01}$. The first step is to show that for every z, Gödel numbers for *adverse* $z\ i\ j$ can be computed uniformly from i and j.

```
lemma phi-translate-adverse: ∃f∈ℛ².∀ i j. φ (the (f i j)) = adverse z i j
proof –
  obtain r where r: recfn 3 r total r (λi j x. eval r [i, j, x]) = adverse z
    using adverse-in-R3 by blast
  let ?p = encode r
  define rf where rf = Cn 2 (r-smn 1 2) [r-dummy 1 (r-const ?p), Id 2 0, Id 2 1]
  then have recfn 2 rf and total rf
    using Mn-free-imp-total by simp-all
  define f where f ≡ λi j. eval rf [i, j]
  with ‹recfn 2 rf› ‹total rf› have f ∈ ℛ² by auto
  have rf: eval rf [i, j] = eval (r-smn 1 2) [?p, i, j] for i j
    unfolding rf-def by simp
  {
    fix i j x
    have φ (the (f i j)) x = eval r-phi [the (f i j), x]
      using phi-def by simp
    also have ... = eval r-phi [the (eval rf [i, j]), x]
      using f-def by simp
```

also have ... = *eval* (*r-universal 1*) [*the* (*eval* (*r-smn 1 2*) [*?p*, *i*, *j*]), *x*]
using *rf r-phi-def* **by** *simp*
also have ... = *eval* (*r-universal* (*2* + *1*)) (*?p* # [*i*, *j*] @ [*x*])
using *smn-lemma*[*of 1* [*i*, *j*] *2* [*x*]] **by** *simp*
also have ... = *eval* (*r-universal 3*) [*?p*, *i*, *j*, *x*]
by *simp*
also have ... = *eval r* [*i*, *j*, *x*]
using *r-universal r* **by** *force*
also have ... = *adverse z i j x*
using *r*(*3*) **by** *metis*
finally have φ (*the* (*f i j*)) *x* = *adverse z i j x* **.**
}
with ‹$f \in \mathcal{R}^2$› **show** *?thesis* **by** *blast*
qed

The second, and final, step is to apply Smullyan's double fixed-point theorem to show the existence of *adverse* functions in V_{01}.

lemma *adverse-in-V01*: $\exists$ *m n*. *adverse 0 m n* $\in V_{01} \wedge$ *adverse 1 m n* $\in V_{01}$
proof −
obtain f_0 **where** *f0*: $f_0 \in \mathcal{R}^2$ $\forall$ *i j*. φ (*the* (f_0 *i j*)) = *adverse 0 i j*
using *phi-translate-adverse*[*of 0*] **by** *auto*
obtain f_1 **where** *f1*: $f_1 \in \mathcal{R}^2$ $\forall$ *i j*. φ (*the* (f_1 *i j*)) = *adverse 1 i j*
using *phi-translate-adverse*[*of 1*] **by** *auto*
obtain *m n* **where** φ *m* = φ (*the* (f_0 *m n*)) **and** φ *n* = φ (*the* (f_1 *m n*))
using *smullyan-double-fixed-point*[*OF f0*(*1*) *f1*(*1*)] **by** *blast*
with *f0*(*2*) *f1*(*2*) **have** φ *m* = *adverse 0 m n* **and** φ *n* = *adverse 1 m n*
by *simp-all*
moreover have *the* (*adverse 0 m n 0*) = *m* **and** *the* (*adverse 1 m n 1*) = *n*
using *adverse-at-01* **by** *simp-all*
ultimately have
φ (*the* (*adverse 0 m n 0*)) = *adverse 0 m n*
φ (*the* (*adverse 1 m n 1*)) = *adverse 1 m n*
by *simp-all*
moreover have *adverse 0 m n* $\in \mathcal{R}$ **and** *adverse 1 m n* $\in \mathcal{R}$
using *adverse-in-R1* **by** *simp-all*
ultimately show *?thesis* **using** *V01-def* **by** *auto*
qed

Before we prove the main result of this section we need some lemmas regarding the shape of the *adverse* functions and hypothesis changes of the strategy.

lemma *adverse-Suc*:
assumes *x* > *0*
shows *adverse z i j* (*Suc x*) ↓=
(*if s* (*e-snoc* ((*adverse z i j*) ▹ *x*) *0*) ≠ *s* ((*adverse z i j*) ▹ *x*)
then 0
else if s (*e-snoc* ((*adverse z i j*) ▹ *x*) *1*) ≠ *s* ((*adverse z i j*) ▹ *x*)
then 1 else z)
proof −
have *adverse z i j* (*Suc x*) ↓=
(*if s* (*list-encode* (*prefixes z i j x* @ [*0*])) ≠ *s* (*list-encode* (*prefixes z i j x*))
then 0
else if s (*list-encode* (*prefixes z i j x* @ [*1*])) ≠ *s* (*list-encode* (*prefixes z i j x*))
then 1 else z)
using *assms adverse-def* **by** *simp*
then show *?thesis* **by** (*simp add*: *init-adverse-eq-prefixes*)

```
qed
```

The process in the proof sketch (page 168) consists of steps (a), (b), and (c). The next abbreviation is true iff. step (a) or (b) applies.

```
abbreviation hyp-change z i j x ≡
  s (e-snoc ((adverse z i j) ▹ x) 0) ≠ s ((adverse z i j) ▹ x) ∨
  s (e-snoc ((adverse z i j) ▹ x) 1) ≠ s ((adverse z i j) ▹ x)
```

If step (c) applies, the process appends z.

```
lemma adverse-Suc-not-hyp-change:
  assumes x > 0 and ¬ hyp-change z i j x
  shows adverse z i j (Suc x) ↓= z
  using assms adverse-Suc by simp
```

While (a) or (b) applies, the process appends a value that forces S to change its hypothesis.

```
lemma while-hyp-change:
  assumes ∀ x≤n. x > 0 ⟶ hyp-change z i j x
  shows ∀ x≤Suc n. adverse z i j x = adverse z' i j x
  using assms
proof (induction n)
  case 0
  then show ?case by (simp add: adverse-def le-Suc-eq)
next
  case (Suc n)
  then have ∀ x≤n. x > 0 ⟶ hyp-change z i j x by simp
  with Suc have ∀ x≤Suc n. x > 0 ⟶ adverse z i j x = adverse z' i j x
    by simp
  moreover have adverse z i j 0 = adverse z' i j 0
    using adverse-at-01 by simp
  ultimately have zz': ∀ x≤Suc n. adverse z i j x = adverse z' i j x
    by auto
  moreover have adverse z i j ∈ ℛ adverse z' i j ∈ ℛ
    using adverse-in-R1 by simp-all
  ultimately have init-zz': (adverse z i j) ▹ (Suc n) = (adverse z' i j) ▹ (Suc n)
    using init-eqI by blast

  have adverse z i j (Suc (Suc n)) = adverse z' i j (Suc (Suc n))
  proof (cases s (e-snoc ((adverse z i j) ▹ (Suc n)) 0) ≠ s ((adverse z i j) ▹ (Suc n)))
    case True
    then have s (e-snoc ((adverse z' i j) ▹ (Suc n)) 0) ≠ s ((adverse z' i j) ▹ (Suc n))
      using init-zz' by simp
    then have adverse z' i j (Suc (Suc n)) ↓= 0
      by (simp add: adverse-Suc)
    moreover have adverse z i j (Suc (Suc n)) ↓= 0
      using True by (simp add: adverse-Suc)
    ultimately show ?thesis by simp
  next
    case False
    then have s (e-snoc ((adverse z' i j) ▹ (Suc n)) 0) = s ((adverse z' i j) ▹ (Suc n))
      using init-zz' by simp
    then have adverse z' i j (Suc (Suc n)) ↓= 1
      using init-zz' Suc.prems adverse-Suc by (smt le-refl zero-less-Suc)
    moreover have adverse z i j (Suc (Suc n)) ↓= 1
      using False Suc.prems adverse-Suc by auto
```

ultimately show *?thesis* **by** *simp*
qed
with zz' **show** *?case* **using** *le-SucE* **by** *blast*
qed

The next result corresponds to Case 1 from the proof sketch.

lemma *always-hyp-change-no-lim*:
assumes $\forall x{>}0.$ *hyp-change z i j x*
shows $\neg$ *learn-lim* φ {*adverse z i j*} *s*
proof (*rule infinite-hyp-changes-not-Lim*[*of adverse z i j*])
show *adverse z i j* $\in$ {*adverse z i j*} **by** *simp*
show $\forall n.\ \exists m_1{>}n.\ \exists m_2{>}n.\ s\ (adverse\ z\ i\ j \triangleright m_1) \neq s\ (adverse\ z\ i\ j \triangleright m_2)$
proof
fix *n*
from *assms* **obtain** m_1 **where** *m1*: $m_1 > n$ *hyp-change z i j* m_1
by *auto*
have $s\ (adverse\ z\ i\ j \triangleright m_1) \neq s\ (adverse\ z\ i\ j \triangleright (Suc\ m_1))$
proof (*cases* $s\ (e\text{-}snoc\ ((adverse\ z\ i\ j) \triangleright m_1)\ 0) \neq s\ ((adverse\ z\ i\ j) \triangleright m_1)$)
case *True*
then have *adverse z i j* (*Suc* m_1) $\downarrow= 0$
using *m1 adverse-Suc* **by** *simp*
then have $(adverse\ z\ i\ j) \triangleright (Suc\ m_1) = e\text{-}snoc\ ((adverse\ z\ i\ j) \triangleright m_1)\ 0$
by (*simp add*: *init-Suc-snoc*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*
then have *adverse z i j* (*Suc* m_1) $\downarrow= 1$
using *m1 adverse-Suc* **by** *simp*
then have $(adverse\ z\ i\ j) \triangleright (Suc\ m_1) = e\text{-}snoc\ ((adverse\ z\ i\ j) \triangleright m_1)\ 1$
by (*simp add*: *init-Suc-snoc*)
with *False m1*(*2*) **show** *?thesis* **by** *simp*
qed
then show $\exists m_1{>}n.\ \exists m_2{>}n.\ s\ (adverse\ z\ i\ j \triangleright m_1) \neq s\ (adverse\ z\ i\ j \triangleright m_2)$
using *less-SucI m1*(*1*) **by** *blast*
qed
qed

The next result corresponds to Case 2 from the proof sketch.

lemma *no-hyp-change-no-cons*:
assumes $x > 0$ **and** $\neg$ *hyp-change z i j x*
shows $\neg$ *learn-cons* φ {*adverse 0 i j*, *adverse 1 i j*} *s*
proof −
let $?P = \lambda x.\ x > 0 \land \neg\ hyp\text{-}change\ z\ i\ j\ x$
define *xmin* **where** *xmin* = *Least ?P*
with *assms* **have** *xmin*:
?P xmin
$\bigwedge x.\ x < xmin \Longrightarrow \neg\ ?P\ x$
using *LeastI*[*of ?P*] *not-less-Least*[*of - ?P*] **by** *simp-all*
then have $xmin > 0$ **by** *simp*

have $\forall x{\le}xmin - 1.\ x > 0 \longrightarrow hyp\text{-}change\ z\ i\ j\ x$
using *xmin* **by** (*metis One-nat-def Suc-pred le-imp-less-Suc*)
then have
$\forall x{\le}xmin.\ adverse\ z\ i\ j\ x = adverse\ 0\ i\ j\ x$
$\forall x{\le}xmin.\ adverse\ z\ i\ j\ x = adverse\ 1\ i\ j\ x$
using *while-hyp-change*[*of xmin − 1 z i j 0*]

using *while-hyp-change*[*of xmin* − *1 z i j 1*]
by *simp-all*
then have
init-z0: (*adverse z i j*) ▷ *xmin* = (*adverse 0 i j*) ▷ *xmin* **and**
init-z1: (*adverse z i j*) ▷ *xmin* = (*adverse 1 i j*) ▷ *xmin*
using *adverse-in-R1 init-eqI* **by** *blast+*
then have
a0: *adverse 0 i j* (*Suc xmin*) ↓= *0* **and**
a1: *adverse 1 i j* (*Suc xmin*) ↓= *1*
using *adverse-Suc-not-hyp-change xmin*(*1*) *init-z1*
by *metis+*
then have
i0: (*adverse 0 i j*) ▷ (*Suc xmin*) = *e-snoc* ((*adverse z i j*) ▷ *xmin*) *0* **and**
i1: (*adverse 1 i j*) ▷ (*Suc xmin*) = *e-snoc* ((*adverse z i j*) ▷ *xmin*) *1*
using *init-z0 init-z1* **by** (*simp-all add*: *init-Suc-snoc*)
moreover have
s (*e-snoc* ((*adverse z i j*) ▷ *xmin*) *0*) = *s* ((*adverse z i j*) ▷ *xmin*)
s (*e-snoc* ((*adverse z i j*) ▷ *xmin*) *1*) = *s* ((*adverse z i j*) ▷ *xmin*)
using *xmin* **by** *simp-all*
ultimately have
s ((*adverse 0 i j*) ▷ (*Suc xmin*)) = *s* ((*adverse z i j*) ▷ *xmin*)
s ((*adverse 1 i j*) ▷ (*Suc xmin*)) = *s* ((*adverse z i j*) ▷ *xmin*)
by *simp-all*
then have
s ((*adverse 0 i j*) ▷ (*Suc xmin*)) = *s* ((*adverse 1 i j*) ▷ (*Suc xmin*))
by *simp*
moreover have (*adverse 0 i j*) ▷ (*Suc xmin*) ≠ (*adverse 1 i j*) ▷ (*Suc xmin*)
using *a0 a1 i0 i1* **by** (*metis append1-eq-conv list-decode-encode zero-neq-one*)
ultimately show ¬ *learn-cons* φ {*adverse 0 i j*, *adverse 1 i j*} *s*
using *same-hyp-different-init-not-cons* **by** *blast*
qed

Combining the previous two lemmas shows that V_{01} cannot be learned consistently in the limit by the total strategy S.

lemma *V01-not-in-R-cons*: ¬ *learn-cons* φ V_{01} *s*
proof −
obtain *m n* **where**
mn0: *adverse 0 m n* ∈ V_{01} **and**
mn1: *adverse 1 m n* ∈ V_{01}
using *adverse-in-V01* **by** *auto*
show ¬ *learn-cons* φ V_{01} *s*
proof (*cases* ∀ *x*>*0*. *hyp-change 0 m n x*)
case *True*
then have ¬ *learn-lim* φ {*adverse 0 m n*} *s*
using *always-hyp-change-no-lim* **by** *simp*
with *mn0* **show** *?thesis*
using *learn-cons-def learn-lim-closed-subseteq* **by** *auto*
next
case *False*
then obtain *x* **where** *x*: *x* > *0* ¬ *hyp-change 0 m n x* **by** *auto*
then have ¬ *learn-cons* φ {*adverse 0 m n*, *adverse 1 m n*} *s*
using *no-hyp-change-no-cons*[*OF x*] **by** *simp*
with *mn0 mn1* **show** *?thesis* **using** *learn-cons-closed-subseteq* **by** *auto*
qed
qed

end

V_{01} is in CONS

At first glance, consistently learning V_{01} looks fairly easy. After all every $f \in V_{01}$ provides a Gödel number of itself either at argument 0 or 1. A strategy only has to figure out which one is right. However, the strategy S we are going to devise does not always converge to $f(0)$ or $f(1)$. Instead it uses a technique called "amalgamation". The amalgamation of two Gödel numbers i and j is a function whose value at x is determined by simulating $\varphi_i(x)$ and $\varphi_j(x)$ in parallel and outputting the value of the first one to halt. If neither halts the value is undefined. There is a function $a \in \mathcal{R}^2$ such that $\varphi_{a(i,j)}$ is the amalgamation of i and j.

If $f \in V_{01}$ then $\varphi_{a(f(0),f(1))}$ is total because by definition of V_{01} we have $\varphi_{f(0)} = f$ or $\varphi_{f(1)} = f$ and f is total.

Given a prefix f^n of an $f \in V_{01}$ the strategy S first computes $\varphi_{a(f(0),f(1))}(x)$ for $x = 0, \ldots, n$. For the resulting prefix $\varphi^n_{a(f(0),f(1))}$ there are two cases:

Case 1. It differs from f^n, say at minimum index x. Then for either $z = 0$ or $z = 1$ we have $\varphi_{f(z)}(x) \neq f(x)$ by definition of amalgamation. This implies $\varphi_{f(z)} \neq f$, and thus $\varphi_{f(1-z)} = f$ by definition of V_{01}. We set $S(f^n) = f(1-z)$. This hypothesis is correct and hence consistent.

Case 2. It equals f^n. Then we set $S(f^n) = a(f(0), f(1))$. This hypothesis is consistent by definition of this case.

In both cases the hypothesis is consistent. If Case 1 holds for some n, the same x and z will be found also for all larger values of n. Therefore S converges to the correct hypothesis $f(1-z)$. If Case 2 holds for all n, then S always outputs the same hypothesis $a(f(0), f(1))$ and thus also converges.

The above discussion tacitly assumes $n \geq 1$, such that both $f(0)$ and $f(1)$ are available to S. For $n = 0$ the strategy outputs an arbitrary consistent hypothesis.

Amalgamation uses the concurrent simulation of functions.

```
definition parallel :: nat ⇒ nat ⇒ nat ⇒ nat option where
  parallel i j x ≡ eval r-parallel [i, j, x]

lemma r-parallel': eval r-parallel [i, j, x] = parallel i j x
  using parallel-def by simp

lemma r-parallel'':
  shows eval r-phi [i, x] ↑ ∧ eval r-phi [j, x] ↑ ⟹ eval r-parallel [i, j, x] ↑
    and eval r-phi [i, x] ↓ ∧ eval r-phi [j, x] ↑ ⟹
      eval r-parallel [i, j, x] ↓= prod-encode (0, the (eval r-phi [i, x]))
    and eval r-phi [j, x] ↓ ∧ eval r-phi [i, x] ↑ ⟹
      eval r-parallel [i, j, x] ↓= prod-encode (1, the (eval r-phi [j, x]))
    and eval r-phi [i, x] ↓ ∧ eval r-phi [j, x] ↓ ⟹
      eval r-parallel [i, j, x] ↓= prod-encode (0, the (eval r-phi [i, x])) ∨
      eval r-parallel [i, j, x] ↓= prod-encode (1, the (eval r-phi [j, x]))
proof −
  let ?f = Cn 1 r-phi [r-const i, Id 1 0]
  let ?g = Cn 1 r-phi [r-const j, Id 1 0]
  have *: ⋀x. eval r-phi [i, x] = eval ?f [x] ⋀x. eval r-phi [j, x] = eval ?g [x]
```

by *simp-all*
show *eval r-phi* $[i, x] \uparrow \wedge$ *eval r-phi* $[j, x] \uparrow \Longrightarrow$ *eval r-parallel* $[i, j, x] \uparrow$
and *eval r-phi* $[i, x] \downarrow \wedge$ *eval r-phi* $[j, x] \uparrow \Longrightarrow$
eval r-parallel $[i, j, x] \downarrow=$ *prod-encode* $(0,$ *the* $($*eval r-phi* $[i, x]))$
and *eval r-phi* $[j, x] \downarrow \wedge$ *eval r-phi* $[i, x] \uparrow \Longrightarrow$
eval r-parallel $[i, j, x] \downarrow=$ *prod-encode* $(1,$ *the* $($*eval r-phi* $[j, x]))$
and *eval r-phi* $[i, x] \downarrow \wedge$ *eval r-phi* $[j, x] \downarrow \Longrightarrow$
eval r-parallel $[i, j, x] \downarrow=$ *prod-encode* $(0,$ *the* $($*eval r-phi* $[i, x])) \vee$
eval r-parallel $[i, j, x] \downarrow=$ *prod-encode* $(1,$ *the* $($*eval r-phi* $[j, x]))$
using *r-parallel*[*OF* ∗] **by** *simp-all*
qed

lemma *parallel*:
$\varphi\ i\ x \uparrow \wedge \varphi\ j\ x \uparrow \Longrightarrow$ *parallel* $i\ j\ x \uparrow$
$\varphi\ i\ x \downarrow \wedge \varphi\ j\ x \uparrow \Longrightarrow$ *parallel* $i\ j\ x \downarrow=$ *prod-encode* $(0,$ *the* $(\varphi\ i\ x))$
$\varphi\ j\ x \downarrow \wedge \varphi\ i\ x \uparrow \Longrightarrow$ *parallel* $i\ j\ x \downarrow=$ *prod-encode* $(1,$ *the* $(\varphi\ j\ x))$
$\varphi\ i\ x \downarrow \wedge \varphi\ j\ x \downarrow \Longrightarrow$
parallel $i\ j\ x \downarrow=$ *prod-encode* $(0,$ *the* $(\varphi\ i\ x)) \vee$
parallel $i\ j\ x \downarrow=$ *prod-encode* $(1,$ *the* $(\varphi\ j\ x))$
using *phi-def r-parallel″ r-parallel parallel-def* **by** *simp-all*

lemma *parallel-converg-pdec1-0-or-1*:
assumes *parallel* $i\ j\ x \downarrow$
shows *pdec1* $($*the* $($*parallel* $i\ j\ x)) = 0 \vee$ *pdec1* $($*the* $($*parallel* $i\ j\ x)) = 1$
using *assms parallel*[*of i x j*] *parallel*(*3*)[*of j x i*]
by (*metis fst-eqD option.sel prod-encode-inverse*)

lemma *parallel-converg-either*: $(\varphi\ i\ x \downarrow \vee \varphi\ j\ x \downarrow) = ($*parallel* $i\ j\ x \downarrow)$
using *parallel* **by** (*metis option.simps*(*3*))

lemma *parallel-0*:
assumes *parallel* $i\ j\ x \downarrow=$ *prod-encode* $(0, v)$
shows $\varphi\ i\ x \downarrow= v$
using *parallel assms*
by (*smt option.collapse option.sel option.simps*(*3*) *prod.inject prod-encode-eq zero-neq-one*)

lemma *parallel-1*:
assumes *parallel* $i\ j\ x \downarrow=$ *prod-encode* $(1, v)$
shows $\varphi\ j\ x \downarrow= v$
using *parallel assms*
by (*smt option.collapse option.sel option.simps*(*3*) *prod.inject prod-encode-eq zero-neq-one*)

lemma *parallel-converg-V01*:
assumes $f \in V_{01}$
shows *parallel* $($*the* $(f\ 0))$ $($*the* $(f\ 1))$ $x \downarrow$
proof −
have $f \in \mathcal{R} \wedge (\varphi$ $($*the* $(f\ 0)) = f \vee \varphi$ $($*the* $(f\ 1)) = f)$
using *assms V01-def* **by** *auto*
then have φ $($*the* $(f\ 0)) \in \mathcal{R} \vee \varphi$ $($*the* $(f\ 1)) \in \mathcal{R}$
by *auto*
then have φ $($*the* $(f\ 0))$ $x \downarrow \vee \varphi$ $($*the* $(f\ 1))$ $x \downarrow$
using *R1-imp-total1* **by** *auto*
then show *?thesis* **using** *parallel-converg-either* **by** *simp*
qed

The amalgamation of two Gödel numbers can then be described in terms of *parallel*.

definition *amalgamation* :: *nat* ⇒ *nat* ⇒ *partial1* **where**
amalgamation i j x ≡
if parallel i j x ↑ *then None else Some* (*pdec2* (*the* (*parallel i j x*)))

lemma *amalgamation-diverg*: *amalgamation i j x* ↑ ⟷ *φ i x* ↑ ∧ *φ j x* ↑
using *amalgamation-def parallel* **by** (*metis option.simps*(*3*))

lemma *amalgamation-total*:
assumes *total1* (*φ i*) ∨ *total1* (*φ j*)
shows *total1* (*amalgamation i j*)
using *assms amalgamation-diverg*[*of i j*] *total-def* **by** *auto*

lemma *amalgamation-V01-total*:
assumes *f* ∈ V_{01}
shows *total1* (*amalgamation* (*the* (*f 0*)) (*the* (*f 1*)))
using *assms V01-def amalgamation-total R1-imp-total1 total1-def*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)

definition *r-amalgamation* ≡ *Cn 3 r-pdec2* [*r-parallel*]

lemma *r-amalgamation-recfn*: *recfn 3 r-amalgamation*
unfolding *r-amalgamation-def* **by** *simp*

lemma *r-amalgamation*: *eval r-amalgamation* [*i*, *j*, *x*] = *amalgamation i j x*
proof (*cases parallel i j x* ↑)
case *True*
then have *eval r-parallel* [*i*, *j*, *x*] ↑
by (*simp add*: *r-parallel′*)
then have *eval r-amalgamation* [*i*, *j*, *x*] ↑
unfolding *r-amalgamation-def* **by** *simp*
moreover from *True* **have** *amalgamation i j x* ↑
using *amalgamation-def* **by** *simp*
ultimately show *?thesis* **by** *simp*
next
case *False*
then have *eval r-parallel* [*i*, *j*, *x*] ↓
by (*simp add*: *r-parallel′*)
then have *eval r-amalgamation* [*i*, *j*, *x*] = *eval r-pdec2* [*the* (*eval r-parallel* [*i*, *j*, *x*])]
unfolding *r-amalgamation-def* **by** *simp*
also have ... ↓= *pdec2* (*the* (*eval r-parallel* [*i*, *j*, *x*]))
by *simp*
finally show *?thesis* **by** (*simp add*: *False amalgamation-def r-parallel′*)
qed

The function *amalgamate* computes Gödel numbers of amalgamations. It corresponds to the function *a* from the proof sketch.

definition *amalgamate* :: *nat* ⇒ *nat* ⇒ *nat* **where**
amalgamate i j ≡ *smn 1* (*encode r-amalgamation*) [*i*, *j*]

lemma *amalgamate*: *φ* (*amalgamate i j*) = *amalgamation i j*
proof
fix *x*
have *φ* (*amalgamate i j*) *x* = *eval r-phi* [*amalgamate i j*, *x*]
by (*simp add*: *phi-def*)
also have ... = *eval r-phi* [*smn 1* (*encode r-amalgamation*) [*i*, *j*], *x*]
using *amalgamate-def* **by** *simp*

```
  also have ... = eval r-phi
     [encode (Cn 1 (r-universal 3)
      (r-constn 0 (encode r-amalgamation) # map (r-constn 0) [i, j] @ map (Id 1) [0])), x]
    using smn[of 1 encode r-amalgamation [i, j]] by (simp add: numeral-3-eq-3)
  also have ... = eval r-phi
     [encode (Cn 1 (r-universal 3)
      (r-const (encode r-amalgamation) # [r-const i, r-const j, Id 1 0])), x]
     (is ... = eval r-phi [encode ?f, x])
    by (simp add: r-constn-def)
  finally have φ (amalgamate i j) x = eval r-phi
     [encode (Cn 1 (r-universal 3)
      (r-const (encode r-amalgamation) # [r-const i, r-const j, Id 1 0])), x] .
  then have φ (amalgamate i j) x = eval (r-universal 3) [encode r-amalgamation, i, j, x]
    unfolding r-phi-def using r-universal[of ?f 1] r-amalgamation-recfn by simp
  then show φ (amalgamate i j) x = amalgamation i j x
    using r-amalgamation by (simp add: r-amalgamation-recfn r-universal)
qed

lemma amalgamation-in-P1: amalgamation i j ∈ 𝒫
  using amalgamate by (metis P2-proj-P1 phi-in-P2)

lemma amalgamation-V01-R1:
  assumes f ∈ V01
  shows amalgamation (the (f 0)) (the (f 1)) ∈ ℛ
  using assms amalgamation-V01-total amalgamation-in-P1
  by (simp add: P1-total-imp-R1)

definition r-amalgamate ≡
  Cn 2 (r-smn 1 2) [r-dummy 1 (r-const (encode r-amalgamation)), Id 2 0, Id 2 1]

lemma r-amalgamate-recfn: recfn 2 r-amalgamate
  unfolding r-amalgamate-def by simp

lemma r-amalgamate: eval r-amalgamate [i, j] ↓= amalgamate i j
proof −
  let ?p = encode r-amalgamation
  have rs21: eval (r-smn 1 2) [?p, i, j] ↓= smn 1 ?p [i, j]
    using r-smn by simp
  moreover have eval r-amalgamate [i, j] = eval (r-smn 1 2) [?p, i, j]
    unfolding r-amalgamate-def by auto
  ultimately have eval r-amalgamate [i, j] ↓= smn 1 ?p [i, j]
    by simp
  then show ?thesis using amalgamate-def by simp
qed
```

The strategy S distinguishes the two cases from the proof sketch with the help of the next function, which checks if a hypothesis φ_i is inconsistent with a prefix e. If so, it returns the least $x < |e|$ witnessing the inconsistency; otherwise it returns the length $|e|$. If φ_i diverges for some $x < |e|$, so does the function.

```
definition inconsist :: partial2 where
  inconsist i e ≡
   (if ∃ x<e-length e. φ i x ↑ then None
    else if ∃ x<e-length e. φ i x ↓≠ e-nth e x
         then Some (LEAST x. x < e-length e ∧ φ i x ↓≠ e-nth e x)
         else Some (e-length e))
```

```
lemma inconsist-converg:
  assumes inconsist i e ↓
  shows inconsist i e =
    (if ∃x<e-length e. φ i x ↓≠ e-nth e x
     then Some (LEAST x. x < e-length e ∧ φ i x ↓≠ e-nth e x)
     else Some (e-length e))
    and ∀x<e-length e. φ i x ↓
  using inconsist-def assms by (presburger, meson)

lemma inconsist-bounded:
  assumes inconsist i e ↓
  shows the (inconsist i e) ≤ e-length e
proof (cases ∃x<e-length e. φ i x ↓≠ e-nth e x)
  case True
  then show ?thesis
    using inconsist-converg[OF assms]
    by (smt Least-le dual-order.strict-implies-order dual-order.strict-trans2 option.sel)
next
  case False
  then show ?thesis using inconsist-converg[OF assms] by auto
qed

lemma inconsist-consistent:
  assumes inconsist i e ↓
  shows inconsist i e ↓= e-length e ⟷ (∀x<e-length e. φ i x ↓= e-nth e x)
proof
  show ∀x<e-length e. φ i x ↓= e-nth e x if inconsist i e ↓= e-length e
  proof (cases ∃x<e-length e. φ i x ↓≠ e-nth e x)
    case True
    then show ?thesis
      using that inconsist-converg[OF assms]
      by (metis (mono-tags, lifting) not-less-Least option.inject)
  next
    case False
    then show ?thesis
      using that inconsist-converg[OF assms] by simp
  qed
  show ∀x<e-length e. φ i x ↓= e-nth e x ⟹ inconsist i e ↓= e-length e
    unfolding inconsist-def using assms by auto
qed

lemma inconsist-converg-eq:
  assumes inconsist i e ↓= e-length e
  shows ∀x<e-length e. φ i x ↓= e-nth e x
  using assms inconsist-consistent by auto

lemma inconsist-converg-less:
  assumes inconsist i e ↓ and the (inconsist i e) < e-length e
  shows ∃x<e-length e. φ i x ↓≠ e-nth e x
    and inconsist i e ↓= (LEAST x. x < e-length e ∧ φ i x ↓≠ e-nth e x)
proof -
  show ∃x<e-length e. φ i x ↓≠ e-nth e x
    using assms by (metis (no-types, lifting) inconsist-converg(1) nat-neq-iff option.sel)
  then show inconsist i e ↓= (LEAST x. x < e-length e ∧ φ i x ↓≠ e-nth e x)
    using assms inconsist-converg by presburger
```

qed

lemma *least-bounded-Suc*:
  assumes $\exists x.\ x < upper \wedge P\ x$
  shows $(LEAST\ x.\ x < upper \wedge P\ x) = (LEAST\ x.\ x < Suc\ upper \wedge P\ x)$
proof −
  let $?Q = \lambda x.\ x < upper \wedge P\ x$
  let $?x = Least\ ?Q$
  from *assms* **have** $?x < upper \wedge P\ ?x$
    using *LeastI-ex*[*of ?Q*] **by** *simp*
  then have *1*: $?x < Suc\ upper \wedge P\ ?x$ **by** *simp*
  from *assms* **have** *2*: $\forall y{<}?x.\ \neg\ P\ y$
    using *Least-le*[*of ?Q*] *not-less-Least* **by** *fastforce*
  have $(LEAST\ x.\ x < Suc\ upper \wedge P\ x) = ?x$
  proof (*rule Least-equality*)
    show $?x < Suc\ upper \wedge P\ ?x$ **using** *1 2* **by** *blast*
    show $\bigwedge y.\ y < Suc\ upper \wedge P\ y \Longrightarrow ?x \le y$
      using *1 2 leI* **by** *blast*
  qed
  then show *?thesis* **..**
qed

lemma *least-bounded-gr*:
  fixes $P :: nat \Rightarrow bool$ **and** $m :: nat$
  assumes $\exists x.\ x < upper \wedge P\ x$
  shows $(LEAST\ x.\ x < upper \wedge P\ x) = (LEAST\ x.\ x < upper + m \wedge P\ x)$
proof (*induction m*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc m*)
  moreover have $\exists x.\ x < upper + m \wedge P\ x$
    using *assms trans-less-add1* **by** *blast*
  ultimately show *?case* **using** *least-bounded-Suc* **by** *simp*
qed

lemma *inconsist-init-converg-less*:
  assumes $f \in \mathcal{R}$
    and $\varphi\ i \in \mathcal{R}$
    and $inconsist\ i\ (f \triangleright n) \downarrow$
    and $the\ (inconsist\ i\ (f \triangleright n)) < Suc\ n$
  shows $inconsist\ i\ (f \triangleright (n + m)) = inconsist\ i\ (f \triangleright n)$
proof −
  have *phi-i-total*: $\varphi\ i\ x \downarrow$ **for** x
    using *assms* **by** *simp*
  moreover have *f-nth*: $f\ x \downarrow= e\text{-}nth\ (f \triangleright n)\ x$ **if** $x < Suc\ n$ **for** $x\ n$
    using *that assms*(*1*) **by** *simp*
  ultimately have $(\varphi\ i\ x \neq f\ x) = (\varphi\ i\ x \downarrow\neq e\text{-}nth\ (f \triangleright n)\ x)$ **if** $x < Suc\ n$ **for** $x\ n$
    using *that* **by** *simp*
  then have *cond*: $(x < Suc\ n \wedge \varphi\ i\ x \neq f\ x) =$
      $(x < e\text{-}length\ (f \triangleright n) \wedge \varphi\ i\ x \downarrow\neq e\text{-}nth\ (f \triangleright n)\ x)$ **for** $x\ n$
    using *length-init* **by** *metis*
  then have
    1: $\exists x{<}Suc\ n.\ \varphi\ i\ x \neq f\ x$ **and**
    2: $inconsist\ i\ (f \triangleright n) \downarrow= (LEAST\ x.\ x < Suc\ n \wedge \varphi\ i\ x \neq f\ x)$
    using *assms*(*3*,*4*) *inconsist-converg-less*[*of i f* $\triangleright$ *n*] **by** *simp-all*

```
  then have 3: ∃x<Suc (n + m). φ i x ≠ f x
    using not-add-less1 by fastforce
  then have ∃x<Suc (n + m). φ i x ↓≠ e-nth (f ▹ (n + m)) x
    using cond by blast
  then have ∃x<e-length (f ▹ (n + m)). φ i x ↓≠ e-nth (f ▹ (n + m)) x
    by simp
  moreover have 4: inconsist i (f ▹ (n + m)) ↓
    using assms(2) R1-imp-total1 inconsist-def by simp
  ultimately have inconsist i (f ▹ (n + m)) ↓=
      (LEAST x. x < e-length (f ▹ (n + m)) ∧ φ i x ↓≠ e-nth (f ▹ (n + m)) x)
    using inconsist-converg[OF 4] by simp
  then have 5: inconsist i (f ▹ (n + m)) ↓= (LEAST x. x < Suc (n + m) ∧ φ i x ≠ f x)
    using cond[of - n + m] by simp
  then have (LEAST x. x < Suc n ∧ φ i x ≠ f x) =
      (LEAST x. x < Suc n + m ∧ φ i x ≠ f x)
    using least-bounded-gr[where ?upper=Suc n] 1 3 by simp
  then show ?thesis using 2 5 by simp
qed

definition r-inconsist ≡
  let
    f = Cn 2 r-length [Id 2 1];
    g = Cn 4 r-ifless
      [Id 4 1,
       Cn 4 r-length [Id 4 3],
       Id 4 1,
       Cn 4 r-ifeq
        [Cn 4 r-phi [Id 4 2, Id 4 0],
         Cn 4 r-nth [Id 4 3, Id 4 0],
         Id 4 1,
         Id 4 0]]
   in Cn 2 (Pr 2 f g) [Cn 2 r-length [Id 2 1], Id 2 0, Id 2 1]

lemma r-inconsist-recfn: recfn 2 r-inconsist
  unfolding r-inconsist-def by simp

lemma r-inconsist: eval r-inconsist [i, e] = inconsist i e
proof −
  define f where f = Cn 2 r-length [Id 2 1]
  define len where len = Cn 4 r-length [Id 4 3]
  define nth where nth = Cn 4 r-nth [Id 4 3, Id 4 0]
  define ph where ph = Cn 4 r-phi [Id 4 2, Id 4 0]
  define g where
    g = Cn 4 r-ifless [Id 4 1, len, Id 4 1, Cn 4 r-ifeq [ph, nth, Id 4 1, Id 4 0]]
  have recfn 2 f
    unfolding f-def by simp
  have f: eval f [i, e] ↓= e-length e
    unfolding f-def by simp
  have recfn 4 len
    unfolding len-def by simp
  have len: eval len [j, v, i, e] ↓= e-length e for j v
    unfolding len-def by simp
  have recfn 4 nth
    unfolding nth-def by simp
  have nth: eval nth [j, v, i, e] ↓= e-nth e j for j v
    unfolding nth-def by simp
```

```
have recfn 4 ph
  unfolding ph-def by simp
have ph: eval ph [j, v, i, e] = φ i j for j v
  unfolding ph-def using phi-def by simp
have recfn 4 g
  unfolding g-def using ‹recfn 4 nth› ‹recfn 4 ph› ‹recfn 4 len› by simp
have g-diverg: eval g [j, v, i, e] ↑ if eval ph [j, v, i, e] ↑ for j v
  unfolding g-def using that ‹recfn 4 nth› ‹recfn 4 ph› ‹recfn 4 len› by simp
have g-converg: eval g [j, v, i, e] ↓=
    (if v < e-length e then v else if φ i j ↓= e-nth e j then v else j)
    if eval ph [j, v, i, e] ↓ for j v
  unfolding g-def using that ‹recfn 4 nth› ‹recfn 4 ph› ‹recfn 4 len› len nth ph
  by auto
define h where h ≡ Pr 2 f g
then have recfn 3 h
  by (simp add: ‹recfn 2 f› ‹recfn 4 g›)

let ?invariant = λj i e.
  (if ∃x<j. φ i x ↑ then None
   else if ∃x<j. φ i x ↓≠ e-nth e x
        then Some (LEAST x. x < j ∧ φ i x ↓≠ e-nth e x)
        else Some (e-length e))

have eval h [j, i, e] = ?invariant j i e if j ≤ e-length e for j
  using that
proof (induction j)
  case 0
  then show ?case unfolding h-def using ‹recfn 2 f› f ‹recfn 4 g› by simp
next
  case (Suc j)
  then have j-less: j < e-length e by simp
  then have j-le: j ≤ e-length e by simp
  show ?case
  proof (cases eval h [j, i, e] ↑)
    case True
    then have ∃x<j. φ i x ↑
      using j-le Suc.IH by (metis option.simps(3))
    then have ∃x<Suc j. φ i x ↑
      using less-SucI by blast
    moreover have h: eval h [Suc j, i, e] ↑
      using True h-def ‹recfn 3 h› by simp
    ultimately show ?thesis by simp
  next
    case False
    with Suc.IH j-le have h-j: eval h [j, i, e] =
      (if ∃x<j. φ i x ↓≠ e-nth e x
       then Some (LEAST x. x < j ∧ φ i x ↓≠ e-nth e x)
       else Some (e-length e))
      by presburger
    then have the-h-j: the (eval h [j, i, e]) =
      (if ∃x<j. φ i x ↓≠ e-nth e x
       then LEAST x. x < j ∧ φ i x ↓≠ e-nth e x
       else e-length e)
       (is - = ?v)
      by auto
    have h-Suc: eval h [Suc j, i, e] = eval g [j, the (eval h [j, i, e]), i, e]
```

```
    using False h-def ‹recfn 4 g› ‹recfn 2 f› by auto
  show ?thesis
  proof (cases φ i j ↑)
    case True
    with ph g-diverg h-Suc show ?thesis by auto
  next
    case False
    with h-Suc have eval h [Suc j, i, e] ↓=
      (if ?v < e-length e then ?v
       else if φ i j ↓= e-nth e j then ?v else j)
      (is - ↓= ?lhs)
      using g-converg ph the-h-j by simp
    moreover have ?invariant (Suc j) i e ↓=
      (if ∃x<Suc j. φ i x ↓≠ e-nth e x
       then LEAST x. x < Suc j ∧ φ i x ↓≠ e-nth e x
       else e-length e)
      (is - ↓= ?rhs)
    proof −
      from False have φ i j ↓ by simp
      moreover have ¬ (∃x<j. φ i x ↑)
        by (metis (no-types, lifting) Suc.IH h-j j-le option.simps(3))
      ultimately have ¬ (∃x<Suc j. φ i x ↑)
        using less-Suc-eq by auto
      then show ?thesis by auto
    qed
    moreover have ?lhs = ?rhs
    proof (cases ?v < e-length e)
      case True
      then have
        ex-j: ∃x<j. φ i x ↓≠ e-nth e x and
        v-eq: ?v = (LEAST x. x < j ∧ φ i x ↓≠ e-nth e x)
        by presburger+
      with True have ?lhs = ?v by simp
      from ex-j have ∃x<Suc j. φ i x ↓≠ e-nth e x
        using less-SucI by blast
      then have ?rhs = (LEAST x. x < Suc j ∧ φ i x ↓≠ e-nth e x) by simp
      with True v-eq ex-j show ?thesis
        using least-bounded-Suc[of j λx. φ i x ↓≠ e-nth e x] by simp
    next
      case False
      then have not-ex: ¬ (∃x<j. φ i x ↓≠ e-nth e x)
        using Least-le[of λx. x < j ∧ φ i x ↓≠ e-nth e x] j-le
        by (smt leD le-less-linear le-trans)
      then have ?v = e-length e by argo
      with False have lhs: ?lhs = (if φ i j ↓= e-nth e j then e-length e else j)
        by simp
      show ?thesis
      proof (cases φ i j ↓= e-nth e j)
        case True
        then have ¬ (∃x<Suc j. φ i x ↓≠ e-nth e x)
          using less-SucE not-ex by blast
        then have ?rhs = e-length e by argo
        moreover from True have ?lhs = e-length e
          using lhs by simp
        ultimately show ?thesis by simp
      next case False
```

```
          then have φ i j ↓≠ e-nth e j
            using ‹φ i j ↓› by simp
          with not-ex have (LEAST x. x<Suc j ∧ φ i x ↓≠ e-nth e x) = j
            using LeastI[of λx. x<Suc j ∧ φ i x ↓≠ e-nth e x j] less-Suc-eq
            by blast
          then have ?rhs = j
            using ‹φ i j ↓≠ e-nth e j› by (meson lessI)
          moreover from False lhs have ?lhs = j by simp
          ultimately show ?thesis by simp
        qed
      qed
      ultimately show ?thesis by simp
    qed
  qed
 qed
 then have eval h [e-length e, i, e] = ?invariant (e-length e) i e
   by auto
 then have eval h [e-length e, i, e] = inconsist i e
   using inconsist-def by simp
 moreover have eval (Cn 2 (Pr 2 f g) [Cn 2 r-length [Id 2 1], Id 2 0, Id 2 1]) [i, e] =
     eval h [e-length e, i, e]
   using ‹recfn 4 g› ‹recfn 2 f› h-def by auto
 ultimately show ?thesis
   unfolding r-inconsist-def by (simp add: f-def g-def len-def nth-def ph-def)
qed

lemma inconsist-for-total:
 assumes total1 (φ i)
 shows inconsist i e ↓=
   (if ∃x<e-length e. φ i x ↓≠ e-nth e x
    then LEAST x. x < e-length e ∧ φ i x ↓≠ e-nth e x
    else e-length e)
 unfolding inconsist-def using assms total1-def by (auto; blast)

lemma inconsist-for-V01:
 assumes f ∈ V01 and k = amalgamate (the (f 0)) (the (f 1))
 shows inconsist k e ↓=
   (if ∃x<e-length e. φ k x ↓≠ e-nth e x
    then LEAST x. x < e-length e ∧ φ k x ↓≠ e-nth e x
    else e-length e)
proof -
 have φ k ∈ R
   using amalgamation-V01-R1[OF assms(1)] assms(2) amalgamate by simp
 then have total1 (φ k) by simp
 with inconsist-for-total[of k] show ?thesis by simp
qed
```

The next function computes Gödel numbers of functions consistent with a given prefix. The strategy will use these as consistent auxiliary hypotheses when receiving a prefix of length one.

```
definition r-auxhyp ≡ Cn 1 (r-smn 1 1) [r-const (encode r-prenum), Id 1 0]

lemma r-auxhyp-prim: prim-recfn 1 r-auxhyp
 unfolding r-auxhyp-def by simp

lemma r-auxhyp: φ (the (eval r-auxhyp [e])) = prenum e
```

proof
  fix *x*
  let *?p* = *encode r-prenum*
  let *?p* = *encode r-prenum*
  have *eval r-auxhyp* [*e*] = *eval* (*r-smn 1 1*) [*?p*, *e*]
    unfolding *r-auxhyp-def* **by** *simp*
  then have *eval r-auxhyp* [*e*] ↓= *smn 1 ?p* [*e*]
    by (*simp add*: *r-smn*)
  also have ... ↓= *encode* (*Cn 1* (*r-universal* (*1* + *length* [*e*]))
      (*r-constn* (*1* − *1*) *?p* #
       map (*r-constn* (*1* − *1*)) [*e*] @ *map* (*recf.Id 1*) [*0..<1*]))
    using *smn*[*of 1 ?p* [*e*]] **by** *simp*
  also have ... ↓= *encode* (*Cn 1* (*r-universal* (*1* + *1*))
      (*r-constn 0 ?p* # *map* (*r-constn 0*) [*e*] @ [*Id 1 0*]))
    by *simp*
  also have ... ↓= *encode* (*Cn 1* (*r-universal 2*)
      (*r-constn 0 ?p* # *map* (*r-constn 0*) [*e*] @ [*Id 1 0*]))
    by (*metis one-add-one*)
  also have ... ↓= *encode* (*Cn 1* (*r-universal 2*) [*r-constn 0 ?p*, *r-constn 0 e*, *Id 1 0*])
    by *simp*
  also have ... ↓= *encode* (*Cn 1* (*r-universal 2*) [*r-const ?p*, *r-const e*, *Id 1 0*])
    using *r-constn-def* **by** *simp*
  finally have *eval r-auxhyp* [*e*] ↓=
    encode (*Cn 1* (*r-universal 2*) [*r-const ?p*, *r-const e*, *Id 1 0*]) **.**
  moreover have φ (*the* (*eval r-auxhyp* [*e*])) *x* = *eval r-phi* [*the* (*eval r-auxhyp* [*e*]), *x*]
    by (*simp add*: *phi-def*)
  ultimately have φ (*the* (*eval r-auxhyp* [*e*])) *x* =
      eval r-phi [*encode* (*Cn 1* (*r-universal 2*) [*r-const ?p*, *r-const e*, *Id 1 0*]), *x*]
      (**is** - = *eval r-phi* [*encode ?f*, *x*])
    by *simp*
  then have φ (*the* (*eval r-auxhyp* [*e*])) *x* =
      eval (*Cn 1* (*r-universal 2*) [*r-const ?p*, *r-const e*, *Id 1 0*]) [*x*]
    using *r-phi-def r-universal*[*of ?f 1* [*x*]] **by** *simp*
  then have φ (*the* (*eval r-auxhyp* [*e*])) *x* = *eval* (*r-universal 2*) [*?p*, *e*, *x*]
    by *simp*
  then have φ (*the* (*eval r-auxhyp* [*e*])) *x* = *eval r-prenum* [*e*, *x*]
    using *r-universal* **by** *simp*
  then show φ (*the* (*eval r-auxhyp* [*e*])) *x* = *prenum e x* **by** *simp*
qed

definition *auxhyp* :: *partial1* **where**
  auxhyp e ≡ *eval r-auxhyp* [*e*]

lemma *auxhyp-prenum*: φ (*the* (*auxhyp e*)) = *prenum e*
  using *auxhyp-def r-auxhyp* **by** *metis*

lemma *auxhyp-in-R1*: *auxhyp* ∈ $\mathcal{R}$
  using *auxhyp-def Mn-free-imp-total R1I r-auxhyp-prim* **by** *metis*

Now we can define our consistent learning strategy for V_{01}.

definition *r-sv01* ≡
  let
    at0 = *Cn 1 r-nth* [*Id 1 0*, *Z*];
    at1 = *Cn 1 r-nth* [*Id 1 0*, *r-const 1*];
    m = *Cn 1 r-amalgamate* [*at0*, *at1*];
    c = *Cn 1 r-inconsist* [*m*, *Id 1 0*];

p = *Cn 1 r-pdec1* [*Cn 1 r-parallel* [*at0*, *at1*, *c*]];
g = *Cn 1 r-ifeq* [*c*, *r-length*, *m*, *Cn 1 r-ifz* [*p*, *at1*, *at0*]]
in Cn 1 (*r-lifz r-auxhyp g*) [*Cn 1 r-eq* [*r-length*, *r-const 1*], *Id 1 0*]

lemma *r-sv01-recfn*: *recfn 1 r-sv01*
unfolding *r-sv01-def* **using** *r-auxhyp-prim r-inconsist-recfn r-amalgamate-recfn*
by (*simp add*: *Let-def*)

definition *sv01* :: *partial1* (‹s_{01}›)**where**
sv01 e ≡ *eval r-sv01* [*e*]

lemma *sv01-in-P1*: s_{01} ∈ $\mathcal{P}$
using *sv01-def r-sv01-recfn P1I* **by** *presburger*

We are interested in the behavior of s_{01} only on prefixes of functions in V_{01}. This behavior is linked to the amalgamation of $f(0)$ and $f(1)$, where f is the function to be learned.

abbreviation *amalg01* :: *partial1* ⇒ *nat* **where**
amalg01 f ≡ *amalgamate* (*the* (*f 0*)) (*the* (*f 1*))

lemma *sv01*:
assumes *f* ∈ V_{01}
shows s_{01} (*f* ▷ *0*) = *auxhyp* (*f* ▷ *0*)
and *n* ≠ *0* ⟹
inconsist (*amalg01 f*) (*f* ▷ *n*) ↓= *Suc n* ⟹
s_{01} (*f* ▷ *n*) ↓= *amalg01 f*
and *n* ≠ *0* ⟹
the (*inconsist* (*amalg01 f*) (*f* ▷ *n*)) < *Suc n* ⟹
pdec1 (*the* (*parallel* (*the* (*f 0*)) (*the* (*f 1*)) (*the* (*inconsist* (*amalg01 f*) (*f* ▷ *n*))))) = *0* ⟹
s_{01} (*f* ▷ *n*) = *f 1*
and *n* ≠ *0* ⟹
the (*inconsist* (*amalg01 f*) (*f* ▷ *n*)) < *Suc n* ⟹
pdec1 (*the* (*parallel* (*the* (*f 0*)) (*the* (*f 1*)) (*the* (*inconsist* (*amalg01 f*) (*f* ▷ *n*))))) ≠ *0* ⟹
s_{01} (*f* ▷ *n*) = *f 0*
proof −
have *f-total*: ⋀*x*. *f x* ↓
using *assms V01-def R1-imp-total1* **by** *blast*
define *at0* **where** *at0* = *Cn 1 r-nth* [*Id 1 0*, *Z*]
define *at1* **where** *at1* = *Cn 1 r-nth* [*Id 1 0*, *r-const 1*]
define *m* **where** *m* = *Cn 1 r-amalgamate* [*at0*, *at1*]
define *c* **where** *c* = *Cn 1 r-inconsist* [*m*, *Id 1 0*]
define *p* **where** *p* = *Cn 1 r-pdec1* [*Cn 1 r-parallel* [*at0*, *at1*, *c*]]
define *g* **where** *g* = *Cn 1 r-ifeq* [*c*, *r-length*, *m*, *Cn 1 r-ifz* [*p*, *at1*, *at0*]]
have *recfn 1 g*
unfolding *g-def p-def c-def m-def at1-def at0-def*
using *r-auxhyp-prim r-inconsist-recfn r-amalgamate-recfn*
by *simp*
have *eval* (*Cn 1 r-eq* [*r-length*, *r-const 1*]) [*f* ▷ *0*] ↓= *0*
by *simp*
then have *eval r-sv01* [*f* ▷ *0*] = *eval r-auxhyp* [*f* ▷ *0*]
unfolding *r-sv01-def* **using** ‹*recfn 1 g*› *c-def g-def m-def p-def r-auxhyp-prim*
by (*auto simp add*: *Let-def*)
then show s_{01} (*f* ▷ *0*) = *auxhyp* (*f* ▷ *0*)
by (*simp add*: *auxhyp-def sv01-def*)

have *sv01*: s_{01} (*f* ▷ *n*) = *eval g* [*f* ▷ *n*] **if** *n* ≠ *0*

proof −
  have ∗: *eval* (*Cn 1 r-eq* [*r-length*, *r-const 1*]) [*f* ▷ *n*] ↓≠ *0*
    (**is** *?r-eq* ↓≠ *0*)
    using *that* **by** *simp*
  moreover have *recfn 2* (*r-lifz r-auxhyp g*)
    using ‹*recfn 1 g*› *r-auxhyp-prim* **by** *simp*
  moreover have *eval r-sv01* [*f* ▷ *n*] =
      eval (*Cn 1* (*r-lifz r-auxhyp g*) [*Cn 1 r-eq* [*r-length*, *r-const 1*], *Id 1 0*]) [*f* ▷ *n*]
    using *r-sv01-def* **by** (*metis at0-def at1-def c-def g-def m-def p-def*)
  ultimately have *eval r-sv01* [*f* ▷ *n*] = *eval* (*r-lifz r-auxhyp g*) [*the ?r-eq*, *f* ▷ *n*]
    by *simp*
  then have *eval r-sv01* [*f* ▷ *n*] = *eval g* [*f* ▷ *n*]
    using ∗ ‹*recfn 1 g*› *r-auxhyp-prim* **by** *auto*
  then show *?thesis* **by** (*simp add*: *sv01-def that*)
qed

have *recfn 1 at0*
  unfolding *at0-def* **by** *simp*
have *at0*: *eval at0* [*f* ▷ *n*] ↓= *the* (*f 0*)
  unfolding *at0-def* **by** *simp*
have *recfn 1 at1*
  unfolding *at1-def* **by** *simp*
have *at1*: *n* ≠ *0* ⟹ *eval at1* [*f* ▷ *n*] ↓= *the* (*f 1*)
  unfolding *at1-def* **by** *simp*
have *recfn 1 m*
  unfolding *m-def at0-def at1-def* **using** *r-amalgamate-recfn* **by** *simp*
have *m*: *n* ≠ *0* ⟹ *eval m* [*f* ▷ *n*] ↓= *amalg01 f*
    (**is** - ⟹ - ↓= *?m*)
  unfolding *m-def at0-def at1-def*
  using *at0 at1 amalgamate r-amalgamate r-amalgamate-recfn* **by** *simp*
then have *c*: *n* ≠ *0* ⟹ *eval c* [*f* ▷ *n*] = *inconsist* (*amalg01 f*) (*f* ▷ *n*)
    (**is** - ⟹ - = *?c*)
  unfolding *c-def* **using** *r-inconsist-recfn* ‹*recfn 1 m*› *r-inconsist* **by** *auto*
then have *c-converg*: *n* ≠ *0* ⟹ *eval c* [*f* ▷ *n*] ↓
  using *inconsist-for-V01*[*OF assms*] **by** *simp*
have *recfn 1 c*
  unfolding *c-def* **using** ‹*recfn 1 m*› *r-inconsist-recfn* **by** *simp*

have *par*: *n* ≠ *0* ⟹
    eval (*Cn 1 r-parallel* [*at0*, *at1*, *c*]) [*f* ▷ *n*] = *parallel* (*the* (*f 0*)) (*the* (*f 1*)) (*the ?c*)
    (**is** - ⟹ - = *?par*)
  using *at0 at1 c c-converg m r-parallel′* ‹*recfn 1 at0*› ‹*recfn 1 at1*› ‹*recfn 1 c*›
  by *simp*
with *parallel-converg-V01*[*OF assms*] **have**
    par-converg: *n* ≠ *0* ⟹ *eval* (*Cn 1 r-parallel* [*at0*, *at1*, *c*]) [*f* ▷ *n*] ↓
  by *simp*
then have *p-converg*: *n* ≠ *0* ⟹ *eval p* [*f* ▷ *n*] ↓
  unfolding *p-def* **using** *at0 at1 c-converg* ‹*recfn 1 at0*› ‹*recfn 1 at1*› ‹*recfn 1 c*›
  by *simp*
have *p*: *n* ≠ *0* ⟹ *eval p* [*f* ▷ *n*] ↓= *pdec1* (*the ?par*)
  unfolding *p-def*
  using *at0 at1 c-converg* ‹*recfn 1 at0*› ‹*recfn 1 at1*› ‹*recfn 1 c*› *par par-converg*
  by *simp*
have *recfn 1 p*
  unfolding *p-def* **using** ‹*recfn 1 at0*› ‹*recfn 1 at1*› ‹*recfn 1 m*› ‹*recfn 1 c*›
  by *simp*

let *?r = Cn 1 r-ifz [p, at1, at0]*
have *r: n ≠ 0 ⟹ eval ?r [f ▷ n] = (if pdec1 (the ?par) = 0 then f 1 else f 0)*
using *at0 at1 c-converg ‹recfn 1 at0› ‹recfn 1 at1› ‹recfn 1 c›*
‹recfn 1 m› ‹recfn 1 p› p f-total
by *fastforce*

have *g: n ≠ 0 ⟹*
eval g [f ▷ n] ↓=
(if the ?c = e-length (f ▷ n)
then ?m else the (eval (Cn 1 r-ifz [p, at1, at0]) [f ▷ n]))
unfolding *g-def*
using *‹recfn 1 p› ‹recfn 1 at0› ‹recfn 1 at1› ‹recfn 1 c› ‹recfn 1 m›*
p-converg at1 at0 c c-converg m
by *simp*
{
assume *n ≠ 0* **and** *?c ↓= Suc n*
moreover have *e-length (f ▷ n) = Suc n* **by** *simp*
ultimately have *eval g [f ▷ n] ↓= ?m* **using** *g* **by** *simp*
then show s_{01} *(f ▷ n) ↓= amalg01 f*
using *sv01[OF ‹n ≠ 0›]* **by** *simp*
next
assume *n ≠ 0* **and** *the ?c < Suc n* **and** *pdec1 (the ?par) = 0*
with *g r f-total* **have** *eval g [f ▷ n] = f 1* **by** *simp*
then show s_{01} *(f ▷ n) = f 1*
using *sv01[OF ‹n ≠ 0›]* **by** *simp*
next
assume *n ≠ 0* **and** *the ?c < Suc n* **and** *pdec1 (the ?par) ≠ 0*
with *g r f-total* **have** *eval g [f ▷ n] = f 0* **by** *simp*
then show s_{01} *(f ▷ n) = f 0*
using *sv01[OF ‹n ≠ 0›]* **by** *simp*
}
qed

Part of the correctness of s_{01} is convergence on prefixes of functions in V_{01}.

lemma *sv01-converg-V01*:
assumes *f ∈* V_{01}
shows s_{01} *(f ▷ n) ↓*
proof *(cases n = 0)*
case *True*
then show *?thesis*
using *assms sv01 R1-imp-total1 auxhyp-in-R1* **by** *simp*
next
case *n-gr-0: False*
show *?thesis*
proof *(cases inconsist (amalg01 f) (f ▷ n) ↓= Suc n)*
case *True*
then show *?thesis*
using *n-gr-0 assms sv01* **by** *simp*
next
case *False*
then have *the (inconsist (amalg01 f) (f ▷ n)) < Suc n*
using *assms inconsist-bounded inconsist-for-V01 length-init*
by *(metis (no-types, lifting) le-neq-implies-less option.collapse option.simps(3))*
then show *?thesis*
using *n-gr-0 assms sv01 R1-imp-total1 total1E V01-def*

by (*metis* (*no-types*, *lifting*) *mem-Collect-eq*)
qed
qed

Another part of the correctness of s_{01} is its hypotheses being consistent on prefixes of functions in V_{01}.

lemma *sv01-consistent-V01*:
assumes $f \in V_{01}$
shows $\forall x \le n.\ \varphi\ (the\ (s_{01}\ (f \triangleright n)))\ x = f\ x$
proof (*cases* $n = 0$)
case *True*
then have $s_{01}\ (f \triangleright n) = auxhyp\ (f \triangleright n)$
using *sv01*[*OF assms*] **by** *simp*
then have $\varphi\ (the\ (s_{01}\ (f \triangleright n))) = prenum\ (f \triangleright n)$
using *auxhyp-prenum* **by** *simp*
then show *?thesis*
using *R1-imp-total1 total1E assms* **by** (*simp add*: *V01-def*)
next
case *n-gr-0*: *False*
let $?m = amalg01\ f$
let $?e = f \triangleright n$
let $?c = the\ (inconsist\ ?m\ ?e)$
have *c*: $inconsist\ ?m\ ?e \downarrow$
using *assms inconsist-for-V01* **by** *blast*
show *?thesis*
proof (*cases* $inconsist\ ?m\ ?e \downarrow= Suc\ n$)
case *True*
then show *?thesis*
using *assms n-gr-0 sv01 R1-imp-total1 total1E V01-def is-init-of-def inconsist-consistent not-initial-imp-not-eq length-init inconsist-converg-eq*
by (*metis* (*no-types*, *lifting*) *le-imp-less-Suc mem-Collect-eq option.sel*)
next
case *False*
then have *less*: $the\ (inconsist\ ?m\ ?e) < Suc\ n$
using *c assms inconsist-bounded inconsist-for-V01 length-init*
by (*metis le-neq-implies-less option.collapse*)
then have $the\ (inconsist\ ?m\ ?e) < e\text{-}length\ ?e$
by *auto*
then have
$\exists x < e\text{-}length\ ?e.\ \varphi\ ?m\ x \downarrow\neq e\text{-}nth\ ?e\ x$
$inconsist\ ?m\ ?e \downarrow= (LEAST\ x.\ x < e\text{-}length\ ?e \wedge \varphi\ ?m\ x \downarrow\neq e\text{-}nth\ ?e\ x)$
(**is** - $\downarrow=$ *Least ?P*)
using *inconsist-converg-less*[*OF c*] **by** *simp-all*
then have $?P\ ?c$ **and** $\bigwedge x.\ x < ?c \Longrightarrow \neg\ ?P\ x$
using *LeastI-ex*[*of ?P*] *not-less-Least*[*of - ?P*] **by** (*auto simp del*: *e-nth*)
then have $\varphi\ ?m\ ?c \neq f\ ?c$ **by** *auto*
then have $amalgamation\ (the\ (f\ 0))\ (the\ (f\ 1))\ ?c \neq f\ ?c$
using *amalgamate* **by** *simp*
then have $*$: $Some\ (pdec2\ (the\ (parallel\ (the\ (f\ 0))\ (the\ (f\ 1))\ ?c))) \neq f\ ?c$
using *amalgamation-def* **by** (*metis assms parallel-converg-V01*)
let $?p = parallel\ (the\ (f\ 0))\ (the\ (f\ 1))\ ?c$
show *?thesis*
proof (*cases* $pdec1\ (the\ ?p) = 0$)
case *True*
then have $\varphi\ (the\ (f\ 0))\ ?c \downarrow= pdec2\ (the\ ?p)$
using *assms parallel-0 parallel-converg-V01*

by (*metis option.collapse prod.collapse prod-decode-inverse*)
then have φ (*the* (*f 0*)) *?c* $\neq$ *f ?c*
using $*$ **by** *simp*
then have φ (*the* (*f 0*)) $\neq$ *f* **by** *auto*
then have φ (*the* (*f 1*)) $=$ *f*
using *assms V01-def* **by** *auto*
moreover have s_{01} ($f \triangleright n$) $=$ *f 1*
using *True less n-gr-0 sv01 assms* **by** *simp*
ultimately show *?thesis* **by** *simp*
next
case *False*
then have *pdec1* (*the ?p*) $=$ *1*
by (*meson assms parallel-converg-V01 parallel-converg-pdec1-0-or-1*)
then have φ (*the* (*f 1*)) *?c* $\downarrow=$ *pdec2* (*the ?p*)
using *assms parallel-1 parallel-converg-V01*
by (*metis option.collapse prod.collapse prod-decode-inverse*)
then have φ (*the* (*f 1*)) *?c* $\neq$ *f ?c*
using $*$ **by** *simp*
then have φ (*the* (*f 1*)) $\neq$ *f* **by** *auto*
then have φ (*the* (*f 0*)) $=$ *f*
using *assms V01-def* **by** *auto*
moreover from *False less n-gr-0 sv01 assms* **have** s_{01} ($f \triangleright n$) $=$ *f 0*
by *simp*
ultimately show *?thesis* **by** *simp*
qed
qed
qed

The final part of the correctness is s_{01} converging for all functions in V_{01}.

lemma *sv01-limit-V01*:
assumes $f \in V_{01}$
shows $\exists i.\ \forall^\infty n.\ s_{01}\ (f \triangleright n) \downarrow= i$
proof (*cases* $\forall n>0.\ s_{01}\ (f \triangleright n) \downarrow=$ *amalgamate* (*the* (*f 0*)) (*the* (*f 1*)))
case *True*
then show *?thesis* **by** (*meson less-le-trans zero-less-one*)
next
case *False*
then obtain n_0 **where** *n0*:
$n_0 \neq 0$
$s_{01}\ (f \triangleright n_0) \downarrow\neq$ *amalg01 f*
using ‹$f \in V_{01}$› *sv01-converg-V01* **by** *blast*
then have $*$: *the* (*inconsist* (*amalg01 f*) ($f \triangleright n_0$)) $<$ *Suc* n_0
(**is** *the* (*inconsist ?m* ($f \triangleright n_0$)) $<$ *Suc* n_0)
using *assms* ‹$n_0 \neq 0$› *sv01*(*2*) *inconsist-bounded inconsist-for-V01 length-init*
by (*metis* (*no-types, lifting*) *le-neq-implies-less option.collapse option.simps*(*3*))
moreover have $f \in \mathcal{R}$
using *assms V01-def* **by** *auto*
moreover have φ *?m* $\in \mathcal{R}$
using *amalgamate amalgamation-V01-R1 assms* **by** *auto*
moreover have *inconsist ?m* ($f \triangleright n_0$) $\downarrow$
using *inconsist-for-V01 assms* **by** *blast*
ultimately have $**$: *inconsist ?m* ($f \triangleright (n_0 + m)$) $=$ *inconsist ?m* ($f \triangleright n_0$) **for** *m*
using *inconsist-init-converg-less*[*of f ?m*] **by** *simp*
then have *the* (*inconsist ?m* ($f \triangleright (n_0 + m)$)) $<$ *Suc* $n_0 + m$ **for** *m*
using $*$ **by** *auto*
moreover have

pdec1 (*the* (*parallel* (*the* (*f 0*)) (*the* (*f 1*)) (*the* (*inconsist ?m* (*f* ▷ (n_0 + *m*)))))) =
pdec1 (*the* (*parallel* (*the* (*f 0*)) (*the* (*f 1*)) (*the* (*inconsist ?m* (*f* ▷ n_0)))))
for *m*
using ∗∗ **by** *auto*
moreover have $n_0 + m \neq 0$ **for** *m*
using ‹$n_0 \neq 0$› **by** *simp*
ultimately have s_{01} (*f* ▷ (n_0 + *m*)) = s_{01} (*f* ▷ n_0) **for** *m*
using *assms sv01* ∗ ‹$n_0 \neq 0$› **by** (*metis add-Suc*)
moreover define *i* **where** *i* = s_{01} (*f* ▷ n_0)
ultimately have $\forall n \geq n_0$. s_{01} (*f* ▷ *n*) = *i*
using *nat-le-iff-add* **by** *auto*
then have $\forall n \geq n_0$. s_{01} (*f* ▷ *n*) ↓= *the i*
using *n0*(*2*) **by** *simp*
then show *?thesis* **by** *auto*
qed

lemma *V01-learn-cons*: *learn-cons* φ V_{01} s_{01}
proof (*rule learn-consI2*)
show *environment* φ V_{01} s_{01}
by (*simp add*: *Collect-mono V01-def phi-in-P2 sv01-in-P1 sv01-converg-V01*)
show $\bigwedge f\, n.\ f \in V_{01} \Longrightarrow \forall k \leq n.$ φ (*the* (s_{01} (*f* ▷ *n*))) *k* = *f k*
using *sv01-consistent-V01* **.**
show $\exists\, i\ n_0.\ \forall n \geq n_0.$ s_{01} (*f* ▷ *n*) ↓= *i* **if** $f \in V_{01}$ **for** *f*
using *sv01-limit-V01 that* **by** *simp*
qed

corollary *V01-in-CONS*: V_{01} ∈ *CONS*
using *V01-learn-cons CONS-def* **by** *auto*

Now we can show the main result of this section, namely that there is a consistently learnable class that cannot be learned consistently by a total strategy. In other words, there is no Lemma R for CONS.

lemma *no-lemma-R-for-CONS*: ∃ *U*. *U* ∈ *CONS* ∧ (¬ (∃ *s*. *s* ∈ $\mathcal{R}$ ∧ *learn-cons* φ *U s*))
using *V01-in-CONS V01-not-in-R-cons* **by** *auto*

end

2.9 LIM is a proper subset of BC

theory *LIM-BC*
imports *Lemma-R*
begin

The proper inclusion of LIM in BC has been proved by Barzdin [2] (see also Case and Smith [6]). The proof constructs a class $V \in \text{BC} - \text{LIM}$ by diagonalization against all LIM strategies. Exploiting Lemma R for LIM, we can assume that all such strategies are total functions. From the effective version of this lemma we derive a numbering $\sigma \in \mathcal{R}^2$ such that for all $U \in \text{LIM}$ there is an i with $U \in \text{LIM}_\varphi(\sigma_i)$. The idea behind V is for every i to construct a class V_i of cardinality one or two such that $V_i \notin \text{LIM}_\varphi(\sigma_i)$. It then follows that the union $V := \bigcup_i V_i$ cannot be learned by any σ_i and thus $V \notin \text{LIM}$. At the same time, the construction ensures that the functions in V are "predictable enough" to be learnable in the BC sense.

At the core is a process that maintains a state (b, k) of a list b of numbers and an index $k < |b|$ into this list. We imagine b to be the prefix of the function being constructed, except for position k where we imagine b to have a "gap"; that is, b_k is not defined yet. Technically, we will always have $b_k = 0$, so b also represents the prefix after the "gap is filled" with 0, whereas $b_{k:=1}$ represents the prefix where the gap is filled with 1. For every $i \in \mathbb{N}$, the process starts in state $(i0, 1)$ and computes the next state from a given state (b, k) as follows:

1. if $\sigma_i(b_{<k}) \neq \sigma_i(b)$ then the next state is $(b0, |b|)$,
2. else if $\sigma_i(b_{<k}) \neq \sigma_i(b_{k:=1})$ then the next state is $(b_{k:=1}0, |b|)$,
3. else the next state is $(b0, k)$.

In other words, if σ_i changes its hypothesis when the gap in b is filled with 0 or 1, then the process fills the gap with 0 or 1, respectively, and appends a gap to b. If, however, a hypothesis change cannot be enforced at this point, the process appends a 0 to b and leaves the gap alone. Now there are two cases:

Case 1. Every gap gets filled eventually. Then the process generates increasing prefixes of a total function τ_i, on which σ_i changes its hypothesis infinitely often. We set $V_i := \{\tau_i\}$, and have $V_i \notin \text{LIM}_\varphi(\sigma_i)$.

Case 2. Some gap never gets filled. That means a state (b, k) is reached such that $\sigma_i(b0^t) = \sigma_i(b_{k:=1}0^t) = \sigma_i(b_{<k})$ for all t. Then the process describes a function $\tau_i = b_{<k} \uparrow 0^\infty$, where the value at the gap k is undefined. Replacing the value at k by 0 and 1 yields two functions $\tau_i^{(0)} = b0^\infty$ and $\tau_i^{(1)} = b_{k:=1}0^\infty$, which differ only at k and on which σ_i converges to the same hypothesis. Thus σ_i does not learn the class $V_i := \{\tau_i^{(0)}, \tau_i^{(1)}\}$ in the limit.

Both cases combined imply $V \notin \text{LIM}$.

A BC strategy S for $V = \bigcup_i V_i$ works as follows. Let $f \in V$. On input f^n the strategy outputs a Gödel number of the function

$$g_n(x) = \begin{cases} f(x) & \text{if } x \leq n, \\ \tau_{f(0)}(x) & \text{otherwise.} \end{cases}$$

By definition of V, f is generated by the process running for $i = f(0)$. If $f(0)$ leads to Case 1 then $f = \tau_{f(0)}$, and g_n equals f for all n. If $f(0)$ leads to Case 2 with a forever unfilled gap at k, then g_n will be equal to the correct one of $\tau_i^{(0)}$ or $\tau_i^{(1)}$ for all $n \geq k$. Intuitively, the prefix received by S eventually grows long enough to reveal the value $f(k)$. In both cases S converges to f, but it outputs a different Gödel number for every f^n because g_n contains the "hard-coded" values $f(0), \ldots, f(n)$. Therefore S is a BC strategy but not a LIM strategy for V.

2.9.1 Enumerating enough total strategies

For the construction of σ we need the function *r-limr* from the effective version of Lemma R for LIM.

definition *r-sigma* ≡ *Cn 2 r-phi* [*Cn 2 r-limr* [*Id 2 0*], *Id 2 1*]

lemma *r-sigma-recfn*: *recfn 2 r-sigma*
  unfolding *r-sigma-def* **using** *r-limr-recfn* **by** *simp*

lemma *r-sigma*: *eval r-sigma* [*i*, *x*] = *φ* (*the* (*eval r-limr* [*i*])) *x*
unfolding *r-sigma-def phi-def* **using** *r-sigma-recfn r-limr-total r-limr-recfn*
by *simp*

lemma *r-sigma-total*: *total r-sigma*
using *r-sigma r-limr r-sigma-recfn totalI2*[*of r-sigma*] **by** *simp*

abbreviation *sigma* :: *partial2* (‹*σ*›) **where**
σ i x ≡ *eval r-sigma* [*i*, *x*]

lemma *sigma*: *σ i* = *φ* (*the* (*eval r-limr* [*i*]))
using *r-sigma* **by** *simp*

The numbering σ does indeed enumerate enough total strategies for every LIM learning problem.

lemma *learn-lim-sigma*:
assumes *learn-lim ψ U* (*φ i*)
shows *learn-lim ψ U* (*σ i*)
using *assms sigma r-limr* **by** *simp*

2.9.2 The diagonalization process

The following function represents the process described above. It computes the next state from a given state (b, k).

definition *r-next* ≡
Cn 1 r-ifeq
[*Cn 1 r-sigma* [*Cn 1 r-hd* [*r-pdec1*], *r-pdec1*],
Cn 1 r-sigma [*Cn 1 r-hd* [*r-pdec1*], *Cn 1 r-take* [*r-pdec2*, *r-pdec1*]],
Cn 1 r-ifeq
[*Cn 1 r-sigma* [*Cn 1 r-hd* [*r-pdec1*], *Cn 1 r-update* [*r-pdec1*, *r-pdec2*, *r-const 1*]],
Cn 1 r-sigma [*Cn 1 r-hd* [*r-pdec1*], *Cn 1 r-take* [*r-pdec2*, *r-pdec1*]],
Cn 1 r-prod-encode [*Cn 1 r-snoc* [*r-pdec1*, *Z*], *r-pdec2*],
Cn 1 r-prod-encode
[*Cn 1 r-snoc*
[*Cn 1 r-update* [*r-pdec1*, *r-pdec2*, *r-const 1*], *Z*], *Cn 1 r-length* [*r-pdec1*]]],
Cn 1 r-prod-encode [*Cn 1 r-snoc* [*r-pdec1*, *Z*], *Cn 1 r-length* [*r-pdec1*]]]

lemma *r-next-recfn*: *recfn 1 r-next*
unfolding *r-next-def* **using** *r-sigma-recfn* **by** *simp*

The three conditions distinguished in *r-next* correspond to Steps 1, 2, and 3 of the process: hypothesis change when the gap is filled with 0; hypothesis change when the gap is filled with 1; or no hypothesis change either way.

abbreviation *change-on-0 b k* ≡ *σ* (*e-hd b*) *b* ≠ *σ* (*e-hd b*) (*e-take k b*)

abbreviation *change-on-1 b k* ≡
σ (*e-hd b*) *b* = *σ* (*e-hd b*) (*e-take k b*) ∧
σ (*e-hd b*) (*e-update b k 1*) ≠ *σ* (*e-hd b*) (*e-take k b*)

abbreviation *change-on-neither b k* ≡
σ (*e-hd b*) *b* = *σ* (*e-hd b*) (*e-take k b*) ∧
σ (*e-hd b*) (*e-update b k 1*) = *σ* (*e-hd b*) (*e-take k b*)

lemma *change-conditions*:
  obtains
    (*on-0*) *change-on-0 b k*
  | (*on-1*) *change-on-1 b k*
  | (*neither*) *change-on-neither b k*
  by *auto*

lemma *r-next*:
  assumes *arg* = *prod-encode* (*b*, *k*)
  shows *change-on-0 b k* ⟹ *eval r-next* [*arg*] ↓= *prod-encode* (*e-snoc b 0*, *e-length b*)
    and *change-on-1 b k* ⟹
      eval r-next [*arg*] ↓= *prod-encode* (*e-snoc* (*e-update b k 1*) *0*, *e-length b*)
    and *change-on-neither b k* ⟹ *eval r-next* [*arg*] ↓= *prod-encode* (*e-snoc b 0*, *k*)
proof −
  let *?bhd* = *Cn 1 r-hd* [*r-pdec1*]
  let *?bup* = *Cn 1 r-update* [*r-pdec1*, *r-pdec2*, *r-const 1*]
  let *?bk* = *Cn 1 r-take* [*r-pdec2*, *r-pdec1*]
  let *?bap* = *Cn 1 r-snoc* [*r-pdec1*, *Z*]
  let *?len* = *Cn 1 r-length* [*r-pdec1*]
  let *?thenthen* = *Cn 1 r-prod-encode* [*?bap*, *r-pdec2*]
  let *?thenelse* = *Cn 1 r-prod-encode* [*Cn 1 r-snoc* [*?bup*, *Z*], *?len*]
  let *?else* = *Cn 1 r-prod-encode* [*?bap*, *?len*]
  have *bhd*: *eval ?bhd* [*arg*] ↓= *e-hd b*
    using *assms* **by** *simp*
  have *bup*: *eval ?bup* [*arg*] ↓= *e-update b k 1*
    using *assms* **by** *simp*
  have *bk*: *eval ?bk* [*arg*] ↓= *e-take k b*
    using *assms* **by** *simp*
  have *bap*: *eval ?bap* [*arg*] ↓= *e-snoc b 0*
    using *assms* **by** *simp*
  have *len*: *eval ?len* [*arg*] ↓= *e-length b*
    using *assms* **by** *simp*
  have *else-*: *eval ?else* [*arg*] ↓= *prod-encode* (*e-snoc b 0*, *e-length b*)
    using *bap len* **by** *simp*
  have *thenthen*: *eval ?thenthen* [*arg*] ↓= *prod-encode* (*e-snoc b 0*, *k*)
    using *bap assms* **by** *simp*
  have *thenelse*: *eval ?thenelse* [*arg*] ↓= *prod-encode* (*e-snoc* (*e-update b k 1*) *0*, *e-length b*)
    using *bup len* **by** *simp*
  have *then-*:
    eval
      (*Cn 1 r-ifeq* [*Cn 1 r-sigma* [*?bhd*, *?bup*], *Cn 1 r-sigma* [*?bhd*, *?bk*], *?thenthen*, *?thenelse*])
      [*arg*] ↓=
    (*if the* (σ (*e-hd b*) (*e-update b k 1*)) = *the* (σ (*e-hd b*) (*e-take k b*))
     then prod-encode (*e-snoc b 0*, *k*)
     else prod-encode (*e-snoc* (*e-update b k 1*) *0*, *e-length b*))
    (**is** *eval ?then* [*arg*] ↓= *?then-eval*)
    using *bhd bup bk thenthen thenelse r-sigma r-sigma-recfn r-limr R1-imp-total1* **by** *simp*
  have ∗: *eval r-next* [*arg*] ↓=
    (*if the* (σ (*e-hd b*) *b*) = *the* (σ (*e-hd b*) (*e-take k b*))
     then ?then-eval
     else prod-encode (*e-snoc b 0*, *e-length b*))
    unfolding *r-next-def*
    using *bhd bk then- else- r-sigma r-sigma-recfn r-limr R1-imp-total1 assms*
    by *simp*
  have *r-sigma-neq*: *eval r-sigma* [x_1, y_1] ≠ *eval r-sigma* [x_2, y_2] ⟷
      the (*eval r-sigma* [x_1, y_1]) ≠ *the* (*eval r-sigma* [x_2, y_2])

for x_1 y_1 x_2 y_2
using *r-sigma r-limr totalE*[*OF r-sigma-total r-sigma-recfn*] *r-sigma-recfn r-sigma-total*
by (*metis One-nat-def Suc-1 length-Cons list.size*(*3*) *option.expand*)
{
assume *change-on-0 b k*
then show *eval r-next* [*arg*] ↓= *prod-encode* (*e-snoc b 0*, *e-length b*)
using ∗ *r-sigma-neq* **by** *simp*
next
assume *change-on-1 b k*
then show *eval r-next* [*arg*] ↓= *prod-encode* (*e-snoc* (*e-update b k 1*) *0*, *e-length b*)
using ∗ *r-sigma-neq* **by** *simp*
next
assume *change-on-neither b k*
then show *eval r-next* [*arg*] ↓= *prod-encode* (*e-snoc b 0*, *k*)
using ∗ *r-sigma-neq* **by** *simp*
}
qed

lemma *r-next-total*: *total r-next*
proof (*rule totalI1*)
show *recfn 1 r-next*
using *r-next-recfn* **by** *simp*
show *eval r-next* [*x*] ↓ **for** *x*
proof −
obtain *b k* **where** *x* = *prod-encode* (*b*, *k*)
using *prod-encode-pdec′*[*of x*] **by** *metis*
then show *?thesis* **using** *r-next* **by** *fast*
qed
qed

The next function computes the state of the process after any number of iterations.

definition *r-state* ≡
Pr 1
(*Cn 1 r-prod-encode* [*Cn 1 r-snoc* [*Cn 1 r-singleton-encode* [*Id 1 0*], *Z*], *r-const 1*])
(*Cn 3 r-next* [*Id 3 1*])

lemma *r-state-recfn*: *recfn 2 r-state*
unfolding *r-state-def* **using** *r-next-recfn* **by** *simp*

lemma *r-state-at-0*: *eval r-state* [*0*, *i*] ↓= *prod-encode* (*list-encode* [*i*, *0*], *1*)
proof −
let *?f* = *Cn 1 r-prod-encode* [*Cn 1 r-snoc* [*Cn 1 r-singleton-encode* [*Id 1 0*], *Z*], *r-const 1*]
have *eval r-state* [*0*, *i*] = *eval ?f* [*i*]
unfolding *r-state-def* **using** *r-next-recfn* **by** *simp*
also have ... ↓= *prod-encode* (*list-encode* [*i*, *0*], *1*)
by (*simp add*: *list-decode-singleton*)
finally show *?thesis* **.**
qed

lemma *r-state-total*: *total r-state*
unfolding *r-state-def*
using *r-next-recfn totalE*[*OF r-next-total r-next-recfn*] *totalI3*[*of Cn 3 r-next* [*Id 3 1*]]
by (*intro Pr-total*) *auto*

We call the components of a state (b, k) the *block* b and the *gap* k.

definition *block* :: *nat* ⇒ *nat* ⇒ *nat* **where**

block i t ≡ *pdec1* (*the* (*eval r-state* [*t*, *i*]))

definition *gap* :: *nat* ⇒ *nat* ⇒ *nat* **where**
gap i t ≡ *pdec2* (*the* (*eval r-state* [*t*, *i*]))

lemma *state-at-0*:
block i 0 = *list-encode* [*i*, *0*]
gap i 0 = *1*
unfolding *block-def gap-def r-state-at-0* **by** *simp-all*

Some lemmas describing the behavior of blocks and gaps in one iteration of the process:

lemma *state-Suc*:
assumes *b* = *block i t* **and** *k* = *gap i t*
shows *block i* (*Suc t*) = *pdec1* (*the* (*eval r-next* [*prod-encode* (*b*, *k*)]))
and *gap i* (*Suc t*) = *pdec2* (*the* (*eval r-next* [*prod-encode* (*b*, *k*)]))
proof −
have *eval r-state* [*Suc t*, *i*] =
eval (*Cn 3 r-next* [*Id 3 1*]) [*t*, *the* (*eval r-state* [*t*, *i*]), *i*]
using *r-state-recfn r-next-recfn totalE*[*OF r-state-total r-state-recfn*, *of* [*t*, *i*]]
by (*simp add*: *r-state-def*)
also have ... = *eval r-next* [*the* (*eval r-state* [*t*, *i*])]
using *r-next-recfn* **by** *simp*
also have ... = *eval r-next* [*prod-encode* (*b*, *k*)]
using *assms block-def gap-def* **by** *simp*
finally have *eval r-state* [*Suc t*, *i*] = *eval r-next* [*prod-encode* (*b*, *k*)] **.**
then show
block i (*Suc t*) = *pdec1* (*the* (*eval r-next* [*prod-encode* (*b*, *k*)]))
gap i (*Suc t*) = *pdec2* (*the* (*eval r-next* [*prod-encode* (*b*, *k*)]))
by (*simp add*: *block-def*, *simp add*: *gap-def*)
qed

lemma *gap-Suc*:
assumes *b* = *block i t* **and** *k* = *gap i t*
shows *change-on-0 b k* ⟹ *gap i* (*Suc t*) = *e-length b*
and *change-on-1 b k* ⟹ *gap i* (*Suc t*) = *e-length b*
and *change-on-neither b k*⟹ *gap i* (*Suc t*) = *k*
using *assms r-next state-Suc* **by** *simp-all*

lemma *block-Suc*:
assumes *b* = *block i t* **and** *k* = *gap i t*
shows *change-on-0 b k* ⟹ *block i* (*Suc t*) = *e-snoc b 0*
and *change-on-1 b k* ⟹ *block i* (*Suc t*) = *e-snoc* (*e-update b k 1*) *0*
and *change-on-neither b k*⟹ *block i* (*Suc t*) = *e-snoc b 0*
using *assms r-next state-Suc* **by** *simp-all*

Non-gap positions in the block remain unchanged after an iteration.

lemma *block-stable*:
assumes *j* < *e-length* (*block i t*) **and** *j* ≠ *gap i t*
shows *e-nth* (*block i t*) *j* = *e-nth* (*block i* (*Suc t*)) *j*
proof −
from *change-conditions*[*of block i t gap i t*] **show** *?thesis*
using *assms block-Suc gap-Suc*
by (*cases*, (*simp-all add*: *nth-append*))
qed

Next are some properties of *block* and *gap*.

```
lemma gap-in-block: gap i t < e-length (block i t)
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  with change-conditions[of block i t gap i t] show ?case
  proof (cases)
    case on-0
    then show ?thesis by (simp add: block-Suc(1) gap-Suc(1))
  next
    case on-1
    then show ?thesis by (simp add: block-Suc(2) gap-Suc(2))
  next
    case neither
    then show ?thesis using Suc.IH block-Suc(3) gap-Suc(3) by force
  qed
qed

lemma length-block: e-length (block i t) = Suc (Suc t)
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  with change-conditions[of block i t gap i t] show ?case
    by (cases, simp-all add: block-Suc gap-Suc)
qed

lemma gap-gr0: gap i t > 0
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  with change-conditions[of block i t gap i t] show ?case
    using length-block by (cases, simp-all add: block-Suc gap-Suc)
qed

lemma hd-block: e-hd (block i t) = i
proof (induction t)
  case 0
  then show ?case by (simp add: state-at-0)
next
  case (Suc t)
  from change-conditions[of block i t gap i t] show ?case
  proof (cases)
    case on-0
    then show ?thesis
      using Suc block-Suc(1) length-block by (metis e-hd-snoc gap-Suc(1) gap-gr0)
  next
    case on-1
    let ?b = block i t and ?k = gap i t
    have ?k > 0
      using gap-gr0 Suc by simp
    then have e-nth (e-update ?b ?k 1) 0 = e-nth ?b 0
```

by *simp*
then have ∗: *e-hd* (*e-update ?b ?k 1*) = *e-hd ?b*
using *e-hd-nth0 gap-Suc*(*2*)[*of - i t*] *gap-gr0 on-1* **by** (*metis e-length-update*)
from *on-1* **have** *block i* (*Suc t*) = *e-snoc* (*e-update ?b ?k 1*) *0*
by (*simp add*: *block-Suc*(*2*))
then show *?thesis*
using *e-hd-0 e-hd-snoc Suc length-block* ‹*?k* > *0*› ∗
by (*metis e-length-update gap-Suc*(*2*) *gap-gr0 on-1*)
next
case *neither*
then show *?thesis*
by (*metis Suc block-stable e-hd-nth0 gap-gr0 length-block not-gr0 zero-less-Suc*)
qed
qed

Formally, a block always ends in zero, even if it ends in a gap.

lemma *last-block*: *e-nth* (*block i t*) (*gap i t*) = *0*
proof (*induction t*)
case *0*
then show *?case* **by** (*simp add*: *state-at-0*)
next
case (*Suc t*)
from *change-conditions*[*of block i t gap i t*] **show** *?case*
proof *cases*
case *on-0*
then show *?thesis* **using** *Suc* **by** (*simp add*: *block-Suc*(*1*) *gap-Suc*(*1*))
next
case *on-1*
then show *?thesis* **using** *Suc* **by** (*simp add*: *block-Suc*(*2*) *gap-Suc*(*2*) *nth-append*)
next
case *neither*
then have
block i (*Suc t*) = *e-snoc* (*block i t*) *0*
gap i (*Suc t*) = *gap i t*
by (*simp-all add*: *gap-Suc*(*3*) *block-Suc*(*3*))
then show *?thesis*
using *Suc gap-in-block* **by** (*simp add*: *nth-append*)
qed
qed

lemma *gap-le-Suc*: *gap i t* ≤ *gap i* (*Suc t*)
using *change-conditions*[*of block i t gap i t*]
gap-Suc gap-in-block less-imp-le[*of gap i t e-length* (*block i t*)]
by (*cases*) *simp-all*

lemma *gap-monotone*:
assumes $t_1 \leq t_2$
shows *gap i* $t_1 \leq$ *gap i* t_2
proof −
have *gap i* $t_1 \leq$ *gap i* ($t_1 + j$) **for** *j*
proof (*induction j*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc j*)
then show *?case* **using** *gap-le-Suc dual-order.trans* **by** *fastforce*

qed
then show *?thesis* **using** *assms le-Suc-ex* **by** *blast*
qed

We need some lemmas relating the shape of the next state to the hypothesis change conditions in Steps 1, 2, and 3.

```
lemma state-change-on-neither:
  assumes gap i (Suc t) = gap i t
  shows change-on-neither (block i t) (gap i t)
    and block i (Suc t) = e-snoc (block i t) 0
proof –
  let ?b = block i t and ?k = gap i t
  have ?k < e-length ?b
    using gap-in-block by simp
  from change-conditions[of ?b ?k] show change-on-neither (block i t) (gap i t)
  proof (cases)
    case on-0
    then show ?thesis
      using ‹?k < e-length ?b› assms gap-Suc(1) by auto
  next
    case on-1
    then show ?thesis using assms gap-Suc(2) by auto
  next
    case neither
    then show ?thesis by simp
  qed
  then show block i (Suc t) = e-snoc (block i t) 0
    using block-Suc(3) by simp
qed
```

```
lemma state-change-on-either:
  assumes gap i (Suc t) ≠ gap i t
  shows ¬ change-on-neither (block i t) (gap i t)
    and gap i (Suc t) = e-length (block i t)
proof –
  let ?b = block i t and ?k = gap i t
  show ¬ change-on-neither (block i t) (gap i t)
  proof
    assume change-on-neither (block i t) (gap i t)
    then have gap i (Suc t) = ?k
      by (simp add: gap-Suc(3))
    with assms show False by simp
  qed
  then show gap i (Suc t) = e-length (block i t)
    using gap-Suc(1) gap-Suc(2) by blast
qed
```

Next up is the definition of τ. In every iteration the process determines $\tau_i(x)$ for some x either by appending 0 to the current block b, or by filling the current gap k. In the former case, the value is determined for $x = |b|$, in the latter for $x = k$.

For i and x the function *r-dettime* computes in which iteration the process for i determines the value $\tau_i(x)$. This is the first iteration in which the block is long enough to contain position x and in which x is not the gap. If $\tau_i(x)$ is never determined, because Case 2 is reached with $k = x$, then *r-dettime* diverges.

```
abbreviation determined :: nat ⇒ nat ⇒ bool where
  determined i x ≡ ∃ t. x < e-length (block i t) ∧ x ≠ gap i t

lemma determined-0: determined i 0
  using gap-gr0[of i 0] gap-in-block[of i 0] by force

definition r-dettime ≡
  Mn 2
   (Cn 3 r-and
     [Cn 3 r-less
       [Id 3 2, Cn 3 r-length [Cn 3 r-pdec1 [Cn 3 r-state [Id 3 0, Id 3 1]]]],
      Cn 3 r-neq
       [Id 3 2, Cn 3 r-pdec2 [Cn 3 r-state [Id 3 0, Id 3 1]]]])

lemma r-dettime-recfn: recfn 2 r-dettime
  unfolding r-dettime-def using r-state-recfn by simp

abbreviation dettime :: partial2 where
  dettime i x ≡ eval r-dettime [i, x]

lemma r-dettime:
  shows determined i x ⟹ dettime i x ↓= (LEAST t. x < e-length (block i t) ∧ x ≠ gap i t)
    and ¬ determined i x ⟹ dettime i x ↑
proof −
  define f where f =
   (Cn 3 r-and
     [Cn 3 r-less
       [Id 3 2, Cn 3 r-length [Cn 3 r-pdec1 [Cn 3 r-state [Id 3 0, Id 3 1]]]],
      Cn 3 r-neq
       [Id 3 2, Cn 3 r-pdec2 [Cn 3 r-state [Id 3 0, Id 3 1]]]])
  then have r-dettime = Mn 2 f
    unfolding f-def r-dettime-def by simp
  have recfn 3 f
    unfolding f-def using r-state-recfn by simp
  then have total f
    unfolding f-def using Cn-total r-state-total Mn-free-imp-total by simp
  have f: eval f [t, i, x] ↓= (if x < e-length (block i t) ∧ x ≠ gap i t then 0 else 1) for t
  proof −
    let ?b = Cn 3 r-pdec1 [Cn 3 r-state [Id 3 0, Id 3 1]]
    let ?k = Cn 3 r-pdec2 [Cn 3 r-state [Id 3 0, Id 3 1]]
    have eval ?b [t, i, x] ↓= pdec1 (the (eval r-state [t, i]))
      using r-state-recfn r-state-total by simp
    then have b: eval ?b [t, i, x] ↓= block i t
      using block-def by simp
    have eval ?k [t, i, x] ↓= pdec2 (the (eval r-state [t, i]))
      using r-state-recfn r-state-total by simp
    then have k: eval ?k [t, i, x] ↓= gap i t
      using gap-def by simp
    have eval
         (Cn 3 r-neq [Id 3 2, Cn 3 r-pdec2 [Cn 3 r-state [Id 3 0, Id 3 1]]])
         [t, i, x] ↓=
       (if x ≠ gap i t then 0 else 1)
      using b k r-state-recfn r-state-total by simp
    moreover have eval
         (Cn 3 r-less
           [Id 3 2, Cn 3 r-length [Cn 3 r-pdec1 [Cn 3 r-state [Id 3 0, Id 3 1]]]])
```

[t, i, x] ↓=
(*if* x < *e-length* (*block i t*) *then 0 else 1*)
using *b k r-state-recfn r-state-total* **by** *simp*
ultimately show *?thesis*
unfolding *f-def* **using** *b k r-state-recfn r-state-total* **by** *simp*
qed
{
assume *determined i x*
with *f* **have** ∃ *t. eval f* [t, i, x] ↓= *0* **by** *simp*
then have *dettime i x* ↓= (*LEAST t. eval f* [t, i, x] ↓= *0*)
using ‹*total f*› ‹*r-dettime* = *Mn 2 f*› *r-dettime-recfn* ‹*recfn 3 f*›
eval-Mn-total[*of 2 f* [i, x]]
by *simp*
then show *dettime i x* ↓= (*LEAST t. x* < *e-length* (*block i t*) ∧ *x* ≠ *gap i t*)
using *f* **by** *simp*
next
assume ¬ *determined i x*
with *f* **have** ¬ (∃ *t. eval f* [t, i, x] ↓= *0*) **by** *simp*
then have *dettime i x* ↑
using ‹*total f*› ‹*r-dettime* = *Mn 2 f*› *r-dettime-recfn* ‹*recfn 3 f*›
eval-Mn-total[*of 2 f* [i, x]]
by *simp*
with *f* **show** *dettime i x* ↑ **by** *simp*
}
qed

lemma *r-dettimeI*:
assumes *x* < *e-length* (*block i t*) ∧ *x* ≠ *gap i t*
and ⋀*T. x* < *e-length* (*block i T*) ∧ *x* ≠ *gap i T* ⟹ *t* ≤ *T*
shows *dettime i x* ↓= *t*
proof −
let *?P* = λ*T. x* < *e-length* (*block i T*) ∧ *x* ≠ *gap i T*
have *determined i x*
using *assms*(*1*) **by** *auto*
moreover have *Least ?P* = *t*
using *assms Least-equality*[*of ?P t*] **by** *simp*
ultimately show *?thesis* **using** *r-dettime* **by** *simp*
qed

lemma *r-dettime-0*: *dettime i 0* ↓= *0*
using *r-dettimeI*[*of - i 0*] *determined-0 gap-gr0*[*of i 0*] *gap-in-block*[*of i 0*]
by *fastforce*

Computing the value of $\tau_i(x)$ works by running the process *r-state* for *dettime i x* iterations and taking the value at index x of the resulting block.

definition *r-tau* ≡ *Cn 2 r-nth* [*Cn 2 r-pdec1* [*Cn 2 r-state* [*r-dettime*, *Id 2 0*]], *Id 2 1*]

lemma *r-tau-recfn*: *recfn 2 r-tau*
unfolding *r-tau-def* **using** *r-dettime-recfn r-state-recfn* **by** *simp*

abbreviation *tau* :: *partial2* (‹τ›) **where**
τ *i x* ≡ *eval r-tau* [i, x]

lemma *tau-in-P2*: τ ∈ $\mathcal{P}^2$
using *r-tau-recfn* **by** *auto*

lemma *tau-diverg*:
  assumes ¬ *determined i x*
  shows τ *i x* ↑
  unfolding *r-tau-def* **using** *assms r-dettime r-dettime-recfn r-state-recfn* **by** *simp*

lemma *tau-converg*:
  assumes *determined i x*
  shows τ *i x* ↓= *e-nth* (*block i* (*the* (*dettime i x*))) *x*
proof −
  from *assms* **obtain** *t* **where** *t*: *dettime i x* ↓= *t*
    using *r-dettime*(*1*) **by** *blast*
  then have *eval* (*Cn 2 r-state* [*r-dettime*, *Id 2 0*]) [*i*, *x*] = *eval r-state* [*t*, *i*]
    using *r-state-recfn r-dettime-recfn* **by** *simp*
  moreover have *eval r-state* [*t*, *i*] ↓
    using *r-state-total r-state-recfn* **by** *simp*
  ultimately have *eval* (*Cn 2 r-pdec1* [*Cn 2 r-state* [*r-dettime*, *Id 2 0*]]) [*i*, *x*] =
      eval r-pdec1 [*the* (*eval r-state* [*t*, *i*])]
    using *r-state-recfn r-dettime-recfn* **by** *simp*
  then show *?thesis*
    unfolding *r-tau-def* **using** *r-state-recfn r-dettime-recfn t block-def* **by** *simp*
qed

lemma *tau-converg′*:
  assumes *dettime i x* ↓= *t*
  shows τ *i x* ↓= *e-nth* (*block i t*) *x*
  using *assms tau-converg*[*of x i*] *r-dettime*(*2*)[*of x i*] **by** *fastforce*

lemma *tau-at-0*: τ *i 0* ↓= *i*
proof −
  have τ *i 0* ↓= *e-nth* (*block i 0*) *0*
    using *tau-converg′*[*OF r-dettime-0*] **by** *simp*
  then show *?thesis* **using** *block-def* **by** (*simp add*: *r-state-at-0*)
qed

lemma *state-unchanged*:
  assumes *gap i t* − *1* ≤ *y* **and** *y* ≤ *t*
  shows *gap i t* = *gap i y*
proof −
  have *gap i t* = *gap i* (*gap i t* − *1*)
  proof (*induction t*)
    case *0*
    then show *?case* **by** (*simp add*: *gap-def r-state-at-0*)
  next
    case (*Suc t*)
    show *?case*
    proof (*cases gap i* (*Suc t*) = *t* + *2*)
      case *True*
      then show *?thesis* **by** *simp*
    next
      case *False*
      then show *?thesis*
        using *Suc state-change-on-either*(*2*) *length-block* **by** *force*
    qed
  qed
  moreover have *gap i* (*gap i t* − *1*) ≤ *gap i y*
    using *assms*(*1*) *gap-monotone* **by** *simp*

```
  moreover have gap i y ≤ gap i t
    using assms(2) gap-monotone by simp
  ultimately show ?thesis by simp
qed
```

The values of the non-gap indices x of every block created in the diagonalization process equal $\tau_i(x)$.

```
lemma tau-eq-state:
  assumes j < e-length (block i t) and j ≠ gap i t
  shows τ i j ↓= e-nth (block i t) j
  using assms
proof (induction t)
  case 0
  then have j = 0
    using gap-gr0[of i 0] gap-in-block[of i 0] length-block[of i 0] by simp
  then have τ (e-hd (block i t)) j ↓= e-nth (block i (the (dettime i 0))) 0
    using determined-0 tau-converg hd-block by simp
  then have τ (e-hd (block i t)) j ↓= e-nth (block i 0) 0
    using r-dettime-0 by simp
  then show ?case using ‹j = 0› r-dettime-0 tau-converg' by simp
next
  case (Suc t)
  let ?b = block i t
  let ?bb = block i (Suc t)
  let ?k = gap i t
  let ?kk = gap i (Suc t)
  show ?case
  proof (cases ?kk = ?k)
    case kk-eq-k: True
    then have bb-b0: ?bb = e-snoc ?b 0
      using state-change-on-neither by simp
    show τ i j ↓= e-nth ?bb j
    proof (cases j < e-length ?b)
      case True
      then have e-nth ?bb j = e-nth ?b j
        using bb-b0 by (simp add: nth-append)
      moreover have j ≠ ?k
        using Suc kk-eq-k by simp
      ultimately show ?thesis using Suc True by simp
    next
      case False
      then have j: j = e-length ?b
        using Suc.prems(1) length-block by auto
      then have e-nth ?bb j = 0
        using bb-b0 by simp
      have dettime i j ↓= Suc t
      proof (rule r-dettimeI)
        show j < e-length ?bb ∧ j ≠ ?kk
          using Suc.prems(1,2) by linarith
        show ⋀T. j < e-length (block i T) ∧ j ≠ gap i T ⟹ Suc t ≤ T
          using length-block j by simp
      qed
      with tau-converg' show ?thesis by simp
    qed
  next
    case False
```

```
    then have kk-lenb: ?kk = e-length ?b
      using state-change-on-either by simp
    then show ?thesis
    proof (cases j = ?k)
      case j-eq-k: True
      have dettime i j ↓= Suc t
      proof (rule r-dettimeI)
        show j < e-length ?bb ∧ j ≠ ?kk
          using Suc.prems(1,2) by simp
        show Suc t ≤ T if j < e-length (block i T) ∧ j ≠ gap i T for T
        proof (rule ccontr)
          assume ¬ (Suc t ≤ T)
          then have T < Suc t by simp
          then show False
          proof (cases T < ?k − 1)
            case True
            then have e-length (block i T) = T + 2
              using length-block by simp
            then have e-length (block i T) < ?k + 1
              using True by simp
            then have e-length (block i T) ≤ ?k by simp
            then have e-length (block i T) ≤ j
              using j-eq-k by simp
            then show False
              using that by simp
          next
            case False
            then have ?k − 1 ≤ T and T ≤ t
              using ‹T < Suc t› by simp-all
            with state-unchanged have gap i t = gap i T by blast
            then show False
              using j-eq-k that by simp
          qed
        qed
      qed
      then show ?thesis using tau-converg′ by simp
    next
      case False
      then have j < e-length ?b
        using kk-lenb Suc.prems(1,2) length-block by auto
      then show ?thesis using Suc False block-stable by fastforce
    qed
  qed
qed

lemma tau-eq-state′:
  assumes j < t + 2 and  j ≠ gap i t
  shows τ i j ↓= e-nth (block i t) j
  using assms tau-eq-state length-block by simp
```

We now consider the two cases described in the proof sketch. In Case 2 there is a gap that never gets filled, or equivalently there is a rightmost gap.

abbreviation *case-two i* ≡ (∃ *t*. ∀ *T*. *gap i T* ≤ *gap i t*)

abbreviation *case-one i* ≡ ¬ *case-two i*

Another characterization of Case 2 is that from some iteration on only *change-on-neither* holds.

```
lemma case-two-iff-forever-neither:
  case-two i ⟷ (∃ t. ∀ T≥t. change-on-neither (block i T) (gap i T))
proof
  assume ∃ t. ∀ T≥t. change-on-neither (block i T) (gap i T)
  then obtain t where t: ∀ T≥t. change-on-neither (block i T) (gap i T)
    by auto
  have (gap i T) ≤ (gap i t) for T
  proof (cases T ≤ t)
    case True
    then show ?thesis using gap-monotone by simp
  next
    case False
    then show ?thesis
    proof (induction T)
      case 0
      then show ?case by simp
    next
      case (Suc T)
      with t have change-on-neither ((block i T)) ((gap i T))
        by simp
      then show ?case
        using Suc.IH state-change-on-either(1)[of i T] gap-monotone[of T t i]
        by metis
    qed
  qed
  then show ∃ t. ∀ T. gap i T ≤ gap i t
    by auto
next
  assume ∃ t. ∀ T. gap i T ≤ gap i t
  then obtain t where t: ∀ T. gap i T ≤ gap i t
    by auto
  have change-on-neither (block i T) (gap i T) if T≥t for T
  proof -
    have T: (gap i T) ≥ (gap i t)
      using gap-monotone that by simp
    show ?thesis
    proof (rule ccontr)
      assume ¬ change-on-neither (block i T) (gap i T)
      then have change-on-0 (block i T) (gap i T) ∨ change-on-1 (block i T) (gap i T)
        by simp
      then have gap i (Suc T) > gap i T
        using gap-le-Suc[of i] state-change-on-either(2)[of i] state-change-on-neither(1)[of i]
          dual-order.strict-iff-order
        by blast
      with T have gap i (Suc T) > gap i t by simp
      with t show False
        using not-le by auto
    qed
  qed
  then show ∃ t. ∀ T≥t. change-on-neither (block i T) (gap i T)
    by auto
qed
```

In Case 1, τ_i is total.

lemma *case-one-tau-total*:
  assumes *case-one i*
  shows $\tau\ i\ x \downarrow$
proof (*cases* $x = gap\ i\ x$)
  case *True*
  from *assms* **have** $\forall\, t.\ \exists\, T.\ gap\ i\ T > gap\ i\ t$
    using *le-less-linear gap-def*[*of i x*] **by** *blast*
  then obtain T **where** T: $gap\ i\ T > gap\ i\ x$
    by *auto*
  then have $T > x$
    using *gap-monotone leD le-less-linear* **by** *blast*
  then have $x < T + 2$ **by** *simp*
  moreover from *T True* **have** $x \neq gap\ i\ T$ **by** *simp*
  ultimately show *?thesis* **using** *tau-eq-state′* **by** *simp*
next
  case *False*
  moreover have $x < x + 2$ **by** *simp*
  ultimately show *?thesis* **using** *tau-eq-state′* **by** *blast*
qed

In Case 2, τ_i is undefined only at the gap that never gets filled.

lemma *case-two-tau-not-quite-total*:
  assumes $\forall\, T.\ gap\ i\ T \le gap\ i\ t$
  shows $\tau\ i\ (gap\ i\ t) \uparrow$
    and $x \neq gap\ i\ t \Longrightarrow \tau\ i\ x \downarrow$
proof −
  let $?k = gap\ i\ t$
  have $\neg$ *determined i ?k*
  proof
    assume *determined i ?k*
    then obtain T **where** T: $?k <$ *e-length* (*block i T*) $\wedge\ ?k \neq gap\ i\ T$
      by *auto*
    with *assms* **have** *snd-le*: $gap\ i\ T < ?k$
      by (*simp add*: *dual-order.strict-iff-order*)
    then have $T < t$
      using *gap-monotone* **by** (*metis leD le-less-linear*)
    from *T length-block* **have** $?k < T + 2$ **by** *simp*
    moreover have $?k \neq T + 1$
      using *T state-change-on-either*(*2*) ‹$T < t$› *state-unchanged*
      by (*metis Suc-eq-plus1 Suc-leI add-diff-cancel-right′ le-add1 nat-neq-iff*)
    ultimately have $?k \le T$ **by** *simp*
    then have $gap\ i\ T = gap\ i\ ?k$
      using *state-unchanged*[*of i T ?k*] ‹$?k < T + 2$› *snd-le* **by** *simp*
    then show *False*
      by (*metis diff-le-self state-unchanged leD nat-le-linear gap-monotone snd-le*)
  qed
  with *tau-diverg* **show** $\tau\ i\ ?k \uparrow$ **by** *simp*

  assume $x \neq ?k$
  show $\tau\ i\ x \downarrow$
  proof (*cases* $x < t + 2$)
    case *True*
    with ‹$x \neq ?k$› *tau-eq-state′* **show** *?thesis* **by** *simp*
  next
    case *False*
    then have $gap\ i\ x = ?k$

```
    using assms by (simp add: dual-order.antisym gap-monotone)
  with ‹x ≠ ?k› have x ≠ gap i x by simp
  then show ?thesis using tau-eq-state'[of x x] by simp
 qed
qed

lemma case-two-tau-almost-total:
 assumes ∃ t. ∀ T. gap i T ≤ gap i t (is ∃ t. ?P t)
 shows τ i (gap i (Least ?P)) ↑
  and x ≠ gap i (Least ?P) ⟹ τ i x ↓
proof −
 from assms have ?P (Least ?P)
  using LeastI-ex[of ?P] by simp
 then show τ i (gap i (Least ?P)) ↑ and x ≠ gap i (Least ?P) ⟹ τ i x ↓
  using case-two-tau-not-quite-total by simp-all
qed
```

Some more properties of τ.

```
lemma init-tau-gap: (τ i) ▷ (gap i t − 1) = e-take (gap i t) (block i t)
proof (intro initI')
 show 1: e-length (e-take (gap i t) (block i t)) = Suc (gap i t − 1)
 proof −
  have gap i t > 0
   using gap-gr0 by simp
  moreover have gap i t < e-length (block i t)
   using gap-in-block by simp
  ultimately have e-length (e-take (gap i t) (block i t)) = gap i t
   by simp
  then show ?thesis using gap-gr0 by simp
 qed
 show τ i x ↓= e-nth (e-take (gap i t) (block i t)) x if x < Suc (gap i t − 1) for x
 proof −
  have x-le: x < gap i t
   using that gap-gr0 by simp
  then have x < e-length (block i t)
   using gap-in-block less-trans by blast
  then have ∗: τ i x ↓= e-nth (block i t) x
   using x-le tau-eq-state by auto
  have x < e-length (e-take (gap i t) (block i t))
   using x-le 1 by simp
  then have e-nth (block i t) x = e-nth (e-take (gap i t) (block i t)) x
   using x-le by simp
  then show ?thesis using ∗ by simp
 qed
qed

lemma change-on-0-init-tau:
 assumes change-on-0 (block i t) (gap i t)
 shows (τ i) ▷ (t + 1) = block i t
proof (intro initI')
 let ?b = block i t and ?k = gap i t
 show e-length (block i t) = Suc (t + 1)
  using length-block by simp
 show (τ i) x ↓= e-nth (block i t) x if x < Suc (t + 1) for x
 proof (cases x = ?k)
  case True
```

```
    have gap i (Suc t) = e-length ?b and b: block i (Suc t) = e-snoc ?b 0
      using gap-Suc(1) block-Suc(1) assms by simp-all
    then have x < e-length (block i (Suc t)) x ≠ gap i (Suc t)
      using that length-block by simp-all
    then have τ i x ↓= e-nth (block i (Suc t)) x
      using tau-eq-state by simp
    then show ?thesis using that assms b by (simp add: nth-append)
  next
    case False
    then show ?thesis using that assms tau-eq-state' by simp
  qed
qed

lemma change-on-0-hyp-change:
  assumes change-on-0 (block i t) (gap i t)
  shows σ i ((τ i) ▷ (t + 1)) ≠ σ i ((τ i) ▷ (gap i t − 1))
  using assms hd-block init-tau-gap change-on-0-init-tau by simp

lemma change-on-1-init-tau:
  assumes change-on-1 (block i t) (gap i t)
  shows (τ i) ▷ (t + 1) = e-update (block i t) (gap i t) 1
proof (intro initI')
  let ?b = block i t and ?k = gap i t
  show e-length (e-update ?b ?k 1) = Suc (t + 1)
    using length-block by simp
  show (τ i) x ↓= e-nth (e-update ?b ?k 1) x if x < Suc (t + 1) for x
  proof (cases x = ?k)
    case True
    have gap i (Suc t) = e-length ?b and b: block i (Suc t) = e-snoc (e-update ?b ?k 1) 0
      using gap-Suc(2) block-Suc(2) assms by simp-all
    then have x < e-length (block i (Suc t)) x ≠ gap i (Suc t)
      using that length-block by simp-all
    then have τ i x ↓= e-nth (block i (Suc t)) x
      using tau-eq-state by simp
    then show ?thesis using that assms b nth-append by (simp add: nth-append)
  next
    case False
    then show ?thesis using that assms tau-eq-state' by simp
  qed
qed

lemma change-on-1-hyp-change:
  assumes change-on-1 (block i t) (gap i t)
  shows σ i ((τ i) ▷ (t + 1)) ≠ σ i ((τ i) ▷ (gap i t − 1))
  using assms hd-block init-tau-gap change-on-1-init-tau by simp

lemma change-on-either-hyp-change:
  assumes ¬ change-on-neither (block i t) (gap i t)
  shows σ i ((τ i) ▷ (t + 1)) ≠ σ i ((τ i) ▷ (gap i t − 1))
  using assms change-on-0-hyp-change change-on-1-hyp-change by auto

lemma filled-gap-0-init-tau:
  assumes f0 = (τ i)((gap i t):=Some 0)
  shows f0 ▷ (t + 1) = block i t
proof (intro initI')
  show len: e-length (block i t) = Suc (t + 1)
```

```
    using assms length-block by auto
  show f₀ x ↓= e-nth (block i t) x if x < Suc (t + 1) for x
  proof (cases x = gap i t)
    case True
    then show ?thesis using assms last-block by auto
  next
    case False
    then show ?thesis using assms len tau-eq-state that by auto
  qed
qed

lemma filled-gap-1-init-tau:
  assumes f₁ = (τ i)((gap i t):=Some 1)
  shows f₁ ▷ (t + 1) = e-update (block i t) (gap i t) 1
proof (intro initI')
  show len: e-length (e-update (block i t) (gap i t) 1) = Suc (t + 1)
    using e-length-update length-block by simp
  show f₁ x ↓= e-nth (e-update (block i t) (gap i t) 1) x if x < Suc (t + 1) for x
  proof (cases x = gap i t)
    case True
    moreover have gap i t < e-length (block i t)
      using gap-in-block by simp
    ultimately show ?thesis using assms by simp
  next
    case False
    then show ?thesis using assms len tau-eq-state that by auto
  qed
qed
```

2.9.3 The separating class

Next we define the sets V_i from the introductory proof sketch (page 193).

```
definition V-bclim :: nat ⇒ partial1 set where
  V-bclim i ≡
    if case-two i
    then let k = gap i (LEAST t. ∀ T. gap i T ≤ gap i t)
         in {(τ i)(k:=Some 0), (τ i)(k:=Some 1)}
    else {τ i}

lemma V-subseteq-R1: V-bclim i ⊆ ℛ
proof (cases case-two i)
  case True
  define k where k = gap i (LEAST t. ∀ T. gap i T ≤ gap i t)
  have τ i ∈ 𝒫
    using tau-in-P2 P2-proj-P1 by auto
  then have (τ i)(k:=Some 0) ∈ 𝒫 and (τ i)(k:=Some 1) ∈ 𝒫
    using P1-update-P1 by simp-all
  moreover have total1 ((τ i)(k:=Some v)) for v
     using case-two-tau-almost-total(2)[OF True] k-def total1-def by simp
  ultimately have (τ i)(k:=Some 0) ∈ ℛ and (τ i)(k:=Some 1) ∈ ℛ
    using P1-total-imp-R1 by simp-all
  moreover have V-bclim i = {(τ i)(k:=Some 0), (τ i)(k:=Some 1)}
    using True V-bclim-def k-def by (simp add: Let-def)
  ultimately show ?thesis by simp
next
```

```
  case False
  have V-bclim i = {τ i}
    unfolding V-bclim-def by (simp add: False)
  moreover have τ i ∈ ℛ
    using total1I case-one-tau-total[OF False] tau-in-P2 P2-proj-P1[of τ] P1-total-imp-R1
    by simp
  ultimately show ?thesis by simp
qed

lemma case-one-imp-gap-unbounded:
  assumes case-one i
  shows ∃ t. gap i t − 1 > n
proof (induction n)
  case 0
  then show ?case
    using assms gap-gr0[of i] state-at-0(2)[of i] by (metis diff-is-0-eq gr-zeroI)
next
  case (Suc n)
  then obtain t where t: gap i t − 1 > n
    by auto
  moreover from assms have ∀ t. ∃ T. gap i T > gap i t
    using leI by blast
  ultimately obtain T where gap i T > gap i t
    by auto
  then have gap i T − 1 > gap i t − 1
    using gap-gr0[of i] by (simp add: Suc-le-eq diff-less-mono)
  with t have gap i T − 1 > Suc n by simp
  then show ?case by auto
qed

lemma case-one-imp-not-learn-lim-V:
  assumes case-one i
  shows ¬ learn-lim φ (V-bclim i) (σ i)
proof −
  have V-bclim: V-bclim i = {τ i}
    using assms V-bclim-def by (auto simp add: Let-def)
  have ∃ m1>n. ∃ m2>n. (σ i) ((τ i) ▹ m1) ≠ (σ i) ((τ i) ▹ m2) for n
  proof −
    obtain t where t: gap i t − 1 > n
      using case-one-imp-gap-unbounded[OF assms] by auto
    moreover have ∀ t. ∃ T≥t. ¬ change-on-neither (block i T) (gap i T)
      using assms case-two-iff-forever-neither by blast
    ultimately obtain T where T: T ≥ t ¬ change-on-neither (block i T) (gap i T)
      by auto
    then have (σ i) ((τ i) ▹ (T + 1)) ≠ (σ i) ((τ i) ▹ (gap i T − 1))
      using change-on-either-hyp-change by simp
    moreover have gap i T − 1 > n
      using t T(1) gap-monotone by (simp add: diff-le-mono less-le-trans)
    moreover have T + 1 > n
    proof −
      have gap i T − 1 ≤ T
        using gap-in-block length-block by (simp add: le-diff-conv less-Suc-eq-le)
      then show ?thesis using ‹gap i T − 1 > n› by simp
    qed
    ultimately show ?thesis by auto
  qed
```

```
  with infinite-hyp-changes-not-Lim V-bclim show ?thesis by simp
qed

lemma case-two-imp-not-learn-lim-V:
  assumes case-two i
  shows ¬ learn-lim φ (V-bclim i) (σ i)
proof −
  let ?P = λt. ∀ T. (gap i T) ≤ (gap i t)
  let ?t = LEAST t. ?P t
  let ?k = gap i ?t
  let ?b = e-take ?k (block i ?t)
  have t: ∀ T. gap i T ≤ gap i ?t
    using assms LeastI-ex[of ?P] by simp
  then have neither: ∀ T≥?t. change-on-neither (block i T) (gap i T)
    using gap-le-Suc gap-monotone state-change-on-neither(1)
    by (metis (no-types, lifting) antisym)
  have gap-T: ∀ T≥?t. gap i T = ?k
    using t gap-monotone antisym-conv by blast
  define f_0 where f_0 = (τ i)(?k:=Some 0)
  define f_1 where f_1 = (τ i)(?k:=Some 1)
  show ?thesis
  proof (rule same-hyp-for-two-not-Lim)
    show f_0 ∈ V-bclim i and f_1 ∈ V-bclim i
      using assms V-bclim-def f_0-def f_1-def by (simp-all add: Let-def)
    show f_0 ≠ f_1 using f_0-def f_1-def by (meson map-upd-eqD1 zero-neq-one)
    show ∀ n≥Suc ?t. σ i (f_0 ▹ n) = σ i ?b
    proof −
      have σ i (block i T) = σ i (e-take ?k (block i T)) if T ≥ ?t for T
        using that gap-T neither hd-block by metis
      then have σ i (block i T) = σ i ?b if T ≥ ?t for T
        by (metis (no-types, lifting) init-tau-gap gap-T that)
      then have σ i (f_0 ▹ (T + 1)) = σ i ?b if T ≥ ?t for T
        using filled-gap-0-init-tau[of f_0 i T] f_0-def gap-T that
        by (metis (no-types, lifting))
      then have σ i (f_0 ▹ T) = σ i ?b if T ≥ Suc ?t for T
        using that by (metis (no-types, lifting) Suc-eq-plus1 Suc-le-D Suc-le-mono)
      then show ?thesis by simp
    qed
    show ∀ n≥Suc ?t. σ i (f_1 ▹ n) = σ i ?b
    proof −
      have σ i (e-update (block i T) ?k 1) = σ i (e-take ?k (block i T)) if T ≥ ?t for T
        using neither by (metis (no-types, lifting) hd-block gap-T that)
      then have σ i (e-update (block i T) ?k 1) = σ i ?b if T ≥ ?t for T
        using that init-tau-gap[of i] gap-T by (metis (no-types, lifting))
      then have σ i (f_1 ▹ (T + 1)) = σ i ?b if T ≥ ?t for T
        using filled-gap-1-init-tau[of f_1 i T] f_1-def gap-T that
        by (metis (no-types, lifting))
      then have σ i (f_1 ▹ T) = σ i ?b if T ≥ Suc ?t for T
        using that by (metis (no-types, lifting) Suc-eq-plus1 Suc-le-D Suc-le-mono)
      then show ?thesis by simp
    qed
  qed
qed

corollary not-learn-lim-V: ¬ learn-lim φ (V-bclim i) (σ i)
  using case-one-imp-not-learn-lim-V case-two-imp-not-learn-lim-V
```

```
  by (cases case-two i) simp-all
```

Next we define the separating class.

```
definition V-BCLIM :: partial1 set (‹V_BC−LIM›) where
  V_BC−LIM ≡ ⋃i. V-bclim i

lemma V-BCLIM-R1: V_BC−LIM ⊆ ℛ
  using V-BCLIM-def V-subseteq-R1 by auto

lemma V-BCLIM-not-in-Lim: V_BC−LIM ∉ LIM
proof
  assume V_BC−LIM ∈ LIM
  then obtain s where s: learn-lim φ V_BC−LIM s
    using learn-lim-wrt-goedel[OF goedel-numbering-phi] Lim-def by blast
  moreover obtain i where φ i = s
    using s learn-limE(1) phi-universal by blast
  ultimately have learn-lim φ V_BC−LIM (λx. eval r-sigma [i, x])
    using learn-lim-sigma by simp
  moreover have V-bclim i ⊆ V_BC−LIM
    using V-BCLIM-def by auto
  ultimately have learn-lim φ (V-bclim i) (λx. eval r-sigma [i, x])
    using learn-lim-closed-subseteq by simp
  then show False
    using not-learn-lim-V by simp
qed
```

2.9.4 The separating class is in BC

In order to show $V_{BC-LIM} \in BC$ we define a hypothesis space that for every function τ_i and every list b of numbers contains a copy of τ_i with the first $|b|$ values replaced by b.

```
definition psitau :: partial2 (‹ψ^τ›) where
  ψ^τ b x ≡ (if x < e-length b then Some (e-nth b x) else τ (e-hd b) x)

lemma psitau-in-P2: ψ^τ ∈ 𝒫²
proof −
  define r where r ≡
    Cn 2
     (r-lifz r-nth (Cn 2 r-tau [Cn 2 r-hd [Id 2 0], Id 2 1]))
     [Cn 2 r-less [Id 2 1, Cn 2 r-length [Id 2 0]], Id 2 0, Id 2 1]
  then have recfn 2 r
    using r-tau-recfn by simp
  moreover have eval r [b, x] = ψ^τ b x for b x
  proof −
    let ?f = Cn 2 r-tau [Cn 2 r-hd [Id 2 0], Id 2 1]
    have recfn 2 r-nth recfn 2 ?f
      using r-tau-recfn by simp-all
    then have eval (r-lifz r-nth ?f) [c, b, x] =
        (if c = 0 then eval r-nth [b, x] else eval ?f [b, x]) for c
      by simp
    moreover have eval r-nth [b, x] ↓= e-nth b x
      by simp
    moreover have eval ?f [b, x] = τ (e-hd b) x
      using r-tau-recfn by simp
    ultimately have eval (r-lifz r-nth ?f) [c, b, x] =
```

(*if c* = *0 then Some* (*e-nth b x*) *else* τ (*e-hd b*) *x*) **for** *c*
by *simp*
moreover have *eval* (*Cn 2 r-less* [*Id 2 1*, *Cn 2 r-length* [*Id 2 0*]]) [*b*, *x*] ↓=
(*if x* < *e-length b then 0 else 1*)
by *simp*
ultimately show *?thesis*
unfolding *r-def psitau-def* **using** *r-tau-recfn* **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *psitau-init*:
ψ^τ (*f* ▷ *n*) *x* = (*if x* < *Suc n then Some* (*the* (*f x*)) *else* τ (*the* (*f 0*)) *x*)
proof −
let *?e* = *f* ▷ *n*
have *e-length ?e* = *Suc n* **by** *simp*
moreover have *x* < *Suc n* ⟹ *e-nth ?e x* = *the* (*f x*) **by** *simp*
moreover have *e-hd ?e* = *the* (*f 0*)
using *hd-init* **by** *simp*
ultimately show *?thesis* **using** *psitau-def* **by** *simp*
qed

The class V_{BC-LIM} can be learned BC-style in the hypothesis space ψ^τ by the identity function.

lemma *learn-bc-V-BCLIM*: *learn-bc* ψ^τ V_{BC-LIM} *Some*
proof (*rule learn-bcI*)
show *environment* ψ^τ V_{BC-LIM} *Some*
using *identity-in-R1 V-BCLIM-R1 psitau-in-P2* **by** *auto*
show $\exists n_0. \forall n \geq n_0.$ ψ^τ (*the* (*Some* (*f* ▷ *n*))) = *f* **if** *f* ∈ V_{BC-LIM} **for** *f*
proof −
from *that V-BCLIM-def* **obtain** *i* **where** *i*: *f* ∈ *V-bclim i*
by *auto*
show *?thesis*
proof (*cases case-two i*)
case *True*
let *?P* = λ*t*. ∀ *T*. (*gap i T*) ≤ (*gap i t*)
let *?lmin* = *LEAST t*. *?P t*
define *k* **where** *k* ≡ *gap i ?lmin*
have *V-bclim*: *V-bclim i* = {(τ *i*)(*k*:=*Some 0*), (τ *i*)(*k*:=*Some 1*)}
using *True V-bclim-def k-def* **by** (*simp add*: *Let-def*)
moreover have *0* < *k*
using *gap-gr0*[*of i*] *k-def* **by** *simp*
ultimately have *f 0* ↓= *i*
using *tau-at-0*[*of i*] *i* **by** *auto*
have ψ^τ (*f* ▷ *n*) = *f* **if** *n* ≥ *k* **for** *n*
proof
fix *x*
show ψ^τ (*f* ▷ *n*) *x* = *f x*
proof (*cases x* ≤ *n*)
case *True*
then show *?thesis*
using *R1-imp-total1 V-subseteq-R1 i psitau-init* **by** *fastforce*
next
case *False*
then have ψ^τ (*f* ▷ *n*) *x* = τ (*the* (*f 0*)) *x*
using *psitau-init* **by** *simp*

```
        then have ψ^τ (f ▷ n) x = τ i x
          using ‹f 0 ↓= i› by simp
        moreover have f x = τ i x
          using False V-bclim i that by auto
        ultimately show ?thesis by simp
      qed
    qed
    then show ?thesis by auto
  next
    case False
    then have V-bclim i = {τ i}
      using V-bclim-def by (auto simp add: Let-def)
    then have f: f = τ i
      using i by simp
    have ψ^τ (f ▷ n) = f for n
    proof
      fix x
      show ψ^τ (f ▷ n) x = f x
      proof (cases x ≤ n)
        case True
        then show ?thesis
          using R1-imp-total1 V-BCLIM-R1 psitau-init that by auto
      next
        case False
        then show ?thesis by (simp add: f psitau-init tau-at-0)
      qed
    qed
    then show ?thesis by simp
  qed
 qed
qed
```

Finally, the main result of this section:

```
theorem Lim-subset-BC: LIM ⊂ BC
  using learn-bc-V-BCLIM BC-def Lim-subseteq-BC V-BCLIM-not-in-Lim by auto

end
```

2.10 TOTAL is a proper subset of CONS

```
theory TOTAL-CONS
  imports Lemma-R
    CP-FIN-NUM
    CONS-LIM
begin
```

We first show that TOTAL is a subset of CONS. Then we present a separating class.

2.10.1 TOTAL is a subset of CONS

A TOTAL strategy hypothesizes only total functions, for which the consistency with the input prefix is decidable. A CONS strategy can thus run a TOTAL strategy and check if its hypothesis is consistent. If so, it outputs this hypothesis, otherwise some arbitrary consistent one. Since the TOTAL strategy converges to a correct hypothesis, which is consistent, the CONS strategy will converge to the same hypothesis.

Without loss of generality we can assume that learning takes place with respect to our Gödel numbering φ. So we need to decide consistency only for this numbering.

```
abbreviation r-consist-phi where
  r-consist-phi ≡ r-consistent r-phi

lemma r-consist-phi-recfn [simp]: recfn 2 r-consist-phi
  by simp

lemma r-consist-phi:
  assumes ∀ k<e-length e. φ i k ↓
  shows eval r-consist-phi [i, e] ↓=
    (if ∀ k<e-length e. φ i k ↓= e-nth e k then 0 else 1)
proof −
  have ∀ k<e-length e. eval r-phi [i, k] ↓
    using assms phi-def by simp
  moreover have recfn 2 r-phi by simp
  ultimately have eval (r-consistent r-phi) [i, e] ↓=
     (if ∀ k<e-length e. eval r-phi [i, k] ↓= e-nth e k then 0 else 1)
    using r-consistent-converg assms by simp
  then show ?thesis using phi-def by simp
qed

lemma r-consist-phi-init:
  assumes f ∈ ℛ and φ i ∈ ℛ
  shows eval r-consist-phi [i, f ▷ n] ↓= (if ∀ k≤n. φ i k = f k then 0 else 1)
  using assms r-consist-phi R1-imp-total1 total1E by (simp add: r-consist-phi)

lemma TOTAL-subseteq-CONS: TOTAL ⊆ CONS
proof
  fix U assume U ∈ TOTAL
  then have U ∈ TOTAL-wrt φ
    using TOTAL-wrt-phi-eq-TOTAL by blast
  then obtain t' where t': learn-total φ U t'
    using TOTAL-wrt-def by auto
  then obtain t where t: recfn 1 t ⋀x. eval t [x] = t' x
    using learn-totalE(1) P1E by blast
  then have t-converg: eval t [f ▷ n] ↓ if f ∈ U for f n
    using t' learn-totalE(1) that by auto

  define s where s ≡ Cn 1 r-ifz [Cn 1 r-consist-phi [t, Id 1 0], t, r-auxhyp]
  then have recfn 1 s
    using r-consist-phi-recfn r-auxhyp-prim t(1) by simp

  have consist: eval r-consist-phi [the (eval t [f ▷ n]), f ▷ n] ↓=
     (if ∀ k≤n. φ (the (eval t [f ▷ n])) k = f k then 0 else 1)
    if f ∈ U for f n
  proof −
    have eval r-consist-phi [the (eval t [f ▷ n]), f ▷ n] =
        eval (Cn 1 r-consist-phi [t, Id 1 0]) [f ▷ n]
      using that t-converg t(1) by simp
    also have ... ↓= (if ∀ k≤n. φ (the (eval t [f ▷ n])) k = f k then 0 else 1)
    proof −
      from that have f ∈ ℛ
        using learn-totalE(1) t' by blast
      moreover have φ (the (eval t [f ▷ n])) ∈ ℛ
```

using *t′ t learn-totalE t-converg that* **by** *simp*
ultimately show *?thesis*
using *r-consist-phi-init t-converg t(1) that* **by** *simp*
qed
finally show *?thesis* **.**
qed

have *s-eq-t*: $eval\ s\ [f \triangleright n] = eval\ t\ [f \triangleright n]$
if $\forall k \le n.\ \varphi\ (the\ (eval\ t\ [f \triangleright n]))\ k = f\ k$ **and** $f \in U$ **for** *f n*
using *that consist s-def t r-auxhyp-prim prim-recfn-total*
by *simp*

have *s-eq-aux*: $eval\ s\ [f \triangleright n] = eval\ r\text{-}auxhyp\ [f \triangleright n]$
if $\neg\ (\forall k \le n.\ \varphi\ (the\ (eval\ t\ [f \triangleright n]))\ k = f\ k)$ **and** $f \in U$ **for** *f n*
proof −
from *that* **have** $eval\ r\text{-}consist\text{-}phi\ [the\ (eval\ t\ [f \triangleright n]),\ f \triangleright n] \downarrow= 1$
using *consist* **by** *simp*
moreover have $t'\ (f \triangleright n) \downarrow$ **using** *t′ learn-totalE(1) that(2)* **by** *blast*
ultimately show *?thesis*
using *s-def t r-auxhyp-prim t′ learn-totalE* **by** *simp*
qed

have *learn-cons* $\varphi\ U\ (\lambda e.\ eval\ s\ [e])$
proof (*rule learn-consI*)
have $eval\ s\ [f \triangleright n] \downarrow$ **if** $f \in U$ **for** *f n*
using *that t-converg[OF that, of n] s-eq-t[of n f] prim-recfn-total[of r-auxhyp 1]*
r-auxhyp-prim s-eq-aux[OF - that, of n] totalE
by *fastforce*
then show *environment* $\varphi\ U\ (\lambda e.\ eval\ s\ [e])$
using *t′ ‹recfn 1 s› learn-totalE(1)* **by** *blast*
show $\exists i.\ \varphi\ i = f \wedge (\forall^\infty n.\ eval\ s\ [f \triangleright n] \downarrow= i)$ **if** $f \in U$ **for** *f*
proof −
from *that t′ t learn-totalE* **obtain** $i\ n_0$ **where**
i-n0: $\varphi\ i = f \wedge (\forall n \ge n_0.\ eval\ t\ [f \triangleright n] \downarrow= i)$
by *metis*
then have $\bigwedge n.\ n \ge n_0 \Longrightarrow \forall k \le n.\ \varphi\ (the\ (eval\ t\ [f \triangleright n]))\ k = f\ k$
by *simp*
with *s-eq-t* **have** $\bigwedge n.\ n \ge n_0 \Longrightarrow eval\ s\ [f \triangleright n] = eval\ t\ [f \triangleright n]$
using *that* **by** *simp*
with *i-n0* **have** $\bigwedge n.\ n \ge n_0 \Longrightarrow eval\ s\ [f \triangleright n] \downarrow= i$
by *auto*
with *i-n0* **show** *?thesis* **by** *auto*
qed
show $\forall k \le n.\ \varphi\ (the\ (eval\ s\ [f \triangleright n]))\ k = f\ k$ **if** $f \in U$ **for** *f n*
proof (*cases* $\forall k \le n.\ \varphi\ (the\ (eval\ t\ [f \triangleright n]))\ k = f\ k$)
case *True*
with *that s-eq-t* **show** *?thesis* **by** *simp*
next
case *False*
then have $eval\ s\ [f \triangleright n] = eval\ r\text{-}auxhyp\ [f \triangleright n]$
using *that s-eq-aux* **by** *simp*
moreover have $f \in \mathcal{R}$
using *learn-totalE(1)[OF t′] that* **by** *auto*
ultimately show *?thesis* **using** *r-auxhyp* **by** *simp*
qed
qed

then show $U \in \mathit{CONS}$ **using** *CONS-def* **by** *auto*
qed

2.10.2 The separating class

Definition of the class

The class that will be shown to be in $\mathit{CONS} - \mathit{TOTAL}$ is the union of the following two classes.

definition *V-constotal-1* :: *partial1 set* **where**
$\mathit{V\text{-}constotal\text{-}1} \equiv \{f.\ \exists j\ p.\ f = [j] \odot p \land j \geq 2 \land p \in \mathcal{R}_{01} \land \varphi\ j = f\}$

definition *V-constotal-2* :: *partial1 set* **where**
$\mathit{V\text{-}constotal\text{-}2} \equiv$
$\{f.\ \exists j\ a\ k.$
$f = j \# a @ [k] \odot 0^\infty \land$
$j \geq 2 \land$
$(\forall i<\mathit{length}\ a.\ a\ !\ i \leq 1) \land$
$k \geq 2 \land$
$\varphi\ j = j \# a \odot \uparrow^\infty \land$
$\varphi\ k = f\}$

definition *V-constotal* :: *partial1 set* **where**
$\mathit{V\text{-}constotal} \equiv \mathit{V\text{-}constotal\text{-}1} \cup \mathit{V\text{-}constotal\text{-}2}$

lemma *V-constotal-2I*:
assumes $f = j \# a @ [k] \odot 0^\infty$
and $j \geq 2$
and $\forall i<\mathit{length}\ a.\ a\ !\ i \leq 1$
and $k \geq 2$
and $\varphi\ j = j \# a \odot \uparrow^\infty$
and $\varphi\ k = f$
shows $f \in \mathit{V\text{-}constotal\text{-}2}$
using *assms V-constotal-2-def* **by** *blast*

lemma *V-subseteq-R1*: $\mathit{V\text{-}constotal} \subseteq \mathcal{R}$
proof
fix f **assume** $f \in \mathit{V\text{-}constotal}$
then have $f \in \mathit{V\text{-}constotal\text{-}1} \lor f \in \mathit{V\text{-}constotal\text{-}2}$
using *V-constotal-def* **by** *auto*
then show $f \in \mathcal{R}$
proof
assume $f \in \mathit{V\text{-}constotal\text{-}1}$
then obtain $j\ p$ **where** $f = [j] \odot p\ p \in \mathcal{R}_{01}$
using *V-constotal-1-def* **by** *blast*
then show *?thesis* **using** *prepend-in-R1 RPred1-subseteq-R1* **by** *auto*
next
assume $f \in \mathit{V\text{-}constotal\text{-}2}$
then obtain $j\ a\ k$ **where** $f = j \# a @ [k] \odot 0^\infty$
using *V-constotal-2-def* **by** *blast*
then show *?thesis* **using** *almost0-in-R1* **by** *auto*
qed
qed

The class is in CONS

The class can be learned by the strategy *rmge2*, which outputs the rightmost value greater or equal two in the input f^n. If f is from V_1 then the strategy is correct right from the start. If f is from V_2 the strategy outputs the consistent hypothesis j until it encounters the correct hypothesis k, to which it converges.

```
lemma V-in-CONS: learn-cons φ V-constotal rmge2
proof (rule learn-consI)
  show environment φ V-constotal rmge2
    using V-subseteq-R1 rmge2-in-R1 R1-imp-total1 phi-in-P2 by simp
  have (∃ i. φ i = f ∧ (∀∞n. rmge2 (f ▷ n) ↓= i)) ∧
      (∀ n. ∀ k≤n. φ (the (rmge2 (f ▷ n))) k = f k)
    if f ∈ V-constotal for f
  proof (cases f ∈ V-constotal-1)
    case True
    then obtain j p where
      f: f = [j] ⊙ p and
      j: j ≥ 2 and
      p: p ∈ ℛ01 and
      phi-j: φ j = f
      using V-constotal-1-def by blast
    then have f 0 ↓= j by (simp add: prepend-at-less)
    then have f-at-0: the (f 0) ≥ 2 by (simp add: j)
    have f-at-gr0: the (f x) ≤ 1 if x > 0 for x
      using that f p by (simp add: RPred1-altdef Suc-leI prepend-at-ge)
    have total1 f
      using V-subseteq-R1 that R1-imp-total1 total1-def by auto
    have rmge2 (f ▷ n) ↓= j for n
    proof −
      let ?P = λi. i < Suc n ∧ the (f i) ≥ 2
      have Greatest ?P = 0
      proof (rule Greatest-equality)
        show 0 < Suc n ∧ 2 ≤ the (f 0)
          using f-at-0 by simp
        show ⋀y. y < Suc n ∧ 2 ≤ the (f y) ⟹ y ≤ 0
          using f-at-gr0 by fastforce
      qed
      then have rmge2 (f ▷ n) = f 0
        using f-at-0 rmge2-init-total[of f n, OF ‹total1 f›] by auto
      then show rmge2 (f ▷ n) ↓= j
        by (simp add: ‹f 0 ↓= j›)
    qed
    then show ?thesis using phi-j by auto
  next
    case False
    then have f ∈ V-constotal-2
      using V-constotal-def that by auto
    then obtain j a k where jak:
      f = j # a @ [k] ⊙ 0∞
      j ≥ 2
      ∀ i<length a. a ! i ≤ 1
      k ≥ 2
      φ j = j # a ⊙ ↑∞
      φ k = f
      using V-constotal-2-def by blast
```

```
then have f-at-0: f 0 ↓= j by simp
have f-eq-a: f x ↓= a ! (x − 1) if 0 < x ∧ x < Suc (length a) for x
proof −
  have x − 1 < length a
    using that by auto
  then show ?thesis
    by (simp add: jak(1) less-SucI nth-append that)
qed
then have f-at-a: the (f x) ≤ 1 if 0 < x ∧ x < Suc (length a) for x
  using jak(3) that by auto
from jak have f-k: f (Suc (length a)) ↓= k by auto
from jak have f-at-big: f x ↓= 0 if x > Suc (length a) for x
  using that by simp
let ?P = λn i. i < Suc n ∧ the (f i) ≥ 2
have rmge2: rmge2 (f ▷ n) = f (Greatest (?P n)) for n
proof −
  have ¬ (∀ i<Suc n. the (f i) < 2) for n
    using jak(2) f-at-0 by auto
  moreover have total1 f
    using V-subseteq-R1 R1-imp-total1 that total1-def by auto
  ultimately show ?thesis using rmge2-init-total[of f n] by auto
qed
have Greatest (?P n) = 0 if n < Suc (length a) for n
proof (rule Greatest-equality)
  show 0 < Suc n ∧ 2 ≤ the (f 0)
    using that by (simp add: jak(2) f-at-0)
  show ⋀y. y < Suc n ∧ 2 ≤ the (f y) ⟹ y ≤ 0
    using that f-at-a
    by (metis Suc-1 dual-order.strict-trans leI less-Suc-eq not-less-eq-eq)
qed
with rmge2 f-at-0 have rmge2-small:
  rmge2 (f ▷ n) ↓= j if n < Suc (length a) for n
  using that by simp
have Greatest (?P n) = Suc (length a) if n ≥ Suc (length a) for n
proof (rule Greatest-equality)
  show Suc (length a) < Suc n ∧ 2 ≤ the (f (Suc (length a)))
    using that f-k by (simp add: jak(4) less-Suc-eq-le)
  show ⋀y. y < Suc n ∧ 2 ≤ the (f y) ⟹ y ≤ Suc (length a)
    using that f-at-big by (metis leI le-SucI not-less-eq-eq numeral-2-eq-2 option.sel)
qed
with rmge2 f-at-big f-k have rmge2-big:
  rmge2 (f ▷ n) ↓= k if n ≥ Suc (length a) for n
  using that by simp
then have ∃ i n_0. φ i = f ∧ (∀ n≥n_0. rmge2 (f ▷ n) ↓= i)
  using jak(6) by auto
moreover have ∀ k≤n. φ (the (rmge2 (f ▷ n))) k = f k for n
proof (cases n < Suc (length a))
  case True
  then have rmge2 (f ▷ n) ↓= j
    using rmge2-small by simp
  then have φ (the (rmge2 (f ▷ n))) = φ j by simp
  with True show ?thesis
    using rmge2-small f-at-0 f-eq-a jak(5) prepend-at-less
    by (metis le-less-trans le-zero-eq length-Cons not-le-imp-less nth-Cons-0 nth-Cons-pos)
next
  case False
```

then show *?thesis* **using** *rmge2-big jak* **by** *simp*
qed
ultimately show *?thesis* **by** *simp*
qed
then show $\bigwedge f.\ f \in \textit{V-constotal} \Longrightarrow \exists\, i.\ \varphi\ i = f \wedge (\forall^\infty n.\ \textit{rmge2}\ (f \triangleright n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in \textit{V-constotal} \Longrightarrow \forall\, k{\le}n.\ \varphi\ (\textit{the}\ (\textit{rmge2}\ (f \triangleright n)))\ k = f\ k$
by *simp-all*
qed

The class is not in TOTAL

Recall that V is the union of $V_1 = \{jp \mid j \geq 2 \wedge p \in \mathcal{R}_{01} \wedge \varphi_j = jp\}$ and $V_2 = \{jak0^\infty \mid j \geq 2 \wedge a \in \{0,1\}^* \wedge k \geq 2 \wedge \varphi_j = ja\uparrow^\infty \wedge \varphi_k = jak0^\infty\}$.

The proof is adapted from a proof of a stronger result by Freivalds, Kinber, and Wiehagen [7, Theorem 27] concerning an inference type not defined here.

The proof is by contradiction. If V was in TOTAL, there would be a strategy S learning V in our standard Gödel numbering φ. By Lemma R for TOTAL we can assume S to be total.

In order to construct a function $f \in V$ for which S fails we employ a computable process iteratively building function prefixes. For every j the process builds a function ψ_j. The initial prefix is the singleton $[j]$. Given a prefix b, the next prefix is determined as follows:

1. Search for a $y \geq |b|$ with $\varphi_{S(b)}(y) \downarrow= v$ for some v.
2. Set the new prefix $b0^{y-|b|}\bar{v}$, where $\bar{v} = 1 - v$.

Step 1 can diverge, for example, if $\varphi_{S(b)}$ is the empty function. In this case ψ_j will only be defined for a finite prefix. If, however, Step 2 is reached, the prefix b is extended to a b' such that $\varphi_{S(b)}(y) \neq b'_y$, which implies $S(b)$ is a wrong hypothesis for every function starting with b', in particular for ψ_j. Since $\bar{v} \in \{0,1\}$, Step 2 only appends zeros and ones, which is important for showing membership in V.

This process defines a numbering $\psi \in \mathcal{P}^2$, and by Kleene's fixed-point theorem there is a $j \geq 2$ with $\varphi_j = \psi_j$. For this j there are two cases:

Case 1. Step 1 always succeeds. Then ψ_j is total and $\psi_j \in V_1$. But S outputs wrong hypotheses on infinitely many prefixes of ψ_j (namely every prefix constructed by the process).

Case 2. Step 1 diverges at some iteration, say when the state is $b = ja$ for some $a \in \{0,1\}^*$. Then ψ_j has the form $ja\uparrow^\infty$. The numbering χ with $\chi_k = jak0^\infty$ is in $\mathcal{P}^2$, and by Kleene's fixed-point theorem there is a $k \geq 2$ with $\varphi_k = \chi_k = jak0^\infty$. This $jak0^\infty$ is in V_2 and has the prefix ja. But Step 1 diverged on this prefix, which means there is no $y \geq |ja|$ with $\varphi_{S(ja)}(y)\downarrow$. In other words S hypothesizes a non-total function.

Thus, in both cases there is a function in V where S does not behave like a TOTAL strategy. This is the desired contradiction.

The following locale formalizes this proof sketch.

locale *total-cons* =
fixes *s* :: *partial1*
assumes *s-in-R1*: $s \in \mathcal{R}$

begin

definition *r-s* :: *recf* **where**
r-s ≡ *SOME r-s. recfn 1 r-s* ∧ *total r-s* ∧ *s* = (λ*x. eval r-s* [*x*])

lemma *rs-recfn* [*simp*]: *recfn 1 r-s*
and *rs-total* [*simp*]: ⋀*x. eval r-s* [*x*] ↓
and *eval-rs*: ⋀*x. s x* = *eval r-s* [*x*]
using *r-s-def R1-SOME*[*OF s-in-R1*, *of r-s*] **by** *simp-all*

Performing Step 1 means enumerating the domain of $\varphi_{S(b)}$ until a $y \geq |b|$ is found. The next function enumerates all domain values and checks the condition for them.

definition *r-search-enum* ≡
Cn 2 r-le [*Cn 2 r-length* [*Id 2 1*], *Cn 2 r-enumdom* [*Cn 2 r-s* [*Id 2 1*], *Id 2 0*]]

lemma *r-search-enum-recfn* [*simp*]: *recfn 2 r-search-enum*
by (*simp add*: *r-search-enum-def Let-def*)

abbreviation *search-enum* :: *partial2* **where**
search-enum x b ≡ *eval r-search-enum* [*x*, *b*]

abbreviation *enumdom* :: *partial2* **where**
enumdom i y ≡ *eval r-enumdom* [*i*, *y*]

lemma *enumdom-empty-domain*:
assumes ⋀*x.* φ *i x* ↑
shows ⋀*y. enumdom i y* ↑
using *assms r-enumdom-empty-domain* **by** (*simp add*: *phi-def*)

lemma *enumdom-nonempty-domain*:
assumes φ *i* x_0 ↓
shows ⋀*y. enumdom i y* ↓
and ⋀*x.* φ *i x* ↓ ⟷ (∃ *y. enumdom i y* ↓= *x*)
using *assms r-enumdom-nonempty-domain phi-def* **by** *metis*+

Enumerating the empty domain yields the empty function.

lemma *search-enum-empty*:
fixes *b* :: *nat*
assumes *s b* ↓= *i* **and** ⋀*x.* φ *i x* ↑
shows ⋀*x. search-enum x b* ↑
using *assms r-search-enum-def enumdom-empty-domain eval-rs* **by** *simp*

Enumerating a non-empty domain yields a total function.

lemma *search-enum-nonempty*:
fixes *b y0* :: *nat*
assumes *s b* ↓= *i* **and** φ *i* y_0 ↓ **and** *e* = *the* (*enumdom i x*)
shows *search-enum x b* ↓= (*if e-length b* ≤ *e then 0 else 1*)
proof −
let *?e* = λ*x. the* (*enumdom i x*)
let *?y* = *Cn 2 r-enumdom* [*Cn 2 r-s* [*Id 2 1*], *Id 2 0*]
have *recfn 2 ?y* **using** *assms*(*1*) **by** *simp*
moreover have ⋀*x. eval ?y* [*x*, *b*] = *enumdom i x*
using *assms*(*1*,*2*) *eval-rs* **by** *auto*
moreover from *this* **have** ⋀*x. eval ?y* [*x*, *b*] ↓
using *enumdom-nonempty-domain*(*1*)[*OF assms*(*2*)] **by** *simp*

ultimately have *eval* (*Cn 2 r-le* [*Cn 2 r-length* [*Id 2 1*], *?y*]) [*x*, *b*] ↓=
(*if e-length b* ≤ *?e x then 0 else 1*)
by *simp*
then show *?thesis* **using** *assms* **by** (*simp add*: *r-search-enum-def*)
qed

If there is a y as desired, the enumeration will eventually return zero (representing "true").

lemma *search-enum-nonempty-eq0*:
fixes *b y* :: *nat*
assumes *s b* ↓= *i* **and** *φ i y* ↓ **and** *y* ≥ *e-length b*
shows ∃ *x*. *search-enum x b* ↓= *0*
proof −
obtain *x* **where** *x*: *enumdom i x* ↓= *y*
using *enumdom-nonempty-domain*(*2*)[*OF assms*(*2*)] *assms*(*2*) **by** *auto*
from *assms*(*2*) **have** *φ i y* ↓ **by** *simp*
with *x* **have** *search-enum x b* ↓= *0*
using *search-enum-nonempty*[**where** *?e=y*] *assms* **by** *auto*
then show *?thesis* **by** *auto*
qed

If there is no y as desired, the enumeration will never return zero.

lemma *search-enum-nonempty-neq0*:
fixes *b y0* :: *nat*
assumes *s b* ↓= *i*
and *φ i* y_0 ↓
and ¬ (∃ *y*. *φ i y* ↓ ∧ *y* ≥ *e-length b*)
shows ¬ (∃ *x*. *search-enum x b* ↓= *0*)
proof
assume ∃ *x*. *search-enum x b* ↓= *0*
then obtain *x* **where** *x*: *search-enum x b* ↓= *0*
by *auto*
obtain *y* **where** *y*: *enumdom i x* ↓= *y*
using *enumdom-nonempty-domain*[*OF assms*(*2*)] **by** *blast*
then have *search-enum x b* ↓= (*if e-length b* ≤ *y then 0 else 1*)
using *assms*(*1*−*2*) *search-enum-nonempty* **by** *simp*
with *x* **have** *e-length b* ≤ *y*
using *option.inject* **by** *fastforce*
moreover have *φ i y* ↓
using *assms*(*2*) *enumdom-nonempty-domain*(*2*) *y* **by** *blast*
ultimately show *False* **using** *assms*(*3*) **by** *force*
qed

The next function corresponds to Step 1. Given a prefix b it computes a $y \geq |b|$ with $\varphi_{S(b)}(y)\downarrow$ if such a y exists; otherwise it diverges.

definition *r-search* ≡ *Cn 1 r-enumdom* [*r-s*, *Mn 1 r-search-enum*]

lemma *r-search-recfn* [*simp*]: *recfn 1 r-search*
using *r-search-def* **by** *simp*

abbreviation *search* :: *partial1* **where**
search b ≡ *eval r-search* [*b*]

If $\varphi_{S(b)}$ is the empty function, the search process diverges because already the enumeration of the domain diverges.

lemma *search-empty*:
  assumes $s\ b\ {\downarrow}{=}\ i$ **and** $\bigwedge x.\ \varphi\ i\ x\ \uparrow$
  shows $search\ b\ \uparrow$
proof −
  have $\bigwedge x.\ search\text{-}enum\ x\ b\ \uparrow$
    using *search-enum-empty*[*OF assms*] **by** *simp*
  then have $eval\ (Mn\ 1\ r\text{-}search\text{-}enum)\ [b]\ \uparrow$ **by** *simp*
  then show $search\ b\ \uparrow$ **unfolding** *r-search-def* **by** *simp*
qed

If $\varphi_{S(b)}$ is non-empty, but there is no y with the desired properties, the search process diverges.

lemma *search-nonempty-neq0*:
  fixes *b y0* :: *nat*
  assumes $s\ b\ {\downarrow}{=}\ i$
    and $\varphi\ i\ y_0\ \downarrow$
    and $\neg\ (\exists\ y.\ \varphi\ i\ y\ \downarrow \wedge\ y \geq e\text{-}length\ b)$
  shows $search\ b\ \uparrow$
proof −
  have $\neg\ (\exists\ x.\ search\text{-}enum\ x\ b\ {\downarrow}{=}\ 0)$
    using *assms search-enum-nonempty-neq0* **by** *simp*
  moreover have *recfn 1* (*Mn 1 r-search-enum*)
    by (*simp add*: *assms*(*1*))
  ultimately have $eval\ (Mn\ 1\ r\text{-}search\text{-}enum)\ [b]\ \uparrow$ **by** *simp*
  then show *?thesis* **using** *r-search-def* **by** *auto*
qed

If there is a y as desired, the search process will return one such y.

lemma *search-nonempty-eq0*:
  fixes *b y* :: *nat*
  assumes $s\ b\ {\downarrow}{=}\ i$ **and** $\varphi\ i\ y\ \downarrow$ **and** $y \geq e\text{-}length\ b$
  shows $search\ b\ \downarrow$
    and $\varphi\ i\ (the\ (search\ b))\ \downarrow$
    and $the\ (search\ b) \geq e\text{-}length\ b$
proof −
  have $\exists\ x.\ search\text{-}enum\ x\ b\ {\downarrow}{=}\ 0$
    using *assms search-enum-nonempty-eq0* **by** *simp*
  moreover have $\forall\ x.\ search\text{-}enum\ x\ b\ \downarrow$
    using *assms search-enum-nonempty* **by** *simp*
  moreover have *recfn 1* (*Mn 1 r-search-enum*)
    by *simp*
  ultimately have
    1: $search\text{-}enum\ (the\ (eval\ (Mn\ 1\ r\text{-}search\text{-}enum)\ [b]))\ b\ {\downarrow}{=}\ 0$ **and**
    2: $eval\ (Mn\ 1\ r\text{-}search\text{-}enum)\ [b]\ \downarrow$
    using *eval-Mn-diverg eval-Mn-convergE*[*of 1 r-search-enum* [*b*]]
    by (*metis* (*no-types*, *lifting*) *One-nat-def length-Cons list.size*(*3*) *option.collapse*,
      metis (*no-types*, *lifting*) *One-nat-def length-Cons list.size*(*3*))
  let $?x = the\ (eval\ (Mn\ 1\ r\text{-}search\text{-}enum)\ [b])$
  have $search\ b = eval\ (Cn\ 1\ r\text{-}enumdom\ [r\text{-}s,\ Mn\ 1\ r\text{-}search\text{-}enum])\ [b]$
    unfolding *r-search-def* **by** *simp*
  then have *3*: $search\ b = enumdom\ i\ ?x$
    using *assms 2 eval-rs* **by** *simp*
  then have $the\ (search\ b) = the\ (enumdom\ i\ ?x)$ (**is** *?y* = -)
    by *simp*
  then have *4*: $search\text{-}enum\ ?x\ b\ {\downarrow}{=}\ (if\ e\text{-}length\ b \leq\ ?y\ then\ 0\ else\ 1)$

using *search-enum-nonempty assms* **by** *simp*
from *3* **have** φ *i ?y* ↓
using *enumdom-nonempty-domain assms*(*2*) **by** (*metis option.collapse*)
then show φ *i ?y* ↓
using *phi-def* **by** *simp*
then show *?y* ≥ *e-length b*
using *assms 4 1 option.inject* **by** *fastforce*
show *search b* ↓
using *3 assms*(*2*) *enumdom-nonempty-domain*(*1*) **by** *auto*
qed

The converse of the previous lemma states that whenever the search process returns a value it will be one with the desired properties.

lemma *search-converg*:
assumes *s b* ↓= *i* **and** *search b* ↓ (**is** *?y* ↓)
shows φ *i* (*the ?y*) ↓
and *the ?y* ≥ *e-length b*
proof −
have ∃ *y*. φ *i y* ↓
using *assms search-empty* **by** *meson*
then have ∃ *y*. *y* ≥ *e-length b* ∧ φ *i y* ↓
using *search-nonempty-neq0 assms* **by** *meson*
then obtain *y* **where** *y*: *y* ≥ *e-length b* ∧ φ *i y* ↓ **by** *auto*
then have φ *i y* ↓
using *phi-def* **by** *simp*
then show φ *i* (*the* (*search b*)) ↓
and (*the* (*search b*)) ≥ *e-length b*
using *y assms search-nonempty-eq0*[*OF assms*(*1*) ‹φ *i y* ↓›] **by** *simp-all*
qed

Likewise, if the search diverges, there is no appropriate y.

lemma *search-diverg*:
assumes *s b* ↓= *i* **and** *search b* ↑
shows ¬ (∃ *y*. φ *i y* ↓ ∧ *y* ≥ *e-length b*)
proof
assume ∃ *y*. φ *i y* ↓ ∧ *y* ≥ *e-length b*
then obtain *y* **where** *y*: φ *i y* ↓ *y* ≥ *e-length b*
by *auto*
from *y*(*1*) **have** φ *i y* ↓
by (*simp add*: *phi-def*)
with *y*(*2*) *search-nonempty-eq0* **have** *search b* ↓
using *assms* **by** *blast*
with *assms*(*2*) **show** *False* **by** *simp*
qed

Step 2 extends the prefix by a block of the shape $0^n\bar{v}$. The next function constructs such a block for given n and v.

definition *r-badblock* ≡
let f = *Cn 1 r-singleton-encode* [*r-not*];
g = *Cn 3 r-cons* [*r-constn 2 0*, *Id 3 1*]
in Pr 1 f g

lemma *r-badblock-prim* [*simp*]: *recfn 2 r-badblock*
unfolding *r-badblock-def* **by** *simp*

lemma *r-badblock*: *eval r-badblock* [*n*, *v*] ↓= *list-encode* (*replicate n 0* @ [*1* − *v*])
proof (*induction n*)
  case *0*
  let *?f* = *Cn 1 r-singleton-encode* [*r-not*]
  have *eval r-badblock* [*0*, *v*] = *eval ?f* [*v*]
    unfolding *r-badblock-def* **by** *simp*
  also have ... = *eval r-singleton-encode* [*the* (*eval r-not* [*v*])]
    by *simp*
  also have ... ↓= *list-encode* [*1* − *v*]
    by *simp*
  finally show *?case* **by** *simp*
next
  case (*Suc n*)
  let *?g* = *Cn 3 r-cons* [*r-constn 2 0*, *Id 3 1*]
  have *recfn 3 ?g* **by** *simp*
  have *eval r-badblock* [(*Suc n*), *v*] = *eval ?g* [*n*, *the* (*eval r-badblock* [*n* , *v*]), *v*]
    using ‹*recfn 3 ?g*› *Suc* **by** (*simp add*: *r-badblock-def*)
  also have ... = *eval ?g* [*n*, *list-encode* (*replicate n 0* @ [*1* − *v*]), *v*]
    using *Suc* **by** *simp*
  also have ... = *eval r-cons* [*0*, *list-encode* (*replicate n 0* @ [*1* − *v*])]
    by *simp*
  also have ... ↓= *e-cons 0* (*list-encode* (*replicate n 0* @ [*1* − *v*]))
    by *simp*
  also have ... ↓= *list-encode* (*0* # (*replicate n 0* @ [*1* − *v*]))
    by *simp*
  also have ... ↓= *list-encode* (*replicate* (*Suc n*) *0* @ [*1* − *v*])
    by *simp*
  finally show *?case* **by** *simp*
qed

lemma *r-badblock-only-01*: *e-nth* (*the* (*eval r-badblock* [*n*, *v*])) *i* ≤ *1*
  using *r-badblock* **by** (*simp add*: *nth-append*)

lemma *r-badblock-last*: *e-nth* (*the* (*eval r-badblock* [*n*, *v*])) *n* = *1* − *v*
  using *r-badblock* **by** (*simp add*: *nth-append*)

The following function computes the next prefix from the current one. In other words, it performs Steps 1 and 2.

definition *r-next* ≡
  Cn 1 r-append
   [*Id 1 0*,
    Cn 1 r-badblock
     [*Cn 1 r-sub* [*r-search*, *r-length*],
      Cn 1 r-phi [*r-s*, *r-search*]]]

lemma *r-next-recfn* [*simp*]: *recfn 1 r-next*
  unfolding *r-next-def* **by** *simp*

The name *next* is unavailable, so we go for *nxt*.

abbreviation *nxt* :: *partial1* **where**
  nxt b ≡ *eval r-next* [*b*]

lemma *nxt-diverg*:
  assumes *search b* ↑
  shows *nxt b* ↑

unfolding *r-next-def* **using** *assms* **by** (*simp add*: *Let-def*)

lemma *nxt-converg*:
assumes *search b* ↓= *y*
shows *nxt b* ↓=
e-append b (*list-encode* (*replicate* (*y* − *e-length b*) *0* @ [*1* − *the* (φ (*the* (*s b*)) *y*)]))
unfolding *r-next-def* **using** *assms r-badblock search-converg phi-def eval-rs*
by *fastforce*

lemma *nxt-search-diverg*:
assumes *nxt b* ↑
shows *search b* ↑
proof (*rule ccontr*)
assume *search b* ↓
then obtain *y* **where** *search b* ↓= *y* **by** *auto*
then show *False*
using *nxt-converg assms* **by** *simp*
qed

If Step 1 finds a y, the hypothesis $S(b)$ is incorrect for the new prefix.

lemma *nxt-wrong-hyp*:
assumes *nxt b* ↓= *b′* **and** *s b* ↓= *i*
shows ∃ *y*<*e-length b′*. φ *i y* ↓≠ *e-nth b′ y*
proof −
obtain *y* **where** *y*: *search b* ↓= *y*
using *assms nxt-diverg* **by** *fastforce*
then have *y-len*: *y* ≥ *e-length b*
using *assms search-converg*(*2*) **by** *fastforce*
then have *b′*: *b′* =
(*e-append b* (*list-encode* (*replicate* (*y* − *e-length b*) *0* @ [*1* − *the* (φ *i y*)])))
using *y assms nxt-converg* **by** *simp*
then have *e-nth b′ y* = *1* − *the* (φ *i y*)
using *y-len e-nth-append-big r-badblock r-badblock-last* **by** *auto*
moreover have φ *i y* ↓
using *search-converg y y-len assms*(*2*) **by** *fastforce*
ultimately have φ *i y* ↓≠ *e-nth b′ y*
by (*metis gr-zeroI less-numeral-extra*(*4*) *less-one option.sel zero-less-diff*)
moreover have *e-length b′* = *Suc y*
using *y-len e-length-append b′* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

If Step 1 diverges, the hypothesis $S(b)$ refers to a non-total function.

lemma *nxt-nontotal-hyp*:
assumes *nxt b* ↑ **and** *s b* ↓= *i*
shows ∃ *x*. φ *i x* ↑
using *nxt-search-diverg*[*OF assms*(*1*)] *search-diverg*[*OF assms*(*2*)] **by** *auto*

The process only ever extends the given prefix.

lemma *nxt-stable*:
assumes *nxt b* ↓= *b′*
shows ∀ *x*<*e-length b*. *e-nth b x* = *e-nth b′ x*
proof −
obtain *y* **where** *y*: *search b* ↓= *y*
using *assms nxt-diverg* **by** *fastforce*

then have *y* ≥ *e-length b*
 using *search-converg*(*2*) *eval-rs rs-total* **by** *fastforce*
show *?thesis*
proof (*rule allI*, *rule impI*)
 fix *x* **assume** *x* < *e-length b*
 let *?i* = *the* (*s b*)
 have *b′*: *b′* =
  (*e-append b* (*list-encode* (*replicate* (*y* − *e-length b*) *0* @ [*1* − *the* (*φ ?i y*)])))
  using *assms nxt-converg*[*OF y*] **by** *auto*
 then show *e-nth b x* = *e-nth b′ x*
  using *e-nth-append-small* ‹*x* < *e-length b*› **by** *auto*
qed
qed

The following properties of *r-next* will be used to show that some of the constructed functions are in the class *V*.

lemma *nxt-append-01*:
 assumes *nxt b* ↓= *b′*
 shows ∀ *x*. *x* ≥ *e-length b* ∧ *x* < *e-length b′* ⟶ *e-nth b′ x* = *0* ∨ *e-nth b′ x* = *1*
proof −
 obtain *y* **where** *y*: *search b* ↓= *y*
  using *assms nxt-diverg* **by** *fastforce*
 let *?i* = *the* (*s b*)
 have *b′*: *b′* = (*e-append b* (*list-encode* (*replicate* (*y* − *e-length b*) *0* @ [*1* − *the* (*φ ?i y*)])))
  (**is** *b′* = (*e-append b ?z*))
  using *assms y nxt-converg prod-encode-eq* **by** *auto*
 show *?thesis*
 proof (*rule allI*, *rule impI*)
  fix *x* **assume** *x*: *e-length b* ≤ *x* ∧ *x* < *e-length b′*
  then have *e-nth b′ x* = *e-nth ?z* (*x* − *e-length b*)
   using *b′ e-nth-append-big* **by** *blast*
  then show *e-nth b′ x* = *0* ∨ *e-nth b′ x* = *1*
   by (*metis less-one nat-less-le option.sel r-badblock r-badblock-only-01*)
 qed
qed

lemma *nxt-monotone*:
 assumes *nxt b* ↓= *b′*
 shows *e-length b* < *e-length b′*
proof −
 obtain *y* **where** *y*: *search b* ↓= *y*
  using *assms nxt-diverg* **by** *fastforce*
 let *?i* = *the* (*s b*)
 have *b′*: *b′* =
  (*e-append b* (*list-encode* (*replicate* (*y* − *e-length b*) *0* @ [*1* − *the* (*φ ?i y*)])))
  using *assms y nxt-converg prod-encode-eq* **by** *auto*
 then show *?thesis* **using** *e-length-append* **by** *auto*
qed

The next function computes the prefixes after each iteration of the process *r-next* when started with the list [*j*].

definition *r-prefixes* :: *recf* **where**
 r-prefixes ≡ *Pr 1 r-singleton-encode* (*Cn 3 r-next* [*Id 3 1*])

lemma *r-prefixes-recfn* [*simp*]: *recfn 2 r-prefixes*

unfolding *r-prefixes-def* **by** (*simp add*: *Let-def*)

abbreviation *prefixes* :: *partial2* **where**
prefixes t j ≡ *eval r-prefixes* [*t*, *j*]

lemma *prefixes-at-0*: *prefixes 0 j* ↓= *list-encode* [*j*]
unfolding *r-prefixes-def* **by** *simp*

lemma *prefixes-at-Suc*:
assumes *prefixes t j* ↓ (**is** *?b* ↓)
shows *prefixes* (*Suc t*) *j* = *nxt* (*the ?b*)
using *r-prefixes-def assms* **by** *auto*

lemma *prefixes-at-Suc′*:
assumes *prefixes t j* ↓= *b*
shows *prefixes* (*Suc t*) *j* = *nxt b*
using *r-prefixes-def assms* **by** *auto*

lemma *prefixes-prod-encode*:
assumes *prefixes t j* ↓
obtains *b* **where** *prefixes t j* ↓= *b*
using *assms surj-prod-encode* **by** *force*

lemma *prefixes-converg-le*:
assumes *prefixes t j* ↓ **and** *t′* ≤ *t*
shows *prefixes t′ j* ↓
using *r-prefixes-def assms eval-Pr-converg-le*[*of 1 - -* [*j*]]
by *simp*

lemma *prefixes-diverg-add*:
assumes *prefixes t j* ↑
shows *prefixes* (*t* + *d*) *j* ↑
using *r-prefixes-def assms eval-Pr-diverg-add*[*of 1 - -* [*j*]]
by *simp*

Many properties of *r-prefixes* can be derived from similar properties of *r-next*.

lemma *prefixes-length*:
assumes *prefixes t j* ↓= *b*
shows *e-length b* > *t*
proof (*insert assms*, *induction t arbitrary*: *b*)
case *0*
then show *?case* **using** *prefixes-at-0 prod-encode-eq* **by** *auto*
next
case (*Suc t*)
then have *prefixes t j* ↓
using *prefixes-converg-le Suc-n-not-le-n nat-le-linear* **by** *blast*
then obtain *b′* **where** *b′*: *prefixes t j* ↓= *b′*
using *prefixes-prod-encode* **by** *blast*
with *Suc* **have** *e-length b′* > *t* **by** *simp*
have *prefixes* (*Suc t*) *j* = *nxt b′*
using *b′ prefixes-at-Suc′* **by** *simp*
with *Suc* **have** *nxt b′* ↓= *b* **by** *simp*
then have *e-length b′* < *e-length b*
using *nxt-monotone* **by** *simp*
then show *?case* **using** ‹*e-length b′* > *t*› **by** *simp*
qed

lemma *prefixes-monotone*:
  assumes *prefixes t j* ↓= *b* **and** *prefixes* (*t* + *d*) *j* ↓= *b'*
  shows *e-length b* ≤ *e-length b'*
proof (*insert assms*, *induction d arbitrary*: *b'*)
  case *0*
  then show *?case* **using** *prod-encode-eq* **by** *simp*
next
  case (*Suc d*)
  moreover have *t* + *d* ≤ *t* + *Suc d* **by** *simp*
  ultimately have *prefixes* (*t* + *d*) *j* ↓
    using *prefixes-converg-le* **by** *blast*
  then obtain *b''* **where** *b''*: *prefixes* (*t* + *d*) *j* ↓= *b''*
    using *prefixes-prod-encode* **by** *blast*
  with *Suc* **have** *prefixes* (*t* + *Suc d*) *j* = *nxt b''*
    by (*simp add*: *prefixes-at-Suc'*)
  with *Suc* **have** *nxt b''* ↓= *b'* **by** *simp*
  then show *?case* **using** *nxt-monotone Suc b''* **by** *fastforce*
qed

lemma *prefixes-stable*:
  assumes *prefixes t j* ↓= *b* **and** *prefixes* (*t* + *d*) *j* ↓= *b'*
  shows ∀ *x*<*e-length b*. *e-nth b x* = *e-nth b' x*
proof (*insert assms*, *induction d arbitrary*: *b'*)
  case *0*
  then show *?case* **using** *prod-encode-eq* **by** *simp*
next
  case (*Suc d*)
  moreover have *t* + *d* ≤ *t* + *Suc d* **by** *simp*
  ultimately have *prefixes* (*t* + *d*) *j* ↓
    using *prefixes-converg-le* **by** *blast*
  then obtain *b''* **where** *b''*: *prefixes* (*t* + *d*) *j* ↓= *b''*
    using *prefixes-prod-encode* **by** *blast*
  with *Suc* **have** *prefixes* (*t* + *Suc d*) *j* = *nxt b''*
    by (*simp add*: *prefixes-at-Suc'*)
  with *Suc* **have** *b'*: *nxt b''* ↓= *b'* **by** *simp*
  show ∀ *x*<*e-length b*. *e-nth b x* = *e-nth b' x*
  proof (*rule allI*, *rule impI*)
    fix *x* **assume** *x*: *x* < *e-length b*
    then have *e-nth b x* = *e-nth b'' x*
      using *Suc b''* **by** *simp*
    moreover have *x* ≤ *e-length b''*
      using *x prefixes-monotone b'' Suc* **by** *fastforce*
    ultimately show *e-nth b x* = *e-nth b' x*
      using *b'' nxt-stable Suc b' prefixes-monotone x*
      by (*metis leD le-neq-implies-less*)
  qed
qed

lemma *prefixes-tl-only-01*:
  assumes *prefixes t j* ↓= *b*
  shows ∀ *x*>*0*. *e-nth b x* = *0* ∨ *e-nth b x* = *1*
proof (*insert assms*, *induction t arbitrary*: *b*)
  case *0*
  then show *?case* **using** *prefixes-at-0 prod-encode-eq* **by** *auto*
next

```
  case (Suc t)
  then have prefixes t j ↓
    using prefixes-converg-le Suc-n-not-le-n nat-le-linear by blast
  then obtain b′ where b′: prefixes t j ↓= b′
    using prefixes-prod-encode by blast
  show ∀x>0. e-nth b x = 0 ∨ e-nth b x = 1
  proof (rule allI, rule impI)
    fix x :: nat
    assume x: x > 0
    show e-nth b x = 0 ∨ e-nth b x = 1
    proof (cases x < e-length b′)
      case True
      then show ?thesis
        using Suc b′ prefixes-at-Suc′ nxt-stable x by metis
    next
      case False
      then show ?thesis
        using Suc.prems b′ prefixes-at-Suc′ nxt-append-01 by auto
    qed
  qed
qed

lemma prefixes-hd:
  assumes prefixes t j ↓= b
  shows e-nth b 0 = j
proof −
  obtain b′ where b′: prefixes 0 j ↓= b′
    by (simp add: prefixes-at-0)
  then have b′ = list-encode [j]
    by (simp add: prod-encode-eq prefixes-at-0)
  then have e-nth b′ 0 = j by simp
  then show e-nth b 0 = j
    using assms prefixes-stable[OF b′, of t b] prefixes-length[OF b′] by simp
qed

lemma prefixes-nontotal-hyp:
  assumes prefixes t j ↓= b
    and prefixes (Suc t) j ↑
    and s b ↓= i
  shows ∃x. φ i x ↑
  using nxt-nontotal-hyp[OF - assms(3)] assms(2) prefixes-at-Suc′[OF assms(1)] by simp
```

We now consider the two cases from the proof sketch.

```
abbreviation case-two j ≡ ∃t. prefixes t j ↑

abbreviation case-one j ≡ ¬ case-two j
```

In Case 2 there is a maximum convergent iteration because iteration 0 converges.

```
lemma case-two:
  assumes case-two j
  shows ∃t. (∀t′≤t. prefixes t′ j ↓) ∧ (∀t′>t. prefixes t′ j ↑)
proof −
  let ?P = λt. prefixes t j ↑
  define t₀ where t₀ = Least ?P
  then have ?P t₀
    using assms LeastI-ex[of ?P] by simp
```

then have *diverg*: *?P t* **if** $t \geq t_0$ **for** *t*
  using *prefixes-converg-le that* **by** *blast*
from t_0*-def* **have** *converg*: ¬ *?P t* **if** $t < t_0$ **for** *t*
  using *Least-le*[*of ?P*] *that not-less* **by** *blast*
have $t_0 > 0$
proof (*rule ccontr*)
  assume ¬ $0 < t_0$
  then have $t_0 = 0$ **by** *simp*
  with ‹*?P* t_0› *prefixes-at-0* **show** *False* **by** *simp*
qed
let *?t* = $t_0 - 1$
have ∀ *t'*≤*?t*. *prefixes t' j* ↓
  using *converg* ‹$0 < t_0$› **by** *auto*
moreover have ∀ *t'*>*?t*. *prefixes t' j* ↑
  using *diverg* **by** *simp*
ultimately show *?thesis* **by** *auto*
qed

Having completed the modelling of the process, we can now define the functions ψ_j it computes. The value $\psi_j(x)$ is computed by running *r-prefixes* until the prefix is longer than x and then taking the x-th element of the prefix.

definition *r-psi* ≡
  let f = *Cn 3 r-less* [*Id 3 2*, *Cn 3 r-length* [*Cn 3 r-prefixes* [*Id 3 0*, *Id 3 1*]]]
  in Cn 2 r-nth [*Cn 2 r-prefixes* [*Mn 2 f*, *Id 2 0*], *Id 2 1*]

lemma *r-psi-recfn*: *recfn 2 r-psi*
  unfolding *r-psi-def* **by** *simp*

abbreviation *psi* :: *partial2* (‹ψ›) **where**
  ψ j x ≡ *eval r-psi* [*j*, *x*]

lemma *psi-in-P2*: $\psi \in \mathcal{P}^2$
  using *r-psi-recfn* **by** *auto*

The values of ψ can be read off the prefixes.

lemma *psi-eq-nth-prefix*:
  assumes *prefixes t j* ↓= *b* **and** *e-length b* > *x*
  shows *ψ j x* ↓= *e-nth b x*
proof −
  let *?f* = *Cn 3 r-less* [*Id 3 2*, *Cn 3 r-length* [*Cn 3 r-prefixes* [*Id 3 0*, *Id 3 1*]]]
  let *?P* = λ*t*. *prefixes t j* ↓ ∧ *e-length* (*the* (*prefixes t j*)) > *x*
  from *assms* **have** *ex-t*: ∃ *t*. *?P t* **by** *auto*
  define t_0 **where** t_0 = *Least ?P*
  then have *?P* t_0
    using *LeastI-ex*[*OF ex-t*] **by** *simp*
  from *ex-t* **have** *not-P*: ¬ *?P t* **if** $t < t_0$ **for** *t*
    using *ex-t that Least-le*[*of ?P*] *not-le* t_0*-def* **by** *auto*

  have *?P t* **using** *assms* **by** *simp*
  with *not-P* **have** $t_0 \leq t$ **using** *leI* **by** *blast*
  then obtain b_0 **where** *b0*: *prefixes* t_0 *j* ↓= b_0
    using *assms*(*1*) *prefixes-converg-le* **by** *blast*

  have *eval ?f* [t_0, *j*, *x*] ↓= *0*
  proof −

```
    have eval (Cn 3 r-prefixes [Id 3 0, Id 3 1]) [t_0, j, x] ↓= b_0
      using b0 by simp
    then show ?thesis using ‹?P t_0› by simp
  qed
  moreover have eval ?f [t, j, x] ↓≠ 0 if t < t_0 for t
  proof −
    obtain bt where bt: prefixes t j ↓= bt
      using prefixes-converg-le[of t_0 j t] b0 ‹t < t_0› by auto
    moreover have ¬ ?P t
      using that not-P by simp
    ultimately have e-length bt ≤ x by simp
    moreover have eval (Cn 3 r-prefixes [Id 3 0, Id 3 1]) [t, j, x] ↓= bt
      using bt by simp
    ultimately show ?thesis by simp
  qed
  ultimately have eval (Mn 2 ?f) [j, x] ↓= t_0
    using eval-Mn-convergI[of 2 ?f [j, x] t_0] by simp
  then have ψ j x ↓= e-nth b_0 x
    unfolding r-psi-def using b0 by simp
  then show ?thesis
    using ‹t_0 ≤ t› assms(1) prefixes-stable[of t_0 j b_0 t − t_0 b] b0 ‹?P t_0›
    by simp
qed

lemma psi-converg-imp-prefix:
  assumes ψ j x ↓
  shows ∃ t b. prefixes t j ↓= b ∧ e-length b > x
proof −
  let ?f = Cn 3 r-less [Id 3 2, Cn 3 r-length [Cn 3 r-prefixes [Id 3 0, Id 3 1]]]
  have eval (Mn 2 ?f) [j, x] ↓
  proof (rule ccontr)
    assume ¬ eval (Mn 2 ?f) [j, x] ↓
    then have eval (Mn 2 ?f) [j, x] ↑ by simp
    then have ψ j x ↑
      unfolding r-psi-def by simp
    then show False
      using assms by simp
  qed
  then obtain t where t: eval (Mn 2 ?f) [j, x] ↓= t
    by blast
  have recfn 2 (Mn 2 ?f) by simp
  then have f-zero: eval ?f [t, j, x] ↓= 0
    using eval-Mn-convergE[OF - t]
    by (metis (no-types, lifting) One-nat-def Suc-1 length-Cons list.size(3))
  have prefixes t j ↓
  proof (rule ccontr)
    assume ¬ prefixes t j ↓
    then have prefixes t j ↑ by simp
    then have eval ?f [t, j, x] ↑ by simp
    with f-zero show False by simp
  qed
  then obtain b' where b': prefixes t j ↓= b' by auto
  moreover have e-length b' > x
  proof (rule ccontr)
    assume ¬ e-length b' > x
    then have eval ?f [t, j, x] ↓= 1
```

using *b′* **by** *simp*
with *f-zero* **show** *False* **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *psi-converg-imp-prefix′*:
assumes *ψ j x ↓*
shows *∃ t b. prefixes t j ↓= b ∧ e-length b > x ∧ ψ j x ↓= e-nth b x*
using *psi-converg-imp-prefix*[*OF assms*] *psi-eq-nth-prefix* **by** *blast*

In both Case 1 and 2, ψ_j starts with j.

lemma *psi-at-0*: *ψ j 0 ↓= j*
using *prefixes-hd prefixes-length psi-eq-nth-prefix prefixes-at-0* **by** *fastforce*

In Case 1, ψ_j is total and made up of j followed by zeros and ones, just as required by the definition of V_1.

lemma *case-one-psi-total*:
assumes *case-one j* **and** *x > 0*
shows *ψ j x ↓= 0 ∨ ψ j x ↓= 1*
proof −
obtain *b* **where** *b*: *prefixes x j ↓= b*
using *assms*(*1*) **by** *auto*
then have *e-length b > x*
using *prefixes-length* **by** *simp*
then have *ψ j x ↓= e-nth b x*
using *b psi-eq-nth-prefix* **by** *simp*
moreover have *e-nth b x = 0 ∨ e-nth b x = 1*
using *prefixes-tl-only-01*[*OF b*] *assms*(*2*) **by** *simp*
ultimately show *ψ j x ↓= 0 ∨ ψ j x ↓= 1*
by *simp*
qed

In Case 2, ψ_j is defined only for a prefix starting with j and continuing with zeros and ones. This prefix corresponds to ja from the definition of V_2.

lemma *case-two-psi-only-prefix*:
assumes *case-two j*
shows *∃ y. (∀ x. 0 < x ∧ x < y ⟶ ψ j x ↓= 0 ∨ ψ j x ↓= 1) ∧*
(∀ x ≥ y. ψ j x ↑)
proof −
obtain *t* **where**
t-le: *∀ t′≤t. prefixes t′ j ↓* **and**
t-gr: *∀ t′>t. prefixes t′ j ↑*
using *assms case-two* **by** *blast*
then obtain *b* **where** *b*: *prefixes t j ↓= b*
by *auto*
let *?y = e-length b*
have *ψ j x ↓= 0 ∨ ψ j x ↓= 1* **if** *x > 0 ∧ x < ?y* **for** *x*
using *t-le b that* **by** (*metis prefixes-tl-only-01 psi-eq-nth-prefix*)
moreover have *ψ j x ↑* **if** *x ≥ ?y* **for** *x*
proof (*rule ccontr*)
assume *ψ j x ↓*
then obtain *t′ b′* **where** *t′*: *prefixes t′ j ↓= b′* **and** *e-length b′ > x*
using *psi-converg-imp-prefix* **by** *blast*
then have *e-length b′ > ?y*

```
      using that by simp
    with t′ have t′ > t
      using prefixes-monotone b by (metis add-diff-inverse-nat leD)
    with t′ t-gr show False by simp
  qed
  ultimately show ?thesis by auto
qed

definition longest-prefix :: nat ⇒ nat where
  longest-prefix j ≡ THE y. (∀ x<y. ψ j x ↓) ∧ (∀ x≥y. ψ j x ↑)

lemma longest-prefix:
  assumes case-two j and z = longest-prefix j
  shows (∀ x<z. ψ j x ↓) ∧ (∀ x≥z. ψ j x ↑)
proof −
  let ?P = λz. (∀ x<z. ψ j x ↓) ∧ (∀ x≥z. ψ j x ↑)
  obtain y where y:
    ∀ x. 0 < x ∧ x < y ⟶ ψ j x ↓= 0 ∨ ψ j x ↓= 1
    ∀ x≥y. ψ j x ↑
    using case-two-psi-only-prefix[OF assms(1)] by auto
  have ?P (THE z. ?P z)
  proof (rule theI[of ?P y])
    show ?P y
    proof
      show ∀ x<y. ψ j x ↓
      proof (rule allI, rule impI)
        fix x assume x < y
        show ψ j x ↓
        proof (cases x = 0)
          case True
          then show ?thesis using psi-at-0 by simp
        next
          case False
          then show ?thesis using y(1) ‹x < y› by auto
        qed
      qed
      show ∀ x≥y. ψ j x ↑ using y(2) by simp
    qed
    show z = y if ?P z for z
    proof (rule ccontr, cases z < y)
      case True
      moreover assume z ≠ y
      ultimately show False
        using that ‹?P y› by auto
    next
      case False
      moreover assume z ≠ y
      then show False
        using that ‹?P y› y(2) by (meson linorder-cases order-refl)
    qed
  qed
  then have (∀ x<(THE z. ?P z). ψ j x ↓) ∧ (∀ x≥(THE z. ?P z). ψ j x ↑)
    by blast
  moreover have longest-prefix j = (THE z. ?P z)
    unfolding longest-prefix-def by simp
  ultimately show ?thesis using assms(2) by metis
```

qed

lemma *case-two-psi-longest-prefix*:
  assumes *case-two j* **and** $y = longest\text{-}prefix\ j$
  shows $(\forall x.\ 0 < x \land x < y \longrightarrow \psi\ j\ x \downarrow= 0 \lor \psi\ j\ x \downarrow= 1) \land$
    $(\forall x \geq y.\ \psi\ j\ x \uparrow)$
  using *assms longest-prefix case-two-psi-only-prefix*
  by (*metis prefixes-tl-only-01 psi-converg-imp-prefix′*)

The prefix cannot be empty because the process starts with prefix $[j]$.

lemma *longest-prefix-gr-0*:
  assumes *case-two j*
  shows $longest\text{-}prefix\ j > 0$
  using *assms case-two-psi-longest-prefix psi-at-0* **by** *force*

lemma *psi-not-divergent-init*:
  assumes $prefixes\ t\ j \downarrow= b$
  shows $(\psi\ j) \triangleright (e\text{-}length\ b - 1) = b$
proof (*intro initI*)
  show $0 < e\text{-}length\ b$
    using *assms prefixes-length* **by** *fastforce*
  show $\psi\ j\ x \downarrow= e\text{-}nth\ b\ x$ **if** $x < e\text{-}length\ b$ **for** x
    using *that assms psi-eq-nth-prefix* **by** *simp*
qed

In Case 2, the strategy S outputs a non-total hypothesis on some prefix of ψ_j.

lemma *case-two-nontotal-hyp*:
  assumes *case-two j*
  shows $\exists n{<}longest\text{-}prefix\ j.\ \neg\ total1\ (\varphi\ (the\ (s\ ((\psi\ j) \triangleright n))))$
proof −
  obtain t **where** $\forall t'{\leq}t.\ prefixes\ t'\ j \downarrow$ **and** *t-gr*: $\forall t'{>}t.\ prefixes\ t'\ j \uparrow$
    using *assms case-two* **by** *blast*
  then obtain b **where** b: $prefixes\ t\ j \downarrow= b$
    by *auto*
  moreover obtain i **where** i: $s\ b \downarrow= i$
    using *eval-rs* **by** *fastforce*
  moreover have *div*: $prefixes\ (Suc\ t)\ j \uparrow$
    using *t-gr* **by** *simp*
  ultimately have $\exists x.\ \varphi\ i\ x \uparrow$
    using *prefixes-nontotal-hyp* **by** *simp*
  then obtain x **where** $\varphi\ i\ x \uparrow$ **by** *auto*
  moreover have *init*: $\psi\ j \triangleright (e\text{-}length\ b - 1) = b$ (**is** $- \triangleright\ ?n = b$)
    using *psi-not-divergent-init*[*OF b*] **by** *simp*
  ultimately have $\varphi\ (the\ (s\ (\psi\ j \triangleright ?n)))\ x \uparrow$
    using i **by** *simp*
  then have $\neg\ total1\ (\varphi\ (the\ (s\ (\psi\ j \triangleright ?n))))$
    by *auto*
  moreover have $?n < longest\text{-}prefix\ j$
    using *case-two-psi-longest-prefix init b div psi-eq-nth-prefix*
    by (*metis length-init lessI not-le-imp-less option.simps*(*3*))
  ultimately show *?thesis* **by** *auto*
qed

Consequently, in Case 2 the strategy does not TOTAL-learn any function starting with the longest prefix of ψ_j.

```
lemma case-two-not-learn:
  assumes case-two j
    and f ∈ ℛ
    and ⋀x. x < longest-prefix j ⟹ f x = ψ j x
  shows ¬ learn-total φ {f} s
proof –
  obtain n where n:
    n < longest-prefix j
    ¬ total1 (φ (the (s (ψ j ▷ n))))
    using case-two-nontotal-hyp[OF assms(1)] by auto
  have f ▷ n = ψ j ▷ n
    using assms(3) n(1) by (intro init-eqI) auto
  with n(2) show ?thesis by (metis R1-imp-total1 learn-totalE(3) singletonI)
qed
```

In Case 1 the strategy outputs a wrong hypothesis on infinitely many prefixes of ψ_j and thus does not learn ψ_j in the limit, much less in the sense of TOTAL.

```
lemma case-one-wrong-hyp:
  assumes case-one j
  shows ∃n>k. φ (the (s ((ψ j) ▷ n))) ≠ ψ j
proof –
  have all-t: ∀t. prefixes t j ↓
    using assms by simp
  then obtain b where b: prefixes (Suc k) j ↓= b
    by auto
  then have length: e-length b > Suc k
    using prefixes-length by simp
  then have init: ψ j ▷ (e-length b − 1) = b
    using psi-not-divergent-init b by simp
  obtain i where i: s b ↓= i
    using eval-rs by fastforce
  from all-t obtain b' where b': prefixes (Suc (Suc k)) j ↓= b'
    by auto
  then have ψ j ▷ (e-length b' − 1) = b'
    using psi-not-divergent-init by simp
  moreover have ∃y<e-length b'. φ i y ↓≠ e-nth b' y
    using nxt-wrong-hyp b b' i prefixes-at-Suc by auto
  ultimately have ∃y<e-length b'. φ i y ≠ ψ j y
    using b' psi-eq-nth-prefix by auto
  then have φ i ≠ ψ j by auto
  then show ?thesis
    using init length i by (metis Suc-less-eq length-init option.sel)
qed
```

```
lemma case-one-not-learn:
  assumes case-one j
  shows ¬ learn-lim φ {ψ j} s
proof (rule infinite-hyp-wrong-not-Lim[of ψ j])
  show ψ j ∈ {ψ j} by simp
  show ∀n. ∃m>n. φ (the (s (ψ j ▷ m))) ≠ ψ j
    using case-one-wrong-hyp[OF assms] by simp
qed
```

```
lemma case-one-not-learn-V:
  assumes case-one j and j ≥ 2 and φ j = ψ j
  shows ¬ learn-lim φ V-constotal s
```

```
proof -
  have ψ j ∈ V-constotal-1
  proof -
    define p where p = (λx. (ψ j) (x + 1))
    have p ∈ 𝓡01
    proof -
      from p-def have p ∈ 𝓟
        using skip-P1[of ψ j 1] psi-in-P2 P2-proj-P1 by blast
      moreover have p x ↓= 0 ∨ p x ↓= 1 for x
        using p-def assms(1) case-one-psi-total by auto
      moreover from this have total1 p by fast
      ultimately show ?thesis using RPred1-def by auto
    qed
    moreover have ψ j = [j] ⊙ p
      by (intro prepend-eqI, simp add: psi-at-0, simp add: p-def)
    ultimately show ?thesis using assms(2,3) V-constotal-1-def by blast
  qed
  then have ψ j ∈ V-constotal using V-constotal-def by auto
  moreover have ¬ learn-lim φ {ψ j} s
    using case-one-not-learn assms(1) by simp
  ultimately show ?thesis using learn-lim-closed-subseteq by auto
qed
```

The next lemma embodies the construction of χ followed by the application of Kleene's fixed-point theorem as described in the proof sketch.

```
lemma goedel-after-prefixes:
  fixes vs :: nat list and m :: nat
  shows ∃n≥m. φ n = vs @ [n] ⊙ 0^∞
proof -
  define f :: partial1 where f ≡ vs ⊙ 0^∞
  then have f ∈ 𝓡
    using almost0-in-R1 by auto
  then obtain n where n:
    n ≥ m
    φ n = (λx. if x = length vs then Some n else f x)
    using goedel-at[of f m length vs] by auto
  moreover have φ n x = (vs @ [n] ⊙ 0^∞) x for x
  proof -
    consider x < length vs | x = length vs | x > length vs
      by linarith
    then show ?thesis
      using n f-def by (cases) (auto simp add: prepend-associative)
  qed
  ultimately show ?thesis by blast
qed
```

If Case 2 holds for a $j \geq 2$ with $\varphi_j = \psi_j$, that is, if $\psi_j \in V_1$, then there is a function in V, namely ψ_j, on which S fails. Therefore S does not learn V.

```
lemma case-two-not-learn-V:
  assumes case-two j and j ≥ 2 and φ j = ψ j
  shows ¬ learn-total φ V-constotal s
proof -
  define z where z = longest-prefix j
  then have z > 0
    using longest-prefix-gr-0[OF assms(1)] by simp
```

```
define vs where vs = prefix (ψ j) (z − 1)
then have vs ! 0 = j
  using psi-at-0 ‹z > 0› by simp
define a where a = tl vs
then have vs: vs = j # a
  using vs-def ‹vs ! 0 = j›
  by (metis length-Suc-conv length-prefix list.sel(3) nth-Cons-0)
obtain k where k: k ≥ 2 and phi-k: φ k = j # a @ [k] ⊙ 0^∞
  using goedel-after-prefixes[of 2 j # a] by auto
have phi-j: φ j = j # a ⊙ ↑^∞
proof (rule prepend-eqI)
  show ⋀x. x < length (j # a) ⟹ φ j x ↓= (j # a) ! x
    using assms(1,3) vs vs-def ‹0 < z›
      length-prefix[of ψ j z − 1]
      prefix-nth[of - - ψ j]
      psi-at-0[of j]
      case-two-psi-longest-prefix[OF - z-def]
      longest-prefix[OF - z-def]
    by (metis One-nat-def Suc-pred option.collapse)
  show ⋀x. φ j (length (j # a) + x) ↑
    using assms(3) vs-def
    by (simp add: vs assms(1) case-two-psi-longest-prefix z-def)
qed
moreover have φ k ∈ V-constotal-2
proof (intro V-constotal-2I[of - j a k])
  show φ k = j # a @ [k] ⊙ 0^∞
    using phi-k .
  show 2 ≤ j
    using ‹2 ≤ j› .
  show 2 ≤ k
    using ‹2 ≤ k› .
  show ∀ i<length a. a ! i ≤ 1
  proof (rule allI, rule impI)
    fix i assume i: i < length a
    then have Suc i < z
      using z-def vs-def length-prefix ‹0 < z› vs
      by (metis One-nat-def Suc-mono Suc-pred length-Cons)
    have a ! i = vs ! (Suc i)
      using vs by simp
    also have ... = the (ψ j (Suc i))
      using vs-def vs i length-Cons length-prefix prefix-nth
      by (metis Suc-mono)
    finally show a ! i ≤ 1
      using case-two-psi-longest-prefix ‹Suc i < z› z-def
      by (metis assms(1) less-or-eq-imp-le not-le-imp-less not-one-less-zero
        option.sel zero-less-Suc)
  qed
qed (auto simp add: phi-j)
then have φ k ∈ V-constotal
  using V-constotal-def by auto
moreover have ¬ learn-total φ {φ k} s
proof −
  have φ k ∈ ℛ
    by (simp add: phi-k almost0-in-R1)
  moreover have ⋀x. x < longest-prefix j ⟹ φ k x = ψ j x
    using phi-k vs-def z-def length-prefix phi-j prepend-associative prepend-at-less
```

by (*metis One-nat-def Suc-pred* ‹*0* < *z*› ‹*vs* = *j* # *a*› *append-Cons assms*(*3*))
ultimately show *?thesis*
using *case-two-not-learn*[*OF assms*(*1*)] **by** *simp*
qed
ultimately show ¬ *learn-total* φ *V-constotal s*
using *learn-total-closed-subseteq* **by** *auto*
qed

The strategy S does not learn V in either case.

lemma *not-learn-total-V*: ¬ *learn-total* φ *V-constotal s*
proof −
obtain *j* **where** $j \geq 2$ φ $j = \psi$ j
using *kleene-fixed-point psi-in-P2* **by** *auto*
then show *?thesis*
using *case-one-not-learn-V learn-total-def case-two-not-learn-V*
by (*cases case-two j*) *auto*
qed

end

lemma *V-not-in-TOTAL*: *V-constotal* ∉ *TOTAL*
proof (*rule ccontr*)
assume ¬ *V-constotal* ∉ *TOTAL*
then have *V-constotal* ∈ *TOTAL* **by** *simp*
then have *V-constotal* ∈ *TOTAL-wrt* φ
by (*simp add*: *TOTAL-wrt-phi-eq-TOTAL*)
then obtain *s* **where** *learn-total* φ *V-constotal s*
using *TOTAL-wrt-def* **by** *auto*
then obtain *s'* **where** *s'*: *s'* ∈ $\mathcal{R}$ *learn-total* φ *V-constotal s'*
using *lemma-R-for-TOTAL-simple* **by** *blast*
then interpret *total-cons s'*
by (*simp add*: *total-cons-def*)
have ¬ *learn-total* φ *V-constotal s'*
by (*simp add*: *not-learn-total-V*)
with *s'*(*2*) **show** *False* **by** *simp*
qed

lemma *TOTAL-neq-CONS*: *TOTAL* ≠ *CONS*
using *V-not-in-TOTAL V-in-CONS CONS-def* **by** *auto*

The main result of this section:

theorem *TOTAL-subset-CONS*: *TOTAL* ⊂ *CONS*
using *TOTAL-subseteq-CONS TOTAL-neq-CONS* **by** *simp*

end

2.11 $\mathcal{R}$ is not in BC

theory *R1-BC*
imports *Lemma-R*
CP-FIN-NUM
begin

We show that $U_0 \cup V_0$ is not in BC, which implies $\mathcal{R} \notin BC$.

The proof is by contradiction. Assume there is a strategy S learning $U_0 \cup V_0$ behaviorally correct in the limit with respect to our standard Gödel numbering φ. Thanks to Lemma R for BC we can assume S to be total. Then we construct a function in $U_0 \cup V_0$ for which S fails.

As usual, there is a computable process building prefixes of functions ψ_j. For every j it starts with the singleton prefix $b = [j]$ and computes the next prefix from a given prefix b as follows:

1. Simulate $\varphi_{S(b0^k)}(|b| + k)$ for increasing k for an increasing number of steps.
2. Once a k with $\varphi_{S(b0^k)}(|b| + k) = 0$ is found, extend the prefix by $0^k 1$.

There is always such a k because by assumption S learns $b0^\infty \in U_0$ and thus outputs a hypothesis for $b0^\infty$ on almost all of its prefixes. Therefore for almost all prefixes of the form $b0^k$, we have $\varphi_{S(b0^k)} = b0^\infty$ and hence $\varphi_{S(b0^k)}(|b| + k) = 0$. But Step 2 constructs ψ_j such that $\psi_j(|b| + k) = 1$. Therefore S does not hypothesize ψ_j on the prefix $b0^k$ of ψ_j. And since the process runs forever, S outputs infinitely many incorrect hypotheses for ψ_j and thus does not learn ψ_j.

Applying Kleene's fixed-point theorem to $\psi \in \mathcal{R}^2$ yields a j with $\varphi_j = \psi_j$ and thus $\psi_j \in V_0$. But S does not learn any ψ_j, contradicting our assumption.

The result $\mathcal{R} \notin BC$ can be obtained more directly by running the process with the empty prefix, thereby constructing only one function instead of a numbering. This function is in $\mathcal{R}$, and S fails to learn it by the same reasoning as above. The stronger statement about $U_0 \cup V_0$ will be exploited in Section 2.12.

In the following locale the assumption that S learns U_0 suffices for analyzing the process. However, in order to arrive at the desired contradiction this assumption is too weak because the functions built by the process are not in U_0.

locale *r1-bc* =
  fixes *s* :: *partial1*
  assumes *s-in-R1*: $s \in \mathcal{R}$ **and** *s-learn-U0*: *learn-bc* φ U_0 *s*
begin

lemma *s-learn-prenum*: $\bigwedge b$. *learn-bc* φ {*prenum b*} *s*
  using *s-learn-U0 U0-altdef learn-bc-closed-subseteq* **by** *blast*

A *recf* for the strategy:

definition *r-s* :: *recf* **where**
  r-s ≡ *SOME rs. recfn 1 rs* ∧ *total rs* ∧ *s* = (λ*x. eval rs* [*x*])

lemma *r-s-recfn* [*simp*]: *recfn 1 r-s*
  and *r-s-total*: $\bigwedge x$. *eval r-s* [*x*] ↓
  and *eval-r-s*: $\bigwedge x$. *s x* = *eval r-s* [*x*]
  using *r-s-def R1-SOME*[*OF s-in-R1*, *of r-s*] **by** *simp-all*

We begin with the function that finds the k from Step 1 of the construction of ψ.

definition *r-find-k* ≡
  let k = *Cn 2 r-pdec1* [*Id 2 0*];
    r = *Cn 2 r-result1*
      [*Cn 2 r-pdec2* [*Id 2 0*],
      Cn 2 r-s [*Cn 2 r-append-zeros* [*Id 2 1*, *k*]],
      Cn 2 r-add [*Cn 2 r-length* [*Id 2 1*], *k*]]

```
  in Cn 1 r-pdec1 [Mn 1 (Cn 2 r-eq [r, r-constn 1 1])]

lemma r-find-k-recfn [simp]: recfn 1 r-find-k
  unfolding r-find-k-def by (simp add: Let-def)
```

There is always a suitable k, since the strategy learns $b0^\infty$ for all b.

```
lemma learn-bc-prenum-eventually-zero:
  ∃ k. φ (the (s (e-append-zeros b k))) (e-length b + k) ↓= 0
proof −
  let ?f = prenum b
  have ∃ n≥e-length b. φ (the (s (?f ▷ n))) = ?f
    using learn-bcE s-learn-prenum by (meson le-cases singletonI)
  then obtain n where n: n ≥ e-length b φ (the (s (?f ▷ n))) = ?f
    by auto
  define k where k = Suc n − e-length b
  let ?e = e-append-zeros b k
  have len: e-length ?e = Suc n
    using k-def n e-append-zeros-length by simp
  have ?f ▷ n = ?e
  proof −
    have e-length ?e > 0
      using len n(1) by simp
    moreover have ?f x ↓= e-nth ?e x for x
    proof (cases x < e-length b)
      case True
      then show ?thesis using e-nth-append-zeros by simp
    next
      case False
      then have ?f x ↓= 0 by simp
      moreover from False have e-nth ?e x = 0
        using e-nth-append-zeros-big by simp
      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis using initI[of ?e] len by simp
  qed
  with n(2) have φ (the (s ?e)) = ?f by simp
  then have φ (the (s ?e)) (e-length ?e) ↓= 0
    using len n(1) by auto
  then show ?thesis using e-append-zeros-length by auto
qed

lemma if-eq-eq: (if v = 1 then (0 :: nat) else 1) = 0 ⟹ v = 1
  by presburger

lemma r-find-k:
  shows eval r-find-k [b] ↓
    and let k = the (eval r-find-k [b])
          in φ (the (s (e-append-zeros b k))) (e-length b + k) ↓= 0
proof −
  let ?k = Cn 2 r-pdec1 [Id 2 0]
  let ?argt = Cn 2 r-pdec2 [Id 2 0]
  let ?argi = Cn 2 r-s [Cn 2 r-append-zeros [Id 2 1, ?k]]
  let ?argx = Cn 2 r-add [Cn 2 r-length [Id 2 1], ?k]
  let ?r = Cn 2 r-result1 [?argt, ?argi, ?argx]
  define f where f ≡
    let k = Cn 2 r-pdec1 [Id 2 0];
```

```
    r = Cn 2 r-result1
          [Cn 2 r-pdec2 [Id 2 0],
           Cn 2 r-s [Cn 2 r-append-zeros [Id 2 1, k]],
           Cn 2 r-add [Cn 2 r-length [Id 2 1], k]]
  in Cn 2 r-eq [r, r-constn 1 1]
then have recfn 2 f by (simp add: Let-def)
have total r-s
  by (simp add: r-s-total totalI1)
then have total f
  unfolding f-def using Cn-total Mn-free-imp-total by (simp add: Let-def)

have eval ?argi [z, b] = s (e-append-zeros b (pdec1 z)) for z
  using r-append-zeros ‹recfn 2 f› eval-r-s by auto
then have eval ?argi [z, b] ↓= the (s (e-append-zeros b (pdec1 z))) for z
  using eval-r-s r-s-total by simp
moreover have recfn 2 ?r using ‹recfn 2 f› by auto
ultimately have r: eval ?r [z, b] =
    eval r-result1 [pdec2 z, the (s (e-append-zeros b (pdec1 z))), e-length b + pdec1 z]
    for z
  by simp
then have f: eval f [z, b] ↓= (if the (eval ?r [z, b]) = 1 then 0 else 1) for z
  using f-def ‹recfn 2 f› prim-recfn-total by (auto simp add: Let-def)

have ∃ k. φ (the (s (e-append-zeros b k))) (e-length b + k) ↓= 0
  using s-learn-prenum learn-bc-prenum-eventually-zero by auto
then obtain k where φ (the (s (e-append-zeros b k))) (e-length b + k) ↓= 0
  by auto
then obtain t where eval r-result1 [t, the (s (e-append-zeros b k)), e-length b + k] ↓= Suc 0
  using r-result1-converg-phi(1) by blast
then have t: eval r-result1 [t, the (s (e-append-zeros b k)), e-length b + k] ↓= Suc 0
  by simp

let ?z = prod-encode (k, t)
have eval ?r [?z, b] ↓= Suc 0
  using t r by (metis fst-conv prod-encode-inverse snd-conv)
with f have fzb: eval f [?z, b] ↓= 0 by simp
moreover have eval (Mn 1 f) [b] =
  (if (∃ z. eval f ([z, b]) ↓= 0)
   then Some (LEAST z. eval f [z, b] ↓= 0)
   else None)
  using eval-Mn-total[of 1 f [b]] ‹total f› ‹recfn 2 f› by simp
ultimately have mn1f: eval (Mn 1 f) [b] ↓= (LEAST z. eval f [z, b] ↓= 0)
  by auto
with fzb have eval f [the (eval (Mn 1 f) [b]), b] ↓= 0 (is eval f [?zz, b] ↓= 0)
  using ‹total f› ‹recfn 2 f› LeastI-ex[of %z. eval f [z, b] ↓= 0] by auto
moreover have eval f [?zz, b] ↓= (if the (eval ?r [?zz, b]) = 1 then 0 else 1)
  using f by simp
ultimately have (if the (eval ?r [?zz, b]) = 1 then (0 :: nat) else 1) = 0 by auto
then have the (eval ?r [?zz, b]) = 1
  using if-eq-eq[of the (eval ?r [?zz, b])] by simp
then have
   eval r-result1
     [pdec2 ?zz, the (s (e-append-zeros b (pdec1 ?zz))), e-length b + pdec1 ?zz] ↓=
   1
  using r r-result1-total r-result1-prim totalE
  by (metis length-Cons list.size(3) numeral-3-eq-3 option.collapse)
```

then have ∗: *φ* (*the* (*s* (*e-append-zeros b* (*pdec1 ?zz*)))) (*e-length b* + *pdec1 ?zz*) ↓= *0*
  by (*simp add*: *r-result1-some-phi*)

define *Mn1f* **where** *Mn1f* = *Mn 1 f*
then have *eval Mn1f* [*b*] ↓= *?zz*
  using *mn1f* **by** *auto*
moreover have *recfn 1* (*Cn 1 r-pdec1* [*Mn1f*])
  using ‹*recfn 2 f*› *Mn1f-def* **by** *simp*
ultimately have *eval* (*Cn 1 r-pdec1* [*Mn1f*]) [*b*] = *eval r-pdec1* [*the* (*eval* (*Mn1f*) [*b*])]
  by *auto*
then have *eval* (*Cn 1 r-pdec1* [*Mn1f*]) [*b*] = *eval r-pdec1* [*?zz*]
  using *Mn1f-def* **by** *blast*
then have *1*: *eval* (*Cn 1 r-pdec1* [*Mn1f*]) [*b*] ↓= *pdec1 ?zz*
  by *simp*
moreover have *recfn 1* (*Cn 1 S* [*Cn 1 r-pdec1* [*Mn1f*]])
  using ‹*recfn 2 f*› *Mn1f-def* **by** *simp*
ultimately have *eval* (*Cn 1 S* [*Cn 1 r-pdec1* [*Mn1f*]]) [*b*] =
    eval S [*the* (*eval* (*Cn 1 r-pdec1* [*Mn1f*]) [*b*])]
  by *simp*
then have *eval* (*Cn 1 S* [*Cn 1 r-pdec1* [*Mn1f*]]) [*b*] = *eval S* [*pdec1 ?zz*]
  using *1* **by** *simp*
then have *eval* (*Cn 1 S* [*Cn 1 r-pdec1* [*Mn1f*]]) [*b*] ↓= *Suc* (*pdec1 ?zz*)
  by *simp*
moreover have *eval r-find-k* [*b*] = *eval* (*Cn 1 r-pdec1* [*Mn1f*]) [*b*]
  unfolding *r-find-k-def Mn1f-def f-def* **by** *metis*
ultimately have *r-find-ksb*: *eval r-find-k* [*b*] ↓= *pdec1 ?zz*
  using *1* **by** *simp*
then show *eval r-find-k* [*b*] ↓ **by** *simp-all*

from *r-find-ksb* **have** *the* (*eval r-find-k* [*b*]) = *pdec1 ?zz*
  by *simp*
moreover have *φ* (*the* (*s* (*e-append-zeros b* (*pdec1 ?zz*)))) (*e-length b* + *pdec1 ?zz*) ↓= *0*
  using ∗ **by** *simp*
ultimately show *let k* = *the* (*eval r-find-k* [*b*])
    in φ (*the* (*s* (*e-append-zeros b k*))) (*e-length b* + *k*) ↓= *0*
  by *simp*
qed

lemma *r-find-k-total*: *total r-find-k*
  by (*simp add*: *s-learn-prenum r-find-k*(*1*) *totalI1*)

The following function represents one iteration of the process.

abbreviation *r-next* ≡
  Cn 3 r-snoc [*Cn 3 r-append-zeros* [*Id 3 1*, *Cn 3 r-find-k* [*Id 3 1*]], *r-constn 2 1*]

Using *r-next* we define the function *r-prefixes* that computes the prefix after every iteration of the process.

definition *r-prefixes* :: *recf* **where**
  r-prefixes ≡ *Pr 1 r-singleton-encode r-next*

lemma *r-prefixes-recfn*: *recfn 2 r-prefixes*
  unfolding *r-prefixes-def* **by** *simp*

lemma *r-prefixes-total*: *total r-prefixes*
proof −

have *recfn 3 r-next* **by** *simp*
then have *total r-next*
using ‹*recfn 3 r-next*› *r-find-k-total Cn-total Mn-free-imp-total* **by** *auto*
then show *?thesis*
by (*simp add*: *Mn-free-imp-total Pr-total r-prefixes-def*)
qed

lemma *r-prefixes-0*: *eval r-prefixes* [*0*, *j*] ↓= *list-encode* [*j*]
unfolding *r-prefixes-def* **by** *simp*

lemma *r-prefixes-Suc*:
eval r-prefixes [*Suc n*, *j*] ↓=
(*let b* = *the* (*eval r-prefixes* [*n*, *j*])
in e-snoc (*e-append-zeros b* (*the* (*eval r-find-k* [*b*]))) *1*)
proof −
have *recfn 3 r-next* **by** *simp*
then have *total r-next*
using ‹*recfn 3 r-next*› *r-find-k-total Cn-total Mn-free-imp-total* **by** *auto*
have *eval-next*: *eval r-next* [*t*, *v*, *j*] ↓=
e-snoc (*e-append-zeros v* (*the* (*eval r-find-k* [*v*]))) *1*
for *t v j*
using *r-find-k-total* ‹*recfn 3 r-next*› *r-append-zeros* **by** *simp*
then have *eval r-prefixes* [*Suc n*, *j*] = *eval r-next* [*n*, *the* (*eval r-prefixes* [*n*, *j*]), *j*]
using *r-prefixes-total* **by** (*simp add*: *r-prefixes-def*)
then show *eval r-prefixes* [*Suc n*, *j*] ↓=
(*let b* = *the* (*eval r-prefixes* [*n*, *j*])
in e-snoc (*e-append-zeros b* (*the* (*eval r-find-k* [*b*]))) *1*)
using *eval-next* **by** *metis*
qed

Since *r-prefixes* is total, we can get away with introducing a total function.

definition *prefixes* :: *nat* ⇒ *nat* ⇒ *nat* **where**
prefixes j t ≡ *the* (*eval r-prefixes* [*t*, *j*])

lemma *prefixes-Suc*:
prefixes j (*Suc t*) =
e-snoc (*e-append-zeros* (*prefixes j t*) (*the* (*eval r-find-k* [*prefixes j t*]))) *1*
unfolding *prefixes-def* **using** *r-prefixes-Suc* **by** (*simp-all add*: *Let-def*)

lemma *prefixes-Suc-length*:
e-length (*prefixes j* (*Suc t*)) =
Suc (*e-length* (*prefixes j t*) + *the* (*eval r-find-k* [*prefixes j t*]))
using *e-append-zeros-length prefixes-Suc* **by** *simp*

lemma *prefixes-length-mono*: *e-length* (*prefixes j t*) < *e-length* (*prefixes j* (*Suc t*))
using *prefixes-Suc-length* **by** *simp*

lemma *prefixes-length-mono'*: *e-length* (*prefixes j t*) ≤ *e-length* (*prefixes j* (*t* + *d*))
proof (*induction d*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc d*)
then show *?case* **using** *prefixes-length-mono le-less-trans* **by** *fastforce*
qed

lemma *prefixes-length-lower-bound*: *e-length* (*prefixes j t*) ≥ *Suc t*
proof (*induction t*)
  case *0*
  then show *?case* **by** (*simp add*: *prefixes-def r-prefixes-0*)
next
  case (*Suc t*)
  moreover have *Suc* (*e-length* (*prefixes j t*)) ≤ *e-length* (*prefixes j* (*Suc t*))
    using *prefixes-length-mono* **by** (*simp add*: *Suc-leI*)
  ultimately show *?case* **by** *simp*
qed

lemma *prefixes-Suc-nth*:
  assumes *x* < *e-length* (*prefixes j t*)
  shows *e-nth* (*prefixes j t*) *x* = *e-nth* (*prefixes j* (*Suc t*)) *x*
proof −
  define *k* **where** *k* = *the* (*eval r-find-k* [*prefixes j t*])
  let *?u* = *e-append-zeros* (*prefixes j t*) *k*
  have *prefixes j* (*Suc t*) =
      e-snoc (*e-append-zeros* (*prefixes j t*) (*the* (*eval r-find-k* [*prefixes j t*]))) *1*
    using *prefixes-Suc* **by** *simp*
  with *k-def* **have** *prefixes j* (*Suc t*) = *e-snoc ?u 1*
    by *simp*
  then have *e-nth* (*prefixes j* (*Suc t*)) *x* = *e-nth* (*e-snoc ?u 1*) *x*
    by *simp*
  moreover have *x* < *e-length ?u*
    using *assms e-append-zeros-length* **by** *auto*
  ultimately have *e-nth* (*prefixes j* (*Suc t*)) *x* = *e-nth ?u x*
    using *e-nth-snoc-small* **by** *simp*
  moreover have *e-nth ?u x* = *e-nth* (*prefixes j t*) *x*
    using *assms e-nth-append-zeros* **by** *simp*
  ultimately show *e-nth* (*prefixes j t*) *x* = *e-nth* (*prefixes j* (*Suc t*)) *x*
    by *simp*
qed

lemma *prefixes-Suc-last*: *e-nth* (*prefixes j* (*Suc t*)) (*e-length* (*prefixes j* (*Suc t*)) − *1*) = *1*
  using *prefixes-Suc* **by** *simp*

lemma *prefixes-le-nth*:
  assumes *x* < *e-length* (*prefixes j t*)
  shows *e-nth* (*prefixes j t*) *x* = *e-nth* (*prefixes j* (*t* + *d*)) *x*
proof (*induction d*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc d*)
  have *x* < *e-length* (*prefixes j* (*t* + *d*))
    using *s-learn-prenum assms prefixes-length-mono′*
    by (*simp add*: *less-eq-Suc-le order-trans-rules*(*23*))
  then have *e-nth* (*prefixes j* (*t* + *d*)) *x* = *e-nth* (*prefixes j* (*t* + *Suc d*)) *x*
    using *prefixes-Suc-nth* **by** *simp*
  with *Suc* **show** *?case* **by** *simp*
qed

The numbering ψ is defined via *prefixes*.

definition *psi* :: *partial2* (‹ψ›) **where**
  ψ *j x* ≡ *Some* (*e-nth* (*prefixes j* (*Suc x*)) *x*)

lemma *psi-in-R2*: $\psi \in \mathcal{R}^2$
proof
  define *r* **where** $r \equiv$ *Cn 2 r-nth* [*Cn 2 r-prefixes* [*Cn 2 S* [*Id 2 1*], *Id 2 0*], *Id 2 1*]
  then have *recfn 2 r*
    using *r-prefixes-recfn* **by** *simp*
  then have *eval r* [*j*, *x*] ↓= *e-nth* (*prefixes j* (*Suc x*)) *x* **for** *j x*
    unfolding *r-def prefixes-def* **using** *r-prefixes-total r-prefixes-recfn e-nth* **by** *simp*
  then have *eval r* [*j*, *x*] = *ψ j x* **for** *j x*
    unfolding *psi-def* **by** *simp*
  then show $\psi \in \mathcal{P}^2$
    using ‹*recfn 2 r*› **by** *auto*
  show *total2 ψ*
    unfolding *psi-def* **by** *auto*
qed

lemma *psi-eq-nth-prefixes*:
  assumes *x* < *e-length* (*prefixes j t*)
  shows *ψ j x* ↓= *e-nth* (*prefixes j t*) *x*
proof (*cases Suc x* < *t*)
  case *True*
  have *x* ≤ *e-length* (*prefixes j x*)
    using *prefixes-length-lower-bound* **by** (*simp add*: *Suc-leD*)
  also have ... < *e-length* (*prefixes j* (*Suc x*))
    using *prefixes-length-mono s-learn-prenum* **by** *simp*
  finally have *x* < *e-length* (*prefixes j* (*Suc x*)) **.**
  with *True* **have** *e-nth* (*prefixes j* (*Suc x*)) *x* = *e-nth* (*prefixes j t*) *x*
    using *prefixes-le-nth*[*of x j Suc x t − Suc x*] **by** *simp*
  then show *?thesis* **using** *psi-def* **by** *simp*
next
  case *False*
  then have *e-nth* (*prefixes j* (*Suc x*)) *x* = *e-nth* (*prefixes j t*) *x*
    using *prefixes-le-nth*[*of x j t Suc x − t*] *assms* **by** *simp*
  then show *?thesis* **using** *psi-def* **by** *simp*
qed

lemma *psi-at-0*: *ψ j 0* ↓= *j*
  using *psi-eq-nth-prefixes*[*of 0 j 0*] *prefixes-length-lower-bound*[*of 0 j*]
  by (*simp add*: *prefixes-def r-prefixes-0*)

The prefixes output by the process *prefixes j* are indeed prefixes of ψ_j.

lemma *prefixes-init-psi*: *ψ j* ▷ (*e-length* (*prefixes j* (*Suc t*)) − *1*) = *prefixes j* (*Suc t*)
proof (*rule initI*[*of prefixes j* (*Suc t*)])
  let *?e* = *prefixes j* (*Suc t*)
  show *e-length ?e* > *0*
    using *prefixes-length-lower-bound*[*of Suc t j*] **by** *auto*
  show ⋀*x*. *x* < *e-length ?e* ⟹ *ψ j x* ↓= *e-nth ?e x*
    using *prefixes-Suc-nth psi-eq-nth-prefixes* **by** *simp*
qed

Every prefix of ψ_j generated by the process *prefixes j* (except for the initial one) is of the form $b0^k1$. But k is chosen such that $\varphi_{S(b0^k)}(|b|+k) = 0 \neq 1 = b0^k1_{|b|+k}$. Therefore the hypothesis $S(b0^k)$ is incorrect for ψ_j.

lemma *hyp-wrong-at-last*:
  φ (*the* (*s* (*e-butlast* (*prefixes j* (*Suc t*))))) (*e-length* (*prefixes j* (*Suc t*)) − *1*) ≠

ψ *j* (*e-length* (*prefixes j* (*Suc t*)) − *1*)
(**is** *?lhs* ≠ *?rhs*)
proof −
let *?b* = *prefixes j t*
let *?k* = *the* (*eval r-find-k* [*?b*])
let *?x* = *e-length* (*prefixes j* (*Suc t*)) − *1*
have *e-butlast* (*prefixes j* (*Suc t*)) = *e-append-zeros ?b ?k*
using *s-learn-prenum prefixes-Suc* **by** *simp*
then have *?lhs* = φ (*the* (*s* (*e-append-zeros ?b ?k*))) *?x*
by *simp*
moreover have *?x* = *e-length ?b* + *?k*
using *prefixes-Suc-length* **by** *simp*
ultimately have *?lhs* = φ (*the* (*s* (*e-append-zeros ?b ?k*))) (*e-length ?b* + *?k*)
by *simp*
then have *?lhs* ↓= *0*
using *r-find-k*(*2*) *r-s-total s-learn-prenum* **by** *metis*
moreover have *?x* < *e-length* (*prefixes j* (*Suc t*))
using *prefixes-length-lower-bound le-less-trans linorder-not-le s-learn-prenum*
by *fastforce*
ultimately have *?rhs* ↓= *e-nth* (*prefixes j* (*Suc t*)) *?x*
using *psi-eq-nth-prefixes*[*of ?x j Suc t*] **by** *simp*
moreover have *e-nth* (*prefixes j* (*Suc t*)) *?x* = *1*
using *prefixes-Suc prefixes-Suc-last* **by** *simp*
ultimately have *?rhs* ↓= *1* **by** *simp*
with ‹*?lhs* ↓= *0*› **show** *?thesis* **by** *simp*
qed

corollary *hyp-wrong*: φ (*the* (*s* (*e-butlast* (*prefixes j* (*Suc t*))))) ≠ ψ *j*
using *hyp-wrong-at-last*[*of j t*] **by** *auto*

For all j, the strategy S outputs infinitely many wrong hypotheses for ψ_j

lemma *infinite-hyp-wrong*: ∃ *m*>*n*. φ (*the* (*s* (ψ *j* ▹ *m*))) ≠ ψ *j*
proof −
let *?b* = *prefixes j* (*Suc* (*Suc n*))
let *?bb* = *e-butlast ?b*
have *len-b*: *e-length ?b* > *Suc* (*Suc n*)
using *prefixes-length-lower-bound* **by** (*simp add*: *Suc-le-lessD*)
then have *len-bb*: *e-length ?bb* > *Suc n* **by** *simp*
define *m* **where** *m* = *e-length ?bb* − *1*
with *len-bb* **have** *m* > *n* **by** *simp*
have ψ *j* ▹ *m* = *?bb*
proof −
have ψ *j* ▹ (*e-length ?b* − *1*) = *?b*
using *prefixes-init-psi* **by** *simp*
then have ψ *j* ▹ (*e-length ?b* − *2*) = *?bb*
using *init-butlast-init psi-in-R2 R2-proj-R1 R1-imp-total1 len-bb length-init*
by (*metis Suc-1 diff-diff-left length-butlast length-greater-0-conv*
list.size(*3*) *list-decode-encode not-less0 plus-1-eq-Suc*)
then show *?thesis* **by** (*metis diff-Suc-1 length-init m-def*)
qed
moreover have φ (*the* (*s ?bb*)) ≠ ψ *j*
using *hyp-wrong* **by** *simp*
ultimately have φ (*the* (*s* (ψ *j* ▹ *m*))) ≠ ψ *j*
by *simp*
with ‹*m* > *n*› **show** *?thesis* **by** *auto*
qed

```
lemma U0-V0-not-learn-bc: ¬ learn-bc φ (U₀ ∪ V₀) s
proof −
  obtain j where j: φ j = ψ j
    using R2-imp-P2 kleene-fixed-point psi-in-R2 by blast
  moreover have ∃m>n. φ (the (s ((ψ j) ▷ m))) ≠ ψ j for n
    using infinite-hyp-wrong[of - j] by simp
  ultimately have ¬ learn-bc φ {ψ j} s
    using infinite-hyp-wrong-not-BC by simp
  moreover have ψ j ∈ V₀
  proof −
    have ψ j ∈ ℛ (is ?f ∈ ℛ)
      using psi-in-R2 by simp
    moreover have φ (the (?f 0)) = ?f
      using j psi-at-0[of j] by simp
    ultimately show ?thesis by (simp add: V0-def)
  qed
  ultimately show ¬ learn-bc φ (U₀ ∪ V₀) s
    using learn-bc-closed-subseteq by auto
qed

end

lemma U0-V0-not-in-BC: U₀ ∪ V₀ ∉ BC
proof
  assume in-BC: U₀ ∪ V₀ ∈ BC
  then have U₀ ∪ V₀ ∈ BC-wrt φ
    using BC-wrt-phi-eq-BC by simp
  then obtain s where learn-bc φ (U₀ ∪ V₀) s
    using BC-wrt-def by auto
  then obtain s' where s': s' ∈ ℛ learn-bc φ (U₀ ∪ V₀) s'
    using lemma-R-for-BC-simple by blast
  then have learn-U0: learn-bc φ U₀ s'
    using learn-bc-closed-subseteq[of φ U₀ ∪ V₀ s'] by simp
  then interpret r1-bc s'
    by (simp add: r1-bc-def s'(1))
  have ¬ learn-bc φ (U₀ ∪ V₀) s'
    using learn-bc-closed-subseteq U0-V0-not-learn-bc by simp
  with s'(2) show False by simp
qed

theorem R1-not-in-BC: ℛ ∉ BC
proof −
  have U₀ ∪ V₀ ⊆ ℛ
    using V0-def U0-in-NUM by auto
  then show ?thesis
    using U0-V0-not-in-BC BC-closed-subseteq by auto
qed

end
```

2.12 The union of classes

```
theory Union
  imports R1-BC TOTAL-CONS
```

begin

None of the inference types introduced in this chapter are closed under union of classes. For all inference types except FIN this follows from *U0-V0-not-in-BC*.

```
lemma not-closed-under-union:
  ∀ℐ∈{CP, TOTAL, CONS, LIM, BC}. U₀ ∈ ℐ ∧ V₀ ∈ ℐ ∧ U₀ ∪ V₀ ∉ ℐ
  using U0-in-CP U0-in-NUM V0-in-FIN
    FIN-subseteq-CP
    NUM-subseteq-TOTAL
    CP-subseteq-TOTAL
    TOTAL-subseteq-CONS
    CONS-subseteq-Lim
    Lim-subseteq-BC
    U0-V0-not-in-BC
  by blast
```

In order to show the analogous result for FIN consider the classes $\{0^\infty\}$ and $\{0^n 1 0^\infty \mid n \in \mathbb{N}\}$. The former can be learned finitely by a strategy that hypothesizes 0^∞ for every input. The latter can be learned finitely by a strategy that waits for the 1 and hypothesizes the only function in the class with a 1 at that position. However, the union of both classes is not in FIN. This is because any FIN strategy has to hypothesize 0^∞ on some prefix of the form 0^n. But the strategy then fails for the function $0^n 1 0^\infty$.

```
lemma singleton-in-FIN: f ∈ ℛ ⟹ {f} ∈ FIN
proof -
  assume f ∈ ℛ
  then obtain i where i: φ i = f
    using phi-universal by blast
  define s :: partial1 where s = (λ-. Some (Suc i))
  then have s ∈ ℛ
    using const-in-Prim1[of Suc i] by simp
  have learn-fin φ {f} s
  proof (intro learn-finI)
    show environment φ {f} s
      using ‹s ∈ ℛ› ‹f ∈ ℛ› by (simp add: phi-in-P2)
    show ∃ i n₀. φ i = g ∧ (∀ n<n₀. s (g ▷ n) ↓= 0) ∧ (∀ n≥n₀. s (g ▷ n) ↓= Suc i)
      if g ∈ {f} for g
    proof -
      from that have g = f by simp
      then have φ i = g
        using i by simp
      moreover have ∀ n<0. s (g ▷ n) ↓= 0 by simp
      moreover have ∀ n≥0. s (g ▷ n) ↓= Suc i
        using s-def by simp
      ultimately show ?thesis by auto
    qed
  qed
  then show {f} ∈ FIN using FIN-def by auto
qed

definition U-single :: partial1 set where
  U-single ≡ {(λx. if x = n then Some 1 else Some 0)| n. n ∈ UNIV}

lemma U-single-in-FIN: U-single ∈ FIN
proof -
  define psi :: partial2 where psi ≡ λn x. if x = n then Some 1 else Some 0
```

have $psi \in \mathcal{R}^2$
  using *psi-def* **by** (*intro R2I*[*of Cn 2 r-not* [*r-eq*]]) *auto*
define *s* :: *partial1* **where**
  $s \equiv \lambda b.$ *if findr b* $\downarrow=$ *e-length b then Some 0 else Some* (*Suc* (*the* (*findr b*)))
have $s \in \mathcal{R}$
proof (*rule R1I*)
  let *?r* = *Cn 1 r-ifeq* [*r-findr*, *r-length*, *Z*, *Cn 1 S* [*r-findr*]]
  show *recfn 1 ?r* **by** *simp*
  show *total ?r* **by** *auto*
  show *eval ?r* [*b*] = *s b* **for** *b*
  proof −
    let *?b* = *the* (*findr b*)
    have *eval ?r* [*b*] = (*if ?b* = *e-length b then Some 0 else Some* (*Suc* (*?b*)))
      using *findr-total* **by** *simp*
    then show *eval ?r* [*b*] = *s b*
      by (*metis findr-total option.collapse option.inject s-def*)
  qed
qed
have $U\text{-}single \subseteq \mathcal{R}$
proof
  fix *f*
  assume $f \in U\text{-}single$
  then obtain *n* **where** $f = (\lambda x.$ *if x = n then Some 1 else Some 0*)
    using *U-single-def* **by** *auto*
  then have *f* = *psi n*
    using *psi-def* **by** *simp*
  then show $f \in \mathcal{R}$
    using ‹$psi \in \mathcal{R}^2$› **by** *simp*
qed
have *learn-fin psi U-single s*
proof (*rule learn-finI*)
  show *environment psi U-single s*
    using ‹$psi \in \mathcal{R}^2$› ‹$s \in \mathcal{R}$› ‹$U\text{-}single \subseteq \mathcal{R}$› **by** *simp*
  show $\exists i\ n_0.\ psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \triangleright n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \triangleright n) \downarrow= Suc\ i)$
    if $f \in U\text{-}single$ **for** *f*
  proof −
    from *that* **obtain** *i* **where** *i*: $f = (\lambda x.$ *if x = i then Some 1 else Some 0*)
      using *U-single-def* **by** *auto*
    then have *psi i* = *f*
      using *psi-def* **by** *simp*
    moreover have $\forall n{<}i.\ s\ (f \triangleright n) \downarrow= 0$
      using *i s-def findr-def* **by** *simp*
    moreover have $\forall n{\geq}i.\ s\ (f \triangleright n) \downarrow= Suc\ i$
    proof (*rule allI*, *rule impI*)
      fix *n*
      assume $n \geq i$
      let *?e* = *init f n*
      have $\exists i{<}e\text{-}length\ ?e.\ e\text{-}nth\ ?e\ i \neq 0$
        using ‹$n \geq i$› *i* **by** *simp*
      then have *less*: *the* (*findr ?e*) < *e-length ?e*
        and *nth-e*: *e-nth ?e* (*the* (*findr ?e*)) $\neq$ *0*
        using *findr-ex* **by** *blast+*
      then have *s ?e* $\downarrow=$ *Suc* (*the* (*findr ?e*))
        using *s-def* **by** *auto*
      moreover have *the* (*findr ?e*) = *i*
        using *nth-e less i* **by** (*metis length-init nth-init option.sel*)

ultimately show $s\ ?e \downarrow= Suc\ i$ **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed
qed
then show $U\text{-}single \in FIN$ **using** *FIN-def* **by** *blast*
qed

lemma *zero-U-single-not-in-FIN*: $\{0^\infty\} \cup U\text{-}single \notin FIN$
proof
assume $\{0^\infty\} \cup U\text{-}single \in FIN$
then obtain *psi s* **where** *learn*: *learn-fin psi* $(\{0^\infty\} \cup U\text{-}single)$ *s*
using *FIN-def* **by** *blast*
then have *learn-fin psi* $\{0^\infty\}$ *s*
using *learn-fin-closed-subseteq* **by** *auto*
then obtain $i\ n_0$ **where** *i*:
$psi\ i = 0^\infty$
$\forall n<n_0.\ s\ (0^\infty \triangleright n) \downarrow= 0$
$\forall n\geq n_0.\ s\ (0^\infty \triangleright n) \downarrow= Suc\ i$
using *learn-finE*(*2*) **by** *blast*
let $?f = \lambda x.\ if\ x = Suc\ n_0\ then\ Some\ 1\ else\ Some\ 0$
have $?f \neq 0^\infty$ **by** (*metis option.inject zero-neq-one*)
have $?f \in U\text{-}single$
using *U-single-def* **by** *auto*
then have *learn-fin psi* $\{?f\}$ *s*
using *learn learn-fin-closed-subseteq* **by** *simp*
then obtain $j\ m_0$ **where** *j*:
$psi\ j = ?f$
$\forall n<m_0.\ s\ (?f \triangleright n) \downarrow= 0$
$\forall n\geq m_0.\ s\ (?f \triangleright n) \downarrow= Suc\ j$
using *learn-finE*(*2*) **by** *blast*
consider
(*less*) $m_0 < n_0$ | (*eq*) $m_0 = n_0$ | (*gr*) $m_0 > n_0$
by *linarith*
then show *False*
proof (*cases*)
case *less*
then have $s\ (0^\infty \triangleright m_0) \downarrow= 0$
using *i* **by** *simp*
moreover have $0^\infty \triangleright m_0 = ?f \triangleright m_0$
using *less init-eqI*[*of* m_0 $?f$ 0^∞] **by** *simp*
ultimately have $s\ (?f \triangleright m_0) \downarrow= 0$ **by** *simp*
then show *False* **using** *j* **by** *simp*
next
case *eq*
then have $0^\infty \triangleright m_0 = ?f \triangleright m_0$
using *init-eqI*[*of* m_0 $?f$ 0^∞] **by** *simp*
then have $s\ (0^\infty \triangleright m_0) = s\ (?f \triangleright m_0)$ **by** *simp*
then have $i = j$
using *i j eq* **by** *simp*
then have $psi\ i = psi\ j$ **by** *simp*
then show *False* **using** $\langle ?f \neq 0^\infty \rangle$ *i j* **by** *simp*
next
case *gr*
have $0^\infty \triangleright n_0 = ?f \triangleright n_0$
using *init-eqI*[*of* n_0 $?f$ 0^∞] **by** *simp*

moreover have *s* $(0^\infty \triangleright n_0) \downarrow= Suc\ i$
using *i* **by** *simp*
moreover have *s* $(?f \triangleright n_0) \downarrow= 0$
using *j gr* **by** *simp*
ultimately show *False* **by** *simp*
qed
qed

lemma *FIN-not-closed-under-union*: $\exists\ U\ V.\ U \in FIN \wedge V \in FIN \wedge U \cup V \notin FIN$
proof −
have $\{0^\infty\} \in FIN$
using *singleton-in-FIN const-in-Prim1* **by** *simp*
moreover have *U-single* $\in$ *FIN*
using *U-single-in-FIN* **by** *simp*
ultimately show *?thesis*
using *zero-U-single-not-in-FIN* **by** *blast*
qed

In contrast to the inference types, NUM is closed under the union of classes. The total numberings that exist for each NUM class can be interleaved to produce a total numbering encompassing the union of the classes. To define the interleaving, modulo and division by two will be helpful.

definition *r-div2* $\equiv$
r-shrink
(*Pr 1 Z*
(*Cn 3 r-ifle*
[*Cn 3 r-mul* [*r-constn 2 2*, *Cn 3 S* [*Id 3 0*]], *Id 3 2*, *Cn 3 S* [*Id 3 1*], *Id 3 1*]))

lemma *r-div2-prim* [*simp*]: *prim-recfn 1 r-div2*
unfolding *r-div2-def* **by** *simp*

lemma *r-div2* [*simp*]: *eval r-div2* [*n*] $\downarrow=$ *n div 2*
proof −
let *?p* = *Pr 1 Z*
(*Cn 3 r-ifle*
[*Cn 3 r-mul* [*r-constn 2 2*, *Cn 3 S* [*Id 3 0*]], *Id 3 2*, *Cn 3 S* [*Id 3 1*], *Id 3 1*])
have *eval ?p* [*i*, *n*] $\downarrow=$ *min* (*n div 2*) *i* **for** *i*
by (*induction i*) *auto*
then have *eval ?p* [*n*, *n*] $\downarrow=$ *n div 2* **by** *simp*
then show *?thesis* **unfolding** *r-div2-def* **by** *simp*
qed

definition *r-mod2* $\equiv$ *Cn 1 r-sub* [*Id 1 0*, *Cn 1 r-mul* [*r-const 2*, *r-div2*]]

lemma *r-mod2-prim* [*simp*]: *prim-recfn 1 r-mod2*
unfolding *r-mod2-def* **by** *simp*

lemma *r-mod2* [*simp*]: *eval r-mod2* [*n*] $\downarrow=$ *n mod 2*
unfolding *r-mod2-def* **using** *Rings.semiring-modulo-class.minus-mult-div-eq-mod*
by *auto*

lemma *NUM-closed-under-union*:
assumes *U* $\in$ *NUM* **and** *V* $\in$ *NUM*
shows *U* $\cup$ *V* $\in$ *NUM*
proof −

```
  from assms obtain psi-u psi-v where
    psi-u: psi-u ∈ R² ⋀f. f ∈ U ⟹ ∃ i. psi-u i = f and
    psi-v: psi-v ∈ R² ⋀f. f ∈ V ⟹ ∃ i. psi-v i = f
    by fastforce
  define psi where psi ≡ λi. if i mod 2 = 0 then psi-u (i div 2) else psi-v (i div 2)
  from psi-u(1) obtain u where u: recfn 2 u total u ⋀x y. eval u [x, y] = psi-u x y
    by auto
  from psi-v(1) obtain v where v: recfn 2 v total v ⋀x y. eval v [x, y] = psi-v x y
    by auto
  let ?r-psi = Cn 2 r-ifz
    [Cn 2 r-mod2 [Id 2 0],
     Cn 2 u [Cn 2 r-div2 [Id 2 0], Id 2 1],
     Cn 2 v [Cn 2 r-div2 [Id 2 0], Id 2 1]]
  show ?thesis
  proof (rule NUM-I[of psi])
    show psi ∈ R²
    proof (rule R2I)
      show recfn 2 ?r-psi
        using u(1) v(1) by simp
      show eval ?r-psi [x, y] = psi x y for x y
        using u v psi-def prim-recfn-total R2-imp-total2[OF psi-u(1)]
          R2-imp-total2[OF psi-v(1)]
        by simp
      moreover have psi x y ↓ for x y
        using psi-def psi-u(1) psi-v(1) by simp
      ultimately show total ?r-psi
        using ‹recfn 2 ?r-psi› totalI2 by simp
    qed
    show ∃ i. psi i = f if f ∈ U ∪ V for f
    proof (cases f ∈ U)
      case True
      then obtain j where psi-u j = f
        using psi-u(2) by auto
      then have psi (2 ∗ j) = f
        using psi-def by simp
      then show ?thesis by auto
    next
      case False
      then have f ∈ V
        using that by simp
      then obtain j where psi-v j = f
        using psi-v(2) by auto
      then have psi (Suc (2 ∗ j)) = f
        using psi-def by simp
      then show ?thesis by auto
    qed
  qed
qed

end
```

Bibliography

[1] D. Angluin and C. H. Smith. Inductive inference. In *Encyclopedia of Artificial Intelligence*, pages 409–418. J. Wiley and Sons, New York, 1987.

[2] J. M. Barzdin. Two theorems on the limiting synthesis of functions. In *Theory of Algorithms and Programs*, volume 1, pages 82–88. Latvian State University, Riga, 1974. In Russian.

[3] J. M. Barzdin. Inductive inference of automata, functions and programs. In *Amer. Math. Soc. Transl.*, pages 107–122, 1977.

[4] Y. M. Barzdin. Inductive inference of automata, functions and programs. In *Proceedings International Congress of Mathematics*, pages 455–460, 1974.

[5] L. Blum and M. Blum. Toward a mathematical theory of inductive inference. *Inform. Control*, 28(2):125–155, June 1975.

[6] J. Case and C. H. Smith. Comparison of identification criteria for machine inductive inference. *Theoret. Comput. Sci.*, 25:193–220, 1983.

[7] R. Freivalds, E. B. Kinber, and R. Wiehagen. How inductive inference strategies discover their errors. *Inform. Comput.*, 118(2):208–226, 1995.

[8] E. M. Gold. Limiting recursion. *J. Symbolic Logic*, 30:28–48, 1965.

[9] E. M. Gold. Language identification in the limit. *Inform. Control*, 10(5):447–474, 1967.

[10] K. P. Jantke and H.-R. Beick. Combining postulates of naturalness in inductive inference. *Elektronische Informationsverarbeitung und Kybernetik*, 17(8/9):465–484, 1981.

[11] S. C. Kleene. Recursive predicates and quantifiers. *Trans. Amer. Math. Soc.*, 53(1):41–73, 1943.

[12] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* The MIT Press, 2nd edition, 1987.

[13] R. J. Solomonoff. A formal theory of inductive inference: Part 1. *Inform. Control*, 7:1–22, 1964.

[14] R. J. Solomonoff. A formal theory of inductive inference: Part 2. *Inform. Control*, 7:224–254, 1964.

[15] R. Wiehagen. Limes-Erkennung rekursiver Funktionen durch spezielle Strategien. *Journal of Information Processing and Cybernetics (EIK)*, 12:93–99, 1976.

[16] R. Wiehagen and T. Zeugmann. Ignoring data may be the only way to learn efficiently. *J. of Experimental and Theoret. Artif. Intell.*, 6(1):131–144, 1994.

[17] Wikipedia contributors. Kleene's recursion theorem — Wikipedia, the free encyclopedia, 2020. [Online; accessed 28-March-2020].

[18] J. Xu, X. Zhang, C. Urban, and S. J. C. Joosten. Universal turing machine. *Archive of Formal Proofs*, Feb. 2019. http://isa-afp.org/entries/Universal_Turing_Machine.html, Formal proof development.