Space-efficient Computation by Interaction

A Type System for Logarithmic Space

Ulrich Schöpp

Ludwig-Maximilians-Universität München
Oettingenstraße 67, D-80538 München, Germany
schoepp@tcs.ifi.lmu.de

Abstract. We introduce a typed functional programming language for logarithmic space. Its type system is an annotated subsystem of Hofmann's polytime LFPL. To guide the design of the programming language and to enable the proof of LOGSPACE-soundness, we introduce a realisability model over a variant of the Geometry of Interaction. This realisability model, which takes inspiration from Møller-Neergaard and Mairson's work on BC_{ε}^{-}, provides a general framework for modelling space-restricted computation.

1 Introduction

Many important complexity classes can be captured by programming languages and logics [11]. Such implicit characterisations of complexity classes are desirable for many reasons. For instance, one may want to avoid technical details in the construction and manipulation of Turing Machines, or one may want to get insight into which kinds of high-level programming principles are available in certain complexity classes. It has also become apparent that implicit characterisation of complexity classes can help in devising methods for analysing the resource-consumption of programs and in finding compilation strategies that guarantee certain resource bounds [19].

In this paper we address the question of how to design a functional programming language that captures LOGSPACE. Existing functional characterisations of LOGSPACE, such as [18, 17, 4, 13, 14], are all minimal by design. Although minimal languages are well-suited for theoretical study, they tend not to be very convenient for writing programs in. Here we take the first steps towards extending the ideas in loc. cit. to a less minimal programming language for LOGSPACE. One of the main issues in carrying out such an extension lies in the manageability of the computation model. While for the small languages in loc. cit. it is possible to show LOGSPACE-soundness by considering the whole language, this becomes increasingly difficult the more constructs are added to the programming language. Hence, an important goal in the design of a language for LOGSPACE is to capture its compilation in a modular, compositional way.

1.1 Modelling Space-efficient Computation by Interaction

In this paper we give a compositional account of space-efficient computation, which is based on modelling computation by interaction. Our approach is motivated by the LOGSPACE-evaluation of Møller-Neergaard and Mairson's function algebra BC_{ε}^{-} [17]. The evaluation of BC_{ε}^{-} may be thought of as a question/answer dialogue, where a question to a natural number n is a number i, which represents the number of the bit to be computed, and an answer is the value of the bit at position i in the binary representation of n. Importantly, this approach admits a space-efficient implementation of recursion. Suppose $h\colon \mathbb{N} \to \mathbb{N}$ is a function we want to iterate k times. In general this will not be possible in LOGSPACE, as we cannot store intermediate results. However, it can be done if h has the special property that to compute one bit of its output only one bit of its input is needed. Suppose we want to compute bit i of $h^k(x)$. By the special property of h, it is enough to know at most one bit of $h^d(x)$ for each $d \leq k$. Suppose we already know the right bit of $h^d(x)$ for some $d < k$. To compute the bit of $h^{d+1}(x)$, we begin by asking $h^k(x)$ for bit i. This will result in a question for some bit of $h^{k-1}(x)$, then a question for some bit of $h^{k-2}(x)$, and so on until we reach $h^{d+1}(x)$. Finally, $h^{d+1}(x)$ will ask for one bit of $h^d(x)$, the value of which we already know. By the special property of h, we can thus compute the required bit of $h^{d+1}(x)$. By iteration of this process, the bits of $h^k(x)$ can be computed, assuming we can compute the bits of x. Moreover, this process of computing the bits of $h^k(x)$ can be implemented by storing only one bit of some $h^d(x)$, its recursion depth d, the initial question i and the current question. Under suitable assumptions, such as that k is polynomial in the size of the inputs, such an implementation will be in LOGSPACE.

In this paper we introduce a model for LOGSPACE-computation in which such an implementation of recursion is available. Our approach is based on modelling computation by question/answer-interaction. Such models have been studied extensively in the context of game semantics. We draw on this work, but since we are interested not just in modelling dialogues but also in effectively computing answers to questions, we are lead to considering the particular case of a Geometry of Interaction (GoI) situation, see [2, 8] for a general description and [3] for a the connection to game semantics. In Sect. 3 we build a model on the basis of a GoI situation. By building on this well-studied structure, we can hope to benefit from existing work, such as that on the connections to abstract machines [6] or to machine code generation [16], although we have yet to explore the connections.

As an example of the kind of programming language that can be derived from the model, we introduce LogFPL, a simple functional language for LOGSPACE.

2 A Type System for Logarithmic Space

The type system for LogFPL is an annotated subsystem of LFPL [9]. In LogFPL all variables are annotated with elements of $Z{:=}\{1,\cdot,\infty\} \times \mathbb{N}$. The intended meaning of the annotations is that we may send arbitrarily many questions to each variable marked with ∞, that we may send at most one question to each variable marked with $\cdot$, and that we may send at most one question to *all* the variables marked with 1. The second component of the annotations specifies how many memory locations we may use when asking a question. We define an ordering on Z by letting $1 < \cdot < \infty$ and $\langle z,i\rangle \leq \langle z',i'\rangle \iff (z \leq z') \wedge (i \leq i')$. We define an addition on Z by $\langle m,i\rangle + \langle m',i'\rangle = \langle \max(m,m'), i+i'\rangle$.

A *context* Γ is a finite set of variable declarations $x \overset{z}{:} A$, where $z \in Z$, subject to the usual convention that each variable is declared at most once. For $z \in Z$, we write $!^z\Gamma$ for the context obtained from Γ by replacing each declaration $x \overset{u}{:} A$ with $x \overset{u+z}{:} A$. We write short $!\Gamma$ for $!^{\langle 1,1\rangle}\Gamma$. We write $\Gamma \geq z$ if $u \geq z$ holds for all $x \overset{u}{:} A$ in Γ.

Small types	$A_S ::= I \mid \Diamond \mid \mathbf{B} \mid \mathbf{SN} \mid A_S * A_S$
Types	$A ::= A_S \mid \mathbf{L}(A_S) \mid A \times A \mid A \otimes_1 A \mid A \overset{\cdot,i}{\multimap} A \mid A \overset{\infty,i}{\multimap} A$
Terms	$M ::= x \mid c \mid * \mid \lambda x.M \mid M\,M \mid M \otimes_1 M \mid \text{let } M \text{ be } x \otimes_1 x \text{ in } M$
	$\mid M * M \mid \text{let } M \text{ be } x * x \text{ in } M \mid \langle M, M \rangle \mid \pi_1(M) \mid \pi_2(M)$

The small types are the unit type I, the resource type $\Diamond$, as known from LFPL [9], the type of booleans $\mathbf{B}$, the type of small numbers $\mathbf{SN}$, and the type $A * B$ of pairs of small types. Small types have the property that their elements can be stored in memory. The type $\mathbf{SN}$, for example, contains natural numbers that are no larger than the size of the input, as measured by the number of $\Diamond$s. Using binary encoding, such numbers can be stored in memory. Small types are such that a single question suffices to get the whole value of an element. In contrast, one question to a list, for example, is a question to one of its elements. Thus, $\mathbf{SN}$ differs from the type $\mathbf{L}(I)$ of lists with unit-type elements.

The types are built starting from the small types and from lists $\mathbf{L}(A)$ over small types. The function space $\multimap$ has an annotation that indicates how many questions a function needs to ask of its argument in order to answer a query to its result. The second component of the annotations specifies how much data a function needs to store along with a question. We often write $\multimap$ for $\overset{\cdot,i}{\multimap}$ and $\overset{\infty}{\multimap}$ for $\overset{\infty,i}{\multimap}$ when i is not important. The type $A \otimes_1 B$ consists of pairs $x \otimes_1 y$, which we may use by asking one question *either* of x or of y. For instance, the type of the constant *cons* below expresses that one question to the list $cons(d,a,r)$ can be answered by asking one question of either d, a or r.

We have the following constants for booleans, lists and small numbers:

Booleans	$\mathtt{tt}, \mathtt{ff} \colon \mathbf{B}$	$case_{\mathbf{B}} \colon \mathbf{B} \overset{\infty,k(A)}{\multimap} (A \times A) \overset{\cdot,0}{\multimap} A$
Lists	$nil \colon \mathbf{L}(A)$	$hdtl \colon \mathbf{L}(A) \overset{\cdot,1}{\multimap} \Diamond \otimes_1 A \otimes_1 \mathbf{L}(A)$
	$cons \colon \Diamond \otimes_1 A \otimes_1 \mathbf{L}(A) \overset{\cdot,1}{\multimap} \mathbf{L}(A)$	$empty \colon \mathbf{L}(A) \overset{\cdot,0}{\multimap} \mathbf{B}$
Small numbers	$zero \colon \mathbf{SN}$	$succ \colon \Diamond \overset{\cdot,0}{\multimap} \mathbf{SN} \overset{\cdot,0}{\multimap} \mathbf{SN}$
	$case_{\mathbf{SN}} \colon \mathbf{SN} \overset{\infty,1+k(A)}{\multimap} (A \times (\Diamond \overset{\cdot,i}{\multimap} \mathbf{SN} \overset{\cdot,i}{\multimap} A)) \overset{\cdot,1}{\multimap} A$	

The number $k(A)$ is defined in Fig 2. We note that *hdtl* represents a partial function undefined for *nil*. Furthermore, we have a constant $discard \colon A \overset{\cdot,0}{\multimap} I$ for each type A built without $\multimap$, and a constant $dup_i \colon (A \overset{\cdot,i}{\multimap} \mathbf{B}) \overset{\cdot,k(A)}{\multimap} A \overset{\cdot,0}{\multimap} (\mathbf{B} * A)$ for each small type A.

The typing rules appear in Figs. 1–3. The rules for small types are based on the fact that for small types we can get the whole value of an element with a single question. Rule (∞-$\cdot$), for instance, expresses that instead of asking a small value many times, we may ask just one question, store the result and answer the many questions from memory.

2.1 Soundness and Completeness

Writing $\|M\|$ for the evident functional interpretation of a term M, we have:

Proposition 1 (Soundness). *For any judgement $\vdash M \colon \mathbf{L}(I) \overset{\infty,i}{\multimap} \mathbf{L}(\mathbf{B}) \overset{\infty,i}{\multimap} \mathbf{L}(\mathbf{B})$, there is a* LOGSPACE*-algorithm e, such that, for all $x \in \mathbf{L}(I)$ and all $y \in \mathbf{L}(\mathbf{B})$, if $\|M\|(x,y)$ is defined then e applied to inputs x and y returns $\|M\|(x,y)$.*

We construct the algorithm e in Sect. 3 by construction of a model.

$$\frac{A' \leq A \qquad B \leq B' \qquad u \leq v}{(A \overset{u}{\multimap} B) \leq (A' \overset{v}{\multimap} B')} \qquad \frac{A \leq A' \qquad B \leq B'}{(A \otimes_1 B) \leq (A' \otimes_1 B')}$$

Fig. 1. Subtyping relation

$$(\textsc{Axiom})\ \frac{}{x \overset{z}{:} A \vdash x\colon A} \qquad (\textsc{Const})\ \frac{c\colon A}{\vdash c\colon A} \qquad (\textsc{Sub})\ \frac{\Gamma \vdash M\colon A \qquad A \leq B}{\Gamma \vdash M\colon B}$$

$$(I\text{-I})\ \frac{}{\vdash *\colon I} \qquad (I\text{-E})\ \frac{\Gamma \vdash M\colon I \qquad \Delta \vdash N\colon A}{\Gamma, \Delta \vdash \text{let } M \text{ be } * \text{ in } N\colon A}$$

$$(\multimap\text{-I})\ \frac{\Gamma, x \overset{z}{:} A \vdash M\colon B}{\Gamma \vdash \lambda x. M\colon A \overset{z}{\multimap} B}\ z \geq \langle \cdot, 0\rangle \qquad (\multimap\text{-E})\ \frac{\Gamma \vdash M\colon A \overset{z}{\multimap} B \qquad \Delta \vdash N\colon A}{\Gamma, !^{z}\Delta \vdash M\,N\colon B}$$

$$(\otimes_1\text{-I})\ \frac{\Gamma \vdash M\colon A \qquad \Delta \vdash N\colon B}{\Gamma, \Delta \vdash M \otimes_1 N\colon A \otimes_1 B} \qquad (\otimes_1\text{-E})\ \frac{\Gamma \vdash M\colon A \otimes_1 B \qquad \Delta, x \overset{\langle 1,i\rangle}{:} A, y \overset{\langle 1,i\rangle}{:} B \vdash N\colon C}{\Delta, !^{\langle 1,i\rangle}\Gamma \vdash \text{let } M \text{ be } x \otimes_1 y \text{ in } N\colon C}$$

$$(\times\text{-I})\ \frac{\Gamma \vdash M\colon A \qquad \Gamma \vdash N\colon B}{!^{\langle\infty,1\rangle}\Gamma \vdash \langle M, N\rangle\colon A \times B} \qquad (\times\text{-E1})\ \frac{\Gamma \vdash M\colon A \times B}{\Gamma \vdash \pi_1(M)\colon A} \qquad (\times\text{-E2})\ \frac{\Gamma \vdash M\colon A \times B}{\Gamma \vdash \pi_2(M)\colon B}$$

Recursion types $R ::= A_S \mid \mathbf{L}(\mathbf{B}) \mid R \otimes_1 R$

$$(\textsc{Rec-}\mathbf{L})\ \frac{\Gamma \vdash g\colon B \qquad \vdash f\colon (B \overset{\cdot,j}{\multimap} \mathbf{B}^k) \overset{\infty,i}{\multimap} \Diamond \overset{\cdot,i}{\multimap} A \overset{\infty,i}{\multimap} B \overset{\cdot,i}{\multimap} B \qquad \Delta \vdash d\colon B \overset{\cdot,j}{\multimap} \mathbf{B}^k}{\Gamma, !^{\langle\infty,s(i,B)\rangle}\Delta \vdash rec_{\mathbf{L}(A)}\ g\ f\ d\colon \mathbf{L}(A) \xrightarrow{\infty,s(i,B)}\!\!\circ\ B}\ B \in R$$

$$(\textsc{Rec-}\mathbf{SN})\ \frac{\Gamma \vdash g\colon B \qquad \vdash f\colon (B \overset{\cdot,j}{\multimap} \mathbf{B}^k) \overset{\infty,i}{\multimap} \Diamond \overset{\cdot,i}{\multimap} B \overset{\cdot,i}{\multimap} B \qquad \Delta \vdash d\colon B \overset{\cdot,j}{\multimap} \mathbf{B}^k}{\Gamma, !^{\langle\infty,s(i,B)\rangle}\Delta \vdash rec_{\mathbf{SN}}\ g\ f\ d\colon \mathbf{SN} \xrightarrow{\cdot,s(i,B)}\!\!\circ\ B}\ B \in R$$

Here, $s(i,A) = i + 4 + 7 \cdot k(A)$, where $k(A) = 1$ for $A \in \{I, \Diamond, \mathbf{B}, \mathbf{SN}\}$, $k(\mathbf{L}(A)) = 2 + k(A)$ and $k(A * B) = k(A \times B) = k(A \otimes_1 B) = k(A \overset{z}{\multimap} B) = 1 + k(A) + k(B)$. The definition of $k(A)$ is such that any type A is $k(A)$-encodable, see Def. 6.

Fig. 2. General typing rules

In the following rules, A, B and C must be small types.

$$(*\text{-I})\ \frac{\Gamma \vdash M\colon A \qquad \Delta \vdash N\colon B}{!^{k(B)}\Gamma, !^{k(A)}\Delta \vdash M{*}N\colon A{*}B}\ \Gamma \geq \langle \cdot, 0\rangle \vee \Delta \geq \langle \cdot, 0\rangle$$

$$(*\text{-E})\ \frac{\Gamma \vdash M\colon A * B \qquad \Delta, x \overset{\langle \cdot,i\rangle}{:} A, y \overset{\langle \cdot,i\rangle}{:} B \vdash N\colon C}{!^{k(A*B)}\Delta, !^{\langle \cdot, i+k(C)\rangle}\Gamma \vdash \text{let } M \text{ be } x{*}y \text{ in } N\colon C}$$

$$(\infty\text{-}\cdot)\ \frac{\Delta, x \overset{\langle\infty,i\rangle}{:} A \vdash M\colon B}{!^{k(A)}\Delta, x \overset{\langle \cdot, i+k(B)\rangle}{:} A \vdash M\colon B}$$

Fig. 3. Special rules for small types

First we sketch how LOGSPACE-predicates can be represented in LogFPL. Assume a LOGSPACE Turing Machine over a binary alphabet. By a suitable encoding of the input, we can assume that it never moves its input head beyond the end of the input.

We encode the computation of the Turing Machine. Its input tape is given as a binary string, i.e. as a list in $\mathbf{L}(\mathbf{B})$. We represent the position of the input head by a value in $\mathbf{SN}$. Access to the input tape is given by a zipper-like function *focus* of type $\mathbf{SN} \multimap \mathbf{L}(\mathbf{B}) \multimap \mathbf{SN} \otimes_1 \mathbf{L}(\mathbf{B}) \otimes_1 \mathbf{L}(\mathbf{B})$, with the following meaning:

$$\mathit{focus}\ i\ \langle x_0,\dots,x_k\rangle = i \otimes_1 \langle x_{i-1},\dots,x_0\rangle \otimes_1 \langle x_i,\dots,x_k\rangle \qquad 0 \le i \le k$$

The function *focus* is defined as $\lambda l.\,(\mathit{rec}_{\mathbf{SN}}\ \mathit{base}\ \mathit{step})$ (omitting d for brevity) with:

$$l\colon \mathbf{L}(\mathbf{B}) \vdash \mathit{base}{:=}\mathit{zero}\otimes_1 \mathit{nil} \otimes_1 l\colon \mathbf{SN} \otimes_1 \mathbf{L}(\mathbf{B}) \otimes_1 \mathbf{L}(\mathbf{B})$$

$$\begin{array}{ll} & \lambda d.\,\lambda r.\,\text{let } r \text{ be } n \otimes_1 h \otimes_1 t \text{ in} \\ \vdash \mathit{step}{:=} & \qquad \text{let } (\mathit{hdtl}\ t) \text{ be } d' \otimes_1 a \otimes_1 t' \text{ in } (\mathit{succ}\ d\ n) \otimes_1 (\mathit{cons}\ d'\ a\ h) \otimes_1 t' \\ & \qquad\quad : \Diamond \multimap \mathbf{SN} \otimes_1 \mathbf{L}(\mathbf{B}) \otimes_1 \mathbf{L}(\mathbf{B}) \multimap \mathbf{SN} \otimes_1 \mathbf{L}(\mathbf{B}) \otimes_1 \mathbf{L}(\mathbf{B}) \end{array}$$

To represent the work tape we use small numbers $\mathbf{SN}$, since the work tape has only logarithmic size and since $\mathbf{SN}$ is much more flexible than lists. To encode the transitions of the TM, we use a number of helper functions. We encode 'bounded small numbers' by pairs in $\mathbf{SN}_B{:=}\mathbf{SN}{*}\mathbf{SN}$, where the first component represents the value of the number and the second component contains memory (in the form of $\Diamond$) that may be used for increasing the number. For instance, the incrementation function maps $m{*}\mathit{zero}\colon \mathbf{SN}_B$ to $m{*}\mathit{zero}$ and $m{*}\mathit{succ}(d,n)$ to $\mathit{succ}(d,m){*}n$. Using the rules for small types, we can represent the evident functions $\mathit{null}\colon \mathbf{SN} \multimap \mathbf{SN}_B$, $\mathit{inc}\colon \mathbf{SN}_B \multimap \mathbf{SN}_B$, $\mathit{dec}\colon \mathbf{SN}_B \multimap \mathbf{SN}_B$, $\mathit{double}\colon \mathbf{SN}_B \multimap \mathbf{SN}_B$, $\mathit{half}\colon \mathbf{SN}_B \multimap \mathbf{SN}_B$ and $\mathit{even}\colon \mathbf{SN}_B \multimap \mathbf{B}{*}\mathbf{SN}_B$.

We encode the state of a Turing Machine by a 4-tuple $l{*}r{*}i{*}s$ in $\mathbf{SN}_B{*}\mathbf{SN}_B{*}\mathbf{SN}_B{*}\mathbf{B}^k$, where l and r represent the parts of the work tape left and right from the work head, i represents the position of the input head and s represents the state of the machine. We abbreviate $\mathbf{SN}_B{*}\mathbf{SN}_B{*}\mathbf{SN}_B{*}\mathbf{B}^k$ by S. It should be clear how to use the above helper functions for implementing the basic operations of a Turing Machine. For instance, moving the head on the work tape to the right is given by mapping a tuple $l{*}r{*}i{*}s$ to the value (let $\mathit{even}(r)$ be $e{*}r$ in $(\mathit{case}_{\mathbf{B}}\ e\ \langle \mathit{double}(l){*}\mathit{half}(r){*}i{*}s, \mathit{inc}(\mathit{double}(l)){*}\mathit{half}(r){*}i{*}s\rangle)$). If, in addition to a 4-tuple in S, we have a function $d\colon \mathbf{SN} \multimap \mathbf{B}$, such that $d(n)$ is the n-th character of the input tape, then, using *dup*, we can thus implement the transition function of the Turing Machine. Hence, we can give the transition function the type $h\colon (\mathbf{SN} \multimap \mathbf{B}) \overset{\infty}{\multimap} S \multimap S$. Let $g\colon S$ be the initial state of the machine. Since, using *focus*, we can define an access function $t\colon \mathbf{L}(\mathbf{B}) \vdash d\colon \mathbf{SN} \multimap \mathbf{B}$, we can model the computation of the machine by $g\colon S, t \overset{\infty}{:} \mathbf{L}(\mathbf{B}) \vdash \mathit{rec}_{\mathbf{SN}}\ g\ h\ d\colon \mathbf{SN} \multimap S$. Now, in order to construct the initial state g as well as an upper bound on the computation length, we need a polynomial number of $\Diamond$s. We obtain these from a given list $\mathbf{L}(I)$, which we split in four numbers in $\mathbf{SN}$ of equal size. Thus, we obtain a term $s \overset{\infty}{:} \mathbf{L}(I), t \overset{\infty}{:} \mathbf{L}(\mathbf{B}) \vdash m\colon S$, which, if s is large enough, computes the final state of the Turing Machine for input t.

Proposition 2 (Completeness). *Each* LOGSPACE*-predicate $A \subseteq \mathbf{L}(\mathbf{B})$ can be represented by a term $M\colon \mathbf{L}(I) \overset{\infty,i}{\multimap} \mathbf{L}(\mathbf{B}) \overset{\infty,i}{\multimap} \mathbf{B}$ in the following sense. There exist natural numbers n and m, such that, for each $a \in \mathbf{L}(\mathbf{B})$ and each list $s \in \mathbf{L}(I)$ that is longer than $|a|^n + m$, we have $\|M\|(a,s) = \mathtt{tt}$ if $a \in A$ and $\|M\|(a,s) = \mathtt{ff}$ if $a \notin A$.*

3 Modelling Space-efficient Computation by Interaction

We compile LogFPL to LOGSPACE-algorithms by interpreting it in an instance of the Geometry of Interaction situation [2]. We use an instance of the GoI situation in which questions can be answered in linear space. This is motivated by the fact that questions will typically be of logarithmic size (think of the bit-addresses of an input number), so that to remain in LOGSPACE we can allow linear space in the size of the questions.

3.1 Linear non-size-increasing functions

The underlying computational model is that of non-size-increasing linear space functions. The restriction to non-size-increasing functions is needed for composition in $\mathcal{G}$, defined below, to remain in linear space. To fix the computation model, we work with multi-tape Turing Machines over some alphabet Σ. The machines take their input on one designated tape, where they also write the output.

The objects of $\mathcal{L}$ are triples (X, c, l), where X is a underlying set, $c\colon X \to \Sigma^*$ is a coding function and $l\colon X \to \mathbb{N}$ is an abstract length measure. An object must be so that there exist constants $m, n \in \mathbb{N}$ such that $\forall x \in X.\, m \cdot l(x) + n \leq |c(x)|$ holds, i.e. the abstract length measure underestimates the actual size at most linearly. The morphisms from (X, c_X, l_X) to (Y, c_Y, l_Y) are the partial functions $f\colon X \to Y$ with the property $\forall x \in X.\, f(x)\downarrow \implies l_Y(f(x)) \leq l_X(x)$, for which in addition there exists a linear space algorithm e satisfying $\forall x \in X.\, f(x)\downarrow \implies e(c_X(x)) = c_Y(f(x))$.

The category $\mathcal{L}$ supports a number of data type constructions, much in the style of LFPL [9]. We use the following constructions, of which we spell out only the underlying sets and the length functions. The disjoint union $A + B$ of the underlying sets of A and B becomes an object with $l_{A+B}(inl(a)) = l_A(a)$ and $l_{A+B}(inr(b)) = l_B(b)$. The object $A \otimes B$ has as underlying set the set of pairs of elements of A and B with length function $l_{A\otimes B}(\langle a, b\rangle) = l_A(a) + l_B(b)$. The evident projections $\pi_1\colon A \otimes B \to A$ and $\pi_2\colon A \otimes B \to B$ are morphisms of $\mathcal{L}$. We also have booleans $\mathbf{B} = \{\mathtt{ff}, \mathtt{tt}\}$ with $l_{\mathbf{B}}(\mathtt{ff}) = l_{\mathbf{B}}(\mathtt{tt}) = 0$, lists $\mathbf{L}(A) = A^*$ with $l_{\mathbf{L}(A)}(\langle a_1, \ldots, a_n\rangle) = \sum_{i=1}^{n} 1 + l_A(a_i)$, and trees $\mathbf{T}(A) = A + \mathbf{T}(A) \otimes \mathbf{T}(A)$ with $l_{\mathbf{T}(A)}(inl(a)) = l_A(a)$ and $l_{\mathbf{T}(A)}(inr(a, a')) = 1 + l_{\mathbf{T}(A)}(a) + l_{\mathbf{T}(A)}(a')$. An abstract resource type $\Diamond$ is given by $\Diamond = \{\Diamond\}$ with $l(\Diamond) = 1$. It differs from the unit object $1 = \{*\}$ with $l(*) = 0$ only in its length measure.

Given $f\colon A + B \to C + B$, we define its trace $tr(f)\colon A \to C$ by $tr(f) = t \circ inl$, where $t\colon A + B \to C$ is defined by

$$t(x) = \begin{cases} c & \text{if } f(x) = inl(c), \\ t(inr(b)) & \text{if } f(x) = inr(b). \end{cases}$$

Making essential use of the fact that all morphisms in $\mathcal{L}$ are non-size-increasing and that the length measure underestimates the real size at most linearly, it follows that $tr(f)$ is again a morphism in $\mathcal{L}$. That the definition of $tr(f)$ satisfies the equations for trace, see e.g. [8, Def. 2.1.16], can be seen by observing that the forgetful functor from $\mathcal{L}$ to the traced monoidal category **Pfn** of sets and partial functions, see e.g. [8, Sec. 8.2], is faithful and preserves both finite coproducts and trace.

3.2 Linear-space interaction

Computation by interaction is modelled by a GoI situation over $\mathcal{L}$. For space reasons, we can only give the basic definitions. We refer to [2, 8] for more details and discussion.

The category $\mathcal{G}$ has as objects pairs (A^+,A^-) of two objects of $\mathcal{L}$. Here, A^- is thought of as a set of questions and A^+ as a set of answers. The morphisms in $\mathcal{G}$ from (A^+,A^-) to (B^+,B^-) are the $\mathcal{L}$-morphisms of type $A^+ + B^- \to A^- + B^+$. We will often use these two views of morphisms interchangeably. We write $\mathcal{G}(A,B)$ for the set of morphisms in $\mathcal{G}$ from A to B. The identity on A is given by $[inr, inl]\colon A^+ + A^- \to A^- + A^+$. Composition $g \cdot f$ of two morphisms $f\colon A \to B$ and $g\colon B \to C$ is given by the trace of $(A^- + g) \circ (f + C^-)\colon A^+ + B^- + C^- \to A^- + B^- + C^+$ with respect to B^-.

Of the structure of $\mathcal{G}$, we spell out the symmetric monoidal structure $\otimes$, given on objects by $A \otimes B = (A^+ + B^+, A^- + B^-)$ and on morphisms by using $+$ on the underlying $\mathcal{L}$-morphisms. A unit for $\otimes$ is $I = (\emptyset, \emptyset)$. Note $I \otimes I = I$. Note also that morphisms of type $I \to A$ in $\mathcal{G}$ are the same as $\mathcal{L}$-maps of type $A^- \to A^+$ and that morphisms of type $A \to I$ are the same as $\mathcal{L}$-maps of type $A^+ \to A^-$.

A monoidal closed structure is given on objects by $(A \multimap B) = (B^+ + A^-, B^- + A^+)$. We write $\varepsilon\colon (A \multimap B) \otimes A \to B$ for the application map and $\Lambda f\colon A \to (B \multimap C)$ for the abstraction of $f\colon A \otimes B \to C$.

We use a storage functor $!(-)$. On objects it is defined by $!A = (A^+ \otimes \mathbf{S}, A^- \otimes \mathbf{S})$, where $\mathbf{S} = \mathbf{B} \otimes \mathbf{T}(\mathbf{L}(\mathbf{B}))$. The intention is that a store in $\mathbf{S}$ is passed along with questions and answers. The underlying $\mathcal{L}$-map of the morphism $!f$ is the $\mathcal{L}$-map $f \otimes \mathbf{S}$.

3.3 Realisability and co-Realisability

As a way of formalising the compilation of functions into interaction-programs, we define a category $\mathcal{R}$. Its definition is parameterised over a commutative monoid $(\mathcal{M}, +, 0)$, equipped with a pre-order $\leq$ that is compatible with $+$.

Objects. The objects are tuples $(|A|, A^*, A_{\mathcal{G}}, \Vdash, \Vdash^*)$ consisting of a set $|A|$ of underlying elements, a set A^* of underlying co-elements, an object $A_{\mathcal{G}}$ of $\mathcal{G}$, a realisation relation $\Vdash \subseteq \mathcal{M} \times \mathcal{G}(I, A_{\mathcal{G}}) \times |A|$ and a co-realisation relation $\Vdash^* \subseteq \mathcal{M} \times \mathcal{G}(A_{\mathcal{G}}, I) \times A^*$. The objects are required to have the following properties.

1. For all $a \in |A|$ there exist α and e such that $\alpha, e \Vdash a$ holds. Dually, for all $a^* \in A^*$ there exist α and c such that $\alpha, c \Vdash^* a^*$ holds.
2. If $\alpha, e \Vdash a$ and $\alpha \leq \beta$ hold then so does $\beta, e \Vdash a$. Dually, if $\alpha, e \Vdash^* a^*$ and $\alpha \leq \beta$ hold then so does $\beta, e \Vdash^* a^*$.

Morphisms. A morphism from A to B consists of two functions $f\colon |A| \to |B|$ and $f^*\colon B^* \to A^*$ for which there exists a map $r\colon A_{\mathcal{G}} \to B_{\mathcal{G}}$ in $\mathcal{G}$ satisfying:

$$\forall \alpha \in \mathcal{M}, e\colon I \to A_{\mathcal{G}}, x \in A. \quad \alpha, e \Vdash_A x \implies \alpha, r \cdot e \Vdash_B f(x),$$
$$\forall \beta \in \mathcal{M}, c\colon B_{\mathcal{G}} \to I, k \in B^*. \quad \beta, c \Vdash^*_B k \implies \beta, c \cdot r \Vdash^*_A f^*(k).$$

We also need the following more general form of morphism realisation.

Definition 1. *A morphism* $r\colon A_{\mathcal{G}} \to B_{\mathcal{G}}$ realises (f, f^*) with bound $\varphi \in \mathcal{M}$ *if*

$$\forall \alpha \in \mathcal{M}, e\colon I \to A_{\mathcal{G}}, x \in A. \quad \alpha, e \Vdash_A x \implies \varphi + \alpha, r \cdot e \Vdash_B f(x),$$
$$\forall \beta \in \mathcal{M}, c\colon B_{\mathcal{G}} \to I, k \in B^*. \quad \beta, c \Vdash^*_B k \implies \varphi + \beta, c \cdot r \Vdash^*_A f^*(k).$$

While our definition of realisability is close to well-known instances of realisability, such as e.g. [5, 15], the definition of co-realisability deserves comment. The main purpose for introducing it is for modelling recursion in the way described in the introduction. To implement recursion in this way, we need to control how often a realiser $r\colon A_{\mathcal{G}} \to B_{\mathcal{G}}$ of the recursion step-function sends a question to its argument. Co-Realisability is a way of obtaining control over how r uses $A_{\mathcal{G}}$. Suppose, for example, that both A and B have only a single co-element that is co-realised by the empty function $\emptyset$. If r realises a morphism $A \to B$ then $\emptyset \cdot r$ must also be the empty function. Hence, whenever r receives an answer in $A^+_{\mathcal{G}}$, it cannot ask another question in $A^-_{\mathcal{G}}$, since otherwise $\emptyset \cdot r$ would not be empty. Thus, for any $e\colon I \to A_{\mathcal{G}}$ and any $q \in B^-_{\mathcal{G}}$, in the course of the computation of $(r \cdot e)(q)$, only one query can be sent by r to e. This is the main example of how we control the behaviour of realisers with co-realisability. More generally, co-realisability formalises how an object A may be *used*. A co-realiser $c\colon A^+_{\mathcal{G}} \to A^-_{\mathcal{G}}$ explains which question we may ask after we have received an answer from an object.

A symmetric monoidal structure $\otimes$ on $\mathcal{R}$ is defined by letting the underlying set of $|A \otimes B|$ be the set of pairs in $|A| \times |B|$. The realising object $(A \otimes B)_{\mathcal{G}}$ is $A_{\mathcal{G}} \otimes B_{\mathcal{G}}$, and the realisation relation is the least relation satisfying

$$(\alpha, e_x \Vdash_A x) \wedge (\beta, e_y \Vdash_B y) \implies \alpha + \beta, e_x \otimes e_y \Vdash_{A \otimes B} \langle x, y\rangle$$

in addition to the general requirements for objects of $\mathcal{R}$ (which from now on we will tacitly assume). The set of co-elements and the co-realisation is defined as follows.

$$\begin{aligned}
(A \otimes B)^* &= \{\langle p\colon |A| \to B^*, q\colon |B| \to A^*\rangle \mid \exists c, \gamma.\, (\gamma, c \Vdash^*_{A\otimes B} \langle p, q\rangle)\} \\
\gamma, c \Vdash^*_{A\otimes B} \langle p, q\rangle &\iff (\forall \alpha, e, a.\, (\alpha, e \Vdash_A a) \implies (\gamma + \alpha), c \cdot (e \otimes id) \Vdash^*_B p(a)) \\
&\qquad \wedge (\forall \beta, e, b.\, (\beta, e \Vdash_B b) \implies (\gamma + \beta), c \cdot (id \otimes e) \Vdash^*_A q(b))
\end{aligned}$$

A unit object for $\otimes$ may be defined by $|I| = \{*\}$, $I^* = \{*\}$, $I_{\mathcal{G}} = (1, 1)$, by letting $\alpha, e \Vdash_I *$ hold if and only if e is the unique total function of its type, and by letting $\alpha, c \Vdash^*_I *$ hold if and only if $c\colon 1 \to 1$ is the empty function.

A monoidal exponent $\multimap$ for $\otimes$ is defined by letting $|A \multimap B|$ be the set of pairs $\langle f, f^*\rangle$ in $(|A| \to |B|) \times (B^* \to A^*)$ that are realised by some r with some bound φ, by letting $(A \multimap B)^* = |A| \times B^*$ and $(A \multimap B)_{\mathcal{G}} = A_{\mathcal{G}} \multimap B_{\mathcal{G}}$, and by letting the relations $\Vdash_{A\multimap B}$ and $\Vdash^*_{A\multimap B}$ be the least relations satisfying

$$\begin{gathered}
\varphi, \Lambda r \Vdash_{A\multimap B} f \iff r\colon A_{\mathcal{G}} \to B_{\mathcal{G}} \text{ realises } f \text{ with bound } \varphi, \\
(\alpha, e \Vdash_A a) \wedge (\beta, c \Vdash^*_B b^*) \implies \alpha + \beta, c \cdot \varepsilon \cdot (id \otimes e) \Vdash^*_{A\multimap B} \langle a, b^*\rangle.
\end{gathered}$$

The co-realisation of $\otimes$ is such that both components of a pair $A \otimes B$ can be used sequentially. This is not the only possible choice. Another useful choice is captured by the monoidal structure $\otimes_1$, where $|A \otimes_1 B| = |A \otimes B|$, $(A \otimes_1 B)_{\mathcal{G}} = (A \otimes B)_{\mathcal{G}}$, $\Vdash_{A\otimes_1 B} = \Vdash_{A\otimes B}$, $(A \otimes_1 B)^* = A^* \times B^*$, and where $\Vdash^*_{A\otimes_1 B}$ is the smallest relation satisfying

$$(\alpha, c \Vdash^*_A a^*) \wedge (\beta, c' \Vdash^*_B b^*) \implies (\alpha + \beta, c \otimes c' \Vdash^*_{A\otimes_1 B} \langle a^*, b^*\rangle).$$

It is instructive to consider the case of morphisms $f\colon A \otimes_1 B \to C$, where A and B have only one co-element that is co-realised by the empty function. In this case, the realiser r

of f may ask *either* one question from A *or* one from B. This is in contrast to $\otimes$, where it would be legal to ask one question from A and then another from B.

Lemma 1. *There exists a natural map* $d\colon A\otimes(B\otimes_1 C)\to B\otimes_1(A\otimes C)$ *with* $d(a,\langle b,c\rangle)=\langle b,\langle a,c\rangle\rangle$ *and* $d^*(b^*,\langle p,q\rangle)=\langle \lambda a.\,\langle b^*,p(a)\rangle,\lambda\langle b,c\rangle.\,q(c)\rangle$.

Lemma 2. *There exists a natural map* $i\colon A\otimes B\to A\otimes_1 B$ *with* $i(a,b)=\langle a,b\rangle$ *and* $i^*(a^*,b^*)=\langle \lambda a.\,b^*,\lambda b.\,a^*\rangle$.

A further monoidal structure $\times$ is defined by $|A\times B|=|A|\times|B|$, $(A\times B)_{\mathcal{G}}=A_{\mathcal{G}}\otimes B_{\mathcal{G}}$, $(A\times B)^*=A^*+B^*$, where $\Vdash_{A\times B}$ and $\Vdash^*_{A\times B}$ are the smallest relations satisfying

$$\begin{aligned}
(\alpha,e_a\Vdash_A a)\wedge(\alpha,e_b\Vdash_B b) &\implies \alpha,e_a\otimes e_b\Vdash_{A\times B}\langle a,b\rangle,\\
(\alpha,c_a\Vdash^*_A a^*) &\implies (\alpha,c_a\circ\pi_1\Vdash^*_{A\times B} inl(a^*)),\\
(\beta,c_b\Vdash^*_B b^*) &\implies (\beta,c_b\circ\pi_2\Vdash^*_{A\times B} inr(b^*)).
\end{aligned}$$

The object $A\times B$ is *not* quite a cartesian product, as is witnessed by rule ($\times$-I), which has the context $!^{\langle\infty,1\rangle}\Gamma$ rather than Γ in its conclusion.

For the implementation of recursion, which requires that there is at most one query to the recursion argument, the following smash-product $*$ is useful. It is defined by $|A*B|=|A|\times|B|$, $(A*B)_{\mathcal{G}}=(A^+_{\mathcal{G}}\otimes B^+_{\mathcal{G}},A^-_{\mathcal{G}}\otimes B^-_{\mathcal{G}})$, $(A*B)^*=A^*\times B^*$, $\Vdash^*_{A*B}=\Vdash^*_{A\otimes_1 B}$,

$$\gamma,e\Vdash_{A*B}\langle x,y\rangle \iff \exists\alpha,\beta,e_x,e_y.\ \begin{array}{l}(\alpha+\beta\le\gamma)\wedge(\alpha,e_x\Vdash_A x)\wedge(\beta,e_y\Vdash_B y)\\ \wedge\forall q_A,q_B.\,(e(q_A,q_B)=\langle e_x(q_A),e_y(q_B)\rangle).\end{array}$$

We use the smash-product for example for booleans, where the full information of $\mathbf{B}*\mathbf{B}$ can be obtained with a single question. Notice that with $\mathbf{B}\otimes\mathbf{B}$ we would need two questions, while with $\mathbf{B}\otimes_1\mathbf{B}$ we could only ask for one component.

Often it is useful to remove the co-realisation-information. This can be done with the co-monad $\Box(-)$ defined by $|\Box A|=|A|$, $\Box A^*=\mathcal{G}(A_{\mathcal{G}},I)$, $(\Box A)_{\mathcal{G}}=A_{\mathcal{G}}$, $\Vdash_{\Box A}=\Vdash_A$ and $(\alpha,c\Vdash^*_{\Box A} c')\iff c=c'$.

As the final general construction on $\mathcal{R}$, we define a lifting monad $(-)_\perp$, which we use for modelling partial functions. It is defined by $|A_\perp|=|A|+\{\perp\}$, ${A_\perp}^*=A^*$, $(A_\perp)_{\mathcal{G}}=A_{\mathcal{G}}$, $\Vdash^*_{A_\perp}=\Vdash^*_A$, $(\alpha,e\Vdash_{A_\perp} a\in|A|)\iff(\alpha,e\Vdash_A a)$, and $(\alpha,e\Vdash_{A_\perp}\perp)$ always. There are evident morphisms $A\to A_\perp$, $(A_\perp)_\perp\to A_\perp$, $A\bullet(B_\perp)\to(A\bullet B)_\perp$ for any $\bullet\in\{\otimes,\otimes_1,\times,*\}$, and $\Box(A_\perp)\to(\Box A)_\perp$.

We remark in passing that the definition of $\mathcal{R}$ and the construction of its structure are very similar to the double glueing construction of Hyland and Schalk [12].

3.4 An Instance for Logarithmic Space

We consider $\mathcal{R}$ with respect to the monoid $\mathcal{M}=\{\langle l,k,m\rangle\in\mathbb{N}^3\mid l\le m\}$ with addition $\langle l,k,m\rangle+\langle l',k',m'\rangle=\langle l+l',\max(k,k'),\max(l+l',m,m')\rangle$. The neutral element 0 is $\langle 0,0,0\rangle$. As ordering we use $\langle l,k,m\rangle\le\langle l',k',m'\rangle\iff(l\le l')\wedge(k\le k')\wedge(m\le m')$. The intended meaning of a triple $\langle l,k,m\rangle$ is that l is the abstract length of an object and $\langle k,m\rangle$ is a bound on the additional memory a realiser may use. We will allow realisers to use k numbers with range $\{0,\dots,m\}$.

We now consider the structure of $\mathcal{R}$ with respect to $\mathcal{M}$, starting with an implementation of the base types of LogFPL.

Definition 2. *An object A is* simple *if it enjoys the following properties.*

1. *Whenever $\alpha, e \Vdash a$ holds, then there is a $\mathcal{L}$-map $e' \colon A^+_{\mathcal{G}} \to A^-_{\mathcal{G}}$ with $e' \circ e = id$.*
2. *A^* is a singleton and whenever $\alpha, c \Vdash^* a^*$ holds, then c is the empty function and $0, c \Vdash^* a^*$ holds as well.*
3. *Both $A^-_{\mathcal{G}}$ and $A^+_{\mathcal{G}}$ have at least one element x with $l(x) = 0$.*

We note that if A and B are simple then so are $A \otimes_1 B$ and $A * B$, but not in general $A \otimes B$.

All the basic data types we now define are simple. Since the co-realisation relation is uniquely determined by the definition of simpleness, we just show the realisation part.

Diamond. $|\Diamond| = \{\Diamond\}$, $\Diamond_{\mathcal{G}} = I_{\mathcal{G}}$, $(\alpha + \langle 1, 0, 1\rangle, e \Vdash_{\Diamond} \Diamond) \iff (\alpha, e \Vdash_I *)$
Booleans. $|\mathbf{B}| = \{\mathtt{ff}, \mathtt{tt}\}$, $\mathbf{B}_{\mathcal{G}} = (\{\mathtt{ff}, \mathtt{tt}\}, 1)$, $(\alpha, e \Vdash_{\mathbf{B}} b) \iff (e(*) = b)$
Small Numbers. $|\mathbf{SN}| = \mathbb{N}$, $\mathbf{SN}_{\mathcal{G}} = (\mathbf{L}(\mathbf{B}), \mathbf{L}(1))$ and $(\langle l, k, m\rangle, e \Vdash_{\mathbf{SN}} n)$ holds if and only if both $(l \geq n)$ holds and $e(s)$ equals the last s bits of n in binary.
Lists. Let A be a simple object. Define the simple object $\mathbf{L}(A)$ to have as underlying set $|\mathbf{L}(A)|$ the set of finite lists on $|A|$. The object $\mathbf{L}(A)_{\mathcal{G}}$ is given by $((A^+_{\mathcal{G}} + A^-_{\mathcal{G}}) \otimes \mathbf{L}(\mathbf{B}), A^-_{\mathcal{G}} \otimes \mathbf{L}(\mathbf{B}))$. The intention of $A^-_{\mathcal{G}} \otimes \mathbf{L}(\mathbf{B})$ is that a question consists of a pointer into the list, encoded in binary (without leading zeros) as an element of $\mathbf{L}(\mathbf{B})$, together with a question for the element at the position pointed at.

$$\alpha, e \Vdash \langle a_0, \dots, a_n\rangle \iff \begin{array}{l} \exists \vec{\alpha} \in \mathcal{M}^{n+1}. \exists \vec{e} \in (I \to A_{\mathcal{G}})^{n+1}. \\ \quad (\alpha \geq \langle n+1, 0, n+1\rangle + \alpha_0 + \cdots + \alpha_n) \\ \quad \wedge (\forall i.\ i \leq n \implies \alpha_i, e_i \Vdash_A a_i) \\ \quad \wedge (\forall i, q.\ i \leq n \implies e(q, i) = \langle inl(e_i(q)), i\rangle) \\ \quad \wedge (\forall i, q.\ i > n \implies e(q, i) = \langle inr(q), i\rangle) \end{array}$$

We note that $\mathbf{L}(A)$ as such is not yet very useful, since, for instance, there is no map *tail*: $\mathbf{L}(A) \to \mathbf{L}(A)$, as we do not always have enough space to map a question $\langle q_A, i\rangle$ to $\langle q_A, i+1\rangle$. We address this with the construction $M(-)$ below.

Data storage Often when passing a question to an object, we also need to store some data that we need again once the answer arrives. Such data storage is captured by the functor $!(-)$ defined as follows. The set of underlying elements of $!A$ is inherited from A, i.e. $|!A| = |A|$. The realising object $(!A)_{\mathcal{G}}$ is $!(A_{\mathcal{G}})$ and the realisation relation is the smallest realisation relation satisfying $\langle l, k, m\rangle, e \Vdash_A a \implies \langle l, k+1, m\rangle, !e \Vdash_{!A} a$ in addition to the general requirements for objects in $\mathcal{R}$. The set of co-elements is defined by $(!A)^* = |\mathbf{S}| \to A^*$ and the co-realisation relation on $!A$ is given by

$$\alpha, c \Vdash^*_{!A} a^* \iff \forall s \in |\mathbf{S}|. \exists c_s. (\alpha, c_s \Vdash^*_A a^*(s)) \wedge (\forall q. c(q, s) = \langle c_s(q), s\rangle).$$

While there is a natural dereliction map $!A \to A$, there is no digging map $!A \to !!A$, since we do not have an additional $\Diamond$ that we would need to encode two trees in one.

Lemma 3. *There are natural transformations $\Box !A \to !\Box A$, $(!A \otimes !B) \to !(A \otimes B)$ and $!(A_\perp) \to (!A)_\perp$ as well as a natural isomorphism $(!A \otimes_1 !B) \cong !(A \otimes_1 B)$.*

Memory allocation We have defined the monoid $\mathcal{M}$ with the intuition that if e realises some element with bound $\langle l,k,m\rangle$ then e can use k memory locations of size $\log(m)+1$. However, the above definition of the data types is such that, besides the question itself, no memory can be assumed. We now add a memory supply to the data types.

Let A be a simple object such that for each $a \in |A|$ there exist l, m and e satisfying $\langle l,0,m\rangle, e \Vdash_A a$. To define a simple object MA, let $|MA| = |A|$, $(MA)^* = A^*$ and $(MA)_{\mathcal{G}} = (A^+_{\mathcal{G}} \otimes \mathbf{L}(\mathbf{L}(1)), A^-_{\mathcal{G}} \otimes \mathbf{L}(\mathbf{L}(1)))$. The intended meaning of $(MA)_{\mathcal{G}}$ is that a question in $A^-_{\mathcal{G}}$ comes with a sufficiently large block of memory, viewed as an element of $\mathbf{L}(\mathbf{L}(1))$. This block of memory can be used for computing an answer, but must be returned with the answer at the end of the computation. We define memory blocks as follows.

Definition 3. *Let $\langle k,m\rangle$ be a pair of natural numbers. The set $\mathbf{L}_{\langle k,m\rangle} \subseteq \mathbf{L}(\mathbf{L}(1))$ of $\langle k,m\rangle$-*memory blocks *is the least set containing all the lists $x = \langle x_1,\dots,x_n\rangle$ with $n \geq k$, such that each x_i is a nonempty list in $\mathbf{L}(1)$ and $\forall i.\ (1 \leq i \leq n-1) \implies l(x_i) = \lfloor l(x)/n \rfloor$ and $\lfloor l(x)/n \rfloor \geq (\log(m)+1)$ and $l(x_n) = (l(x) \mod n)$ all hold.*

In short, $\mathbf{L}_{\langle k,m\rangle}$ provides enough space for at least k binary numbers with range $\{0,\dots,m\}$. Given a memory block $x = \langle x_1,\dots,x_n\rangle \in \mathbf{L}_{\langle k,m\rangle}$, we refer to the length of x_1 as the *size of memory locations in x*. The definition of $\mathbf{L}_{\langle k,m\rangle}$ is such that, for any two nonempty lists $s,s' \in \mathbf{L}_{\langle k,m\rangle}$, we have $s = s'$ if and only if both $l(s) = l(s')$ and $l(head(s)) = l(head(s'))$ hold. This makes it easy to reconstruct the original memory block after part of it has been used in a computation.

Now the realisation on MA is defined such that $\langle l,k,m\rangle, e \Vdash_{MA} a$ holds if and only if $\forall s \in \mathbf{L}_{\langle k,m\rangle}.\, \exists e_s.\, (\langle l,0,m\rangle, e_s \Vdash_A a) \wedge (\forall q.\, e(q,s) = \langle e_s(q), s\rangle)$ holds. The co-realisation relation on MA is determined by the requirement that MA be simple.

Lemma 4. *There are isomorphisms $M(A \otimes_1 B) \cong MA \otimes_1 MB$ and $M(A * B) \cong MA * MB$ for all objects A and B for which MA and MB are defined.*

All the base types of LogFPL are interpreted by objects of the form MA, e.g. we use $M\mathbf{L}(M\mathbf{B})$ as the interpretation of $\mathbf{L}(\mathbf{B})$. Some simplification is given by the next lemma.

Lemma 5. *There are maps $M\mathbf{L}(MA) \to M\mathbf{L}(A)$ and $!M\mathbf{L}(A) \to M\mathbf{L}(MA)$.*

Proposition 3. *The underlying function of each map $f\colon\ !^k M\mathbf{L}(\mathbf{B}) \otimes !^k M\mathbf{L}(I) \to M\mathbf{L}(\mathbf{B})$ is* LOGSPACE-*computable.*

Proof (Sketch). We construct a Turing Machine T using a realiser r for f. From r we build a sub-routine of T, which on one work tape takes a (code for a) question $q \in (M\mathbf{L}(\mathbf{B}))^-_{\mathcal{G}}$ and on the same tape returns an answer in $(M\mathbf{L}(\mathbf{B}))^+_{\mathcal{G}}$. The sub-routine works by running r, and whenever r returns a question for its argument, the sub-routine reads the relevant part from the input tape and passes the answer to r. Since r is a morphism in $\mathcal{L}$, this uses only linear space in the size of the question q. We now use this sub-routine to compute the output of T. Write x, y for the two inputs to T. First we write a memory block in $\mathbf{L}_{\langle k,|x|+|y|\rangle}$ to one work tape. This can be done since k is constant and the block has logarithmic size. Using this block, the above sub-routine is used to compute the bits of the output one-by-one. Since f is non-size-increasing, it suffices to continue until bit number $|x|+|y|$. Hence, we only need to consider questions as large as $\log(|x|+|y|)+1$. Since the space needed for answering questions is linear in the size of the question, the whole procedure uses logarithmic space in the size of the inputs.

It remains to show that the output of this algorithm is correct. Let x and y be the two inputs. We then have $\langle |x|, 0, |x| \rangle, e_x \Vdash_{M\mathbf{L}(\mathbf{B})} x$ and $\langle |y|, 0, |y| \rangle, e_y \Vdash_{M\mathbf{L}(I)} y$, where e_x and e_y are programs that answer the given question by reading the input tape (note that x and y are now fixed, so that $|x|$ and $|y|$ appear as constants in the space-usage of e_x and e_y). By the realisation property, we have $\langle |x|+|y|, k, |x|+|y| \rangle, r \cdot (!^k e_x \otimes !^k e_y) \Vdash_{M\mathbf{L}(\mathbf{B})} f(x,y)$. Since the sub-routine described above computes $r \cdot (!^k e_x \otimes !^k e_y)$, it then follows by the definition of the realisation on $M\mathbf{L}(\mathbf{B})$, that the algorithm computes the correct output.

We now implement the constants for lists. We omit the other constants for space reasons.

Lemma 6. *Let $A = MA'$ and $B = MB'$ be simple objects of $\mathcal{R}$. Then there are morphisms $cons\colon M\Diamond \otimes_1 A \otimes_1 !M\mathbf{L}(A) \to M\mathbf{L}(A)$, $hdtl\colon !M\mathbf{L}(A) \to (M\Diamond \otimes_1 A \otimes_1 M\mathbf{L}(A))_\perp$ and $empty\colon M\mathbf{L}(A) \to M\mathbf{B}$ defined by:*

$$cons(a,\vec{a}) = \langle a, \vec{a} \rangle \qquad hdtl(\langle\rangle) = \bot \qquad empty(\langle\rangle) = \mathtt{tt}$$
$$hdtl(\langle a, \vec{a} \rangle) = \langle \Diamond, a, \vec{a} \rangle \qquad empty(\langle a, \vec{a} \rangle) = \mathtt{ff}$$

To complete the constructions for lists, it remains to implement recursion. The properties we need of the result type of a recursion are captured by the next three definitions.

Definition 4. *An object A* has unique answers *if, for all $x \in |A|$ and all $q \in A^-_{\mathcal{G}}$, there exists a unique $a \in A^+_{\mathcal{G}}$ such that, for all α and e, $(\alpha, e \Vdash x)$ implies $e(q) = a$.*

The next definition expresses that we only need to consider small questions. For instance, for an element $x \in \mathbf{L}(\mathbf{B})$ with $\langle l,k,m \rangle, e \Vdash x$, we only ever need to consider questions q with $l(q) \leq \log(m)+1$, since all larger questions will be out of range. We use this property to ensure that we do not run out of memory during a recursion.

Definition 5. *Object A* has n-small questions *if there are maps $\mathsf{cut}\colon A^-_{\mathcal{G}} \otimes \mathbf{L}(1) \to A^-_{\mathcal{G}}$ and $\mathsf{expand}\colon A^-_{\mathcal{G}} \otimes A^+_{\mathcal{G}} \to A^+_{\mathcal{G}}$, such that $\langle l,k,m \rangle, e \Vdash x$ implies $l(\mathsf{cut}(q,s)) \leq l(s)$ and $\mathsf{expand}(q, e(\mathsf{cut}(q,s))) = e(q)$ for all $q \in A^-_{\mathcal{G}}$ and all $s \in \mathbf{L}(1)$ with $l(s) \geq n(\log(m)+1)$.*

Finally, to write a program for recursion we must be able to store questions and answers.

Definition 6. *An object A is* n-encodable *if, for each $X \in \{A^-_{\mathcal{G}}, A^+_{\mathcal{G}}\}$, there are maps $\mathsf{len}_X : X \to \mathbf{L}(1)$, $\mathsf{code}_X : \Diamond^n \otimes X \to \mathbf{S}$ and $\mathsf{decode}_X : \mathbf{S} \to \Diamond^n \otimes X$, with the properties $\forall x \in X.\, l(x) = l(\mathsf{len}_X(x))$ and $\mathsf{decode}_X \circ \mathsf{code}_X = id$.*

Each recursion type, as defined in Fig. 2, is simple and enjoys the properties of these definitions. In general, this holds neither for $A \otimes B$ nor for $A \multimap B$.

Proposition 4. *Let R be a simple object with unique answers and k_R-small questions, which in addition is k_R-encodable. Let $\langle 0,k,m \rangle, e_h \Vdash_{!^i\Diamond \otimes (\Box !^i MA) \otimes !^i MR \multimap MR} h$. Then the function $f_h\colon !^{4+7k_R} MR \otimes \Box !^{i+4+7k_R} M\mathbf{L}(A) \to MR$ given by $f_h(g, \langle\rangle) = g$ and $f_h(g, \langle x, \vec{x} \rangle) = h(\Diamond, x, f_h(g, \vec{x}))$ is realised with bound $\langle 0, k+4+7k_R, m \rangle$.*

Let us explain the idea of the realiser for recursion informally. Suppose, for instance, we want to compute $f_h(g, \langle a_1, a_1, a_0 \rangle)$ for certain elements g, a_0 and a_1. Write h_0 and h_1 for the functions of type $!MR \multimap MR$ that arise from h by instantiating the first argument with a_0 and a_1 respectively. Then, the algorithm for computing $f_h(g, \langle a_1, a_1, a_0 \rangle)$ can be

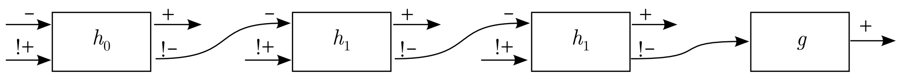

Fig. 4. Illustration of recursion

depicted as in Fig. 4. The edge labelled with $-$ represents an initial question. The edges labelled with $!-$ represent questions of h_j to the recursion argument, the answer to which is expected in the $!+$-port of the same box. The presence of the modality ! expresses that along with the question, h_j may pass some store that must be returned unchanged with the answer. Now, if we were to remember the store of each h_j in the course of the recursion then we would need linear space (in the length of the second argument of f) to compute it. Instead, whenever h_j sends a question $!-$ to its recursion argument, we forget h_j's local store (the value stored in the !) and just pass the question to the next instance of h_j. If g or some h_j gives an answer in its $+$-port, then we would like to pass this answer to the edge labelled with $!+$ into the next box to the left. However, we cannot go from $+$ to $!+$, as there is no way to reconstruct the store that we have discarded before. Nevertheless, it is possible to recompute the store. We remember the answer in $+$ and the position of the box that gave the answer and restart the computation from the beginning. During this computation, the question in $!-$, which is answered by the saved answer, will be asked again. We can then take the store of the question in $!-$ and the saved answer in $+$ to construct an answer in $!+$ and pass it on as an answer to the question in $!-$. In this way, the local store of the step functions h_j can be recomputed. For the correctness of this procedure it is crucial that each h_j asks at most one question of the recursion argument, which follows from the co-realisation information on h. It is instructive to check this property for the step-function of *focus* in Sect. 2.1.

This description of the implementation of recursion can be formalised in $\mathcal{G}$ by a morphism $rec\colon\, !^r(MR)_{\mathcal{G}} \otimes !^r(!^i(\Diamond_{\mathcal{G}} \otimes (MA)_{\mathcal{G}} \otimes (MR)_{\mathcal{G}}) \multimap (MR)_{\mathcal{G}}) \otimes !^r !^i \mathbf{L}(A)_{\mathcal{G}} \to (MR)_{\mathcal{G}}$, where $r = 4 + 7k_R$. The morphism rec is defined such that, for all $\langle l_g, k_g, m_g\rangle, e_g \Vdash_{MR} g$ and $\langle l_x, k_x, m_x\rangle, e_x \Vdash_{M\mathbf{L}(A)} x$, the map $rec \cdot (!^r e_g \otimes !^r e_h \otimes !^{r+i} e_x)$ realises $f_h(g,x)$ with bound $\alpha = \langle 0, k+r, m\rangle + \langle l_g, k_g + r, m_g\rangle + \langle l_x, k_x + r + i, m_x\rangle$. Prop. 4 follows from this property.

More details about the implementation of rec can be found in the appendix. Here, we outline how the r memory locations in the modalities $!^r$ are used by rec. The definition of the realisation on MR is such that we can assume that a question to $rec \cdot (!^r e_g \otimes !^r e_h \otimes !^{r+i} e_x)$ comes together with a memory block $m \in \mathbf{L}_{\langle k_\alpha, m_\alpha\rangle}$. Since we know $\alpha = \langle l_\alpha, \max(k, k_g, k_x + i) + r, \max(m, m_g, m_x)\rangle$, we can thus assume at least r memory locations of size $\log(m_\alpha) + 1$ and we can assume that the rest of the memory is still large enough to satisfy the memory requirements of g, h and x. We use the $r = 4 + 7k_R$ memory cells, to store the following data: the initial question (*startqn*); the current question (*qn*); the current recursion depth (*d*); whether some answer has already been computed (*store*), the stored answer (*sa*) and its recursion depth (*sd*); a flag (*xo*) to record how answers from the argument x should be interpreted. Since R has k_R-small questions and the questions and answers are k_R-encodable, each of the fields *startqn*, *qn* and *sa* can be stored using $2k_R$ memory locations in m. The fields *store*, *d*, *sd* and *xo* can each be stored in a single memory location in m, since d and sd represent numbers in $\{0, \dots, |x|\}$. The remaining k_R memory locations provide enough memory for the current question/answer.

3.5 Interpreting LogFPL

The types of LogFPL are interpreted in $\mathcal{R}$ by the following clauses.

$$\llbracket A \rrbracket = MA, \text{ where } A \in \{I, \Diamond, \mathbf{B}, \mathbf{SN}\}$$

$$\llbracket \mathbf{L}(A) \rrbracket = M(\mathbf{L}(\llbracket A \rrbracket)) \qquad \llbracket A \overset{\cdot,i}{\multimap} B \rrbracket = !^i \llbracket A \rrbracket \multimap \llbracket B \rrbracket_\perp$$

$$\llbracket A \bullet B \rrbracket = \llbracket A \rrbracket \bullet \llbracket B \rrbracket, \text{ where } \bullet \in \{*, \otimes_1, \times\} \qquad \llbracket A \overset{\infty,i}{\multimap} B \rrbracket = \Box !^i \llbracket A \rrbracket \multimap \llbracket B \rrbracket_\perp$$

Contexts are interpreted by $\llbracket \Gamma \rrbracket = \llbracket \Gamma \rrbracket' \otimes \llbracket \Gamma \rrbracket_1$, where $\llbracket () \rrbracket' = \llbracket () \rrbracket_1 = I$ and

$$\llbracket \Gamma, x \overset{\langle \infty, i\rangle}{:} A \rrbracket' = \llbracket \Gamma \rrbracket' \otimes \Box !^i \llbracket A \rrbracket \qquad \llbracket \Gamma, x \overset{\langle \infty, i\rangle}{:} A \rrbracket_1 = \llbracket \Gamma \rrbracket_1$$

$$\llbracket \Gamma, x \overset{\langle \cdot, i\rangle}{:} A \rrbracket' = \llbracket \Gamma \rrbracket' \otimes !^i \llbracket A \rrbracket \qquad \llbracket \Gamma, x \overset{\langle \cdot, i\rangle}{:} A \rrbracket_1 = \llbracket \Gamma \rrbracket_1$$

$$\llbracket \Gamma, x \overset{\langle 1, i\rangle}{:} A \rrbracket' = \llbracket \Gamma \rrbracket' \qquad \llbracket \Gamma, x \overset{\langle 1, i\rangle}{:} A \rrbracket_1 = \llbracket \Gamma \rrbracket_1 \otimes_1 !^i \llbracket A \rrbracket.$$

Proposition 5. *For each judgement $\Gamma \vdash M \colon A$, there is a map $m \colon \llbracket \Gamma \rrbracket \to \llbracket A \rrbracket_\perp$ in $\mathcal{R}$, such that the underlying function of m amounts to the functional interpretation of M.*

Proof. The proof goes by induction on the derivation of $\Gamma \vdash M \colon A$. As a representative case, we consider ($\otimes_1$-E). The induction hypothesis gives $\llbracket \Gamma \rrbracket \to (\llbracket A \rrbracket \otimes_1 \llbracket B \rrbracket)_\perp$ and $\llbracket \Delta' \rrbracket' \otimes (\llbracket \Delta_1 \rrbracket_1 \otimes_1 !^i \llbracket A \rrbracket \otimes_1 !^i \llbracket B \rrbracket) \to \llbracket C \rrbracket_\perp$ for appropriate Δ' and Δ_1 with $\Delta = \Delta', \Delta_1$. With the isomorphism $!(X \otimes_1 Y) \cong !X \otimes_1 !Y$ and the operations on $(-)_\perp$, we obtain a map $\llbracket \Delta' \rrbracket' \otimes (\llbracket \Delta_1 \rrbracket_1 \otimes_1 !^i \llbracket \Gamma \rrbracket) \to \llbracket C \rrbracket_\perp$. Using the morphisms from Lemma 3, we obtain $\llbracket !^{\langle 1,i\rangle} \Gamma \rrbracket \to !^i \llbracket \Gamma \rrbracket$. By use of $X \otimes (Y \otimes_1 Z) \to Y \otimes_1 (X \otimes Z)$, we obtain $\llbracket \Delta, !^{\langle 1,i\rangle} \Gamma \rrbracket \to \llbracket \Delta' \rrbracket' \otimes (\llbracket \Delta_1 \rrbracket_1 \otimes_1 !^i \llbracket \Gamma \rrbracket)$. Putting this together gives the required $\llbracket \Delta, !^{\langle 1,i\rangle} \Gamma \rrbracket \to \llbracket C \rrbracket_\perp$.

To see that Prop. 4 is applicable for the interpretation of recursion, notice that $\langle l, k, m\rangle, e \Vdash_{A \multimap M\mathbf{B}^k} d$ implies $\langle 0, k, m\rangle, e \Vdash_{A \multimap M\mathbf{B}^k} d$. This follows using the definition of the realisation on $\multimap$, since the same property holds by definition for $M\mathbf{B}^k$. Furthermore, each recursion type A (as defined in Fig. 2) is interpreted by a $k(A)$-encodable, simple object with unique answers and $k(A)$-small questions. Thus Prop. 4 can be used for the interpretation of recursion. □

With the interpretation, LOGSPACE-soundness (Prop. 1) now follows as in Prop. 3.

4 Conclusion and Further Work

We have introduced a computation-by-interaction model for space-restricted computation and have designed a type system for LOGSPACE on its basis. We have thus demonstrated that the model captures LOGSPACE-computation in a compositional fashion. As a way of controlling the subtle interaction-behaviour in the model, we have identified the concept of co-realisability. Using this concept, we were able to formalise the subtle differences between types such as $(\Box A) \otimes B$, $A \otimes B$ and $A \otimes_1 B$ in a unified way.

For further work, we plan to consider extensions of LogFPL. We believe that there is a lot of structure left in the model that can be used to justify extensions. For instance, we conjecture that it is possible to define first-order linear functions by recursion, i.e. to allow a form of parameter substitution. We expect that, using an argument similar to the Chu-space approach of [10], co-realisability can be used to reduce recursion with first-order functional result type to iterated base-type recursion.

Other interesting directions for further work include to consider other complexity classes such as polylogarithmic space and to further explore the connections to linear logic. It may also be interesting to find out if recent work on algorithmic game semantics, as in e.g. [7, 1], can be utilised for our purposes. A referee raised the interesting question about the relation of our computation model based on a GoI-situation to oracle-based computing in traditional complexity models. We plan to consider this in further work.

Acknowledgments I thank Martin Hofmann and Jan Johannsen for helpful advice and the anonymous referees for their comments and corrections. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the project Pro.Platz.

References

1. S. Abramsky, D. R. Ghica, A.S. Murawski, and C.-H.L. Ong. Applying game semantics to compositional software modeling and verification. In *TACAS04*, pages 421–435, 2004.
2. S. Abramsky, E. Haghverdi, and P.J. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.
3. S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic (extended abstract). In *FSTTCS92*, pages 291–301, 1992.
4. S.J. Bellantoni. *Predicative Recursion and Computational Complexity*. PhD thesis, University of Toronto, 1992.
5. U. Dal Lago and M. Hofmann. Quantitative models and implicit complexity. In *FSTTCS05*, pages 189–200, 2005.
6. V. Danos, H. Herbelin, and L. Regnier. Game semantics & abstract machines. In *LICS96*, pages 394–405, 1996.
7. D.R. Ghica and G. McCusker. Reasoning about idealized algol using regular languages. In *ICALP*, pages 103–115, 2000.
8. E. Hahgverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness*. PhD thesis, University of Ottawa, 2000.
9. M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183:57–85, 1999.
10. M. Hofmann. Semantics of linear/modal lambda calculus. *Journal of Functional Programming*, 9(3):247–277, 1999.
11. M. Hofmann. Programming languages capturing complexity classes. *SIGACT News*, 31(1):31–42, 2000.
12. J.M.E. Hyland and A. Schalk. Glueing and orthogonality for models of linear logic. *Theoretical Computer Science*, 294(1–2):183–231, 2003.
13. N.D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 228(1–2):151–174, 1999.
14. L. Kristiansen. Neat function algebraic characterizations of LOGSPACE and LINSPACE. *Computational Complexity*, 14:72–88, 2005.
15. J.R. Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1994.
16. I. Mackie. The geometry of interaction machine. In *POPL95*, 1995.
17. P. Møller-Neegaard. *Complexity Aspects of Programming Language Design*. PhD thesis, Brandeis University, 2004.
18. Peter Møller Neergaard. A functional language for logarithmic space. In *APLAS*, pages 311–326, 2004.
19. D. Sannella, M. Hofmann, D. Aspinall, S. Gilmore, I. Stark, L. Beringer, H.-W. Loidl, K. MacKenzie, A. Momigliano, and O. Shkaravska. Mobile Resource Guarantees. In *Trends in Functional Programing*, volume 6, Tallinn, Estonia, Sep 23–24, 2005. Intellect.

A Appendix

We give details of the implementation of the realiser

$$rec\colon\ !^r(MR)_{\mathcal{G}}\otimes !^r(!^i(\Diamond_{\mathcal{G}}\otimes(MA)_{\mathcal{G}}\otimes(MR)_{\mathcal{G}})\multimap(MR)_{\mathcal{G}})\otimes !^r!^i\mathbf{L}(A)_{\mathcal{G}}\to(MR)_{\mathcal{G}},$$

for recursion, as described in Sec. 3.4. To make the description more readable, we make the passing around of store (in $!^r-$) and of memory blocks (in $M-$) implicit. That is, we describe a function rec' that has access to a global persistent store of the form described above and in which memory (de)allocation is handled implicitly. From this description an implementation of rec, which e.g. uses $!^r!^iMR^-_{\mathcal{G}}$ instead of $!^iR^-_{\mathcal{G}}$, can be derived.

The input of rec' is in

$$\underbrace{R^+_{\mathcal{G}}}_{g}+\underbrace{!^i\Diamond^-_{\mathcal{G}}+!^iA^-_{\mathcal{G}}+!^iR^-_{\mathcal{G}}+R^+_{\mathcal{G}}}_{h}+\underbrace{(\mathbf{L}(A))^+_{\mathcal{G}}}_{x}+\underbrace{R^-_{\mathcal{G}}}_{\text{initial question}}$$

and its output is in

$$\underbrace{R^-_{\mathcal{G}}}_{g}+\underbrace{!^i\Diamond^+_{\mathcal{G}}+!^iA^+_{\mathcal{G}}+!^iR^+_{\mathcal{G}}+R^-_{\mathcal{G}}}_{h}+\underbrace{(\mathbf{L}(A))^-_{\mathcal{G}}}_{x}+\underbrace{R^+_{\mathcal{G}}}_{\text{final answer}}\ .$$

We write in_g, in_{h_d}, in_{h_r}, in_{h_x}, in_h, in_x, in_1 for the injections (from left to right) into these types. We assume a global store containing $startqn$, qn, d, $store$, sd, sa, xo. We make memory (de)allocation implicit and write len for the size of memory locations in the memory block. Let q_A be some element of $A^-_{\mathcal{G}}$ with $l(q_A)=0$, which exists because A is a simple object.

With this notation, the morphism rec can be implemented as in Fig 1.

```
$rec'\ in_1(q) =$ (initial question)
    $qn := \mathsf{cut}(q, len)$; $startqn := qn$; $d := 0$; $store := 0$; $xo := 1$;
    **return** $in_x(\langle q_A, d\rangle, nil^i)$
$rec'\ in_g(a) =$ (answer from $g$)
    **if** $d = 0$ **then** (if the recursion depth is 0 then $a$ is the final answer)
        $in_1(\mathsf{expand}(qn, a))$
    **else** (at recursion depth $> 0$ we store the answer and restart)
        $store$:=1; $sa$:=$a$; $sd$:=$d$;
        $qn$:=$startqn$; $d$:=0; $xo$:=1;
        **return** $in_x(\langle q_A, nil\rangle, nil^i)$
    **end**
$rec'\ in_{h_d}(*, s_l) = in_{h_d}(*, s_l)$
$rec'\ in_{h_r}(q, s_l) =$ ($h$ asks for the recursion argument)
    **if** $store = 1 \wedge sd = d + 1$ **then** (if the store contains the answer we return it)
        $in_{h_r}(\mathsf{expand}(q, sa), s_l)$
    **else** (else we free $h$'s local store $s_l$ and continue at recursion depth $d + 1$)
        $qn$:=$\mathsf{cut}(q, len)$; $d$:=$d + 1$; $xo$:=1;
        **return** $in_x(\langle q_A, d\rangle, nil^i)$
    **end**
$rec'\ in_{h_x}(q, s_l) =$ (if $h$ asks $q$ of its first argument, we send $q$ to element $d$ in list $x$)
    $in_x(\langle q, d\rangle, s_l)$
$rec'\ in_h(a) = rec'\ in_g(a)$ (answers from $h$ are treated like those from $g$)
$rec'\ in_x(a, s_l) =$ (answer from $x$)
    **if** $xo = 0$ **then** (if $a$ answers a question from $h$, we pass it back to $h$)
        $in_{h_x}(a, s_l)$
    **else** (else send $qn$ to $g$ if $x$ has no element at position $d$ and to $h$ otherwise)
        $xo$:=0;
        **if** $a = \langle inr(q_A), d\rangle$ **then** ($\langle inr(q_A), d\rangle$ means list $x$ has $\leq d$ elements)
            **if** $store = 0$ **then return** $in_g(qn)$ **else** $\bot$
        **else**
            **return** $in_h(qn)$
        **end**
    **end**
```

Algorithm 1: Implementation of recursion