Prefix sum circuits

Final exam, MPRI 2-4

March 6, 2024 — Duration: 2h45

Answers are judged by their correctness, clarity, conciseness, and accuracy. Although the questions are in English, it is permitted to answer in French or English. It is recommended to answer in French if this is your mother tongue.

Throughout this exam, we will be writing programs in pure F^ω, whose terms, types and kinds are specified through the following syntactic categories:

$t, u ::=$		(Terms)
	x	(term variable)
	$\lambda\, x.\, t$	(term function)
	$t\, u$	(term application)
	(t, u)	(pair)
	$\mathsf{fst}\, t \mid \mathsf{snd}\, t$	(first and second projections)
	$()$	(unit element)

$\tau, \sigma ::=$		(Types)
$\mid$	α	(type variable)
$\mid$	$\tau \to \sigma$	(function type)
$\mid$	$\tau * \sigma$	(product type)
$\mid$	unit	(unit type)
$\mid$	$\forall\, \alpha :: \kappa.\, \tau$	(universal quantification)
$\mid$	$\lambda\, \alpha.\, \tau$	(type function)
$\mid$	$\tau\, \sigma$	(type application)

$\kappa, \mu ::=$		(Kinds)
$\mid$	$\star$	(kind of all types)
$\mid$	$\kappa \Rightarrow \mu$	(function kind)

subject to the usual type system and reduction semantics. Note that, to lighten notation, type abstraction and type instantiation are kept implicit in the term syntax.

F^ω being strongly normalizing, it enjoys the strong Church-Rosser property: every reduction sequence leads to a normal form and normal forms are unique. In effect, this means that we can legitimately reason by equational reasoning (we can replace equals by equals, independently of context) without having to worry about effects or divergence.

1 Algebraic Structures

We recall that a type constructor $F : \star \Rightarrow \star$ is said to be a ***functor*** if it is equipped with an action map_F of type $\forall\, \alpha\ \beta :: \star.\ (\alpha \to \beta) \to F\ \alpha \to F\ \beta$ that satisfies the functor laws:

Map-Id: for all types τ and for all terms t of type $F\ \tau$, we have $map_F\ (\lambda\, x.\ x)\ t = t$.

Map-Compose: for all types τ, σ, ν, for all terms t of type $F\ \tau$, f of type $\tau \to \sigma$ and g of type $\sigma \to \nu$, we have $map_F\ g\ (map_F\ f\ t) = map_F\ (\lambda\, x.\ g\ (f\ x))\ t$.

Question 1 *Let (F, map_F) be a functor.*
Implement a program of type

$$\forall\, \alpha\ \beta :: \star.\ F\ (\alpha * \beta) \to F\ \alpha * F\ \beta$$

Answer:

There is not much choice but to implement a generalized "split" operation:

$$split_F \triangleq \lambda\, x.\ (map_F\ \mathsf{fst}\ x, map_F\ \mathsf{snd}\ x)$$

□

We define the type-level constant and identity functions:

$$\begin{array}{lcl} \mathsf{Unit} & :: & \star \to \star \\ \mathsf{Unit} & \triangleq & \lambda\, \alpha.\ \mathsf{unit} \end{array} \qquad \begin{array}{lcl} \mathsf{Id} & :: & \star \to \star \\ \mathsf{Id} & \triangleq & \lambda\, \alpha.\ \alpha \end{array}$$

Both type constructors are functors, *i.e.* we have

$$\begin{array}{lcl} map_{\mathsf{Unit}} & : & \forall\, \alpha\ \beta :: \star.\ (\alpha \to \beta) \to \mathsf{Unit}\ \alpha \to \mathsf{Unit}\ \beta \\ map_{\mathsf{Unit}} & \triangleq & \lambda\, f.\ \lambda\, ().\ () \end{array}$$

$$\begin{array}{lcl} map_{\mathsf{Id}} & : & \forall\, \alpha\ \beta :: \star.\ (\alpha \to \beta) \to \mathsf{Id}\ \alpha \to \mathsf{Id}\ \beta \\ map_{\mathsf{Id}} & \triangleq & \lambda\, f.\ \lambda\, x.\ f\ x \end{array}$$

The functor laws are easily verified for map_{Unit} (since it does not actually depend on the function). The definition map_{Id} satisfies Map-Id since, for any function $f : \alpha \to \beta$ and any term $t : \mathsf{Id}\ \alpha$, we have:

$$\begin{array}{ll} map_{\mathsf{Id}}\ (\lambda\, x.\ x)\ t & \\ \quad = (\lambda\, x.\ x)\ t & \text{(by definition of } map_{\mathsf{Id}}\text{)} \\ \quad = t & \text{(by reduction)} \\ \text{(QED.)} & \end{array}$$

and it satisfies Map-Compose since, for any pair of functions $f : \alpha \to \beta$, $g : \beta \to \gamma$ and any term $t : \mathsf{Id}\ \alpha$, we have:

$$\begin{array}{ll} map_{\mathsf{Id}}\ g\ (map_{\mathsf{Id}}\ f\ t) & \\ \quad = g\ (map_{\mathsf{Id}}\ f\ t) & \text{(by definition of } map_{\mathsf{Id}}\text{)} \\ \quad = g\ (f\ t) & \text{(by definition of } map_{\mathsf{Id}}\text{)} \\ \quad = (\lambda\, x.\ g\ (f\ x))\ t & \text{(by reduction)} \\ \quad = map_{\mathsf{Id}}\ (\lambda\, x.\ g\ (f\ x))\ t & \text{(by definition of } map_{\mathsf{Id}}\text{)} \\ \text{(QED.)} & \end{array}$$

Let $F, G :: \star \Rightarrow \star$ be two functors. We define their product as

$$\begin{array}{lcl} \mathsf{Prod}\ F\ G & :: & \star \to \star \\ \mathsf{Prod}\ F\ G & \triangleq & \lambda\, \alpha.\ (F\ \alpha) * (G\ \alpha) \end{array}$$

Question 2 *Show that* $\mathsf{Prod}\ F\ G$ *is functorial by implementing its functorial action and proving that it satisfies the functor laws.*

Hint: when writing code samples, feel free to use the let-binding form $\mathsf{let}\ x = t\ \mathsf{in}\ u$*, which stands for* $(\lambda\, x.\ u)\ t$*.*

Answer:

We have

$$map_{\mathsf{Prod}\ F\ G} \triangleq \lambda\, f.\ \lambda\, (x, y).\ (map_F\ f\ x, map_G\ f\ y)$$

It satisfies Map-Id:

$$\begin{array}{l} map_{\mathsf{Prod}\ F\ G}\ (\lambda\, x.\ x)\ (t, u) \\ \quad \text{(definition of } map_{\mathsf{Prod}\ F\ G}) \\ = (map_F\ (\lambda\, x.\ x)\ t, map_G\ (\lambda\, x.\ x)\ u) \\ \quad \text{(application of Map-Id for } F \text{ and } G) \\ = (t, u) \end{array}$$

It satisfies Map-Compose:

$$\begin{array}{l} map_{\mathsf{Prod}\ F\ G}\ (f \circ g)\ (t, u) \\ \quad \text{(definition of } map_{\mathsf{Prod}\ F\ G}) \\ = (map_F\ (f \circ g)\ t, map_G\ (f \circ g)\ u) \\ \quad \text{(application of Map-Compose for } F \text{ and } G) \\ = (map_F\ f\ (map_F\ g\ t), map_G\ f\ (map_G\ g\ u)) \\ \quad \text{(definition of } map_{\mathsf{Prod}\ F\ G}) \\ = map_{\mathsf{Prod}\ F\ G}\ f\ (map_{\mathsf{Prod}\ F\ G}\ g\ (t, u)) \end{array}$$

□

We recall that a type M equipped with a binary operation $\cdot : M \to M \to M$ and a unary operation $\epsilon : M$ is a ***monoid*** if it satisfies

Left-Identity: for all $m : M$, we have $\epsilon \cdot m = m$.

Right-Identity: for all $m : M$, we have $m \cdot \epsilon = m$

Compose-Associative: for all $m, n, o : M$, we have

$$m \cdot (n \cdot o) = (m \cdot n) \cdot o$$

Programmatically, we package the two monoid operations in a dictionary:

$$\begin{array}{l} \mathit{IsMonoid} :: \star \Rightarrow \star \\ \mathit{IsMonoid} \triangleq \lambda\, M.\ (M \to M \to M) * M \end{array}$$

In the following, we assume the existence of a datatype of integers, written int. Notably, $(\mathit{int}, +, 0)$ forms a monoid.

A functor F is said to be ***foldable*** if it supports an operation fold_F of type

$$\forall\, M :: \star.\ \mathit{IsMonoid}\ M \to F\ M \to M$$

Given a foldable functor F, we define

$$\begin{array}{l} \mathit{card} : \forall\, \alpha :: \star.\ F\ \alpha \to \mathit{int} \\ \mathit{card} \triangleq \lambda\, t.\ \mathit{fold}_F\ (+, 0)\ (map_F\ (\lambda\ \_.\ 1)\ t) \end{array}$$

This program first functorially turns an $F\,\alpha$ into an $F\,\mathit{int}$, each element of type α being replaced with the integer "1". Then, the foldable instance is used to add these numbers across the whole structure. In effect, $F\,\alpha$ is understood as a data container storing elements of type α and the function *card* computes the number of elements in the container.

Question 3 *Let F, G be two foldable functors.*

1. *Propose an implementation of the operation $\mathit{fold}_{\mathsf{Unit}}$ for the type constructor* Unit *such that $\mathit{card}\ () = 0$.*
2. *Propose an implementation of the operation $\mathit{fold}_{\mathsf{Id}}$ for the type constructor* Id *such that, for any term $t : \tau$, we have $\mathit{card}\ t = 1$.*
3. *Propose an implementation of the operation $\mathit{fold}_{\mathsf{Prod}\ F\ G}$ for the type constructor* Prod F G *such that, for any terms $t : F\,\tau$ and $u : G\,\tau$, we have*

$$\mathit{card}\ (t, u) = \mathit{card}\ t + \mathit{card}\ u$$

.
Answer:

$$\begin{array}{l}
\mathit{fold}_{\mathsf{Unit}} : \forall\ M :: \star.\ \mathit{IsMonoid}\ M \to \mathsf{Unit}\ M \to M \\
\mathit{fold}_{\mathsf{Unit}} \triangleq \lambda\ (\cdot, \epsilon)\ ().\ \epsilon
\end{array}$$

$$\begin{array}{l}
\mathit{fold}_{\mathsf{Id}} : \forall\ M :: \star.\ \mathit{IsMonoid}\ M \to \mathsf{Id}\ M \to M \\
\mathit{fold}_{\mathsf{Id}} \triangleq \lambda\ (\cdot, \epsilon)\ m.\ m
\end{array}$$

$$\begin{array}{l}
\mathit{fold}_{\mathsf{Prod}\ F\ G} : \forall\ M :: \star.\ \mathit{IsMonoid}\ M \to \mathsf{Prod}\ F\ G\ M \to M \\
\mathit{fold}_{\mathsf{Prod}\ F\ G} \triangleq \lambda\ (\cdot, \epsilon)\ (ms, ns).\ \mathit{fold}_F\ (\cdot, \epsilon)\ ms \cdot \mathit{fold}_G\ (\cdot, \epsilon)\ ns
\end{array}$$

□

Given a foldable functor F, we define its ***cardinality*** $|F|$ to be the integer s, if it exists, such that for all types τ and all terms $t : F\,\tau$, we have $\mathit{card}\ t = s$. In effect, $|F|$ is defined if and only if the functor F represents a data container of statically-known size. For instance, we have $|\mathsf{Unit}| = 0$ and $|\mathsf{Id}| = 1$. Similarly, we have $|\mathsf{Prod}\ F\ G| = |F| + |G|$, assuming the cardinalities of F and G are defined.

2 Sequential prefix sum

In the following, we assume the existence of a datatype of lists, written $\mathit{List} :: \star \Rightarrow \star$. We follow the standard OCaml notations for its constructors and pattern-matching constructs. In keeping with our deference to termination, we shall nonetheless restrict ourselves to structurally recursive definitions. As is well known, *List* is a functor.

Question 4 *Propose an implementation of the operation $\mathit{fold}_{\mathit{List}}$ for the type constructor* List.
Answer:

$$\begin{array}{l}
\mathit{fold}_{\mathit{List}} : \forall\ M :: \star.\ \mathit{IsMonoid}\ M \to \mathit{List}\ M \to M \\
\mathit{fold}_{\mathit{List}} \triangleq \lambda\ (\cdot, \epsilon)\ x.\ \mathsf{match}\ x\ \mathsf{with}\ \left\{ \begin{array}{ll} [].\ & \epsilon \\ x :: y.\ & x \cdot \mathit{fold}_{\mathit{List}}\ (\cdot, \epsilon)\ y \end{array} \right.
\end{array}$$

□

Question 5 *Is* $|List|$*, the cardinality of* $List$*, defined? If so, what is its value?*
Answer:

Undefined, because lists do not have statically known size.

□

The prefix sum is informally specified as follows: given an input consisting of n elements $[m_0, m_1, \ldots, m_n]$ of a monoid $(M, \cdot, \epsilon)$, the prefix sum operation consists in computing the $n+1$ outputs elements $([m'_0, m'_1, \ldots, m'_n], m_{out})$ defined by

$$\begin{aligned} m'_0 &= \epsilon \\ m'_1 &= \epsilon \cdot m_0 \\ m'_2 &= \epsilon \cdot m_0 \cdot m_1 \\ &\ldots \\ m'_n &= \epsilon \cdot m_0 \cdot m_1 \ldots \cdot m_{n-1} \\ m'_{out} &= \epsilon \cdot m_0 \cdot m_1 \ldots \cdot m_{n-1} \cdot m_n \end{aligned}$$

The prefix sum has some very broad applications, such as polynomial evaluation, sorting, lexical analysis, regular expression search, *etc.* The ability to efficiently parallelize this computation has lead to significant research in the field of parallel algorithms, targeting both hardware and software (GPU, clusters) platforms. In the present section, we focus on a purely sequential implementation.

We model this operation through a function $\mathit{prefix\text{-}sum}_{List}$ of type

$$\forall\, M :: \star.\ \mathit{IsMonoid}\ M \to \mathit{List}\ M \to \mathit{List}\ M * M$$

where the second component of the result corresponds to the distinguished element m'_{out}.

Question 6 *Let* $(M, \cdot, \epsilon)$ *be a monoid and* m, n, o *be elements of this monoid.*
Interpreting the description of prefix sum above, express the result of

1. $\mathit{prefix\text{-}sum}_{List}\ (\cdot, \epsilon)\ []$
2. $\mathit{prefix\text{-}sum}_{List}\ (\cdot, \epsilon)\ [m]$
3. $\mathit{prefix\text{-}sum}_{List}\ (\cdot, \epsilon)\ [m; n; o]$

in terms of the monoid elements m*,* n*,* o*, monoid composition* $\cdot$ *and monoid identity* ϵ*.*
Answer:

1. $\mathit{prefix\text{-}sum}_{List}\ (\cdot, \epsilon)\ [] = ([], \epsilon)$
2. $\mathit{prefix\text{-}sum}_{List}\ (\cdot, \epsilon)\ [m] = ([\epsilon], \epsilon \cdot m)$
3. $\mathit{prefix\text{-}sum}_{List}\ (\cdot, \epsilon)\ [m; n; o] = ([\epsilon; \epsilon \cdot m; \epsilon \cdot m \cdot n], \epsilon \cdot m \cdot n \cdot o)$

□

3 Algebraic interlude

A functor $T :: \star \Rightarrow \star$ is said to be ***applicative*** if it supports an operation *pure* of type $\forall\, \alpha :: \star.\ \alpha \to T\,\alpha$ and a binary operation $\circledast$ of type $\forall\, \alpha\ \beta :: \star.\ T\,(\alpha \to \beta) \to T\,\alpha \to T\,\beta$ subject to the laws seen in class, which play no role in the present exam. We package these two operations in a dictionary

$$\begin{aligned} &\mathit{IsApplicative} :: (\star \Rightarrow \star) \Rightarrow \star \\ &\mathit{IsApplicative} \triangleq \lambda\, T.\ (\forall\, \alpha :: \star.\ \alpha \to T\,\alpha) * (\forall\, \alpha\ \beta :: \star.\ T\,(\alpha \to \beta) \to T\,\alpha \to T\,\beta) \end{aligned}$$

Finally, a foldable functor F is said to be ***traversable*** if it supports an operation $\mathit{traverse}_F$ of type

$$\forall\, T :: \star \Rightarrow \star.\, \forall\, \alpha\ \beta :: \star.\, \mathit{IsApplicative}\ T \to (\alpha \to T\ \beta) \to F\ \alpha \to T\ (F\ \beta)$$

which we abbreviate as $\mathit{IsTraversable}\ F$ in the following.

Question 7 *We claim that traversable functors offer a generic notion of accumulated map. Given a function $\alpha \to \gamma \to \beta * \gamma$ modeling a computation taking input of type α, producing output of type γ and keeping a local state of type β, an accumulated map takes a value of type $F\ \alpha$ to an output of type $F\ \gamma$ together with the final state of the computation, of type β.*

Concretely, this amounts to the following tentative definition:

$$\begin{aligned}&\mathit{map\text{-}accum} : \forall\, \alpha\ \beta\ \gamma :: \star.\, \forall\, F :: \star \Rightarrow \star.\\ &\qquad \mathit{IsTraversable}\ F \to (\alpha \to \gamma \to \beta * \gamma) \to \gamma \to F\ \alpha \to F\ \beta * \gamma\\ &\mathit{map\text{-}accum} \triangleq \lambda\, \mathit{traverse}_F\ f\ x\ \mathit{as}.\ \mathit{traverse}_F\ ?\ f\ \mathit{as}\ x\end{aligned}$$

where a question mark has been inserted where we should have instantiated the applicative functor supporting this computation. Identify the functor used in this computation and define its applicative structure.

Answer:

The functor is $\mathit{State}_\gamma \triangleq \lambda\, \beta.\, \gamma \Rightarrow \beta * \gamma$ which corresponds to the state monad, hence, a fortiori an applicative.

□

4 Parallel prefix sum

Being based on the *List* datastructure, the implementation explored in Section 2 is inherently sequential. In the following, we develop alternative choices of datastructure.

From a modeling standpoint, the challenge consists in appropriately expressing the notion of a collection of n elements. From a programming standpoint, the challenge consists in enabling parallel computation, as the definition of prefix sum exhibits both a significant amount of redundancy (due to the running monoidal sums) as well as parallelization opportunities (due to associativity of monoid composition).

To address both aspects, we resort to traversable functors to model collections of elements. We shall say that a traversable functor F admits a ***prefix sum*** if it supports an operation $\mathit{prefix\text{-}sum}_F$ of type

$$\forall\, M :: \star.\, \mathit{IsMonoid}\ M \to F\ M \to F\ M * M$$

such that for any monoid $(M, \cdot, \epsilon)$ and term t of type $F\ M$, we have

$$\begin{aligned}&\mathit{prefix\text{-}sum}_F\ (\cdot, \epsilon)\ t\\ &\quad = \mathit{map\text{-}accum}\ (\_ : \mathit{IsTraversable}\ F)\ (\lambda\, x\ \mathit{acc}.\ (\mathit{acc}, \mathit{acc} \cdot x))\ \epsilon\ t\end{aligned}$$

Note that this specification is still as computationally inefficient as the one studied in Section 2: it is strictly sequential and highly redundant.

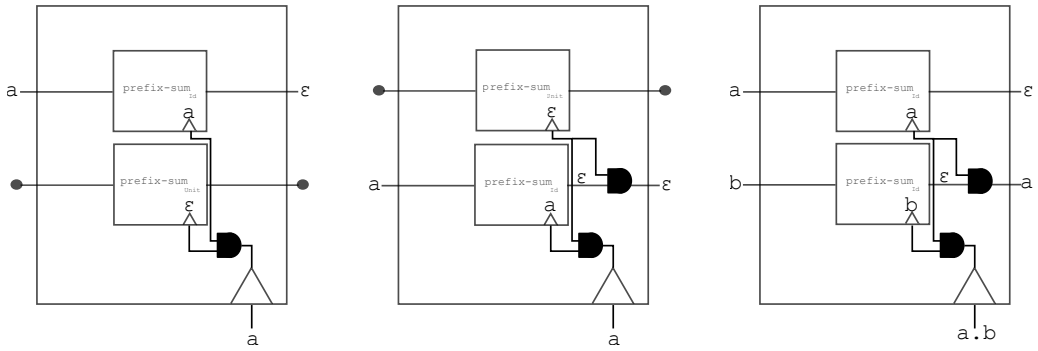
Given a functor F of cardinality n, its prefix sum can be understood as a circuit taking its n input wires (the elements of $F\ M$) to $n+1$ output wires (the elements of $F\ M * M$) whose gates consists solely of $\cdot$ operations (two inputs, one output). The efficiency of a parallel

prefix sum is measured in terms of the amount of ***work*** – defined as the total number of $\cdot$ gates used – and the ***depth*** of the computation – defined as the maximal number of $\cdot$ gates on the path from an input to an ouput. Designing a prefix sum computation is a trade-off between parallelism (aiming to reduce the depth) and work duplication.

Question 8 *Let F, G be two traversable functors each equipped with a prefix sum operation, respectively prefix-sum$_F$ and prefix-sum$_G$.*

1. *Show that the functor* Unit *supports a prefix sum operation. What is the work —written work*(Unit)*— and depth —written depth*(Unit)*— of your implementation?*
2. *Show that the functor* Id *supports a prefix sum operation. What is the work —written work*(Id)*— and depth —written depth*(Id)*— of your implementation?*
3. *Show that the product functor* Prod F G *supports a prefix sum operation. Express the work —written work*(Prod F G)*— and depth —written depth*(Prod F G)*— of your implementation in terms of work*(F)*, work*(G)*, depth*(F)*, depth*(G)*, $|F|$ and $|G|$.*

Hint: diagrammatically, the circuits corresponding to the prefix sum of Prod Id Unit *(left),* Prod Unit Id *(center) and* Prod Id Id *(right) are represented as follows:*



.

Answer:

We have:

$$\begin{aligned} &\textit{prefix-sum}_{\mathsf{Unit}} : \forall\, M :: \star.\ \textit{IsMonoid}\ M \to \mathsf{unit} \to \mathsf{unit} * M \\ &\textit{prefix-sum}_{\mathsf{Unit}} \triangleq \lambda\, (\cdot, \epsilon)\ ().\ ((), \epsilon) \end{aligned}$$

Therefore:

$$\begin{aligned} \mathrm{work}(\mathsf{Unit}) &= 0 \\ \mathrm{depth}(\mathsf{Unit}) &= 0 \end{aligned}$$

We have:

$$\begin{aligned} &\textit{prefix-sum}_{\mathsf{Id}} : \forall\, M :: \star.\ \textit{IsMonoid}\ M \to M \to M * M \\ &\textit{prefix-sum}_{\mathsf{Id}} \triangleq \lambda\, (\cdot, \epsilon)\ x.\ (\epsilon, x) \end{aligned}$$

Therefore:

$$\begin{aligned} \mathrm{work}(\mathsf{Id}) &= 0 \\ \mathrm{depth}(\mathsf{Id}) &= 0 \end{aligned}$$

We have:

$$\begin{aligned}
&\mathit{prefix\text{-}sum}_{\mathsf{Prod}\ F\ G} : \forall\ M :: \star.\ \mathit{IsMonoid}\ M \to F\ M * G\ M \to (F\ M * G\ M) * M \\
&\mathit{prefix\text{-}sum}_{\mathsf{Prod}\ F\ G} \triangleq \lambda\ (\cdot, \epsilon)\ (fa, ga). \\
&\qquad \mathsf{let}\ (fa', x) = \mathit{prefix\text{-}sum}_F\ _\ fa\ \mathsf{in} \\
&\qquad \mathsf{let}\ (ga', y) = \mathit{prefix\text{-}sum}_G\ _\ ga\ \mathsf{in} \\
&\qquad ((fa', \mathit{map}_G\ (x\ \cdot)\ ga'), x \cdot y)
\end{aligned}$$

Therefore:

$$\begin{aligned}
\mathrm{work}(\mathsf{Prod}\ F\ G) &= 1 + \mathrm{work}(F) + \mathrm{work}(G) + |G| \\
\mathrm{depth}(\mathsf{Prod}\ F\ G) &= 1 + \max(\mathrm{depth}(F), \mathrm{depth}(G))
\end{aligned}$$

□

We postulate the existence of two families of type constructors $(\mathit{LVec}_s)_{s\in\mathbb{N}}$ and $(\mathit{RVec}_s)_{s\in\mathbb{N}}$ characterized by the following identities:

$$\begin{array}{rclcrcl}
\mathit{LVec}_0 & = & \mathsf{Unit} & \qquad & \mathit{RVec}_0 & = & \mathsf{Unit} \\
\mathit{LVec}_{n+1} & = & \mathsf{Prod}\ \mathit{LVec}_n\ \mathsf{Id} & & \mathit{RVec}_{n+1} & = & \mathsf{Prod}\ \mathsf{Id}\ \mathit{RVec}_n
\end{array}$$

The type constructors LVec_s model left-nested vectors of length s (*i.e.*, $|\mathit{LVec}_s| = s$ for any $s \in \mathbb{N}$), while the type constructors RVec_s model right-nested vectors of length s (*i.e.*, $|\mathit{RVec}_s| = s$ for any $s \in \mathbb{N}$). The sequence of integers from 0 to 2 is encoded as:

$$\begin{aligned}
((((), 2), 1), 0) &: \mathit{LVec}_3\ \mathit{int} \\
(0, (1, (2, ()))) &: \mathit{RVec}_3\ \mathit{int}
\end{aligned}$$

Following our earlier results, we immediately have that, for any $s \in \mathbb{N}$, both LVec_s and RVec_s are functors supporting a prefix sum operation.

Question 9 *In this question, we consider the prefix sum induced by the the left-nested vector LVec_3 and the right-nested vector RVec_3 of 3 elements. Draw the corresponding circuits, taking 3 input wires to $3+1$ output wires.*

□

Question 10 *Express the work —written work(LVec_s), resp. work(RVec_s)— and depth — written depth(LVec_s), resp. depth(RVec_s)— of the prefix sum over LVec_s and, respectively, RVec_s in asymptotic terms over s.*

Which representation yields a more efficient circuit in absolute terms?

Answer:

We have

- $\mathrm{depth}(\mathit{RVec}_s) = n$ (exactly)
- $\mathrm{work}(\mathit{RVec}_s) = O(n^2)$ (asymptotically)
- $\mathrm{depth}(\mathit{LVec}_s) = n$ (exactly)
- $\mathrm{work}(\mathit{LVec}_s) = 2n$ (exactly)

Left-nested vectors are more efficient overall.

□

5 Parametricity

In this section, we postulate the existence of a function

$$take : \forall\, \alpha :: \star.\ int \to List\, \alpha \to List\, \alpha$$

which extracts a prefix of given length from the input list,

$$append : \forall\, \alpha :: \star.\ List\, \alpha \to List\, \alpha \to List\, \alpha$$

which appends two lists, and $belowseq : int \to List\ int$ where $belowseq\, k$ computes the list $[0; \ldots; k-1]$.

For the remainder of the exam, we return to the sequential implementation $prefix\text{-}sum_{List}$ presented in Section 2 and focus solely on its first component. We thus define a shorthand

$$\begin{aligned} prefix\text{-}sum^0_{List} &: \forall\, M :: \star.\ IsMonoid\ M \to List\, M \to List\, M \\ prefix\text{-}sum^0_{List} &\triangleq \lambda\, (\cdot, \epsilon)\ x.\ \mathsf{fst}\ (prefix\text{-}sum_F\ (\cdot, \epsilon)\ x) \end{aligned}$$

An executable specification of $prefix\text{-}sum^0_{List}$ can be given in terms of the foldability of lists (Question 4). The sum of the first k elements is specified as

$$\begin{aligned} fold\text{-}prefix &: \forall\, M :: \star.\ IsMonoid\ M \to List\, M \to int \to M \\ fold\text{-}prefix &\triangleq \lambda\, (\cdot, \epsilon)\ xs\ k.\ fold_{List}\ (\cdot, \epsilon)\ (take\, k\ xs) \end{aligned}$$

and, therefore, for any monoid $(M, \cdot, \epsilon)$ and any list $xs : List M$, a candidate implementation $prefix\text{-}sum^0_{List}$ ought to satisfy the following equation:

$$\begin{aligned} &prefix\text{-}sum^0_{List}\ (\cdot, \epsilon)\ xs \\ &\quad = map_{List}\ (fold\text{-}prefix\ (\cdot, \epsilon)\ xs)\ (belowseq\ (length\, xs)) \end{aligned} \qquad \text{(Prefix0-Spec)}$$

The following result is due to Mary Sheeran:

Lemma 1. A polymorphic definition of $prefix\text{-}sum^0_F$ is correct —*i.e.*, satisfies the executable specification (Prefix0-Spec)— if and only if it produces the correct result for the following instance:

$$prefix\text{-}sum^0_F\ (append, [])\ (map_{List}\ (\lambda\, x.\ [x])\ (belowseq\, n)) \qquad \text{(Prefix0-Int)}$$

Question 11 *Express the expected result of (Prefix0-Int) in terms of n, the operation belowseq and map_{List}.*

Hint: proceed by equational reasoning.

Answer:

We expect this computation to be equivalent to

$$map_{List}\ belowseq\ (belowseq\, n)$$

□

A key step to prove Lemma 1 is the following observation. Given a monoid $(M, \cdot, \epsilon)$, a type τ, a function $h : \tau \to M$ and a list $xs : List\, \tau$, we have

$$\begin{aligned} &prefix\text{-}sum^0_{List}\ (\cdot, \epsilon)\ (map_{List}\, h\ xs) \\ &= map_{List}\ (collect\, h) \\ &\qquad (prefix\text{-}sum^0_{List}\ (append, [])\ (map_{List}\ (\lambda\, x.\ [x])\ xs)) \end{aligned} \qquad \text{(Prefix0-Map)}$$

where *collect* is defined as

$$\begin{aligned} &\mathit{collect} : (\tau \to M) \to \mathit{List}\,\tau \to M \\ &\mathit{collect} \triangleq \lambda\, f.\, \lambda\, xs.\, \mathit{fold}_{\mathit{List}}\,(\cdot, \epsilon)\,(\mathit{map}_{\mathit{List}}\, f\, xs) \end{aligned}$$

The Free Theorem induced by the parametric interpretation of the type of $\mathit{prefix\text{-}sum}^0_{\mathit{List}}$ states that for any pair of monoids $(M, \cdot, \epsilon)$ and $(N, \cdot, \epsilon)$, for any function $f : M \to N$ such that

1. $f\,\epsilon = \epsilon$
2. for all $t, u : M$, $f\,(t \cdot u) = (f\,t) \cdot (f\,u)$

and for any list $xs : \mathit{List}\,M$, we have

$$\mathit{map}_{\mathit{List}}\, f\,(\mathit{prefix\text{-}sum}^0_{\mathit{List}}\,(\cdot, \epsilon)\, xs) = \mathit{prefix\text{-}sum}^0_{\mathit{List}}\,(\cdot, \epsilon)\,(\mathit{map}_{\mathit{List}}\, f\, xs)$$

Question 12 *Recall the parametric interpretation of the type of* $\mathit{prefix\text{-}sum}^0_{\mathit{List}}$ *and show that it implies the Free Theorem.*

□

Question 13 *Prove the statement (*PREFIX0-MAP*) by equational reasoning. In the process, you will need some general properties of* $\mathit{fold}_{\mathit{List}}$, $\mathit{map}_{\mathit{List}}$ *and operations on lists: state them explicitly, without proving them.*

Hint: the free theorem of $\mathit{prefix\text{-}sum}^0_F$ *plays a key role in the proof. Besides and since this statement is instrumental to prove Lemma 1 and, in turn, the statement* PREFIX0-SPEC*, your proof should* ***not*** *depend on these properties.*

□