by the definition of "$\lfloor \ldots \rfloor$". Thus, z is *smallest* such that $x/y < z + 1$, or such that $x < y(z + 1)$. ◀

It follows that, for $y > 0$, the search in $(*)$ yields the "normal math" value for $\lfloor x/y \rfloor$, while it re-defines $\lfloor x/0 \rfloor$ as $= x + 1$.

(2) $\lambda xy.rem(x, y)$ (the remainder of the division x/y). $rem(x, y) = x \dot{-} y\lfloor x/y \rfloor$.

(3) $\lambda xy.x|y$ (x divides y). $x|y \equiv rem(y, x) = 0$; now apply 2.2.2. Note that if $y > 0$, we cannot have $0|y$ —a good thing!— since $rem(y, 0) = y$. Our redefinition of $\lfloor x/y \rfloor$ yields, however, $0|0$, but we can live with this in practice.

(4) $Pr(x)$ (x is a prime). $Pr(x) \equiv x > 1 \wedge (\forall y)_{\leq x}(y|x \to y = 1 \vee y = x)$.

(5) $\pi(x)$ (the number of primes $\leq x$).[†] The following primitive recursion certifies the claim: $\pi(0) = 0$, and $\pi(x + 1) =$ if $Pr(x + 1)$ then $\pi(x) + 1$ else $\pi(x)$.

(6) $\lambda n.p_n$ (the nth prime). First note that the graph $y = p_n$ is primitive recursive: $y = p_n \equiv Pr(y) \wedge \pi(y) = n + 1$. Next note that, for all n, $p_n \leq 2^{2^n}$ (see Exercise 2.4.1 below), thus $p_n = (\mu y)_{\leq 2^{2^n}}(y = p_n)$, which settles the claim.

(7) $\lambda nx.\exp(n, x)$ (*the* exponent of p_n in the prime factorization of x). $\exp(n, x) = (\mu y)_{\leq x}\neg(p_n^{y+1}|x)$.

Pause. Is x a good bound? *Yes!* $x = \ldots p_n^y \ldots \geq p_n^y \geq 2^y > y$. ◀

(8) $Seq(x)$ (says that x's prime number factorization contains 2, and has no gaps; no prime, between 2 and the largest in the factorisation, is missing.)

$Seq(x) \equiv x > 1 \wedge (\forall y)_{\leq x}(\forall z)_{\leq x}(Pr(y) \wedge Pr(z) \wedge y < z \wedge z|x \to y|x)$. □

2.3.12 Remark. What makes $\exp(n, x)$ "*the* exponent of p_n in the prime factorization of x", rather than *an* exponent, is Euclid's prime number factorization theorem: *Every number $x > 1$ has a unique factorization —within permutation of factors— as a product of primes.* □

2.4 Coding sequences; special recursions

2.4.1 Exercise. Prove by induction on n, that for all n we have $p_n \leq 2^{2^n}$.

Hint. Consider, as Euclid did,[†] $p_0 p_1 \cdots p_n + 1$. If this number is prime, then it is greater than or equal to p_{n+1} (Why?). If it is composite, then none of the primes up to p_n divide it. So any prime factor of it is greater than or equal to p_{n+1} (Why?). □

[†]The π-function plays a central role in number theory, figuring in the so-called *prime number theorem*. See, for example, [LeV56].

[†]In his proof that there are infinitely many primes.

2.4.2 Definition. (Coding Sequences) Any sequence of numbers, $a_0, \ldots, a_n$, $n \geq 0$, is *coded* by the number denoted by the symbol $\langle a_0, \ldots, a_n \rangle$ and defined as

$$\prod_{i \leq n} p_i^{a_i+1} \qquad \square$$

For *coding* to be useful, we need a simple *decoding* scheme. By Remark 2.3.12 there is no way to have $z = \langle a_0, \ldots, a_n \rangle = \langle b_0, \ldots, b_m \rangle$, *unless*

(i) $n = m$

and

(ii) for $i = 0, \ldots, n$, $a_i = b_i$.

Thus, it makes sense to correspondingly define the *decoding expressions*:

(*i*) $lh(z)$ (pronounced "length of z") as shorthand for $(\mu y)_{\leq z} \neg(p_y | z)$

A **comment** and a **question**:

- The comment: If p_y is the first prime *not* in the decomposition of z, and $Seq(z)$ holds, then since numbering of primes starts at 0, the length of the coded sequence z is indeed y.
- Is the bound z sufficient? *Yes!*

$$z = 2^{a+1} 3^{b+1} \cdots p_{y \dot{-} 1}^{exp(y \dot{-} 1, z)} \geq \underbrace{2 \cdot 2 \cdots 2}_{y \text{ times}} = 2^y > y$$

(*ii*) $(z)_i$ is *shorthand* for $\exp(i, z) \dot{-} 1$

Note that

(a) $\lambda iz.(z)_i$ and $\lambda z.lh(z)$ are in $\mathcal{PR}$.

(b) If $Seq(z)$, then $z = \langle a_0, \ldots, a_n \rangle$ for some $a_0, \ldots, a_n$. In this case, $lh(z)$ equals the number of distinct primes in the decomposition of z, that is, the length $n + 1$ of the coded sequence. Then $(z)_i$, for $i < lh(z)$, equals a_i. For larger i, $(z)_i = 0$. Note that if $\neg Seq(z)$ then $lh(z)$ need not equal the number of distinct primes in the decomposition of z. For example, 10 has 2 primes, but $lh(10) = 1$.

The tools lh, $Seq(z)$, and $\lambda iz.(z)_i$ are sufficient to perform *decoding*, *primitive recursively*, once the truth of $Seq(z)$ is established. This coding/decoding scheme is essentially that of [Göd31], and will be the one we use throughout these notes.

2.4.1 Concatenating (coded) sequences; stacks

Consider the two sequences $a_0, \ldots, a_m$ and $b_0, \ldots, b_n$. Is there an easy (e.g., primitive recursive) computational way to obtain $\langle a_0, \ldots, a_m, b_0, \ldots, b_n\rangle$ from $\langle a_0, \ldots, a_m\rangle$ and $\langle b_0, \ldots, b_n\rangle$? Yes, there is: Note that by 2.4.2 we have

$$\langle a_0, \ldots, a_m\rangle = \prod_{i \le m} p_i^{a_i+1} \tag{1}$$

and

$$\langle b_0, \ldots, b_n\rangle = \prod_{i \le n} p_i^{b_i+1}$$

Thus, let us write $z = \langle a_0, \ldots, a_m\rangle$ and $w = \langle b_0, \ldots, b_n\rangle$. Then we have

$$\begin{aligned}
\langle a_0, \ldots, a_m, b_0, \ldots, b_n\rangle &= \Big(\textstyle\prod_{i \le m} p_i^{a_i+1}\Big) \textstyle\prod_{i \le n} p_{i+m+1}^{b_i+1} \\
&= \langle a_0, \ldots, a_m\rangle \textstyle\prod_{i \le n} p_{i+m+1}^{b_i+1} && \text{using (1)} \\
&= z \textstyle\prod_{i < lh(w)} p_{i+lh(z)}^{\exp(i,w)} && \text{using notation from 2.3.11}
\end{aligned}$$

The last right hand side expression above is meaningful regardless of whether z and w code sequences —i.e., even if one or both of $Seq(z)$ and $Seq(w)$ are false.

Thus we define

2.4.3 Definition. The function $\lambda zw.z * w$ given for *all* z, w in $\mathbb{N}$ is defined by

$$z * w \overset{Def}{=} z \prod_{i < lh(w)} p_{i+lh(z)}^{\exp(i,w)}$$

We call it the concatenation of natural numbers function. □

Our discussion above yields at once

2.4.4 Proposition. *If $x = \langle a_0, \ldots, a_m\rangle$ and $y = \langle b_0, \ldots, b_n\rangle$, then $x * y = \langle a_0, \ldots, a_m, b_0, \ldots, b_n\rangle$.*

But is $\lambda xy.x * y$ easily computable? Well, yes, it is if the qualifier "easily" means primitive recursively.

This will need a simple lemma:

2.4.5 Lemma. *If $\lambda x\vec{y}.f(x, \vec{y}) \in \mathcal{PR}$ then so is $\lambda z\vec{y}.\prod_{x<z} f(x, \vec{y})$.*

Proof. Set $g = \lambda z\vec{y}.\prod_{x<z} f(x, \vec{y})$ for convenience.

Then the following primitive recursion settles it.

$$\begin{aligned}
g(0, \vec{y}) &= 1 \\
g(z+1, \vec{y}) &= g(z, \vec{y}) f(z, \vec{y})
\end{aligned}$$

□

2.4.6 Corollary. *If $\lambda x\vec{y}.f(x,\vec{y}) \in \mathcal{R}$ then so is $\lambda z\vec{y}.\prod_{x<z} f(x,\vec{y})$.*

2.4.7 Corollary. $\lambda xy.x * y \in \mathcal{PR}$.

Proof. Simply use 2.4.5, the discussion and notation following 2.4.2, and the results of 2.1.16 and 2.3.11 to see that $\lambda w.\prod_{i<lh(w)} p_{i+lh(z)}^{\exp(i,w)} \in \mathcal{PR}$. □

2.4.8 Example. $lh(1) = 0$. Thus even though $Seq(1)$ is false, 1 is a good candidate to code the *empty sequence*. More evidence that this is so is the fact that $1 * z = z * 1 = z$ for all z (Exercise, using 2.4.3). □

The reader most likely is familiar with the *stack* data structure from elementary programming.

2.4.9 Definition. (Stacks) A stack is an *ordered list* of *natural numbers*, where we may *access* (that is, read) its data only on *one* end of the list, called the *top of the stack*. We can also *add* to or *delete* from, *one item of data at a time*, but only at the top.

The terminology for adding to the stack is to *push* on the stack; that of deleting is to *pop* from the stack.

Adding creates a new top (the old top is "below" the new one after the addition. Deletion, if the stack is not *empty*, will remove the *current* top item. The new top is the item that was below the item we deleted, before the deletion. □

2.4.10 Remark. In a course in programming a stack may contain elements of any data type that the programming language in hand allows. But we only use natural numbers in this volume.

In our computability theory it is easy to implement a stack using prime power coding. If the elements of the stack s are in the sequence $a_0, a_1, \ldots, a_n$, with a_n being at the top, then

$$s = \langle a_0, a_1, \ldots, a_n \rangle$$

The top, expressed as a function of s, is $(s)_{lh(s)\dot{-}1}$ if the stack is not empty ($\neq 1$), else it is undefined.

The three operations —in C-like pseudo-code— are implemented as follows:

- *read the top and assign to* $\mathbf{x}$: if $s > 1$ (s non-empty), then $\mathbf{x} \leftarrow (s)_{lh(s)\dot{-}1}$
- *push* $b \in \mathbb{N}$ *to the stack*: $s \leftarrow s * \langle b \rangle$
- *pop the stack*: If $s > 1$, then $s \leftarrow \left\lfloor s/p_{lh(s)\dot{-}1}^{\exp(lh(s)\dot{-}1,s)} \right\rfloor$.

  If we want to save into the variable $\mathbf{x}$ what we popped, then we do:

  If $s > 1$, then

  $\{$

```
    1. $\mathbf{x} \leftarrow (s)_{lh(s) \dot{-} 1}$
    2. pop $s$
}
```

□

2.4.2 Course-of-values recursion

Primitive recursion defines a function "at $n+1$" in terms of its value "at n". However we also have examples of "recursions" (or "recurrences"), one of the best known perhaps being the *Fibonacci sequence*, $0, 1, 1, 2, 3, 5, 8, \ldots$, that is given by $F_0 = 0$, $F_1 = 1$ and (for $n \geq 1$) $F_{n+1} = F_n + F_{n-1}$, where the value at $n+1$ depends on both the values at n and $n-1$.

We may also have recursions where the value at $n+1$ depends, in general, on the entire *history*, or *course-of-values*, of the function values at inputs $n, n-1, n-2, \ldots, 1, 0$. The easiest way to represent the entire history of values of a *total* number-theoretic function f "at (input) x", namely, the set $\{f(0, \vec{y}), f(1, \vec{y}), \ldots, f(x, \vec{y})\}$, is to code it by a single number!

2.4.11 Definition. (Course-of-Values Recursion) We say that f, of $n+1$ arguments, is defined from two total functions —namely, the *basis* function $\lambda \vec{y}_n.b(\vec{y}_n)$ and the *iterator* $\lambda x\vec{y}_n z.g(x, \vec{y}_n, z)$— by *course-of-values recursion* if, for all $x, \vec{y}_n$, the following equations hold:

$$\begin{cases} f(0, \vec{y}_n) & = b(\vec{y}_n) \\ f(x+1, \vec{y}_n) & = g(x, \vec{y}_n, H(x, \vec{y}_n)) \end{cases} \tag{1}$$

where $\lambda x\vec{y}_n.H(x, \vec{y}_n)$ is the *history function*, which is given "at x" (for all $\vec{y}_n$) by

$$\langle f(0, \vec{y}), f(1, \vec{y}), \ldots, f(x, \vec{y}) \rangle$$ □

2.4.12 Exercise. Prove that f given by (1) is total.
Hint. Use strong induction on x. □

The major result in this subsection is that both $\mathcal{PR}$ and $\mathcal{R}$ are closed under course-of-values recursion.

2.4.13 Theorem. *$\mathcal{PR}$ is closed under course-of-values recursion.*

Proof. So, let b and g be in $\mathcal{PR}$. We will show that $f \in \mathcal{PR}$. It suffices to prove that the history function H is primitive recursive, for then $f = \lambda x\vec{y}_n.\big(H(x, \vec{y}_n)\big)_x$ and we are done by Grzegorczyk substitution. To this end, the following equations —true for all $x, \vec{y}_n$— settle the case:

$$H(0, \vec{y}_n) = \langle b(\vec{y}_n) \rangle$$

$$H(x+1,\vec{y}_n) = H(x,\vec{y}_n)p_{x+1}^{g(x,\vec{y}_n,H(x,\vec{y}_n))+1} \qquad \Box$$

The same proof with trivial adjustments yields:

2.4.14 Corollary. *$\mathcal{R}$ is closed under course-of-values recursion.*

2.4.15 Example. The Fibonacci sequence, $(F_n)_{n\geq 0}$, is given by

$$F_0 = 0$$
$$F_1 = 1$$

otherwise, for $n > 0$,

$$F_{n+1} = F_n + F_{n-1}$$

The sequence can be viewed as the function $\lambda n.F_n$. As such it is in $\mathcal{PR}$.

Indeed, letting H_n be the history of the sequence at n —that is, $\langle F_0, \ldots, F_n\rangle$— we have the following course-of-values recursion for $\lambda n.F_n$ in terms of functions known to be in $\mathcal{PR}$.

$$\begin{aligned} F_0 &= 0 \\ F_{n+1} &= \text{if } n = 0 \text{ then } 1 \\ &\quad\ \text{else } \big(H_n\big)_n + \big(H_n\big)_{n \dot{-} 1} \end{aligned} \qquad \Box$$

2.4.3 Simultaneous primitive recursion

This recursion is instrumental toward the study of URM program and *loop program* computation. Loop programs are introduced in a later chapter.

2.4.16 Definition. Given total functions h_i, g_i, for $i = 0, 1, \ldots, k$. We say that the following equations-schema defines —for all $x, \vec{y}$— the new functions f_i from the given functions by a *simultaneous primitive recursion.*

$$\begin{cases} f_0(0,\vec{y}) & = h_1(\vec{y}) \\ \vdots & \\ f_k(0,\vec{y}) & = h_k(\vec{y}) \\ f_0(x+1,\vec{y}) & = g_0(x,\vec{y},f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})) \\ \vdots & \\ f_k(x+1,\vec{y}) & = g_k(x,\vec{y},f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})) \end{cases} \tag{2}$$

$\Box$

Hilbert and Bernays [HB68] proved the following:

2.4.17 Theorem. *If all the h_i and g_i are in $\mathcal{PR}$ (resp. $\mathcal{R}$), then so are all the f_i obtained by the schema (2) of simultaneous recursion.*

Proof. Define, for all x and $\vec{y}$,

$$F(x,\vec{y}) \overset{\text{Def}}{=} \langle f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})\rangle$$

$$H(\vec{y}_n) \overset{\text{Def}}{=} \langle h_0(\vec{y}),\ldots,h_k(\vec{y})\rangle$$

$$G(x,\vec{y},z) \overset{\text{Def}}{=} \langle g_0(x,\vec{y},(z)_0,\ldots,(z)_k),\ldots,g_k(x,\vec{y},(z)_0,\ldots,(z)_k)\rangle$$

We readily have that $H \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) and $G \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) depending on where we assumed the h_i and g_i to be. We can now rewrite schema (2) as

$$\begin{cases} F(0,\vec{y}) & = H(\vec{y}) \\ F(x+1,\vec{y}) & = G\Big(x,\vec{y},F\Big(x,\vec{y}\Big)\Big) \end{cases} \tag{3}$$

▶ The 2nd line of (3) is condenced from

$$\begin{aligned} F(x+1,\vec{y}) &= \langle f_0(x+1,\vec{y}),\ldots,f_k(x+1,\vec{y})\rangle \\ &= \Big\langle g_0\Big(x,\vec{y},f_0(x,\vec{y}),\ldots,f_k(x,\vec{y})\Big),\ldots,g_k\Big(\ldots\Big)\Big\rangle \\ &= \Big\langle g_0\Big(x,\vec{y},\big(F(x,\vec{y})\big)_0,\ldots,\big(F(x,\vec{y})\big)_k\Big),\ldots,g_k\Big(\ldots\Big)\Big\rangle \end{aligned}$$

By the above remarks, $F \in \mathcal{PR}$ (resp. $\in \mathcal{R}$) depending on where we assumed the h_i and g_i to be. In particular, this holds for each f_i since, for all $x, \vec{y}$, $f_i(x,\vec{y}) = \big(F(x,\vec{y})\big)_i$ (Grzegorczyk operations). □

2.4.18 Example. We saw one way to justify that $\lambda x.rem(x,2) \in \mathcal{PR}$ in 2.3.11. A direct way is the following. Setting $f(x) = rem(x,2)$, for all x, we notice that the sequence of outputs (for $x = 0,1,2,\ldots$) of f is

$$0,1,0,1,0,1\ldots$$

Thus, ignoring the result from 2.3.11, the following primitive recursion shows that $f \in \mathcal{PR}$:

$$\begin{cases} f(0) & = 0 \\ f(x+1) & = 1 \dot{-} f(x) \end{cases}$$

Here is a way, via simultaneous recursion, to obtain a proof that $f \in \mathcal{PR}$, *without using any arithmetic*! Notice the infinite "matrix"

$$\begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 & \ldots \\ 1 & 0 & 1 & 0 & 1 & 0 & \ldots \end{matrix}$$

Let us call g the function that has as its sequence of outputs the entries of the second row —obtained by shifting the first row by one position to the left. The first row still represents our f. Now

$$\begin{cases} f(0) & = 0 \\ g(0) & = 1 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) \end{cases} \tag{1}$$

$\square$

2.4.19 Example. We saw one way to justify that $\lambda x. \lfloor x/2 \rfloor \in \mathcal{PR}$ in 2.3.11. A direct way is the following.

$$\begin{cases} \left\lfloor \dfrac{0}{2} \right\rfloor & = 0 \\ \left\lfloor \dfrac{x+1}{2} \right\rfloor & = \left\lfloor \dfrac{x}{2} \right\rfloor + rem(x, 2) \end{cases}$$

where rem is in $\mathcal{PR}$ by 2.4.18.

Alternatively, here is a way that can do it —via simultaneous recursion— and with only the knowledge of how to add 1. Consider the matrix

$$\begin{array}{ccccccccc} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & \ldots \\ 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & \ldots \end{array}$$

The top row represents $\lambda x. \lfloor x/2 \rfloor$, let us call it "$f$". The bottom row we call g and is, again, the result of shifting row one to the left by one position. Thus, we have a simultaneous recursion

$$\begin{cases} f(0) & = 0 \\ g(0) & = 0 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) + 1 \end{cases} \tag{2}$$

$\square$

2.4.4 Simulating a URM computation

We have defined in 1.1.2 what a URM computation is. This subsection will deal with mathematically *simulating* a computation.

We can do so by pencil and paper if we are sufficiently patient to engage in a potentially infinite task.

What we need to do is keep track of the values of *all the variables* of the given URM as the computation process *reaches each* instruction. As we saw in 1.1.2, *at most one* variable changes at each instruction execution (instructions of types 1–3), while the **if-statement** and **stop** change no variable. Moreover, the **stop** instruction *practically ends* the computation, even if the latter, in theory, continues forever: *All variable contents have converged to* final *values once the execution reaches* **stop**.

In this subsection we will see good justification for allowing all computations to go forever, even after they reached the instruction **stop**.

The simulation simply *requires us to record* what *each* variable's value is when we *reach* an instruction "$L: \ldots$"

Thus we record an array of values for said instruction,

$$L; a_1, a_2, \ldots$$

where L is the label or *instruction counter* and the a_i are the values of *all* the (finitely many!) $\mathbf{x}_i$, the latter variables of the given URM being *ordered* in an *arbitrary* manner, but *fixed* for the length of the simulation.

This array is a snapshot of the computation, or an *instantaneous description*, for short *ID*.

One way to order the variables in the head row of the table below is lexicographically, seeing that each is a string $X1^n$, for $n \geq 0$.

The semantics in 1.1.2 allow us to build the next snapshot, since we know which instruction of M is labelled L.

A sequence of *discrete events* —such as *reaching* and recording an instruction in the course of a computation of a URM M— when arranged in a sequence are implicitly associated with their position-numbers in the sequence, $0, 1, 2, \ldots$

The event in position i we say occurred in *step* or *time* i. The very first event occurs in time 0, it is the event at which instruction 1 is reached.

In the table below, for each instruction reached, *we record the full ID.*

Our main tool for the simulation is a growing matrix like this:

Table 2.1: Simulation Table

y	IC	$\ldots$	$\mathbf{x}$	$\ldots$	**Comment**
0	1	$\ldots$	a	$\ldots$	$a = 0$ if $\mathbf{x}$ noninput
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	
i	L	$\ldots$	b	$\ldots$	use 1.1.2 to *add* the $(i+1)$st row (L', c, etc)
$i+1$	L'	$\ldots$	c	$\ldots$	
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	

Deliberately, the bottom bounding line is missing to convey that this matrix is growing —one row at a time— downwards *forever.*

The very first entry is the ID corresponding to reaching instruction 1 *for the very first time.* It is the *initial ID*:

$$1; a_1, a_2, \ldots$$

where each a_i corresponding to an *input* variable $\mathbf{x}_i$ is some input number, while all the other a_j equal 0.

So, at each *time*, or *step*, $y = 0, 1, 2, \ldots$, row y contains (a pointer "L" to) the instruction we need to do *next*, and what values *each* variable of M holds, *before* we execute said (next) instruction.

Let us fix a URM M. Say, $\mathbf{x}_i$, for $i = 1, \ldots m$, are *all* its variables, of which, without loss of generality, the first n are input ($n \leq m$). We define $\lambda y\vec{a}_n.X_i(y, \vec{a}_n)$, for $i = 1, \ldots m$, by

$$\text{For all } y, \vec{a}_n,\ X_i(y, \vec{a}_n) = \mathbf{x}_i \text{ contents at } \textit{time } y, \text{ if } \vec{a}_n \text{ was the } \textit{input.} \tag{1}$$

Similarly, we define $\lambda y\vec{a}_n.IC(y, \vec{a}_n)$ by

$$\text{For all } y, \vec{a}_n,\ IC(y, \vec{a}_n) = \text{the current label at time } y, \text{ if } \vec{a}_n \text{ was the } \textit{input.} \tag{2}$$

It turns out that these functions are definitionally very simple; enough so to be primitive recursive. To ensure that we can *prove* this, first off we must ensure they are *total.* This explains our choice to trivially continue a computation forever, even though it has reached **stop** . This convention causes all computations to have infinite length, but the *convergent* ones —that is, those that do reach **stop**— have all contents of variables converge to final values.

Thus $\lambda y\vec{a}_n.X_i(y, \vec{a}_n)$ and $\lambda y\vec{a}_n.IC(y, \vec{a}_n)$ are total.

2.4.20 Simulation Theorem.

Let M be a URM with variables $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_{n+1}, \mathbf{x}_{n+2}, \ldots, \mathbf{x}_m$, of which $\mathbf{x}_i$, for $i = 1, \ldots, n$, are input variables while $\mathbf{x}_1$ is also *the output variable.*

With reference to 1.1.2 and the above discussion, the simulating functions *$\lambda y\vec{a}_n.X_i(y, \vec{a}_n)$, for $i = 1, \ldots m$, and $\lambda y\vec{a}_n.IC(y, \vec{a}_n)$ are in $\mathcal{PR}$.*

Proof. We have the following simultaneous recursion that defines the simulating functions:

$$\begin{aligned} X_i(0, \vec{a}_n) &= a_i, \text{ for } i = 1, \ldots, n \\ X_i(0, \vec{a}_n) &= 0, \text{ for } i = n+1, \ldots, m \\ IC(0, \vec{a}_n) &= 1 \end{aligned}$$

For $y \geq 0$ and $i = 1, \ldots, m$,

$$X_i(y{+}1, \vec{a}_n) = \begin{cases} c & \text{if } IC(y, \vec{a}_n) = L \text{ and "} L : \mathbf{x}_i \leftarrow c \text{" is in } M \\ X_i(y, \vec{a}_n) + 1 & \text{if } IC(y, \vec{a}_n) = L \text{ and "} L : \mathbf{x}_i \leftarrow \mathbf{x}_i + 1 \text{" is in } M \\ X_i(y, \vec{a}_n) \dot{-} 1 & \text{if } IC(y, \vec{a}_n) = L \text{ and "} L : \mathbf{x}_i \leftarrow \mathbf{x}_i \dot{-} 1 \text{" is in } M \\ X_i(y, \vec{a}_n) & \text{othw} \end{cases}$$

The above if-parts are of necessity wordy and indirect, as we can only give descriptive conditions because *we have not "looked inside" M to be able to discuss a specific instruction set.* We will do so in two examples below.

In general, suffice it to say that for each $\mathbf{x}_i$ we scan the program M from top to bottom and write a case for each instruction of types 1–3 that involves $\mathbf{x}_i$ (1.1.1):

Thus, e.g., *if* the $k_j : \mathbf{x}_i = \mathbf{x}_i + 1$ (for $j = 1, \ldots, r$) are the *only* instructions in M that involve the "+1" instruction with $\mathbf{x}_i$, and *if* there are *no* " $\dot{-}\ 1$" instructions for this variable, *but* we have exactly two of these:

$$R: \ \mathbf{x}_i \leftarrow 2$$

$$Q: \ \mathbf{x}_i \leftarrow 5$$

then we will add the cases below (two for the above two instructions and r for the "+1" instructions):

$$X_i(y+1, \vec{a}_n) = \begin{cases} 2 & \text{if } IC(y, \vec{a}_n) = R \\ 5 & \text{if } IC(y, \vec{a}_n) = Q \\ \vdots & \vdots \\ X_i(y, \vec{a}_n) + 1 & \text{if } IC(y, \vec{a}_n) = k_j \\ \vdots & \vdots \\ X_i(y, \vec{a}_n) & \text{othw} \end{cases}$$

The descriptive text

"and '$L : \mathbf{x}_i \leftarrow \mathbf{x}_i + 1$' is in M"

has been replaced by the precise r cases "if $IC(y, \vec{a}_n) = k_j$", one for each $j = 1, \ldots, r$.

The recurrence for IC is

$$IC(y{+}1, \vec{a}_n) = \begin{cases} L' & \text{if } IC(y, \vec{a}_n) = L \text{ and ``}L : \textbf{if } \mathbf{x}_i = 0 \textbf{ goto } L' \textbf{ else} \\ & \textbf{goto } L''\text{'' is in } M \text{ and } X_i(y, \vec{a}_n) = 0 \\ L'' & \text{if } IC(y, \vec{a}_n) = L \text{ and ``}L : \textbf{if } \mathbf{x}_i = 0 \textbf{ goto } L' \textbf{ else} \\ & \textbf{goto } L''\text{'' is in } M \text{ and } X_i(y, \vec{a}_n) > 0 \\ k & \text{if } IC(y, \vec{a}_n) = k \text{ and ``}k : \textbf{stop}\text{'' is in } M \\ IC(y, \vec{a}_n) + 1 & \text{othw} \end{cases}$$

Again to remove the descriptive inexact nature of the conditions above, in general we will write two conditions for each if-statement in the program. Say we have only two if-statments in our M as shown below. Moreover, let us assume that **stop** has label k.

$\vdots \quad \vdots$
L : if $\mathbf{x}_5 = 0$ **goto** L' **else goto** L''
$\vdots \quad \vdots$
R : if $\mathbf{x}_{45} = 0$ **goto** R' **else goto** R''
$\vdots \quad \vdots$

Then we can make the definition of IC precise:

$$IC(y+1, \vec{a}_n) = \begin{cases} L' & \text{if } IC(y, \vec{a}_n) = L \wedge X_5(y, \vec{a}_n) = 0 \\ L'' & \text{if } IC(y, \vec{a}_n) = L \wedge X_5(y, \vec{a}_n) > 0 \\ R' & \text{if } IC(y, \vec{a}_n) = R \wedge X_{45}(y, \vec{a}_n) = 0 \\ R'' & \text{if } IC(y, \vec{a}_n) = R \wedge X_{45}(y, \vec{a}_n) > 0 \\ k & \text{if } IC(y, \vec{a}_n) = k \\ IC(y, \vec{a}_n) + 1 & \text{othw} \end{cases}$$

Since the iterator functions only utilize the functions $\lambda z.a$, $\lambda z.z+1$, $\lambda z.z \dot{-} 1$, $\lambda z.z$, and predicates $\lambda z.z = 0$, and $\lambda z.z > 0$ —all in $\mathcal{PR}$ and $\mathcal{PR}_*$— it follows that all the simulating functions are in $\mathcal{PR}$, as claimed. □

2.4.21 Example. Let M be the program below

1 : **if** $\mathbf{x}_2 = 0$ **goto** 5 **else goto** 2
2 : $\mathbf{x}_1 \leftarrow \mathbf{x}_1 + 1$
3 : $\mathbf{x}_2 \leftarrow \mathbf{x}_2 \dot{-} 1$
4 : **if** $\mathbf{x}_1 = 0$ **goto** 1 **else goto** 1
5 : **stop**

Let us assume that $\mathbf{x}_2$ is the input variable. The simulating equations take the concrete form below, where a denotes the input value:

$$X_1(0, a) = 0$$
$$X_2(0, a) = a$$
$$IC(0, a) = 1$$

For $y \geq 0$ we have

$$X_1(y+1, a) = \begin{cases} X_1(y, a) + 1 & \text{if } IC(y, a) = 2 \\ X_1(y, a) & \text{otherwise} \end{cases}$$

$$X_2(y+1, a) = \begin{cases} X_2(y, a) \dot{-} 1 & \text{if } I(y, a) = 3 \\ X_2(y, a) & \text{otherwise} \end{cases}$$

$$IC(y+1,a) = \begin{cases} 5 & \text{if } IC(y,a) = 1 \wedge X_2(y,a) = 0 \\ 2 & \text{if } IC(y,a) = 1 \wedge X_2(y,a) > 0 \\ 1 & \text{if } IC(y,a) = 4 \wedge X_1(y,a) = 0 \\ 1 & \text{if } IC(y,a) = 4 \wedge X_1(y,a) > 0 \\ 5 & \text{if } I(y,a) = 5 \\ IC(y,a)+1 & \text{otherwise} \end{cases}$$

□

2.4.22 Example. Let M be the program below

```
1 : $\mathbf{x}_1 \leftarrow \mathbf{x}_1 + 1$
2 : $\mathbf{x}_1 \leftarrow \mathbf{x}_1 \dot{-} 1$
3 : $\mathbf{x}_1 \leftarrow a$
4 : if $\mathbf{x}_1 = 0$ goto 1 else goto 1
5 : stop
```

where $\mathbf{x}_1$ is the input variable. As of necessity, $\mathbf{x}_1$ is the output variable as well. The simulation equations are

$$X_1(0,a) = 0$$
$$IC(0,a) = 1$$

For $y \geq 0$ we have

$$X_1(y+1,a) = \begin{cases} X_1(y,a)+1 & \text{if } IC(y,a) = 1 \\ X_1(y,a) \dot{-} 1 & \text{if } IC(y,a) = 2 \\ a & \text{if } IC(y,a) = 3 \\ X_1(y,a) & \text{otherwise} \end{cases}$$

$$IC(y+1,a) = \begin{cases} 1 & \text{if } IC(y,a) = 4 \wedge X_1(y,a) = 0 \\ 1 & \text{if } IC(y,a) = 4 \wedge X_1(y,a) > 0 \\ 5 & \text{if } I(y,a) = 5 \\ IC(y,a)+1 & \text{otherwise} \end{cases}$$

□

Here is ubiquitous application:

2.4.23 Theorem. *For every URM $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ —where the input / output choice has been indicated— there is a primitive recursive predicate $T_M(\vec{a}_n, y)$ —depending on M— and a primitive recursive "output function" $\lambda y\vec{a}_n.out_M(y, \vec{a}_n)$ —also depending on M— that behave as follows*

$$T_M(\vec{a}_n, y) \equiv \textit{the URM } M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1} \textit{ has reached } \textbf{\textit{stop}} \textit{ in } y \textit{ steps}$$

and

$$M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1} = \lambda \vec{a}_n.out_M\Big((\mu y)T_M(\vec{a}_n, y), \vec{a}_n\Big) \qquad (1)$$

Proof. Let X_i be the simulating functions for the variables of M and IC as above. Let **stop** be the k-th instruction of M. Then

$$T_M(\vec{a}_n, y) \equiv IC(y, \vec{a}_n) = k$$

and

$$out_M(y, \vec{a}_n) = X_1(y, \vec{a}_n), \text{ for all } y, \vec{a}_n$$

Now, if $(\exists y)T_M(\vec{a}_n, y)$, then $(\mu y)T_M(\vec{a}_n, y)$ finds the smallest such y.

Of course, $\lambda\vec{a}_n.(\mu y)T_M(\vec{a}_n, y) \in \mathcal{P}$. The validity of (1) is now obvious if we take $out_M = X_1$. □

2.4.24 Remark. The predicates below are all in $\mathcal{PR}_*$ by closure operations (2.2.5 and 2.2.14)

- $\neg T_M(\vec{a}_n, y)$; says "URM $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ with input $\vec{a}_n$ *did not yet reach* **stop** in y steps". Could it not be that $T_M(z, \vec{a}_n)$, for $z < y$? No, because of the **stop** semantics (1.1.2).

- $$T_M(\vec{a}_n, y) \wedge (\forall z)_{<y}\neg T_M(\vec{a}_n, z)$$

  says; "URM $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ with input $\vec{a}_n$ *first* reached **stop** in y steps".

- $(\exists z)_{<y}T_M(\vec{a}_n, z)$; says "URM $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ with input $\vec{a}_n$ reached **stop** *before* y steps".

For short, the three bullets above can be rephrased in plainer English (in order):

- The URM $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ with input $\vec{a}_n$ requires $> y$ steps to converge (if it does converge).

- The URM $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ with input $\vec{a}_n$ converges in a minimum of y steps.

- The URM $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ with input $\vec{a}_n$ converges in $< y$ steps. □

2.4.25 Corollary. *Let $M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$ be as above and let $\lambda\vec{a}_n.f(\vec{a}_n) = M^{\vec{\mathbf{x}}_n}_{\mathbf{x}_1}$. Then with T_M as in 2.4.23 we have, for all $\vec{a}_n$,*

$$f(\vec{a}_n)\downarrow \equiv (\exists y)T_M(\vec{a}_n, y)$$

and

$$f(\vec{a}_n) = out_M\Big((\mu y)T_M(\vec{a}_n, y), \vec{a}_n\Big)$$

2.4.5 Pairing functions

Coding of sequences $a_0, a_1, \ldots, a_n$, for $n \geq 1$, has a special case; pairing functions, that is, the case of $n = 2$. This special case is important in various theoretical considerations, especially when one wants to distance oneself from prime-power coding due to the latter's inefficiency, as compared to coding pairs with polynomial functions (rather than exponential).

2.4.26 Definition. A *total, 1-1* function $J : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is called a *pairing function.* $\square$

2.4.27 Remark. A decoder is a pair of **total** functions K, L from $\mathbb{N} \to \mathbb{N}$ such that, for any z that is equal to $J(x, y)$ for some x and y, they "compute" said x and y:

$$K(z) = x$$

and

$$L(z) = y$$

That is, K and L decode appropriate z values.

On a non-code z, K and L give some **nonsense answer** —but answer they will; they are total!— just as the decoder $\lambda zi.(z)_i$ does when $Seq(z)$ is false.

One usually encounters the (capital) letters K, L in the literature as (generic) names for projection functions of some (generic) pairing function. In turn, the generic symbol for the latter is J rather than "f". We will conform to this notational convention in what follows. $\square$

The set of "tools" consisting of a pairing function J and its two projections K and L is a coding/decoding scheme for sequences of length two. We want to have computable such schemes and indeed there is an abundance of *primitive recursive* pairing functions that also have primitive recursive projections.

Some of those we will indicate in the examples below and others we will let the reader to discover in the exercises section.

2.4.28 Example. The function $J = \lambda xy.\, \langle x, y \rangle$ is pairing. Good decoders/projections are $K = \lambda z.(z)_0$ and $L = \lambda z.(z)_1$. All three are already known to us as members of $\mathcal{PR}$.

This J is not onto. For example, $5 \notin \text{ran}(J)$. Nevertheless, K and L are total —because $\lambda iz.(z)_i$ is; indeed is in $\mathcal{PR}$. $\square$

2.4.29 Example. The function $J = \lambda xy.2^x 3^y$ is pairing. As its projections we normally take $K = \lambda z.\exp(0, z)$ and $L = \lambda z.\exp(1, z)$ (cf. 2.3.11). All three are already known to us as members of $\mathcal{PR}$.

This J is not onto either. Again, $5 \notin \text{ran}(J)$. Nevertheless, K and L are total —because $\lambda iz.\exp(i, z)$ is; indeed is in $\mathcal{PR}$. $\square$

2.4.30 Example. The function $J = \lambda xy.2^x(2y+1)$ due to Grzegorczyk is pairing. Indeed, since 2 is the only even prime, "$2^x(2y+1)$" is a forgetful depiction of a number's prime-number decomposition, where all powers of odd primes are lumped together in "$2y+1$". Clearly it is in $\mathcal{PR}$ since we have addition, multiplication, successor and $\lambda x.2^x$ in $\mathcal{PR}$.

$K = \lambda z.\exp(0,z)$ works: If $z = 2^x(2y+1)$, then $Kz = x$.[†] What is L in closed form? (*Hint.* You may use functions from 2.3.11.) □

2.4.31 Example. In 2.4.29 and 2.4.30 we note that $J(x,y) \ge x$ and $J(x,y) \ge y$, for all x, y. Thus an alternative way to prove that the related K and L are in $\mathcal{PR}$ is to compute as follows:

$$Kz = (\mu x)_{\le z}(\exists y)_{\le z}(J(x,y) = z) \tag{1}$$

and

$$Lz = (\mu y)_{\le z}(\exists x)_{\le z}(J(x,y) = z) \tag{2}$$

Equipped with theorems 2.2.14 and 2.3.2, and Definition 2.3.9, we see that (1) and (2) establish that K and L are in $\mathcal{PR}$. □

2.4.32 Example. Here is a pairing function that does not require exponentiation. Let $J(x,y) = (x+y)^2 + x$. Clearly, $J \in \mathcal{PR}$.

So let us set $z = (x+y)^2 + x$ and solve this "equation" for x and y (uniquely, hopefully). Well, $(x+y)^2 \le z < (x+y+1)^2$. Thus $x+y \le \sqrt{z} < x+y+1$, hence

$$x + y = \lfloor \sqrt{z} \rfloor \tag{1}$$

Then, $z = \lfloor \sqrt{z} \rfloor^2 + x$ and therefore $Kz = z \dot{-} \lfloor \sqrt{z} \rfloor^2$. By (1), $Lz = \lfloor \sqrt{z} \rfloor \dot{-} Kz$.

As in 2.4.31, the J here satisfies $J(x,y) \ge x$ and $J(x,y) \ge y$.

The primitive recursiveness of K, L also follows from the calculations $Kz = (\mu x)_{\le z}(\exists y)_{\le z}(J(x,y) = z)$ and $Lz = (\mu y)_{\le z}(\exists x)_{\le z}(J(x,y) = z)$. □

Why bother about pairing functions when we have the coding of sequences scheme of the previous subsection? Because prime-power coding is computationally very inefficient, while quadratic schemes such as that of the previous example allow us to significantly reduce the "computational complexity" of coding/decoding. But can we code arbitrary length sequences efficiently?

Yes, because any J, K, L scheme can lead to a coding/decoding scheme for sequences $a_1, \ldots, a_n$, $n \ge 2$, for both the cases of a fixed or variable n.

2.4.33 Definition. Given a primitive recursive pairing scheme J, K, L.

For any fixed $n \ge 2$ we define by recursion on n the symbol $[\![a_1, \ldots, a_n]\!]^{(n)}$: $[\![x, y]\!]^{(2)} = J(x,y)$; and $[\![x, y_1, \ldots, y_n]\!]^{(n+1)} = J(x, [\![y_1, \ldots, y_n]\!]^{(n)})$. □

2.4.34 Exercise. Prove by induction on $n \ge 2$ that,

[†]In 2.1.8 we agreed to omit brackets whenever we can get away with it, as in SSx and here as in Kx.

- The function $\lambda \vec{x}_n . [\![a_1, \ldots, a_n]\!]^{(n)}$ is total
- The function $\lambda \vec{x}_n . [\![a_1, \ldots, a_n]\!]^{(n)}$ is 1-1
- The function $\lambda \vec{x}_n . [\![a_1, \ldots, a_n]\!]^{(n)}$ is primitive recursive.

Therefore it codes n-tuples into numbers. $\square$

2.4.35 Example. By the second and third bullets above, we can define Π_i^n, for $i = 1, \ldots, n$ —the projections— to satisfy

$$\text{If } z = [\![a_1, \ldots, a_n]\!]^{(n)}, \text{ then } \Pi_i^n(z) = a_i, \text{ for } i = 1, \ldots, n \tag{1}$$

We may use a recursive definition with n as the recursion variable:

$$\begin{aligned} \Pi_1^2 &= K \\ \Pi_2^2 &= L \\ &\text{and, for } n \geq 2 \\ \Pi_1^{n+1} &= K \\ \Pi_{i+1}^{n+1} &= \Pi_i^n L, \text{ for } i = 1, \ldots, n \end{aligned}$$

$\square$

2.4.36 Exercise. By induction on n, show that

- $\Pi_i^n\Big([\![a_1, \ldots, a_n]\!]^{(n)} \Big) = a_i$, for $i = 1, \ldots, n$.
- The Π_i^n are total.
- The Π_i^n are onto.
- The Π_i^n are in $\mathcal{PR}$. $\square$