Recursion and Subrecursion over Finite and Infinite Words with Applications to Computable Real Valued Functions

DISSERTATION
zur Erlangung des Grades eines Doktors
der Naturwissenschaften

vorgelegt von
Holger Schulz
geb. am 1969-06-14 in Langenhagen

eingereicht beim Fachbereich 6 · Mathematik
der Universität Siegen

Siegen 2006

Dekan: Prof. Dr. Wolgang Hein

1. Gutachter: Prof. Dr. Dieter Spreen
2. Gutachter: Prof. Dr. Vasco Brattka

Datum der mündlichen Prüfung: 2006-06-26

Acknowledgement

First of all I have to thank Professor Dieter Spreen for enabling me to write this thesis. This work would not have been possible without his support and advice. I owe him a lot.

Special thanks go to Professor Vasco Brattka for kindly accepting to referee this thesis.

Further all my colleagues must not stay unmentioned. I still like the working environment they provide.

Abstract

A recursion scheme for functions on intervals to approximate real-valued functions is described and compared with recursion schemes on finite and infinite words.

Zusammenfassung

Es wird ein Rekursionschema für Funktionen auf Intervallen, die reellwertige Funktionen approximieren, beschrieben und mit Rekursionsschemata auf endlichen und unendlichen Wörtern verglichen.

Contents

Chapter 1

Introduction

Computability is one of the most important topics in theoretical computer science and mathematical logic. In the beginning computabality over discrete sets — like the set of natural numbers or sets of finite words — was considered. While this classical theory is deemed as complete and balanced, the question of computability over uncountable sets, e.g. the real numbers, has still a lot of open aspects and is subject to current research.

There are several approaches to computability over the real numbers, which may lead to different sets of computable functions depending on the definition of real numbers they are based on and on which classical computational models are used.

At first we like to draw attention to a machine model. Based on the work of Turing ([Tur37a, Tur37b]) and Grzegorczyk ([Grz57]) from the thirties and fifties of the twentieth century, respectively, Weihrauch and his followers developed the model of the *Type Two Turing Machine*([Wei97a, Wei00]). Type Two Turing machines are Turing Machines which compute a infinite output from an infinite input.

This model takes account of the fact that machines cannot compute on the real numbers itself but on representations of them. To be able to represent uncountable sets, infinite information, written as infinite words, is needed. As a main result it turns out that computability of a real valued function depends on the chosen representation. Some of them appear to be more useful than others. For instance, sequences of intervals converging to a real number can be used as a reasonable representation. We obtain an equivalent representation using digits, when we add a digit -1, written as $\bar{1}$, to the binary digits 0 and 1.

This representation allows to define the length of (finite prefixes of) of input and output and thus leads to a notion of complexity of real valued functions. One can ask, how many steps a machine must compute to gain an output of certain length. This defines the time complexity. On the other hand, one can ask how many input digits are needed to compute a certain number of output digits. This defines the *lookahead* of a function.

Another model is based on the theory of *domains*, i.e. *complete partial orders (* CPO*-s).* These are structures which allow to order their elements by their amount of information. We write $x \sqsubseteq y$, iff y carries at least as much information as x. Complete partials orders — developed by Scott ([Sco70]) — are mainly used to describe semantics of programming languages.

Using intervals to approximate real numbers, the accuracy of the approximation can be understood as the information carried by an interval. A smaller interval approximates a real number in a more accurate way than a larger one. We can order the interval by inverse set inclusion, i.e. $\mathbf{x} \sqsubseteq \mathbf{y} \iff \mathbf{x} \supseteq \mathbf{y}$ for intervals $\mathbf{x}$ and $\mathbf{y}$, and obtain a CPO. Escardo ([Esc96b, Esc96a]) calls this CPO the *partial real line* — or rather the *partial unit interval* $\mathcal{I}$ in the case of interval boundaries between 0 and 1 — and uses them to describe the semantics *Real PCF*. Real PCF is (the theoretical model of) a functional programming languages with a type of real numbers interpreted by intervals. Real PCF defines a computability notion for real numbers and real valued functions equivalent to that based on the Type Two machines ([Sch97, SS00]).

Real PCF is based on the lambda calculus and uses recursion as a programming concept. It turns out that a lot of functions can be described by means of recursion schemes on intervals, called the *dyadic recursion*.

Since recursion is at the heart of of the classical recursion theory, it is well studied for discrete objects. Recursion depth as a measure for complexity of functions has been considered for natural valued computable functions ([Grz53, Sch69, Hei61]) and adopted to computable functions over other countable sets like words ([Wei74]) or terms ([Spr95]).

We want to transfer this approach to dyadic recursion. In particular, we ask how dyadic recursion can be simulated by primitive recursion over words with letters $\overline{1}$, 0 and 1. The base functions for dyadic recursion are left concatenation of the intervals $L = [0, 1/2]$, $C = [1/4, 3/4]$ and $R = [1/2, 1]$. These operations correspond to the use of the letters $\overline{1}$, 0 and 1 in the signed digit representation

We develop *uniform* and *ε-uniform recursion*, to define functions on finite words that approximate functions on infinite words. The definition of these schemes will mimic dyadic recursion as close as possible. In uniform recursion all parameters are recursion parameters, there are no side parameters. ε-recursion can be seen as recursion without a start value. There are no start values in the dyadic recursion.

We show that ε-recursive functions are monotonic. This fits very well into our concepts, because monotonicity is needed to approximate infinite computations by finite ones.

In dyadic recursion one has to test whether a given interval lies in the left or the right half of the unit interval ([ES97]). The crucial point in the simulation of interval functions by word functions is a situation in which this test cannot give a clear answer. Think of an interval, e.g. $[1/4, 3/4]$, which overlaps the center of the unit interval. One cannot say this interval is located either in the left or in the right half of the unit intervals, but it can be considered as having two parts, one in the left half and one in the right half of the unit interval.

In this case two parallel computations will start, one for the part of the considered interval lying in the left half of the unit interval, and one for the right part. Both results are combined by subsuming their common information, which is domain-theoretically represented by the infimum. In case of interval-CPO-s the infimum is defined by

$$[\xi, \xi'] \sqcap [\eta, \eta'] := [\min(\xi, \eta), \max(\xi', \eta')]$$

Assume these intervals are results of two branches in a parallel computation of the interval function $F : \mathcal{I} \to \mathcal{I}$ in the following context:

Let $[\zeta, \zeta']$ be an interval, that contains the center of the unit interval, i.e. $\zeta \leq \frac{1}{2} \leq \zeta'$ and $F([\zeta, 1/2]) = [\xi, \xi']$ and $F([1/2, \zeta'] = [\eta, \eta']$. Then $[\xi, \xi'] \cap [\eta, \eta'] \neq \emptyset$ if F is continuous. The infimum is the union of sets.

Further, if two sequences $([\xi_i, \xi_i'])_{i\in\mathbb{N}}$ and $([\eta_i, \eta_i'])_{i\in\mathbb{N}}$ are converging to the same real number, say $\alpha \in [0, 1]$, then $\alpha \in [\xi_i, \xi_i'] \cap [\eta_i, \eta_i']$ and the sequence

$$\big([\xi_i, \xi_i'] \cup [\eta_i, \eta_i']\big)_{i\in\mathbb{N}} = \big([\xi_i, \xi_i'] \sqcap [\eta_i, \eta_i']\big)_{i\in\mathbb{N}}$$

will converge to α, as well.

We have to simulate parallel computations like this on words. We need a word function simulating the infimum. We use the properties described above. Whenever a computation is split into two parallel branches, two situations might occur, both have to be handled.

- Both branches might be valid, i.e. both of them deliver an approximation of the result. The result can be approximated by the common information of the result of the branches.
- One of the branches may turn out to be invalid, i.e. the other branch will approximate the result. The invalid branch has to be dismissed.

Our simulation of parallelism must be able to handle both situations. In every step of the computation a branch might turn out as invalid. But the case that both branches stay valid is possible, too. We have to be prepared for both possibilities. In particular, we cannot wait until the validity question is solved.

To understand how parallel computations appear in word functions, one has to know how words can be interpreted as intervals or real numbers. A finite word $v_1 \ldots v_l \in \Sigma^*$ over the alphabet $\Sigma = \{\bar{1}, 0, 1\}$ denotes the interval

$$\left[\frac{1}{2} + \left(\sum_{i=1}^{l} v_i \cdot 2^{-(i+1)}\right) - 2^{-(l+1)}, \frac{1}{2} + \left(\sum_{i=1}^{l} v_i \cdot 2^{-(i+1)}\right) + 2^{-(l+1)}\right]$$

E.g. consider the sequence (w_i) of words $w_i = 0^i$ denoting the interval $[\xi_i, \xi_i'] = \left[\frac{2^i-1}{2^{i+1}}, \frac{2^i+1}{2^{i+1}}\right]$.

We split these intervals into two halves $\left[\frac{2^i-1}{2^{i+1}}, \frac{1}{2}\right]$ and $\left[\frac{1}{2}, \frac{2^i+1}{2^{i+1}}\right]$, denoted by the words $\bar{1}1^i$ and $1\bar{1}^i$, respectively.

Both interval parts converge to $1/2$, or rather $[1/2, 1/2]$, and thus the function values will both converge to $F([1/2, 1/2])$ when applying a function, say $F : \mathcal{I} \to \mathcal{I}$.

The simulating word function f inherits the cases for digits $\bar{1}$ and 1 directly from the dyadic recursion scheme of F. So the function values $f(\bar{1}1^i)$ and $f(1\bar{1}^i)$ can easily be computed. Our infimum operator *inf* allows us the computation of $f(w_i)$ by $f(w_i) = \mathit{inf}(f(\bar{1}1^i), f(1\bar{1}^i))$.

On the other hand consider the case that a prefix of the input value appears to denote an interval containing 1/2, but actually a further digit may disclose that one of the branches is invalid. Consider the word $w_i' = 0^i 1$. On reading the first i input digits both branches appear valid. Reading the next input digit proofs the left branch invalid. It is cut off, we continue with the right branch only.

An infimum operator on words solves this task. It can be defined with recursion depth one. We use the this operator to show that every dyadic recursive function with a certain recursion depth can be simulated by ε-recursive function with the same recursion depth.

This is a further tile in the puzzle of connections between classical computability theory and the theory of computable real valued functions.

Chapter 2 presents some preliminaries on primitive recursion, CPO-s and computability of real valued functions.

Chapter 3 will present recursion on word functions. The scheme of uniform recursion is developed and monotonic functions are considered.

Chapter 4 introduces interval-CPO-s and dyadic recursion. The simulation of dyadic recursion is developed. Several cases that might occur are considered. This results in a definition of an infimum on words which is used for the construction of ε-uniformly recursive functions to simulate dyadic recursive functions.

Chapter 2

Preliminaries

2.1 Conventions

Symbols will be used in certain contexts in this work as described in the following.

l, r, n k and similar stand for natural numbers.

We use n for the hierarchy levels, k for the arity of a function (the number of arguments) and r for the number of letters in the respective alphabet. l usually denotes the word length and similar stuff.

x, y and z and similar denote natural arguments and results of functions, where x is usually the recursion parameter. $\overline{y} = (y_1, \dots, y_k)$ denotes a tupel of natural numbers.

u, v and w will denote words. Σ will denote an alphabet, Σ^* the set of finite words, Σ^ω the set of infinite words, and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ the set of finite and infinite words over Σ.

f, g, h and similar denote functions on natural numbers or words.

Greek letters like ξ, ζ, η and similar denote real numbers, and φ, γ denote real-valued functions.

Bold faced letters like $\mathbf{x}$ and $\mathbf{y}$ are used for the elements of the partial unit interval or rather the partial real line, and we use capital letters like F and G for interval functions. Longer names of interval functions will start with a capital letter like *Mir* or *Id*.

Aberrantly the basic functions on intervals like cons_L and tail_L etc. are written in roman letters.

You might also want to consult the Index of Symbols (page 107).

2.2 Primitive Recursion and the Classical Grzegorczyk Hierarchy

The increase of a natural valued function may be bounded by another function. The Grzegorczyk hierarchy classifies functions by these bounds and allows to compare bounds and recursion depths.

We start with some definitions.

2.2.1 Definition

1. *A function $f : \mathbb{N}^k \to \mathbb{N}$ is defined by* simultaneous substitution *from the functions $g : \mathbb{N}^l \to \mathbb{N}$ and $h_1, \dots, h_l : \mathbb{N}^k \to \mathbb{N}$, if for all $x_1, \dots, x_k \in \mathbb{N}$ holds*

$$f(x_1, \dots, x_k) = g(h_1(x_1, \dots, x_k), \dots, h_l(x_1, \dots, x_k))$$

   We write $f = Sub(g; h_1, \dots, h_l)$

2. *A function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined by* primitive recursion *from the functions $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$ if*

$$\begin{aligned} f(0, \overline{y}) &= g(\overline{y}) \\ f(x+1, \overline{y}) &= h(x, f(x, \overline{y}), \overline{y}) \end{aligned}$$

   We write $f = Prim(g; h)$

3. *The functions $f_1, \dots, f_m : \mathbb{N}^{k+1} \to \mathbb{N}$ are defined by* simultaneous primitive recursion *from the functions $g_1, \dots, g_m : \mathbb{N}^k \to \mathbb{N}$ and $h_1, \dots, h_m : \mathbb{N}^{n+k+1} \to \mathbb{N}$ if*

$$\begin{aligned} f_i(0, \overline{y}) &= g_i(\overline{y}) \\ f_i(x+1, \overline{y}) &= h_i(x, f_1(x, \overline{y}), \dots, f_m(x, \overline{y}), \overline{y}) \end{aligned}$$

   We write $\overline{f} = SimPr(\overline{g}; \overline{h})$.

We use $\overline{y}$ as a short hand for $(y_1, \dots, y_k)$. In the recursion scheme we call x the *recursion parameter* and $\overline{y}$ the *side parameters*.

2.2.2 Definition

1. *The* basic functions *are*

   - *the* projections $proj^k_i : \mathbb{N}^k \to \mathbb{N}, \overline{y} \mapsto y_i$ *for* $1 \leq i \leq k$
   - *the* constant functions $const^k_j : \mathbb{N}^k \to \mathbb{N}, \overline{y} \mapsto j$ *for* $j, k \in \mathbb{N}$
   - *the* successor function $succ : \mathbb{N} \to \mathbb{N} : x \mapsto x + 1$

2. *Let PR_0 be the smallest set, that contains all basic functions and is closed under simultaneous substitution.*

3. *Let PR_{n+1} be the smallest set, that contains*

$$PR_n \cup \{f \mid f = \mathit{Prim}(g;h),\ g,h \in PR_n\}$$

   and is closed under simultaneous substitution.

4. *The number n is called the* recursion depth *of the functions in PR_n.*

5. $PR = \bigcup_{n\in\mathbb{N}} PR_n$

6. *The classes SPR and SPR_n of simultaneous primitive recursive functions are defined analogously.*

7. *A function f is called* primitive recursive *if $f \in PR$.*

2.2.3 Lemma
For $n \geq 1$ the class PR_n is closed under case distinction*: Let $h_0, h_1 : \mathbb{N}^k \to \mathbb{N} \in PR_n$ and*

$$h : \mathbb{N}^{k+1} \to \mathbb{N}, \begin{cases} (0,\overline{y}) & \mapsto h_0(\overline{y}) \\ (x+1,\overline{y}) & \mapsto h_1(\overline{y}) \end{cases}$$

Then $h \in PR_n$

In this case we write $h = \mathit{If}(h_0,h_1)$

Proof. Let

$$h' : \mathbb{N}^3 \to \mathbb{N}, \begin{cases} (0,y,z) & \mapsto y \\ (x+1,y,z) & \mapsto z \end{cases}$$

Thus $h' = \mathit{Prim}(\mathit{proj}_1^2; \mathit{proj}_4^4) \in PR_1$ and with $h(x,\overline{y}) = h'(x, h_0(\overline{y}), h_1(\overline{y}))$ follows $h = \mathit{Sub}(h'; \mathit{proj}_1^{k+1}, \mathit{Sub}(h_0; \mathit{proj}_2^{k+1}, \ldots, \mathit{proj}_{k+1}^{k+1}), \mathit{Sub}(h_1; \mathit{proj}_2^{k+1}, \ldots, \mathit{proj}_{k+1}^{k+1})) \in PR_n$. ∎

2.2.4 Corollary
$h_0, h_1 \in PR_0 \Rightarrow \mathit{If}(h_0,h_1) \in PR_1$.

2.2.5 Remark
With aid of the substitution we can construct more general case distinctions like

$$h(\overline{y}) : \mathbb{N}^k \to \mathbb{N}, \overline{y} \mapsto \begin{cases} h_0(\overline{y}) & \text{if } g(\overline{y}) = 0 \\ h_1(\overline{y}) & \text{if } g(\overline{y}) > 0 \end{cases}$$

with $h_0, h_1, g : \mathbb{N}^k \to \mathbb{N}$.

Obviously $f = \mathit{Sub}(\mathit{If}(h_0,h_1); g, \mathit{proj}_1^k, \ldots \mathit{proj}_k^k))$

2.2.6 Example
1. $\mathit{add} : \mathbb{N}^2 \to \mathbb{N}, (x,y) \mapsto x + y \in PR_1$

2. $mult : \mathbb{N}^2 \to \mathbb{N}, (x, y) \mapsto x \cdot y \in PR_2$

3. $exp : \mathbb{N} \to \mathbb{N}, x \mapsto 2^x \in PR_2$

2.2.7 Definition
A function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined by bounded primitive recursion *from the functions $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ and $j : \mathbb{N}^{k+1} \to \mathbb{N}$ if $f = Prim(g; h)$ and for all $(x, \overline{y}) \in \mathbb{N}^{n+1}$*

$$f(x, \overline{y}) \leq j(x, \overline{y})$$

We write $f = BoundPr(g, h; j)$.

2.2.8 Definition
We define the n-th Ackermann function *B_n by*

1. *$B_0 : \mathbb{N} \to \mathbb{N}$ with*

$$B_0 : \begin{cases} 0 \mapsto 1 \\ 1 \mapsto 2 \\ x + 2 \mapsto x + 4 \end{cases}$$

2. *$B_{n+1} : \mathbb{N} \to \mathbb{N}, x \mapsto B_n^x(1)$*

3. *$B : \mathbb{N}^2 \to \mathbb{N}, (x, y) \mapsto B_x(y)$*

2.2.9 Lemma
1. *$B_n \in PR$ for all $n \in \mathbb{N}$*

2. *$B \notin PR$*

2.2.10 Definition
The n-th Grzegorczyk class *$\mathcal{E}_n$ is the smallest set which contains the basic functions, the n-th Ackermann function B_n and is closed under substitution and bounded primitive recursion.*

2.2.11 Theorem (Hierarchy)
1. *$PR_1 \subsetneq SPR_1 \subsetneq \mathcal{E}_1$*

2. *$\mathcal{E}_n \subsetneq PR_n = SPR_n = \mathcal{E}_{n+1}$ for all $n \geq 2$*

3. *$\mathcal{E}_n \subsetneq \mathcal{E}_{n+1}$ for all $n \geq 0$*

2.2.12 Definition
$f : \mathbb{N}^{k+1} \to \mathbb{N}$ is built by application of the bounded μ-operator from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ and $h : \mathbb{N}^{k+1} \to \mathbb{N}$ *(or: from h with bound g) if*

$$\begin{aligned} f(x, \overline{y}) &= \mu z \leq g(x, \overline{y}).\,[h(z, \overline{y}) = 0] \\ &= \begin{cases} \min M & \text{if } M \neq \emptyset \\ g(x, \overline{y}) + 1 & \text{else} \end{cases} \end{aligned}$$

with

$$M = \{z \mid z \leq g(x,\overline{y}) \wedge h(z,\overline{y}) = 0\}$$

We write $f = \mu^{\leq}(g,h)$*, in case* $g = proj_1^{n+1}$ *we write* $f = \mu^{\leq}(h)$*.*

The bounded μ-operator can easily be simulated by bounded primitive recursion.

Let $f(x,\overline{y}) = \mu z \leq g(x,\overline{y}).\,[h(z,\overline{y}) = 0]$. Then

$$f(x,\overline{y}) = f'(g(x,\overline{y}),\overline{y})$$

with

$$\begin{aligned} f'(0,\overline{y}) &= \begin{cases} 0 & \text{if } h(0,\overline{y}) = 0 \\ 1 & \text{otherwise} \end{cases} \\ f'(z+1,\overline{y}) &= \begin{cases} f'(z,y) & \text{if } f'(z,\overline{y}) \leq z \\ z+1 & \text{if } f'(z,\overline{y}) = z+1 \wedge h(z,\overline{y}) = 0 \\ z+2 & \text{otherwise} \end{cases} \end{aligned}$$

and obviously

$$f(x,\overline{y}) \leq succ(g(x,\overline{y}))$$

This shows that $\mathcal{E}_n$ is closed under application of the bounded μ-operator for $n \geq 1$.

2.2.13 Example
The bounded μ-operator can be used to compute inverse functions of monotonically increasing functions. The scheme used for that purpose can be seen in the following examples.

1. $\lfloor\sqrt{x}\rfloor = \mu z \leq x.\,(z+1)^2 > x$
2. $x \operatorname{div} y = \mu z \leq x.(z+1)\cdot y > x$

Omitting the bounds of the μ-operator leads to a larger set of functions.

2.2.14 Definition
$f : \mathbb{N}^k \to \mathbb{N}$ *is built by use of the* μ-operator from $h : \mathbb{N}^{k+1} \to \mathbb{N}$ *if*

$$\begin{aligned} f(x,\overline{y}) &= \mu z.\,[h(z,\overline{y}) = 0] \\ &= \begin{cases} \min M & \textit{if } M \neq \emptyset \\ \uparrow & \textit{else} \end{cases} \end{aligned}$$

with

$$M = \{z \mid h(z, \overline{y}) = 0 \wedge \forall j < z.\, h(j, \overline{y})\downarrow\}$$

$f(x)\downarrow$ means that f is defined for input value x, and $f(x)\uparrow$ means that f is not defined for input value x. This notion is necessary as functions defined via the μ-operator may not be total.

The set of *μ-recursive functions* is the smallest set which contains the basic functions and is closed under substitution, primitive recursion and application of the μ-operator. These functions are also called *recursive* or *computable*.

The last notion is based on the fact that recursive functions coincide with functions defined by other computability models, like Turing machines and others.

The set of primitive recursive functions is a proper subset of the set of μ-recursive functions. Primitive recursive functions are always total, μ-recursive functions may be partial.

Consider the Ackerman function B. It is μ-recursive and total but not primitive recursive.

2.3 Type Two Turing Machines and Infinite Computations

The Type-Two-Theory of computability considers how the notion of a computable function with infinite input and output can be defined in terms of classical recursion theory. Details can be found in [Wei97a, Wei97b, Wei00]

We could say, a real number ξ is computable iff a total recursive function $f_\xi : \mathbb{N} \to \mathbb{N}$ exists with $f_\xi(0) = \langle s, p\rangle$ where $s \in \{0, 1\}, p \in \mathbb{N}$ and $f_\xi(i) \in \{0, \ldots, 9\}$ for $i > 0$, so that

$$\xi = (2(f_\xi(0))_0 - 1)\left((f_\xi(0))_1 + \sum_{i=1}^{\infty} f_\xi(i) \cdot 10^{-i}\right)$$

Remember that $\langle\cdot\rangle : \mathbb{N}^* \to \mathbb{N}, (x_0, \ldots, x_{n-1}) \mapsto \langle x_0, \ldots, x_{n-1}\rangle$ is a primitive recursive coding function, which has primitive recursive projections $(\cdot)_k : \mathbb{N} \to \mathbb{N}, \langle x_0, \ldots, x_{n-1}\rangle \mapsto x_k,\ k = 0, \ldots, n-1$ and a length function $lth : \mathbb{N} \to \mathbb{N} : \langle x_0, \ldots, x_{l-1}\rangle \mapsto l$. We can concatenate two codes via $* : \mathbb{N} \times \mathbb{N} \to \mathbb{N}(\langle x_0, \ldots, x_{l-1}\rangle, \langle x_l, \ldots, x_{l+m}\rangle \mapsto \langle x_0, \ldots, x_{l+m}\rangle$.

s represents the sign of ξ, p the integer part. All other values of f_ξ represent single digits.

A first observation is that we have to compute all values of f_ξ to know ξ. This can be understood as an infinite computation.

Of course a real number can be described in other ways than listing its digits. As known from classical calculus real numbers can be approximated by rational numbers. We will need a *numbering* of the rational numbers, i.e. a surjective function from the naturals onto the rationals, which might be given by

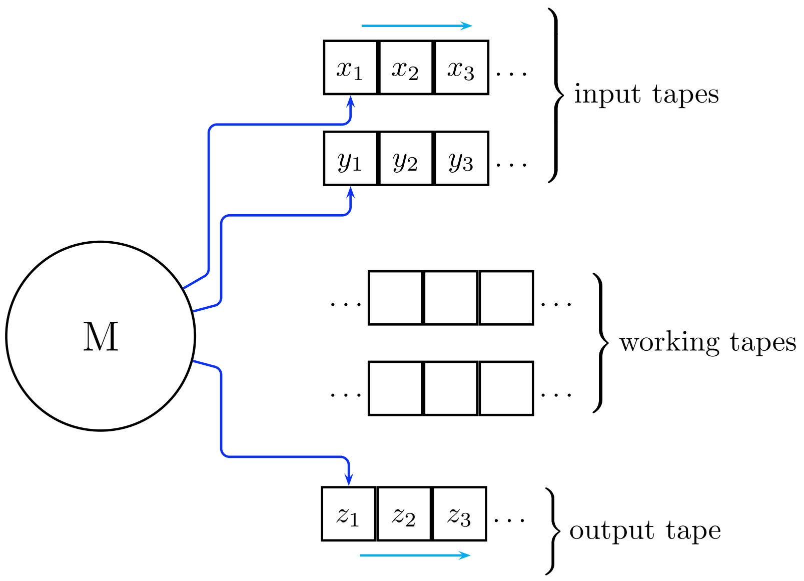


Figure 2.1: Principle of a TTM

$$\nu_{\mathbb{Q}} : \mathbb{N} \rightharpoonup \mathbb{Q} : \langle i, j, k \rangle \mapsto \begin{cases} \frac{i-j}{k} & \text{if } k \neq 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We can extend it to $\overline{\mathbb{Q}} = \mathbb{Q} \cup \{-\infty, +\infty\}$ by

$$\nu_{\overline{\mathbb{Q}}} : \mathbb{N} \to \overline{\mathbb{Q}} : \langle i, j, k \rangle \mapsto \begin{cases} \frac{i-j}{k} & \text{if } k \neq 0 \\ -\infty & \text{if } k = 0 \wedge j < i \\ +\infty & \text{if } k = 0 \wedge i \leq j \end{cases}$$

Now we might define a real number ξ to be computable if a total computable function g_ξ exists with

$$\xi = \sup_{n \in \mathbb{N}} \nu_{\mathbb{Q}}(g_\xi(2n)) = \inf_{n \in \mathbb{N}} \nu_{\mathbb{Q}}(g_\xi(2n+1))$$

Are those definitions equivalent? It is easy to see that you can compute nested intervals from decimal digits. But it is not possible to compute decimal digits from nested intervals (see Lemma 2.3.6).

Moreover we must ask how a computable function can be defined based on those definitions. We could think of functions that compute on Kleene indices of functions that represent real numbers. It is not our aim to transform programs but to compute with real numbers as directly as possible. So we have to compute on the *representations*.

Computing w.r.t. to representations is easier to handle, when we use Turing machines instead of recursive functions. Infinite computations can be represented more intuitively by reading from an infinite input tape and writing on an infinite output tape.

A *Type Two Turing Machine* is a Turing Machine allowing infinite input and output. The only restriction is that the input and output tapes are used in one direction only. This is indicated in Figure 2.3. That restriction directly delivers the finiteness property.

2.3.1 Theorem (Finiteness property)
Every finite prefix of the output depends on a finite prefix of the input only.

2.3.2 Definition
Let Σ be an arbitrary (finite) alphabet and M a set with cardinality of the continuum. A surjective function $\Sigma^\omega \rightharpoonup M$ is called a representation.

In the similar way, we can define a *notation* of a countable set M to be a surjective function $\Sigma^* \rightharpoonup M$ [Wei97a, Wei97b]. We could use numberings as well. It can be shown that numberings and notations are equivalent[Sch97].

As announced we will use representations to define computability on the reals.

2.3.3 Definition

1. *A real number ξ is called ρ-computable for a representation $\rho : \Sigma^\omega \rightharpoonup \mathbb{R}$ if there is a TTM exist produces an infinite output $\sigma_0\sigma_1\ldots$ so that*

$$\rho(\sigma_0\sigma_1\ldots) = \xi$$

2. *Let $\rho_0, \ldots, \rho_n : \Sigma^\omega \rightharpoonup \mathbb{R}$ be representations of the reals and $f : \mathbb{R}^n \rightharpoonup \mathbb{R}$ a function, f is called $(\rho_1, \ldots, \rho_n; \rho_0)$-*computable *if a machine M exists which computes a function $f_M : (\Sigma^\omega)^n \rightharpoonup \Sigma^\omega$ with*

$$f(\rho_1(w_1), \ldots, \rho_n(w_n)) = \rho_0(f_M(w_1, \ldots, w_n)),\ \textit{whenever}\ f(\rho_1(w_1), \ldots, \rho_n(w_n))\downarrow$$

2.3.4 Lemma

1. *Let ξ be ρ-computable and $f : \mathbb{R} \rightharpoonup \mathbb{R}$ $(\rho; \rho')$-computable, then $f(\xi)$ is ρ'-computable.*
2. *Let $f, g : \mathbb{R} \rightharpoonup \mathbb{R}$ be $(\rho'; \rho'')$ and $(\rho; \rho')$-computable, then $f \circ g$ is $(\rho; \rho'')$-computable.*

PROOF. With the aid of the finiteness property the computation of two machines can be simulated by a single machine, which computes what is expected. ■

Defining computability of real functions with respect to the used representations is necessary because of a lack of equivalence, i.e. if we use different representations the classes of computable functions may also differ.

We call a representation ρ *reducible to* ρ' ($\rho \leq \rho'$) if a machine M exists which computes a function $f_M : \Sigma^\omega \rightharpoonup \Sigma^\omega$ with $\rho'(f_M(w)) = \rho(w)$ for all $w \in dom(\rho)$, i.e. $id_{\mathbb{R}} : \xi \mapsto \xi$ is $(\rho; \rho')$-computable. Two representations are *equivalent* ($\rho \equiv \rho'$) if each of them is reducible to the other.

We have given some ideas how to represent real numbers. We will show how they fit in the defined notions.

2.3.5 Definition

- *One of the most important representations of the reals is the* interval representation*:*

$$\rho_I : \Sigma^\omega \rightharpoonup \mathbb{R}, u_0\sharp v_0\sharp u_1\sharp v_1 \ldots \mapsto \xi$$

for $\xi = \inf \dot{v}_i = \sup \dot{u}_i$*, where* $u \mapsto \dot{u}$ *is a numbering of the rationals and* Σ *is a suitable alphabet.*

- *Of course we may give a definition based on the decimal digits.*

$$\delta_{dec} : \Sigma^\omega \rightharpoonup \mathbb{R}, \sigma a_n \ldots a_0.a_{-1}a_{-2} \ldots \mapsto \sigma \sum_{i=-\infty}^{n} a_i 10^i$$

with $\sigma \in \{-,+\}$ *and* $a_i \in \{0,\ldots,9\}$*.*

2.3.6 Lemma
$\delta_{dec} \leq \rho_I$ *but* $\rho_I \not\leq \delta_{dec}$

Proof.

- To a given decimal representation $w = \sigma a_n \ldots a_0.a_{-1}a_{-2}\ldots$ an interval representation $w' = u_0\sharp v_0\sharp u_1\sharp v_1 \ldots$ with

$$[\dot{u}_i, \dot{v}_i] = \left[\sum_{j=-i}^{n} a_j 10^j - 10^{-i}, \sum_{j=-i}^{n} a_j 10^j + 10^{-i}\right]$$

can be computed by a TTM. In particular using the above numbering of $\mathbb{Q}$ the output consists of

$$u_i = \begin{cases} \langle \sum_{j=0}^{n+i} a_j 10^j, 1, 10^i \rangle & \text{if } \sigma = - \\ \langle 0, \sum_{j=0}^{n+i} a_j 10^j + 1, 10^i \rangle & \text{if } \sigma = + \end{cases}$$

and similar for v_i.

- Now we assume, that the interval representation is reducible to the decimal representation, i.e. we have a machine that computes decimal representations from intervals. Assume that the machine has to deal with an input $w = u_0\sharp v_0\sharp u_1\sharp v_1 \ldots$ with

$$[u_i, v_i] = \left[\sum_{j=-1}^{-i} 9 \cdot 10^j, \sum_{j=0}^{-i} 10^j\right]$$

Because of the finiteness property the machine cannot decide whether the first digit of the output has to be a "0" or a "1". Assume after reading n intervals of the above kind the machine decides "1" to be the first digit of the output. It has to produce the same output for another input $w' = u'_0\sharp v'_0\sharp u'_1\sharp v'_1\sharp \ldots$ like

$$[u'_i, v'_i] = \begin{cases} = & [u_i, v_i] & \text{if } 0 \leq i \leq n \\ & \left[\sum_{j=-1}^{-n} 9 \cdot 10^j, \sum_{j=-1}^{-n} 9 \cdot 10^j\right] & \text{if } i > n \end{cases}$$

Thus

$$\rho_I(w') = 0.\underbrace{9\ldots9}_{n \text{ digits}} < 1$$

So the output cannot begin with the digit “1”.

The same contradiction occurs in the other case.

■

But a non-uniform version of the equivalence holds, in the sense that a real number is ρ_I-computable iff it is δ_{dec}-computable (see [Wei00, Sch97]).

We can improve digit representations if we allow negative digits to be used. We give the *negative digit binary representation*

$$\rho_2 : \Sigma^\omega \rightharpoonup \mathbb{R}, a_n \ldots a_0.a_{-1}a_{-2}\ldots \mapsto \sum_{i=-\infty}^{n} a_i 2^i$$

with $a_i \in \{\bar{1}, 0, 1\}$.

2.3.7 Lemma

$$\rho_2 \equiv \rho_I$$

Proof. [Wei00, Sch97] ■

Additionally it may be mentioned that all digit representations allowing negative digits are equivalent, i.e we are free to choose one of the representations. The negative digit binary representation will serve as an example for all others.

2.4 Domain Theoretic Preliminaries

The following preliminaries are not used until Chapter 4.

2.4.1 Definition

1. *A* partial order *is a pair $(D, \sqsubseteq)$ where D is a non-empty set and $\sqsubseteq \subseteq D \times D$ is a reflexive, transitive and symmetric relation on D.*

2. *A non-empty subset $X \subseteq D$ of a partial ordered set is called* directed *if $\forall x, y \in X. \exists z \in X : x \sqsubseteq z \wedge y \sqsubseteq z$, i.e. for each two elements of X a common upper bound exists.*

3. *$z \in D$ is called* least upper bound (lub, supremum) *of a subset $X \subseteq D$ if*

   - $\forall x \in X.\, x \sqsubseteq z$
   - $\forall y \in D.\, (\forall x \in X.\, x \sqsubseteq y) \Rightarrow z \sqsubseteq y$

4. $(D, \sqsubseteq, \bot)$ *is called a* complete partial order (CPO) *if*

   - $(D, \sqsubseteq)$ *is a partial order,*
   - $\bot$ *is a the least element of* D*, i.e.* $\forall d \in D : \bot \sqsubseteq d$*,*
   - *every directed subset of* D *has a lub.*

We write D instead of $(D, \sqsubseteq, \bot)$ if no confusions are expected. The lub of a directed set X is written as $\bigsqcup X$. We write $x \sqcup y$ instead of $\bigsqcup\{x, y\}$.

2.4.2 Definition
Two elements x, y *of a* CPO *are called* compatible *if they have a common upper bound, i.e.* $\exists z \in D : x \sqsubseteq z \wedge y \sqsubseteq z$*. We write* $x \uparrow y$*.*

For any set M we can define a *flat* CPO $(M_\bot, \sqsubseteq, \bot)$, with $M_\bot := M \mathbin{\dot\cup} \{\bot\}$ and

$$x \sqsubseteq y \iff (x = \bot \vee x = y)$$

In this way we obtain the flat CPO-s of naturals $\mathbb{N}_\bot$ and of Boolean values $\{\mathbf{tt}, \mathbf{ff}\}_\bot$.

2.4.3 Definition
For CPO*-s* D *and* E *a function* $f : D \to E$ *is called* continuous *if for every directed set* $X \subseteq D$

- *the set* $f(X) \subseteq E$ *is directed and*
- $f(\bigsqcup X) = \bigsqcup f(X)$*.*

2.4.4 Lemma
Every continuous function is monotonic.

Proof.[AJ94, DW80] ■

2.4.5 Lemma
The set of continuous functions $[D \to E]$ *with the pointwise ordering*

$$f \sqsubseteq_{[D \to E]} g \iff (\forall d \in D \; f(d) \sqsubseteq_D g(d))$$

and least element

$$\bot_{[D \to E]} = \lambda d \in D.\bot_E$$

is a CPO*.*

Proof.[AJ94, DW80] ■

2.4.6 Definition
A pair of continuous functions $s : D \to E$ and $r : D \to E$ is called

- *a* section-retraction pair *if $r \circ s = \mathrm{id}_E$,*
- *a* projection-embedding pair *if additionally $s \circ r \sqsubseteq \mathrm{id}_E$.*

2.4.7 Definition
Let D be a CPO*.*

1. *The* way-below order *$\ll \subseteq D \times D$ is defined by*

$$d \ll e :\iff \forall \text{ directed } M \subseteq D : e \sqsubseteq \bigsqcup M \Rightarrow \exists m \in M : d \sqsubseteq m$$

2. *$d \in D$ is called* compact (isolated) *if $d \ll d$.*

The set of compact elements of D is denoted by $b(D)$.

2.4.8 Definition
$B \subseteq D$ is called a basis *of D if for all $d \in D$ the set $\{b \in B \mid b \ll d\}$ is directed and $d = \bigsqcup \{b \in B \mid b \ll d\}$.*

A CPO *is called*

Ê

- continuous *if it has a basis and*
- ω-continuous *if it has a a countable basis.*

2.4.9 Lemma
For a continuos CPO *D with basis B holds: $b(D) \subseteq B$.*

In some cases two elements $d, e \in D$ of a CPO D might share common information f, i.e. $f \sqsubseteq d$ and $f \sqsubseteq e$. We are interested in maximal common information, called the *infimum* or *greatest lower bound* of two elements.

2.4.10 Definition
2.4.1 Let $(D, \sqsubseteq, \bot)$ be a CPO *and $d, e \in D$. $f \in D$ is called the* infimum *of d and e if*

- *$f \sqsubseteq d$ and $f \sqsubseteq e$,*
- *if $f' \sqsubseteq d$ and $f' \sqsubseteq e$ then $f' \sqsubseteq f$.*

A very important property of continuous functions is the existence of a least fixed point. This is described in the *Fixed Point Theorem of Kleene and Tarski* (see [AJ94]).

2.4.11 Theorem

1. *Let* $(D, \sqsubseteq, \bot)$ *be a* CPO *and* $f : D \to D$ *be a continuous function, then a least fixed point* $\mathsf{fix}(f)$ *exists with*

$$\mathsf{fix}(f) = \bigsqcup_{i \in \mathbb{N}} \big(f^i(\bot)\big)$$

2. *The fixed point operator*

$$\mathsf{fix} : [D \to D] \to D, f \mapsto \bigsqcup_{i \in \mathbb{N}} \big(f^i(\bot)\big)$$

is continuous.

Chapter 3

Word functions

Words over a given alphabet are a common model for data to be processed by computers. Any finite data can be encoded into finite words. There are also extensions like considering infinite words as representations of infinite data, like real numbers.

Natural numbers can be understood as words over a singleton alphabet, say $\{|\}$. We want to extend the notions presented in Section 2.2.

3.1 Basics

From now on we consider a finite alphabet $\Sigma = \{a_1, \ldots, a_r\}$ with $r \geq 2$. Let Σ^* denote the set of finite words over that alphabet. The *empty word* is denoted ε.

3.1.1 Definition
The prefix relation $\leq_p$ *and the* length comparison $\leq_l$ *are defined as follows:*

Let $x, y \in \Sigma^*$*, then*

1. $x \leq_p y \iff \exists z \in \Sigma^*. x \cdot z = y$
2. $x \leq_l y \iff |x| \leq |y|$

The prefix relation is reasonable for infinite words as well. Let Σ^ω be the set of infinite words over the alphabet Σ. Then let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ the set of finite and infinite words.

The concatenation of words is usually defined as a function on finite words, i.e.

$$\cdot : (\Sigma^*)^2 \to \Sigma^*, (v_1 \ldots v_l, w_1 \ldots w_m) \mapsto v_1 \ldots v_l w_1 \ldots w_m$$

The second word may also be infinite. In this case we get

$$\cdot : \Sigma^* \times \Sigma^\omega \to \Sigma^*, (v_1 \ldots v_l, w_1 \ldots) \mapsto v_1 \ldots v_l w_1 \ldots$$

Obviously the concatenation of two infinite words is not reasonable.

3.1.2 Definition
The prefix relation $\leq_p \subseteq (\Sigma^\infty)^2$ is defined by

$$x \leq_p y \iff (x \in \Sigma^\omega \wedge x = y) \vee (x \in \Sigma^* \wedge \exists z \in \Sigma^\infty .\, y = x \cdot z)$$

3.1.3 Remark
$(\Sigma^\infty, \leq_p, \varepsilon)$ *is a* CPO.

3.2 Primitive Recursion on Word Functions

We are going to transfer the notion of primitive recursion to word functions. We have to say which the basic functions are and how the primitive recursion scheme is supposed to look like. Substitution is adopted directly. The following definitions are based on [Wei74].

3.2.1 Definition
The base functions *on words are*

1. *the* successor functions $con_i : \Sigma^* \to \Sigma^*, w \mapsto a_i \cdot w$, $1 \leq i \leq r$,
2. projections $proj_i^k : (\Sigma^*)^k \to \Sigma^*, (w_1, \ldots, w_k) \mapsto w_i$, $1 \leq i \leq k$,
3. constant functions $const_i^k : (\Sigma^*)^k \to \Sigma^*, (w_1, \ldots, w_k) \mapsto a_i$, $k \in \mathbb{N}$, $1 \leq i, \leq r$.

3.2.2 Definition

1. *A function $f : (\Sigma^*)^k \to \Sigma^*$ is defined by* simultaneous substitution *from functions $g : (\Sigma^*)^l \to \Sigma^*$ and $h_1, \ldots, h_l : (\Sigma^*)^k \to \Sigma^*$, if f.a. $\overline{x} \in (\Sigma^*)^k$*

   $$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_l(\overline{x}))$$

   We write $f = Sub(g; h_1, \ldots, h_l)$ or $f = Sub(g; \overline{h})$.

2. *A function $f : (\Sigma^*)^{k+1} \to \Sigma^*$ is defined by* primitive recursion *from functions $g : (\Sigma^*)^k \to \Sigma^*$ and $h_1, \ldots, h_r : (\Sigma^*)^{k+2} \to \Sigma^*$ if*

   $$\begin{aligned} f(\varepsilon, \overline{y}) &= g(\overline{y}) \\ f(a_i x, \overline{y}) &= h_i(x, f(x, \overline{y}), \overline{y}) \end{aligned}$$

   We write $f = Prim(g; h_1, \ldots, h_r)$ or $f = Prim(g; \overline{h})$

3. *Functions $f_1, \ldots, f_l : (\Sigma^*)^{k+1} \to \Sigma^*$ are defined by* simultaneous primitive recursion *from functions $g_1, \ldots, g_j : (\Sigma^*)^k \to \Sigma^*$ and $h_{1,1}, \ldots h_{l,k} : (\Sigma^*)^{n+k+1} \to \Sigma^*$ if*

$$\begin{array}{rcl} f_j(\varepsilon, \overline{w}) & = & g_j(\overline{w}) \\ f_j(a_i v, \overline{w}) & = & h_{j,i}(v, f_1(x, \overline{w}), \ldots, f_l(v, \overline{w}), \overline{w}) \end{array}$$

We write $(f_1, \ldots, f_l) = SimPr(\overline{g}, \overline{h};)$.

3.2.3 Definition (Recursion classes)

1. *$PR(\Sigma)$ is the set of* primitive recursive functions *over Σ^*. It is defined as the smallest set which contains the basic functions and is closed under substitution and primitive recursion.*

2. *$PR_n(\Sigma)$ is the set of primitive recursive functions with recursion depth n, i.e. the set of functions which are defined with at most n nested primitive recursions.*

   Formally $PR_n(\Sigma)$ is defined inductively:

   - *$PR_0(\Sigma)$ is the smallest set that contains the basic functions and is closed under substitution.*
   - *Let $n \in \mathbb{N}$. Then $PR_{n+1}(\Sigma)$ is the smallest set that contains*

     $$PR_n(\Sigma) \cup \{Prim(g; h_1, \ldots, h_r) \mid g, h_1, \ldots, h_r \in PR_n(\Sigma)\}$$

     and is closed under substitution.

3. *The classes $SPR(\Sigma)$ and $SPR_n(\Sigma)$ of simultaneously primitive recursive functions are defined analogously.*

Obviously

$$PR(\Sigma) = \bigcup_{n \in \mathbb{N}} PR_n(\Sigma)$$

3.2.4 Definition

A function $f : (\Sigma^*)^{k+1} \to \Sigma^*$ is defined by bounded primitive recursion *from $g : (\Sigma^*)^k \to \Sigma^*$, $h_1, \ldots, h_r : (\Sigma^*)^{k+2}$ and $j : (\Sigma^*)^{k+1} \to \Sigma^*$ if*

$$f = Prim(g; h_1, \ldots, h_r)$$

and

for all $v_1, \ldots, v_{k+1}$

$$f(v_1, \ldots, v_{k+1}) \leq_l j(v_1, \ldots, v_{k+1})$$

We write $f = BoundPr(j, g; h_1, \ldots, h_r)$

Case distinction can be simulated by primitive recursion, this might be useful in the definition of functions (compare Lemma 2.2.3).

3.2.5 Lemma
For $n \geq 1$ $PR_n(\Sigma)$ is closed under case distinction, i.e. $g, h \in PR_n(\Sigma)$ then $f \in PR_n(\Sigma)$ if

$$f(v, \overline{w}) = \begin{cases} g(w) & \textit{if } v = \varepsilon \\ h(w) & \textit{else} \end{cases}$$

Proof. Consider the recursion scheme

$$\begin{aligned} c(\varepsilon, v, w) &= v \\ c(a_i u, v, w) &= w \end{aligned}$$

Then $f(v, w) = c(v, g(w), h(w))$. ∎

3.2.6 Definition
The Ackermann functions *on words are defined by*

1. $B_0^\Sigma(v, w) = con_1(v)$
2. $B_{n+1}^\Sigma(\varepsilon, w) = w$
   and $B_{n+1}^\Sigma(a_i v, w) = B_n^\Sigma(B_{n+1}^\Sigma(v, w), w)$

B_n is called the n-th Ackermann function over Σ.

3.2.7 Definition
The class $\mathcal{E}_n(\Sigma)$, the n-th Grzegorczyk class *over Σ, is the smallest set of functions which contains the basic functions, the n-th Ackermann function over Σ and is closed under simultaneous substitution and bounded primitive recursion.*

3.2.8 Example
We consider some examples of primitive recursive functions. Some of them might be useful.

To ensure the position in the hierarchy we give recursion schemes and bounding functions if not obvious.

1. *first* : $\Sigma^* \to \Sigma^*, \begin{cases} \varepsilon \mapsto \varepsilon \\ a_i v \mapsto a_i \end{cases} \in PR_1(\Sigma) \cap \mathcal{E}_0(\Sigma)$

2. *rest* : $\Sigma^* \to \Sigma^*, \begin{cases} \varepsilon \mapsto \varepsilon \\ a_i v \mapsto v \end{cases} \in PR_1(\Sigma) \cap \mathcal{E}_0(\Sigma)$

3. *cut* : $(\Sigma^*)^2 \to \Sigma^*, (v_1 \dots v_l, w_1 \dots w_r) \mapsto \begin{cases} \varepsilon & \text{if } l \geq r \\ w_{l+1} \dots w_r & \text{if } l < r \end{cases} \in PR_2(\Sigma) \cap \mathcal{E}_0(\Sigma)$

$$\begin{aligned} cut(\varepsilon, w) &= w \\ cut(av, w) &= rest(cut(v, w)) \end{aligned}$$

and $cut(v, w) \leq_l w \leq_l B_0^\Sigma(w, w)$

4. $rev : \Sigma^* \to \Sigma^*, v_1 \dots v_r \mapsto v_r \dots v_1 \in PR_2(\Sigma) \cap \mathcal{E}_0(\Sigma)$
   Consider

$$\begin{aligned} r_i(\varepsilon) &= a_i \\ r_i(a_j v) &= a_j r_i(v) \end{aligned}$$

   and

$$\begin{aligned} rev(\varepsilon) &= \varepsilon \\ rev(a_i v) &= r_i(rev(v)) \end{aligned}$$

5. $rcut : (\Sigma^*)^2 \to \Sigma^*, (v_1, \dots, v_l, w_1, \dots, w_r) \mapsto \begin{cases} \varepsilon & \text{if } l \geq r \\ w_1 \dots w_{r-l} & \text{if } l < r \end{cases} \in PR_2(\Sigma) \cap \mathcal{E}_0(\Sigma)$
   $rcut(v, w) = rev(cut(v, rev))$

6. $conc : (\Sigma^*)^2 \to \Sigma^*, (v_1 \dots v_l, w_1 \dots w_r) \mapsto v_1 \dots v_l w_1 \dots w_r \in PR_1(\Sigma) \cap \mathcal{E}_1(\Sigma)$

$$\begin{aligned} conc(\varepsilon, w) &= w \\ conc(av, w) &= a \cdot conc(v, w) \end{aligned}$$

   and $|conc(v, w)| = |v| + |w| \leq |B_1^\Sigma(w)|$

7. $shuffle_\varepsilon : (\Sigma^*)^2 \to \Sigma^*, (v_1 \dots v_l, w_1 \dots w_r) \mapsto v_1 w_1 \dots v_k w_k$ with $k = \min(l, r)$
   Consider the following simultaneously recursive functions $s, t : (\Sigma^*)^2 \to \Sigma^*$

$$\begin{aligned} t(\varepsilon, w) &= w \\ s(\varepsilon, w) &= \varepsilon \\ \\ t(av, w) &= rest(t(v, w)) \\ s(av, w) &= a \cdot first(t(v, w)) \cdot s(v, w) \\ &= a \cdot (conc(first(t(v, w)), s(v, w))) \end{aligned}$$

   Then $s, t \in SPR_2(\Sigma)$.

$$shuffle_\varepsilon(v, w) = s(rcut(cut(w, v), v), rev(rcut(cut(v, w), w)))$$

   Then $shuffle_\varepsilon \in PR_2(\Sigma) \cap \mathcal{E}_1(\Sigma)$

8. $\mathit{shuffle} : (\Sigma^*)^2 \to \Sigma^*, (v_1 \dots v_l, w_1 \dots w_r) \mapsto \begin{cases} v_1 w_1 \dots v_l w_l w_{l+1} \dots w_r & \text{if } l < r \\ v_1 w_1 \dots v_l w_l & \text{if } l = r \\ v_1 w_1 \dots v_r w_r v_{r+1} \dots v_l & \text{if } l > r \end{cases}$

   $\mathit{shuffle}(v, w) = \mathit{shuffle}_\varepsilon(v, w) \cdot \mathit{cut}(v, w) \cdot \mathit{cut}(w, v) \in \mathit{SPR}_2(\Sigma) \cap \mathcal{E}_1(\Sigma)$

   Note, that the recursion scheme for s and t in this example might not be recognized as a simultaneous recursion scheme on first sight, since s relies on former values of t but not vice versa. We could define s and t by an ordinary recursion scheme, as well. But that would make as climb higher within the hierarchy and we had $s \in \mathit{PR}_3(\Sigma)$.

9. $\mathit{half} : \Sigma^* \to \Sigma^*, v_1 v_2 \dots v_l \mapsto v_2 v_4 \dots v_{l \,\mathrm{DIV}\, 2} \in \mathit{SPR}_1(\Sigma)$

$$\begin{aligned} \mathit{half}(\varepsilon) &= \varepsilon \\ \mathit{half}(av) &= \mathit{half}'(v) \\ \mathit{half}'(\varepsilon) &= \varepsilon \\ \mathit{half}'(av) &= a \cdot \mathit{half}(v) \end{aligned}$$

   Obviously $\mathit{half}' : \Sigma^* \to \Sigma^*, v_1 v_2 \dots v_l \mapsto v_1 v_3 \dots v_{(l+1) \,\mathrm{DIV}\, 2} \in \mathit{SPR}_1(\Sigma)$

We cite important results from [Wei74]:

3.2.9 Lemma
Let f be primitive recursive. Then

$$f \in \mathcal{E}_n(\Sigma) \iff \exists f_l : \mathbb{N} \to \mathbb{N} \in \mathcal{E}_n. \left[\forall \overline{v} \in (\Sigma^*)^k. \, |f(v_1, \dots, v_k)| \le f_l(|v_1|, \dots, |v_k|) \right]$$

3.2.10 Theorem (Hierarchy Theorem)

1. $\mathit{PR}_1(\Sigma) \subsetneqq \mathit{SPR}_1(\Sigma) \subsetneqq \mathcal{E}_1(\Sigma)$
2. $\mathcal{E}_n(\Sigma) \subsetneqq \mathit{PR}_n(\Sigma) = \mathit{SPR}_n(\Sigma) = \mathcal{E}_{n+1}(\Sigma)$ *for all* $n \ge 2$.

Sometimes we need auxiliary functions which use additional letters. Those can be simulated.

3.2.11 Lemma
Let $\Sigma = \{a_1, \dots, a_r\}$ be an alphabet and let $\Sigma' = \{a_1, \dots, a_r, a_{r+1}\}$ be an alphabet with an additional symbol $a_{r+1} \notin \Sigma$.

Let $f' : (\Sigma')^* \to (\Sigma')^* \in \mathit{PR}_n(\Sigma')$ be a primitive recursive word function over Σ'.

Let $f'|_{\Sigma^*}$ denote the restriction of f' to Σ^*, i.e. $f'|_{\Sigma^*} : \Sigma^* \to (\Sigma')^*$ with $f'|_{\Sigma^*}(v) = f'(v)$ for all $v \in \Sigma^*$.

If $f'(\Sigma^*) = \{f(v) \mid v \in \Sigma^*\} \subseteq \Sigma^*$, then $f'|_{\Sigma^*} \in \mathit{SPR}_n(\Sigma)$.

Proof.

Primitive recursion over Σ' can be simulated by simultaneous primitive recursion over Σ.

We encode words from Σ' in Σ by

$$in_1 : \Sigma' \to \Sigma, a_i \mapsto \begin{cases} a_1 a_i & \text{if } 1 \le i \le r \\ a_2 a_1 & \text{if } i = r+1 \end{cases}$$

and

$in : (\Sigma')^* \to \Sigma^*, v_1 \dots v_l \mapsto in_1(v_1) \dots in_1(v_l)$

Note, that $in|_{\Sigma^*} \in PR_1(\Sigma)$

The inverse function *out* is defined straightforwardly.

We construct a function $f \in SPR_n(\Sigma)$ for every function $f' \in PR_n(\Sigma')$ with

$$f' = out \circ f \circ in$$

Then the claim follows from $f'(\Sigma^*) \subseteq \Sigma^*$. We can choose $out = half$ in this case.

We use structural induction:

- The basic functions can be simulated easily. There is nothing to do for the projection. For every constant function $f' : v \mapsto w$ choose $f : v \mapsto in(w)$ and for the successor functions $f' = con_i$ choose $f : v \mapsto in_1(a_i) \cdot v$.
- $f' = Sub(g'; \overline{h}')$. Choose $f = Sub(g; \overline{h})$
- $f' = Prim(g'; \overline{h}')$, i.e.

$$\begin{aligned} f'(\varepsilon, \overline{w}) &= g'(w) \\ f'(a_i v, \overline{w}) &= h'_i(v, f'(v), \overline{w}) \end{aligned}$$

  Choose

$$\begin{aligned} f(\varepsilon, \overline{w}) &= g(\overline{w}) \\ f(a_1 v, \overline{w}) &= f_1(v, \overline{w}) \\ f(a_2 v, \overline{w}) &= f_2(v, \overline{w}) \\ \\ f_1(a_i v, \overline{w}) &= h_i(v, f(v), \overline{w}) \\ f_2(a_1 v, \overline{w}) &= h_{r+1}(v, f(v), \overline{w}) \end{aligned}$$

  This scheme shows only the occurring cases. The gaps might be filled arbitrarily to obtain a complete scheme.

■

3.3 Monotonic functions

Computing on real numbers requires to represent them as infinite objects (usually words). We want functions on those infinite words to have the *finiteness property*, i.e. each finite prefix of the output depends on only a finite prefix of the input. As a direct consequence an output cannot be revoked by reading more input, but the output can only be extended.

To approximate these infinite computations by finite functions they have to respect this property. This is done by *monotonic functions*.

If we additionally want the infinite functions to be total, i.e. every infinite input produces an infinite output, we need their finite approximations to extend every output by extending the input. These functions are called *fully monotonic*.

3.3.1 Definition
A word function $f : (\Sigma^*)^k \to \Sigma^*$ is called

- monotonic *if for all $w_1, \dots, w_k, w'_1, \dots, w'_k \in \Sigma^*$ holds*

$$f(w_1, \dots, w_k) \leq_p f(w_1 w'_1, \dots, w_k w'_k)$$

- fully monotonic *if f is monotonic and for any natural number l and all words $v_1, \dots, v_k \in \Sigma^*$ words $v'_1, \dots, v'_k \in \Sigma^*$ exist with*

$$|f(v_1 v'_1, \dots, v_k v_{k'})| \geq l$$

As we will see, monotonic function are an import class of functions. An easy way to construct monotonic functions is to restrict the scheme of primitive recursion. This leads to an new principle, called ε-primitive recursion.

3.3.2 Definition

1. *A function $f : (\Sigma^*)^{k+1} \to \Sigma^*$ is defined by* ε-primitive recursion *from $h_1, \dots, h_r : (\Sigma^*)^{k+2} \to \Sigma^*$ if*

$$\begin{aligned} f(\varepsilon, \overline{w}) &= \varepsilon \\ f(a_i v, \overline{w}) &= h_i(v, f(v, \overline{w}), \overline{w}) \qquad i \in \{1, \dots, r\} \end{aligned}$$

   We write $f = Prim_\varepsilon(h_1, \dots, h_l) = Prim_\varepsilon(\overline{h})$.

2. *The classes ε-$PR_n(\Sigma)$ are defined analogously to $PR_n(\Sigma)$. ε-$PR(\Sigma) = \bigcup_{n \in \mathbb{N}} \varepsilon$-$PR_n(\Sigma)$ is the class of ε-primitive recursive functions.*

3. *Simultaneous ε-primitive recursion and the respective classes ε-$SPR_n(\Sigma)$ and ε-$SPR(\Sigma)$ are defined analogously*

There is a notion of ε_0-recursion, which is something completely different. We will use the notion of ε-recursion assuming that no confusion arrises.

Note that being ε-primitive recursive does not mean, that a function maps ε to ε. We did only change the recursion scheme but not the scheme of simultaneous substitution.

3.3.3 Lemma
Every ε-recursive function is monotonic.

Proof. By induction over the construction of ε-primitive recursive functions:

- All basic function are monotonic.
- $f = Sub(g; h_1, \ldots, h_l)$.
  W.l.o.g. let $l = 1$, i.e. $f = g \circ h$. By induction hypothesis g and h are monotonic. We have
  $$h(v) \leq_p h(vw)$$
  and
  $$g(h(v)) \leq_p g(h(vw))$$
- $f^{(k)} = Prim_\varepsilon(h_1^{(k+1)}, \ldots, h_r^{(k+1)})$
  By induction hypothesis $h_1, \ldots, h_r$ are monotonic.
  W.o.l.g. let $k = 2$.
  We will show $f(v, v_2) \leq_p f(vw, v_2w_2)$
  Induction over the length $|v|$ of the recursion parameter
  - $v = \varepsilon$.
    $$f(\varepsilon, v_2) = \varepsilon \leq_p f(w, v_2w_2)$$
  - $v = a_iv'$.
    $$f(a_iv', v_2) = h_i(v', f(v', v_2), v_2) \overset{(+)}{\leq_p} h_i(v'w, f(v'w, v_2w_2), v_2w_2) = f(vw, v_2w_2)$$
    (+) uses the monotonicity of h_i and the induction hypothesis.

■

3.3.4 Corollary
Every simultaneously ε-recursive function is monotonic.

The proof is obviously the same as in the case before.

3.3.5 Example
Some monotonic functions and their (ε)-recursion schemes:

1. The function $\mathit{shuffle}_\varepsilon$ (Example 3.2.8) is monotonic, but the function *shuffle* is not. The functions *half* and half' are monotonic, as well.
2. $\mathit{cont}(w_1 \dots w_l) = w_1 w_1 w_2 w_1 w_2 w_3 \dots w_1 \dots w_l$

   We define an auxiliary function

   $$\mathit{cont}'(w_1 \dots w_l) = a_1 w_1 a_1 w_1 a_2 w_2 a_1 w_1 a_2 w_2 a_2 w_3 \dots a_1 w_1 a_2 w_2 \dots a_2 w_l$$

   via the recursion scheme

   $$\begin{array}{rcl} \mathit{cont}'(\varepsilon) & = & \varepsilon \\ \mathit{cont}'(a_i w) & = & a_1 a_i \cdot \mathit{ins}_i(\mathit{cont}'(w)) \end{array}$$

   with

   $$\begin{array}{rcll} \mathit{ins}_i(\varepsilon) & = & \varepsilon & \\ \mathit{ins}_i(a_1 w) & = & a_1 a_i a_2 \mathit{ins}'_i(w) & \\ & & & \\ \mathit{ins}'_i(\varepsilon) & = & \varepsilon & \\ \mathit{ins}'_i(a_j w) & = & a_j \cdot \mathit{ins}'_i(w) & \qquad j \neq 1 \\ \mathit{ins}'_i(a w) & = & a \cdot \mathit{ins}_i(w) & \end{array}$$

   We get

   $$\mathit{cont} = \mathit{half} \circ \mathit{cont}'$$

   and

   $$\mathit{cont} \in \varepsilon\text{-}\mathit{SPR}_2(\Sigma) \cap \mathcal{E}_2(\Sigma)$$

3. The next example is similar:

   $$\mathit{rcont} : \Sigma^* \to \Sigma^* : w_1 \dots w_l \mapsto w_1 w_2 w_1 w_3 w_2 w_1 \dots w_l \dots w_1 \in PR_2(\Sigma)$$

   rcont is monotonic.

   Look at the recursion scheme:

   $$\begin{array}{rcl} \mathit{rcont}(\varepsilon) & = & \varepsilon \\ \mathit{rcont}(av) & = & \mathit{rcont}(v) \cdot \mathit{rev}(av) \\ & = & \mathit{conc}(\mathit{rcont}(v), \mathit{rev}(av)) \end{array}$$

This scheme shows $rcont \in PR_3(\Sigma)$ since $rev \in PR_2(\Sigma)$ and $conc \in PR_1(\Sigma)$

This scheme uses the functions *rev* and *conc* which are not monotonic and thus not ε-recursive.

But we can construct an ε-recursive version of *rcont* analogously to the previous example *cont*.

We define

$$\begin{aligned} rins_i(\varepsilon) &= \varepsilon \\ rins_i(a_j v) &= a_j \cdot rins'_i(v) \\ \\ rins'_i(\varepsilon) &= \varepsilon \\ rins'_i(a_1 v) &= a_2 a_j a_1 \cdot rins_i(v) \\ rins'_i(a_j v) &= a_j \cdot rins_i(v) \qquad j \neq 1 \end{aligned}$$

and

$$\begin{aligned} rcont'(\varepsilon) &= \varepsilon \\ rcont'(a_i v) &= a_i a_1 \cdot rins_i(rcont'(v)) \end{aligned}$$

and

$$rcont = half' \circ rcont'$$

This shows $rins_i \in \varepsilon\text{-}SPR_1(\Sigma)$ and

$$rcont \in \varepsilon\text{-}SPR_2(\Sigma) \subseteq SPR_2(\Sigma) = PR_2(\Sigma)$$

3.3.6 Definition
A function $\pi : (\Sigma^*)^k \to \Sigma^*$ is called a tuple coding, *if functions $\pi_1, \ldots, \pi_k : \Sigma^* \to \Sigma^*$ exist such that for all $1 \leq i \leq k$ and all $v_1, \ldots, v_k \in \Sigma^*$*

$$\pi_i(\pi(v_1, \ldots, v_k)) = v_i$$

The functions $\pi_1, \ldots, \pi_k$ are called the projections of *π.*

3.3.7 Lemma (Non-monotonicity of codings)
For $k \geq 2$ let $\pi : (\Sigma^*)^k \to k$ be a tuple coding and $\pi_1, \ldots, \pi_k : \Sigma^* \to \Sigma^*$ the respective projections, i.e. $\pi_i(\pi(w_1, \ldots, w_k)) = w_i$ for all $i \in \{1, \ldots, k\}$.

Then at least one of the functions $\pi, \pi_1, \ldots, \pi_k$ is not monotonic.

Proof. W.l.o.g. let $k = 2$. Assume π, π_1 and π_2 are monotonic. Then

$$\begin{array}{rcl}\pi(\varepsilon, a) & \leq_p & \pi(b, a)\\ \pi(b, \varepsilon) & \leq_p & \pi(b, a)\end{array}$$

where $a, b \in \Sigma$ are symbols.

Then either $\pi(\varepsilon, a) \leq_p \pi(b, \varepsilon)$ or $\pi(b, \varepsilon) \leq_p \pi(\varepsilon, a)$. In case $\pi(\varepsilon, a) \leq_p \pi(b, \varepsilon)$ we have

$$a = \pi_2(\pi(\varepsilon, a)) \leq_p \pi_2(\pi(b, \varepsilon)) = \varepsilon$$

which is an contradiction to $a \in \Sigma$. ■

3.3.8 Definition
A function $\pi : (\Sigma^*)^k \to \Sigma^*$ is called a (tuple) pre-coding, *if functions $\pi_1, \ldots, \pi_k : \Sigma^* \to \Sigma^*$ exist such that for all $1 \leq i \leq k$ and all $v_1, \ldots, v_k \in \Sigma^*$*

$$\pi_i(\pi(v_1, \ldots, v_k)) \leq_p v_i$$

and

$$|\pi_i(\pi(v_1, \ldots, v_k))| \geq \min\{|v_1|, \ldots, |v_k|\}$$

The functions $\pi_1, \ldots, \pi_k$ are called the projections of π.

3.3.9 Example

1. Every coding is a pre-coding.
2. *shuffle*$_\varepsilon$ is a monotonic pre-coding with projections *half* and *half*$'$.

3.3.10 Definition
Let $f : (\Sigma^*)^k \to \Sigma^*$ be a fully monotonic word function. A monotonic function $f_{slow} : (\Sigma^*)^k \to \Sigma^*$ is called a slow version *of f if*

- *$f_{slow}(\overline{v}) \leq_p f(\overline{v})$ for all $\overline{v} \in (\Sigma^*)^k$ and*
- *for all $\overline{u} \in (\Sigma^*)^k$ exists $\overline{v} \in (\Sigma^*)^k$ with $u_i \leq_p v_i$ and $f(\overline{u}) \leq_p f_{slow}(\overline{v})$*

3.3.11 Lemma
The slow versions of fully monotonic word functions are fully monotonic.

Proof. Follows directly from the definitions. ■

3.3.12 Example
Let $\pi : (\Sigma^*)^k \to \Sigma^*$ be a pre-coding with projections $\pi_1, \ldots, \pi_k$. Then $\pi_i \circ \pi$ is a slow version of the projection $proj_i^k$ for all $1 \leq i \leq k$.

To show $SPR_n \subseteq PR_n$ in [Wei74] a coding function is necessary. Since we do not have a coding function in ε-$SPR_n(\Sigma)$ we cannot adopt that proof.

3.4 Uniform and truly uniform recursion

In primitive recursion we distinguish between the recursion parameter and (a couple of) side parameters. Side parameters are useful in finite computations only.

We consider the following example of a primitive recursive and monotonic function which has a side parameter.

Let $f \in \varepsilon$-$PR_1(\Sigma)$ with $f = Prim_\varepsilon(\overline{h})$. The ε-recursion enforces $f(\varepsilon, v) = \varepsilon$ for all $v \in \Sigma^*$, that does not contradict the monotonicity.

Further consider the value $f(a_i, w)$, we obtain:

$$f(a_i, w) = h_i(\varepsilon, \varepsilon, w)$$

The purpose of the functions $h_i \in \varepsilon$-$PR_0(\Sigma)$ is to append a word, say u_i, to one of its parameters. Appending it to the recursion parameter or the preceding value would ignore the side parameters. Appending it to one of the side parameters can be understood as changing the role of this side parameter and the recursion parameter, the remainder of the former recursion parameter is ignored.

If we want to use all parameters in an infinite computation, they must have equal rights in the recursion scheme.

To be able to consider infinite computation we introduce *uniform recursion* in which we have only recursion parameters but no side parameters.

We will see, that the difference remains in recursion depth one, but will vanish in higher levels.

We give a definition for two parameters. The definition would hardly be readable with more parameters.

3.4.1 Definition
A function $f : (\Sigma^*)^2 \to \Sigma^*$ *is defined by* uniform recursion *from* $w_0 \in \Sigma^*$, $g'_1, \dots g'_r, g''_1, \dots, g''_r : (\Sigma^*)^2 \to \Sigma^*$ *and* $h_{1,1}, \dots, h_{1,r}, \dots h_{r,1}, \dots, h_{r,r} : (\Sigma^*)^3 \to \Sigma^*$ *if*

$$\begin{aligned} f(\varepsilon, \varepsilon) &= w_0 && (3.1) \\ f(a_i u, \varepsilon) &= g'_i(u, f(u, \varepsilon)) && (3.2) \\ f(\varepsilon, a_j v) &= g''_j(v, f(\varepsilon, v)) && (3.3) \\ f(a_i u, a_j v) &= h_{i,j}(u, v, f(u, v)) \qquad i, j \in \{0, \dots, r\} && (3.4) \end{aligned}$$

We write

$$f = \mathit{Unif}(w_0; g'_1, \ldots, g'_r, g''_1, \ldots, g''_r; h_{11}, \ldots, h_{1r}, \ldots, h_{r1}, \ldots, h_{rr})$$

or

$$f = \mathit{Unif}(w_0; \overline{g'}, \overline{g''}; \overline{h})$$

Let $UR(\Sigma)$ denote the set of uniformly recursive functions, i.e. the smallest set which contains the basic functions and is closed under composition and uniform recursion, and $UR_n(\Sigma)$ the subset of functions which use at most n nested uniform recursions.

Simultaneously uniform recursion and its classes $SUR_n(\Sigma)$ and $SUR(\Sigma)$ are defined analogously.

We will use functions with only two parameters whenever possible. Proofs and definitions are essentially the same with higher arities, but with a more confusing notation.

In case of one parameter the definition of primitive and uniform recursion coincide.

Analogously to 2.2.3 and 3.2.5 we can define case distinctions.

3.4.2 Lemma
$UR_n(\Sigma)$ is closed under case distinction.

3.4.3 Example
Some examples of uniformly recursive functions:

1. $\mathit{shuffle} \in UR_1(\Sigma)$

$$\begin{aligned} \mathit{shuffle}(\varepsilon, \varepsilon) &= \varepsilon \\ \mathit{shuffle}(a_i u, \varepsilon) &= a_i \cdot \mathit{shuffle}(u, \varepsilon) \\ \mathit{shuffle}(\varepsilon, a_j v) &= a_j \cdot \mathit{shuffle}(\varepsilon, v) \\ \mathit{shuffle}(a_i u, a_j v) &= a_i a_j \cdot \mathit{shuffle}(u, v) \end{aligned}$$

2. Let $1 \leq i \leq r$ and $\mathit{insert}_i : (\Sigma^*)^2 \to \Sigma^*$

$$\begin{aligned} \mathit{insert}_i(\varepsilon, v) &= a_i v \qquad (3.5) \\ \mathit{insert}_i(au, \varepsilon) &= a_i \\ \mathit{insert}_i(au, bv) &= b \cdot \mathit{insert}_i(u, v) \end{aligned}$$

Line 3.5 combine the lines $\mathit{insert}_i(\varepsilon, bv) = a_i bv$ and $\mathit{insert}_i(\varepsilon, \varepsilon) = a_i$ in the recursion scheme.

Then

$$insert_i(u_1 \dots u_l, v_1 \dots v_m) = \begin{cases} v_1 \dots v_l a_i v_{l+1} \dots v_m & \text{if } l \leq m \\ v_1 \dots v_m a_i & \text{if } m < l \end{cases}$$

and $insert_i \in UR_1(\Sigma)$

3. $cut : (\Sigma^*)^2 \to \Sigma^*, (u_1 \dots u_l, v_1 \dots v_m) \mapsto v_{m \dot{-} l} \dots v_m \in UR_1(\Sigma)$

$$\begin{aligned} cut(\varepsilon, v) &= v \\ cut(au, \varepsilon) &= \varepsilon \\ cut(au, bv) &= cut(u, v) \end{aligned}$$

4. $conc : (\Sigma^*)^2 \to \Sigma^*, (v, w) \mapsto v \cdot w \in UR_2(\Sigma)$

$$\begin{aligned} conc(u, \varepsilon) &= u \\ conc(\varepsilon, bv) &= bv \\ conc(a_i u, a_j v) &= a_i \cdot insert_j(u, conc(u, v)) \end{aligned}$$

5. $cont : \Sigma^* \to \Sigma^* \in UR_3(\Sigma)$

We compare uniform and primitive recursion.

3.4.4 Lemma

1. *$UR_1(\Sigma) \subseteq \mathcal{E}_1$*
2. *For $n \geq 2$ holds $UR_n(\Sigma) \subseteq \mathcal{E}_{n+1}$*

Proof. W.l.o.g. we consider functions of arity $k = 2$. The proof is essentially the same for higher arities. For arity 1 it is obvious. Notice that we need higher arities in the induction hypothesis.

To simulate the equal rights of the parameters of a uniformly recursive function we have to merge them into a single parameter. This can be done by shuffling. The recently defined function *shuffle* is not sufficient, it will cause problems on sorting out the parameters again. We define:

$$Shuffle^{(2)} : (\Sigma^*)^2 \to \Sigma^*, (u_1 \dots u_l, v_1 \dots v_m) \mapsto$$
$$\mapsto \begin{cases} a_1 u_1 a_1 v_1 a_1 u_2 a_1 v_2 \dots a_1 u_l a_1 v_l & \text{if } l = m \\ a_1 u_1 a_1 v_1 a_1 u_2 a_1 v_2 \dots a_1 u_l a_1 v_l a_2 a_2 v_{l+1} \dots v_m & \text{if } l < m \\ a_1 u_1 a_1 v_1 a_1 u_2 a_1 v_2 \dots a_1 u_m a1 v_m a_2 a_1 u_{m+1} \dots u_l & \text{if } l > m \end{cases} \tag{3.6}$$

Since

$$Shuffle^{(2)} = inj_1(shuffle_\varepsilon(u, v) \cdot ifcon(a_2 a_2, cut(u, v)) \cdot ifcon(a_2 a_1, cut(v, u)))$$

with

$$inj_i : \Sigma^* \to \Sigma^*, v_1 \dots v_l \to a_i v_1 \dots a_i v_l$$

and

$$ifcon : (\Sigma^*)^2 \to \Sigma^*, \begin{cases} (u, \varepsilon) & \mapsto \varepsilon \\ (u, av) & \mapsto u \cdot a \cdot v \end{cases}$$

we obtain

$$Shuffle^{(2)} \in \mathcal{E}_1(\Sigma) \cap SPR_2(\Sigma) \tag{3.7}$$

We can define $Shuffle^{(k)} : (\Sigma^*)^k \to \Sigma^*$ for all $n \geq 1$ by

$$\begin{array}{rcl} Shuffle^{(1)}(v) & = & v \\ Shuffle^{(k+1)}(v_1, \dots, v_k, v_{k+1}) & = & Shuffle^{(2)}(Shuffle^{(n)}(v_1, \dots, v_k), v_{k+1}) \end{array}$$

The inverse functions

$$Deshuffle_{k,i} : (\Sigma^*)^2 \to \Sigma^* : Deshuffle_{2,i}(Shuffle^{(k)}(v_1, \dots, v_k)) = v_i \qquad \text{for } 1 \leq i \leq k, k \geq 1$$

are defined straightforwardly by simultaneous recursion and thus

$$Deshuffle_{k,i} \in SPR_1(\Sigma) \tag{3.8}$$

Let $f \in UR_n(\Sigma)$, we construct a function f_s with

$$f(\overline{v}) = f_s(Shuffle^{(k)}(\overline{v})$$

by structural induction

- $f = Unif(y_0; \overline{g}; \overline{h})$.

  1. $n = 1$
     We show, that function $f_s \in SPR_1(\Sigma) \subseteq \mathcal{E}_1(\Sigma)$ exists, with $f = f_s \circ Shuffle^{(2)}$
     We consider the recursion scheme of f:

$$\begin{aligned} f(\varepsilon,\varepsilon) &= y_0 \\ f(a_i u,\varepsilon) &= g'_i(u, f(u,\varepsilon)) \\ f(\varepsilon, a_j v) &= g''_j(v, f(\varepsilon, v)) \\ f(a_i u, a_j v) &= h_{i,j}(u, v, f(u,v)) \end{aligned}$$

With $\overline{g}, \overline{h} \in UR_0(\Sigma) = PR_0(\Sigma)$
The definition of the simultaneous recursion scheme of f_s is straightforward:

$$\begin{aligned} f_s(\varepsilon) &= q_0 && (3.9) \\ f_s(a_1 a_i a_2 a_j v) &= h^s_{ij}(v, f'_s, f''_s, f_s(v)) && (3.10) \\ f_s(a_2 a_2 a_1 a_i v) &= (g'_i)^s(v, f_s(v)) && (3.11) \\ f_s(a_2 a_1 a_i v) &= (g'_i)^s(v, f_s(v)) && (3.12) \end{aligned}$$

where e.g. line 3.10 is shorthand for the simultaneous recursion

$$\begin{aligned} f_s(a_1 v) &= f'(v) && (3.13) \\ f'(a_i v) &= f'_i(v) && (3.14) \\ f'_i(a_1 v) &= f''_i(v) && (3.15) \\ f''_i(a_j v) &= h_{i,j}(v, f_s(v)) && (3.16) \end{aligned}$$

and $h_{ij,s} \in PR_0(\Sigma)$ is defined by

$$h_{ij,s}(u,v,w,x) = \begin{cases} w_{ij} \cdot x & \text{if } h_{ij}(v,w,x) = w_{ij} \cdot x \\ w_{ij} \cdot v & \text{if } h_{ij}(v,w,x) = w_{ij} \cdot v \\ w_{ij} \cdot w & \text{if } h_{ij}(v,w,x) = w_{ij} \cdot w \end{cases}$$

f'_s and f''_s are simultaneously defined such that $f'_s(v) = \mathit{Deshuffle}_{2,1}(v)$ and $f''_s(v) = \mathit{Deshuffle}_{2,2}(v)$.
$g'_{i,s}$ and $g''_{i,s}$ are defined analogously from g'_i and g''_i respectively.

2. $n \geq 2$
We define a function f_s analogously to the previous case with

$$f(v_1, \ldots, v_k) = f_s\left(\mathit{Shuffle}^{(k)}(v_1, \ldots, v_l)\right)$$

In case $k = 2$ this would look like

$$\begin{aligned} f_s(\varepsilon) &= y_0 \\ f_s(a_1 a_i a_1 a_j v) &= h_{ij,s}(\mathit{Shuffle}^{(2)}(v, f_s(v))) \\ f_s(a_2 a_1 a_i v) &= g'_{i,s}(\mathit{Deshuffle}_{2,1} v) \\ f_s(a_2 a_2 a_i v) &= g''_{i,s}(v) \end{aligned}$$

Then we have

  - $n = 2$, then $\overline{g}, \overline{h} \in UR_1(\Sigma)$ and $\overline{g}_s, \overline{h}_s \in \mathcal{E}_1(\Sigma)$. Thus $f_s \in \mathcal{E}_3(\Sigma)$
  - $n > 2$, then $\overline{g}, \overline{h} \in UR_{n-1}(\Sigma)$ and $\overline{g}_s, \overline{h}_s \in \mathcal{E}_n(\Sigma)$. Thus $f_s \in \mathcal{E}_{n+1}(\Sigma)$

- $f = Sub(g; h_1, \ldots, h_k)$ with $g, h_1, \ldots, h_k \in UR_1(\Sigma) \subseteq \mathcal{E}_1(\Sigma)$, then:

$$f_s(\overline{v}) = g_s\left(\mathit{Shuffle}^{(k)}(h_{1,s}(\overline{v}), \ldots, h_{k,s}(\overline{v}))\right)$$

  $\mathcal{E}_n(\Sigma)$ is closed under substitution.

This construction shows

1. $f \in UR_1(\Sigma) \Rightarrow f_s \in \mathcal{E}_1(\Sigma)$ and
2. $f \in UR_n(\Sigma) \Rightarrow f_s \in \mathcal{E}_{n+1}(\Sigma)$

And with $f = f_s \circ \mathit{Shuffle}^{(k)}$ the claim of the lemma follows.

■

3.4.5 Corollary

1. *For all $n \geq 2$ holds $UR_n(\Sigma) \subseteq PR_n(\Sigma)$*
2. *$UR(\Sigma) \subseteq PR(\Sigma)$*

The result in the other direction is a bit weaker.

3.4.6 Lemma
$PR_n(\Sigma) \subseteq UR_{n+1}(\Sigma)$

Proof. By structural induction, let $f \in PR_n(\Sigma)$.

- There is nothing to show for the basic functions.
- $f = Sub(g; \overline{h})$. Obviously, $UR_{n+1}(\Sigma)$ is closed under substitution for all $n \in \mathbb{N}$.
- $f = Prim(g; h_1, \ldots, h_r)$, i.e. $f \in PR_n(\Sigma)$ for $n > 0$ and

$$g, h_1, \ldots, h_r \in PR_{n-1}(\Sigma) \overset{\text{I.H}}{\subseteq} UR_n(\Sigma)$$

  W.l.o.g let $f : (\Sigma^*)^2 \to \Sigma^*$. Remember: $conc \in UR_2(\Sigma)$ and $cut \in UR_1(\Sigma)$, see Example 3.4.3 (page 40).

  We construct a function $f_c \in UR_{n+1}(\Sigma)$ with

$$f(v, w) = f_c(v, vw) = f_c(v, conc(v, w))$$

  Then $f \in UR_{n+1}(\Sigma)$, since $n + 1 = \max(2, n + 1)$ for all $n \geq 1$.

  Let

$$\begin{aligned} f(\varepsilon, w) &= g(w) \\ f(a_i v, w) &= h_i(v, w, f(v, w)) \end{aligned}$$

Then

$$\begin{aligned} f_c(\varepsilon, w) &= g(w) \\ f_c(a_i v, a_j w) &= h_i(v, cut(v, w), f_c(v, w)) \end{aligned}$$

With $g, h_1, \ldots, h_r \in UR_n(\Sigma)$ we obtain $f_c \in UR_{n+1}(\Sigma)$.

With this construction $f(v, w) = f_c(v, vw)$ follows easily by induction over $|v|$

- $|v| = 0$.
  Then
  $$f(\varepsilon, w) = g(w) = f_c(\varepsilon, \varepsilon \cdot w)$$
- $|v| > 0$, i.e. $v = a_i v'$.
  Then
  $$f(a_i v', w) = h_i(v', w, f(v', w)) \overset{\star}{=} h_i(v', cut(v', v'w), f_c(v', v'w)) = f_c(a_i v', a_i v'w)$$
  $\star$ uses the induction hypothesis.

■

3.4.7 Corollary
$PR(\Sigma) \subseteq UR(\Sigma)$

Together with Corollary 3.4.5-2 we obtain:

3.4.8 Corollary
$UR(\Sigma) = PR(\Sigma)$

3.4.9 Example

- $UR_1(\Sigma) \not\subseteq PR_1$ since *shuffle* $\in UR_1(\Sigma)$ but obviously *shuffle* $\in PR_2 \setminus PR_1$.
- $PR_1 \not\subseteq UR_1(\Sigma)$ since *conc* $\in PR_1$ but *conc* $\in UR_2(\Sigma) \setminus UR_1(\Sigma)$

Analogously to the ε-primitive recursion the ε-uniform recursion can be defined to obtain monotonic functions.

3.4.10 Definition

1. *A function* $f : \Sigma^{*2} \to \Sigma^*$ *is defined by* ε-uniform recursion *from* $g_0^{(0)}, g_1^{(1)}, g_2^{(1)}, h_{1,1}, \dots, h_{k,k}$ *if*

$$\begin{aligned} f(\varepsilon,\varepsilon) &= \varepsilon && (3.17)\\ f(\varepsilon,v) &= \varepsilon && (3.18)\\ f(u,\varepsilon) &= \varepsilon && (3.19)\\ f(a_iu,a_jv) &= h_{i,j}(u,v,f(u,v)) \qquad i,j\in\{1,\dots,r\} && (3.20)\end{aligned}$$

   We write $f = \mathit{Unif}_\varepsilon(h_{1,1},\dots,h_{r,r}) = \mathit{Unif}_\varepsilon(\overline{h})$

2. *Let* ε-$UR(\Sigma)$ *denote the set of* ε*-uniformly-recursive functions, i.e. the smallest set which contains the basic functions and is closed under composition and* ε*-uniform recursion, and* ε-$UR_n(\Sigma)$ *the subset of these functions which use at most* n *nested* ε*-uniform recursions.*

Consider lines 3.18 and 3.19. If at least one of the input parameters is the empty word the output will be the empty word. In other words: all input values must be non-empty to produce a non-empty output. Assume the input is read from left to right. Uniform recursion makes the input values to be read simultaneously. Whenever one of the input values ends while being read, the whole computation ends.

In the case of functions with a single input value ε-uniform recursion coincides with ε-primitive recursion. This is shown in case of the function *double* in the following example.

3.4.11 Example
Some ε-uniformly recursive functions:

1. $\mathit{shuffle}_\varepsilon \in \varepsilon\text{-}UR_1(\Sigma)$

$$\begin{aligned} \mathit{shuffle}_\varepsilon(\varepsilon,v) &= \varepsilon\\ \mathit{shuffle}_\varepsilon(u,\varepsilon) &= \varepsilon\\ \mathit{shuffle}_\varepsilon(a_iu,a_jv) &= \mathit{con}_i(\mathit{con}_j(\mathit{shuffle}_\varepsilon(u,v)))\end{aligned}$$

2. $\mathit{double} : \Sigma^* \to \Sigma^*, v_1\dots v_l \mapsto v_1v_1\dots v_lv_l \in \varepsilon\text{-}UR_1(\Sigma)$

   has the following ε-uniform recursion scheme

$$\begin{aligned} \mathit{double}(\varepsilon) &= \varepsilon\\ \mathit{double}(a_iv) &= a_ia_i\cdot \mathit{double}(v)\end{aligned}$$

3. The *uniform projections*

$$\mathit{pro}_i^k : (\Sigma^*)^k \to \Sigma^*(v_{1,1}\dots v_{1,l_1},\dots,v_{k,1}\dots v_{1,l_k}) \mapsto v_{i,1}\dots v_{i,\min\{l_1,\dots,l_k\}}$$

   with $1\le i\le k$.

$$\begin{array}{rcl}
pro_i^k(\varepsilon, v_2, \ldots, v_k) & = & \varepsilon \\
pro_i^k(v_1, \varepsilon, v_3, \ldots, v_k) & = & \varepsilon \\
& \vdots & \\
pro_i^k(v_1, \ldots, v_{k-1}, \varepsilon) & = & \varepsilon \\
\\
pro_i^k(a_{j_1}v_1, \ldots, a_{j_k}v_k) & = & a_{j_i} \cdot pro_i^k(v_1, \ldots, v_k)
\end{array}$$

$pro_i^k \in \varepsilon\text{-}UR_1(\Sigma)$.

The uniform projection pro_i^k is a slow version of the projection $proj_i^k$.

3.4.12 Lemma
Every ε-uniformly recursive function is monotonic.

PROOF. Analogous to the case of ε-primitive recursive functions by structural induction.

- All basic functions are monotonic.
- The class of monotonic functions is closed under composition.
- Let $f = Unif_\varepsilon(\overline{h})$, with $h \in \varepsilon\text{-}UR(\Sigma)$, i.e.

$$\begin{array}{rcl}
f(\varepsilon, w) & = & \varepsilon \\
f(v, \varepsilon) & = & \varepsilon \\
f(a_i v, a_j w) & = & h_{ij}(v, w, f(v, w))
\end{array}$$

We show the monotonicity, i.e. $f(v,w) \leq_p f(vv', ww')$ by simultaneous induction over the length of v and w

- $v = \varepsilon$ or $w = \varepsilon$, then we have

$$f(\varepsilon, w) = \varepsilon \leq_p f(v', ww'')$$

and

$$f(v, \varepsilon) = \varepsilon \leq_p f(vv', w'')$$

respectively
- $v = a_i v''$ and $w = a_j w''$, then

$$\begin{array}{l}
f(a_i v'', a_j w'') = h_{ij}(v'', w'', f(v'', w'')) \overset{\star}{\leq_p} \\
\qquad\qquad \overset{\star}{\leq_p} h_{ij}(v''v', w''w', f(v''v', w''w')) = \\
\qquad\qquad\qquad\qquad = f(a_i v''v', a_j w''w') = f(vv', ww')
\end{array}$$

$\star$ holds because of the monotonicity of h_{ij} and the induction hypothesis.

■

Now we can define how to simulate an infinite word function by a finite one.

3.4.13 Definition
A fully monotonic function $f : (\Sigma^*)^k \to \Sigma^*$ approximates *a continuous function* $f^\omega : (\Sigma^\omega)^k \to \Sigma^\omega$ *if for all* $\overline{y} \in (\Sigma^*)^k$ *and* $\overline{z} \in (\Sigma^\omega)^k$

$$y_1 \leq_p z_1, \ldots, y_k \leq_p z_k \Rightarrow f(\overline{y}) \leq_p f^\omega(\overline{z})$$

Obviously, every continuous function $f^\omega : (\Sigma^\omega)^k \to \Sigma^\omega$ can be approximated by a fully monotonic function $f : (\Sigma^*)^k \to \Sigma^*$, and every fully monotonic function $f : (\Sigma^*)^k \to \Sigma^*$ approximates a continuous function $f^\omega : (\Sigma^\omega)^k \to \Sigma^\omega$.

The latter is described a bit more detailed.

Let $v_0, v_1, \ldots \in \Sigma^*$ and $v_0 \leq_p v_1 \leq_p v_2 \leq_p \ldots$ and for all $l \in \mathbb{N}$ exists $i_l \in \mathbb{N}$ with $|v_{i_l}| \geq l$.

Then we can define $\lim_{i\in\mathbb{N}} v_i = v \in \Sigma^\omega$ with $v_i \leq_p v$. The sequence $(v_i)_{i\in\mathbb{N}}$ is monotonic and boundless, therefore v exists and is unique.

Obviously, for a fully monotonic function f and an infinite word $w = w_1 w_2 \ldots$ the sequence $(f(w_1 \ldots w_i))_{i\in\mathbb{N}}$ is monotonic and boundless. Then f approximates the function

$$f^\omega : \Sigma^\omega \to \Sigma^\omega, w_1 w_2 \ldots \mapsto \lim_{i\in\mathbb{N}} f(w_1 \ldots w_i)$$

3.4.14 Definition
The set ε-$UR^t(\Sigma)$ *of* truly uniformly recursive functions *is defined like* ε-$UR_n(\Sigma)$ *but with the uniform projections*

$$pro_i^k : (\Sigma^*)^k \to \Sigma^*, v_{1,1} \ldots v_{1,l_1}, \ldots, v_{k,1} \ldots v_{k,l_k} \mapsto v_{i,1} \ldots v_{i,\min_{1\leq j\leq k}(l_j)}$$

instead of the projections $proj_i^k$ *within the set of basic function, i.e.* ε-$UR^t(\Sigma)$ *is the smallest set of functions which contains the concatenations, the constant functions and the uniform projections and is closed under substitution and uniform recursion.*

As usual the sets ε-$SUR^t(\Sigma)$, ε-$UR_n^t(\Sigma)$ *and* ε-$SUR_n^t(\Sigma)$ *can be defined analogously.*

3.4.15 Remark
For all $k \geq 1$, $1 \leq i \leq k$ *and* $\overline{v} \in (\Sigma^*)^k$

$$pro_i^k(\overline{v}) \leq_p proj_i^k(\overline{v})$$

3.4.16 Lemma
For every function $f \in \varepsilon$-$UR^t(\Sigma)$ *exists a natural number* m *such that for all* $v_1, \ldots, v_k \in \Sigma^*$

$$|f(v_1, \dots, v_k)| \leq \min_{1\leq j\leq k}(|v_j|) + m$$

PROOF. Structural induction.

- Choose $m = 1$ for the successor functions and and $m = 0$ for the uniform projections.
- $f = Sub(g; \overline{h})$.

  By induction hypothesis respective natural numbers m_0 for g and m_i for h_i exist.

  Choose $m = m_0 + \min(m_1, .., m_l)$, then

$$\begin{aligned} |f(v_1, \dots, v_k)| &= |g(h_1(v_1, \dots, v_k), \dots, h_l(v_1, \dots, v_k)| \\ &\leq \min_{1\leq i\leq l}(|h_i(v_1, \dots, v_k)|) + m_0 \\ &= \min_{1\leq i\leq l}(\min_{1\leq j\leq k}|v_j|) + m_i) + m_0 \\ &= \min_{1\leq j\leq k}(|v_j|) + \min_{1\leq i\leq l}(m_i) + m_0 \\ &= \min_{1\leq j\leq k}(|v_j|) + m \end{aligned}$$

- $f^2 = Unif_\varepsilon(\overline{h})$.

  Choose $m = \max_{1\leq i\leq r, 1\leq j\leq r}(m_{ij})$, where m_{ij} is the respective constant for h_{ij} existing by induction hypothesis.

  - $|f(\varepsilon, v)| = |f(u, \varepsilon)| = |\varepsilon| = 0 \leq 0 + m$
  -

$$\begin{aligned} |f(a_i u, a_j v)| &= |h_{ij}(u, v, f(u, v))| + m_{ij} \\ &\leq \min(|u|, |v|, |f(u, v)|) + m_{ij} \\ &\leq \min(|u|, |v|) + m_{ij} \\ &\leq \min(|u|, |v|) + m \end{aligned}$$

■

3.4.17 Lemma

1. *ε-$UR^t(\Sigma) \subseteq \varepsilon$-$UR(\Sigma)$.*
2. *ε-$UR^t_n(\Sigma) \subseteq \varepsilon$-$UR_n(\Sigma)$ for all $n \geq 1$.*

PROOF.

1. Follows directly from $pro^k_i \in \varepsilon$-$UR(\Sigma)$.
2. Inductively replace f by $Sub(f; pro^k_1, \dots, pro^k_k)$. The claim follows from $pro^k_i \in \varepsilon$-$UR_1(\Sigma)$.

■

3.4.18 Corollary
$\varepsilon\text{-}UR^t(\Sigma) \subsetneqq \varepsilon\text{-}UR(\Sigma)$

3.4.19 Lemma
For every fully monotonic function $f \in \varepsilon\text{-}UR_n(\Sigma)$ there exists a slow version f_t of f with $f_t \in \varepsilon\text{-}UR_n^t(\Sigma)$

Proof. The idea is to replace all projections by uniform projections. A problem might occur when a function uses only a finite part of a certain argument. We add some dummy arguments in this case.

This is done inductively.

We consider only the interesting case. All other cases are treated straightforwardly.

In this case we have $f(a_i v) = h_i(v, f(v))$ with $h_i = h_i' \circ proj_1^2$ and $f(v) \leq_l v$. The replacement of $proj_1^2$ by pro_1^2 might shorten the value of h_i too much. We define

$$f_t(a_i v) = h_i'(pro_1^2(pro_1^2(v, f_t(v)), con_i(pro_2^2(v, f_t(v)))))$$

The second argument $f_t(v)$ is artificially extended by replacing it with $con_i(pro_2^2(v, f_t(v)))$. This construction ensures, that f_t will still increase and be fully monotonic.

This construction is illustrated in Example 3.4.21-2.

In all other cases the projections might be replaced by the uniform projections directly. ■

3.4.20 Corollary
For every fully monotonic function $f : (\Sigma^*)^k \to \Sigma^* \in \varepsilon\text{-}UR_n(\Sigma)$ there exists a fully monotonic function $f_t : (\Sigma^*)^k \to \Sigma^* \in \varepsilon\text{-}UR^t(\Sigma)$ such that f and f_t approximate the same function $f^\omega : (\Sigma^\omega)^k \to \Sigma^\omega$

Proof. Obviously, a function and all of its slow versions approximate the same function. Choose the truly uniformly recursive slow version from the lemma above. ■

3.4.21 Example

1. The function $double : \Sigma^* \to \Sigma^*, v_1 \dots v_l \mapsto v_1 v_1 \dots v_l v_l$ recursively defined by

$$\begin{array}{rcl} double(\varepsilon) & = & \varepsilon \\ double(a_i v) & = & a_i a_i \cdot double(v) \end{array}$$

   (compare Example 3.4.9-2, page 45) can be easily expressed by terms of truly uniform recursion.

   We explicate the projections

$$\begin{aligned} double(\varepsilon) &= \varepsilon \\ double(a_i v) &= con_i \circ con_i proj_2^2(v, double(v)) \end{aligned}$$

and replace them by uniform projections

$$\begin{aligned} double_t(\varepsilon) &= \varepsilon \\ double_t(a_i v) &= con_i \circ con_i pro_2^2(v, double_t(v)) \end{aligned}$$

We get

$$|double(v)| = 2 \cdot |v|$$

and

$$\begin{aligned} |double_t(v)| &= \begin{cases} 0 & \text{if } v = \varepsilon \\ |v| + 1 & \text{else} \end{cases} \\ &= sg(|v|) \cdot (|v| + 1) \\ &\leq |v| + 1 \end{aligned}$$

2. The function $rest : \Sigma^* \to \Sigma^*$ is defined by

$$\begin{aligned} rest(\varepsilon) &= \varepsilon \\ rest(a_i v) &= v \end{aligned}$$

With explicit projections we get

$$\begin{aligned} rest(\varepsilon) &= \varepsilon \\ rest(a_i v) &= proj_1^2(v, rest(v)) \end{aligned}$$

Replacing the projections by the uniform variants would give

$$\begin{aligned} rest'_t(\varepsilon) &= \varepsilon \\ rest'_t(a_i v) &= pro_1^2(v, rest'_t(v)) \end{aligned}$$

But this function is not fully monotonic. We can rather show by induction that for all $v \in \Sigma^*$ holds $rest'_t(v) = \varepsilon$.

To prevent that the (unused) projection parameter $rest'_t(v)$ shortens the used parameter v too much, we have to extend it artificially.

$$\begin{aligned} rest_t(\varepsilon) &= \varepsilon \\ rest_t(a_i v) &= pro_1^2(pro_1^2(v, rest_t(v)), con_i(pro_2^2(v, rest_t(v)))) \end{aligned}$$

Then we get

$$rest_t(v_1 \dots v_l) = v_2 \dots v_{l-1}$$

and

$$|rest(v)| = |v| \dot{-} 1$$

and

$$|rest_t(v)| = |v| \dot{-} 2$$

3.5 Lookahead of fully monotonic functions

Lemma 3.4.16 prevents us from distinguishing truly uniformly recursive functions by their increase. On the other hand, we have functions $f : \Sigma^* \to \Sigma^*$ for which

$$|f(\overline{v})| \leq |v|$$

3.5.1 Definition (Lookahead)

1. *A computable function $f : (\Sigma^*)^k \to \Sigma^*$ can be computed with* lookahead $lh_f : \mathbb{N} \to \mathbb{N}$ *if for all $x \in \mathbb{N}$ and $v_1, .., v_k \in \Sigma^*$*

$$[\forall 1 \leq i \leq k. |v_i| \geq lh_f(x)] \Rightarrow |f(v_1, \dots, v_k)| \geq x$$

2. *A computable function $f : (\Sigma^*)^k \to \Sigma^*$ can be computed with* minimal lookahead lh_f, *if f can be computed with lookahead lh_f and for all functions $l : \mathbb{N} \to \mathbb{N}$ with $l \leq lh_f$ such that f can be computed with lookahead l holds $l = lh_f$.*

We say, for short, f *has lookahead* lh_f, instead of f *can be computed with minimal lookahead* lh_f.

Obviously all fully monotonic functions can be computed with a certain lookahead.

We prove a simple but useful property. It is easy to estimate the lookahead a of function defined by substitution.

3.5.2 Lemma

Let $g : (\Sigma^*)^l \to \Sigma^*$ be computable with lookahead lh_g and $h_1, \dots, h_l : (\Sigma^*)^k \to \Sigma^*$ be computable with lookaheads $lh_1, \dots, lh_l$, respectively.

Then $Sub(g; h_1, \ldots, h_l)$ can be computed with lookahead

$$lh : \mathbb{N} \to \mathbb{N}, x \mapsto lh_g \left(\max_{1 \leq j \leq l} (lh_j(x)) \right)$$

PROOF. Let $x \in \mathbb{N}$ and $v_1, \ldots, v_l \in \Sigma^*$ with $|v_i| \geq lh(x)$ for all $i \in \mathbb{N}$, i.e.

$$x \geq lh_g \left(\max\{lh_i(x)) \in 1 \leq j \leq l\}\right) \geq lh_g(lh_j(x))$$

for all $1 \leq j \leq l$.

By definition of the lookaheads

$$|h_j(v_1, \ldots, v_k)| \geq lh_j(x)$$

for all $1 \leq j \leq l$ and

$$|g(h_1(v_1, \ldots, v_k), \ldots, h_l(v_1, \ldots, v_k))| \geq x$$

This proves that $Sub(g; h_1, \ldots, h_l)$ can be computed with lookahead lh. ■

3.5.3 Example
Consider the function $f : \Sigma^* \to \Sigma^*$ with

$$\begin{aligned} f(\varepsilon) &= \varepsilon \\ f(a_1 v) &= a_1 \cdot f(v) \\ f(a_i v) &= f(v) \qquad 2 \leq i \leq r \end{aligned}$$

This function is obviously monotonic but not fully monotonic. It cannot be computed with a certain lookahead. Since inputs of form a_2^l do not deliver an output for any $l \in \mathbb{N}$. For any $l \in \mathbb{N}$ there exists an input word w with length $|w| = l$ which produces an empty output. In other words, there is no input length which ensures a non-empty output.

In other words, considering lookaheads is reasonable only for fully monotonic word functions. This is done from now on.

The easiest case is recursion depth one. In this case a simultaneous recursion can be understood as a finite transducer with the simultaneously defined functions serving as states. This results in a boundary for the lookahead, which is proven similarly to the *Pumping-Lemma* for regular languages (see [HU79, Lew81]).

The transducer must run in a loop when the input length exceeds the number of states. Each cycle must produce at least one letter of output, otherwise an arbitrarily large input without output can be constructed by repeating that cycle. This contradicts the full monotonicity.

The use of finite transducers for the computation of real valued functions was examined in [Kon98, Kon00]. We will use this notion to support our imagination, but we will not give a formal definition. Of course, we can consider final transducers as special cases of TTMs.

3.5.4 Lemma
Let $f \in \varepsilon\text{-}SUR_1(\Sigma)$ be a fully monotonic function. Then a natural number t exists, such that f can be computed with lookahead $\lambda k.\, tk$.

Proof. By structural induction.

- $f = \mathit{Unif}_\varepsilon(\overline{g})$

  We will construct the loops mentioned before and estimate their lengths. The maximal lengths of each cycle will give the maximal number of input letters to produce at least one output letter. This is an upper bound for the lookahead.

  Let $k = 2$, therefore $f_1, \ldots, f_m : (\Sigma^*)^2 \to \Sigma^*$ and $f = f_1$

  Let $f_h(a_i u, a_j v) = g^h_{ij}(u, v, f_1(u, v), \ldots, f_m(u, v))$ for $1 \le h \le m$ be the simultaneous recursion scheme of $f_1 \ldots f_l$, where the functions g^h_{ij} are compositions of successor functions and projections, i.e. there exist $w^h_{ij} \in \Sigma^*$ and $s^h_{ij} \in \{1, \ldots, m+2\}$ for $1 \le i \le r$, $1 \le j \le r$ and $1 \le h \le m$ with

$$g^h_{ij}(\overline{w}) = w^h_{ij} \cdot \mathit{proj}^{m+2}_{s_{h,ij}}(\overline{w}) \tag{3.21}$$

  We want the simultaneously defined functions $f_1, \ldots, f_l$ to be understood as *states*. The equation 3.5 then tells us, when in state f_h the letters a_i and a_j are read, the word w^h_{ij} is concatenated to the output and the computation is continued at the consecutive state $f_{s_{h,ij}-2}$. In case $s_{h,ij} = 1$ or $s_{h,ij} = 2$ the output is continued by concatenating the unread remainder of one of the input parameters. This can be understood as an additional state which will not be left.

  Choose $t = \max(m, 2)$.

  We consider input words $a_{i_1} \ldots a_{i_l}$ and $a_{j_1} \ldots a_{j_l}$ and estimate the length of $f(a_{i_1} \ldots a_{i_l}, a_{j_1} \ldots a_{j_l})$.

  We have to distinguish cases since the lookahead behaves differently if a projection onto an input value occurs.

  - Assume that for all $h \in \{1, \ldots m\}$ and $i, j \in \{1, \ldots, r\}$ holds $s_{h,ij} > 2$.

    Let $h_1 \in \{1, \ldots, m\}$ and $h_{q+1} = s_{h_q,ij} - 2$ for $q \ge 1$. The sequence $h_0, h_1, h_2, \ldots, h_l$ denotes the states which are visited.

    Then from equation 3.5

$$f_{h_1}(a_{i_1} \ldots a_{i_l}, a_{j_1} \ldots a_{j_l}) = w^{h_1}_{i_1 j_1} \cdot \cdots \cdot w^{h_l}_{i_l j_l}$$

    can be derived.

    Consider $l_0, l_1 \in \mathbb{N}$ with $1 \le l_0 < l_1 \le l$ and $l_1 - l_0 \ge m$. Then $l \ge m$ and further $l' \in \mathbb{N}$ and $l'' \in \mathbb{N}$ exist with $l_1 \le l' < l'' \le l_1$ and $h_{l'} = h_{l''}$.

    Now assume

$$f_{h_1}(a_{i_{l_0}} \dots a_{i_{l_1}}, a_{j_{l_0}} \dots a_{j_{l_1}}) = \varepsilon \tag{3.22}$$

Then

$$f_{h_{l'}}(a_{i_{l'}} \dots a_{i_{l''}}, a_{j_{l'}} \dots a_{j_{l''}}) = w^{h_{l'}}_{i_{l'}j_{l'}} \cdot \dots \cdot w^{h_{l'}}_{i_{l'}j_{l'}} = \varepsilon$$

Since $h_{l'} = h_{l''}$ we can repeat this part of the computation. We obtain for all $p \in \mathbb{N}$

$$f_{h_{l'}}((a_{i_{l'}} \dots a_{i_{l''}})^p, (a_{j_{l'}} \dots a_{j_{l''}})^p) = \left(w^{h_{l'}}_{i_{l'}j_{l'}} \dots w^{h_{l'}}_{i_{l'}j_{l'}}\right)^p = \varepsilon$$

Then neither $f_{h_{l'}}$ nor f_{h_1} would be fully monotonic. With $h_1 = 1$ this is a contradiction to the f being fully monotonic. For each input sequence of length m at least one letter of output must be produced:

$$|u|, |v| \geq mk \Rightarrow |f(u,v)| \geq k$$

f can be computed with lookahead $\lambda k.\, mk \leq \lambda k.\, tk$.

– Assume a number $l' \in \mathbb{N}$ exists with $s_{h_{l'}, i_{l'}j_{l'}} \leq 2$, more precise choose $l' = \min\left\{l \mid s_{h_{l'}, i_{l'}j_{l'}} \in \{1,2\}\right\}$. Then $l' < m$, otherwise a loop would occur and lead us to the previous case again.
W.l.o.g. we assume $s_{h_{l'}, i_{l'}j_{l'}} = 1$.
Then

$$f_{h_1}(a_{i_1} \dots a_{i_{l'}} a_{i_{l'+1}} \dots a_l, a_{i_1} \dots a_{i_{l'}} a_{i_{l'+1}} \dots a_l) = w^{h_1}_{i_1 j_1} w^{h_{l'}}_{i_{l'}j_{l'}} a_{l'+1} \dots a_l$$

and

$$|f_{h_1}(a_{i_1} \dots a_{i_{l'}} a_{i_{l'+1}} \dots a_l, a_{i_1} \dots a_{i_{l'}} a_{i_{l'+1}} \dots a_l)| \geq |a_{i_{l'+1}} \dots a_l| \geq l - m$$

or rather

$$|u|, |v| \geq m + k \Rightarrow f_{h_1} \geq k$$

f_{h_1} can be computed with lookahead $\lambda k.\, k + m \leq \lambda k.\, k + t$.
Since $k + t \leq t \cdot k$ for $k \geq 1, t \geq 2$ f_{h_1} can be computed with lookahead $\lambda k.\, tk$.

Both cases show that f can be computed with lookahead $\lambda k.\, tk$.

- $f = Sub(g; \overline{h})$

  If g and $\overline{h}$ are fully monotonic, this case follows directly from Lemma 3.5.2.

  Otherwise, consider the case that h_i is not fully monotonic. Then f is not fully monotonic or it is "ignored" by a projection. We can also ignore it in the consideration of the lookahead.

  Consider the case that g is not fully monotonic. Since f is fully monotonic, g has to be fully monotonic on the subset $Im(h_1) \times \dots \times Im(h_l)$. We can apply the presented proof technique to g with respect to that subset of input values. This results in an analogous estimation of the lookahead of g on this subset of inputs. Lemma 3.5.2 can be applied analogously.

■

3.5.5 Corollary
A word function $f \in \varepsilon\text{-}UR_1(\Sigma)$ can be computed with lookahead $lh_f \in \mathcal{E}_1$.

It seems that higher lookaheads can be achieved via simultaneous recursion. Before displaying further results we consider some examples.

3.5.6 Example
Some fully monotonic functions and their lookaheads:

1. The projections and uniform projections can be computed with lookahead $id_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}, x \mapsto x$
2. The function $rest : v_1 \dots v_l \mapsto v_2 \dots v_l$ has lookahead $\lambda k.k+1$.
3. The functions

$$\left.\begin{array}{l} half : \Sigma^* \to \Sigma^*, v_1 v_2 \dots v_{l \div 2} \\ half' : \Sigma^* \to \Sigma^*, v_1 v_2 \dots v_{(l+1)\div 2} \end{array}\right\} \in SPR_1(\Sigma)$$

   (Example 3.2.8-9, page 32) have lookahead $\lambda x.\, 2x$ and $\lambda x.\, 2x \dot{-} 1$, respectively. This is an example for Lemma 3.5.4.

4. The function $half'$ can be defined in another way, we need some auxiliary functions:

   - An alternating function $alt : \Sigma^* \to \Sigma^*$
     with

     $$alt : v_1 \dots v_l \mapsto a_{i_1} a_{i_2} \dots a_{i_l}$$

     with

     $$i_j = \begin{cases} 1 & \text{if } j \text{ is odd} \\ 2 & \text{if } j \text{ is even} \end{cases}$$

     is defined by the the following simultaneous recursion scheme:

     $$\begin{array}{rcl} alt(a_i v) & = & a_1 \cdot alt'(v) \\ alt'(a_i v) & = & a_2 \cdot alt(v) \end{array}$$

     $ap : (\Sigma^*)^2 \to \Sigma^*$ defined by

     $$\begin{array}{rcl} ap(a_i u, a_j v) & = & a_j \cdot ap_i(u,v) \\ ap_i(a_i u, a_j v) & = & ap_i(u,v) \\ ap_i(a_{i'} u, a_j v) & = & a_j \cdot ap_{i'}(u,v) \end{array}$$

Then

$$\mathit{half}' = \mathit{ap}(\mathit{double}(\mathit{alt}(v), v))$$

5. We can use this principle to create other lookaheads:

   (a) $\mathit{rest} : v_1 \dots v_l \mapsto v_2 \dots v_l$

   $$\mathit{rest}(v) = \mathit{ap}(\mathit{con}_1(\mathit{alt}(v), v))$$

   (b) $\mathit{log} : \Sigma^* \to \Sigma^*$ with

   $$\mathit{log}(v) = \mathit{ap}(\mathit{md}_t(\mathit{alt}(v), v))$$

   Then

   $$\mathit{log}(v_1 \dots v_{2^l}) = v_1 v_2 v_4 v_8 \dots v_{2^{l-1}} v_{2^l}$$

   and *log* has lookahead $\lambda x.\, 2^x$.

It seems that the functions $\mathit{double} \circ \mathit{alt}$ or $\mathit{md}_t \circ \mathit{alt}$ produce a kind of *pattern* that can be applied to an input to get an output with a certain lookahead.

3.5.7 Definition
Let $f : \Sigma^* \to \Sigma^*$ be a fully monotonic word function and for all $v \in \Sigma^*$ with $f(v) \neq \varepsilon$ exists a word $v' \in \Sigma^*$ such that

- $f(v) = wa_i$ *with* $w \in \Sigma^*$
- $f(vv') = wa_i w' a_j w''$ *with* $w', w'' \in \Sigma^*$
- $i \neq j$

Then f is called a pattern.

We call

$$\mathit{pa}_f : x \mapsto \min\{|f(v)| \mid \mathit{ch}(f(v)) \geq x\}$$

with

$$\mathit{ch} : \Sigma^* \to \mathbb{N}, a_{i_1} \dots a_{i_l} \mapsto |\{j \mid i_j \neq i_{j+1}\}|$$

the pattern length *of f.*

Reconsidering our latest example shows that we can use certain patterns to construct functions with certain lookaheads. We will generalize the idea of these examples. We observe a connection between lookaheads and patterns.

We can apply a pattern to a word.

3.5.8 Lemma
Let $f \in \varepsilon\text{-}UR^t_n(\Sigma)$ $(n \geq 1)$ be a pattern with pattern length pa_f. Then a function $f' \in \varepsilon\text{-}UR^t_n(\Sigma)$ exists which has lookahead pa_f.

Proof. Let the function $ap : (\Sigma^*)^2 \to \Sigma^*$ be defined by the following simultaneous recursion scheme:

$$\begin{aligned} ap(a_i u, a_j v) &= a_j \cdot ap_i(u,v) \\ ap_i(a_i u, a_j v) &= ap_i(u,v) \\ ap_i(a_{i'} u, a_j v) &= a_j \cdot ap_{i'}(u,v) \end{aligned}$$

Then $ap \in \varepsilon\text{-}UR^t_1(\Sigma)$.

Let $f' = ap \circ (f, pro^1_1)$, i.e. $f'(v) = ap(f(v), v)$. Then f' obviously has lookahead pa_f. ■

The function ap is monotonic but not fully monotonic, but $ap \circ f$ is fully monotonic if f is a pattern.

The next step is to construct patterns with arbitrary increasing pattern lengths. We use the increase of ε-recursive functions. We will use a set of functions similar to the Ackermann functions to serve as patterns.

3.5.9 Definition
We define some functions:

1. *We define m'_n for $n \geq 1$ inductively:*

$$\begin{aligned} m'_1 &= double \\ m'_{n+1} &= Unif_\varepsilon((con_i \circ m'_n \circ proj^2_2)_{1\leq i\leq r}) \end{aligned}$$

In other words, m'_{n+1} has the following recursion scheme:

$$\begin{aligned} m'_{n+1}(\varepsilon) &= \varepsilon \\ m'_{n+1}(a_i v) &= a_i \cdot m'_n(m'_{n+1}(v)) \end{aligned}$$

2. *$m_n = m'_n \circ alt$ for all $n \geq 1$*

3. $l_n : v \mapsto ap(m_n(v), v)$ *for all* $n \geq 1$

These functions are built to obtain patterns with increasing pattern lengths. We will describe these patterns and their lengths a bit more detailed.

3.5.10 Remark

1. Obviously $m_n, m'_n, m''_n \in \varepsilon\text{-}SUR_n(\Sigma)$.

2. m_n *is a pattern for all* $n \geq 1$.

3.5.11 Lemma

1. $m'_1 \in \mathcal{E}_1(\Sigma) \setminus \mathcal{E}_0(\Sigma)$

2. $m'_n \in \mathcal{E}_{n+1}(\Sigma) \setminus \mathcal{E}_n(\Sigma)$ *for all* $n \geq 2$

Proof.

1. From $|double(v)| = 2 \cdot |v|$ it follows that $double \in \mathcal{E}_1(\Sigma) \setminus \mathcal{E}_0(\Sigma)$

2. By induction

   - $n = 2$

     By induction over $l \geq 1$ it can be shown

     - $l = 0$. $m'_2(\varepsilon) = \varepsilon$
     - $l > 0$:

$$\begin{aligned} m'_2(v_1 \dots v_l) &= v_1 \cdot double(m'_2(v_2 \dots v_l) \\ &\overset{\text{I.H.}}{=} v_1 \cdot double(v_2^1 \dots v_l^{2^{l-2}}) \\ &= v_1 v_2^2 \dots v_l^{l-1} \end{aligned}$$

     Hence

$$|m_2(v)| = 2^{|v|} \dot{-} 1 = B_3(|v|)$$

     and

     $m'_2 \in \mathcal{E}_3(\Sigma) \setminus \mathcal{E}_2(\Sigma)$

   - $n > 2$

     By induction hypothesis we know that $m_{n-1} \in \mathcal{E}_n(\Sigma) \setminus \mathcal{E}_{n-1}(\Sigma)$.

     By induction over l we can show

$$m'_n(v_1 \dots v_l) = {m'_{n-1}}^0(v_1) \cdot \ldots \cdot {m'_{n-1}}^{l-1}(v_l)$$

     - $l = 0$: $m'_n(\varepsilon) = \varepsilon$

– $l > 0$:

$$\begin{aligned} m'_n(v_1 \dots v_l) &= v_1 \cdot m'_{n-1}(m_n(v_2 \dots v_l)) \\ &= v_1 \cdot m'_{n-1}({m'_{n-1}}^0(v_2) \cdot \dots \cdot {m'_{n-1}}^{l-2}){m'_{n-1}}^0(v_1) \cdot \dots \cdot {m'_{n-1}}^{l-1}(v_l) \end{aligned}$$

The consideration of the output lengths and the induction hypothesis shows

$$m'_n \in \mathcal{E}_{n+1}(\Sigma) \setminus \mathcal{E}_n(\Sigma)$$

■

From the form of the function values shown in the last proof we can derive the following statements.

3.5.12 Corollary

1. *m_1 is a pattern with pattern length $\lambda k.\, 2(k+1)$*
2. *m_n is a pattern with pattern length $pa_n \in \mathcal{E}_{n+1}(\Sigma) \setminus \mathcal{E}_n(\Sigma)$*
3. *l_1 is a function which has lookahead $\lambda k.\, 2k$*
4. *l_n is a function which has lookahead $lh_n \in \mathcal{E}_{n+1}(\Sigma) \setminus \mathcal{E}_n(\Sigma)$*

3.5.13 Corollary

ε-$SUR_n(\Sigma) \subsetneq \varepsilon$-$SUR_{n+1}(\Sigma)$ for all $n \in \mathbb{N}$.

This shows that we can construct truly uniformly recursive word functions with arbitrary large lookaheads.

We can obtain a pattern from a lookahead. We redefine the recursion scheme of a function by marking the positions which do not produce an additional output.

3.5.14 Lemma

Let $f \in \varepsilon$-$UR_n(\Sigma)$ $(n \geq 1)$ be a fully monotonic function with lookahead lh_f. Then a pattern $f' \in SUR_n(\Sigma)$ exists with pattern length lh_f.

PROOF.

We consider cases, depending on the definition of f using substitution after the recursion or not.

- $(f, \overline{f}) = Unif_\varepsilon(\overline{h})$

  We consider the recursion scheme of f

$$f(a_i v, a_j w) = h_{ij}(v, w, f(v, w))$$

We define f and $\overline{f}$ simultaneously by

$$\begin{aligned} \overline{f}(\varepsilon) &= \varepsilon \\ \overline{f}(a_i v, a_j w) &= \begin{cases} a_2 \cdot \overline{f} & \text{if } cut(f(v,w), h_{ij}(v,w,f(v,w))) = \varepsilon \\ a_1 \cdot \overline{f} & \text{else} \end{cases} \end{aligned}$$

We compare the current value of f with the previous one. If no output is added, this is stored in the symbol a_2. The symbol a_1 stores an additional output.

This function is not yet a pattern, but its output has a structure that can easily be turned into a pattern:

Let

$$\begin{aligned} a(a_1 v) &= a_2 a'(v) \\ a(a_2 v) &= a_1 a(v) \\ a'(a_1 v) &= a_1 a(v) \\ a'(a_2 v) &= a_2 a'(v) \end{aligned}$$

$$v \mapsto a\left(\overline{f}(a_1^{|v|})\right)$$

- $f = Sub(g; \overline{h})$. W.l.o.g we can assume that g is defined by recursion. Let g' be constructed from g as in the previous case. Then choose $f' = Sub(g'; h)$.

■

3.5.15 Lemma
Let $f \in PR_n(\Sigma)$ be a pattern. Then for pattern length pa_f

1. $n = 1 \Rightarrow pa_f \in \mathcal{E}_n$
2. $n > 1 \Rightarrow pa_f \in \mathcal{E}_{n+1}$

PROOF. For each pattern $|v| = |w|$ implies $f(v) = f(w)$. We can assume that f is built from a function f' and an alternating function a, like *alt*, with $a \in PR_1(\Sigma)$ and $f' \in PR_n(\Sigma)$.

Applying f' to the initial pattern a means expanding the alternating parts according to the increase of f'.

Therefore the pattern length cannot increase faster than the length of values of f'. ■

The latest two lemmas prove the bounds of lookaheads

3.5.16 Lemma
For $n \geq 2$ every fully monotonic word function $f : \Sigma^* \rightarrow \Sigma^* \in \varepsilon\text{-}SUR_n(\Sigma)$ can be computed with a lookahead $lh_f \in \mathcal{E}_{n+1}$.

Starting with a truly uniformly recursive function and replacing it by its fast version (by replacing all uniform projections by projections) shows that these functions are arranged in a hierarchy, as well.

3.5.17 Corollary
ε-$UR^t_n(\Sigma) \subsetneqq \varepsilon$-$UR^t_{n+1}(\Sigma)$ for all $n \in \mathbb{N}$.

Functions with large lookaheads will occur when computing real valued functions. This is shown in the next section.

3.6 Real Valued Functions

To compute real valued functions with our model, we have to write real numbers as infinite words.

Computability and complexity of th real valued functions depend on the representation of real numbers. [Wei97a] shows that the signed digit representations are adequate for considering complexity issues on reals (see Section 2.3 for some details).

We repeat the definition

3.6.1 Definition

$$\rho_2 : \{., \overline{1}, 0, 1\}^\omega \rightharpoonup \mathbb{R}, a_k \ldots a_0.a_{-1}a_{-2} \ldots \mapsto \sum_{i=-\infty}^{k} a_i 2^i$$

where $\overline{1}$ should be read as -1.

If no confusion is possible we use the same name for finite digit representations:

$$\rho_2 : \{., \overline{1}, 0, 1\}^* \rightharpoonup \mathbb{R}, a_k \ldots a_0.a_{-1}a_{-2} \ldots a_{-l} \mapsto \sum_{i=-l}^{k} a_i 2^i$$

and

$$\rho_2^* : \{., \overline{1}, 0, 1\}^* \rightharpoonup \mathscr{P}(\mathbb{R}), a_k \ldots a_0.a_{-1}a_{-2} \ldots a_{-l} \mapsto$$
$$\mapsto \left[\rho(a_k \ldots a_0.a_{-1}a_{-2} \ldots a_{-l}) - 2^{-l}, \rho(a_k \ldots a_0.a_{-1}a_{-2} \ldots a_{-l}) + 2^{-l}\right]$$

We want to restrict the following considerations to the interval [0,1].

3.6.2 Definition

$$\rho : \{\overline{1}, 0, 1\}^\omega \to [0,1], a_1 a_2 \ldots \mapsto \frac{1}{2} + \sum_{i \geq 1} a_i 2^{-(i+1)}$$

We want to use word functions over the alphabet $\Sigma = \{\overline{1}, 0, 1\}$ to compute functions on the interval $[0,1]$.

3.6.3 Definition
Let $\Sigma = \{\overline{1}, 0, 1\}$.

A fully monotonic function $f : (\Sigma^*)^k \to \Sigma^*$ approximates *a function $\varphi : [0,1]^k \to [0,1]$, if f approximates a function $f^\omega : (\Sigma^\omega)^k \to \Sigma^\omega$ with*

$$\rho(f^\omega(w_1, \ldots, w_k)) = \varphi(\rho(w_1, \ldots, w_k))$$

for all $(w_1, \ldots, w_k) \in (\Sigma^\omega)^k$.

We say f computes *φ, if f in computable in some sense in the above configuration.*

Obviously f computes φ if for all $w_1, \ldots, w_k \in \Sigma^*$ and $\xi_1, \ldots, \xi_k \in [0,1]$

$$(\forall 1 \leq i \leq k.\, \xi_i \in \rho^*(w_i)) \Rightarrow \varphi(\xi_1, \ldots, \xi_k) \in \rho^*(f(w_1, \ldots, w_n))$$

3.6.4 Definition
A function $g : \mathbb{N} \to \mathbb{N}$ is called sublinear *if $k, l \in \mathbb{N}$ with $0 < l < k$ and $x_0 \in \mathbb{N}$ exist such that $k \cdot g(x) \leq l \cdot x$ for all $x > x_0$.*

A function is sublinear if from a certain point it grows less than the identity.

3.6.5 Example
Examples of sublinear functions

1. Every constant function is sublinear.
2. The function $\mathbb{N} \to \mathbb{N} : x \mapsto \lfloor \frac{x}{k} \rfloor$is sublinear for all $k \geq 2$.
3. The "inverse functions" of the Ackerman functions B_n with $n \geq 2$ defined by

   $$b_n(x) = \mu z \leq x.\, B_n(z) \geq x$$

   are sublinear.

We will now consider real valued functions approximated with sublinear lookahead.

3.6.6 Lemma
Let $\varphi : [0,1] \to [0,1]$ be a a real valued function, computed by a fully monotonic function $f : \Sigma^* \to \Sigma^*$ such that f has a sublinear lookahead. Then φ is constant.

Proof. Let f be computable with lookahead $g : \mathbb{N} \to \mathbb{N}$. The definition of sublinearity shows that from an input of length $m + kt$ an output of length $l \cdot t$ with $k > 1$ can be computed.

Let $w = a_0 \dots a_m$ and $v = b_1 \dots b_r = f(w)$. Let $\xi = \rho(w)$ and $\zeta = \rho(v)$ the center of the intervals $I = \rho^*(w)$ and $J = \rho^*(v)$ respectively.

We have $f(I) \subseteq J$ obviously. We want to determine the range of the values of f over I.

Setting $l(M) = |\max(M) - \min(M)|$ for an interval M, we obtain $l(f(I)) \leq l(J)$.

Obviously $l(J) \leq l(\rho^*(f(w\overline{1}))) + l(\rho^*(f(w1)))$. This is shown in Figure 3.1. The crosshatched area shows, where the graph of φ is expected. Adding k more input digits delivers at least l more output digits

$$l(J) \leq 2^k \cdot 2^{-(r+l)}$$

Repeating this intersection i times we get

$$\begin{aligned} l(J) &\leq 2^{ik} \cdot 2^{-(r+il)} \\ &= 2^{-(r+i(\overbrace{l-k}^{>0}))} \end{aligned}$$

This equation holds for all $i \in \mathbb{N}$, i.e. the J becomes arbitrary small. φ is constant on all intervals I with $l(I) \leq 2^{-m}$. Since φ is continuous, it is constant. ■

In other words, a sublinear lookahead is useless, since it can be used for constant functions only, which are computable without any lookahead.

Non constant, total, real valued functions need at least linear lookahead. The most simple non constant total function is the identity. It is not very surprising, that more complex functions cannot be computed with smaller lookahead.

3.6.7 Example
We consider the function $f : \Sigma^* \to \Sigma^*$ with the following recursion scheme

$$\begin{aligned} f(\varepsilon) &= \varepsilon \\ f(aw) &= 0af(w) \end{aligned}$$

which results in

$$f(a_1 \dots a_k) = 0a_1 0a_2 \dots 0a_k$$

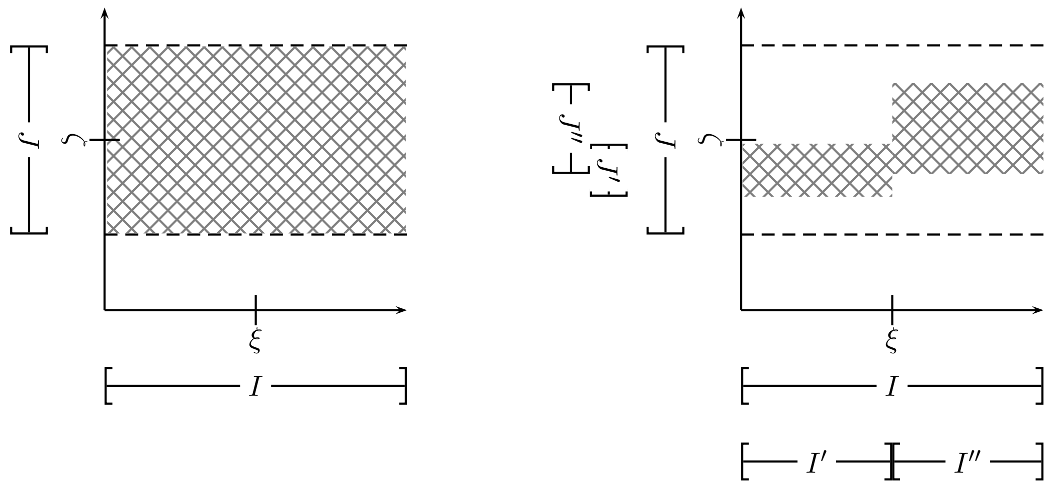


Figure 3.1: Possible range of a real valued function. Actually this picture shows the case $k = 1$.

To determine if this function approximates a real valued function on the interval $[-1, 1]$, we assume there is such a function which we call φ. We notice that f is computed with lookahead $k \mapsto \lfloor \frac{k}{2} \rfloor$, which is sublinear, hence φ must be constant.

We consider $f(11) = 0101$ and $\rho^*(0101) = [\frac{1}{4}, \frac{3}{8}]$, therefore $\varphi(1) \geq \frac{3}{8} > 0$. Analogously we obtain $\varphi{-1} < 0$. Since this means that φ is not constant this is a contradiction, i.e. f does not compute a real valued function.

We consider this example from another point of we. We consider two different representations of 0. From $\rho^*(f(1\bar{1}))\rho^*(010\bar{1}) = [\frac{1}{3}, \frac{1}{4}]$ it follows that $\varphi(0) > 0$, from $\rho^*(f(\bar{1}1)) = [-\frac{1}{3}, -\frac{1}{4}]$ it follows that $\varphi(0) < 0$. This is obviously an contradiction, as well.

We already have a set of functions guaranteeing certain lookaheads.

3.6.8 Lemma
A truly uniformly recursive function cannot be computed with a sublinear lookahead.

Proof. As shown in Lemma 3.4.16 (page 48) an increase of the output of a truly uniformly recursive function is bounded rather strictly, namely

$$|f(v_1, \ldots, v_k)| \leq \min_{1 \leq j \leq l}(|v_l|) + m$$

for words $v_1, \ldots, v_k \in \Sigma^*$ with a constant $m \in \mathbb{N}$. In other words, at least $r - m$ input digits are needed to produce at least r output digits. The lookahead is at least linear. ∎

The proof of Lemma 3.6.6 does not use the totality of the considered function, we do only need it to be total on a certain interval.

3.6.9 Corollary
Let $\varphi : \mathbb{R} \rightharpoonup \mathbb{R}$ be a function and $I \subseteq Def(\varphi)$ be an interval. If φ can be computed with

sublinear lookahead on I then φ is constant on I.

Chapter 4

Dyadic Recursion on Intervals

PCF is theoretical equivalent for functional programming languages. It uses *abstraction* and *application* from the λ-calculus. The constants — zero, successor function, a conditional and a fixed point operator — are chosen to compute μ-recursive functions [Plo77].

Real PCF was introduced in [Esc96b, Esc96a]. It is an extension of *PCF*. A type of intervals and corresponding constants are added. Computation on intervals is used to approximate computations on real numbers, again.

For our purpose we will not need to introduce Real PCF. We need some of the underlying techniques. Particularly we need interval-CPO-s.

4.1 Interval-CPO-s

Some preliminaries on CPO-s can be found in Section 2.4 (p. 22). The omitted proofs can be found in [Esc96b] or [Sch97].

We consider some facts on interval-CPO-s.

4.1.1 Lemma
$(\mathcal{R}, \sqsubseteq, \bot)$ *defined by*

- $\mathcal{R} = \{[\xi, \xi'] \mid \xi, \xi' \in \mathbb{R}, -\infty < \xi \leq \xi' < +\infty\} \cup [-\infty, +\infty]$
- $\mathbf{x} \sqsubseteq \mathbf{y} \iff \mathbf{x} \supseteq \mathbf{y}$
- $\bot = [-\infty, +\infty]$

is an ω-continuous CPO. *Its basis is*

$$\mathcal{Q} := \{[\xi, \xi'] \mid \xi, \xi' \in \mathbb{Q}, -\infty < \xi < \xi' < +\infty\} \cup \{[-\infty, +\infty]\}$$

Obviously $\mathbf{x} \uparrow \mathbf{y} \iff \mathbf{x} \cap \mathbf{y} \neq \emptyset$. In this case we have $\mathbf{x} \sqcup \mathbf{y} = \mathbf{x} \cap \mathbf{y}$

4.1.2 Definition
We call $\mathcal{R}$ the partial real line*.*

The singleton intervals are the maximal elements of $(\mathcal{R}, \sqsubseteq, \bot)$.

4.1.3 Lemma
The set $\{[\xi, \xi] \mid \xi \in \mathbb{R}\}$ of maximal elements with the subspace topology of the Scott topology of $\mathcal{R}$ is homeomorphic to the real line with standard topology.

That is why we can identify a real number ξ with the interval $\tilde{\xi} := [\xi, \xi]$

4.1.4 Definition
The partial unit interval *$(\mathcal{I}, \sqsubseteq, \bot)$ is defined by*

- $\mathcal{I} = \{[\xi, \xi'] \mid \xi, \xi' \in \mathbb{R}, 0 \le \xi \le \xi \le 1\}$.
- *The order $\sqsubseteq$ is the same as in the case of $\mathcal{R}$, i.e. $[\xi, \xi'] \sqsubseteq [\eta, \eta'] \iff \xi \le \eta \wedge \eta' \le \xi'$*
- *The least element is $\bot_{\mathcal{I}} = [0, 1]$.*

4.1.5 Lemma
The partial unit interval $(\mathcal{I}, \sqsubseteq, \bot)$ is a CPO*.*

Obviously $\mathbf{x} \uparrow \mathbf{y} \iff \mathbf{x} \cap \mathbf{y} \neq \emptyset$ for all $\mathbf{x}, \mathbf{y} \in \mathcal{I}$.

4.1.6 Lemma
1. *For two intervals $[\xi, \xi'], [\eta, \eta'] \in \mathcal{I}$ an infimum exists with*

$$[\xi, \xi'] \sqcap [\eta, \eta'] = [\min(\xi, \eta), \max(\xi'\eta')]$$

2. *The function $\mathcal{I} \times \mathcal{I} \to \mathcal{I}, (\mathbf{x}, \mathbf{y}) \rightharpoonup \mathbf{x} \sqcap \mathbf{y}$ is continuous.*

We want to compute on intervals, we define our basic functions.

4.1.7 Definition
The concatenation *of intervals is defined by:*

$$\cdot : \mathcal{I} \times \mathcal{I} \to \mathcal{I}, ([\xi, \xi'], [\eta, \eta']) \mapsto [(\xi' - \xi)\eta + \xi, (\xi' - \xi)\eta' + \xi]$$

The function $\cdot$ is continuous in the second argument, but not in the first.

4.1.8 Lemma
The function

$$\mathrm{cons}_{\mathbf{x}} : \mathcal{I} \to \mathcal{I}, \mathbf{y} \mapsto \mathbf{x} \cdot \mathbf{y}$$

is continuous for every $\mathbf{x} \in \mathcal{I}$.

If $\mathbf{x}$ is non-maximal $\mathrm{cons}_\mathbf{x}$ will be injective, hence a left inverse of $\mathrm{cons}_\mathbf{x}$ exists. We define $\mathbf{x}\backslash\mathbf{z} := \mathrm{cons}_\mathbf{x}^{-1}(\mathbf{z}) = \mathbf{y}$ with $\mathbf{xy} = \mathbf{z}$.

We need a left inverse of $\mathrm{cons}_\mathbf{x}$ for the general case.

4.1.9 Definition
We define

$$\mathrm{idem}_{[\xi,\xi']} : \mathcal{I} \to \mathcal{I}, [\zeta,\zeta'] \mapsto \begin{cases} [\xi,\xi] & \textit{if } \zeta' < \xi \\ [\zeta,\zeta'] \sqcup [\xi,\xi'], & \textit{if } [\zeta,\zeta'] \uparrow [\xi,\xi'] \\ [\xi',\xi'] & \textit{if } \xi' < \zeta \end{cases}$$

and

$$\mathrm{tail}_\mathbf{x}(\mathbf{z}) = \mathbf{x}\backslash\mathrm{idem}_\mathbf{x}(\mathbf{z})$$

4.1.10 Lemma
The functions $\mathrm{cons}_\mathbf{x}$ *and* $\mathrm{tail}_\mathbf{x}$ *are continuous for every* $\mathbf{x} \in \mathcal{I}$*. They form a* section-retraction pair*, i.e.* $\mathrm{tail}_\mathbf{x}(\mathrm{cons}_\mathbf{x}(\mathbf{y})) = \mathbf{y}$*, for every non-maximal* $\mathbf{x} \in \mathcal{I}$*.*

Since $\mathrm{cons}_\mathbf{x}(\mathrm{tail}_\mathbf{x}(\mathbf{y})) \not\sqsubseteq \mathbf{y}$ the functions $\mathrm{cons}_\mathbf{x}$ and $\mathrm{tail}_\mathbf{x}$ do not form a *projection-embedding-pair*.

We present some examples to become familiar with these functions.

4.1.11 Example
Let $L := \left[0, \frac{1}{2}\right]$, $C := \left[\frac{1}{4}, \frac{3}{4}\right]$ and $R := \left[\frac{1}{2}, 1\right]$.

- $\mathrm{cons}_R(L) = \mathrm{cons}_C(R) = [1/2, 3/4]$
- $\bigsqcup_{i\in\mathbb{N}} C^i = [1/2, 1/2]$
- $\mathrm{cons}_R(L) = \mathrm{cons}_C(R) = \left[\frac{1}{2}, \frac{3}{4}\right]$
- $\mathrm{cons}_{[\xi,\xi]}(\mathbf{y}) = [\xi,\xi]$ for all $\xi \in [0,1]$, $\mathbf{y} \in \mathcal{I}$
- $\mathrm{idem}_C([1/2,3/4]) = [1/2,3/4]$ and $\mathrm{tail}_C([1/2,3/4]) = R$
- $\mathrm{idem}_\mathbf{x}(\bot) = \mathbf{x}$ and $\mathrm{tail}_\mathbf{x}(\bot) = \bot$
- $\mathrm{idem}_L(C) = [1/4,1/2]$ and $\mathrm{tail}_L(C) = R$
- $\mathrm{idem}_R(C) = [1/2,3/4]$ and $\mathrm{tail}_L(C) = L$
- $\mathrm{idem}_L(R) = [1/2,1/2]$ and $\mathrm{tail}_L(R) = [1,1]$
- $\mathrm{idem}_R(L) = [1/2,1/2]$ and $\mathrm{tail}_R(L) = [0,0]$
- $\mathrm{idem}_C([0,1/10]) = [1/4,1/4]$ and $\mathrm{tail}_C([0,1/10]) = [0,0]$
- $\mathrm{idem}_C([0,0]) = [1/4,1/4]$ and $\mathrm{tail}_C([0,0]) = [0,0]$

$\mathbf{x} \sqsubseteq \mathrm{cons}_{\mathbf{x}}(\mathbf{y})$ for all $\mathbf{x}, \mathbf{y} \in \mathcal{I}$, i.e. $\mathrm{cons}_{\mathbf{x}}$ refines the information contained in $\mathbf{x}$ by $\mathbf{y}$. This is similar to refining the information of a word by adding a letter or a digit. There is a connection between intervals and digits.

4.1.12 Lemma

1. *Let* $\mathcal{Q}_e \subseteq_e (\{[p,q] \mid 0 \leq p < q \leq 1]\} \setminus \{[0,1]\}) =: \{\mathbf{x}_1, \ldots, \mathbf{x}_r\}$ *be a finite set of intervals with* $\bigcup_{\mathbf{x} \in \mathcal{Q}_e} \mathbf{x} = [0,1]$. *Then*

$$\begin{aligned} \epsilon_{\mathcal{Q}_e} : \{1, \ldots, r\}^\omega \to [0,1], (j_i)_{i \in \mathbb{N}} &\mapsto \mathrm{cons}_{\mathbf{x}_{j_0}}(\mathrm{cons}_{\mathbf{x}_{j_1}}(\ldots)\ldots) \\ &= \lim_{i \in \mathbb{N}} (\mathrm{cons}_{\mathbf{x}_{j_0}}(\mathrm{cons}_{\mathbf{x}_{j_1}}(\ldots(\mathrm{cons}_{\mathbf{x}_{j_i}}([0,1])))) \\ &= \lim_{i \in \mathbb{N}} x_{j_0} \cdot \ldots \cdot x_{j_i} \end{aligned}$$

   is a representation of $[0,1]$ *and*

$$\epsilon_{\mathcal{Q}_e} \leq \rho$$

2. *Let* $0 < b < a < 1$ *and* $l = [0,a]$ *and* $r = [b,1]$ *then* $\epsilon_{\{l,r\}} \equiv \rho$

Proof.[Sch97] ■

Considering intervals as (generalized) digits, Case 2. delivers a representation based on two digits that is equivalent to our standard representation which needs three digits.

We consider a certain set of intervals, which we met already in Example 4.1.11.

4.1.13 Definition
Let

- $L := \left[0, \frac{1}{2}\right]$,
- $C := \left[\frac{1}{4}, \frac{3}{4}\right]$ *and*
- $R := \left[\frac{1}{2}, 1\right]$.

These intervals are of special interest because they correspond to the digits $\overline{1}$,0 and 1 respectively.

4.1.14 Lemma
Let $\epsilon_{\{L,C,R\}} : \{1,2,3\}^\omega \to \mathbb{R}$ *be defined as in Lemma 4.1.12. Then*

$$\epsilon_{\{L,C,R\}} \equiv \rho$$

Proof.[Sch97] ■

4.1.15 Definition
We define some functions on intervals

$$\mathtt{pif} : \{\mathbf{tt}, \mathbf{ff}\}_\perp \times \mathcal{I} \times \mathcal{I} \to \mathcal{I}, (b, \mathbf{x}, \mathbf{y}) \mapsto \begin{cases} \mathbf{x} & \textit{if } b = \mathbf{tt} \\ \mathbf{y} & \textit{if } b = \mathbf{ff} \\ \mathbf{x} \sqcap \mathbf{y} & \textit{if } b = \perp \end{cases}$$

We write $\mathtt{pif}\, b\, \mathtt{then}\, \mathbf{x}\, \mathtt{else}\, \mathbf{y}$ *instead of* $\mathtt{pif}(b, \mathbf{x}, \mathbf{y})$.

$$<_\perp : \mathcal{I} \times \mathcal{I} \to \{\mathbf{tt}, \mathbf{ff}\}_\perp, ([\xi, \xi'], [\eta, \eta']) \mapsto \begin{cases} \mathbf{tt} & \textit{if } \xi' < \eta \\ \mathbf{ff} & \textit{if } \eta' < \xi \\ \perp & \textit{if } [\xi, \xi'] \cap [\eta, \eta'] \neq \emptyset \end{cases}$$

and

$$\text{left} : \mathcal{I} \to \{\mathbf{tt}, \mathbf{ff}\}_\perp, x \mapsto x <_\perp \frac{1}{2}$$

4.1.16 Lemma
The functions `pif`, $<_\perp^{\mathcal{I}}$ *and* left *are continuous.*

Functions on intervals are useful to compute functions on real numbers.

4.1.17 Definition
A continuous interval function $F : \mathcal{I}^k \to \mathcal{I}$ approximates *a real valued function* $\varphi : [0,1]^k \to [0,1]$*, if for all* $\xi_1, \dots, \xi_k \in [0,1]$ *holds*

$$F([\xi_1, \xi_1], \dots, [\xi_k, \xi_k]) = [\varphi(\xi_1, \dots, \xi_k), \varphi(\xi_1, \dots, \xi_k)]$$

We say F computes φ *if in addition* F *is computable in some sense.*

A function $F : \mathcal{I} \to \mathcal{I}$ approximates $\varphi : [0,1] \to [0,1]$, iff

$$\forall \mathbf{x} \in \mathcal{I}, \xi \in [0,1].\ (\xi \in \mathbf{x} \Rightarrow \varphi(\xi) \in F(\mathbf{x}))$$

This follows directly from the monotonicity of a continuous interval function.

4.2 Dyadic Recursion

[Esc96b] does not give an explicit definition of *dyadic recursion*, but it is easy to assemble.

4.2.1 Definition

1. *A function $F : \mathcal{I}^k \to \mathcal{I}$ is defined from $H_1, \dots, H_{k^2} : \mathcal{I}^{k+1} \to \mathcal{I}$ by* dyadic recursion *if*

$$\begin{aligned}
F(\mathbf{x}_1, \dots, \mathbf{x}_k) = {} & \texttt{pif}\,\mathrm{left}(\mathbf{x}_1)\,\texttt{then}\,\texttt{pif}\,\mathrm{left}(\mathbf{x}_2) \dots \\
& \dots \texttt{pif}\,\mathrm{left}(\mathbf{x}_k)\,\texttt{then}\,H_1(\mathrm{tail}_L(\mathbf{x}_1), \dots, \mathrm{tail}_L(\mathbf{x}_k), F(\mathrm{tail}_L(\mathbf{x}_1), \dots, \mathrm{tail}_L(\mathbf{x}_k))) \\
& \qquad \texttt{else}\,H_2(\mathrm{tail}_L(\mathbf{x}_1), \dots, \mathrm{tail}_R(\mathbf{x}_k), F(\mathrm{tail}_L(\mathbf{x}_1), \dots, \mathrm{tail}_R(\mathbf{x}_k))) \\
& \qquad \vdots \\
& \texttt{pif}\,\mathrm{left}(\mathbf{x}_k)\,\texttt{then}\,H_{2^k-1}(\mathrm{tail}_R(\mathbf{x}_1), \dots, \mathrm{tail}_L(\mathbf{x}_k), F(\mathrm{tail}_R(\mathbf{x}_1), \dots, \mathrm{tail}_L(\mathbf{x}_k))) \\
& \qquad \texttt{else}\,H_{2^k}(\mathrm{tail}_R(\mathbf{x}_1), \dots, \mathrm{tail}_R(\mathbf{x}_k), F(\mathrm{tail}_R(\mathbf{x}_1), \dots, \mathrm{tail}_R(\mathbf{x}_k)))
\end{aligned}$$

We write $F = Dyad(H_1, \dots, H_{k^2})$.

2. Simultaneous dyadic recursion *is defined analogously to the former cases.*

This scheme might be hard to read in the general case. In cases of one or two parameters things become clearer:

$$\begin{aligned}
F(\mathbf{x}) = {} & \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,H_L(\mathrm{tail}_L(\mathbf{x}), F(\mathrm{tail}_L(\mathbf{x})) \\
& \qquad \texttt{else}\,H_R(\mathrm{tail}_R(\mathbf{x}), F(\mathrm{tail}_R(\mathbf{x}))
\end{aligned}$$

$$\begin{aligned}
F(\mathbf{x}, \mathbf{y}) = {} & \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,H_{LL}(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}), F(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}))) \\
& \qquad \texttt{else}\,H_{LR}(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}), F(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}))) \\
& \texttt{else}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,H_{RL}(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}), F(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}))) \\
& \qquad \texttt{else}\,H_{RR}(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}), F(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{y})))
\end{aligned}$$

The definition of dyadic recursive functions should be obvious, we will present it in brief.

4.2.2 Definition

The basic functions are

1. *the* concatenations $\mathrm{cons}_L, \mathrm{cons}_C, \mathrm{cons}_R : \mathcal{I} \to \mathcal{I}$,
2. *the* projections $proj^k_i : \mathcal{I}^k \to \mathcal{I}, (\mathbf{x}_1, \dots, \mathbf{x}_k) \mapsto \mathbf{x}_i$ *and*
3. *the* constant function $\mathcal{I} \to \mathcal{I}, \mathbf{x} \to [0, 1]$

4.2.3 Definition

The class DR is the smallest set of functions that contains the basic functions and is closed under substitution and dyadic recursion.

As usual DR_n is the subset of those functions which need at most n nested dyadic recursions. The sets for simultaneous dyadic recursion SDR and SDR_n are defined analogously.

4.2.4 Lemma
Every dyadic recursive function is continuous.

PROOF. Follows from the continuity of the basic functions and the fixed point operator. ■

As usual we consider some examples for this kind of recursion.

4.2.5 Example
Some simple recursion schemes with one parameter and recursion depth one.

1. The identity

$$Id : \mathcal{I} \to \mathcal{I}, \mathbf{x} \mapsto \mathbf{x}$$

   has a very easy recursion scheme:

$$\begin{aligned} Id(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,&\mathrm{cons}_L(Id(\mathrm{tail}_L(\mathbf{x}))) \\ \texttt{else}\,&\mathrm{cons}_R(Id(\mathrm{tail}_R(\mathbf{x}))) \end{aligned}$$

2. The function

$$Mir : \mathcal{I} \to \mathcal{I}, [\xi, \xi'] \mapsto [1-\xi', 1-\xi]$$

   has a similar recursion scheme:

$$\begin{aligned} Mir(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,&\mathrm{cons}_R(Mir(\mathrm{tail}_L(\mathbf{x}))) \\ \texttt{else}\,&\mathrm{cons}_L(Mir(\mathrm{tail}_R(\mathbf{x}))) \end{aligned}$$

4.2.6 Example
The (binary) *mediation* or *average operator*

$$\oplus : \mathcal{I}^2 \to \mathcal{I}, ([\xi, \xi'], [\eta, \eta']) \mapsto \left[\frac{\xi+\eta}{2}, \frac{\xi'+\eta'}{2}\right]$$

can be defined by the following dyadic recursion scheme

$$\begin{aligned} \mathbf{x} \oplus \mathbf{y} = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,&\mathrm{cons}_L(\mathrm{tail}_L(\mathbf{x}) \oplus \mathrm{tail}_L(\mathbf{y})) \\ \texttt{else}\,&\mathrm{cons}_C(\mathrm{tail}_L(\mathbf{x}) \oplus \mathrm{tail}_R(\mathbf{y})) \\ \texttt{else}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,&\mathrm{cons}_C(\mathrm{tail}_R(\mathbf{x}) \oplus \mathrm{tail}_L(\mathbf{y})) \\ \texttt{else}\,&\mathrm{cons}_R(\mathrm{tail}_R(\mathbf{x}) \oplus \mathrm{tail}_R(\mathbf{y})) \end{aligned}$$

See [Esc96b].

4.2.7 Example
The ternary mediation operator $\oplus^3 : \mathcal{I}^3 \to \mathcal{I}, ([x,x'],[y,y'],[z,z']) \mapsto \left[\frac{x+y+z}{3}, \frac{x'+y'+z'}{3}\right]$

could be defined with a recursion scheme like

$$
\begin{aligned}
&\oplus^3(\mathbf{x},\mathbf{y},\mathbf{z}) = \\
&= \texttt{pif}\ \mathrm{left}(\mathbf{x})\ \texttt{then}\ \texttt{pif}\ \mathrm{left}(\mathbf{y})\ \texttt{then}\ \texttt{pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \mathrm{cons}_L(\oplus^3(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_L(\mathbf{z}))) \\
&\qquad\qquad \texttt{else}\ \mathrm{cons}_{[\frac{1}{6},\frac{2}{3}]}(\oplus^3(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_R(\mathbf{z}))) \\
&\qquad\qquad \vdots \\
&\qquad \texttt{else}\ \texttt{pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \mathrm{cons}_{[\frac{1}{3},\frac{5}{6}]}(\oplus^3(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z}))) \\
&\qquad\qquad \texttt{else}\ \mathrm{cons}_R(\oplus^3(\mathrm{tail}_R(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z})))
\end{aligned}
$$

Since this function is symmetric in the arguments, there is no need to write down all cases, actually there are four different cases, these are considered here.

This scheme uses the functions $\mathrm{cons}_{[\frac{1}{6},\frac{2}{3}]}$ and $\mathrm{cons}_{[\frac{1}{6},\frac{2}{3}]}$ which are not among our basic functions.

We use the following equations to define a simultaneous dyadic recursion scheme:

$$\left[\frac{1}{6},\frac{2}{3}\right]\cdot L = \left[\frac{1}{6},\frac{5}{12}\right] = L\cdot\left[\frac{1}{3},\frac{5}{6}\right]$$

$$\left[\frac{1}{6},\frac{2}{3}\right]\cdot\left[\frac{1}{6},\frac{2}{3}\right] = \left[\frac{1}{4},\frac{1}{2}\right] = L\cdot R$$

$$\left[\frac{1}{6},\frac{2}{3}\right]\cdot\left[\frac{1}{6},\frac{2}{3}\right] = \left[\frac{1}{3},\frac{7}{12}\right] = C\cdot\left[\frac{1}{6},\frac{2}{3}\right]$$

$$
\begin{aligned}
&\oplus^3(\mathbf{x},\mathbf{y},\mathbf{z}) = \\
&= \texttt{pif}\ \mathrm{left}(\mathbf{x})\ \texttt{then}\ \texttt{pif}\ \mathrm{left}(\mathbf{y})\ \texttt{then}\ \texttt{pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \mathrm{cons}_L(\oplus^3(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_L(\mathbf{z}))) \\
&\qquad\qquad \texttt{else}\ \oplus^3_{LLR}(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_R(\mathbf{z})) \\
&\qquad\qquad \vdots \\
&\qquad \texttt{else}\ \texttt{pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \oplus^3_{LRR}(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z})) \\
&\qquad\qquad \texttt{else}\ \mathrm{cons}_R(\oplus^3(\mathrm{tail}_R(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z})))
\end{aligned}
$$

$$\begin{aligned}
&\oplus^3_{LLR}(\mathbf{x},\mathbf{y},\mathbf{z}) = \\
&= \texttt{pif}\ \mathrm{left}(\mathbf{x})\ \texttt{then pif}\ \mathrm{left}(\mathbf{y})\ \texttt{then pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \mathrm{cons}_L(\oplus^3_{LRR}(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_L(\mathbf{z}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \mathrm{cons}_{LR}(\oplus^3(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_R(\mathbf{z}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \mathrm{cons}_C(\oplus^3_{LLR}(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \mathrm{cons}_C(\oplus^3_{LRR}(\mathrm{tail}_R(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z})))
\end{aligned}$$

$$\begin{aligned}
&\oplus^3_{LRR}(\mathbf{x},\mathbf{y},\mathbf{z}) = \\
&= \texttt{pif}\ \mathrm{left}(\mathbf{x})\ \texttt{then pif}\ \mathrm{left}(\mathbf{y})\ \texttt{then pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \mathrm{cons}_C(\oplus^3_{LRR}(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_L(\mathbf{z}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \mathrm{cons}_C(\oplus^3_{LRR}(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_L(\mathbf{y}),\mathrm{tail}_R(\mathbf{z}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else pif}\ \mathrm{left}(\mathbf{z})\ \texttt{then}\ \mathrm{cons}_{RL}(\oplus^3(\mathrm{tail}_L(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \mathrm{cons}_R(\oplus^3_{LLR}(\mathrm{tail}_R(\mathbf{x}),\mathrm{tail}_R(\mathbf{y}),\mathrm{tail}_R(\mathbf{z})))
\end{aligned}$$

This recursion scheme can understood as a finite transducer (see Figure 4.1). In this picture the intervals are interpreted as letters. The nodes or states represent the currently called function, which has to do the next step in the computation. The labels at the arrows denote the input/output behavior.

4.2.8 Example
We can define a function

$$T : \mathcal{I} \longrightarrow, [\xi, \xi'] \mapsto \left[\frac{1}{3}\cdot\xi, \frac{1}{3}\cdot\xi'\right]$$

as

$$T(\mathbf{x}) = \oplus^3(\mathbf{x}, [0,0], [0,0])$$

On the other hand we can derive a recursion scheme from the recursion scheme of $\oplus^3$ by picking the relevant cases. We obtain

$$\begin{aligned}
T(\mathbf{x}) = \texttt{pif}\ &\mathrm{left}(\mathbf{x})\ \texttt{then}\ \mathrm{cons}_L(T(\mathrm{tail}_L(\mathbf{x}))) \\
&\qquad\qquad \texttt{else}\ T_R(\mathrm{tail}_L(\mathbf{x}))
\end{aligned}$$

$$\begin{aligned}
T_R(\mathbf{x}) = \texttt{pif}\ &\mathrm{left}(\mathbf{x})\ \texttt{then}\ \mathrm{cons}_L(T_{RR}(\mathrm{tail}_L(\mathbf{x}))) \\
&\qquad\qquad \texttt{else}\ \mathrm{cons}_{LR}(T(\mathrm{tail}_R(\mathbf{x})))
\end{aligned}$$

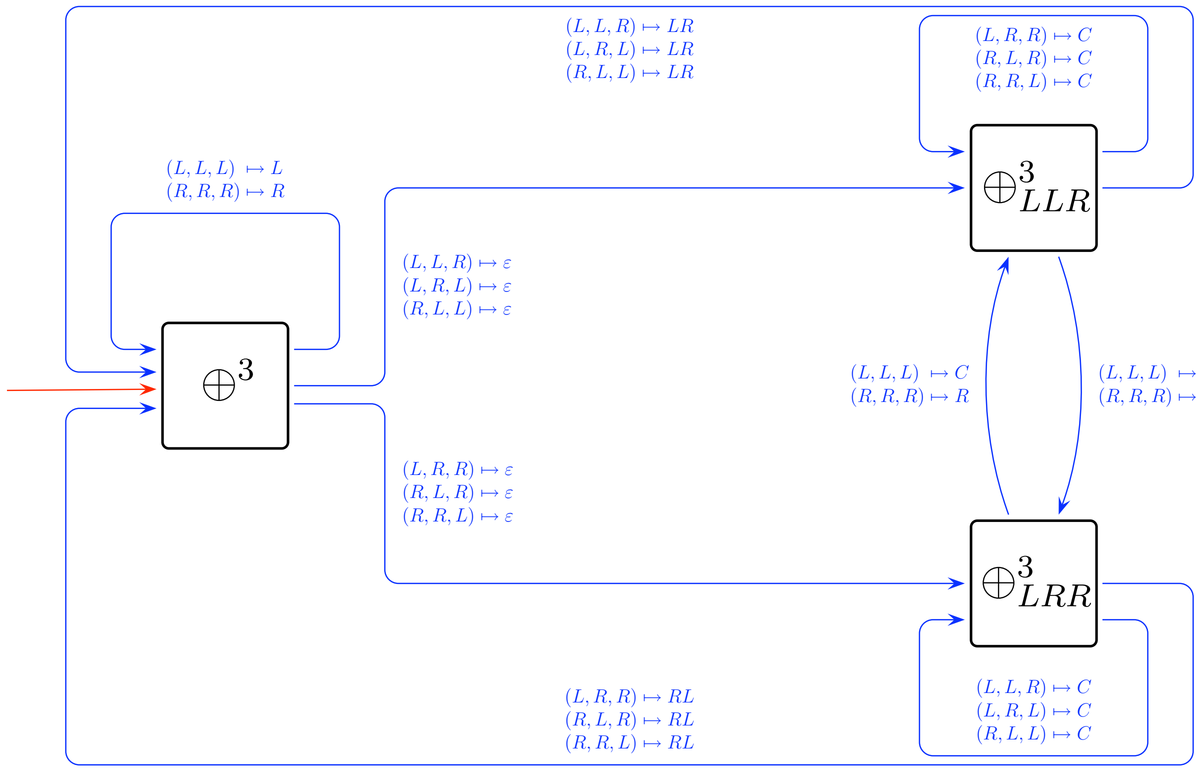


Figure 4.1: The ternary mediation operator as a finite transducer

$$\begin{aligned} T_{RR}(\mathbf{x}) = \texttt{pif}\, \mathrm{left}(\mathbf{x})\, \texttt{then}\, &\mathrm{cons}_C(T_R(\mathrm{tail}_L(\mathbf{x}))) \\ \texttt{else}\, &\mathrm{cons}_R(T_{RR}(tailR(\mathbf{x}))) \end{aligned}$$

This simultaneous recursion scheme can be written as a finite transducer again (Figure 4.2), which is obviously derived from the transducer representing the ternary mediation operator (Figure 4.1) by removing the irrelevant parts and keeping all relevant stuff.

Strange enough, there are recursion schemes for constant functions.

4.2.9 Example

1. The constant function

$$One : \mathcal{I} \to \mathcal{I}, \mathbf{x} \mapsto [1,1]$$

has the following dyadic recursion scheme:

$$\begin{aligned} One(\mathbf{x}) = \texttt{pif}\, \mathrm{left}(\mathbf{x})\, \texttt{then}\, &\mathrm{cons}_R(One(\mathrm{tail}_L(\mathbf{x}))) \\ \texttt{else}\, &\mathrm{cons}_R(One(\mathrm{tail}_R(\mathbf{x}))) \end{aligned}$$

2. The constant functions

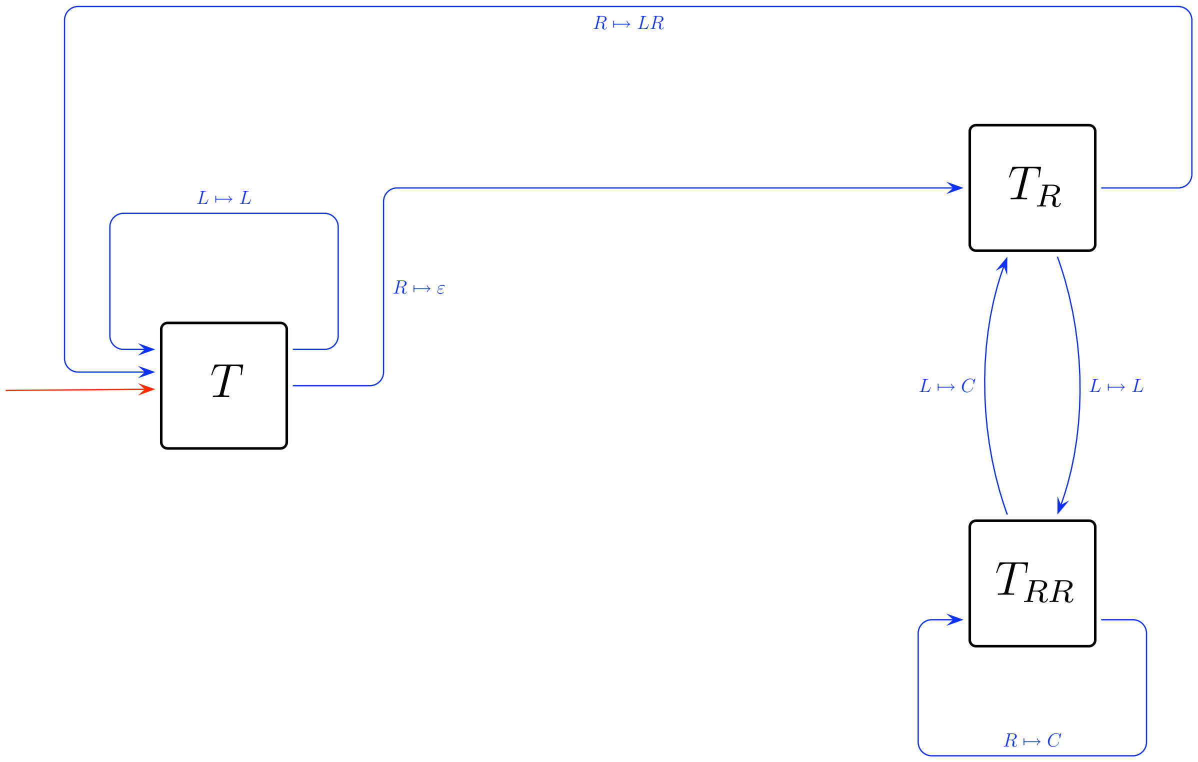


Figure 4.2: The function T as a finite transducer

$$Zero : \mathcal{I} \to \mathcal{I}, \mathbf{x} \mapsto [0, 0]$$

and

$$Ahalf : \mathcal{I} \to \mathcal{I}, \mathbf{x} \mapsto \left[\frac{1}{2}, \frac{1}{2}\right]$$

have similar recursion schemes.

3. The constant function

$$Athird : \mathcal{I} \to \mathcal{I}, \mathbf{x} \mapsto \left[\frac{1}{3}, \frac{1}{3}\right]$$

can be composed using the function T from Example 4.2.8

$$Athird(\mathbf{x}) = T(\tilde{1}, \tilde{0}, \tilde{0})$$

We might also give a simultaneous recursion scheme.

$$\begin{aligned} Athird(\mathbf{x}) = \texttt{pif}\, \text{left}(\mathbf{x})\, \texttt{then}\, \text{cons}_L(Athird'(\text{tail}_L(\mathbf{x}))) \\ \texttt{else}\, \text{cons}_L(Athird'(\text{tail}_R(\mathbf{x}))) \end{aligned}$$

and

$$\begin{aligned}\mathit{Athird}'(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,&\mathrm{cons}_R(\mathit{Athird}(\mathrm{tail}_L(\mathbf{x})))\\ \texttt{else}\,&\mathrm{cons}_R(\mathit{Athird}(\mathrm{tail}_R(\mathbf{x})))\end{aligned}$$

But we can also get rid of the simultaneity since this function is constant.

$$\begin{aligned}\mathit{Athird}(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,&\mathrm{cons}_{LR}(\mathit{Athird}(\mathrm{tail}_L(\mathbf{x})))\\ \texttt{else}\,&\mathrm{cons}_{LR}(\mathit{Athird}(\mathrm{tail}_R(\mathbf{x})))\end{aligned}$$

Our next example will be the multiplication which has recursion depth 2. Note that the recursion scheme given in [Esc96b] contains a small mistake. When multiplying two real numbers from the interval $\left[\frac{1}{2}, 1\right]$ the result will reside in the interval $\left[\frac{1}{4}, 1\right]$, which is too large to be represented by a digit. In this case, the next pair of digits has to be considered — smells like simultaneous recursion.

4.2.10 Example

1. Multiplication on intervals

$$\times : \mathcal{I}^2 \to \mathcal{I}, ([\xi, \xi'], [\eta, \eta']) \mapsto [\xi \cdot \eta, \xi' \cdot \eta']$$

can be computed using the following simultaneous dyadic recursion scheme

$$\begin{aligned}\mathbf{x} \times \mathbf{y} = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,&\mathrm{cons}_{LL}(\mathrm{tail}_L(\mathbf{x}) \times \mathrm{tail}_L(\mathbf{y}))\\ \texttt{else}\,&\mathrm{cons}_L\left(\frac{\mathrm{tail}_L(\mathbf{x}) + (\mathrm{tail}_L(\mathbf{x}) \times \mathrm{tail}_R(\mathbf{y}))}{2}\right)\\ \texttt{else}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,&\mathrm{cons}_L\left(\frac{(\mathrm{tail}_R(\mathbf{x}) \times \mathrm{tail}_L(\mathbf{y})) + \mathrm{tail}_L(\mathbf{y})}{2}\right)\\ \texttt{else}\,&\mathrm{tail}_R(\mathbf{x}) \times_{RR} \mathrm{tail}_R(\mathbf{y})\end{aligned}$$

and

$$\begin{aligned}\mathbf{x} \times_{RR} \mathbf{y} = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,&\mathrm{cons}_C\left(\frac{2 \cdot \mathrm{tail}_L(\mathbf{x}) + 2 \cdot \mathrm{tail}_L(\mathbf{y}) + (\mathrm{tail}_L(\mathbf{x}) \times \mathrm{tail}_R(\mathbf{y}))}{8}\right)\\ \texttt{else}\,&\mathrm{cons}_C\left(\frac{\tilde{2} + 3 \cdot \mathrm{tail}_L(\mathbf{x}) + 2 \cdot \mathrm{tail}_R(\mathbf{y}) + (\mathrm{tail}_R(\mathbf{x}) \times \mathrm{tail}_R(\mathbf{y}))}{8}\right)\\ \texttt{else}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,&\mathrm{cons}_C\left(\frac{\tilde{2} + 2 \cdot \mathrm{tail}_R(\mathbf{x}) + 3 \cdot \mathrm{tail}_L(\mathbf{y}) + (\mathrm{tail}_R(\mathbf{x}) \times \mathrm{tail}_R(\mathbf{y}))}{8}\right)\\ \texttt{else}\,&\mathrm{cons}_R\left(\frac{\tilde{1} + 3 \cdot \mathrm{tail}_R(\mathbf{x}) + 3 \cdot \mathrm{tail}_R(\mathbf{y}) + (\mathrm{tail}_R(\mathbf{x}) \times \mathrm{tail}_R(\mathbf{y}))}{8}\right)\end{aligned}$$

Terms like

$$\frac{\tilde{1} + 3 \cdot \mathrm{tail}_R(\mathbf{x}) + 3 \cdot \mathrm{tail}_R(\mathbf{y}) + (\mathrm{tail}_R(\mathbf{x}) \times \mathrm{tail}_R(\mathbf{y}))}{8}$$

can be understood as a short hand for terms like

$$\oplus^8 ([1,1], \mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}), \mathrm{tail}_R(\mathbf{y}), \mathrm{tail}_R(\mathbf{y}), \mathrm{tail}_R(\mathbf{x}) \times \mathrm{tail}_R(\mathbf{y}))$$

We explain this scheme. We have to consider equations like

$$\begin{aligned}
\mathrm{cons}_L(\mathbf{x}) \times \mathrm{cons}_L(\mathbf{x}) &= (L \cdot [\xi, \xi']) \times (L \cdot [\eta, \eta']) \\
&= \left[\frac{\xi}{2}, \frac{\xi'}{2}\right] \times \left[\frac{\eta}{2}, \frac{\eta'}{2}\right] \\
&= \left[\frac{\xi\eta}{4}, \frac{\xi'\eta'}{4}\right] \\
&= \left[0, \frac{1}{4}\right] \cdot [\xi\eta, \xi'\eta'] \\
&= \mathrm{cons}_{LL}(\mathbf{x} \times \mathbf{y})
\end{aligned}$$

or

$$\begin{aligned}
\mathrm{cons}_R(\mathrm{cons}_R(\mathbf{x})) \times \mathrm{cons}_R(\mathrm{cons}_R(\mathbf{y})) &= \left(\left[\frac{3}{4}, 1\right] \cdot [\xi, \xi']\right) \times \left(\left[\frac{3}{4}, 1\right] \cdot [\eta, \eta']\right) \\
&= \left[\frac{3+\xi}{4}, \frac{3+\xi'}{4}\right] \times \left[\frac{3+\eta}{4}, \frac{3+\eta'}{4}\right] \\
&= \left[\frac{9 + 3\xi + 3\eta + \xi\eta}{16}, \frac{9 + 3\xi' + 3\eta + \xi\eta'}{16}\right] \\
&= \left[\frac{1}{2}, 1\right] \cdot \left[\frac{1 + 3\xi + 3\eta + \xi\eta}{8}, \frac{1 + 3\xi' + 3\eta' + \xi'\eta'}{8}\right] \\
&= \mathrm{cons}_R(\oplus^8(\tilde{1}, \mathbf{x}, \mathbf{x}, \mathbf{x}, \mathbf{y}, \mathbf{y}, \mathbf{y}, \mathbf{x} \times \mathbf{y}))
\end{aligned}$$

We can derive a recursion scheme for the function

$$Sqr : \mathcal{I} \to \mathcal{I}, [\xi, \xi'] \mapsto [\xi^2, \xi'^2] \tag{4.1}$$

namely

$$\begin{aligned}
Sqr(\mathbf{x}) = \texttt{pif}\, \mathrm{left}(\mathbf{x})\, \texttt{then}\, &\mathrm{cons}_{LL}(Sqr(\mathrm{tail}_L(\mathbf{x})) \\
&\texttt{else}\, Sqr_R(\mathrm{tail}_R(\mathbf{x}))
\end{aligned}$$

and

$$\begin{aligned} Sqr_R(\mathbf{x}) = \mathtt{pif}\,\mathrm{left}(\mathbf{x})\,\mathtt{then}\,\mathrm{cons}_C & \left(\frac{4 \cdot \mathrm{tail}_L(\mathbf{x}) + Sqr(\mathrm{tail}_L(\mathbf{x}))}{8}\right) \\ \mathtt{else}\,\mathrm{cons}_R & \left(\frac{\tilde{1} + 6 \cdot \mathrm{tail}_R(\mathbf{x}) + Sqr(\mathbf{x})}{8}\right) \end{aligned}$$

4.3 Simulation by uniformly recursive word functions

The intervals $L = [0, \frac{1}{2}]$ and $R = [\frac{1}{2}, 1]$ can be interpreted as digits $\bar{1}$ and 1, respectively. Thus the behavior of a TTM simulating a dyadic recursive function is easily determined on inputs $w \in \{\bar{1}, 1\}^\omega$

The problem to consider is the digit 0, which has the interval $C = [\frac{1}{4}, \frac{3}{4}]$ as an equivalent, but that does not have a separate case in the dyadic recursion scheme. Whenever an interval I with $\frac{1}{2} \in I$ appears, the branches are computed in parallel.

First of all we note, that results of parallel computations behave properly, they are connected.

4.3.1 Lemma (Connection)
Let φ be a total real valued function computed by a dyadic recursive function F. Then for every interval $\mathbf{x} \in \mathcal{I}$ with $\frac{1}{2} \in \mathbf{x}$

$$F(\mathrm{tail}_L(\mathbf{x})) \cap F(\mathrm{tail}_R(\mathbf{x})) \neq \emptyset$$

Proof. Assume there is an interval $\mathbf{x}$

$$F(\mathrm{tail}_L(\mathbf{x})) \cap F(\mathrm{tail}_R(\mathbf{x})) = \emptyset$$

tail_L and tail_R are continuous and thus monotonic. Furthermore F is continuous and thus monotonic. We consider an interval $\mathbf{y}$ with $\mathbf{x} \sqsubseteq \mathbf{y}$ and $\frac{1}{2} \in \mathbf{y}$. Then we obtain

$$F(\mathrm{tail}_L(\mathbf{y})) \cap F(\mathrm{tail}_R(\mathbf{y})) \subseteq F(\mathrm{tail}_L(\mathbf{x})) \cap F(\mathrm{tail}_R(\mathbf{x})) = \emptyset$$

The definition of $\sqcap$ gives us that

$$width(F(\mathbf{y})) = width(F(\mathrm{tail}_L(\mathbf{y})) \sqcap F(\mathrm{tail}_R(\mathbf{y}))) > 0$$

with $width([\xi, \xi']) = \xi' - \xi$.

I.e. $F(\mathbf{y})$ cannot be a singleton interval. Considering the case $\mathbf{y} = [\frac{1}{2}, \frac{1}{2}]$ shows that $F([\frac{1}{2}, \frac{1}{2}])$ is not a singleton interval, which would lead to $\varphi(\frac{1}{2})\uparrow$. This is a contradiction, φ was assumed to be total. ■

4.3.2 Example
Consider the function F with the following dyadic recursion scheme

$$\begin{aligned} F(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\mathrm{then}\,\mathrm{cons}_{LL}(F(\mathrm{tail}_L(\mathbf{x}))) \\ \mathrm{else}\,\mathrm{cons}_{RR}(F(\mathrm{tail}_R(\mathbf{x}))) \end{aligned}$$

Each interval $\mathbf{x} = [\xi, \xi']$ with $\frac{1}{2} \in \mathbf{x}$ can be written as $\mathbf{x} = \left[\xi, \frac{1}{2}\right] \sqcap \left[\frac{1}{2}, \xi'\right]$. In this case we obtain

$$\begin{aligned} F(\mathbf{x}) &= \texttt{pif}\ \overbrace{\mathrm{left}(\mathbf{x})}^{\perp}\,\mathrm{then}\,\mathrm{cons}_{LL}(F(\mathrm{tail}_L(\mathbf{x}))) \\ &\qquad\qquad \mathrm{else}\,\mathrm{cons}_{RR}(F(\mathrm{tail}_R(\mathbf{x}))) \\ &\quad \mathrm{cons}_{LL}(F(\mathrm{tail}_L(\mathbf{x}))) \sqcap \mathrm{cons}_{RR}(F(\mathrm{tail}_R(\mathbf{x}))) \\ &= \mathrm{cons}_{LL}(F(\mathrm{tail}_L\left(\left[\xi, \frac{1}{2}\right]\right))) \sqcap \mathrm{cons}_{RR}(F(\mathrm{tail}_R\left(\left[\frac{1}{2}, \xi'\right]\right))) \end{aligned}$$

In case $\mathbf{x} = \left[\frac{1}{2}, \frac{1}{2}\right]$ we obtain

$$\begin{aligned} F\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right) &= \texttt{pif}\,\mathrm{left}\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right)\mathrm{then}\,\mathrm{cons}_{LL}(F(\mathrm{tail}_L\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right))) \\ &\qquad\qquad \mathrm{else}\,\mathrm{cons}_{RR}(F(\mathrm{tail}_R\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right))) \\ &= \mathrm{cons}_{LL}(F(\mathrm{tail}_L\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right))) \sqcap \mathrm{cons}_{RR}(F(\mathrm{tail}_R\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right))) \\ &= \mathrm{cons}_{LL}(F(\tilde{1}) \sqcap \mathrm{cons}_{RR}(F(\tilde{0})) \\ &= \mathrm{cons}_{LL}(\tilde{1}) \sqcap \mathrm{cons}_{RR}(\tilde{0}) \\ &= \frac{\tilde{1}}{4} \sqcap \frac{\tilde{3}}{4} \\ &= \left[\frac{1}{4}, \frac{3}{4}\right] \end{aligned}$$

In case of a dyadic rational, i.e $\mathbf{x} = [\frac{i}{2^j}, \frac{i}{2^j}]$ with an odd number i and $i < 2^j$, we can write $\mathbf{x} = I_1 \ldots I_j \cdot [\frac{1}{2}, \frac{1}{2}]$ with $I_1, \ldots I_j \in \{L, R\}$. Then

$$F(\mathbf{x}) = I_1^2 \cdot \ldots \cdot I_j^2 \cdot C$$

I.e. the computed function $\varphi : [0,1] \rightharpoonup [0,1]$ does not terminate on dyadic rationals.

We found that results of parallel computations are connected, i.e. they have a non-empty intersection. We use this property to merge the parallel outputs. We have to think about simulation of the infimum on intervals by a function on words.

4.3.1 The Infimum

Dyadic recursion depends on a certain parallelism, the parallel conditional `pif`. It is necessary in all cases, in which an interval cannot be determined to reside either in the right or in the left half of the unit interval. In these cases, the parallel conditional computes two branches and takes the infimum of both results as its result.

A similar situation occurs when we simulate dyadic recursion on intervals by primitive or uniform recursion on words. The letter or digit "0" corresponds to the interval C which needs parallelism. We have to simulate the computation of the infimum of the two respective branches.

In this section we explain the principle of a word function that computes the infimum (or rather the join) of the intervals denoted by two words. We provide the basic knowledge of dealing with overlapping intervals.

Since both branches of a parallel are connected, there are pairs of values which might not occur as results of a parallel computation. The feasible cases lead us directly to a definition of an infimum operator on infinite words.

Our first construction assumes that we have two infinite words converging to the same real number.

We consider how the intervals are connected in several cases and what we already know about the approximated real number. We will consider up to three consequent digits. We use these to make the intervals become smaller by adding more digits. Then the infimum becomes smaller as well, and we can determine a digit representation for it.

The cases where the two input intervals turn out to have no intersection occur when one of the branches is no more valid. We call them *non-fitting* situations. We will consider these cases in the next Section 4.3.2. The infimum operator will experience several refinements to handle those situations.

In Figure 4.3 the positions of intervals are shown. In case both intervals are the same, everything is clear. In the other cases, we have to consider the subsequent digit. This is done in Figure 4.5 and Figure 4.4. In some cases even a third digit has to be considered, this is shown in Figures 4.6 and 4.7. All other cases are symmetric variants of these.

That gives us the following simultaneous recursion scheme for the infimum.

4.3.3 Definition
We define the functions $inf, inf_{LR}, inf_{LC}, inf_{CR}, inf_{LC/CC}, inf_{CR/CC} : (\Sigma^*)^2 \to \Sigma^*$ by the following simultaneous recursion scheme:

$$\begin{aligned}
inf(\varepsilon, w) &= \varepsilon \\
inf(v, \varepsilon) &= \varepsilon \\
inf(av, aw) &= a \cdot inf(v, w) \\
inf(\bar{1}v, 1w) &= inf_{LR}(v, w)
\end{aligned}$$

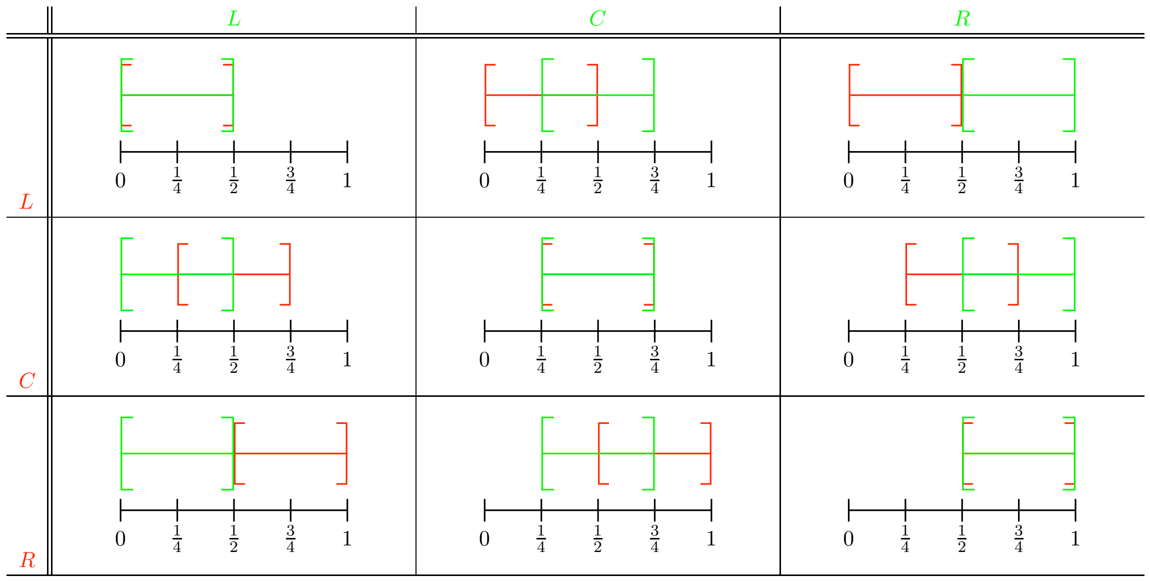


Figure 4.3: Positions of pairs of intervals

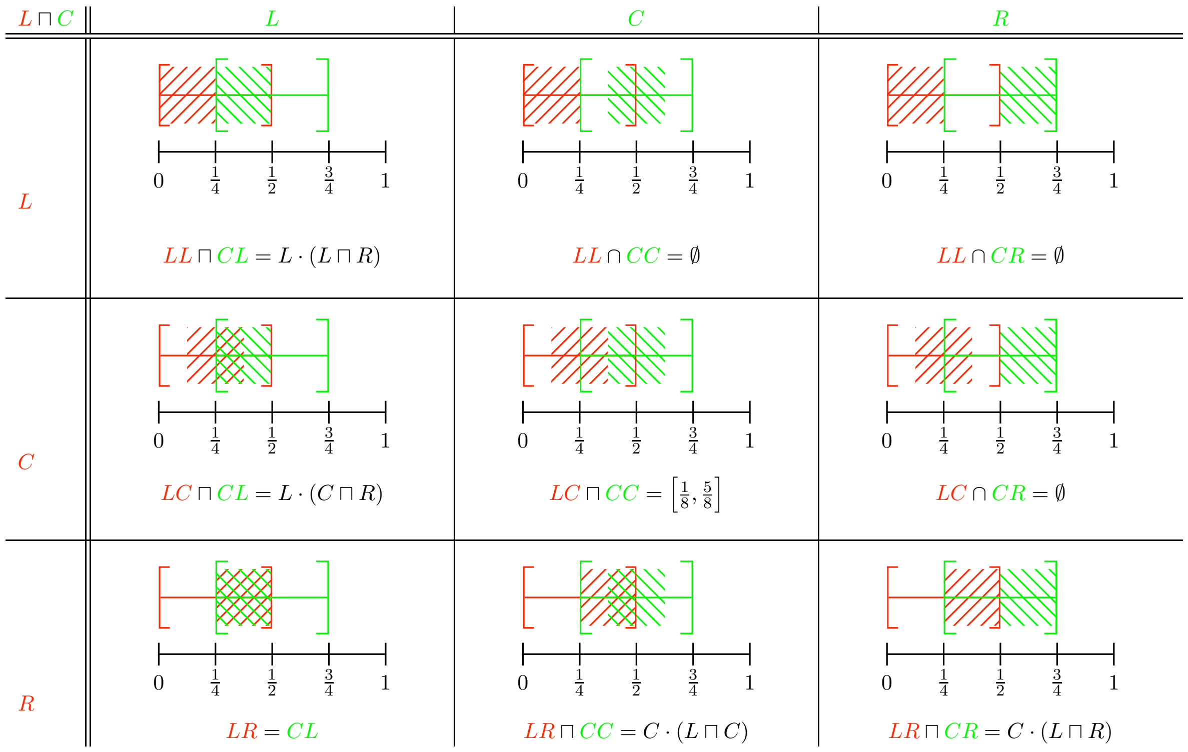

$L \sqcap C$
L
C
R
L
C
R
0 $\frac{1}{4}$ $\frac{1}{2}$ $\frac{3}{4}$ 1
$LL \sqcap CL = L \cdot (L \sqcap R)$
$LL \cap CC = \emptyset$
$LL \cap CR = \emptyset$
$LC \sqcap CL = L \cdot (C \sqcap R)$
$LC \sqcap CC = \left[\frac{1}{8}, \frac{5}{8}\right]$
$LC \cap CR = \emptyset$
$LR = CL$
$LR \sqcap CC = C \cdot (L \sqcap C)$
$LR \sqcap CR = C \cdot (L \sqcap R)$

Figure 4.4: Positions of intervals. Assume the first pair of intervals was (L, C).

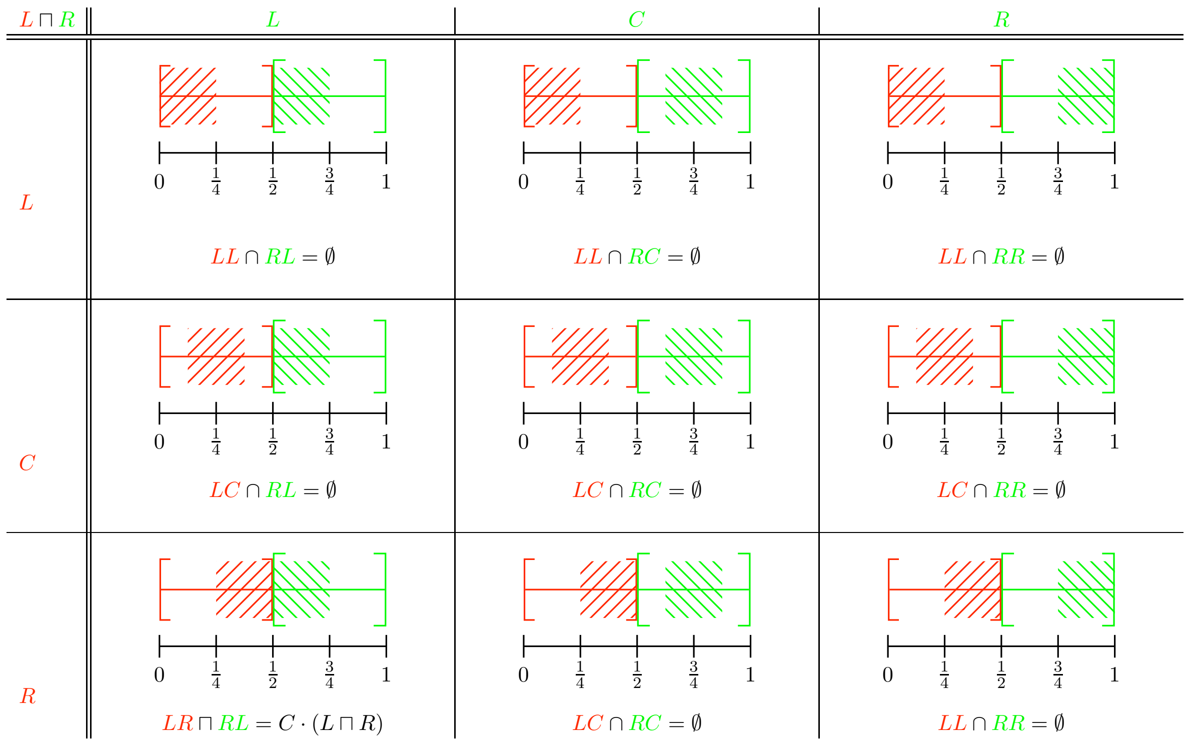


Figure 4.5: Positions of intervals. Assume the first pair of intervals was (L, R).

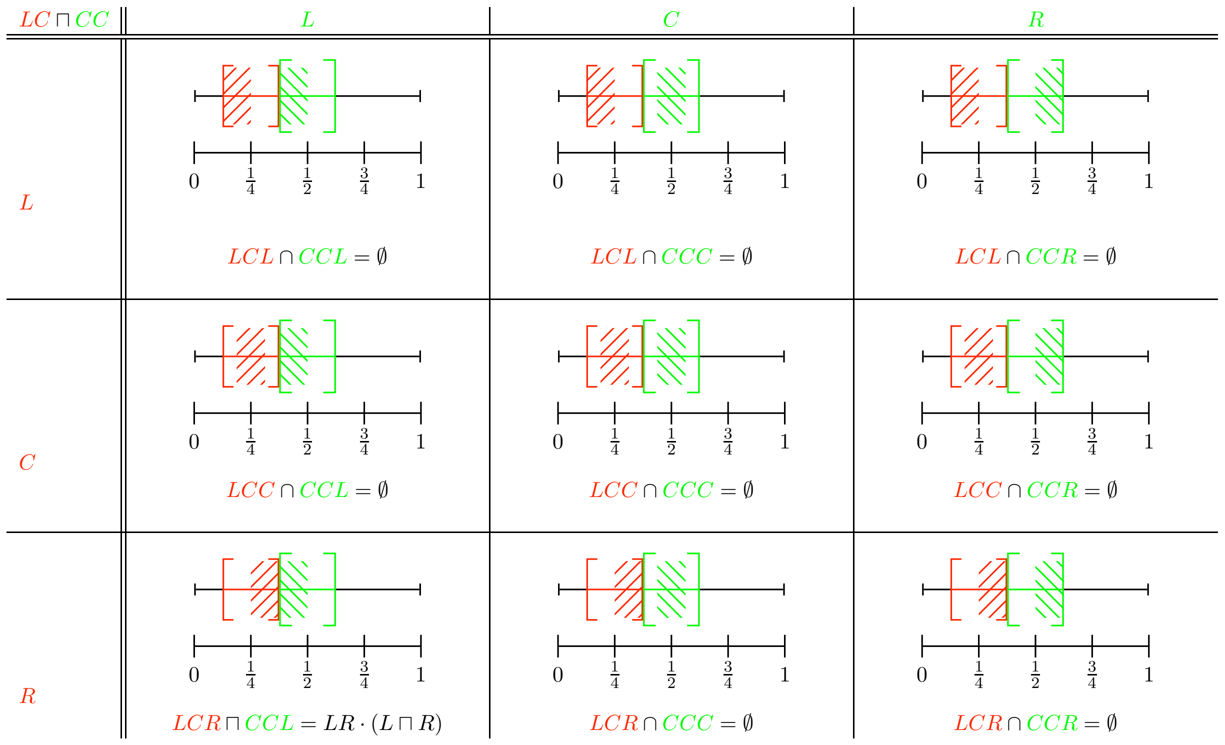
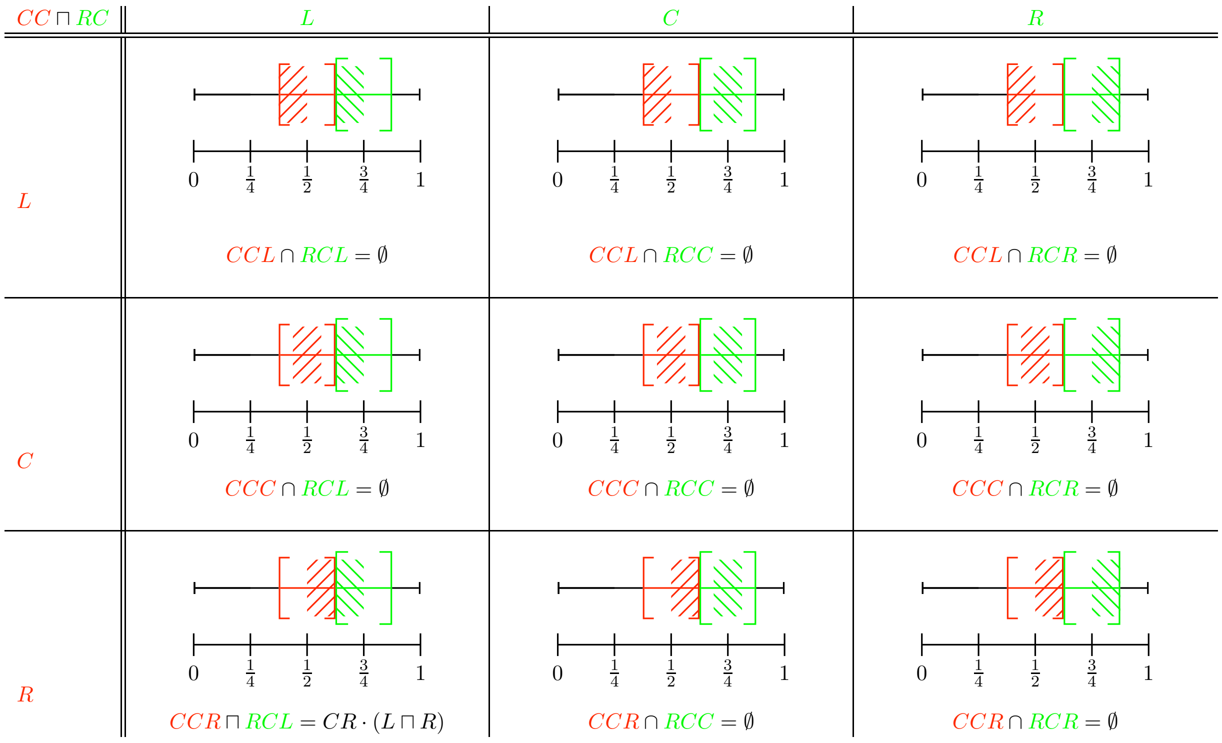

$LC \sqcap CC$
L
C
R
L
$LCL \cap CCL = \emptyset$
$LCL \cap CCC = \emptyset$
$LCL \cap CCR = \emptyset$
C
$LCC \cap CCL = \emptyset$
$LCC \cap CCC = \emptyset$
$LCC \cap CCR = \emptyset$
R
$LCR \sqcap CCL = LR \cdot (L \sqcap R)$
$LCR \cap CCC = \emptyset$
$LCR \cap CCR = \emptyset$
0 $\frac{1}{4}$ $\frac{1}{2}$ $\frac{3}{4}$ 1

Figure 4.6: Positions of intervals. Assume the first pair of intervals was (LC, CC).



$CC \sqcap RC$
L
C
R
L
C
R
0
$\frac{1}{4}$
$\frac{1}{2}$
$\frac{3}{4}$
1
$CCL \cap RCL = \emptyset$
$CCL \cap RCC = \emptyset$
$CCL \cap RCR = \emptyset$
$CCC \cap RCL = \emptyset$
$CCC \cap RCC = \emptyset$
$CCC \cap RCR = \emptyset$
$CCR \sqcap RCL = CR \cdot (L \sqcap R)$
$CCR \cap RCC = \emptyset$
$CCR \cap RCR = \emptyset$

Figure 4.7: Positions of intervals. Assume the first pair of intervals was (CC, RC).

$$
\begin{aligned}
inf(\overline{1}v, 0w) &= inf_{LC}(v, w)\\
inf(0v, \overline{1}w) &= inf_{LC}(w, v)\\
inf(0v, 1w) &= inf_{CR}(v, w)\\
inf(1v, \overline{1}w) &= inf_{LR}(w, v)\\
inf(1v, 0w) &= inf_{CR}(w, v)
\end{aligned}
$$

$$
\begin{aligned}
inf_{LC}(\varepsilon, v) &= \varepsilon\\
inf_{LC}(v, \varepsilon) &= \varepsilon\\
inf_{LC}(\overline{1}v, \overline{1}w) &= \overline{1} \cdot inf_{LR}(v, w)\\
inf_{LC}(\overline{1}v, 0w) &= \varepsilon\\
inf_{LC}(\overline{1}v, 1w) &= \varepsilon\\
inf_{LC}(0v, \overline{1}w) &= \overline{1} \cdot inf_{CR}(v, w)\\
inf_{LC}(0v, 0w) &= inf_{LC/CC}(v, w)\\
inf_{LC}(0v, 1w) &= \varepsilon\\
inf_{LC}(1v, \overline{1}w) &= \overline{1}1 \cdot inf(v, w)\\
inf_{LC}(1v, 0w) &= 0 \cdot inf_{LC}(v, w)\\
inf_{LC}(1v, 1w) &= 0 \cdot inf LR(v, w)
\end{aligned}
$$

$$
\begin{aligned}
inf_{LR}(\varepsilon, w) &= \varepsilon\\
inf_{LR}(v, \varepsilon) &= \varepsilon\\
inf_{LR}(\overline{1}v, aw) &= \varepsilon\\
inf_{LR}(0v, aw) &= \varepsilon\\
inf_{LR}(1v, \overline{1}w) &= 0 \cdot inf_{LR}(v, w)\\
inf_{LR}(1v, 0w) &= \varepsilon\\
inf_{LR}(1v, 1w) &= \varepsilon
\end{aligned}
$$

$$\begin{array}{rcl}
\mathit{inf}_{CR}(\varepsilon, w) & = & \varepsilon \\
\mathit{inf}_{CR}(v, \varepsilon) & = & \varepsilon \\
\mathit{inf}_{CR}(\bar{1}v, \bar{1}w) & = & 0 \cdot \mathit{inf}_{LR}(v, w) \\
\mathit{inf}_{CR}(\bar{1}v, 0w) & = & \varepsilon \\
\mathit{inf}_{CR}(\bar{1}v, 1w) & = & \varepsilon \\
\mathit{inf}_{CR}(0v, 1w) & = & 0 \cdot \mathit{inf}_{CR}(v, w) \\
\mathit{inf}_{CR}(0v, 0w) & = & \mathit{inf}_{CR/CC}(v, w) \\
\mathit{inf}_{CR}(0v, 1w) & = & \varepsilon \\
\mathit{inf}_{CR}(1v, \bar{1}w) & = & 01 \cdot \mathit{inf}(v, w) \\
\mathit{inf}_{CR}(1v, 0w) & = & 1 \cdot \mathit{inf}_{LC}(v, w) \\
\mathit{inf}_{CR}(1v, 1w) & = & 1 \cdot \mathit{inf}_{LR}(v, w)
\end{array}$$

Two cases need a third step

$$\begin{array}{rr}
\mathit{inf}_{LC/CC}(v, \varepsilon) = \varepsilon & \mathit{inf}_{CR/CC}(v, \varepsilon) = \varepsilon \\
\mathit{inf}_{LC/CC}(\varepsilon, w) = \varepsilon & \mathit{inf}_{CR/CC}(\varepsilon, w) = \varepsilon \\
\mathit{inf}_{LC/CC}(\bar{1}v, aw) = \varepsilon & \mathit{inf}_{CR/CC}(\bar{1}v, aw) = \varepsilon \\
\mathit{inf}_{LC/CC}(0v, aw) = \varepsilon & \mathit{inf}_{CR/CC}(0v, aw) = \varepsilon \\
\mathit{inf}_{LC/CC}(1v, \bar{1}w) = 10 \cdot \mathit{inf}_{LR}(v, w) & \mathit{inf}_{CR/CC}(1v, \bar{1}w) = 10 \cdot \mathit{inf}_{LR}(v, w) \\
\mathit{inf}_{LC/CC}(1v, 0w) = \varepsilon & \mathit{inf}_{CR/CC}(1v, 0w) = \varepsilon \\
\mathit{inf}_{LC/CC}(1v, 1w) = \varepsilon & \mathit{inf}_{CR/CC}(1v, 1w) = \varepsilon
\end{array}$$

It is obvious, that in the diverging cases — in the non-fitting situations — the output will not be continued, i.e. this function is not fully monotonic. The case, that one of the branches of the parallel computation is invalid, is left unconsidered yet.

We will deal with these cases in the following section.

4.3.2 Examples

The simulation of the dyadic recursion by uniform recursion follows a rather straight idea of simulating two parallel computations and merging their respective results with the infimum operator.

The technical implementation of this idea has to handle a bunch of cases that might occur. We decided to show representative examples for those cases, each displaying a problem and the principle of its solution. All these cases will be reconsidered in the next section, where a general construction is derived from the principles. We hope that will help to understand the construction.

Two cases will occur. When both computations are valid, both will converge to representations of the same result, and the infimum operator will deliver a representation of that result.

If, within the computation, one of the branches occurs to be invalid, this branch is stopped and only the other one is continued. The infimum operator must dismiss the invalid branch.

We will explain that in the following examples. The construction in the general case, which will be presented in the next Section (4.3.3), is based on these examples.

$(\overline{1}1)^\omega$ is the representation of $\frac{1}{3}$ generated by the dyadic scheme of the identity:

$$\left[\frac{1}{3},\frac{1}{3}\right] = \bigsqcup_{n\in\mathbb{N}} (\mathrm{cons}_{LR}^{n}([0,1]))$$

We have $\mathrm{left}(\left[\frac{1}{3},\frac{1}{3}\right]) = \mathbf{tt}$ and $\mathrm{tail}_L(\left[\frac{1}{3},\frac{1}{3}\right]) = \left[\frac{2}{3},\frac{2}{3}\right]$. And analogously $\mathrm{left}(\left[\frac{2}{3},\frac{2}{3}\right]) = \mathbf{ff}$ and $\mathrm{tail}_R(\left[\frac{2}{3},\frac{2}{3}\right]) = \left[\frac{1}{3},\frac{1}{3}\right]$

The advantage of this representation becomes clear, on computing a dyadic recursive function, e.g. $\mathit{Mir} : \mathbf{x} \mapsto 1-\mathbf{x}$ (Example 4.2.5-2, p. 73).

That recursion scheme can easily be translated to ε-primitive recursion

$$\begin{aligned}
\mathit{mir}(\varepsilon) &= \varepsilon \\
\mathit{mir}(\overline{1}w) &= 1\cdot \mathit{mir}(w) \\
\mathit{mir}(1w) &= \overline{1}\cdot \mathit{mir}(w)
\end{aligned}$$

We obtain $\mathit{mir}((\overline{1}1)^l) = (1\overline{1})^l$ and $\mathit{mir}^\omega((\overline{1}1)^\omega) = (1\overline{1})^\omega$.

A less canonical representation of $\frac{1}{3}$ is $0\overline{1}(\overline{1}1)^\omega$. To compute $1-\frac{1}{3}$ with this representation the above recursion scheme is insufficient. We do not have a case $\mathit{mir}(0w) = \ldots$. We remind

$$C = LR \sqcap RL \tag{4.2}$$

or rather

$$\begin{aligned}
C^iL &= LR^i \\
C^iR &= RL^i
\end{aligned}$$

and

$$\bigsqcup_{i\in\mathbb{N}} C^i = \bigsqcup_{i\in\mathbb{N}} LR^i = \bigsqcup_{i\in\mathbb{N}} RL^i \tag{4.3}$$

We define

$$\begin{aligned} \mathit{left}(\varepsilon) &= \varepsilon & \mathit{right}(\varepsilon) &= \varepsilon \\ \mathit{left}(\bar{1}v) &= 1v & \mathit{right}(\bar{1}v) &= \Diamond \\ \mathit{left}(0v) &= 1 \cdot \mathit{left}(v) & \mathit{right}(0v) &= 1 \cdot \mathit{right}(v) \\ \mathit{left}(1v) &= \Diamond & \mathit{right}(1v) &= 1v \end{aligned}$$

In this scheme we use an additional digit $\Diamond$. This represents the detection of an invalid branch. E.g. the line $\mathit{left}(1v) = \Diamond$ denotes that a digit 1 may not occur in a left branch.

A $\Diamond$-digit within an input words makes a function know that it deals with an invalid branch.

Since we do have that additional digit $\Diamond$ in our alphabet $\Sigma = \{\bar{1}, 0, 1\}$, we have to simulate it by a simultaneous recursion. This can be done analogously to Lemma 3.2.11 (p. 32).

Now we have to complete the recursion schemes of the infimum operator in the obvious way

$$\begin{aligned} \mathit{inf}(av, \Diamond w) &= av \\ \mathit{inf}(\Diamond v, aw) &= aw \\ \\ \mathit{inf}_{LC}(av, \Diamond w) &= \bar{1} \cdot av \\ \mathit{inf}_{LC}(\Diamond v, aw) &= 0 \cdot aw \\ \\ \mathit{inf}_{LR}(av, \Diamond w) &= \bar{1} \cdot av \\ \mathit{inf}_{LR}(\Diamond v, aw) &= 1 \cdot aw \\ \\ \mathit{inf}_{CR}(av, \Diamond w) &= 0 \cdot av \\ \mathit{inf}_{LR}(\Diamond v, aw) &= 1 \cdot aw \\ \\ \mathit{inf}_{LC/CC}(av, \Diamond w) &= \bar{1}0 \cdot av \\ \mathit{inf}_{LC/CC}(\Diamond v, aw) &= 00 \cdot aw \\ \\ \mathit{inf}_{CR/CC}(av, \Diamond w) &= 00 \cdot av \\ \mathit{inf}_{CR/CC}(\Diamond v, aw) &= 10 \cdot aw \end{aligned}$$

A line like $\mathit{inf}(av, \Diamond w) = av$ shows how invalid branches are handled. The right branch is invalid, it is ignored. Only the left branch delivers the result.

We complete the recursion scheme by

$$\mathit{mir}(0w) = \mathit{inf}(1 \cdot \mathit{mir}(\mathit{left}(v)), \bar{1} \cdot \mathit{mir}(\mathit{right}(v)))$$

The principle of uniform or primitive recursion does only allow the use of preceding values, like $mir(v)$ to compute $mir(av)$. The use of $mir(left(v))$is not allowed.

We overcome this with further simultaneous recursion.

$$mir(0w) = inf(1 \cdot mir_L(w), \overline{1} \cdot mir_R(w))$$

$$\begin{array}{rcl} mir_L(\overline{1}v) & = & \overline{1} \cdot mir(v) \\ mir_L(0v) & = & \overline{1} \cdot mir_L(v) \\ mir_L(1v) & = & \Diamond \end{array} \qquad \begin{array}{rcl} mir_R(\overline{1}v) & = & \Diamond \\ mir_R(0v) & = & 1 \cdot mir_R(v) \\ mir_R(1v) & = & 1 \cdot mir(v) \end{array}$$

Since mir now uses $inf \in \varepsilon\text{-}UR_1(\Sigma)$ in its recursion scheme, we have $mir \in \varepsilon\text{-}UR_2(\Sigma)$. That is insufficient, we need $mir \in \varepsilon\text{-}UR_1(\Sigma)$. We cannot use the infimum operator within the recursion scheme. We have to use an *external infimum operator* inf^e.

Our aim is to compute mir in the following way:

$$mir(x) = inf^e(mir'(left_e(x)), mir'(r(x)))$$

where $left_e$ and $right_e$ are the external versions of $left$ and $right$. They are defined analogously, but computation will be continued after a $\Diamond$-digit. $\Diamond$ works like a reset, which cuts of the invalid branch and uses it for a new one if necessary.

4.3.4 Definition
The functions $left_e, right_e : \{\overline{1}, 0, 1\} \to \{\overline{1}, 0, 1, \Diamond\}$ *are defined by the following recursion schemes*

$$\begin{array}{rcl} left_e(\varepsilon) & = & \varepsilon \\ left_e(\overline{1}v) & = & \overline{1} \cdot left_e(v) \\ left_e(0v) & = & \overline{1} \cdot left'_e(v) \\ left_e(1v) & = & \Diamond \cdot left_e(v) \end{array} \qquad \begin{array}{rcl} right_e(\varepsilon) & = & \varepsilon \\ right_e(\overline{1}v) & = & \Diamond \cdot right_e(v) \\ right_e(0v) & = & 1 \cdot right'_e(v) \\ right_e(1v) & = & 1 \cdot right_e(v) \end{array}$$

$$\begin{array}{rcl} left'_e(\varepsilon) & = & \varepsilon \\ left'_e(\overline{1}v) & = & 1 \cdot left_e(v) \\ left'_e(0v) & = & 1 \cdot left'_e(v) \\ left'_e(1v) & = & \Diamond \cdot left_e(v) \end{array} \qquad \begin{array}{rcl} right'_e(\varepsilon) & = & \varepsilon \\ right'_e(\overline{1}v) & = & \Diamond \cdot right_e(v) \\ right'_e(0v) & = & \overline{1} \cdot right'_e(v) \\ right'_e(1v) & = & \overline{1} \cdot right_e(v) \end{array}$$

4.3.5 Definition
inf^e *is defined like* inf *with the following changes in the scheme:*

$$\begin{aligned}
\mathit{inf}^e(\Diamond v, aw) &= a \cdot \mathit{inf}^e(v,w) \\
\mathit{inf}^e(av, \Diamond w) &= a \cdot \mathit{inf}^e(v,w)
\end{aligned}$$

$$\begin{aligned}
\mathit{inf}^e_{LR}(\Diamond v, aw) &= 1a \cdot \mathit{inf}^e(v,w) \\
\mathit{inf}^e_{LR}(av, \Diamond w) &= \bar{1}a \cdot \mathit{inf}^e(v,w)
\end{aligned}$$

Further parts of the recursion schemes are modified analogously.

We reconsider our example.

4.3.6 Example

$$\mathit{mir}(v) = \mathit{inf}^e(\mathit{mir}'(\mathit{left}_e(v)), \mathit{mir}'(\mathit{right}_e(v)))$$

where mir' is defined in the obvious way

$$\begin{aligned}
\mathit{mir}'(\varepsilon) &= \varepsilon \\
\mathit{mir}'(\bar{1}v) &= 1 \cdot \mathit{mir}'(v) \\
\mathit{mir}'(1v) &= \bar{1}\mathit{mir}'(v) \\
\mathit{mir}'(\Diamond v) &= \Diamond \cdot \mathit{mir}'(v)
\end{aligned}$$

Note that we have no rule for the case $\mathit{mir}'(0v)$. We do not need it. mir' will not be called with these input values. Any definition in this case is sufficient.

$$\begin{aligned}
\mathit{mir}(100010\bar{1}) &= \mathit{inf}^e(\mathit{mir}'(l(100010\bar{1}), \mathit{mir}'(\mathit{right}_e(100010\bar{1})))) \\
&= \mathit{inf}^e(\mathit{mir}'(\Diamond\bar{1}11\Diamond\bar{1}1), (\mathit{mir}'(11\bar{1}\bar{1}\bar{1}1\Diamond)) \\
&= \mathit{inf}^e(\Diamond 1\bar{1}\bar{1}\Diamond 1\bar{1}, \bar{1}\bar{1}111\bar{1}\Diamond) \\
&= \bar{1} \cdot \mathit{inf}^e(1\bar{1}\bar{1}\bar{1}\Diamond 1\bar{1}, \bar{1}\bar{1}1111\bar{1}\Diamond) \\
&= \bar{1} \cdot \mathit{inf}^e_{RL}(\bar{1}\bar{1}\Diamond 1\bar{1}, 111\bar{1}\Diamond) \\
&= \bar{1}0 \cdot \mathit{inf}^e_{RL}(\bar{1}\Diamond 1\bar{1}, 11\bar{1}\Diamond) \\
&= \bar{1}00 \cdot \mathit{inf}^e_{RL}(\Diamond 1\bar{1}, 1\bar{1}\Diamond) \\
&= \bar{1}00\bar{1}1 \cdot \mathit{inf}^e(1\bar{1}, \bar{1}\Diamond) \\
&= \bar{1}00\bar{1}1 \cdot \mathit{inf}^e_{RL}(\bar{1}, \Diamond) \\
&= \bar{1}00\bar{1}11\bar{1}
\end{aligned}$$

The previous example is mostly harmless, things become more complicated with more input parameters.

4.3.7 Example
We repeat the dyadic recursion scheme of the mediation operator

$$\begin{aligned}
\mathbf{x} \oplus \mathbf{y} = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,\mathrm{cons}_L(\mathrm{tail}_L(\mathbf{x}) \oplus \mathrm{tail}_L(\mathbf{y})) && (4.4)\\
\texttt{else}\,\mathrm{cons}_C(\mathrm{tail}_L(\mathbf{x}) \oplus \mathrm{tail}_R(\mathbf{y})) && (4.5)\\
\texttt{else}\,\texttt{pif}\,\mathrm{left}(\mathbf{y})\,\texttt{then}\,\mathrm{cons}_C(\mathrm{tail}_R(\mathbf{x}) \oplus \mathrm{tail}_L(\mathbf{y})) && (4.6)\\
\texttt{else}\,\mathrm{cons}_C(\mathrm{tail}_R(\mathbf{x}) \oplus \mathrm{tail}_R(\mathbf{y})) && (4.7)
\end{aligned}$$

That gives us

$$\begin{aligned}
med(\varepsilon, w) &= \varepsilon\\
med(v, \varepsilon) &= \varepsilon\\[1em]
med(\bar{1}v, \bar{1}w) &= \bar{1} \cdot med(v, w)\\
med(\bar{1}v, 1w) &= 0 \cdot med(v, w)\\
med(1v, \bar{1}w) &= 0 \cdot med(v, w)\\
med(1v, 1w) &= 1 \cdot med(v, w)\\[1em]
med(\bar{1}v, 0w) &= inf(\bar{1} \cdot med(\bar{1}med(v, left(w)), 0med(v, right(w)))\\
med(0v, \bar{1}w) &= inf(\bar{1} \cdot med(left(v), w), 0 \cdot med(right(v), w))\\[1em]
med(0v, 0w) &= inf(inf(\bar{1} \cdot med(left(v), left(w)), 0med(left(v), right(w))),\\
&\qquad inf(0 \cdot med(right(v), left(w)), 1 \cdot med(right(v), right(w))))\\
med(0v, 1w) &= inf(0 \cdot med(left(v), w), 1 \cdot med(right(v), w))\\
med(1v, 0w) &= inf(0 \cdot med(v, left(w)), 1 \cdot med(v, right(w)))\\[1em]
med(v, \Diamond w) &= \Diamond\\
med(\Diamond v, w) &= \Diamond
\end{aligned}$$

With the external infimum operator inf^e we obtain

$$\begin{aligned}
med = inf^e(&inf^e(med'(left_e(v), left_e(w)), med'(left_e(v), right_e(w))),\\
&inf^e(med'(right_e(v), left_e(w)), med'(right_e(v), right_e(w))))
\end{aligned}$$

where the definition of med' is derived from the definition of $\oplus$ in the obvious way.

4.3.8 Example
A further example is the function T (Example 4.2.8) which uses simultaneous recursion.

We have two problems to handle when using the external infimum inf^e. The first is that the output is not synchronized, i.e. we do not have the same number of output digits in every branch. In particular, reading a single input digit may produce none, one or two output digits.

$$t(x) = \mathit{inf}^e(t'(l(x), \mathit{right}_e(x)), t'(\mathit{right}_e(x), l(x)))$$

Note that the function t' has two parameters. We need the second parameter, to check which state the other branch of a parallel computation has. We need to know that, in case a branch is discontinued. Then we know the snap back position.

The lower index records the state of the current branch, the upper index the state of the other one.

We have to record snap backs in the upper index, i.e setting the upper index to the next lower index, when a $\Diamond$ occurs in the second argument.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
t'(\overline{1}v, \Diamond w) & = & \overline{1} \cdot t'(v,w) & t'_R(\overline{1}v, 1w) & = & \overline{1} \cdot t^R_{RR}(v,w) \\
t'(\overline{1}v, 1w) & = & \overline{1} \cdot t^R(v,w) & t'_R(\overline{1}v, \Diamond w) & = & \overline{1} \cdot t^{RR}_{RR}(v,w) \\
t'(1v, \overline{1}w) & = & t'_R(v,w) & t'_R(1v, \overline{1}w) & = & \overline{1}1 \cdot t'(v,w) \\
t'(1v, \Diamond w) & = & t'_R(v,w) & t'_R(1v, \Diamond w) & = & \overline{1}1 \cdot t'(v,w) \\
t'(\Diamond v, \overline{1}w) & = & \Diamond \cdot t'(v,w) & t'_R(\Diamond v, \overline{1}w) & = & \Diamond\Diamond \cdot t'(v,w) \\
t'(\Diamond v, 1w) & = & \Diamond \cdot t^R_R & t'_R(\Diamond v, 1w) & = & \Diamond \cdot t^R_R(v,w)
\end{array}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
t^R(\overline{1}v, \Diamond w) & = & \overline{1} \cdot t'(v,w) & t^R_R(\overline{1}v, 1w) & = & \overline{1} \cdot t'_{RR}(v,w) \\
t^R(\overline{1}v, 1w) & = & \overline{1} \cdot t'(v,w) & t^R_R(\overline{1}v, \Diamond w) & = & \overline{1} \cdot t^{RR}_{RR}(v,w) \\
t^R(1v, \overline{1}w) & = & t^{RR}_R(v,w) & t^R_R(1v, \overline{1}w) & = & \overline{1}1 \cdot t^{RR}(v,w) \\
t^R(1v, \Diamond w) & = & t^R_R(v,w) & t^R_R(1v, \Diamond w) & = & \overline{1}1 \cdot t'(v,w) \\
t^R(\Diamond v, \overline{1}w) & = & \Diamond \cdot t^{RR}_{RR}(v,w) & t^R_R(\Diamond v, \overline{1}w) & = & \Diamond\Diamond \cdot t'(v,w) \\
t^R(\Diamond v, 1w) & = & \Diamond \cdot t'(v,w) & t^R_R(\Diamond v, 1w) & = & \Diamond \cdot t^R_R
\end{array}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
t^{RR}(\overline{1}v, \Diamond w) & = & \overline{1} \cdot t'(v,w) & t^{RR}_R(\overline{1}v, 1w) & = & \overline{1} \cdot t'_{RR}(v,w) \\
t^{RR}(\overline{1}v, 1w) & = & \overline{1} \cdot t^{RR}(v,w) & t^{RR}_R(\overline{1}v, \Diamond w) & = & \overline{1} \cdot t^{RR}_{RR}(v,w) \\
t^{RR}(1v, \overline{1}w) & = & t^R_R(v,w) & t^{RR}_R(1v, \overline{1}w) & = & \overline{1}1 \cdot t'(v,w) \\
t^{RR}(1v, \Diamond w) & = & t^R_R(v,w) & t^{RR}_R(1v, \Diamond w) & = & \overline{1}1 \cdot t'(v,w) \\
t^{RR}(\Diamond v, \overline{1}w) & = & \Diamond \cdot t^R_R(v,w) & t^{RR}_R(\Diamond v, \overline{1}w) & = & \Diamond \cdot t^R_R(v,w) \\
t^{RR}(\Diamond v, 1w) & = & \Diamond \cdot t^{RR}_{RR}(v,w) & t^{RR}_R(\Diamond v, 1w) & = & \Diamond \cdot t^{RR}_{RR}(v,w)
\end{array}$$

$$\begin{array}{rcl}
t'_{RR}(\bar{1}v, 1w) & = & 0 \cdot t^R_R(v,w) \\
t'_{RR}(\bar{1}v, \Diamond w) & = & 0 \cdot t^R_R(v,w) \\
t'_{RR}(1v, \bar{1}w) & = & 0 \cdot t'_{RR}(v,w) \\
t'_{RR}(1v, \Diamond w) & = & 0 \cdot t^{RR}_{RR}(v,w) \\
t'_{RR}(\Diamond v, \bar{1}w) & = & \Diamond\Diamond \cdot t'(v,w) \\
t'_{RR}(\Diamond v, 1w) & = & \Diamond \cdot t^R_R(v,w)
\end{array}$$

$$\begin{array}{rcl}
t^R_{RR}(\bar{1}v, 1w) & = & 0 \cdot t'_R(v,w) \\
t^R_{RR}(\bar{1}v, \Diamond w) & = & 0 \cdot t^R_R(v,w) \\
t^R_{RR}(1v, \bar{1}w) & = & 0 \cdot t^{RR}_{RR}(v,w) \\
t^R_{RR}(1v, \Diamond w) & = & 0 \cdot t^{RR}_{RR}(v,w) \\
t^R_{RR}(\Diamond v, \bar{1}w) & = & \Diamond\Diamond \cdot t'(v,w) \\
t^R_{RR}(\Diamond v, 1w) & = & \Diamond \cdot t^R_R
\end{array}$$

$$\begin{array}{rcl}
t^{RR}_{RR}(\bar{1}v, 1w) & = & 0 \cdot t^{RR}_R(v,w) \\
t^{RR}_{RR}(\bar{1}v, \Diamond w) & = & 0 \cdot t^R_R(v,w) \\
t^{RR}_{RR}(1v, \bar{1}w) & = & 0 \cdot t^R_{RR}(v,w) \\
t^{RR}_{RR}(1v, \Diamond w) & = & 0 \cdot t^{RR}_{RR}(v,w) \\
t^{RR}_{RR}(\Diamond v, \bar{1}w) & = & \Diamond \cdot t^R_R(v,w) \\
t^{RR}_{RR}(\Diamond v, 1w) & = & \Diamond \cdot t^{RR}_{RR}(v,w)
\end{array}$$

Consider the line $t'(\Diamond v, 1w) = \Diamond t^R$, which tells to remember that the right branch switches to state T_R.

Note that we have two $\Diamond$-digits for t'_R and t'_{RR}. This is necessary to synchronize the output streams. While switching from state T to T_R, no output is produced, i.e. that stream is one digit behind. This becomes a problem, when a stream stops and we snap back to the current state of the continued stream.

Simultaneous recursion with use of the external infimum needs to consider the two branches in their interdependency. In both branches we mix up valid and invalid computations. Whenever a computation is detected to be invalid we have to continue that branch from the current state of the other branch. This state is stored in the upper index of the respective function.

Note that we have to remember the offset between the two branches to resynchronize the values. This might be difficult in some case. We recommend something else. The following construction will work in general. We ensure the synchronization of the output by adding additional $\Diamond$-digits. Note that the functions t', t'_R and t'_{RR} produce no, one or two output digits for every input digit.

E.g. we could have lines like the following in our recursion scheme:

$$\begin{array}{rcl} t'(\bar{1}v, \Diamond w) & = & \bar{1}\Diamond \cdot t'(v,w) \\ & \cdots & \\ t^{RR}(\Diamond v, 1w) & = & \Diamond\Diamond \cdot t^{RR}_{RR}(v,w) \\ & \cdots & \end{array}$$

I.e. all output is synchronized. We need a *synchronized infimum* to get rid of additional $\Diamond$-symbols.

4.3.9 Definition
The synchronized infimum inf^s *is defined analogously to the external infimum, except the following case*

$$\mathit{inf}^s(\Diamond v, \Diamond w) = \varepsilon \cdot \mathit{inf}^s(v,w)$$

and analogously for inf^s_{LC} *and all other states of the scheme.*

That will not work with recursion depth two. We consider the function

$$\mathit{Sqr} : \mathcal{I} \to \mathcal{I}, [\xi, \xi'] \mapsto [\xi^2, (\xi')^2]$$

from Example 4.2.10-4.1 (p. 79). We repeat the dyadic recursion scheme:

$$\begin{array}{l} \mathit{Sqr}(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,\mathrm{cons}_{LL}(\mathit{Sqr}(\mathrm{tail}_L(\mathbf{x}) \\ \qquad\qquad\qquad\qquad\qquad \texttt{else}\,\mathit{Sqr}_R(\mathrm{tail}_R(\mathbf{x})) \end{array}$$

and

$$\begin{array}{l} \mathit{Sqr}_R(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\,\mathrm{cons}_C\left(\dfrac{4\cdot\mathrm{tail}_L(\mathbf{x}) + \mathit{Sqr}(\mathrm{tail}_L(\mathbf{x}))}{8}\right) \\ \qquad\qquad\qquad\qquad\qquad \texttt{else}\,\mathrm{cons}_R\left(\dfrac{\tilde{1} + 6\cdot\mathrm{tail}_R(\mathbf{x}) + \mathit{Sqr}(\mathbf{x})}{8}\right) \end{array}$$

It seems, the respective word function sqr could be computed in the following manner

$$\begin{array}{rcl} \mathit{sqr}(\bar{1}v) & = & \overline{11}\cdot \mathit{sqr}(v) \\ \mathit{sqr}(1v) & = & \mathit{sqr}_R(v) \\ \mathit{sqr}(0v) & = & \mathit{inf}(\overline{11}\cdot \mathit{sqr}(\mathit{left}(v)), \mathit{sqr}_R(\mathit{right}(v))) \\ \mathit{sqr}(\Diamond v) & = & \Diamond \end{array}$$

But this will not work. Consider the following computation.

4.3.10 Example

$$\begin{aligned} sqr(01^7) &= inf(\overline{1}\overline{1} \cdot sqr(left(1^7)), sqr_R(right(1^7))) \\ &= inf(\overline{1}\overline{1} \cdot sqr(\Diamond), sqr_R(\overline{1}1^5)) \\ &= inf(\overline{1}\overline{1}\Diamond, 001) \\ &= inf_{LC}(\overline{1}\Diamond, 01) \end{aligned}$$

The last function call cannot be answered successfully, the infimum operator will try to compute the infimum of $LL = \left[0, \frac{1}{4}\right]$ and $CC = \left[\frac{3}{8}, \frac{5}{8}\right]$. But $LL \cap CC = \emptyset$. We call this a *non-fitting situation.*

This is caused by the $\Diamond$-digit in the invalid branch coming too late. The left branch of a computation of this function produces two digits of output for every input digit, while the right branch produces a half output digit for every input digit. The right branch is slower, the branches are not synchronized.

We can overcome this problem by a more clever infimum operator. We add an additional state, which means we need a further function simultaneously defined with the infimum operator. This function is used whenever a situation occurs, in which the actual intervals will not fit.

This function *fit* will not be fully monotonic. But this fact does not cause problems. The function is used under secure conditions and the resulting infimum operator will be fully monotonic.

Actually we will call *fit* only if necessary. But we let it run in as a first pass, then let the infimum operator run as a second pass. It can use the results of *fit* if necessary.

If no situation occurs, in which fitting is not possible, the infimum operator does not use the information of *fit*. Note, in these situations *fit* delivers a value that allows the infimum operator to continue.

We give a recursion scheme for fit, at least in principle.

$$\begin{aligned}
\mathit{fit}(a_i v, a_i w) &= a_3 \cdot \mathit{fit}(v, w) && (4.8)\\
\mathit{fit}(\overline{1}v, 0w) &= a_3 \cdot \mathit{fit}_{LC}(v, w) && (4.9)\\
\mathit{fit}(\overline{1}v, 1w) &= a_3 \cdot \mathit{fit}_{LR}(v, w) && (4.10)\\
&\vdots\\
\mathit{fit}_{LC}(\overline{1}v, \overline{1}w) &= a_3 \cdot \mathit{fit}_{LR}(v, w) && (4.11)\\
\mathit{fit}_{LC}(\overline{1}v, 0w) &= \mathit{fit}'(v, w) && (4.12)\\
&\vdots\\
\mathit{fit}'(av, bw) &= \mathit{fit}'(v, w) && (4.13)\\
\mathit{fit}'(av, \Diamond w) &= a_1 && (4.14)\\
\mathit{fit}'(\Diamond v, bw) &= a_2 && (4.15)
\end{aligned}$$

The function works as follows:

It starts with producing a dummy output, as long as no non-fitting situation occurs (lines 4.8 to 4.11). The same state transitions as in case of the function *inf* are applied, knowing the state is necessary to detect an non-fitting situation. During this only dummy output is produced, it has the same length as the input.

Whenever a non-fitting situation occurs (line 4.12), it is ensured, that one of the branches in invalid, no further output is produced (line 4.13) until the $\Diamond$-sign in the invalid branch is found (lines 4.14 and 4.15).

We give a more general recursion scheme. Remember $\rho(\overline{1}) = L$, $\rho(0) = C$ and $\rho(1) = R$.

4.3.11 Definition

$$\begin{aligned}
\mathit{fit}(av, aw) &= a_3 \cdot \mathit{fit}(v, w) && && (4.16)\\
\mathit{fit}(av, bw) &= a_3 \cdot \mathit{fit}_{\rho(a),\rho(b)}(v, w) && && (4.17)\\
\mathit{fit}_{IJ}(av, bw) &= a_3 \cdot \mathit{fit}_{I'J'}(v, w) && \textit{if } \mathit{inf}_{IJ}(av, bw) = u \cdot \mathit{inf}_{I'J'}(v, w) && (4.18)\\
\mathit{fit}_{IJ}(av, bw) &= \mathit{fit}'(v, w) && \textit{if } \mathrm{cons}_I(\rho(a)) \cap \mathrm{cons}_J(\rho(b)) = \emptyset && (4.19)\\
\mathit{fit}'(av, bw) &= \mathit{fit}'(v, w) && \textit{if } \{a, b\} \subseteq \{\overline{1}, 0, 1\} && (4.20)\\
\mathit{fit}'(av, \Diamond w) &= a_1 && && (4.21)\\
\mathit{fit}'(\Diamond v, bw) &= a_2 && && (4.22)
\end{aligned}$$

It is easy to see, that lines 4.17 and 4.18 accord with lines 4.8 to 4.11 from above, and further on.

And further we define a new infimum operator $\mathit{inf}^{(3)} : (\Sigma^*)^3 \to \Sigma^*$. Read $\mathit{inf}_{[0,1][0,1]}$ as *inf* and analogously for $\mathit{inf}^{(3)}$. Further let $\rho^{-1}(L) = \overline{1}$, $\rho^{-1}(LC) = \overline{1}0$ and so on.

4.3.12 Definition
We define the infimum $\mathit{inf} : (\Sigma^*)^2 \to \Sigma^*$*:*

1. *Let* $\mathit{inf}^{(3)} : (\Sigma^*)^3 \to \Sigma^*$ *defined by*

$$\begin{aligned}
\mathit{inf}^{(3)}_{IJ}(av, bw, cu) &= d \cdot \mathit{inf}^{(3)}_{I'J'} && \text{if } \mathit{inf}_{IJ} = d \cdot \mathit{inf}_{I'J'} \\
\mathit{inf}^{(3)}_{IJ}(av, bw, a_1 u) &= \rho^{-1}(I) \cdot av \\
\mathit{inf}^{(3)}_{IJ}(av, bw, a_2 u) &= \rho^{-1}(J) \cdot bw
\end{aligned}$$

with inf, inf_{LC} *etc. defined by as in Definition 4.3.3*

2. *Let* $\mathit{inf} : (\Sigma^*)^2 \to \Sigma^*$ *defined by*

$$\mathit{inf}(v, w) = \mathit{inf}^{(3)}(v, w, \mathit{fit}(v, w))$$

We use the same name *inf* for the new infimum, since it behaves like the old one defined in Definition 4.3.3. Since they behave equivalent in fitting cases, the new infimum can be understood as an extension of the old one to non-fitting cases.

We reconsider our Example 4.3.10 (page 95).

4.3.13 Example

$$\begin{aligned}
\mathit{sqr}(01^7) &= \mathit{inf}(\overline{1}\overline{1} \cdot \mathit{sqr}(\mathit{left}(1^7)), \mathit{sqr}_R(\mathit{right}(1^7))) \\
&= \mathit{inf}(\overline{1}\overline{1} \cdot \mathit{sqr}(\Diamond), \mathit{sqr}_R(\overline{1}1^6)) \\
&= \mathit{inf}(\overline{1}\overline{1}\Diamond, 001) \\
&= \mathit{inf}^{(3)}(\overline{1}\overline{1}\Diamond, 001, \mathit{fit}(\overline{1}\overline{1}\Diamond, 001)) \\
&= \mathit{inf}^{(3)}(\overline{1}\overline{1}\Diamond, 001, a_3 \cdot \mathit{fit}_{LC}(\overline{1}\Diamond, 01)) \\
&= \mathit{inf}^{(3)}(\overline{1}\overline{1}\Diamond, 001, a_3 \cdot \mathit{fit}'(\Diamond, 1)) \\
&= \mathit{inf}^{(3)}(\overline{1}\overline{1}\Diamond, 001, a_3 a_2) \\
&= \mathit{inf}^{(3)}(\overline{1}\overline{1}\Diamond, 001, a_3 a_2) \\
&= \mathit{inf}^{(3)}_{LC}(\overline{1}\Diamond, 01, a_2) \\
&= 001
\end{aligned}$$

4.3.3 General case

In our examples we found that in case of recursion depth one dyadic recursion is very strict, if we want to compute a total function. The intervals to be concatenated in both branches of the dyadic recursion have to be synchronized to ensure that overlapping intervals map to overlapping intervals.

4.3.14 Lemma (Synchronization)
Let $F : \mathcal{I} \times \mathcal{I} \to \mathcal{I} \in DR_1$ *be defined by the following dyadic recursion scheme:*

$$\begin{aligned}F(\mathbf{x},\mathbf{y}) = \texttt{pif}\,\text{left}(\mathbf{x})\,\texttt{then}\,\texttt{pif}\,\text{left}(\mathbf{y})\,\texttt{then}\,&\text{cons}_{p_1^{LL}\dots p_{k_{LL}}^{LL}}(\text{tail}_L(\mathbf{x}),\text{tail}_L(\mathbf{y}),F(\text{tail}_L(\mathbf{x}),\text{tail}_L(\mathbf{y})))\\ \texttt{else}\,&\text{cons}_{p_1^{LR}\dots p_{k_{LR}}^{LR}}(\text{tail}_L(\mathbf{x}),\text{tail}_R(\mathbf{y}),F(\text{tail}_L(\mathbf{x}),\text{tail}_R(\mathbf{y})))\\ \texttt{else}\,\texttt{pif}\,\text{left}(\mathbf{y})\,\texttt{then}\,&\text{cons}_{p_1^{RL}\dots p_{k_{RL}}^{RL}}(\text{tail}_R(\mathbf{x}),\text{tail}_L(\mathbf{y}),F(\text{tail}_R(\mathbf{x}),\text{tail}_L(\mathbf{y})))\\ \texttt{else}\,&\text{cons}_{p_1^{RR}\dots p_{k_{RR}}^{RR}}(\text{tail}_R(\mathbf{x}),\text{tail}_R(\mathbf{y}),F(\text{tail}_R(\mathbf{x}),\text{tail}_R(\mathbf{y})))\end{aligned}$$

with $\{p_1^{LL},\dots,p_{k_{LL}}^{LL},p_1^{LR},\dots,p_{k_{LR}}^{LR},p_1^{RL},\dots,p_{k_{RL}}^{RL},p_1^{RR},\dots,p_{k_{RR}}^{RR}\} \subseteq \{L,C,R\}$

Further let F compute the total function $\varphi : [0,1]\times[0,1] \to [0,1]$. If φ is not constant, then $k_{LL} = k_{LR} = k_{RL} = k_{RR} = 1$.

PROOF.

- "≥ 1"

  Assume $k_{LL} = 0$ then $F([0,0],[0,0]) = [0,1]$, i.e. φ is not total. Contradiction.

  Analogously for k_{LR}, k_{RL} k_{RR}.

- "≤ 1"

  This proof is similar to that of lemma 3.6.6 (page 63).

  Assume $k_{LL} \geq 1$.

  Then $\mathit{width}(F(L,L)) = 2^{-k_{LL}} < \frac{1}{2}$. Since φ is total, $f(L,L) \cap F(L,R) \neq \emptyset$. Further $\mathit{width}(F(L,L) \sqcap F(L,R)) = \tau < 1$.

  By induction we get $f(L^k,L^k) \sqcap f(L^k,R^k) = \tau^k$. This forces φ to be a constant function, which is a contradiction.

■

4.3.15 Remark
The synchronisation problem does not occur for constant functions. A dyadic recursive constant function can be assumed to defined with use of a recursion scheme in which the left and the right branch are the same. These are obviously synchronous.

This fact lets dyadic recursion schemes for constant functions easily be converted into ε-uniform recursion schemes. We need not care about constant functions any more.

4.3.16 Example
We consider the constant function

$$\mathit{Athird} : \mathcal{I} \to \mathcal{I}, \mathbf{x} \mapsto \left[\frac{1}{3},\frac{1}{3}\right]$$

with dyadic recursion scheme

$$\begin{aligned} \mathit{Athird}(\mathbf{x}) = \texttt{pif}\, \mathrm{left}(\mathbf{x})\, \texttt{then}\, & \mathrm{cons}_{LR}(\mathit{Athird}(\mathrm{tail}_L(\mathbf{x}))) \\ \texttt{else}\, & \mathrm{cons}_{LR}(\mathit{Athird}(\mathrm{tail}_R(\mathbf{x}))) \end{aligned}$$

Obviously this function can be simulated by a word function

$$\mathit{athird} : \{\bar{1}, 0, 1\}^* \to \{\bar{1}, 0, 1\}^*$$

with recursion scheme

$$\begin{aligned} \mathit{athird}(\varepsilon) &= \varepsilon \\ \mathit{athird}(av) &= \bar{1}1 \cdot \mathit{athird}(v) \end{aligned}$$

Note that in case of simultaneous dyadic recursion the synchronization is not that clear. In order to be able to use the external infimum, we have to synchronize the output digits manually. Details can be found in the following proofs.

4.3.17 Theorem
Let $F : \mathcal{I}^k \to \mathcal{I} \in DR_n$ be a function that computes the total function $\varphi : [0,1]^k \to [0,1]$, then a word function $f : (\{\bar{1}, 0, 1\}^*)^k \to \{\bar{1}, 0, 1\}^* \in \varepsilon\text{-}SUR_n(\Sigma)$ exists, which computes φ as well.

Proof.

We use the technique developed in the examples in Section 4.3.2.

Let $n \in \mathbb{N}$ and $F : \mathcal{I}^k \to \mathcal{I} \in DR_n$ We define f by structural induction. W.l.o.g. let $k = 2$.

- F is a basic function. Obvious.
- $F = \mathit{Dyad}(H_{LL}, H_{LR}, H_{RL}, H_{RR})$, $H_{LL}, H_{LR}, H_{RL}, H_{RR} \in DR_{n-1}$, $n = 1$
  Due to Lemma 4.3.14 F has the following dyadic recursion scheme:

$$\begin{aligned} F : \mathcal{I} \times \mathcal{I} \to \mathcal{I} : (\mathbf{x}, \mathbf{y}) \mapsto \texttt{pif}\, \mathrm{left}(\mathbf{x})\, \texttt{then}\, \texttt{pif}\, \mathrm{left}(\mathbf{y})\ & \texttt{then}\, \mathrm{cons}_{p^{LL}}(F(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}))) \\ & \texttt{else}\, \mathrm{cons}_{p^{LR}}(F(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}))) \\ \texttt{else}\, \texttt{pif}\, \mathrm{left}(\mathbf{y})\ & \texttt{then}\, \mathrm{cons}_{p^{RL}}(F(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}))) \\ & \texttt{else}\, \mathrm{cons}_{p^{RR}}(F(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}))) \end{aligned}$$

  with $\{p^{LL}, p^{LR}, p^{RL}, p^{RR}\} \subseteq \{L, C, R\}$
  We define the functions

$$\begin{aligned} f(v,w) = inf^e(&inf^e(f'(left_e(v), left_e(w)), f'(left_e(v), right_e(w))),\\ &inf^e(f'(right_e(v), left_e(w)), f'(right_e(v), right_e(w)))) \end{aligned}$$

with inf^e, $left_e$ and $right_e$ defined as in Definitions 4.3.4 and 4.3.5.

and

$$\begin{aligned} f'(\varepsilon, w) &= \varepsilon \\ f'(v, \varepsilon) &= \varepsilon \\ f'(\bar{1}v, \bar{1}w) &= dig(p_{LL}) \cdot f'(v,w) \\ f'(\bar{1}v, 1w) &= dig(p_{LR}) \cdot f'(v,w) \\ f'(1v, \bar{1}w) &= dig(p_{RL}) \cdot f'(v,w) \\ f'(1v, 1w) &= dig(p_{RR}) \cdot f'(v,w) \end{aligned}$$

with

$$dig : \{L, C, R\} \to \{\bar{1}, 0, 1\}, \begin{cases} L \mapsto \bar{1} \\ C \mapsto 0 \\ R \mapsto 1 \end{cases}$$

- $F = Dyad(H_{LL}, H_{LR}, H_{RL}, H_{RR})$ with $H_{LL}, H_{LR}, H_{RL}, H_{RR} \in DR_{n-1}$, $n > 1$

$$\begin{aligned} &F : \mathcal{I} \times \mathcal{I} \to \mathcal{I} : (\mathbf{x}, \mathbf{y}) \mapsto \\ &\quad \mapsto \texttt{pif}\ \mathrm{left}(\mathbf{x})\ \texttt{then}\ \texttt{pif}\ \mathrm{left}(\mathbf{y})\ \texttt{then}\ H_{LL}(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}), F(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}))) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\ \mathrm{else}\ H_{LR}(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}), F(\mathrm{tail}_L(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}))) \\ &\qquad\qquad\quad \mathrm{else}\ \texttt{pif}\ \mathrm{left}(\mathbf{y})\ \texttt{then}\ H_{RL}(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}), F(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_L(\mathbf{y}))) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\ \mathrm{else}\ H_{RR}(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}), F(\mathrm{tail}_R(\mathbf{x}), \mathrm{tail}_R(\mathbf{y}))) \end{aligned}$$

with $H_{LL}, H_{LR}, H_{RL}, H_{RR} \in DR_{n-1}$.

If all functions H_{LL}, H_{LR}, H_{RL} and H_{RR} compute total real valued functions then, by induction hypothesis, there are word functions h_{LL}, h_{LR}, h_{RL} and h_{RR} computing the same functions.

If one of the functions is not fully monotonic, we can use the same construction with the following consideration:

Since F computes a total real valued function for all sequences of intervals $([\xi, \xi'])_{i \in \mathbb{N}}$ converging to a real number the sequence $(F([\xi, \xi']))_{i \in \mathbb{N}}$ will converge to a real number as well.

Assume the function, say H_{LL}, does not converge on some sequences. Then at least H_{LL} delivers a value that when used by one of the other functions will not prevent them from converging, since F must converge.

All used functions are continuous and therefore monotonic. I.e. replacing the problematic interval by a subinterval will result in the same function value of F, since the real numbers are the maximal elements of $\mathcal{I}$. We can extend the output of H_{LL} arbitrarily by extending the respective recursion scheme.

In other words, we can assume that all functions H_{LL}, H_{LR}, H_{RL} and H_{RR} compute total real valued functions, or rather, we can assume the word functions h_{LL}, h_{LR}, h_{RL} and h_{RR} to be fully monotonic.

$$\begin{array}{rcl}
f(\varepsilon, w) & = & \varepsilon \\
f(v, \varepsilon) & = & \varepsilon \\
f(\bar{1}v, \bar{1}w) & = & h_{LL}(v, w, h_{LL}(v, w)) \\
f(\bar{1}v, 0w) & = & \mathit{inf}(h_{LL}(v, \mathit{left}(w), f_{\_,\mathit{left}}(v, w)), h_{LR}(v, \mathit{right}(w), f_{\_,\mathit{right}}(v, w))) \\
f(\bar{1}v, 1w) & = & h_{LR}(f(v, w)) \\
f(0v, \bar{1}w) & = & \mathit{inf}(h_{LL}(\mathit{left}(v), w, f_{\mathit{left},\_}(v, w)), h_{RL}(\mathit{right}(v), w, f_{\mathit{right},\_}(v, w)) \\
f(0v, 0w) & = & \mathit{inf}(\mathit{inf}(h_{LL}(\mathit{left}(v), \mathit{left}(w), f_{\mathit{left},\mathit{left}}(v, w)), h_{LR}(\mathit{left}(v), \mathit{right}(w), f_{\mathit{left},\mathit{right}}(v, w))), \\
& & \quad \mathit{inf}(h_{RL}(\mathit{right}(v), \mathit{left}(w), f_{\mathit{right},\mathit{left}}(v, w)), h_{RR}(\mathit{right}(v), \mathit{right}(w), f_{\mathit{right},\mathit{right}}(v, w)))) \\
f(0v, 1w) & = & \mathit{inf}(h_{LL}(\mathit{left}(v), w, f_{\mathit{left},\_}(v, w)), h_{RL}(\mathit{right}(v), w, f_{\mathit{left},\_}(v, w))) \\
f(1v, \bar{1}w) & = & h_{RL}(f(v, w))
\end{array}$$

with

$$\begin{array}{rcl}
f_{\_,\mathit{left}}(\varepsilon, w) & = & \varepsilon \\
f_{\_,\mathit{left}}(v, \varepsilon) & = & \varepsilon \\
f_{\_,\mathit{left}}(\bar{1}v, \bar{1}w) & = & h_{LR}(v, w, f(v, w)) \\
f_{\_,\mathit{left}}(\bar{1}v, 0) & = & h_{LR}(v, \mathit{left}(w), f_{\_,\mathit{left}}(v, w)) \\
f_{\_,\mathit{left}}(\bar{1}v, 1w) & = & \Diamond \\
f_{\_,\mathit{left}}(0v, \bar{1}w) & = & \mathit{inf}(h_{LR}(v, w, f(v, w)), h_{LR}(v, w, f(v, w))) \\
f_{\_,\mathit{left}}(0v, 0w) & = & \mathit{inf}(h_{LR}(left(v), left(w), f_{\mathit{left},\mathit{left}}(v, w)), h_{RR}(right(v), left(w), f_{\mathit{right},\mathit{left}}(v, w))) \\
f_{\_,\mathit{left}}(0v, 1w) & = & \Diamond \\
f_{\_,\mathit{left}}(0v, 1w) & = & \Diamond \\
f_{\_,\mathit{left}}(1v, \bar{1}w) & = & h_{RR}(v, w, f(v, w)) \\
f_{\_,\mathit{left}}(1v, 0) & = & h_{RR}(v, \mathit{left}(w), f_{\_,\mathit{left}}(v, w)) \\
f_{\_,\mathit{left}}(1v, 1w) & = & \Diamond
\end{array}$$

The functions $f_{\_,\mathit{right}}$, $f_{\mathit{left},\_}$ and $f_{\mathit{right},\_}$ are defined analogously.

As a further detail of that recursion scheme we give:

$$\begin{aligned}
f_{\mathit{left,left}}(\varepsilon, w) &= \varepsilon \\
f_{\mathit{left,left}}(v, \varepsilon) &= \varepsilon \\
f_{\mathit{left,left}}(\overline{1}v, \overline{1}w) &= h_{RR}(v, w, f(v, w)) \\
f_{\mathit{left,left}}(\overline{1}v, 0w) &= h_{RR}(v, \mathit{left}(w), f_{\_,\mathit{left}}(v, w)) \\
f_{\mathit{left,left}}(0v, \overline{1}w) &= h_{RR}(\mathit{left}(v), w, f_{\_,\mathit{left}}(v, w)) \\
f_{\mathit{left,left}}(0v, 0w) &= h_{RR}(v, \mathit{left}(w), f_{\mathit{left,left}}(v, w)) \\
f_{\mathit{left,left}}(av, 1w) &= \Diamond \\
f_{\mathit{left,left}}(1v, aw) &= \Diamond \\
f_{\mathit{left,left}}(av, \Diamond w) &= \Diamond \\
f_{\mathit{left,left}}(\Diamond v, aw) &= \Diamond
\end{aligned}$$

The functions $f_{\mathit{left,right}}$, $f_{\mathit{left,right}}$ and $f_{\mathit{right,right}}$ are defined analogously.

Let *inf* be defined as in Definition 4.3.12

- $F = Sub(G; \overline{H})$. By induction hypothesis there are functions $g, \overline{h} \in \varepsilon\text{-}SUR_n(\Sigma)$, let $f = Sub(g; \overline{h})$

  W.l.o.g. we may assume all functions $g, h_1, \ldots, h_k$ to be fully monotonic. The consideration is similar to the previous case.

■

4.3.18 Corollary
Let $F : \mathcal{I}^k \to \mathcal{I} \in SDR_n$ $(n \geq 0)$ be a function that computes the total function $\varphi : [0,1]^k \to [0,1]$. Then a word function $f : (\{\overline{1}, 0, 1\}^*)^k \to \{\overline{1}, 0, 1\}^* \in \varepsilon\text{-}SUR_n(\Sigma)$ exists, which computes φ as well.

Proof. We have to reconsider the case $F \in SDR_1$. In this case we have to deal with the loss of the synchronization.

All other cases can be derived straightforwardly from the previous proof.

Let $F_1, \ldots, F_m : \mathcal{I} \to \mathcal{I}$ simultaneously dyadic recursive with recursion depth one. Then we have

$$\begin{aligned}
F_i(\mathbf{x}) = \texttt{pif}\,\mathrm{left}(\mathbf{x})\,\texttt{then}\, & p_{i,1} \cdot \ldots \cdot p_{i,l_i} \cdot F_{h_i}(\mathrm{tail}_L(\mathbf{x})) \\
\texttt{else}\, & q_{i,1} \cdot \ldots \cdot q_{i,k_i} \cdot F_{j_i}(\mathrm{tail}_R(\mathbf{x}))
\end{aligned}$$

We synchronize the outputs. Let $d = \max\{l_i, k_i \mid 1 \leq i \leq m\}$

$$u_{i,L} = \mathit{dig}(p_{i,1}) \cdot \ldots \cdot \mathit{dig}(p_{i,l_i}) \cdot \Diamond^{(d-l_i)}$$

According to Example 4.3.8 (page 91) we define

$$\begin{aligned}
f'(\varepsilon,\varepsilon) &= \varepsilon \\
f'_{i,i'}(\overline{1}v, 1w) &= u_{i,L}\cdot f'_{h_i,j_{i'}} \\
f'_{i,i'}(\overline{1}v, \Diamond w) &= u_{i,L}\cdot f'_{h_i,h_i} \\
f'_{i,i'}(1v, \overline{1}w) &= u_{i,R}\cdot f'_{j_i,h_{i'}} \\
f'_{i,i'}(1v, \Diamond w) &= u_{i,R}\cdot f'_{j_i,j_i} \\
f'_{i,i'}(\Diamond v, \overline{1}w) &= \Diamond^d\cdot f'_{h_{i'},h'_i} \\
f'_{i,i'}(\Diamond v, 1w) &= \Diamond^d\cdot f'_{j_{i'},j'_i}
\end{aligned}$$

Gaps in the scheme may be filled arbitrarily. Only the cases above will occur.

Further we define

$$f_i(v) = \mathit{inf}^s(f_{i,i}(\mathit{left}_e(v), \mathit{right}_e(v)), f_{i,i}(\mathit{right}_e(v), \mathit{left}_e(v)))$$

Then f_i simulates the computation of F_i.

In case of more than one parameter, say $F_i : \mathcal{I}^k \to \mathcal{I}$ we need functions

$$f'_{i,i',\ldots,i^{(2^k)}} : (\Sigma^*)^{(2^k)} \to \Sigma^*$$

since we have to prepare for an invalid computation in each branch of each parameter. We have to remember all states of the corresponding branches. The construction of these functions is obvious.

Further we keep the outputs synchronized as long as possible, i.e. inf^s is applied in the last step, we will use inf^e before.

In case $k = 2$ we get

$$\begin{aligned}
f_i(v,w) = \mathit{inf}^s(&\mathit{inf}^e(f_{i,i,i,i}(\mathit{left}_e(v), \mathit{left}_e(w), \mathit{right}_e(v), \mathit{right}_e(w)),\\
&\qquad f_{i,i,i,i}(\mathit{left}_e(v), \mathit{right}_e(w), \mathit{right}_e(v), \mathit{left}_e(w))),\\
&\mathit{inf}^e\ (f_{i,i,i,i}(\mathit{right}_e(v), \mathit{left}_e(w), \mathit{left}_e(v), \mathit{right}_e(w)),\\
&\qquad f_{i,i,i,i}(\mathit{right}_e(v), \mathit{right}_e(w), \mathit{left}_e(v), \mathit{left}_e(w))))
\end{aligned}$$

Then f_i simulates F_i.

■

Chapter 5

Conclusion

We have considered several recursion principles to check their use for real number computation. Computations on infinite objects can be approximated by fully monotonic computations on finite words.

We found ε-recursions an appropriate method to obtain monotonic functions. And further we developed connections between primitive recursion, which distinguishes between recursion and side parameters, and uniform recursion, which only knows recursion parameters. In particular we found that primitive and uniform recursion do not coincide in case of recursion depth one.

Then we found that ε-uniform recursion is able to simulate dyadic recursion, which defines functions, that do not compute on words but on intervals.

We showed how the parallel conditional can be simulated by sequential functions on words.

Appendix A

Index of Symbols

Sets

Σ	(finite) alphabet, $\Sigma = \{a_1, \dots a_r\}$		p. 27
Σ^*	set of finite words over Σ		p. 27
Σ^ω	set of infinite words over Σ		p. 27
Σ^∞	$\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$		p 27
$\mathbb{N}$	set of natural numbers		
$\mathbb{R}$	set of real numbers		
$\mathcal{I}$	partial unit interval	L. 4.1.1	p. 67
$\mathcal{R}$	partial real line	Def. 4.1.4	p. 68

Functions

con_i	i-th successor function on words, $con_i : \Sigma^* \to \Sigma^*, w \mapsto a_i \cdot w$	Def. 3.2.1	p. 28
pro_i^k	uniform projections, $pro_i^k : (\Sigma^*)^k \to \Sigma^*, (x_1, \dots, x_k \mapsto x_i)$	Def. 3.2.1	p. 28
$proj_i^k$	projections, $proj_i^k : M^k \to M, (x_1, \dots, x_k \mapsto x_i)$	Def. 3.2.1	p. 28
$succ$	successor function on natural numbers, $succ : \mathbb{N} \to \mathbb{N}, i \mapsto i + 1$	Def. 2.2.2	p. 14
inf	infimum on words	Def. 4.3.3	p. 82
		Def. 4.3.12	p. 96
inf^e	external infimum	Def. 4.3.5	p. 89
inf^s	synchronized infimum	Def. 4.3.9	p. 94

Relations

$\leq_p$	prefix relation on words	Def. 3.1.1	p. 27
	$v \leq_p w \iff \exists u.\, w = uv$	Def. 3.1.2	p. 28
$\leq_l$	length comparison on words	Def. 3.1.1	p. 27
	$v \leq_l w \iff \lvert v\rvert \leq \lvert w\rvert$		
$\sqsubseteq$	order on an CPO	Def. 2.4.1	p. 22

Operators

BoundPr	bounded primitive recursion		
Prim	primitive recursion	Def. 2.2.1	p. 14
		Def. 3.2.2	p. 28
$Prim_\varepsilon$	ε-primitive recursion	Def. 3.3.2	p. 34
SimPr	simultaneously primitive recursion		
SimUnif	simultaneously uniform recursion		
Sub	(simultaneous) substitution	Def. 3.2.2	p. 28
		Def. 2.2.1	p. 14
Unif	uniform recursion	Def. 3.4.1	p. 39
$Unif_\varepsilon$	ε-uniform recursion	Def. 3.4.10	p. 46

Classes

PR	primitive recursive functions on natural numbers	Def. 2.2.2	p. 14
PR_n	... with recursion depth n	Def. 2.2.2	p. 14
$PR(\Sigma)$	primitive recursive functions on words over Σ	Def. 3.2.3	p. 29
$PR_n(\Sigma)$	... with recursion depth n	Def. 3.2.3	p. 29
$SPR(\Sigma)$	simultaneously primitive recursive functions on words over Σ	Def. 3.2.3	p. 29
$SPR_n(\Sigma)$	... with recursion depth n	Def. 3.2.3	p. 29
$\varepsilon\text{-}PR(\Sigma)$	ε-primitive recursive functions on words over Σ	Def. 3.3.2	p. 34
$\varepsilon\text{-}PR_n(\Sigma)$	... with recursion depth n	Def. 3.3.2	p. 34
$\varepsilon\text{-}SPR(\Sigma)$	simultaneously ε-primitive recursive functions on words over Σ	Def. 3.3.2	p. 34
$\varepsilon\text{-}SPR_n(\Sigma)$	... with recursion depth n	Def. 3.3.2	p. 34
$UR(\Sigma)$	uniformly recursive functions on words over Σ	Def. 3.4.1	p. 39
$UR_n(\Sigma)$	... with recursion depth n	Def. 3.4.1	p. 39
$SUR(\Sigma)$	simultaneously uniformly recursive functions on words over Σ		
$SUR_n(\Sigma)$	... with recursion depth n	Def. 3.4.1	p. 39
$\varepsilon\text{-}UR(\Sigma)$	ε-uniformly recursive functions on words over Σ	Def. 3.4.10	p. 46
$\varepsilon\text{-}UR_n(\Sigma)$	... with recursion depth n	Def. 3.4.10	p. 46
$\varepsilon\text{-}SUR(\Sigma)$	simultaneously ε-uniformly recursive functions on words over Σ	Def. 3.4.10	p. 46
$\varepsilon\text{-}SUR_n(\Sigma)$	... with recursion depth n	Def. 3.4.10	p. 46

Appendix B

Some Implementations

This appendix presents some implementations of examples developed within this work. Implementations are done in Objective Caml (O'Caml), syntax highlighting uses the Tuareg mode in Emacs.

More information on O'Caml can be found on the respective Web site [Tea05].

The source codes will be available for download at `http://kuerzer.de/disshs`.

B.1 Examples of primitive and uniformly recursive functions

This sections presents some basic examples on word functions. Words are represented as lists. Those are a basic data type in functional programming languages and deliver an easy syntax to denote the first letter and the rest of a word, which is needed to write down recursion schemes.

We use the currified versions of the respective functions.

example_aux.ml **1/2**

```
(* example_aux.ml *)


(*Auxiliary fuctions for displaying values *)

(* nats *)
(* Delivers first n+1 natural numbers as a list, i.e. [0;...;n] *)

(* We will use lists of naturals instead of words in some examples *)

let nats = fun n ->
  let  rec do_it = fun i j -> if j = 0 then [i]
  else  i :: (do_it (i+1) (j-1))
  in do_it 0 n;;

(* Same for non-positive numbers *)

let nats' = fun n -> List.map (fun x -> ~-x) (nats n);;

(* Positive and negative numbers, i.e. get rid of the zeros if neccessary *)

let pos = fun n -> List.tl (nats n);;
let negs = fun n -> List.tl (nats' n);;

(* Displays list of values *)

let  values = fun f n ->
  let rec do_it = fun (i,k) -> if i = 0 then [f (pos k)]
  else (f (pos k)) :: (do_it (i-1,k+1))
  in do_it (n,0);;

(* Displays length of output value depending on the length of input values
 * to observe increase and lookahead *)

let lengthes = fun f n ->
  let rec index : int -> 'a list -> (int * 'a) list = fun i ->
     (function [] -> []
        | a :: l -> (i,a) :: (index (i+1) l)
     )
  in index 0 (List.map  List.length (values f n));;


(* Composition of functions *)

let comp : ('b -> 'c) -> ('a -> 'b) -> ('a -> 'c) =
  fun f g -> (fun x -> f (g x));;

(* iteration, always useful *)

let rec iterate : int -> ('a -> 'a) -> ('a -> 'a) = function
    0 -> ( function _ ->  ( fun x-> x))
  | n -> ( function f -> comp f (iterate (n-1) f));;


let repeatlist = fun n l -> iterate n (List.append l) [];;

let multitail : int -> 'a list -> 'a list = fun n l -> (iterate n List.tl) l;;



(* Base functions *)

(* The successor functions  *)

(* Note that we use a parameter to define the letter to be appended,
   in theory we have a fixed function con_i for each letter a_i.
   That would force us to use a fixed finite alphabet and we would loose
   flexibility in the implemenation of our examples  *)

(* 'con' is obviously nothing more than syntactical sugar *)

let con i = function v -> i :: v;;
```

example_aux.ml **2/2**

```
(* Some projections *)
let proj'2_2= fun x y -> y;;
let proj'2_1= fun x y -> x;;
```

example_prim_rec.ml **1/2**

```
#use "example_aux.ml";;

(* Some primitive recursive functions on finite lists *)
(* Some of these function might already be  defined in O'Caml,
 * we want to implement their recursion schemes as exactly as possible *)

(* Note that we use curried forms *)

let con i = function v -> i :: v;;

(* Example 3.2.8 *)

let first = function [] -> []
  | a :: l -> [a];;

let rest = function [] -> []
  | a :: l -> l;;

let rec cut = function [] -> (function w -> w)
  | a :: v ->   (function w ->  rest (cut v w));;


(* 3.2.8-4 *)
(* In that example the right concatenation r_i is a unary function for each i,
 * but this implementation does not use a fixed alphabet,
 * which would be neccessary.

 * We use an additional parameter i instead.

 * The same holds for the function con defined above *)

let rec r = fun i -> (function [] -> [i]
                        | a :: v -> a :: (r i v));;

let rec rev = function [] -> []
  | a :: v -> r a (rev v);;

(* 3.2.8-5+6 *)
let rcut = fun v w -> rev (cut v (rev w));;

let rec conc = function [] -> (fun w -> w)
  | a :: v -> (fun w -> con a  (conc v w));;


(* 3.2.8-7 *)
let rec t : 'a list -> 'a list -> 'a list = function [] -> (fun w -> w)
  | a :: v -> (fun w -> rest (t v w)
             )
and s : 'a list -> 'a list -> 'a list = function [] -> (fun _ -> [])
  | a :: v -> (fun w -> a :: ((first (t v w)) @ (s v w))
             ) ;;


let shuffle_eps = fun v w ->
  s (rcut (cut w v) v) (rev (rcut (cut v w) w));;

(* 3.2.8-8 *)
let shuffle = fun v w -> conc (shuffle_eps v w) (conc (cut v w) (cut w v));;

let rec half = function [] -> []
  | a :: v -> half' v

and half' = function [] -> []
  | a :: v -> con a (half v);;

(* Example 3.3.5-2 *)

let rec ins = fun i -> (function [] -> []
                          | 1 :: w -> conc [1;i;2] (ins' i w)
                          | j :: w -> con j (ins' i w)
                        )
```

example_prim_rec.ml **2/2**

```
and ins' = fun i -> (function [] -> []
                        | a :: w -> con a (ins i w)
                    );;


let rec cont' = function [] -> []
  | i :: w  -> con 1 (con i (ins i (cont' w)));;


let cont = fun v -> half (cont' v);;



(* Example 3.3.5-3 *)

let rec rins' = fun i -> (function [] -> []
                           | 1 :: w -> [2;i;1] @ (rins i w)
                           | j :: w -> con j (rins i w)
                        )

and rins = fun i -> (function [] -> []
                        | a :: w -> con a (rins' i w)
                    );;


let rec rcont' = function [] -> []
  | i :: w  -> con i (con 1 (rins i (rcont' w)));;


let rcont = fun v -> half' (rcont' v);;
```

example_unif_rec.ml **1/3**

```
#use "example_aux.ml";;


let proj'2_1 = fun u v -> u;;

(* Example 3.2.8-9 *)
let rec half = function [] -> []
  | a :: l -> half' l

and half' = function [] -> []
  | a :: l -> a :: (half l);;


(* Example 3.4.3-1 *)
let rec shuffle = fun l l' ->
match (l,l') with ([],[])  -> []
  | ([], k) | (k, []) -> k
  | a :: k, a' :: k' -> a :: ( a' :: (shuffle k k'));;



(* Example 3.4.3-5 *)
let rec ms = function [] -> []
  | a :: l -> shuffle  l (ms l);;

let rec ms' = function [] -> []
  | a:: l -> a :: shuffle l (ms' l);;

(* Example 3.4.3-2 *)
let rec insert = fun i ->
 (function [] -> (function v -> i :: v)
        | a :: u -> (function [] -> [i]
        | b :: v -> b :: (insert i u v)));;



(* Example 3.4.3-3 *)

let rec cut = function [] -> (fun v -> v)
  | a :: u -> (function [] -> []
                 | b :: v -> cut u v);;

(* Example 3.4.3-4 *)
let rec conc = fun u' v' ->
match (u',v') with
    (u,[]) -> u
  | ([], b :: v) -> b :: v
  | (a :: u, j :: v) -> a :: (insert j u (conc u v));;

let rec ins = fun i -> (function [] -> []
                          | 1 :: w -> conc [1;i;2] (ins' i w)
                          | j :: w -> con j (ins' i w)
                       )

and ins' = fun i -> (function [] -> []
                       | a :: w -> con a (ins i w)
                    );;


let rec cont' = function [] -> []
  | i :: w  -> con 1 (con i (ins i (cont' w)));;


let cont = fun v -> half (cont' v);;

(*Some epsilon-uniformly recursive functions *)



(* Example 3.4.10-2 *)
let rec double = function [] -> []
  | a :: l -> [a;a] @ (double l);;
```

example_unif_rec.ml **2/3**

```
(* Example 3.4.10-1 *)

let rec shuffle_eps = fun u v -> match (u,v) with
    (_,[]) | ([],_) -> []
  | (a :: u', b :: v') -> con a (con b (shuffle_eps u' v'));;

let rec ms_eps = function
    [] -> []
  | a :: v -> con a  (shuffle_eps v (ms_eps v));;

(* Example 5.1.1-1 *)
let ds = fun l -> shuffle_eps l (double l);;


(* meta double, a "fast" increasing fully monotous function *)

let rec md = function [] -> []
  | a :: v -> a :: double (md v);;

let rec md2 = function [] -> []
  |a :: v -> [a;a] @ double (md v);;

(* uniform  projections *)
(* Example 3.4.10-3 *)

let rec pro'3_3 = fun u v w -> match (u,v,w) with
    ([],_,_) | (_,[],_) | (_,_,[]) -> []
  | (a :: u',b:: v', c:: w') -> c :: (pro'3_3 u' v' w');;

let rec pro'3_2 = fun u v w -> match (u,v,w) with
    ([],_,_) | (_,[],_) | (_,_,[]) -> []
  | (a :: u',b:: v', c:: w') -> b :: (pro'3_2 u' v' w');;

let rec pro'2_2 = fun u v  -> match (u,v) with
    ([],_) | (_,[]) -> []
  | (a :: u',b:: v') -> b :: (pro'2_2 u' v' );;

let rec pro'2_1 = fun u v  -> match (u,v) with
    ([],_) | (_,[]) -> []
  | (a :: u',b:: v') -> a :: (pro'2_1 u' v' );;


(* some truly uniformly recursive functions *)

(*Example 3.4.21-1 *)

let rec double_t = function [] -> []
  | a :: l -> [a;a] @ (pro'2_2 l (double_t l));;

let rec triple_t = function [] -> []
  | a :: l -> [a;a;a] @ (pro'2_2 l (triple_t l));;

let rec md_t = function [] -> []
  |a :: v -> a :: (double_t (pro'2_2 v (md_t v)));;

let rec shuffle_eps_t = fun u v -> match (u,v) with
    ([],_) | (_,[]) -> []
  | (a :: u',b:: v') -> a :: (b :: (pro'3_3 u' v' (shuffle_eps_t u' v')));;


(* Example *)

(* An alternating function *)

let rec alt = function [] -> []
  | a :: v -> 1 :: (alt' v)

and alt' = function [] -> []
  | a :: v -> 2 :: (alt v);;
```

example_unif_rec.ml **3/3**

```
(* Applying patterns *)
(* Example 3.5.6, Lemma 3.5.8 *)


let rec ap : 'a list -> 'a list -> 'a list = fun u v -> match (u,v) with
     (_,[]) | ([],_) -> []
  | (a :: u', i :: v') -> con a (pro'3_3 u' v' (ap_i i u' v'))

and ap_i : 'a -> 'a list -> 'a list -> 'a list = fun j u v -> match (j,u,v) with
    (_,[],_) | (_,_,[]) -> []
  | (i, i' :: u, a :: v) ->
      if i = i' then ap_i i u v else a :: (ap_i i' u v);;

(* Pattern length *)

let rec pa : 'a list  -> int = function
     [] -> 0
  |  i :: v -> (pa_i i v)

and pa_i : 'a ->  'a list -> int = fun j v -> match (j,v) with
    (_,[]) -> 0
  | (i, i' ::  v) ->
      if i = i' then pa_i i v else 1 + (pa_i i' v);;

(* Some truly uniformly recursive functions with interesting lookaheads *)

let rest = fun v -> ap (con 1 (alt v)) v;;

let log = function v -> ap (md_t (alt v)) v;;

let half = fun v -> ap  (double (alt v)) v;;

(* let half_t = fun v -> ap (t 1 (double_t (s v))) v;; *)

(* truly uniformly recursive  versions of some uniformly recursive functions *)
(* Example 3.3.17-2 *)

let rec rest  = function
    [] -> []
  | a :: v -> proj'2_1 v (rest v);;

let rec rest_t' = function
    [] -> []
  | a ::v -> pro'2_1 v (rest_t' v);;

let rec rest_t'' = function
    [] -> []
  | a :: v -> pro'2_1 v (a :: (rest_t'' v));;

let rec rest_t = function
    [] -> []
  | a :: v -> let w = rest v in
      pro'2_1 (pro'2_1 v w) (con a (pro'2_2 v w));;

let rec half' = function [] -> []
  | a :: v -> (pro'3_3 v (a :: (half' v)) (half1' v))

and half1' = function [] -> []
  | a :: v -> (a :: (pro'3_2 v (half' v) (half1' v)));;


let rec mmd = function
    [] -> []
  | a :: v -> a :: md (mmd v);;

let rec mmd_t = function
    [] -> []
  | a :: v -> a :: (md_t (pro'2_2 v (mmd_t v)));;
```

B.2 Uniformly recursive simulation of dyadic recursive functions

This section implements some examples of word functions simulating dyadic recursive functions.

We present the mediation operator and the multiplication.

In case of recursion depth 1, we did not use the external infimum.

dyad.ml 1/9

```
(* simulation of dyadic recursive function by word functions *)

type digit = int;;


exception IllegalDigit of int * int;;
exception WontFit of string * int * int;;

let rec makelist = fun n a -> if n = 0 then [] else (a :: (makelist (n-1) a));;


(* The infimum operator is defined by simultanous, uniform epsilon-recursion *)
(* We use "illegal digits", e.g. -2, to denote an discontinued stream/list *)

(* We use the symmetry like inf (0v,-1w) = inf (-1w,0v) *)


let legal : int -> bool = function -1 | 0 | 1 -> true | _ -> false;;
let illegal : int -> bool = fun x -> not (legal x);;


let rec inf = fun l l' ->
match (l,l') with [],_ | _ ,[] -> []
  | -1 :: l1, -1 :: l2 -> -1 :: inf l1 l2
  | -0 :: l1, 0 :: l2 -> 0 :: inf l1 l2
  | 1 :: l1, 1 :: l2 -> 1 :: inf l1 l2
  | -1 :: l1, 1 :: l2 | 1 :: l2 , -1 :: l1 -> infLR l1 l2
  | -1 :: l1, 0 :: l2 | 0 :: l2, -1 :: l1 -> infLC l1 l2
  | 0 :: l1, 1 :: l2 | 1 :: l2, 0 :: l1 -> infCR l1 l2
  | x :: l1, y :: l2 -> if illegal x then (if legal y then l' else [-2])
    else (if legal y then raise (Failure "That should ... ") else l)


(* L \sqcap C, consider second pair of digits *)
and infLC = fun l l' ->
match (l,l') with [],_ | _,[] -> []
  | -1 :: l1, -1 :: l2 -> -1 :: (infLR l1 l2)
  | -1 :: _, 0 :: _ -> raise (WontFit ("infLC",-1,0))
  | -1 :: _, 1 :: _ -> raise (WontFit ("infLC",-1,1))

  | 0 :: l1, -1 :: l2 -> -1 :: (infCR l1 l2)
  | 0 :: l1, 0 :: l2 -> infLC_CC l1 l2
  | 0 :: _ , 1 :: _ -> raise (WontFit ("infLC",0,1))

  | 1 :: l1, -1 :: l2 -> 0 :: (- 1 :: (inf l1 l2))
  | 1 :: l1, 0 :: l2 -> 0 :: (infLC l1 l2)
  | 1 :: l1 , 1 :: l2 -> 0 :: (infLR l1 l2)

  | x :: l1, y :: l2 -> (match (legal x, legal y) with
        (true,true) -> raise (Failure "That should ...")
      | (true,false) -> -1 :: l
      | (false,true) -> 0 :: l'
      | (false,false) ->[-2]
                        )


(* LC \sqcap CC, consider third pair of digits *)
and infLC_CC = fun  l l' ->
match (l,l') with [],_ | _,[] -> []
  | 1 :: l1, -1 :: l2 -> -1 :: (1 :: (infLR l1 l2))

  | 1 :: l1, x :: _ -> if (x = 0) || (x=1) then raise
   (WontFit ("infLC_CL",1,x))
    else [-1;0;1] @ l1

  | -1 :: l1, x :: _ -> if legal x then raise (WontFit ("infLC_CC", -1, x))
    else [-1;0;-1] @ l1

  | 0 :: l1, x :: _ -> if legal x  then raise (WontFit ("infLC_CC", -1, x))
```

dyad.ml **2/9**

```
    else [-1;0;0] @ l1

  | x :: _, y :: l2 ->  if  legal x then raise
(Failure "That should never happen")

  else if legal y then [0;0;y] @ l2 else [-2]

and infLR = fun l l' ->
match (l,l') with [],_ | _ ,[] -> []
  | 1 :: l1, -1 :: l2 -> 0 :: infLR l1 l2
  | x :: l1, y :: l2 -> (match (legal x, legal y) with
                             (true,true) -> raise (WontFit ("infLR",x,y))
                           | (true,false) -> -1 :: l
                           | (false,true) -> 1 :: l'
                           | (false,false) -> [-2]
                        )

and infCR = fun l l' ->
match (l,l') with [],_ | _ ,[] -> []
  | -1 :: l1, 1 :: l2 -> 0 :: infLR l1 l2
  | -1 :: _ , y :: _ -> if illegal y then 0 :: l else raise (WontFit ("infCR",-1,y))

  | 0 :: l1, -1 :: l2 -> 0 :: (infCR l1 l2)
  | 0 :: l1, 0 :: l2 -> infCR_CC l1 l2
  | 0 :: _, 1 :: _ -> raise (WontFit ("infCR",0,1))
  | 0 :: _, _ :: _ -> 0 :: l

  | 1 :: l1, -1 :: l2 -> 0 :: ( 1:: (inf l1 l2))
  | 1 :: l1, 0 :: l2 -> 1 :: (infLC l1 l2)
  | 1 :: l1, 1 :: l2 -> 1 :: (infLR l1 l2)
  | 1 :: l1, _ :: _ -> 0 :: ( 1 :: l1)

  | x :: _, y :: l2 -> if legal y then 1 :: l' else  [-2]

and infCR_CC = fun l l' ->
match (l,l') with [],_ | _ ,[] -> []
  | 1 :: l1, -1 :: l2 -> 0 :: (1 :: (infLR l1 l2))
  | x :: l1, y :: l2 -> (match (legal x, legal y) with
                             (true,true) -> raise (WontFit ("infCR_CC",x,y))
                           | (true,false) -> [0;0;x] @ l1
                           | (false,true) -> [1;0;y] @ l2
                           | (false,false) ->  [-2]
                        );;


let rec left = function
    [] -> []
  | -1 :: l -> 1 :: l
  | 0 :: l -> 1 :: (left l)
  | 1 :: _ -> [-2]
  | _ -> raise (Failure "Unexpected match");;

let rec right = function
    [] -> []
  | -1 :: _ -> [-2]
  | 0 :: l -> ~-1 :: (right l)
  | 1 :: l -> ~-1 :: l
  | _ -> raise (Failure "Unexpected match");;

(* We do not use the external infimum,
   the internal is easier to implement *)

let rec med2 = fun l l' ->
match (l,l') with
    [],_ | _ ,[] -> [] (*epsilon rule *)

      (* the four rules from the dyadic recursion scheme *)
  | -1 :: l1, -1 :: l2 -> -1 :: (med2 l1 l2)
  | -1 :: l1, 1 ::l2 -> 0 :: (med2 l1 l2)
  | 1 :: l1, -1 :: l2 -> 0 :: (med2 l1 l2)
  | 1 :: l1, 1 :: l2 -> 1 :: (med2 l1 l2)
```

dyad.ml **3/9**

```
      (*derivated rules containing the middle interval *)
    | -1 :: l1, 0 :: l2 -> inf (-1 :: (med2 l1 (left l2)))
                               (0 :: (med2 l1 (right l2)))
    | 0 :: l1, -1 :: l2 -> inf (-1 :: (med2 (left l1) l2))
                               (0 :: (med2 (right l1) l2))

    | 0 :: l1, 0 :: l2 ->
      let (left_l1,right_l1,left_l2,right_l2) =
            (left l1, right l1, left l2, right l2)
     in inf (inf (~-1 :: (med2 left_l1 left_l2)) (0 :: (med2 left_l1 right_l2)))
           (inf (0 :: (med2 right_l1 left_l2)) (1 :: (med2 right_l1 right_l2)))

    | 0 :: l1, 1 :: l2 -> inf (0 :: (med2 (left l1) l2))
                              (1 :: (med2 (right l1) l2))
    | 1 :: l1, 0 :: l2 -> inf (0 :: (med2 l1 (left l2)))
                              (1 :: (med2 l1 (right l2)))

      (* a illegal digit in at least one branch makes
         the whole computation illegal *)

  | _ :: _ , _ :: _ -> [-2];;

  let x = 0;;

  let rec mir = function
      [] -> []
    | -1 :: l -> 1 :: (mir l)
    | 0 :: l -> inf (1 :: (mir (left l))) (-1 :: (mir (right l)))
    | 1 :: l -> ~-1 :: (mir l)
    | _ :: _ -> [-2];;


  let empty = function [] -> true | _ -> false;;

  let sorttriple = fun  (l,l',l'') ->
  if (empty l) || ((empty l') || (empty l'')) then (l,l',l'') else
  match List.sort (fun l1 l2 -> (List.hd l1) - (List.hd l2)) [l;l';l'']
  with
          z :: (z' :: ( z'' :: _)) -> (z,z',z'')
        | _ ->  (l,l',l'');;


  let rec med3 = fun l l' l'' -> match sorttriple (l,l',l'') with
  [],_,_ | _,[],_ | _,_,[] -> [] (* epsilon rule *)

    | (x:: l1,y :: l2, z :: l3) -> (match (x,y,z) with

    (-1,-1,-1) -> -1 :: (med3 l1 l2 l3)
  | (-1,-1,1) -> med3LLR l1 l2 l3
  | (-1,1,1) -> med3LRR l1 l2 l3
  | (1,1,1) -> 1 :: (med3 l1 l2 l3)

  | (-1,-1,0) -> inf (-1 :: (med3 l1 l2 (left l3))) (med3LLR l1 l2 (right l3))
  | (-1,0,0) -> let (left_l2,right_l2,left_l3,right_l3) =
      (left l2, right l2,left l3, right l3)

    in inf (inf (-1 :: (med3 l1 left_l2 left_l3)) (med3LLR l1 left_l2 right_l3))
           (inf (med3LLR l1 right_l2 left_l3) (med3LRR l1 right_l2 right_l3))

  | (-1,0,1) -> inf (med3LLR l1 (left l2) l3) (med3LRR l1 (right l2) l3)

  | (0,0,0) -> let (left_l1,right_l1,left_l2,right_l2,left_l3,right_l3) =
      (left l1, right l1, left l2, right l2, left l3, right l3) in
  inf (inf (inf (-1 :: (med3 left_l1 left_l2 left_l3))
                 (med3LLR left_l1 left_l2 right_l3))
           (inf (med3LLR left_l1 right_l2 left_l3)
                (med3LRR left_l1 right_l2 right_l3)))
      (inf (inf (med3LLR right_l1 left_l2 left_l3)
                (med3LRR right_l1 left_l2 right_l3))
           (inf (med3LRR right_l1 right_l2 left_l3)
                (1 :: (med3 right_l1 right_l2 right_l3))))
```

dyad.ml **4/9**

```
| (0,0,1) -> let (left_l1,right_l1,left_l2,right_l2) =
              (left l1, right l1,left l2, right l2)
  in inf (inf (med3LLR left_l1 left_l1 l3) (med3LRR left_l1 right_l2 l3))
         (inf (med3LRR right_l1 left_l2 l3) (1 :: (med3 right_l1 right_l2 l3)))

| (0,1,1) -> inf (med3LRR (left l1) l2 l3) (1 :: (med3 (right l1) l2 l3))
| _ -> [-2]
                             ) (*match (x,y,z) *)

and  med3LLR = fun l l' l'' -> match sorttriple (l,l',l'') with
[],_,_ | _,[],_ | _,_,[] -> [] (* epsilon rule *)

  | (x:: l1,y :: l2, z :: l3) -> (match (x,y,z) with

  (-1,-1,-1) -> -1 :: (med3LRR l1 l2 l3)
| (-1,-1,1) -> [-1;1] @ (med3 l1 l2 l3)
| (-1,1,1) -> 0 :: (med3LLR l1 l2 l3)
| (1,1,1) -> 0 :: (med3LRR l1 l2 l3)

(* derived rules *)

(* example med3LLR(-1u,-1v,0w = (med3LLR (-1u,-1v,-1left(w)))
                   inf (med3LLR (-1u,-1v,1 right(w)))) *)

| (-1,-1,0) -> inf (-1 :: (med3LRR l1 l2 (left l3)))
                   ([-1;1] @ (med3 l1 l2 (right l3)))

| (-1,0,0) -> let (left_l2,right_l2,left_l3,right_l3) =
                   (left l2, right l2,left l3, right l3)
  in inf (inf (-1 :: (med3LRR l1 left_l2 left_l3))
              ([-1;1] @ med3 l1 left_l2 right_l3))
         (inf ([-1;1] @ (med3 l1 right_l2 left_l3))
              (0 :: (med3LLR l1 right_l2 right_l3)))

| (-1,0,1) -> inf ([-1;1] @ (med3 l1 (left l2) l3))
                  (0 :: (med3LLR l1 (right l2) l3))

| (0,0,0) -> let (left_l1,right_l1,left_l2,right_l2,left_l3,right_l3) =
    (left l1, right l1, left l2, right l2, left l3, right l3) in
inf (inf (inf (-1 :: (med3LRR left_l1 left_l2 left_l3))
([-1;1] @ (med3LLR left_l1 left_l2 right_l3)))
         (inf ([-1;1] @ (med3LLR left_l1 right_l2 left_l3))
(0 :: (med3LLR left_l1 right_l2 right_l3))))
    (inf (inf ([-1;1] @ (med3 right_l1 left_l2 left_l3))
( 0 :: (med3LLR  right_l1 left_l2 right_l3)))
         (inf (0 ::  (med3LLR right_l1 right_l2 left_l3))
(0 :: (med3LRR right_l1 right_l2 right_l3))))

| (0,0,1) -> let (left_l1,right_l1,left_l2,right_l2) =
  (left l1, right l1,left l2, right l2)
  in inf (inf ([-1;1] @ (med3 left_l1 left_l1 l3))
               (0 :: (med3LLR left_l1 right_l2 l3)))
         (inf (0 :: (med3LLR right_l1 left_l2 l3))
              (0 :: (med3LRR right_l1 right_l2 l3)))

| (0,1,1) -> inf (0 :: (med3LLR (left l1) l2 l3))
                 (0 :: (med3LRR (right l1) l2 l3))
| _ -> [-2]
                             ) (* match (x,y,z) , med3LLR *)

and med3LRR = fun l l' l'' -> match sorttriple (l,l',l'') with
[],_,_ | _,[],_ | _,_,[] -> [] (* epsilon rule *)

(* rules from the dyadic recursion scheme *)
  |(x:: l1,y :: l2, z :: l3) -> (match (x,y,z) with

  (-1,-1,-1) -> 0 :: (med3LLR l1 l2 l3)
| (-1,-1,1) -> 0 :: (med3LRR l1 l2 l3)
| (-1,1,1) -> [1;-1] @ (med3 l1 l2 l3)
| (1,1,1) -> 1 :: (med3LLR l1 l2 l3)

(* derived rules including the middle interval*)
```

dyad.ml **5/9**

```
| (-1,-1,0) -> inf (0 :: (med3LLR l1 l2 (left l3)))
                   (0 :: (med3LRR l1 l2 (right l3)))

| (-1,0,0) -> let (left_l2,right_l2,left_l3,right_l3) =
                   (left l2, right l2,left l3, right l3)
  in inf (inf (0 :: (med3LLR l1 left_l2 left_l3))
              (0 ::( med3LRR l1 left_l2 right_l3)))
         (inf (0 ::  (med3LRR l1 right_l2 left_l3))
              ([1;-1] @ (med3 l1 right_l2 right_l3)))

| (-1,0,1) -> inf (0 ::  (med3LRR l1 (left l2) l3))
                  ([1;-1] @ (med3 l1 (right l2) l3))

| (0,0,0) -> let (left_l1,right_l1,left_l2,right_l2,left_l3,right_l3) =
    (left l1, right l1, left l2, right l2, left l3, right l3) in

inf (inf (inf (0 :: (med3LLR left_l1 left_l2 left_l3))
(0 :: (med3LRR left_l1 left_l2 right_l3)))
         (inf (0 :: (med3LRR left_l1 right_l2 left_l3))
              ([1;-1] @ (med3 left_l1 right_l2 right_l3))))
    (inf (inf (0 ::  (med3LRR right_l1 left_l2 left_l3))
     ( [1;-1] @ (med3  right_l1 left_l2 right_l3)))
         (inf ([1;-1] @  (med3 right_l1 right_l2 left_l3))
(1 :: (med3LLR right_l1 right_l2 right_l3))))

| (0,0,1) -> let (left_l1,right_l1,left_l2,right_l2) =
                  (left l1, right l1,left l2, right l2)
  in inf (inf (0 :: (med3LRR left_l1 left_l1 l3))
              ([1;-1] @ (med3 left_l1 right_l2 l3)))
         (inf ([1;-1] @ (med3 right_l1 left_l2 l3))
              (1 :: (med3LLR right_l1 right_l2 l3)))

| (0,1,1) -> inf ([1;-1] @  (med3 (left l1) l2 l3))
                 (1 :: (med3LLR (right l1) l2 l3))
| _ -> [-2]
                             );; (* match (x,y,z) , med3LRR *)

(*New infimum operator for functions with recursion depths 2
 * or higher,
 * allows consideration of non-fitting situations *)

(* We simulate the simultaneous recursion by an additional paramter
 * --- s ---, this should be read as the function index.
 *
 * In particular we remember the currently read digits in a list
 * instead of the state *)

let x = ref [];;
  let y = ref [];;

let (a1,a2,a3)  = (-1,0,1);;

let rec fit’ = fun x y -> match (x,y) with
    ([],_) | (_,[]) -> []
  | (-2 :: v, b :: w) -> [a2]
  | (a :: v, -2 ::  w) ->[ a1]
  | (a :: v, b :: w) -> if ((abs a) <= 1) && ((abs b) <= 1)
    then fit’ v w else raise (Failure "Somethings wrong!");;

let rec fit =  function

(*fit*)
([],[]) -> (fun x y -> match (x,y) with
     ([],_) | (_,[]) -> []
  | (-2 :: _, _ :: _) | (_ :: _,-2 :: _) -> [-2]
  | (a :: v,b ::  w) -> if a=b then a3 :: (fit ([],[]) v w)
    else (if a <= b then a3 :: (fit ([a],[b]) v w)
                    else a3 :: ( fit ([b],[a]) w v)))


(*fitLC*)
```

dyad.ml **6/9**

```
  | ([-1],[0]) -> (fun x y -> match (x,y) with
               ([],_) | (_,[]) -> []
             | a :: v, b :: w -> (match (a,b) with
                   (-1,-1) -> a3 :: (fit ([-1],[1]) v w)
                 | (0,-1) -> a3 :: (fit ([0],[1]) v w)
                 | (0,0) -> a3 :: (fit ([-1;0],[0;0]) v w)
                 | (1,-1) -> a3 :: (fit ([],[]) v w)
                 | (1,0) -> a3 :: (fit ([-1],[0]) v w)
                 | (1,1) -> a3 :: (fit ([-1],[1]) v w)
                 | (-2,_) | (2,-2) -> -2 :: (fit' v w)
                 | (_,_) -> fit' v w))

(* fitLR *)
  | ([-1],[1]) -> (fun x y -> match (x,y) with
                  ([],_) | (_,[]) -> []
                | (1 :: v,-1 :: w) -> a3 :: (fit ( [-1],[1]) v w)
                | (-2 :: _, _ ) | (_, -2 :: _ ) -> [-2]
                | (_ :: v,_ :: w) -> fit' v w)


(* fitCR *)
  | ([0],[1]) -> (fun x y -> match (x,y) with
                  ([],_) | (_,[]) -> []
                | (a :: v, b :: w) -> (match (a,b) with
                   (-1,1) -> a3 :: (fit ([-1],[1]) v w)
                 | (0,-1) -> a3 :: (fit ([0],[1]) v w)
                 | (0,0) -> a3 :: (fit ([0;0],[1;0]) v w)
                 | (1,-1) -> a3 :: (fit ([],[]) v w)
                 | (1,0) -> a3 :: (fit ([-1],[0]) v w)
                 | (1,1) -> a3 :: (fit ([1],[1]) v w)
                 | (-2,_) | (_,-2) -> [-2]
                 | (_,_) -> fit' v w ))

  | ([-1;0],[0;0])
  | ([0;0],[1;0]) -> (fun x y -> match (x,y) with
                       ([],_) | (_,[]) -> []
                     | (1 :: v, -1 :: w) -> a3 :: (fit ([-1],[1]) v w)
                     | ( -2 :: _, _) | (_, -2 :: _) -> [-2]
                     | ( _ :: v , _ :: w) -> fit' v w)


  | a,b ->  raise (Failure "Unexpected call of fit");;



let rec inf3 = fun x y z ->
match (x,y,z) with
( [],_,_) | (_ ,[],_) -> []
  | (-1 :: v, -1 :: w, _ :: u) -> -1 :: (inf3 v w u)
  | (0 :: v, 0 :: w,_ :: u)   -> 0 :: (inf3 v w u)
  | (1 :: v, 1 :: w,_ :: u)     -> 1 :: (inf3 v w u)
  | (-1 :: v, 1 :: w,_:: u)
  | (1 :: w , -1 :: v,_:: u)   -> inf3LR v w u
  | (-1 :: v, 0 :: w,_ :: u)
  | (0 :: w, -1 :: v,_:: u)    -> inf3LC v w u
  | (0 :: v, 1 :: w,_ :: u)
  | (1 :: w, 0 :: v,_ :: u)     -> inf3CR v w u
  | (-2 :: _, -2 :: _,_)   -> [-2]
  | (-2 :: _, v,_)
  | (v, -2 :: _,_)         -> v


(* L \sqcap C, consider second pair of digits *)
and inf3LC = fun x y z ->
match (x,y,z) with ([],_,_) |( _,[],_) | (_,_,[]) -> []
  | (-1 :: v, -1 :: w, _ :: u) -> -1 :: (inf3LR v w u)
  | (-1 :: v, 0 :: _, -1(*a1*) :: _) -> -1 :: (-1 :: v)
  | (-1 :: _, 0 :: w, 0(*a2*) :: _) -> 0 :: (0 :: w)
  | (-1 :: v, 1 :: _, -1 :: _) -> -1 :: (-1 :: v)
  | (-1 :: _, 1 :: w, 0 :: _) -> 0 :: (1 :: w)
```

dyad.ml **7/9**

```
  | (0 :: v, -1 :: w, _ :: u) -> -1 :: (inf3CR v w u)
  | (0 :: v, 0 :: w, _ :: u)  -> inf3LC_CC v w u

  | (0 :: v , 1 :: _, -1 :: _) -> -1 :: (0 :: v)
  | (0 :: _, 1 :: w, 0 :: _)  -> 0 :: (1 :: w)

  | (1 :: v, -1 :: w, _ :: u)  -> 0 :: (- 1 :: (inf3 v w u))
  | (1 :: v, 0 :: w, _ :: u)   -> 0 :: (inf3LC v w u)
  | (1 :: v , 1 :: w, _ :: u)  -> 0 :: (inf3LR v w u)

  | (-2 :: _, -2 :: _,_) -> [-2]
  | (-2 :: _ , w, _)      -> 0 :: w
  | ( v, -2 :: _, _)      -> -1 :: v


(* LC \sqcap CC, consider third pair of digits *)
and inf3LC_CC = fun  x y z ->
match (x, y, z) with ([],_,_) | (_,[],_) | (_,_,[]) -> []
  | (1 :: v, -1 :: w, _ :: u) -> -1 :: (1 :: (inf3LR v w u))

  | (1 :: v, 1 :: _, -1 :: _)
  | (1 :: v, 0 :: _, -1 :: _)
  | (1 :: v, -1 :: _ , _ )
  | (1 :: v, -2 :: _,_)       -> [-1;0;1] @ v

  | (-2 :: _, -2 :: _,_)      -> [-2]

  | (a :: v, -2 :: _,_)
  | (a :: v, _, -1 :: _)     -> [-1;0;a] @ v

  | (-2 :: _, b :: w, _)
  | (_ , b :: w, 0 :: _)    -> [0;0;b] @ w


and inf3LR = fun x y z ->
match (x,y,z) with ([],_,_) | (_ ,[],_) | (_,_,[]) -> []
  | (1 :: v, -1 :: w, _ :: u) -> 0 :: (inf3LR v w u)

  | (-2 :: _,-2 :: _, _) -> [-2]
  | (-2 :: _ , w ,_) -> 1 :: w
  | (v, -2 :: _, _)  -> -1 :: v


  | (v, _, -1:: _) -> -1 :: v
  | (_,w, 0 :: _)  -> 1 :: w


and inf3CR = fun x y z->
match (x,y,z) with ([],_,_) | (_ ,[],_) | (_,_,[])  -> []
  | (-2 :: _, -2 :: _, _) -> [-2]

  | (-1 :: v, 1 :: w, _ :: u)  -> 0 :: (inf3LR v w u)
  | (-1 :: v, -2 :: _, _)
  | (-1 :: v, _, -1 :: _)      -> [0;-1] @ v

  | (0 :: v, -1 :: w, _ :: u)  -> 0 :: (inf3CR v w u)
  | (0 :: v, 0 :: w, _ :: u)   -> inf3CR_CC v w u

  | (0 :: v, 1 :: _, -1 :: _)
  | (0 :: v, -2 :: _ ,_ )      -> [0;0] @ v
  | (0 :: _, 1 :: w, 0 :: _)
  | (-2 :: _, 1 :: w, _)       -> [1;1] @ w

  | (1 :: v, -1 :: w, _ :: u)  -> 0 :: ( 1:: (inf3 v w u))
  | (1 :: v, 0 :: w, _ :: u)   -> 1 :: (inf3LC v w u)
  | (1 :: v, 1 :: w, _ :: u)   -> 1 :: (inf3LR v w u)
  | (1 :: v, -2 :: _, _)       -> [0;1] @ v
```

dyad.ml **8/9**

```
and inf3CR_CC = fun x y z ->
match (x,y,z) with ([],_,_) | (_ ,[],_) | (_,_,[])-> []
  | (1 :: v, -1 :: w, _ :: u) -> 0 :: (1 :: (inf3LR v w u))

  | (-2 :: _, -2 :: _, _) -> [-2]
  | (-2 :: _, b :: w, _)
  | (_ , b :: w, 0 :: _) -> [1;0;b] @ w
  | (a :: v, -2 :: _, _)
  | (a :: v, _, -1 :: _)  -> [0;0;a] @ v;;



(* prewash *)
(* The function "prewash" ensures, that all illegal digits are
 * represented by -2, case distinctions are a bit easier then *)

let rec prewash = function [] -> []
                           | a ::  v ->
                            if (abs a) <= 1 then a :: (prewash v)
                              else -2 :: (prewash v);;

let newinf = fun v w -> let (v',w') =(prewash v,prewash w)
in inf3 v' w' (fit ([],[]) v' w');;


(*Further mediations can be derived from the known *)


let med4 w x y z = med2 (med2 w x) (med2 y z);;
let med8 x1 x2 x3 x4 x5 x6 x7 x8 = med2 (med4 x1 x2 x3 x4) (med4 x5 x6 x7 x8);;


(* recursion schemes for the constants *)
let rec one = function [] -> []
  | a :: l -> 1 :: (one l);;

let rec zero = function [] -> []
  | a :: l -> ~-1 :: (zero l);;

let rec mult : int list -> int list -> int list = function [] ->
(function _ -> [])
  | i :: l -> (function [] -> [] (* epsilon case *)
                  | j :: l' -> let (x,y,l1,l2) =
                     if i < j then (i,j,l,l') else (j,i,l',l) in
                        (match (x,y) with
                             -1,-1 -> [-1;-1] @ (mult l1 l2)
                           | -1, 1 -> -1 :: (med2 l1 (mult l1 l2))
                           | 1,1 -> multRR l1 l2

                           | -1,0 -> newinf ([-1;-1] @ (mult l1 (left l2)))
                               (-1 :: (med2 l1 (mult l1 (right l2))))

                           | 0,0 -> let (left_l1,right_l1,left_l2,right_l2) =
                               (left l1, right l1, left l2, right l2)

        in newinf (newinf ([-1;-1] @ (mult left_l1 left_l2))
                          (-1 :: (med2 left_l1 (mult left_l1 right_l2))))
                   (newinf (-1 :: (med2 (mult right_l1 left_l2) left_l2))
                                      (multRR right_l1 right_l2))

                | 0,1 -> let (left_l1,right_l1) = (left l1,right l1)
                in newinf (-1 :: (med2 left_l1 (mult left_l1 l2)))
                                    (multRR  right_l1 l2)
                   | _ -> [-2]
                        )
              )

and multRR : int list -> int list -> int list =
function [] -> (function _ -> [])
  | i :: l -> (function [] -> [] (* epsilon case *)
                 | j :: l' -> let (x,y,l1,l2) =
```

dyad.ml **9/9**

```
                        if i < j then (i,j,l,l’) else (j,i,l’,l) in

                         (match (x,y) with

(*cases from the dyadic recursion scheme *)
         -1,-1 -> (let zl = zero l1 in 0 ::
        med8 zl zl zl l1 l1 l2 l2 (mult l1 l2))
        | -1,1 -> (let ol = one l in 0 ::
                           med8 ol ol l1 l1 l2 l2 l2 (mult l1 l2))
        |1,1 -> 1 :: (med8 (one l1) l1 l1 l1 l2 l2 l2 (mult l1 l2))

                (* derived cases *)
        | -1,0 -> let left_l2,right_l2 = (left l2, right l2) in
        let (zl,ol) =( zero l1, one l1) in
0 :: newinf (med8 zl zl zl l1 l1 left_l2 left_l2 (mult l1 left_l2))
         (med8 ol ol l1 l1 right_l2 right_l2 right_l2 (mult l1 right_l2))

        | 0,1  -> let left_l1,right_l1 = (left l1, right l1) in
        let (zl,ol) =( zero l1, one l1) in
        newinf (0 :: (med8 ol ol left_l1 left_l1 l2 l2 l2 (mult left_l1 l2)))
             (1 :: (med8 ol left_l1 left_l1 left_l1 l2 l2 l2 (mult left_l1 l2)))

        | 0,0 -> let (left_l1,right_l1,left_l2,right_l2) =
       (left l1,right l1,left l2, right l2)
          in let (zl,ol) =( zero l1, one l1) in
            newinf (0 :: newinf (med8 zl zl zl left_l1 left_l1 left_l2 left_l2
                                        (mult left_l1 left_l2))
         (med8 ol ol left_l1 left_l1 right_l2 right_l2 right_l2
         (mult left_l1 right_l2)))
(newinf (0 :: med8 ol ol right_l1 right_l1 right_l1 left_l2 left_l2
(mult right_l1 left_l2))
     (1 :: (med8 (one l1) right_l1 right_l1 right_l1 right_l2 right_l2 right_l2
                  (mult right_l1 right_l2))))

        | _ -> [-2]
));;
```

Index

Bibliography

[AJ94] Samson Abramksy and Achim Jung. Domain theory. In T.S.E. Maibaum S. Abramsky, Dov M. Gabbay, editor, *Handbook of Logik in Computer Science*, volume 3, Semantic Structures, pages 1–168. Oxford Science Publications, Clarendon Press, 1994.

[DW80] Thomas Deil and Klaus Weihrauch. Berechenbarkeit auf cpo-s. Technical Report 63, RWTH Aachen, 1980.

[ES97] M.H. Escardó and T. Streicher. Induction and recursion on the partial real line with applications to real pcf. *Theoretical Computer Science*, 1997. To appear.

[Esc96a] M.H. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162(1):79–115, August 1996.

[Esc96b] M.H. Escardó. *PCF extended with real numbers: A domain-theoretic approach to higher-order exact real number computation.* PhD thesis, Imperial College, Department of Computing, November 1996. http://theory.doc.ic.ac.uk.

[Grz57] Andrei Grzegorczyk. On the definition of computable real continuous functions. *Fund. Math.*, (44):61–71, 157.

[Grz53] Andrei Grzegorczyk. Some classes of recursive functions. *Rozprawy Matemathczne*, IV, 1953.

[Hei61] Walther Heinermann. *Untersuchungen über die Rekursionszahlen rekursiver Funktionen.* PhD thesis, Universität Münster, 1961.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Kon98] Michal Konecny. Real functions incrementally computable by finite automatons. Technical Report CSR-98-07, University of Birmingham, School of Computer Science, October 1998.

[Kon00] Michal Konecny. Real functions computable by finite transducers using affine ifs representations. Technical Report CSR-00-10, University of Birmingham, School of Computer Science, July 2000.

[Lew81] Lewis/Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[Mül74] Helmut Müller. *Klassifizierung der primitiv-rekursiven Funktionen.* PhD thesis, Univ. Münster, 1974.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Sch69] Helmut Schwichtenberg. Rekursionszahlen und die Grzegorczyk-Hierarchie. *Archiv Math. Logik Grundl.*, pages 85–91, 1969.

[Sch97] Holger Schulz. *Berechenbarkeit auf Reellen Zahlen - ein Vergleich.* Diplomarbeit, Universität-GH Siegen, 1997.

[Sco70] D.S. Scott. Outline of mathematical theory of computation. In *4th annual Princeton conference on information science and systems*, pages 169–176, 1970.

[Spr95] Klaus-Hilmar Sprenger. *Hierachies of Primitive Recursive Functions on Term Algebras.* PhD thesis, Universität GH Siegen, Fachbereich Mathematik, 1995.

[SS00] Holger Schulz and Dieter Spreen. On the equivalence of some approaches to computability on the real line. In Klaus Keimel etal., editor, *Domains and Processes, Proc. 1st Intern. Symp. on Domain Theory, Shanghai, China, 1999*, 2000.

[Tea05] The Caml Team. The caml language. `http://caml.inria.fr`, 1995–2005.

[Tur37a] Alan Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1937.

[Tur37b] Alan Turing. On computable numbers with an application to the Entscheidungsproblem. a correction. *Proc. London Math. Soc.*, 43:544–546, 1937.

[Wei74] Klaus Weihrauch. Teilklassen primitiv-rekursiver Wortfunktionen. Technical report, Gesellschaft für Mathematik und Datenverabeitung, Bonn, 1974.

[Wei97a] Klaus Weihrauch. A foundation of computable analysis. In D.S. Bridges, C.S. Calude, J. Gibbons, S. Reeves, and I.H. Witten, editors, *Combinatorics, complexity and logic, discrete mathematics and computer science, Proceedings of DMTCS 96*, pages 66–89. Springer Verlag, Singapore, 1997.

[Wei97b] Klaus Weihrauch. A simple introduction to computable analysis. Technical report, FernUniversität-Gesamthochschule Hagen, 1997. 2nd edition.

[Wei00] Klaus Weihrauch. *Computable Analysis: an Introduction.* Texts in theoretical computer science. Springer Berlin, 2000.