Some classical results in inductive inference of recursive functions

Frank J. Balbach

March 17, 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

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*
⟨*proof*⟩

lemma *wellf-Pr-arity-greater-1*: *wellf* (*Pr n f g*) ⟹ *arity* (*Pr n f g*) > *1*
⟨*proof*⟩

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* (λ*g*. *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*)) (λ*v*. *eval-wellf g* (*x* # *v* # *xs*))
| *eval-wellf* (*Mn n f*) *xs* =
(*let E* = λ*z*. *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*
⟨*proof*⟩

lemma *eval-Z′* [*simp*]: *length xs* = *1* ⟹ *eval Z xs* ↓= *0*
⟨*proof*⟩

lemma *eval-S* [*simp*]: *eval S* [*x*] ↓= *Suc x*
⟨*proof*⟩

lemma *eval-S′* [*simp*]: *length xs* = *1* ⟹ *eval S xs* ↓= *Suc* (*hd xs*)
⟨*proof*⟩

lemma *eval-Id* [*simp*]:
assumes *n* < *m* **and** *m* = *length xs*
shows *eval* (*Id m n*) *xs* ↓= *xs* ! *n*
⟨*proof*⟩

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* (λ*g*. *the* (*eval g xs*)) *gs*)
else None)
⟨*proof*⟩

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*
⟨*proof*⟩

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*) ↑
⟨*proof*⟩

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*)
⟨*proof*⟩

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*) ↑
⟨*proof*⟩

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*) ↓
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

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)
⟨*proof*⟩

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 $\bigwedge$*y*. *y* < *z* ⟹ *eval f* (*y* # *xs*) ↓≠ *0*
shows *eval* (*Mn n f*) *xs* ↓= *z*
⟨*proof*⟩

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 $\bigwedge$*y*. *y* < *z* ⟹ *eval f* (*y* # *xs*) ↓≠ *0*
⟨*proof*⟩

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* ↑
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *exteq-sym*: *f* ≃ *g* ⟹ *g* ≃ *f*
⟨*proof*⟩

lemma *exteq-trans*: *f* ≃ *g* ⟹ *g* ≃ *h* ⟹ *f* ≃ *h*
⟨*proof*⟩

lemma *exteqI*:
assumes *arity f* = *arity g* **and** $\bigwedge$*xs*. *length xs* = *arity f* ⟹ *eval f xs* = *eval g xs*
shows *f* ≃ *g*
⟨*proof*⟩

lemma *exteqI1*:
assumes *arity f* = *1* **and** *arity g* = *1* **and** $\bigwedge$*x*. *eval f* [*x*] = *eval g* [*x*]
shows *f* ≃ *g*
⟨*proof*⟩

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* $\Longrightarrow$ *recfn a* (*wrap-Id f n*)
⟨*proof*⟩

lemma *exteq-wrap-Id*: *recfn a f* $\Longrightarrow$ *f* $\simeq$ *wrap-Id f n*
⟨*proof*⟩

fun *depth* :: *recf* $\Rightarrow$ *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* $\Longrightarrow$ *depth* (*wrap-Id f n*) = *depth f* + *n*
⟨*proof*⟩

lemma *wrap-Id-injective*:
assumes *recfn a f* **and** *wrap-Id f* n_1 = *wrap-Id f* n_2
shows $n_1 = n_2$
⟨*proof*⟩

lemma *exteq-infinite*:
assumes *recfn a f*
shows *infinite* {*g*. *recfn a g* $\wedge$ *g* $\simeq$ *f*} (**is** *infinite ?R*)
⟨*proof*⟩

1.1.3 Primitive recursive and total functions

fun *Mn-free* :: *recf* $\Rightarrow$ *bool* **where**
Mn-free Z = *True*
| *Mn-free S* = *True*
| *Mn-free* (*Id m n*) = *True*
| *Mn-free* (*Cn n f gs*) = (($\forall$ *g* $\in$ *set gs*. *Mn-free g*) $\wedge$ *Mn-free f*)
| *Mn-free* (*Pr n f g*) = (*Mn-free f* $\wedge$ *Mn-free g*)
| *Mn-free* (*Mn n f*) = *False*

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

abbreviation *prim-recfn* :: *nat* $\Rightarrow$ *recf* $\Rightarrow$ *bool* **where**
prim-recfn n f $\equiv$ *recfn n f* $\wedge$ *Mn-free f*

definition *total* :: *recf* $\Rightarrow$ *bool* **where**
total f $\equiv$ $\forall$ *xs*. *length xs* = *arity f* $\longrightarrow$ *eval f xs* $\downarrow$

lemma *totalI* [*intro*]:
assumes $\bigwedge$*xs*. *length xs* = *arity f* $\Longrightarrow$ *eval f xs* $\downarrow$
shows *total f*
⟨*proof*⟩

lemma *totalE* [*simp*]:
assumes *total f* **and** *recfn n f* **and** *length xs* = *n*
shows *eval f xs* $\downarrow$
⟨*proof*⟩

lemma *totalI1* :

assumes *recfn 1 f* **and** $\bigwedge x.$ *eval f* [*x*] ↓
shows *total f*
⟨*proof*⟩

lemma *totalI2*:
assumes *recfn 2 f* **and** $\bigwedge x\ y.$ *eval f* [*x*, *y*] ↓
shows *total f*
⟨*proof*⟩

lemma *totalI3*:
assumes *recfn 3 f* **and** $\bigwedge x\ y\ z.$ *eval f* [*x*, *y*, *z*] ↓
shows *total f*
⟨*proof*⟩

lemma *totalI4*:
assumes *recfn 4 f* **and** $\bigwedge w\ x\ y\ z.$ *eval f* [*w*, *x*, *y*, *z*] ↓
shows *total f*
⟨*proof*⟩

lemma *Mn-free-imp-total* [*intro*]:
assumes *wellf f* **and** *Mn-free f*
shows *total f*
⟨*proof*⟩

lemma *prim-recfn-total*: *prim-recfn n f* ⟹ *total f*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Pr-total*:
assumes *total f* **and** *total g* **and** *recfn* (*Suc n*) (*Pr n f g*)
shows *total* (*Pr n f g*)
⟨*proof*⟩

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)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *r-dummy-recfn* [*simp*]:
  recfn a f ⟹ *recfn* (*a* + *n*) (*r-dummy n f*)
  ⟨*proof*⟩

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

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*⟩

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*)
  ⟨*proof*⟩

lemma *r-shrink-recfn* [*simp*]: *recfn 2 f* ⟹ *recfn 1* (*r-shrink f*)
  ⟨*proof*⟩

lemma *r-shrink* [*simp*]: *recfn 2 f* ⟹ *eval* (*r-shrink f*) [*x*] = *eval f* [*x*, *x*]
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *r-swap-prim* [*simp*]: *prim-recfn 2 f* ⟹ *prim-recfn 2* (*r-swap f*)
  ⟨*proof*⟩

lemma *r-swap* [*simp*]: *recfn 2 f* ⟹ *eval* (*r-swap f*) [*x*, *y*] = *eval f* [*y*, *x*]
  ⟨*proof*⟩

Prepending one dummy parameter:

definition *r-shift* :: *recf* ⇒ *recf* **where**
r-shift f ≡ *Cn* (*Suc* (*arity f*)) *f* (*map* (*λi. 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*)
⟨*proof*⟩

lemma *r-shift-recfn* [*simp*]: *recfn a f* ⟹ *recfn* (*Suc a*) (*r-shift f*)
⟨*proof*⟩

lemma *r-shift* [*simp*]:
assumes *recfn* (*length xs*) *f*
shows *eval* (*r-shift f*) (*x* # *xs*) = *eval f xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-const* [*simp*]: *eval* (*r-const c*) [*x*] ↓= *c*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-constn* [*simp*]: *length xs* = *Suc n* ⟹ *eval* (*r-constn n c*) *xs* ↓= *c*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-add* [*simp*]: *eval r-add* [*a*, *b*] ↓= *a* + *b*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-mul* [*simp*]: *eval r-mul* [*a*, *b*] ↓= *a* * *b*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-dec* [*simp*]: *eval r-dec* [*a*] ↓= *a* − *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-sub* [*simp*]: *eval r-sub* [*a*, *b*] ↓= *a* − *b*
⟨*proof*⟩

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

lemma *r-sign-prim* [*simp*]: *prim-recfn 1 r-sign*
⟨*proof*⟩

lemma *r-sign* [*simp*]: *eval r-sign* [*x*] ↓= (*if x* = *0 then 0 else 1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-not* [*simp*]: *eval r-not* [*x*] ↓= (*if x* = *0 then 1 else 0*)
⟨*proof*⟩

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

lemma *r-nand-prim* [*simp*]: *prim-recfn 2 r-nand*
⟨*proof*⟩

lemma *r-nand* [*simp*]: *eval r-nand* [*x*, *y*] ↓= (*if x* = *0* ∧ *y* = *0 then 1 else 0*)
⟨*proof*⟩

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

lemma *r-and-prim* [*simp*]: *prim-recfn 2 r-and*
⟨*proof*⟩

lemma *r-and* [*simp*]: *eval r-and* [*x*, *y*] ↓= (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
⟨*proof*⟩

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

lemma *r-or-prim* [*simp*]: *prim-recfn 2 r-or*
⟨*proof*⟩

lemma *r-or* [*simp*]: *eval r-or* [*x*, *y*] ↓= (*if x* = *0* ∨ *y* = *0 then 0 else 1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-ifz* [*simp*]: *eval r-ifz* [*cond*, *val0*, *val1*] ↓= (*if cond* = *0 then val0 else val1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-eq* [*simp*]: *eval r-eq* [*x*, *y*] ↓= (*if x* = *y then 0 else 1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-ifeq* [*simp*]: *eval r-ifeq* [*a*, *b*, v_0, v_1] ↓= (*if a* = *b then* v_0 *else* v_1)
⟨*proof*⟩

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

lemma *r-neq-prim* [*simp*]: *prim-recfn 2 r-neq*
⟨*proof*⟩

lemma *r-neq* [*simp*]: *eval r-neq* [*x*, *y*] ↓= (*if x* = *y then 1 else 0*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-ifle* [*simp*]: *eval r-ifle* [*a*, *b*, v_0, v_1] ↓= (*if a* ≤ *b then* v_0 *else* v_1)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-ifless* [*simp*]: *eval r-ifless* [*a*, *b*, v_0, v_1] ↓= (*if a* < *b then* v_0 *else* v_1)
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *r-ifeq-else-diverg* [*simp*]:
eval r-ifeq-else-diverg [*a*, *b*, *v*] = (*if* $a = b$ *then Some v else None*)
⟨*proof*⟩

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* $x \in X$ *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*⟩

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*
⟨*proof*⟩

lemma *r-triangle*: *eval r-triangle* [*n*] ↓= *Sum* {*0..n*}
⟨*proof*⟩

lemma *r-triangle-eq-triangle* [*simp*]: *eval r-triangle* [*n*] ↓= *triangle n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-prod-encode* [*simp*]: *eval r-prod-encode* [*m*, *n*] ↓= *prod-encode* (*m*, *n*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pdec2-le*: *pdec2 i* ≤ *i*
⟨*proof*⟩

lemma *pdec-less*: *pdec2 i* < *Suc i*
⟨*proof*⟩

lemma *pdec1-zero*: *pdec1 0* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-maxletr*: *eval r-maxletr* [x_0, x_1] ↓= (*GREATEST y*. *y* ≤ x_0 ∧ *triangle y* ≤ x_1)
⟨*proof*⟩

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

lemma *r-maxlt-prim*: *prim-recfn 1 r-maxlt*

⟨*proof*⟩

lemma *r-maxlt*: *eval r-maxlt* [*e*] ↓= (*GREATEST y. triangle y* ≤ *e*)
⟨*proof*⟩

definition *pdec1′ e* ≡ *e* − *triangle* (*GREATEST y. triangle y* ≤ *e*)

definition *pdec2′ e* ≡ (*GREATEST y. triangle y* ≤ *e*) − *pdec1′ e*

lemma *max-triangle-bound*: *triangle z* ≤ *e* ⟹ *z* ≤ *e*
⟨*proof*⟩

lemma *triangle-greatest-le*: *triangle* (*GREATEST y. triangle y* ≤ *e*) ≤ *e*
⟨*proof*⟩

lemma *prod-encode-pdec′*: *prod-encode* (*pdec1′ e*, *pdec2′ e*) = *e*
⟨*proof*⟩

lemma *pdec′*:
pdec1′ e = *pdec1 e*
pdec2′ e = *pdec2 e*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-pdec1* [*simp*]: *eval r-pdec1* [*e*] ↓= *pdec1 e*
⟨*proof*⟩

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

lemma *r-pdec2-prim* [*simp*]: *prim-recfn 1 r-pdec2*
⟨*proof*⟩

lemma *r-pdec2* [*simp*]: *eval r-pdec2* [*e*] ↓= *pdec2 e*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-pdec12* [*simp*]: *eval r-pdec12* [*e*] ↓= *pdec12 e*
⟨*proof*⟩

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

lemma *r-pdec22-prim* [*simp*]: *prim-recfn 1 r-pdec22*
⟨*proof*⟩

lemma *r-pdec22* [*simp*]: *eval r-pdec22* [*e*] ↓= *pdec22 e*
⟨*proof*⟩

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

lemma *r-pdec122-prim* [*simp*]: *prim-recfn 1 r-pdec122*
⟨*proof*⟩

lemma *r-pdec122* [*simp*]: *eval r-pdec122* [*e*] ↓= *pdec122 e*
⟨*proof*⟩

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

lemma *r-pdec222-prim* [*simp*]: *prim-recfn 1 r-pdec222*
⟨*proof*⟩

lemma *r-pdec222* [*simp*]: *eval r-pdec222* [*e*] ↓= *pdec222 e*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *tuple-encode-decode′* [*simp*]: *tuple-encode n* (*tuple-decode n i*) = *i*
⟨*proof*⟩

lemma *tuple-decode-encode*:
assumes *length xs* > *0*
shows *tuple-decode* (*length xs* − *1*) (*tuple-encode* (*length xs* − *1*) *xs*) = *xs*
⟨*proof*⟩

lemma *tuple-decode-encode′* [*simp*]:
assumes *length xs* = *Suc n*
shows *tuple-decode n* (*tuple-encode n xs*) = *xs*
⟨*proof*⟩

lemma *tuple-decode-length* [*simp*]: *length* (*tuple-decode n i*) = *Suc n*
⟨*proof*⟩

lemma *tuple-decode-nonzero*:
assumes *n* > *0*
shows *tuple-decode n i* = *pdec1 i* # *tuple-decode* (*n* − *1*) (*pdec2 i*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-tuple-encode*:
assumes *length xs* = *Suc n*
shows *eval* (*r-tuple-encode n*) *xs* ↓= *tuple-encode n xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *e-tuple-nth-gr* [*simp*]: *n* > *a* ⟹ *e-tuple-nth a i n* = *0*
⟨*proof*⟩

lemma *tuple-decode-pdec2*: *tuple-decode a* (*pdec2 es*) = *tl* (*tuple-decode* (*Suc a*) *es*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *iterate-additive'*: *iterate* (t_1 + t_2) *f x* = *iterate* t_2 *f* (*iterate* t_1 *f x*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *r-nth-inbounds*:
$k \le a \Longrightarrow$ *eval r-nth-inbounds* $[a, i, k] \downarrow=$ *e-tuple-nth a i k*
eval r-nth-inbounds $[a, i, k] \downarrow$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-tuple-nth* [*simp*]: *eval r-tuple-nth* $[a, i, k] \downarrow=$ *e-tuple-nth a i k*
⟨*proof*⟩

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* = []
⟨*proof*⟩

lemma *list-encode-1*: *list-encode* [*0*] = *1*
⟨*proof*⟩

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*⟩

lemma *list-decode-encode* [*simp*]: *list-decode* (*list-encode xs*) = *xs*
⟨*proof*⟩

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

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

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-singleton-encode* [*simp*]: *eval r-singleton-encode* [*x*] ↓= *singleton-encode x*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-list-encode*:
assumes *length xs* = *Suc n*
shows *eval* (*r-list-encode n*) *xs* ↓= *list-encode xs*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *r-tl* [*simp*]: *eval r-tl* [*e*] ↓= *e-tl e*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *e-hd-0* [*simp*]: *e-hd 0 = 0*
  ⟨*proof*⟩

lemma *e-hd-neq-0* [*simp*]:
  assumes *e ≠ 0*
  shows *e-hd e = hd (list-decode e)*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-hd* [*simp*]: *eval r-hd [e] ↓= e-hd e*
⟨*proof*⟩

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

lemma *e-length-0*: *e-length e = 0 ⟹ e = 0*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-length* [*simp*]: *eval r-length [e] ↓= e-length e*
  ⟨*proof*⟩

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)
  ⟨*proof*⟩

lemma *e-hd-nth0*: *e-hd e = e-nth e 0*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-nth* [*simp*]: *eval r-nth* [*e*, *n*] ↓= *e-nth e n*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-rev-aux-total*: *eval r-rev-aux* [*i*, *e*] ↓
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-rev* [*simp*]: *eval r-rev* [*e*] ↓= *list-encode* (*rev* (*list-decode e*))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *r-cons-else*:

eval r-cons-else [*e*, *es*] ↓=
  Suc (*prod-encode* (*e-length es*, *prod-encode* (*e*, *pdec2* (*es* − *1*))))
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-cons* [*simp*]: *eval r-cons* [*e*, *es*] ↓= *e-cons e es*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *e-hd-snoc* [*simp*]:
  assumes *e-length b* > *0*
  shows *e-hd* (*e-snoc b x*) = *e-hd b*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-snoc* [*simp*]: *eval r-snoc* [*es*, *e*] ↓= *e-snoc es e*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-take*:
  assumes *x* = *list-encode es*
  shows *eval r-take* [*n*, *x*] ↓= *list-encode* (*take n es*)
⟨*proof*⟩

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

⟨*proof*⟩

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

lemma *r-last-prim* [*simp*]: *prim-recfn 1 r-last*
⟨*proof*⟩

lemma *r-last* [*simp*]:
assumes *e* = *list-encode xs* **and** *length xs* > *0*
shows *eval r-last* [*e*] ↓= *last xs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-update* [*simp*]: *eval r-update* [*b*, *j*, *v*] ↓= *e-update b j v*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *e-nth-append-small*:
assumes *n* < *e-length xs*
shows *e-nth* (*e-append xs ys*) *n* = *e-nth xs n*
⟨*proof*⟩

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*)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-append* [*simp*]: *eval r-append* [*a*, *b*] ↓= *e-append a b*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *e-nth-append-zeros-big*:
assumes *i* ≥ *e-length b*
shows *e-nth* (*e-append-zeros b z*) *i* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-append-zeros*: *eval r-append-zeros* [*b*, *z*] ↓= *e-append-zeros b z*
⟨*proof*⟩

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 $nat \Rightarrow nat$ 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 × nat list × nat list*

type-synonym *configuration = frame list × 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*
  ⟨*proof*⟩

lemma *reachable-transitive* [*trans*]:
  assumes *reachable x y* **and** *reachable y z*
  shows *reachable x z*
  ⟨*proof*⟩

lemma *reachable-refl*: *reachable x x*
⟨*proof*⟩

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*) = []
⟨*proof*⟩

lemma *reachable-empty-stack*:
assumes *fst x* = [] **and** *reachable x y*
shows *fst y* = []
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *valid-ConsE*: *valid* ((*f*, *xs*, *locs*) # *rest*) ⟹ *valid rest* ∧ *recfn* (*length xs*) *f*
⟨*proof*⟩

lemma *valid-ConsI*: *valid rest* ⟹ *recfn* (*length xs*) *f* ⟹ *valid* ((*f*, *xs*, *locs*) # *rest*)
⟨*proof*⟩

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*⟩

corollary *iterate-step-valid*:
assumes *valid stack*
shows *valid* (*fst* (*iterate t step* (*stack*, *rv*)))
⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*⟩

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* (λ*y*. *the* (*eval* (*Pr n f g*) (*y* # *xs*))) (*rev* [*0*..<*Suc x*])

lemma *trace-length*: *length* (*trace n f g xs x*) = *Suc x*
  ⟨*proof*⟩

lemma *trace-hd*: *hd* (*trace n f g xs x*) = *the* (*eval* (*Pr n f g*) (*x* # *xs*))
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *reachable-Pr*:
  assumes *valid* (((*Pr n f g*), *x* # *xs*, []) # *rest*) (**is** *valid ?stack*)
    and $\bigwedge$*xs rest*. *valid* ((*f*, *xs*, []) # *rest*) $\Longrightarrow$ *correct* ((*f*, *xs*, []) # *rest*, *None*)
    and $\bigwedge$*xs rest*. *valid* ((*g*, *xs*, []) # *rest*) $\Longrightarrow$ *correct* ((*g*, *xs*, []) # *rest*, *None*)
    and $y \leq x$
    and *eval* (*Pr n f g*) (*y* # *xs*) $\downarrow$
  shows *reachable* (*?stack*, *None*) ((*Pr n f g*, *x* # *xs*, *trace n f g xs y*) # *rest*, *None*)
  ⟨*proof*⟩

lemma *step-Pr-correct*:
  assumes *valid* (((*Pr n f g*), *xs*, []) # *rest*) (**is** *valid ?stack*)
    and $\bigwedge$*xs rest*. *valid* ((*f*, *xs*, []) # *rest*) $\Longrightarrow$ *correct* ((*f*, *xs*, []) # *rest*, *None*)
    and $\bigwedge$*xs rest*. *valid* ((*g*, *xs*, []) # *rest*) $\Longrightarrow$ *correct* ((*g*, *xs*, []) # *rest*, *None*)
  shows *correct* (*?stack*, *None*)
⟨*proof*⟩

lemma *reachable-Mn*:
  assumes *valid* ((*Mn n f*, *xs*, []) # *rest*) (**is** *valid ?stack*)
    and $\bigwedge$*xs rest*. *valid* ((*f*, *xs*, []) # *rest*) $\Longrightarrow$ *correct* ((*f*, *xs*, []) # *rest*, *None*)
    and $\forall y<z$. *eval f* (*y* # *xs*) $\notin$ {*None*, *Some 0*}
  shows *reachable* (*?stack*, *None*) ((*f*, *z* # *xs*, []) # (*Mn n f*, *xs*, [*z*]) # *rest*, *None*)
  ⟨*proof*⟩

lemma *iterate-step-empty-stack*: *iterate t step* ([], *rv*) = ([], *rv*)
  ⟨*proof*⟩

lemma *reachable-iterate-step-empty-stack*:
  assumes *reachable cfg* ([], *rv*)
  shows $\exists t$. *iterate t step cfg* = ([], *rv*) $\wedge$ ($\forall t'<t$. *fst* (*iterate t' step cfg*) $\neq$ [])
⟨*proof*⟩

lemma *step-Mn-correct*:
  assumes *valid* ((*Mn n f*, *xs*, []) # *rest*) (**is** *valid ?stack*)
    and $\bigwedge$*xs rest*. *valid* ((*f*, *xs*, []) # *rest*) $\Longrightarrow$ *correct* ((*f*, *xs*, []) # *rest*, *None*)
  shows *correct* (*?stack*, *None*)
⟨*proof*⟩

theorem *step-correct*:
  assumes *valid* ((*f*, *xs*, []) # *rest*)
  shows *correct* ((*f*, *xs*, []) # *rest*, *None*)
  ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *encode-not-Z-or-S*: *encode f = prod-encode (a, b) ⟹ a > 1 ⟹ f ≠ Z ∧ f ≠ S*
⟨*proof*⟩

lemma *encode-injective*: *encode f = encode g ⟹ f = g*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *encode-kind-1*: *encode-kind (encode S) = 1*
⟨*proof*⟩

lemma *encode-kind-2*: *encode-kind (encode (Id m n)) = 2*
⟨*proof*⟩

lemma *encode-kind-3*: *encode-kind (encode (Cn n f gs)) = 3*
⟨*proof*⟩

lemma *encode-kind-4*: *encode-kind (encode (Pr n f g)) = 4*
⟨*proof*⟩

lemma *encode-kind-5*: *encode-kind (encode (Mn n f)) = 5*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *encode-kind-Pr*:
assumes *encode-kind (encode f) = 4*
shows *∃ n f′ g. f = Pr n f′ g*
⟨*proof*⟩

lemma *encode-kind-Mn*:
assumes *encode-kind (encode f) = 5*
shows *∃ n g. f = Mn n g*
⟨*proof*⟩

lemma *pdec2-encode-Id*: *pdec2 (encode (Id m n)) = prod-encode (m, n)*
⟨*proof*⟩

lemma *pdec2-encode-Pr*: *pdec2* (*encode* (*Pr n f g*)) = *triple-encode n* (*encode f*) (*encode g*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *e2stack-stack* [*simp*]:
e2stack (*encode-config* (*ss*, *rv*)) = *list-encode* (*map encode-frame ss*)
⟨*proof*⟩

lemma *e2tail-tail* [*simp*]:
e2tail (*encode-config* (*s* # *ss*, *rv*)) = *list-encode* (*map encode-frame ss*)
⟨*proof*⟩

lemma *e2frame-frame* [*simp*]:
e2frame (*encode-config* (*s* # *ss*, *rv*)) = *encode-frame s*
⟨*proof*⟩

lemma *e2i-f* [*simp*]:
e2i (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *encode f*
⟨*proof*⟩

lemma *e2xs-xs* [*simp*]:
e2xs (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *list-encode xs*
⟨*proof*⟩

lemma *e2ls-ls* [*simp*]:
e2ls (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *list-encode ls*
⟨*proof*⟩

lemma *e2lenas-lenas* [*simp*]:
e2lenas (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *length xs*
⟨*proof*⟩

lemma *e2lenls-lenls* [*simp*]:
e2lenls (*encode-config* ((*f*, *xs*, *ls*) # *ss*, *rv*)) = *length ls*
⟨*proof*⟩

lemma *e2stack-0-iff-Nil*:
assumes *e* = *encode-config* (*ss*, *rv*)
shows *e2stack e* = *0* ⟷ *ss* = []
⟨*proof*⟩

lemma *e2ls-0-iff-Nil* [*simp*]: *list-decode* (*e2ls e*) = [] ⟷ *e2ls e* = *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
 ⟨*proof*⟩

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*⟩

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*)
 ⟨*proof*⟩

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*⟩

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*⟩

lemma *estep*:
  assumes *valid* (*fst c*)
  shows *estep* (*encode-config c*) = *encode-config* (*step c*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-e2stack* [*simp*]: *eval r-e2stack* [*e*] ↓= *e2stack e*
  ⟨*proof*⟩

definition *r-e2rv* ≡ *r-pdec2*

lemma *r-e2rv-prim*: *prim-recfn 1 r-e2rv*
  ⟨*proof*⟩

lemma *r-e2rv* [*simp*]: *eval r-e2rv* [*e*] ↓= *e2rv e*
  ⟨*proof*⟩

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

lemma *r-e2tail-prim*: *prim-recfn 1 r-e2tail*
  ⟨*proof*⟩

lemma *r-e2tail* [*simp*]: *eval r-e2tail* [*e*] ↓= *e2tail e*
  ⟨*proof*⟩

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

lemma *r-e2frame-prim*: *prim-recfn 1 r-e2frame*
  ⟨*proof*⟩

lemma *r-e2frame* [*simp*]: *eval r-e2frame* [*e*] ↓= *e2frame e*
  ⟨*proof*⟩

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

lemma *r-e2i-prim*: *prim-recfn 1 r-e2i*
  ⟨*proof*⟩

lemma *r-e2i* [*simp*]: *eval r-e2i* [*e*] ↓= *e2i e*
  ⟨*proof*⟩

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

lemma *r-e2xs-prim*: *prim-recfn 1 r-e2xs*
  ⟨*proof*⟩

lemma *r-e2xs* [*simp*]: *eval r-e2xs* [*e*] ↓= *e2xs e*

⟨*proof*⟩

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

lemma *r-e2ls-prim*: *prim-recfn 1 r-e2ls*
⟨*proof*⟩

lemma *r-e2ls* [*simp*]: *eval r-e2ls* [*e*] ↓= *e2ls e*
⟨*proof*⟩

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

lemma *r-e2lenls-prim*: *prim-recfn 1 r-e2lenls*
⟨*proof*⟩

lemma *r-e2lenls* [*simp*]: *eval r-e2lenls* [*e*] ↓= *e2lenls e*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-kind*: *eval r-kind* [*e*] ↓= *encode-kind e*
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-step-Cn*: *eval r-step-Cn* [*e*] ↓= *estep-Cn e*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-step-Pr*: *eval r-step-Pr* [*e*] ↓= *estep-Pr e*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-step-Mn*: *eval r-step-Mn* [*e*] ↓= *estep-Mn e*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-step*: *eval r-step* [*e*] ↓= *estep e*
  ⟨*proof*⟩

theorem *r-step-equiv-step*:
  assumes *valid* (*fst c*)
  shows *eval r-step* [*encode-config c*] ↓= *encode-config* (*step c*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-leap-total*: *eval r-leap* [*t*, *i*, *x*] ↓
  ⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-result-total*: *total r-result*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*⟩

lemma *r-result-converg*:
  assumes *i* = *encode f*
    and *recfn* (*e-length x*) *f*
    and *eval f* (*list-decode x*) ↓= *v*
  shows ∃ *t*.
    (∀ $t' \geq t$. *eval r-result* [t', *i*, *x*] ↓= *Suc v*) ∧
    (∀ $t' < t$. *eval r-result* [t', *i*, *x*] ↓= *0*)
⟨*proof*⟩

lemma *r-result-diverg*:
  assumes $i = encode\ f$
    and $recfn\ (e\text{-}length\ x)\ f$
    and $eval\ f\ (list\text{-}decode\ x) \uparrow$
  shows $eval\ r\text{-}result\ [t,\ i,\ x] \downarrow= 0$
⟨*proof*⟩

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*
  ⟨*proof*⟩

theorem *r-univ*:
  assumes $i = encode\ f$ **and** $recfn\ (e\text{-}length\ x)\ f$
  shows $eval\ r\text{-}univ\ [i,\ x] = eval\ f\ (list\text{-}decode\ x)$
⟨*proof*⟩

theorem *r-univ′*:
  assumes $recfn\ (e\text{-}length\ x)\ f$
  shows $eval\ r\text{-}univ\ [encode\ f,\ x] = eval\ f\ (list\text{-}decode\ x)$
  ⟨*proof*⟩

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

definition *r-universal* :: *nat* ⇒ *recf* **where**
  $r\text{-}universal\ n \equiv Cn\ (Suc\ n)\ r\text{-}univ\ [Id\ (Suc\ n)\ 0,\ r\text{-}shift\ (r\text{-}list\text{-}encode\ (n - 1))]$

lemma *r-universal-recfn* [*simp*]: $n > 0 \Longrightarrow recfn\ (Suc\ n)\ (r\text{-}universal\ n)$
  ⟨*proof*⟩

lemma *r-universal*:
  assumes $recfn\ n\ f$ **and** $length\ xs = n$
  shows $eval\ (r\text{-}universal\ n)\ (encode\ f\ \#\ xs) = eval\ f\ xs$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-result1-total*: *total r-result1*
  ⟨*proof*⟩

lemma *r-result1* [*simp*]:
  $eval\ r\text{-}result1\ [t,\ i,\ x] = eval\ r\text{-}result\ [t,\ i,\ singleton\text{-}encode\ x]$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

theorem *r-phi*:
  assumes *i* = *encode f* **and** *recfn 1 f*
  shows *eval r-phi* [*i*, *x*] = *eval f* [*x*]
  ⟨*proof*⟩

corollary *r-phi'*:
  assumes *recfn 1 f*
  shows *eval r-phi* [*encode f*, *x*] = *eval f* [*x*]
  ⟨*proof*⟩

lemma *r-phi''*: *eval r-phi* [*i*, *x*] = *eval r-univ* [*i*, *singleton-encode x*]
  ⟨*proof*⟩

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 j_1 j_2 ≡
    Cn (*Suc* (*Suc n*)) *r-univ*
     [*Cn* (*Suc* (*Suc n*)) *r-ifz* [*Id* (*Suc* (*Suc n*)) *0*, *r-constn* (*Suc n*) j_1, *r-constn* (*Suc n*) j_2],
      r-shift (*r-list-encode n*)]

lemma *r-lazyifzero-recfn*: *recfn* (*Suc* (*Suc n*)) (*r-lazyifzero n* j_1 j_2)
  ⟨*proof*⟩

lemma *r-lazyifzero*:
  assumes *length xs* = *Suc n*
    and j_1 = *encode* f_1
    and j_2 = *encode* f_2
    and *recfn* (*Suc n*) f_1
    and *recfn* (*Suc n*) f_2
  shows *eval* (*r-lazyifzero n* j_1 j_2) (*c* # *xs*) = (*if c* = *0 then eval* f_1 *xs else eval* f_2 *xs*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *r-result-saturating*:
assumes *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
shows *eval r-result* [*t* + *d*, *i*, *x*] ↓= *Suc v*
⟨*proof*⟩

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*⟩

lemma *r-result-diverg'*:
assumes *eval r-univ* [*i*, *x*] ↑
shows *eval r-result* [*t*, *i*, *x*] ↓= *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-result-Some'*:
assumes *eval r-result* [*t*, *i*, *x*] ↓= *Suc v*
shows *eval r-univ* [*i*, *x*] ↓= *v*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *r-result1-diverg'*:

assumes *eval r-phi* $[i, x] \uparrow$
shows *eval r-result1* $[t, i, x] \downarrow= 0$
⟨*proof*⟩

lemma *r-result1-Some′*:
assumes *eval r-result1* $[t, i, x] \downarrow=$ *Suc v*
shows *eval r-phi* $[i, x] \downarrow= v$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-dovetail*:
eval r-dovetail $[i, z] \downarrow=$
(*if the* (*eval r-result1* [*pdec2 z*, *i*, *pdec1 z*]) > *0 then 0 else 1*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-dovedelay*:
eval r-dovedelay $[z, i, y] \downarrow=$
(*if the* (*eval r-result1* [*pdec2 z*, *i*, *pdec1 z*]) > *0* ∧ *y* ≤ *z then 0 else 1*)
⟨*proof*⟩

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

lemma *r-enumdom-recfn* [*simp*]: *recfn 2 r-enumdom*
⟨*proof*⟩

lemma *r-enumdom* [*simp*]:
eval r-enumdom $[i, y] =$
(*if* ∃ *z*. *eval r-dovedelay* $[z, i, y] \downarrow= 0$
then Some (*pdec1* (*LEAST z*. *eval r-dovedelay* $[z, i, y] \downarrow= 0$))
else None)

⟨*proof*⟩

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

lemma *r-enumdom-empty-domain*:
  assumes $\bigwedge x.$ *eval r-phi* $[i, x] \uparrow$
  shows $\bigwedge y.$ *eval r-enumdom* $[i, y] \uparrow$
  ⟨*proof*⟩

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* $[i, x_0] \downarrow$
  shows $\bigwedge y.$ *eval r-enumdom* $[i, y] \downarrow$
    and $\bigwedge x.$ *eval r-phi* $[i, x] \downarrow \longleftrightarrow (\exists y.$ *eval r-enumdom* $[i, y] \downarrow= x)$
⟨*proof*⟩

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* $[i, x_0] \downarrow$
  shows $\exists g.$ *recfn 1 g* $\wedge$ *total g* $\wedge (\forall x.$ *eval r-phi* $[i, x] \downarrow \longleftrightarrow (\exists y.$ *eval g* $[y] \downarrow= x))$
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-both*:
  assumes $\bigwedge x.$ *eval r-phi* $[i, x] =$ *eval f* $[x]$
    and $\bigwedge x.$ *eval r-phi* $[j, x] =$ *eval g* $[x]$
  shows *eval f* $[x] \uparrow \wedge$ *eval g* $[x] \uparrow \Longrightarrow$ *eval r-both* $[t, i, j, x] \downarrow=$ *prod-encode* $(2, 0)$
    and ⟦*eval r-result1* $[t, i, x] \downarrow= 0$; *eval r-result1* $[t, j, x] \downarrow= 0$⟧ $\Longrightarrow$
      eval r-both $[t, i, j, x] \downarrow=$ *prod-encode* $(2, 0)$
    and *eval r-result1* $[t, i, x] \downarrow=$ *Suc v* $\Longrightarrow$
      eval r-both $[t, i, j, x] \downarrow=$ *prod-encode* $(0,$ *the* $($*eval f* $[x]))$
    and ⟦*eval r-result1* $[t, i, x] \downarrow= 0$; *eval r-result1* $[t, j, x] \downarrow=$ *Suc v*⟧ $\Longrightarrow$
      eval r-both $[t, i, j, x] \downarrow=$ *prod-encode* $(1,$ *the* $($*eval g* $[x]))$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-parallel*:
assumes $\bigwedge x.$ *eval r-phi [i, x] = eval f [x]*
and $\bigwedge x.$ *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*⟩

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])
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-univ-exteq-r-normal-form*: *r-univ* ≃ *r-normal-form*
⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

The normal form has only one μ-recursion.

lemma *count-Mn-normal-form*: *count-Mn r-normal-form* = *1*

⟨*proof*⟩

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

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

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

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-code-const1-aux*:
eval r-code-const1-aux [*i*, *r*, *c*] ↓= *quad-encode 3 1 1* (*singleton-encode r*)
⟨*proof*⟩

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

lemma *r-code-const1-prim*: *prim-recfn 1 r-code-const1*
⟨*proof*⟩

lemma *r-code-const1*: *eval r-code-const1* [*c*] ↓= *code-const1 c*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *r-code-constn*: *eval (r-code-constn n) [c] ↓= code-constn n c*
⟨*proof*⟩

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*
⟨*proof*⟩

The functions s_n^m 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*⟩

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*)
  ⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

lemma *r-smn*:
  assumes *n* > *0* **and** *length cs* = *m*
  shows *eval* (*r-smn n m*) (*p* # *cs*) ↓= *smn n p cs*
  ⟨*proof*⟩

lemma *map-eval-Some-the*:
  assumes *map* (*λg. eval g xs*) *gs* = *map Some ys*
  shows *map* (*λg. the* (*eval g xs*)) *gs* = *ys*
  ⟨*proof*⟩

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*⟩

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*))

⟨*proof*⟩

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
$\forall\, i\ x.\ eval\ psi\ [i,\ x] = eval\ r\text{-}phi\ [the\ (eval\ c\ [i]),\ x]$
⟨*proof*⟩

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*: $inj\ (\lambda x.\ smn\ 1\ p\ [x])$
⟨*proof*⟩

definition $r\text{-}univuniv \equiv Cn\ 2\ r\text{-}phi\ [Cn\ 2\ r\text{-}phi\ [Id\ 2\ 0,\ Id\ 2\ 0],\ Id\ 2\ 1]$

lemma *r-univuniv-recfn*: *recfn 2 r-univuniv*
⟨*proof*⟩

lemma *r-univuniv-converg*:
assumes $eval\ r\text{-}phi\ [x,\ x] \downarrow$
shows $eval\ r\text{-}univuniv\ [x,\ y] = eval\ r\text{-}phi\ [the\ (eval\ r\text{-}phi\ [x,\ x]),\ y]$
⟨*proof*⟩

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 $\exists\, n \geq k.\ \forall\, x.\ eval\ r\text{-}phi\ [n,\ x] = eval\ r\text{-}phi\ [the\ (eval\ f\ [n]),\ x]$
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*]
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *decidable-complement*: *decidable X* ⟹ *decidable* (− *X*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *r-contains*: *eval* (*r-contains xs*) [*x*] ↓= (*if x* ∈ *set xs then 1 else 0*)
⟨*proof*⟩

lemma *finite-set-decidable*: *finite X* ⟹ *decidable X*
⟨*proof*⟩

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*⟩

lemma *decidable-imp-semidecidable*: *decidable X* ⟹ *semidecidable X*
⟨*proof*⟩

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*⟩

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*⟩

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *rice-theorem*:

assumes *index-set I* **and** $I \neq UNIV$ **and** $I \neq \{\}$
shows ¬ *decidable I*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *total2I* [*intro*]: (⋀*x y*. *f x y* ↓) ⟹ *total2 f*
⟨*proof*⟩

lemma *total1E* [*dest*, *simp*]: *total1 f* ⟹ *f x* ↓
⟨*proof*⟩

lemma *total2E* [*dest*, *simp*]: *total2 f* ⟹ *f x y* ↓
⟨*proof*⟩

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**
$\mathcal{P}^2$ ≡ {λ*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*]: $f \in \mathcal{R} \Longrightarrow f \in \mathcal{P}$
⟨*proof*⟩

lemma *R2-imp-P2* [*simp*, *elim*]: $f \in \mathcal{R}^2 \Longrightarrow f \in \mathcal{P}^2$
⟨*proof*⟩

lemma *Prim1-imp-R1* [*simp*, *elim*]: $f \in Prim1 \Longrightarrow f \in \mathcal{R}$
⟨*proof*⟩

lemma *Prim2-imp-R2* [*simp*, *elim*]: $f \in Prim2 \Longrightarrow f \in \mathcal{R}^2$
⟨*proof*⟩

lemma *P1E* [*elim*]:
assumes $f \in \mathcal{P}$
obtains *r* **where** *recfn 1 r* **and** $\forall x.\ eval\ r\ [x] = f\ x$
⟨*proof*⟩

lemma *P2E* [*elim*]:
assumes $f \in \mathcal{P}^2$
obtains *r* **where** *recfn 2 r* **and** $\forall x\ y.\ eval\ r\ [x, y] = f\ x\ y$
⟨*proof*⟩

lemma *P1I* [*intro*]:
assumes *recfn 1 r* **and** $(\lambda x.\ eval\ r\ [x]) = f$
shows $f \in \mathcal{P}$
⟨*proof*⟩

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*⟩

lemma *R1I* [*intro*]:
assumes *recfn 1 r* **and** *total r* **and** $\bigwedge x.\ eval\ r\ [x] = f\ x$
shows $f \in \mathcal{R}$
⟨*proof*⟩

lemma *R1E* [*elim*]:
assumes $f \in \mathcal{R}$
obtains *r* **where** *recfn 1 r* **and** *total r* **and** $f = (\lambda x.\ eval\ r\ [x])$
⟨*proof*⟩

lemma *R2I* [*intro*]:

assumes *recfn 2 r* **and** *total r* **and** $\bigwedge x\ y.\ eval\ r\ [x,\ y] = f\ x\ y$
shows $f \in \mathcal{R}^2$
⟨*proof*⟩

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*⟩

lemma *R2E* [*elim*]:
assumes $f \in \mathcal{R}^2$
obtains *r* **where** *recfn 2 r* **and** *total r* **and** $f = (\lambda x_1\ x_2.\ eval\ r\ [x_1,\ x_2])$
⟨*proof*⟩

lemma *R1-imp-total1* [*simp*]: $f \in \mathcal{R} \Longrightarrow total1\ f$
⟨*proof*⟩

lemma *R2-imp-total2* [*simp*]: $f \in \mathcal{R}^2 \Longrightarrow total2\ f$
⟨*proof*⟩

lemma *Prim1I* [*intro*]:
assumes *prim-recfn 1 r* **and** $\bigwedge x.\ f\ x = eval\ r\ [x]$
shows $f \in Prim1$
⟨*proof*⟩

lemma *Prim2I* [*intro*]:
assumes *prim-recfn 2 r* **and** $\bigwedge x\ y.\ f\ x\ y = eval\ r\ [x,\ y]$
shows $f \in Prim2$
⟨*proof*⟩

lemma *P1-total-imp-R1* [*intro*]:
assumes $f \in \mathcal{P}$ **and** *total1 f*
shows $f \in \mathcal{R}$
⟨*proof*⟩

lemma *P2-total-imp-R2* [*intro*]:
assumes $f \in \mathcal{P}^2$ **and** *total2 f*
shows $f \in \mathcal{R}^2$
⟨*proof*⟩

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*⟩

lemma *P2-proj-P1* [*simp*, *elim*]:
assumes $\psi \in \mathcal{P}^2$

shows $\psi\ i \in \mathcal{P}$
⟨*proof*⟩

lemma *R2-proj-R1* [*simp*, *elim*]:
assumes $\psi \in \mathcal{R}^2$
shows $\psi\ i \in \mathcal{R}$
⟨*proof*⟩

lemma *const-in-Prim1*: $(\lambda\text{-}.\ \mathit{Some}\ c) \in \mathit{Prim1}$
⟨*proof*⟩

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

lemma *P1-update-P1*:
assumes $f \in \mathcal{P}$
shows $f(x{:=}z) \in \mathcal{P}$
⟨*proof*⟩

lemma *swap-P2*:
assumes $f \in \mathcal{P}^2$
shows $(\lambda x\ y.\ f\ y\ x) \in \mathcal{P}^2$
⟨*proof*⟩

lemma *swap-R2*:
assumes $f \in \mathcal{R}^2$
shows $(\lambda x\ y.\ f\ y\ x) \in \mathcal{R}^2$
⟨*proof*⟩

lemma *skip-P1*:
assumes $f \in \mathcal{P}$
shows $(\lambda x.\ f\ (x + n)) \in \mathcal{P}$
⟨*proof*⟩

lemma *skip-R1*:
assumes $f \in \mathcal{R}$
shows $(\lambda x.\ f\ (x + n)) \in \mathcal{R}$
⟨*proof*⟩

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**
$\mathit{goedel\text{-}numbering}\ \psi \equiv \psi \in \mathcal{P}^2 \wedge (\forall \chi{\in}\mathcal{P}^2.\ \exists c{\in}\mathcal{R}.\ \forall i.\ \chi\ i = \psi\ (\mathit{the}\ (c\ i)))$

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

⟨*proof*⟩

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

lemma *goedel-numbering-universal*:
assumes *goedel-numbering* ψ **and** $f \in \mathcal{P}$
shows $\exists\, i.\ \psi\ i = f$
⟨*proof*⟩

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

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

lemma *phi-in-P2*: $\varphi \in \mathcal{P}^2$
⟨*proof*⟩

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*⟩

corollary *goedel-numbering-phi*: *goedel-numbering* φ
⟨*proof*⟩

corollary *phi-universal*:
assumes $f \in \mathcal{P}$
obtains i **where** $\varphi\ i = f$
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *prefix-nth* [*simp*]:
assumes *k* < *Suc n*
shows *prefix f n* ! *k* = *the* (*f k*)
⟨*proof*⟩

lemma *prefixI*:
assumes *length vs* > *0* **and** ⋀*x*. *x* < *length vs* ⟹ *f x* ↓= *vs* ! *x*
shows *prefix f* (*length vs* − *1*) = *vs*
⟨*proof*⟩

lemma *prefixI'*:
assumes *length vs* = *Suc n* **and** ⋀*x*. *x* < *Suc n* ⟹ *f x* ↓= *vs* ! *x*
shows *prefix f n* = *vs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *prefix-eqI*:
assumes ⋀*x*. *x* ≤ *n* ⟹ *f x* = *g x*
shows *prefix f n* = *prefix g n*
⟨*proof*⟩

lemma *prefix-0*: *prefix f 0* = [*the* (*f 0*)]
⟨*proof*⟩

lemma *prefix-Suc*: *prefix f* (*Suc n*) = *prefix f n* @ [*the* (*f* (*Suc n*))]
⟨*proof*⟩

lemma *take-prefix*:
assumes *f* ∈ $\mathcal{R}$ **and** *k* ≤ *n*
shows *prefix f k* = *take* (*Suc k*) (*prefix f n*)
⟨*proof*⟩

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* ▷ *n* ≠ *0*
⟨*proof*⟩

lemma *init-prefixE* [*elim*]: *prefix f n* = *prefix g n* ⟹ *f* ▷ *n* = *g* ▷ *n*
⟨*proof*⟩

lemma *init-eqI*:

assumes $\bigwedge x.\ x \le n \Longrightarrow f\ x = g\ x$
shows $f \triangleright n = g \triangleright n$
⟨*proof*⟩

lemma *initI*:
assumes *e-length e* > 0 **and** $\bigwedge x.\ x <$ *e-length e* $\Longrightarrow f\ x \downarrow=$ *e-nth e x*
shows $f \triangleright$ (*e-length e* $- 1$) $= e$
⟨*proof*⟩

lemma *initI'*:
assumes *e-length e* $=$ *Suc n* **and** $\bigwedge x.\ x <$ *Suc n* $\Longrightarrow f\ x \downarrow=$ *e-nth e x*
shows $f \triangleright n = e$
⟨*proof*⟩

lemma *init-iff-list-eq-upto*:
assumes $f \in \mathcal{R}$ **and** *e-length vs* > 0
shows ($\forall x$<*e-length vs*. $f\ x \downarrow=$ *e-nth vs x*) $\longleftrightarrow$ *prefix f* (*e-length vs* $- 1$) $=$ *list-decode vs*
⟨*proof*⟩

lemma *length-init* [*simp*]: *e-length* ($f \triangleright n$) $=$ *Suc n*
⟨*proof*⟩

lemma *init-Suc-snoc*: $f \triangleright$ (*Suc n*) $=$ *e-snoc* ($f \triangleright n$) (*the* (*f* (*Suc n*)))
⟨*proof*⟩

lemma *nth-init*: $i <$ *Suc n* $\Longrightarrow$ *e-nth* ($f \triangleright n$) $i =$ *the* ($f\ i$)
⟨*proof*⟩

lemma *hd-init* [*simp*]: *e-hd* ($f \triangleright n$) $=$ *the* ($f\ 0$)
⟨*proof*⟩

lemma *list-decode-init* [*simp*]: *list-decode* ($f \triangleright n$) $=$ *prefix f n*
⟨*proof*⟩

lemma *init-eq-iff-eq-upto*:
assumes $g \in \mathcal{R}$ **and** $f \in \mathcal{R}$
shows ($\forall j$<*Suc n*. $g\ j = f\ j$) $\longleftrightarrow g \triangleright n = f \triangleright n$
⟨*proof*⟩

definition *is-init-of* :: *nat* $\Rightarrow$ *partial1* $\Rightarrow$ *bool* **where**
is-init-of t f $\equiv \forall i$<*e-length t*. $f\ i \downarrow=$ *e-nth t i*

lemma *not-initial-imp-not-eq*:
assumes $\bigwedge x.\ x <$ *Suc n* $\Longrightarrow f\ x \downarrow$ **and** $\neg$ (*is-init-of* ($f \triangleright n$) g)
shows $f \neq g$
⟨*proof*⟩

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*⟩

corollary *neq-fun-neq-init*:
assumes $f \in \mathcal{R}$ **and** $g \in \mathcal{R}$ **and** $f \neq g$
shows $\exists n.\ f \triangleright n \neq g \triangleright n$
⟨*proof*⟩

lemma *eq-init-forall-le*:
  assumes $f \triangleright n = g \triangleright n$ **and** $m \leq n$
  shows $f \triangleright m = g \triangleright m$
⟨*proof*⟩

corollary *neq-init-forall-ge*:
  assumes $f \triangleright n \neq g \triangleright n$ **and** $m \geq n$
  shows $f \triangleright m \neq g \triangleright m$
  ⟨*proof*⟩

lemma *e-take-init*:
  assumes $f \in \mathcal{R}$ **and** $k < Suc\ n$
  shows $e\text{-}take\ (Suc\ k)\ (f \triangleright n) = f \triangleright k$
  ⟨*proof*⟩

lemma *init-butlast-init*:
  assumes *total1 f* **and** $f \triangleright n = e$ **and** $n > 0$
  shows $f \triangleright (n - 1) = e\text{-}butlast\ e$
⟨*proof*⟩

Some definitions make use of recursive predicates, that is, 01-valued functions.

definition *RPred1* :: *partial1 set* (‹$\mathcal{R}_{01}$›) **where**
  $\mathcal{R}_{01} \equiv \{f.\ f \in \mathcal{R} \land (\forall x.\ f\ x \downarrow= 0 \lor f\ x \downarrow= 1)\}$

lemma *RPred1-subseteq-R1*: $\mathcal{R}_{01} \subseteq \mathcal{R}$
  ⟨*proof*⟩

lemma *const0-in-RPred1*: $(\lambda\text{-}.\ Some\ 0) \in \mathcal{R}_{01}$
  ⟨*proof*⟩

lemma *RPred1-altdef*: $\mathcal{R}_{01} = \{f.\ f \in \mathcal{R} \land (\forall x.\ the\ (f\ x) \leq 1)\}$
  (**is** $\mathcal{R}_{01} = ?S$)
⟨*proof*⟩

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**
  $\psi \in \mathcal{R}^2 \Longrightarrow NUM\text{-}wrt\ \psi \equiv \{U.\ \forall f \in U.\ \exists i.\ \psi\ i = f\}$

lemma *NUM-I* [*intro*]:
  assumes $\psi \in \mathcal{R}^2$ **and** $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f$
  shows $U \in NUM$
  ⟨*proof*⟩

lemma *NUM-E* [*dest*]:
  assumes $U \in NUM$
  shows $U \subseteq \mathcal{R}$
    and $\exists \psi \in \mathcal{R}^2.\ \forall f \in U.\ \exists i.\ \psi\ i = f$
  ⟨*proof*⟩

lemma *NUM-closed-subseteq*:
assumes $U \in NUM$ **and** $V \subseteq U$
shows $V \in NUM$
⟨*proof*⟩

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*⟩

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* ≡
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*
⟨*proof*⟩

lemma *r-prenum* [*simp*]:
eval r-prenum [*e*, *x*] ↓= (*if* $x <$ *e-length e then e-nth e x else 0*)
⟨*proof*⟩

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* $\in \mathcal{R}^2$
⟨*proof*⟩

lemma *prenum* [*simp*]: *prenum e x* ↓= (*if* $x <$ *e-length e then e-nth e x else 0*)
⟨*proof*⟩

lemma *prenum-encode*:
prenum (*list-encode vs*) *x* ↓= (*if* $x <$ *length vs then vs* ! *x else 0*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *prepend-total*: *total1 f* ⟹ *total1* (*vs* ⊙ *f*)
⟨*proof*⟩

lemma *prepend-at-less*:
assumes $n <$ *length vs*
shows (*vs* ⊙ *f*) *n* ↓= *vs* ! *n*
⟨*proof*⟩

lemma *prepend-at-ge*:
assumes $n \geq$ *length vs*
shows (*vs* ⊙ *f*) *n* = *f* ($n -$ *length vs*)

⟨*proof*⟩

lemma *prefix-prepend-less*:
assumes $n < length\ vs$
shows $prefix\ (vs \odot f)\ n = take\ (Suc\ n)\ vs$
⟨*proof*⟩

lemma *prepend-eqI*:
assumes $\bigwedge x.\ x < length\ vs \Longrightarrow g\ x \downarrow= vs\ !\ x$
and $\bigwedge x.\ g\ (length\ vs + x) = f\ x$
shows $g = vs \odot f$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *prepend-in-P1*:
assumes $f \in \mathcal{P}$
shows $vs \odot f \in \mathcal{P}$
⟨*proof*⟩

lemma *prepend-in-R1*:
assumes $f \in \mathcal{R}$
shows $vs \odot f \in \mathcal{R}$
⟨*proof*⟩

lemma *prepend-associative*: $(us\ @\ vs) \odot f = us \odot vs \odot f$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

abbreviation *constant-divergent* :: *partial1* (‹$\uparrow^\infty$›) **where**
$\uparrow^\infty \equiv \lambda$-. *None*

abbreviation *constant-zero* :: *partial1* (‹0^∞›) **where**
$0^\infty \equiv \lambda$-. *Some 0*

lemma *almost0-in-R1*: $vs \odot 0^\infty \in \mathcal{R}$
⟨*proof*⟩

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* ($\langle U_0 \rangle$) **where**
$U_0 \equiv \{vs \odot 0^\infty \mid vs.\ vs \in UNIV\}$

The class U_0 contains exactly the functions in the numbering *prenum*.

lemma *U0-altdef*: $U_0 = \{prenum\ e \mid e.\ e \in UNIV\}$ (**is** $U_0 = ?W$)
⟨*proof*⟩

lemma *U0-in-NUM*: $U_0 \in NUM$
⟨*proof*⟩

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*⟩

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* ⇒ (*partial1 set*) ⇒ *partial1* ⇒ *bool* **where**
$environment\ \psi\ U\ s \equiv \psi \in \mathcal{P}^2 \wedge U \subseteq \mathcal{R} \wedge s \in \mathcal{P} \wedge (\forall f{\in}U.\ \forall n.\ s\ (f \triangleright n) \downarrow)$

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* ⇒ *bool*) ⇒ *bool* (**binder** $\langle\forall^\infty\rangle$ *10*) **where**
$\forall^\infty n.\ P\ n \equiv \exists n_0.\ \forall n{\geq}n_0.\ P\ n$

definition *learn-lim* :: *partial2* ⇒ (*partial1 set*) ⇒ *partial1* ⇒ *bool* **where**
$learn\text{-}lim\ \psi\ U\ s \equiv$
$environment\ \psi\ U\ s \wedge$
$(\forall f{\in}U.\ \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i))$

lemma *learn-limE*:
assumes $learn\text{-}lim\ \psi\ U\ s$
shows $environment\ \psi\ U\ s$
and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i)$
⟨*proof*⟩

lemma *learn-limI*:
assumes $environment\ \psi\ U\ s$
and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \triangleright n) \downarrow= i)$
shows $learn\text{-}lim\ \psi\ U\ s$
⟨*proof*⟩

definition *LIM-wrt* :: *partial2* $\Rightarrow$ *partial1 set set* **where**
LIM-wrt $\psi \equiv \{U.\ \exists s.\ \textit{learn-lim}\ \psi\ U\ s\}$

definition *Lim* :: *partial1 set set* (‹*LIM*›) **where**
$LIM \equiv \{U.\ \exists \psi\ s.\ \textit{learn-lim}\ \psi\ U\ s\}$

LIM is closed under the the subset relation.

lemma *learn-lim-closed-subseteq*:
assumes $\textit{learn-lim}\ \psi\ U\ s$ **and** $V \subseteq U$
shows $\textit{learn-lim}\ \psi\ V\ s$
⟨*proof*⟩

corollary *LIM-closed-subseteq*:
assumes $U \in LIM$ **and** $V \subseteq U$
shows $V \in LIM$
⟨*proof*⟩

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 $\neg\ \textit{learn-lim}\ \psi\ U\ s$
⟨*proof*⟩

lemma *always-hyp-change-not-Lim*:
assumes $\bigwedge x.\ s\ (f \triangleright (Suc\ x)) \neq s\ (f \triangleright x)$
shows $\neg\ \textit{learn-lim}\ \psi\ \{f\}\ s$
⟨*proof*⟩

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 $\neg\ \textit{learn-lim}\ \psi\ U\ s$
⟨*proof*⟩

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 $\neg\ \textit{learn-lim}\ \psi\ U\ s$
⟨*proof*⟩

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 $\textit{learn-lim}\ \psi\ U\ s$
and $\bigwedge f\ n.\ f \in U \Longrightarrow extra\ U\ f\ n\ (\psi\ (the\ (s\ (f \triangleright n))))$
shows $\exists t.\ \textit{learn-lim}\ \chi\ U\ t \wedge (\forall f \in U.\ \forall n.\ extra\ U\ f\ n\ (\chi\ (the\ (t\ (f \triangleright n)))))$
⟨*proof*⟩

lemma *learn-lim-wrt-goedel*:
  assumes *goedel-numbering* χ **and** *learn-lim* ψ *U s*
  shows $\exists$ *t*. *learn-lim* χ *U t*
  ⟨*proof*⟩

lemma *LIM-wrt-phi-eq-Lim*: *LIM-wrt* φ = *LIM*
  ⟨*proof*⟩

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 ψ *U s* ≡
    environment ψ *U s* ∧
    $(\forall f \in U.\ \forall^\infty n.\ \psi\ (the\ (s\ (f \triangleright n))) = f)$

lemma *learn-bcE*:
  assumes *learn-bc* ψ *U s*
  shows *environment* ψ *U s*
   and $\bigwedge f.\ f \in U \Longrightarrow \forall^\infty n.\ \psi\ (the\ (s\ (f \triangleright n))) = f$
  ⟨*proof*⟩

lemma *learn-bcI*:
  assumes *environment* ψ *U s*
   and $\bigwedge f.\ f \in U \Longrightarrow \forall^\infty n.\ \psi\ (the\ (s\ (f \triangleright n))) = f$
  shows *learn-bc* ψ *U s*
  ⟨*proof*⟩

definition *BC-wrt* :: *partial2* ⇒ *partial1 set set* **where**
  BC-wrt ψ ≡ $\{U.\ \exists s.\ learn\text{-}bc\ \psi\ U\ s\}$

definition *BC* :: *partial1 set set* **where**
  BC ≡ $\{U.\ \exists \psi\ s.\ learn\text{-}bc\ \psi\ U\ s\}$

BC is a superset of LIM and closed under the subset relation.

lemma *learn-lim-imp-BC*: *learn-lim* ψ *U s* ⟹ *learn-bc* ψ *U s*
  ⟨*proof*⟩

lemma *Lim-subseteq-BC*: *LIM* ⊆ *BC*
  ⟨*proof*⟩

lemma *learn-bc-closed-subseteq*:
  assumes *learn-bc* ψ *U s* **and** *V* ⊆ *U*
  shows *learn-bc* ψ *V s*
  ⟨*proof*⟩

corollary *BC-closed-subseteq*:
  assumes *U* ∈ *BC* **and** *V* ⊆ *U*
  shows *V* ∈ *BC*
  ⟨*proof*⟩

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** $\forall n. \exists m>n. \psi\ (the\ (s\ (f \rhd m))) \neq f$
  shows $\neg$ *learn-bc* $\psi\ U\ s$
⟨*proof*⟩

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*) ⇒ *partial1* ⇒ *nat* ⇒ *partial1* ⇒ *bool*
  assumes *goedel-numbering* χ
    and *learn-bc* $\psi\ U\ s$
    and $\bigwedge f\ n.\ f \in U \Longrightarrow extra\ U\ f\ n\ (\psi\ (the\ (s\ (f \rhd n))))$
  shows $\exists t.$ *learn-bc* $\chi\ U\ t \wedge (\forall f{\in}U. \forall n.\ extra\ U\ f\ n\ (\chi\ (the\ (t\ (f \rhd n)))))$
⟨*proof*⟩

corollary *learn-bc-wrt-goedel*:
  assumes *goedel-numbering* χ **and** *learn-bc* $\psi\ U\ s$
  shows $\exists t.$ *learn-bc* $\chi\ U\ t$
  ⟨*proof*⟩

corollary *BC-wrt-phi-eq-BC*: *BC-wrt* φ = *BC*
  ⟨*proof*⟩

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 $\psi\ U\ s \equiv$
    learn-lim $\psi\ U\ s \wedge$
    $(\forall f{\in}U. \forall n. \forall k{\leq}n.\ \psi\ (the\ (s\ (f \rhd n)))\ k = f\ k)$

definition *CONS-wrt* :: *partial2* ⇒ *partial1 set set* **where**
  CONS-wrt $\psi \equiv \{U. \exists s.$ *learn-cons* $\psi\ U\ s\}$

definition *CONS* :: *partial1 set set* **where**
  CONS $\equiv \{U. \exists \psi\ s.$ *learn-cons* $\psi\ U\ s\}$

lemma *CONS-subseteq-Lim*: *CONS* ⊆ *LIM*
  ⟨*proof*⟩

lemma *learn-consI*:
  assumes *environment* $\psi\ U\ s$
    and $\bigwedge f.\ f \in U \Longrightarrow \exists i.\ \psi\ i = f \wedge (\forall^\infty n.\ s\ (f \rhd n) \downarrow= i)$
    and $\bigwedge f\ n.\ f \in U \Longrightarrow \forall k{\leq}n.\ \psi\ (the\ (s\ (f \rhd n)))\ k = f\ k$
  shows *learn-cons* $\psi\ U\ s$
  ⟨*proof*⟩

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*⟩

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 $\bigwedge f\ n.\ f \in U \Longrightarrow \forall\, k \le n.\ \psi\ (the\ (s\ (f \triangleright n)))\ k = f\ k$
⟨*proof*⟩

lemma *learn-cons-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-cons* ψ *U s*
shows $\exists\, t.$ *learn-cons* χ *U t*
⟨*proof*⟩

lemma *CONS-wrt-phi-eq-CONS*: *CONS-wrt* φ = *CONS*
⟨*proof*⟩

lemma *learn-cons-closed-subseteq*:
assumes *learn-cons* ψ *U s* **and** $V \subseteq U$
shows *learn-cons* ψ *V s*
⟨*proof*⟩

lemma *CONS-closed-subseteq*:
assumes $U \in$ *CONS* **and** $V \subseteq U$
shows $V \in$ *CONS*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\psi \equiv \{U.\ \exists\, s.\ learn\text{-}total\ \psi\ U\ s\}$

definition *TOTAL* :: *partial1 set set* **where**
TOTAL $\equiv \{U.\ \exists\, \psi\ s.\ learn\text{-}total\ \psi\ U\ s\}$

lemma *TOTAL-subseteq-LIM*: $TOTAL \subseteq LIM$
⟨*proof*⟩

lemma *learn-totalI*:
assumes *environment* ψ U s
and $\bigwedge f.\ f \in U \Longrightarrow \exists\, i.\ \psi\ i = f \wedge (\forall^{\infty} n.\ s\ (f \rhd n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \psi\ (the\ (s\ (f \rhd n))) \in \mathcal{R}$
shows *learn-total* ψ U s
⟨*proof*⟩

lemma *learn-totalE*:
assumes *learn-total* ψ U s
shows *environment* ψ U s
and $\bigwedge f.\ f \in U \Longrightarrow \exists\, i\ n_0.\ \psi\ i = f \wedge (\forall\, n{\geq}n_0.\ s\ (f \rhd n) \downarrow= i)$
and $\bigwedge f\ n.\ f \in U \Longrightarrow \psi\ (the\ (s\ (f \rhd n))) \in \mathcal{R}$
⟨*proof*⟩

lemma *learn-total-wrt-goedel*:
assumes *goedel-numbering* χ **and** *learn-total* ψ U s
shows $\exists\, t.$ *learn-total* χ U t
⟨*proof*⟩

lemma *TOTAL-wrt-phi-eq-TOTAL*: *TOTAL-wrt* $\varphi = TOTAL$
⟨*proof*⟩

lemma *learn-total-closed-subseteq*:
assumes *learn-total* ψ U s **and** $V \subseteq U$
shows *learn-total* ψ V s
⟨*proof*⟩

lemma *TOTAL-closed-subseteq*:
assumes $U \in TOTAL$ **and** $V \subseteq U$
shows $V \in TOTAL$
⟨*proof*⟩

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* ⇒ (*partial1 set*) ⇒ *partial1* ⇒ *bool* **where**
learn-cp ψ U $s \equiv$
learn-lim ψ U s $\wedge$
$(\forall f{\in}U.\ \forall\, n.\ \psi\ (the\ (s\ (f \rhd n))) \in U)$

definition *CP-wrt* :: *partial2* ⇒ *partial1 set set* **where**
CP-wrt $\psi \equiv \{U.\ \exists\, s.$ *learn-cp* ψ U $s\}$

definition *CP* :: *partial1 set set* **where**
$CP \equiv \{U.\ \exists\, \psi\ s.$ *learn-cp* ψ 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
⟨*proof*⟩

corollary *CP-wrt-phi*: $CP = CP\text{-}wrt\ \varphi$
⟨*proof*⟩

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
⟨*proof*⟩

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 $\bigwedge f\ n.\ f \in U \Longrightarrow \psi\ (the\ (s\ (f \triangleright n))) \in U$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *CP-subseteq-TOTAL*: $CP \subseteq TOTAL$
⟨*proof*⟩

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 $s \equiv$
environment ψ U $s \wedge$
$(\forall f \in U.\ \exists i\ n_0.\ \psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \triangleright n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \triangleright n) \downarrow= Suc\ i))$

definition *FIN-wrt* :: *partial2* ⇒ *partial1 set set* **where**
FIN-wrt $\psi \equiv \{U.\ \exists s.\ learn\text{-}fin\ \psi\ U\ s\}$

definition *FIN* :: *partial1 set set* **where**
$FIN \equiv \{U.\ \exists \psi\ s.\ learn\text{-}fin\ \psi\ U\ s\}$

lemma *learn-finI*:
assumes *environment* ψ U s
and $\bigwedge f.\ f \in U \Longrightarrow$
$\exists i\ n_0.\ \psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \triangleright n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \triangleright n) \downarrow= Suc\ i)$
shows *learn-fin* ψ U s
⟨*proof*⟩

lemma *learn-finE*:
assumes *learn-fin* ψ U s
shows *environment* ψ U s
and $\bigwedge f.\ f \in U \Longrightarrow$
$\exists i\ n_0.\ \psi\ i = f \wedge (\forall n{<}n_0.\ s\ (f \triangleright n) \downarrow= 0) \wedge (\forall n{\geq}n_0.\ s\ (f \triangleright n) \downarrow= Suc\ i)$
⟨*proof*⟩

lemma *learn-fin-closed-subseteq*:
  assumes *learn-fin* ψ *U s* **and** $V \subseteq U$
  shows *learn-fin* ψ *V s*
  ⟨*proof*⟩

lemma *learn-fin-wrt-goedel*:
  assumes *goedel-numbering* χ **and** *learn-fin* ψ *U s*
  shows $\exists$ *t. learn-fin* χ *U t*
⟨*proof*⟩

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 $U \neq \{\}$ **and** $U \in$ *FIN-wrt* ψ
  shows $U \in$ *CP-wrt* ψ
⟨*proof*⟩

lemma *FIN-wrt-impl-CP*:
  assumes $U \in$ *FIN-wrt* ψ
  shows $U \in$ *CP-wrt* ψ
⟨*proof*⟩

corollary *FIN-subseteq-CP*: *FIN* $\subseteq$ *CP*
⟨*proof*⟩

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* $\downarrow$
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *r-findr [simp]*: *eval r-findr [e] = findr e*
⟨*proof*⟩

lemma *U0-in-CP*: $U_0 \in CP$
⟨*proof*⟩

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*⟩

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*: $U_0 \notin FIN$
⟨*proof*⟩

theorem *FIN-subset-CP*: $FIN \subset CP$
  ⟨*proof*⟩

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**
  $V_0 = \{f.\ f \in \mathcal{R} \wedge \varphi\ (the\ (f\ 0)) = f\}$

lemma *V0-altdef*: $V_0 = \{[i] \odot f \mid i\ f.\ f \in \mathcal{R} \wedge \varphi\ i = [i] \odot f\}$
  (**is** $V_0 = ?W$)
⟨*proof*⟩

lemma *V0-in-FIN*: $V_0 \in FIN$
⟨*proof*⟩

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 $f \in \mathcal{R}$
shows $\exists\, i.\ \varphi\ i = [i] \odot f$
⟨*proof*⟩

lemma *V0-in-FIN-minus-NUM*: $V_0 \in FIN - NUM$
⟨*proof*⟩

corollary *FIN-not-subseteq-NUM*: $\neg\ FIN \subseteq NUM$
⟨*proof*⟩

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*: $\neg\ CP \subseteq NUM$
⟨*proof*⟩

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*⟩

2.6 NUM is a proper subset of TOTAL

A NUM class U is embedded in a total numbering ψ. The strategy S with $S(f^n) = \min\{i \mid \forall k \le n : \psi_i(k) = f(k)\}$ 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 $NUM \subseteq TOTAL$.

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 $x \le |e|$. 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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*] ↑
⟨*proof*⟩

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*⟩

lemma *consistent-in-P2*:
assumes $\psi \in \mathcal{P}^2$
shows *consistent* $\psi \in \mathcal{P}^2$
⟨*proof*⟩

lemma *consistent-for-R2*:
assumes $\psi \in \mathcal{R}^2$
shows *consistent* ψ *i e* =
(*if* ∀ *j*<*e-length e*. ψ *i j* ↓= *e-nth e j then Some 0 else Some 1*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *consistent-in-R2*:
assumes $\psi \in \mathcal{R}^2$
shows *consistent* $\psi \in \mathcal{R}^2$

⟨*proof*⟩

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*⟩

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*⟩

corollary *NUM-subseteq-TOTAL*: *NUM* ⊆ *TOTAL*
⟨*proof*⟩

The class V_0 is in *TOTAL* − *NUM*.

theorem *NUM-subset-TOTAL*: *NUM* ⊂ *TOTAL*
⟨*proof*⟩

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**
$U_{LIM-CONS}$ ≡ {*vs* @ [*j*] ⊙ *p*| *vs j p*. *j* ≥ *2* ∧ *p* ∈ $\mathcal{R}_{01}$ ∧ φ *j* = *vs* @ [*j*] ⊙ *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*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *rmge2-in-R1*: *rmge2* ∈ $\mathcal{R}$
⟨*proof*⟩

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*⟩

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*⟩

lemma *U-LIMCONS-prefix-complete*:
assumes *length vs* > *0*
shows ∃ *f*∈$U_{LIM-CONS}$. *prefix f* (*length vs* − *1*) = *vs*
⟨*proof*⟩

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*⟩

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* ∈ $\mathcal{R}$
begin

A *recf* computing the strategy:

definition *r-s* :: *recf* **where**
  r-s ≡ *SOME r-s. recfn 1 r-s* ∧ *total r-s* ∧ *s* = (*λx. 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* = (*λx. eval r-s* [*x*])
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

definition *r-prefixes* ≡ *r-swap* (*Pr 1 r-singleton-encode r-prefixes-aux*)

lemma *r-prefixes-recfn*: *recfn 2 r-prefixes*
  ⟨*proof*⟩

lemma *r-prefixes*: *eval r-prefixes* $[i, n] \downarrow=$ *list-encode* (*prefixes i n*)
⟨*proof*⟩

lemma *prefixes-neq-nil*: *length* (*prefixes i x*) > 0
⟨*proof*⟩

The actual numbering can then be defined via *prefixes*.

definition *psi* :: *partial2* (‹ψ›) **where**
$\psi\ i\ x \equiv$ *Some* (*last* (*prefixes i x*))

lemma *psi-in-R2*: $\psi \in \mathcal{R}^2$
⟨*proof*⟩

lemma *psi-0-or-1*:
assumes $n > 0$
shows $\psi\ i\ n \downarrow= 0 \vee \psi\ i\ n \downarrow= 1$
⟨*proof*⟩

The function *prefixes* does indeed provide the prefixes for ψ.

lemma *psi-init*: $(\psi\ i) \triangleright x =$ *list-encode* (*prefixes i x*)
⟨*proof*⟩

One of the functions ψ_i is in $U_{LIM-CONS}$.

lemma *ex-psi-in-U*: $\exists j.\ \psi\ j \in U_{LIM-CONS}$
⟨*proof*⟩

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*: $\neg$ *learn-cons* $\varphi\ U_{LIM-CONS}\ s$
⟨*proof*⟩

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*⟩

The main result of this section:

theorem *CONS-subset-Lim*: $CONS \subset LIM$
⟨*proof*⟩

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 φ *i x* ↓= *v*
  shows ∃ *t*.
    (∀ *t'*≥*t*. *eval r-result1* [*t'*, *i*, *x*] ↓= *Suc v*) ∧
    (∀ *t'*<*t*. *eval r-result1* [*t'*, *i*, *x*] ↓= *0*)
  ⟨*proof*⟩

lemma *r-result1-bivalent'*:
  assumes *eval r-phi* [*i*, *x*] ↓= *v*
  shows *eval r-result1* [*t*, *i*, *x*] ↓= *Suc v* ∨ *eval r-result1* [*t*, *i*, *x*] ↓= *0*
  ⟨*proof*⟩

lemma *r-result1-bivalent-phi*:
  assumes φ *i x* ↓= *v*
  shows *eval r-result1* [*t*, *i*, *x*] ↓= *Suc v* ∨ *eval r-result1* [*t*, *i*, *x*] ↓= *0*
  ⟨*proof*⟩

lemma *r-result1-diverg-phi*:
  assumes φ *i x* ↑
  shows *eval r-result1* [*t*, *i*, *x*] ↓= *0*
  ⟨*proof*⟩

lemma *r-result1-some-phi*:
  assumes *eval r-result1* [*t*, *i*, *x*] ↓= *Suc v*
  shows φ *i x* ↓= *v*
  ⟨*proof*⟩

lemma *r-result1-saturating'*:
  assumes *eval r-result1* [*t*, *i*, *x*] ↓= *Suc v*
  shows *eval r-result1* [*t* + *d*, *i*, *x*] ↓= *Suc v*
  ⟨*proof*⟩

lemma *r-result1-saturating-the*:
  assumes *the* (*eval r-result1* [*t*, *i*, *x*]) > *0* **and** *t'* ≥ *t*
  shows *the* (*eval r-result1* [*t'*, *i*, *x*]) > *0*

⟨*proof*⟩

lemma *Greatest-bounded-Suc*:
 fixes *P* :: *nat* ⇒ *nat*
 shows (*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*) =
 (*if* ∃ *j*<*Suc n*. *P j* > *0 then Suc* (*GREATEST j*. *j* < *Suc n* ∧ *P j* > *0*) *else 0*)
 (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *r-delay-aux-total*: *total r-delay-aux*
 ⟨*proof*⟩

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*⟩

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*
 ⟨*proof*⟩

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)
 ⟨*proof*⟩

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* ∈ $\mathcal{R}^2$
⟨*proof*⟩

lemma *delay-le-length*: *the (delay i x) ≤ e-length x*
⟨*proof*⟩

lemma *e-take-delay-init*:
assumes *f* ∈ $\mathcal{R}$ **and** *the (delay i (f ▷ n)) > 0*
shows *e-take (the (delay i (f ▷ n))) (f ▷ n) = f ▷ (the (delay i (f ▷ n)) − 1)*
⟨*proof*⟩

lemma *delay-gr0-converg*:
assumes *the (delay i x) > 0*
shows *φ i (e-take (the (delay i x)) x) ↓*
⟨*proof*⟩

lemma *delay-unbounded*:
fixes *n :: nat*
assumes *f* ∈ $\mathcal{R}$ **and** *∀ n. φ i (f ▷ n) ↓*
shows *∃ m. the (delay i (f ▷ m)) > n*
⟨*proof*⟩

lemma *delay-monotone*:
assumes *f* ∈ $\mathcal{R}$ **and** $n_1 \leq n_2$
shows *the (delay i (f ▷* n_1*)) ≤ the (delay i (f ▷* n_2*))*
(**is** *the (delay i ?x1) ≤ the (delay i ?x2)*)
⟨*proof*⟩

lemma *delay-unbounded-monotone*:
fixes *n :: nat*
assumes *f* ∈ $\mathcal{R}$ **and** *∀ n. φ i (f ▷ n) ↓*
shows *∃* m_0*. ∀ m≥*m_0*. the (delay i (f ▷ m)) > n*
⟨*proof*⟩

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)*
⟨*proof*⟩

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*⟩

lemma *r-totalizer-total*: *total* (*r-totalizer d*)
⟨*proof*⟩

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 $f \in \mathcal{R}$
shows *totalizer d i* ($f \triangleright n$) =
(*if the* (*delay i* ($f \triangleright n$)) = *0 then Some d*
else φ *i* ($f \triangleright$ (*the* (*delay i* ($f \triangleright n$)) − *1*)))
⟨*proof*⟩

lemma *totalizer-in-R2*: *totalizer d* $\in \mathcal{R}^2$
⟨*proof*⟩

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*⟩

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*:
$\exists\, g \in \mathcal{R}.\ \forall\, i.$
φ (*the* (*g i*)) $\in \mathcal{R}$ ∧
(∀ *U* ψ. *learn-lim* ψ *U* (φ *i*) ⟶ *learn-lim* ψ *U* (φ (*the* (*g i*))))
⟨*proof*⟩

In order for us to use the previous lemma, we need a function that performs the actual computation:

definition *r-limr* ≡
SOME g.
recfn 1 g ∧
total g ∧
(∀ *i*. φ (*the* (*eval g* [*i*])) $\in \mathcal{R}$ ∧
(∀ *U* ψ. *learn-lim* ψ *U* (φ *i*) ⟶ *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*) ⟹ *learn-lim* ψ *U* (φ (*the* (*eval r-limr* [*i*])))
⟨*proof*⟩

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*⟩

corollary *lemma-R-for-BC-simple*:
  assumes *learn-bc* ψ *U s*
  shows $\exists\, s' \in \mathcal{R}$. *learn-bc* ψ *U* s'
  ⟨*proof*⟩

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* (φ *i*)
  shows *learn-fin* ψ *U* (*totalizer 0 i*)
⟨*proof*⟩

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 $\exists\, d.\ \forall\, U.\ \forall\, i.$ *learn-total* ψ *U* (φ *i*) $\longrightarrow$ *learn-total* ψ *U* (*totalizer d i*)
⟨*proof*⟩

corollary *lemma-R-for-TOTAL-simple*:
  assumes *learn-total* ψ *U s*
  shows $\exists\, s' \in \mathcal{R}$. *learn-total* ψ *U* s'
  ⟨*proof*⟩

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 $\exists\, d$. *learn-cp* ψ *U* (*totalizer d i*)
⟨*proof*⟩

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* (‹V_{01}›) **where**
  $V_{01} = \{f.\ f \in \mathcal{R} \wedge (\varphi\ (the\ (f\ 0)) = f \vee \varphi\ (the\ (f\ 1)) = f)\}$

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 $S(f^x) \neq S(f^x 0)$. 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 $S(f^x) \neq S(f^x 1)$. 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 $h := S(f^x) = S(f^x 0) = S(f^x 1)$. But the hypothesis h cannot be consistent with both prefixes $f^x 0$ and $f^x 1$. Running the process once with $z = 0$ and once with $z = 1$ yields two functions starting with $f^x 0$ and $f^x 1$, 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
  ⟨proof⟩
```

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⟩
```

```
lemma adverse-at-01:
  adverse z i j 0 ↓= i
  adverse z i j 1 ↓= j
  ⟨proof⟩
```

Had we introduced ternary partial recursive functions, the *adverse z* functions would be among them.

lemma *adverse-in-R3*: $\exists\, r.\ recfn\ 3\ r \wedge total\ r \wedge (\lambda i\ j\ x.\ eval\ r\ [i,\ j,\ x]) = adverse\ z$
⟨*proof*⟩

lemma *adverse-in-R1*: $adverse\ z\ i\ j \in \mathcal{R}$
⟨*proof*⟩

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*: $\exists f{\in}\mathcal{R}^2.\forall\, i\ j.\ \varphi\ (the\ (f\ i\ j)) = adverse\ z\ i\ j$
⟨*proof*⟩

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*⟩

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 $adverse\ z\ i\ j\ (Suc\ x) \downarrow=$
    $(if\ s\ (e\text{-}snoc\ ((adverse\ z\ i\ j) \triangleright x)\ 0) \neq s\ ((adverse\ z\ i\ j) \triangleright x)$
     $then\ 0$
     $else\ if\ s\ (e\text{-}snoc\ ((adverse\ z\ i\ j) \triangleright x)\ 1) \neq s\ ((adverse\ z\ i\ j) \triangleright x)$
        $then\ 1\ else\ z)$
⟨*proof*⟩

The process in the proof sketch (page 86) consists of steps (a), (b), and (c). The next abbreviation is true iff. step (a) or (b) applies.

abbreviation $hyp\text{-}change\ z\ i\ j\ x \equiv$
  $s\ (e\text{-}snoc\ ((adverse\ z\ i\ j) \triangleright x)\ 0) \neq s\ ((adverse\ z\ i\ j) \triangleright x) \vee$
  $s\ (e\text{-}snoc\ ((adverse\ z\ i\ j) \triangleright x)\ 1) \neq s\ ((adverse\ z\ i\ j) \triangleright x)$

If step (c) applies, the process appends z.

lemma *adverse-Suc-not-hyp-change*:
  assumes $x > 0$ **and** $\neg\ hyp\text{-}change\ z\ i\ j\ x$
  shows $adverse\ z\ i\ j\ (Suc\ x) \downarrow= z$
  ⟨*proof*⟩

While (a) or (b) applies, the process appends a value that forces S to change its hypothesis.

lemma *while-hyp-change*:
  assumes $\forall\, x{\le}n.\ x > 0 \longrightarrow hyp\text{-}change\ z\ i\ j\ x$
  shows $\forall\, x{\le}Suc\ n.\ adverse\ z\ i\ j\ x = adverse\ z'\ i\ j\ x$
  ⟨*proof*⟩

The next result corresponds to Case 1 from the proof sketch.

lemma *always-hyp-change-no-lim*:
  assumes $\forall\, x{>}0.\ hyp\text{-}change\ z\ i\ j\ x$
  shows $\neg\ learn\text{-}lim\ \varphi\ \{adverse\ z\ i\ j\}\ s$

⟨*proof*⟩

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*⟩

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*: $\neg$ *learn-cons* φ V_{01} *s*
⟨*proof*⟩

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* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat option* **where**
  parallel i j x $\equiv$ *eval r-parallel* [*i*, *j*, *x*]

lemma *r-parallel'*: *eval r-parallel* [*i*, *j*, *x*] = *parallel i j x*
  ⟨*proof*⟩

lemma *r-parallel''*:
  shows *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* $[j, x]))$
⟨*proof*⟩

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* $(\varphi\ j\ x))$
  ⟨*proof*⟩

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*$)) = 1$
  ⟨*proof*⟩

lemma *parallel-converg-either*: $(\varphi\ i\ x \downarrow \vee \varphi\ j\ x \downarrow) = ($*parallel i j x* $\downarrow)$
  ⟨*proof*⟩

lemma *parallel-0*:
  assumes *parallel i j x* $\downarrow=$ *prod-encode* $(0, v)$
  shows $\varphi\ i\ x \downarrow= v$
  ⟨*proof*⟩

lemma *parallel-1*:
  assumes *parallel i j x* $\downarrow=$ *prod-encode* $(1, v)$
  shows $\varphi\ j\ x \downarrow= v$
  ⟨*proof*⟩

lemma *parallel-converg-V01*:
  assumes $f \in V_{01}$
  shows *parallel* $($*the* $(f\ 0))$ $($*the* $(f\ 1))$ $x \downarrow$
⟨*proof*⟩

The amalgamation of two Gödel numbers can then be described in terms of *parallel*.

definition *amalgamation* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *partial1* **where**
  amalgamation i j x $\equiv$
    if parallel i j x $\uparrow$ *then None else Some* $($*pdec2* $($*the* $($*parallel i j x*$)))$

lemma *amalgamation-diverg*: *amalgamation i j x* $\uparrow \longleftrightarrow \varphi\ i\ x \uparrow \wedge \varphi\ j\ x \uparrow$
  ⟨*proof*⟩

lemma *amalgamation-total*:
  assumes *total1* $(\varphi\ i) \vee$ *total1* $(\varphi\ j)$
  shows *total1* $($*amalgamation i j*$)$
  ⟨*proof*⟩

lemma *amalgamation-V01-total*:
  assumes $f \in V_{01}$
  shows *total1* (*amalgamation* (*the* (*f 0*)) (*the* (*f 1*)))
  ⟨*proof*⟩

definition *r-amalgamation* ≡ *Cn 3 r-pdec2* [*r-parallel*]

lemma *r-amalgamation-recfn*: *recfn 3 r-amalgamation*
  ⟨*proof*⟩

lemma *r-amalgamation*: *eval r-amalgamation* [*i*, *j*, *x*] = *amalgamation i j x*
⟨*proof*⟩

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*⟩

lemma *amalgamation-in-P1*: *amalgamation i j* ∈ $\mathcal{P}$
  ⟨*proof*⟩

lemma *amalgamation-V01-R1*:
  assumes $f \in V_{01}$
  shows *amalgamation* (*the* (*f 0*)) (*the* (*f 1*)) ∈ $\mathcal{R}$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *r-amalgamate*: *eval r-amalgamate* [*i*, *j*] ↓= *amalgamate i j*
⟨*proof*⟩

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 $\forall x<$*e-length e*. φ *i x* $\downarrow$
⟨*proof*⟩

lemma *inconsist-bounded*:
assumes *inconsist i e* $\downarrow$
shows *the* (*inconsist i e*) $\leq$ *e-length e*
⟨*proof*⟩

lemma *inconsist-consistent*:
assumes *inconsist i e* $\downarrow$
shows *inconsist i e* $\downarrow=$ *e-length e* $\longleftrightarrow$ ($\forall x<$*e-length e*. φ *i x* $\downarrow=$ *e-nth e x*)
⟨*proof*⟩

lemma *inconsist-converg-eq*:
assumes *inconsist i e* $\downarrow=$ *e-length e*
shows $\forall x<$*e-length e*. φ *i x* $\downarrow=$ *e-nth e x*
⟨*proof*⟩

lemma *inconsist-converg-less*:
assumes *inconsist i e* $\downarrow$ **and** *the* (*inconsist i e*) $<$ *e-length e*
shows $\exists x<$*e-length e*. φ *i x* $\downarrow\neq$ *e-nth e x*
and *inconsist i e* $\downarrow=$ (*LEAST x*. $x <$ *e-length e* $\wedge$ φ *i x* $\downarrow\neq$ *e-nth e x*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *inconsist-init-converg-less*:
assumes $f \in \mathcal{R}$
and φ $i \in \mathcal{R}$
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*⟩

definition *r-inconsist* $\equiv$
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*
⟨*proof*⟩

lemma *r-inconsist*: *eval r-inconsist* [*i*, *e*] = *inconsist i e*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *inconsist-for-V01*:
assumes *f* ∈ V_{01} **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*⟩

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*
⟨*proof*⟩

lemma *r-auxhyp*: φ (*the* (*eval r-auxhyp* [*e*])) = *prenum e*
⟨*proof*⟩

definition *auxhyp* :: *partial1* **where**
auxhyp e ≡ *eval r-auxhyp* [*e*]

lemma *auxhyp-prenum*: φ (*the* (*auxhyp e*)) = *prenum e*
⟨*proof*⟩

lemma *auxhyp-in-R1*: *auxhyp* ∈ $\mathcal{R}$
⟨*proof*⟩

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*

⟨*proof*⟩

definition *sv01* :: *partial1* (‹s_{01}›)**where**
sv01 e ≡ *eval r-sv01* [*e*]

lemma *sv01-in-P1*: $s_{01} \in \mathcal{P}$
⟨*proof*⟩

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 $f \in V_{01}$
shows s_{01} ($f \triangleright 0$) = *auxhyp* ($f \triangleright 0$)
and $n \neq 0 \Longrightarrow$
inconsist (*amalg01 f*) ($f \triangleright n$) ↓= *Suc n* ⟹
s_{01} ($f \triangleright n$) ↓= *amalg01 f*
and $n \neq 0 \Longrightarrow$
the (*inconsist* (*amalg01 f*) ($f \triangleright n$)) < *Suc n* ⟹
pdec1 (*the* (*parallel* (*the* (*f 0*)) (*the* (*f 1*)) (*the* (*inconsist* (*amalg01 f*) ($f \triangleright n$))))) = 0 ⟹
s_{01} ($f \triangleright n$) = *f 1*
and $n \neq 0 \Longrightarrow$
the (*inconsist* (*amalg01 f*) ($f \triangleright n$)) < *Suc n* ⟹
pdec1 (*the* (*parallel* (*the* (*f 0*)) (*the* (*f 1*)) (*the* (*inconsist* (*amalg01 f*) ($f \triangleright n$))))) ≠ 0 ⟹
s_{01} ($f \triangleright n$) = *f 0*
⟨*proof*⟩

Part of the correctness of s_{01} is convergence on prefixes of functions in V_{01}.

lemma *sv01-converg-V01*:
assumes $f \in V_{01}$
shows s_{01} ($f \triangleright n$) ↓
⟨*proof*⟩

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 ∀ *x*≤*n*. φ (*the* (s_{01} ($f \triangleright n$))) *x* = *f x*
⟨*proof*⟩

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 ∃ *i*. $\forall^\infty$*n*. s_{01} ($f \triangleright n$) ↓= *i*
⟨*proof*⟩

lemma *V01-learn-cons*: *learn-cons* φ V_{01} s_{01}
⟨*proof*⟩

corollary *V01-in-CONS*: $V_{01} \in$ *CONS*
⟨*proof*⟩

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*: $\exists\, U.\ U \in \mathit{CONS} \wedge (\neg\ (\exists\, s.\ s \in \mathcal{R} \wedge \mathit{learn\text{-}cons}\ \varphi\ U\ s))$
⟨*proof*⟩

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 $V \notin$ 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 $f(0), \dots, f(n)$. 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*
⟨*proof*⟩

lemma *r-sigma*: *eval r-sigma* [*i*, *x*] = *φ* (*the* (*eval r-limr* [*i*])) *x*
⟨*proof*⟩

lemma *r-sigma-total*: *total r-sigma*
⟨*proof*⟩

abbreviation *sigma* :: *partial2* (‹*σ*›) **where**
σ i x ≡ *eval r-sigma* [*i*, *x*]

lemma *sigma*: *σ i* = *φ* (*the* (*eval r-limr* [*i*]))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
 ⟨*proof*⟩

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*⟩

lemma *r-next-total*: *total r-next*
⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *r-state-at-0*: *eval r-state* [*0*, *i*] ↓= *prod-encode* (*list-encode* [*i*, *0*], *1*)
⟨*proof*⟩

lemma *r-state-total*: *total r-state*

⟨proof⟩

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*
⟨proof⟩

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⟩

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*
⟨proof⟩

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*
⟨proof⟩

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⟩

Next are some properties of *block* and *gap*.

lemma *gap-in-block*: *gap i t* < *e-length* (*block i t*)
⟨proof⟩

lemma *length-block*: *e-length* (*block i t*) = *Suc* (*Suc t*)
⟨proof⟩

lemma *gap-gr0*: *gap i t* > *0*
⟨proof⟩

lemma *hd-block*: *e-hd* (*block i t*) = *i*
⟨proof⟩

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*⟩

lemma *gap-le-Suc*: *gap i t* ≤ *gap i* (*Suc t*)
⟨*proof*⟩

lemma *gap-monotone*:
assumes $t_1 \leq t_2$
shows *gap i* t_1 ≤ *gap i* t_2
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\neg$ *determined i x* $\Longrightarrow$ *dettime i x* $\uparrow$
⟨*proof*⟩

lemma *r-dettimeI*:
assumes $x <$ *e-length* (*block i t*) $\wedge$ $x \neq$ *gap i t*
and $\bigwedge T.\ x <$ *e-length* (*block i T*) $\wedge$ $x \neq$ *gap i T* $\Longrightarrow t \leq T$
shows *dettime i x* $\downarrow=$ *t*
⟨*proof*⟩

lemma *r-dettime-0*: *dettime i 0* $\downarrow=$ *0*
⟨*proof*⟩

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* $\equiv$ *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*
⟨*proof*⟩

abbreviation *tau* :: *partial2* (‹τ›) **where**
τ *i x* $\equiv$ *eval r-tau* [*i*, *x*]

lemma *tau-in-P2*: $\tau \in \mathcal{P}^2$
⟨*proof*⟩

lemma *tau-diverg*:
assumes $\neg$ *determined i x*
shows τ *i x* $\uparrow$
⟨*proof*⟩

lemma *tau-converg*:
assumes *determined i x*
shows τ *i x* $\downarrow=$ *e-nth* (*block i* (*the* (*dettime i x*))) *x*
⟨*proof*⟩

lemma *tau-converg'*:
assumes *dettime i x* $\downarrow=$ *t*
shows τ *i x* $\downarrow=$ *e-nth* (*block i t*) *x*
⟨*proof*⟩

lemma *tau-at-0*: τ *i 0* $\downarrow=$ *i*
⟨*proof*⟩

lemma *state-unchanged*:
assumes *gap i t* $-$ *1* $\leq y$ **and** $y \leq t$
shows *gap i t* $=$ *gap i y*
⟨*proof*⟩

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** $j \neq$ *gap i t*
shows τ *i j* $\downarrow=$ *e-nth* (*block i t*) *j*
⟨*proof*⟩

lemma *tau-eq-state'*:
assumes $j < t + 2$ **and** $j \neq gap\ i\ t$
shows $\tau\ i\ j \downarrow= e\text{-}nth\ (block\ i\ t)\ j$
⟨*proof*⟩

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 $case\text{-}two\ i \equiv (\exists\, t.\ \forall\, T.\ gap\ i\ T \leq gap\ i\ t)$

abbreviation $case\text{-}one\ i \equiv \neg\ case\text{-}two\ i$

Another characterization of Case 2 is that from some iteration on only *change-on-neither* holds.

lemma *case-two-iff-forever-neither*:
$case\text{-}two\ i \longleftrightarrow (\exists\, t.\ \forall\, T \geq t.\ change\text{-}on\text{-}neither\ (block\ i\ T)\ (gap\ i\ T))$
⟨*proof*⟩

In Case 1, τ_i is total.

lemma *case-one-tau-total*:
assumes *case-one i*
shows $\tau\ i\ x \downarrow$
⟨*proof*⟩

In Case 2, τ_i is undefined only at the gap that never gets filled.

lemma *case-two-tau-not-quite-total*:
assumes $\forall\, T.\ gap\ i\ T \leq gap\ i\ t$
shows $\tau\ i\ (gap\ i\ t) \uparrow$
and $x \neq gap\ i\ t \Longrightarrow \tau\ i\ x \downarrow$
⟨*proof*⟩

lemma *case-two-tau-almost-total*:
assumes $\exists\, t.\ \forall\, T.\ gap\ i\ T \leq gap\ i\ t$ (**is** $\exists\, t.\ ?P\ t$)
shows $\tau\ i\ (gap\ i\ (Least\ ?P)) \uparrow$
and $x \neq gap\ i\ (Least\ ?P) \Longrightarrow \tau\ i\ x \downarrow$
⟨*proof*⟩

Some more properties of τ.

lemma *init-tau-gap*: $(\tau\ i) \triangleright (gap\ i\ t - 1) = e\text{-}take\ (gap\ i\ t)\ (block\ i\ t)$
⟨*proof*⟩

lemma *change-on-0-init-tau*:
assumes $change\text{-}on\text{-}0\ (block\ i\ t)\ (gap\ i\ t)$
shows $(\tau\ i) \triangleright (t + 1) = block\ i\ t$
⟨*proof*⟩

lemma *change-on-0-hyp-change*:
assumes $change\text{-}on\text{-}0\ (block\ i\ t)\ (gap\ i\ t)$
shows $\sigma\ i\ ((\tau\ i) \triangleright (t + 1)) \neq \sigma\ i\ ((\tau\ i) \triangleright (gap\ i\ t - 1))$
⟨*proof*⟩

lemma *change-on-1-init-tau*:
assumes $change\text{-}on\text{-}1\ (block\ i\ t)\ (gap\ i\ t)$
shows $(\tau\ i) \triangleright (t + 1) = e\text{-}update\ (block\ i\ t)\ (gap\ i\ t)\ 1$
⟨*proof*⟩

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*))
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

lemma *filled-gap-0-init-tau*:
  assumes f_0 = (τ *i*)((*gap i t*):=*Some 0*)
  shows f_0 ▷ (*t* + *1*) = *block i t*
⟨*proof*⟩

lemma *filled-gap-1-init-tau*:
  assumes f_1 = (τ *i*)((*gap i t*):=*Some 1*)
  shows f_1 ▷ (*t* + *1*) = *e-update* (*block i t*) (*gap i t*) *1*
⟨*proof*⟩

2.9.3 The separating class

Next we define the sets V_i from the introductory proof sketch (page 95).

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* ⊆ $\mathcal{R}$
⟨*proof*⟩

lemma *case-one-imp-gap-unbounded*:
  assumes *case-one i*
  shows ∃ *t*. *gap i t* − *1* > *n*
⟨*proof*⟩

lemma *case-one-imp-not-learn-lim-V*:
  assumes *case-one i*
  shows ¬ *learn-lim* φ (*V-bclim i*) (σ *i*)
⟨*proof*⟩

lemma *case-two-imp-not-learn-lim-V*:
  assumes *case-two i*
  shows ¬ *learn-lim* φ (*V-bclim i*) (σ *i*)
⟨*proof*⟩

corollary *not-learn-lim-V*: ¬ *learn-lim* φ (*V-bclim i*) (σ *i*)
  ⟨*proof*⟩

Next we define the separating class.

definition *V-BCLIM* :: *partial1 set* (‹V_{BC-LIM}›) **where**
  $V_{BC-LIM} \equiv \bigcup i.$ *V-bclim i*

lemma *V-BCLIM-R1*: $V_{BC-LIM} \subseteq \mathcal{R}$
⟨*proof*⟩

lemma *V-BCLIM-not-in-Lim*: $V_{BC-LIM} \notin LIM$
⟨*proof*⟩

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**
$\psi^\tau\ b\ x \equiv (\textit{if}\ x < \textit{e-length}\ b\ \textit{then Some}\ (\textit{e-nth}\ b\ x)\ \textit{else}\ \tau\ (\textit{e-hd}\ b)\ x)$

lemma *psitau-in-P2*: $\psi^\tau \in \mathcal{P}^2$
⟨*proof*⟩

lemma *psitau-init*:
$\psi^\tau\ (f \triangleright n)\ x = (\textit{if}\ x < \textit{Suc}\ n\ \textit{then Some}\ (\textit{the}\ (f\ x))\ \textit{else}\ \tau\ (\textit{the}\ (f\ 0))\ x)$
⟨*proof*⟩

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*⟩

Finally, the main result of this section:

theorem *Lim-subset-BC*: $LIM \subset BC$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-consist-phi*:
assumes $\forall k<e\text{-}length\ e.\ \varphi\ i\ k\ \downarrow$
shows *eval r-consist-phi* $[i, e] \downarrow=$
$(if\ \forall k<e\text{-}length\ e.\ \varphi\ i\ k \downarrow= e\text{-}nth\ e\ k\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *r-consist-phi-init*:
assumes $f \in \mathcal{R}$ **and** $\varphi\ i \in \mathcal{R}$
shows *eval r-consist-phi* $[i, f \triangleright n] \downarrow= (if\ \forall k\leq n.\ \varphi\ i\ k = f\ k\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *TOTAL-subseteq-CONS*: $TOTAL \subseteq CONS$
⟨*proof*⟩

2.10.2 The separating class

Definition of the class

The class that will be shown to be in $CONS - TOTAL$ is the union of the following two classes.

definition *V-constotal-1* :: *partial1 set* **where**
$V\text{-}constotal\text{-}1 \equiv \{f.\ \exists j\ p.\ f = [j] \odot p \land j \geq 2 \land p \in \mathcal{R}_{01} \land \varphi\ j = f\}$

definition *V-constotal-2* :: *partial1 set* **where**
V-constotal-2 ≡
$\{f.\ \exists j\ a\ k.$
$f = j \# a @ [k] \odot 0^\infty \land$
$j \geq 2 \land$
$(\forall i<length\ a.\ a\ !\ i \leq 1) \land$
$k \geq 2 \land$
$\varphi\ j = j \# a \odot \uparrow^\infty \land$
$\varphi\ k = f\}$

definition *V-constotal* :: *partial1 set* **where**
V-constotal ≡ *V-constotal-1* ∪ *V-constotal-2*

lemma *V-constotal-2I*:
assumes $f = j \# a @ [k] \odot 0^\infty$
and $j \geq 2$
and $\forall i<length\ a.\ a\ !\ i \leq 1$
and $k \geq 2$
and $\varphi\ j = j \# a \odot \uparrow^\infty$
and $\varphi\ k = f$
shows $f \in$ *V-constotal-2*
⟨*proof*⟩

lemma *V-subseteq-R1*: *V-constotal* $\subseteq \mathcal{R}$
⟨*proof*⟩

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*⟩

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**
$r\text{-}s \equiv SOME\ r\text{-}s.\ recfn\ 1\ r\text{-}s \wedge\ total\ r\text{-}s \wedge s = (\lambda x.\ eval\ r\text{-}s\ [x])$

lemma *rs-recfn* [*simp*]: *recfn 1 r-s*
and *rs-total* [*simp*]: $\bigwedge x.\ eval\ r\text{-}s\ [x] \downarrow$
and *eval-rs*: $\bigwedge x.\ s\ x = eval\ r\text{-}s\ [x]$
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\bigwedge x.\ \varphi\ i\ x \uparrow$
shows $\bigwedge y.\ enumdom\ i\ y \uparrow$
⟨*proof*⟩

lemma *enumdom-nonempty-domain*:
assumes $\varphi\ i\ x_0 \downarrow$
shows $\bigwedge y.\ enumdom\ i\ y \downarrow$
and $\bigwedge x.\ \varphi\ i\ x \downarrow \longleftrightarrow (\exists\, y.\ enumdom\ i\ y \downarrow= x)$
⟨*proof*⟩

Enumerating the empty domain yields the empty function.

lemma *search-enum-empty*:
fixes *b* :: *nat*
assumes $s\ b \downarrow= i$ **and** $\bigwedge x.\ \varphi\ i\ x \uparrow$
shows $\bigwedge x.\ search\text{-}enum\ x\ b \uparrow$
⟨*proof*⟩

Enumerating a non-empty domain yields a total function.

lemma *search-enum-nonempty*:
fixes *b y0* :: *nat*
assumes $s\ b \downarrow= i$ **and** $\varphi\ i\ y_0 \downarrow$ **and** $e = the\ (enumdom\ i\ x)$
shows $search\text{-}enum\ x\ b \downarrow= (if\ e\text{-}length\ b \leq e\ then\ 0\ else\ 1)$
⟨*proof*⟩

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 $s\ b \downarrow= i$ **and** $\varphi\ i\ y \downarrow$ **and** $y \geq e\text{-}length\ b$

shows $\exists\, x.$ *search-enum* x b $\downarrow=$ *0*
⟨*proof*⟩

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 $\downarrow=$ i
and φ i y_0 $\downarrow$
and $\neg$ $(\exists\, y.\ \varphi\ i\ y \downarrow \wedge\ y \geq$ *e-length* $b)$
shows $\neg$ $(\exists\, x.$ *search-enum* x b $\downarrow=$ *0*$)$
⟨*proof*⟩

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* $\equiv$ *Cn 1 r-enumdom* [*r-s*, *Mn 1 r-search-enum*]

lemma *r-search-recfn* [*simp*]: *recfn 1 r-search*
⟨*proof*⟩

abbreviation *search* :: *partial1* **where**
search b $\equiv$ *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*⟩

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 φ i y_0 $\downarrow$
and $\neg$ $(\exists\, y.\ \varphi\ i\ y \downarrow \wedge\ y \geq$ *e-length* $b)$
shows *search* b $\uparrow$
⟨*proof*⟩

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** φ i y $\downarrow$ **and** $y \geq$ *e-length* b
shows *search* b $\downarrow$
and φ i (*the* (*search* b)) $\downarrow$
and *the* (*search* b) $\geq$ *e-length* b
⟨*proof*⟩

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 $\downarrow=$ i **and** *search* b $\downarrow$ (**is** *?y* $\downarrow$)
shows φ i (*the ?y*) $\downarrow$
and *the ?y* $\geq$ *e-length* b

⟨proof⟩

Likewise, if the search diverges, there is no appropriate y.

lemma *search-diverg*:
  assumes $s\ b \downarrow= i$ **and** $search\ b \uparrow$
  shows $\neg (\exists y.\ \varphi\ i\ y \downarrow \wedge y \geq e\text{-}length\ b)$
⟨proof⟩

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*
  ⟨proof⟩

lemma *r-badblock*: *eval r-badblock* [*n*, *v*] ↓= *list-encode* (*replicate n 0* @ [*1* − *v*])
⟨proof⟩

lemma *r-badblock-only-01*: *e-nth* (*the* (*eval r-badblock* [*n*, *v*])) *i* ≤ *1*
  ⟨proof⟩

lemma *r-badblock-last*: *e-nth* (*the* (*eval r-badblock* [*n*, *v*])) *n* = *1* − *v*
  ⟨proof⟩

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*
  ⟨proof⟩

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* ↑
  ⟨proof⟩

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*)]))
  ⟨proof⟩

lemma *nxt-search-diverg*:

assumes *nxt b* ↑
shows *search b* ↑
⟨*proof*⟩

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*⟩

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* ↑
⟨*proof*⟩

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*⟩

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*⟩

lemma *nxt-monotone*:
assumes *nxt b* ↓= *b′*
shows *e-length b* < *e-length b′*
⟨*proof*⟩

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*
⟨*proof*⟩

abbreviation *prefixes* :: *partial2* **where**
prefixes t j ≡ *eval r-prefixes* [*t*, *j*]

lemma *prefixes-at-0*: *prefixes 0 j* ↓= *list-encode* [*j*]
⟨*proof*⟩

lemma *prefixes-at-Suc*:
assumes *prefixes t j* ↓ (**is** *?b* ↓)
shows *prefixes* (*Suc t*) *j* = *nxt* (*the ?b*)
⟨*proof*⟩

lemma *prefixes-at-Suc′*:

assumes *prefixes t j* $\downarrow=$ *b*
shows *prefixes* (*Suc t*) *j* = *nxt b*
⟨*proof*⟩

lemma *prefixes-prod-encode*:
assumes *prefixes t j* $\downarrow$
obtains *b* **where** *prefixes t j* $\downarrow=$ *b*
⟨*proof*⟩

lemma *prefixes-converg-le*:
assumes *prefixes t j* $\downarrow$ **and** $t' \leq t$
shows *prefixes* t' *j* $\downarrow$
⟨*proof*⟩

lemma *prefixes-diverg-add*:
assumes *prefixes t j* $\uparrow$
shows *prefixes* (*t* + *d*) *j* $\uparrow$
⟨*proof*⟩

Many properties of *r-prefixes* can be derived from similar properties of *r-next*.

lemma *prefixes-length*:
assumes *prefixes t j* $\downarrow=$ *b*
shows *e-length b* > *t*
⟨*proof*⟩

lemma *prefixes-monotone*:
assumes *prefixes t j* $\downarrow=$ *b* **and** *prefixes* (*t* + *d*) *j* $\downarrow=$ b'
shows *e-length b* $\leq$ *e-length* b'
⟨*proof*⟩

lemma *prefixes-stable*:
assumes *prefixes t j* $\downarrow=$ *b* **and** *prefixes* (*t* + *d*) *j* $\downarrow=$ b'
shows $\forall$ *x*<*e-length b*. *e-nth b x* = *e-nth* b' *x*
⟨*proof*⟩

lemma *prefixes-tl-only-01*:
assumes *prefixes t j* $\downarrow=$ *b*
shows $\forall$ *x*>*0*. *e-nth b x* = *0* $\vee$ *e-nth b x* = *1*
⟨*proof*⟩

lemma *prefixes-hd*:
assumes *prefixes t j* $\downarrow=$ *b*
shows *e-nth b 0* = *j*
⟨*proof*⟩

lemma *prefixes-nontotal-hyp*:
assumes *prefixes t j* $\downarrow=$ *b*
and *prefixes* (*Suc t*) *j* $\uparrow$
and *s b* $\downarrow=$ *i*
shows $\exists$ *x*. φ *i x* $\uparrow$
⟨*proof*⟩

We now consider the two cases from the proof sketch.

abbreviation *case-two j* $\equiv$ $\exists$ *t*. *prefixes t j* $\uparrow$

abbreviation *case-one j* $\equiv$ $\neg$ *case-two j*

In Case 2 there is a maximum convergent iteration because iteration 0 converges.

lemma *case-two*:
  assumes *case-two j*
  shows $\exists\, t.\ (\forall\, t' \le t.\ prefixes\ t'\ j\ \downarrow) \wedge (\forall\, t' > t.\ prefixes\ t'\ j\ \uparrow)$
⟨*proof*⟩

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*
  ⟨*proof*⟩

abbreviation *psi* :: *partial2* (‹ψ›) **where**
  $\psi\ j\ x \equiv eval\ r\text{-}psi\ [j,\ x]$

lemma *psi-in-P2*: $\psi \in \mathcal{P}^2$
  ⟨*proof*⟩

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 $\psi\ j\ x \downarrow= e\text{-}nth\ b\ x$
⟨*proof*⟩

lemma *psi-converg-imp-prefix*:
  assumes $\psi\ j\ x\ \downarrow$
  shows $\exists\, t\ b.\ prefixes\ t\ j \downarrow= b \wedge e\text{-}length\ b > x$
⟨*proof*⟩

lemma *psi-converg-imp-prefix'*:
  assumes $\psi\ j\ x\ \downarrow$
  shows $\exists\, t\ b.\ prefixes\ t\ j \downarrow= b \wedge e\text{-}length\ b > x \wedge \psi\ j\ x \downarrow= e\text{-}nth\ b\ x$
  ⟨*proof*⟩

In both Case 1 and 2, ψ_j starts with j.

lemma *psi-at-0*: $\psi\ j\ 0 \downarrow= j$
  ⟨*proof*⟩

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 $\psi\ j\ x \downarrow= 0 \vee \psi\ j\ x \downarrow= 1$
⟨*proof*⟩

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 $\exists\, y.\ (\forall\, x.\ 0 < x \wedge x < y \longrightarrow \psi\ j\ x \downarrow= 0 \vee \psi\ j\ x \downarrow= 1) \wedge$

$(\forall\, x \geq y.\ \psi\ j\ x\ \uparrow)$
⟨*proof*⟩

definition *longest-prefix* :: *nat* ⇒ *nat* **where**
longest-prefix j ≡ *THE y*. $(\forall\, x{<}y.\ \psi\ j\ x\ \downarrow) \wedge (\forall\, x{\geq}y.\ \psi\ j\ x\ \uparrow)$

lemma *longest-prefix*:
assumes *case-two j* **and** *z* = *longest-prefix j*
shows $(\forall\, x{<}z.\ \psi\ j\ x\ \downarrow) \wedge (\forall\, x{\geq}z.\ \psi\ j\ x\ \uparrow)$
⟨*proof*⟩

lemma *case-two-psi-longest-prefix*:
assumes *case-two j* **and** *y* = *longest-prefix j*
shows $(\forall\, x.\ 0 < x \wedge x < y \longrightarrow \psi\ j\ x \downarrow= 0 \vee \psi\ j\ x \downarrow= 1) \wedge$
$(\forall\, x \geq y.\ \psi\ j\ x\ \uparrow)$
⟨*proof*⟩

The prefix cannot be empty because the process starts with prefix $[j]$.

lemma *longest-prefix-gr-0*:
assumes *case-two j*
shows *longest-prefix j* > *0*
⟨*proof*⟩

lemma *psi-not-divergent-init*:
assumes *prefixes t j* ↓= *b*
shows $(\psi\ j) \triangleright (e\text{-}length\ b - 1) = b$
⟨*proof*⟩

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 ∃ *n*<*longest-prefix j*. ¬ *total1* $(\varphi\ (the\ (s\ ((\psi\ j) \triangleright n))))$
⟨*proof*⟩

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 $f \in \mathcal{R}$
and ⋀*x*. *x* < *longest-prefix j* ⟹ $f\ x = \psi\ j\ x$
shows ¬ *learn-total* φ $\{f\}$ *s*
⟨*proof*⟩

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 $\exists\, n{>}k.\ \varphi\ (the\ (s\ ((\psi\ j) \triangleright n))) \neq \psi\ j$
⟨*proof*⟩

lemma *case-one-not-learn*:
assumes *case-one j*
shows ¬ *learn-lim* φ $\{\psi\ j\}$ *s*
⟨*proof*⟩

lemma *case-one-not-learn-V*:
  assumes *case-one j* **and** $j \geq 2$ **and** $\varphi\ j = \psi\ j$
  shows $\neg$ *learn-lim* φ *V-constotal s*
⟨*proof*⟩

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 $\exists\, n{\geq}m.\ \varphi\ n = vs\ @\ [n] \odot 0^\infty$
⟨*proof*⟩

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** $j \geq 2$ **and** $\varphi\ j = \psi\ j$
  shows $\neg$ *learn-total* φ *V-constotal s*
⟨*proof*⟩

The strategy S does not learn V in either case.

lemma *not-learn-total-V*: $\neg$ *learn-total* φ *V-constotal s*
⟨*proof*⟩

end

lemma *V-not-in-TOTAL*: *V-constotal* $\notin$ *TOTAL*
⟨*proof*⟩

lemma *TOTAL-neq-CONS*: *TOTAL* $\neq$ *CONS*
  ⟨*proof*⟩

The main result of this section:

theorem *TOTAL-subset-CONS*: *TOTAL* $\subset$ *CONS*
  ⟨*proof*⟩

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^k1.

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*
  ⟨*proof*⟩

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*]
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

lemma *if-eq-eq*: (*if v* = *1 then* (*0* :: *nat*) *else 1*) = *0* ⟹ *v* = *1*
⟨*proof*⟩

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*⟩

lemma *r-find-k-total*: *total r-find-k*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *r-prefixes-total*: *total r-prefixes*
⟨*proof*⟩

lemma *r-prefixes-0*: *eval r-prefixes* [*0*, *j*] ↓= *list-encode* [*j*]
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *prefixes-Suc-length*:
e-length (*prefixes j* (*Suc t*)) =
Suc (*e-length* (*prefixes j t*) + *the* (*eval r-find-k* [*prefixes j t*]))
⟨*proof*⟩

lemma *prefixes-length-mono*: *e-length* (*prefixes j t*) < *e-length* (*prefixes j* (*Suc t*))
⟨*proof*⟩

lemma *prefixes-length-mono'*: *e-length* (*prefixes j t*) ≤ *e-length* (*prefixes j* (*t* + *d*))

⟨*proof*⟩

lemma *prefixes-length-lower-bound*: *e-length* (*prefixes j t*) ≥ *Suc t*
⟨*proof*⟩

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*⟩

lemma *prefixes-Suc-last*: *e-nth* (*prefixes j* (*Suc t*)) (*e-length* (*prefixes j* (*Suc t*)) − *1*) = *1*
⟨*proof*⟩

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*⟩

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*⟩

lemma *psi-eq-nth-prefixes*:
assumes $x <$ *e-length* (*prefixes j t*)
shows ψ *j x* ↓= *e-nth* (*prefixes j t*) *x*
⟨*proof*⟩

lemma *psi-at-0*: ψ *j 0* ↓= *j*
⟨*proof*⟩

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*⟩

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*⟩

corollary *hyp-wrong*: φ (*the* (*s* (*e-butlast* (*prefixes j* (*Suc t*))))) ≠ ψ *j*
⟨*proof*⟩

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*⟩

lemma *U0-V0-not-learn-bc*: ¬ *learn-bc* φ ($U_0 \cup V_0$) *s*
⟨*proof*⟩

end

lemma *U0-V0-not-in-BC*: $U_0 \cup V_0 \notin BC$
⟨*proof*⟩

theorem *R1-not-in-BC*: $\mathcal{R} \notin BC$
⟨*proof*⟩

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*:
$\forall \mathcal{I} \in \{CP, TOTAL, CONS, LIM, BC\}.\ U_0 \in \mathcal{I} \wedge V_0 \in \mathcal{I} \wedge U_0 \cup V_0 \notin \mathcal{I}$
⟨*proof*⟩

In order to show the analogous result for FIN consider the classes $\{0^\infty\}$ and $\{0^n10^\infty \mid n \in \mathbb{N}\}$. 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^n10^∞.

lemma *singleton-in-FIN*: $f \in \mathcal{R} \Longrightarrow \{f\} \in FIN$
⟨*proof*⟩

definition *U-single* :: *partial1 set* **where**
U-single ≡ {(λ*x*. *if x* = *n then Some 1 else Some 0*)| *n*. *n* ∈ *UNIV*}

lemma *U-single-in-FIN*: *U-single* ∈ *FIN*
⟨*proof*⟩

lemma *zero-U-single-not-in-FIN*: $\{0^\infty\} \cup$ *U-single* ∉ *FIN*
⟨*proof*⟩

lemma *FIN-not-closed-under-union*: $\exists\, U\ V.\ U \in FIN \wedge V \in FIN \wedge U \cup V \notin FIN$
⟨*proof*⟩

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* ≡
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*
⟨*proof*⟩

lemma *r-div2* [*simp*]: *eval r-div2* [*n*] ↓= *n div 2*
⟨*proof*⟩

definition *r-mod2* ≡ *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*
⟨*proof*⟩

lemma *r-mod2* [*simp*]: *eval r-mod2* [*n*] ↓= *n mod 2*
⟨*proof*⟩

lemma *NUM-closed-under-union*:
assumes *U* ∈ *NUM* **and** *V* ∈ *NUM*
shows *U* ∪ *V* ∈ *NUM*
⟨*proof*⟩

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.