Functional Programming in Sublinear Space

Ugo Dal Lago[1] and Ulrich Schöpp[2]*

[1] University of Bologna, Italy
[2] University of Munich, Germany

Abstract. We consider the problem of functional programming with data in external memory, in particular as it appears in sublinear space computation. Writing programs with sublinear space usage often requires one to use special implementation techniques for otherwise easy tasks, e.g. one cannot compose functions directly for lack of space for the intermediate result, but must instead compute and recompute small parts of the intermediate result on demand. In this paper, we study how the implementation of such techniques can be supported by functional programming languages.
Our approach is based on modeling computation by interaction using the Int construction of Joyal, Street & Verity. We derive functional programming constructs from the structure obtained by applying the Int construction to a term model of a given functional language. The thus derived functional language is formulated by means of a type system inspired by Baillot & Terui's Dual Light Affine Logic. We assess its expressiveness by showing that it captures LOGSPACE.

1 Introduction

A central goal in programming language theory is to design programming languages that allow a programmer to express efficient algorithms in a convenient way. The programmer should be able to focus on algorithmic issues as much as possible and the programming language should give him the means to delegate inessential implementation details to the computer.

In this paper we study programming language constructs that are useful for expressing sublinear space algorithms. Sublinear space algorithms use less memory space than would be needed to store their input. They are useful for computing with large inputs that do not fit into memory.

When writing programs with sublinear space usage, one must often use special techniques for tasks that would normally be simple. A typical example is the composition of two algorithms. In order to remain in sublinear space, one cannot run them one after the other, as there may not be enough space to store the intermediate result. Instead, one can implement composition by storing at any time only a small part of the intermediate value and by (re)computing small parts only as they are needed.

Since it is easy to make mistakes in the implementation of such on-demand recomputation of intermediate values, we believe that programming language support should be very useful for such tasks. Instead of implementing composition with on-demand

* This work was carried out while Ulrich Schöpp was supported by a fellowship of the Institute of Advanced Studies at the University of Bologna.

recomputation by hand, the programmer should be able to write function composition in the usual way and have a compiler generate the complicated program.

The possibilities of doing this have been explored in work on implicit characterisations of LOGSPACE. A number of characterisations of this complexity class have been explored in terms of function algebras [14] and linear logics [15] (there is more work for LOGSPACE-predicates, e.g. [10, 5], but we focus on functions here). However, these characterisations are still far away from being real programming languages.

Here we work towards the goal of making the abstractions explored in this line of work usable in programming. In contrast to previous work [14, 15], we aim to enrich the language with constructs for working with on-demand recomputation conveniently, rather than hiding it completely. This is in line with the fact that sublinear space algorithms are usually not used in isolation and will generally appear within larger programs. Language support for writing sublinear space algorithms should not become a hindrance in other parts of the program that do not operate on large data.

In this paper we present the functional programming language INTML for programming with sublinear space. This language is derived from an instance of the Int construction [7]. Our thesis is that the Int construction naturally captures space bounded computation and thus exposes mathematical structure that is useful for writing space bounded programs. Even though the type system of INTML is quite simple when compared to earlier higher-order type systems for LOGSPACE [15], INTML allows LOGSPACE algorithms to be written in a natural way, as is discussed in Sec. 3.2.

2 Space-Bounded Computation

Our approach to representing sublinear space computation in a functional programming language is best explained by analysing the definition of space complexity classes.

In the definition of space complexity classes, in particular sublinear space complexity classes, one uses *Offline Turing Machines* (OTMs) instead of standard Turing Machines. Offline Turing Machines are multi-tape Turing Machines that differ from the standard ones in that the input tape is read-only, the output tape is write-only and the output head may be moved in one direction only; finally the input and output tapes do not count towards the space usage of an Offline Turing Machine.

The definition of Offline Turing Machine captures a special class of Turing Machines that do not store their input or output in memory, but instead have (random) access to some externally stored input, and that give their output as a stream of characters. Since neither input nor output must be stored in memory, it is justified to count only the work tape towards the space usage of an Offline Turing Machine.

More formally, while a normal Turing Machine computes a function $\Sigma^* \to \Sigma^*$ on words over an alphabet Σ, an Offline Turing Machine may be seen as a function of type

$$(\mathit{State} \times \Sigma) + \mathbb{N} \longrightarrow (\mathit{State} \times \mathbb{N}) + \Sigma \ ,$$

where $A + B$ denotes the (tagged) disjoint union $\{inl(x) \mid x \in A\} \cup \{inr(y) \mid y \in B\}$. An input $n \in \mathbb{N}$ stands for the request to compute the n-th character on the OTM's output tape. An output in Σ is a response to this request. Whenever the OTM wants to read a character from its input tape, it outputs a pair $\langle s, n\rangle \in \mathit{State} \times \mathbb{N}$, where n is the number

of the input character it wants to read and s is its machine state, comprising finite control state, work tape contents, etc. Receiving this request, the environment looks up the n-th input character i_n and restarts the machine with input $\langle s, i_n \rangle \in \mathit{State} \times \Sigma$. It supplies the machine state s that was part of the input request, so that the machine can resume its computation from the point where it requested an input.

In this way, we can consider each Offline Turing Machine as a normal Turing Machine with the special input/output interface given by the type above. This special machine needs space only to store the work tape(s) of the OTM and the positions of the OTM's input and output heads. This view justifies the exclusion of the input and output tapes in the definition of the space usage of OTMs, as long as the space usage is at least logarithmic. We will not consider OTMs with sublogarithmic space usage here and indeed 'Classes of languages accepted within sublogarithmic space depend heavily on the machine models and the mode of space complexity' [16].

While at first sight, the step from Turing Machines to Offline Turing Machines appears to be just a technicality, it is a step from unidirectional to bidirectional computation. For example, while standard Turing Machines are composed just by running them one after the other, composition of Offline Turing Machines involves bidirectional data flow. This composition is implemented as a dialogue between the machines: one starts by requesting an output character from the second machine and every time this machine queries a character of its input, the first machine is started to compute this character.

Bidirectional computation is thus an integral feature of space-bounded computation, which must be accounted for in a programming language for space-bounded functions. We argue that a good way of accounting for this bidirectionality is to study space-bounded computation in terms of the Int construction of Joyal, Street & Verity [7, 6]. The Int construction is a general algebraic method of constructing a bidirectional universe from a unidirectional one. It appears in categorical formulations of the Geometry of Interaction [1, 2] and has many applications, e.g. to Attribute Grammars [8].

2.1 Structuring Space Bounded Computation

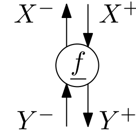
In this section we observe that the step from Turing Machines to Offline Turing Machines can be understood in terms of the Int construction. This simple observation gives us guidance for structuring space-bounded computation, since it allows us to draw on existing work on the structure obtained by the Int construction.

The idea is to start from a computational model, e.g. the partial computable functions as formalised by Turing Machines, and then apply the Int construction to this model. The result is a model that still contains the original one, but in addition also captures bidirectional (or interactive) computation in the style of Offline Turing Machines.

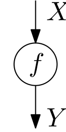
In general, the Int construction starts with a traced monoidal category $\mathbf{B}$ and yields a category $\mathrm{Int}(\mathbf{B})$ that represents bidirectional computation in $\mathbf{B}$. For the argument in this paper, it suffices to describe just one particular instance of $\mathrm{Int}(\mathbf{B})$, that where $\mathbf{B}$ is the category $\mathbf{Pfn}$ of sets and partial functions. In this example we drop computability for the sake of simplicity and assume that our 'computational' model consists just of partial functions between arbitrary sets.

If we apply the Int construction to $\mathbf{Pfn}$ with respect to tagged disjoint union as monoidal structure, then we obtain the following category $\mathrm{Int}(\mathbf{Pfn})$. Its objects are pairs

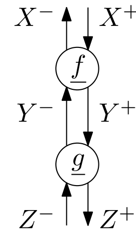
(X^-, X^+) of sets X^- and X^+. A morphism f from $X = (X^-, X^+)$ to $Y = (Y^-, Y^+)$ is a partial function $\underline{f}\colon X^+ + Y^- \to Y^+ + X^-$.



Morphisms capture bidirectional computation. Let us think of the partial function $\underline{f}$ as a message-passing node with two input wires and two output wires as drawn on the right. When an input value arrives on an input wire then the function $\underline{f}$ is applied to it and the resulting value is passed along the corresponding output wire.



We will combine the two edges for X^- and X^+ into a single edge in which messages may be passed both ways (and likewise for Y). Thus we obtain the node on the right, whose edges are bidirectional in the sense that an edge with label X allows any message from X^+ to be passed in the forward direction and any message from X^- to be passed in the backwards direction.



Composition in $\mathrm{Int}(\mathbf{Pfn})$ allows one to build message passing networks out of such nodes. The composition $g \circ f\colon X \to Z$ of $f\colon X \to Y$ and $g\colon Y \to Z$ is obtained simply by connecting the two nodes. The underlying partial function $\underline{g \circ f}\colon X^+ + Z^- \to Z^+ + X^-$ is most easily described in terms of message passing. An input in X^+ is given to $\underline{f}$ and one in Z^- to $\underline{g}$. If either $\underline{f}$ or $\underline{g}$ give an output in X^- or Z^+ then this is the output of $\underline{g \circ f}$. If, however, $\underline{f}$ (resp. $\underline{g}$) outputs a message on Y^+ (resp. Y^-), this message is given as an input to $\underline{g}$ (resp. $\underline{f}$). This may lead to a looping computation and $\underline{g \circ f}$ may be partial even if $\underline{g}$ and $\underline{f}$ are both total.

Offline Turing Machines appear as morphisms of type $(\mathit{State} \times \mathbb{N}, \mathit{State} \times \Sigma) \to (\mathbb{N}, \Sigma)$ in $\mathrm{Int}(\mathbf{Pfn})$. This follows immediately from the definition of morphisms and the discussion in Sec. 2. What we gain from viewing OTMs as morphisms in $\mathrm{Int}(\mathbf{Pfn})$ is that we can use the well-known structure of this category for constructing and manipulating them. For example, $\mathrm{Int}(\mathbf{Pfn})$ is compact closed and therefore allows us to use linear lambda calculus and higher-order functions for the manipulation of OTMs.

We list the structure in $\mathrm{Int}(\mathbf{Pfn})$ that we use in the definition of INTML.

Partial Functions. First we note that inside $\mathrm{Int}(\mathbf{Pfn})$ we still find $\mathbf{Pfn}$. For any set A we have an object $\mathcal{I}A = (\emptyset, A)$. A morphism from $\mathcal{I}A$ to $\mathcal{I}B$ is a partial function of type $A + \emptyset \to \emptyset + B$, so that the morphisms of that type are in one-to-one correspondence with the partial functions from A to B.

Thunks. Also useful is the object $[A] = (\{*\}, A)$, where a singleton replaces the empty set. We will use $*$ as question that signals an explicit request for a value of type A. Thus, one may think of $[A]$ as a type of thunks that are evaluated on demand.

Higher-order Functions. $\mathrm{Int}(\mathbf{Pfn})$ has a monoidal structure $\otimes$ that on objects is defined by $X \otimes Y = (X^- + Y^-, X^+ + Y^+)$. In addition, there is a dualising operation $(-)^*$ that exchanges question and answer sets, i.e. $(X^-, X^+)^* = (X^+, X^-)$. Together, $\otimes$ and $(-)^*$ make $\mathrm{Int}(\mathbf{Pfn})$ a compact closed category. As a consequence, we obtain a linear function space by letting $X \multimap Y = X^* \otimes Y$.

Indexed Tensor. Of further use is an indexed tensor product $\bigotimes_A X$. The object $\bigotimes_A X$ is isomorphic to $X \otimes \cdots \otimes X$ ($|A|$ times). It is defined by $(\bigotimes_A X)^- = A \times X^-$ and $(\bigotimes_A X)^+ = A \times X^+$. The first component in the messages indicates which component of the tensor product we are communicating with.

To see how this structure is useful for working with OTMs, consider a morphism of type

$$\bigotimes_{\mathit{State}} (\mathcal{I}\mathbb{N} \multimap \mathcal{I}\Sigma) \longrightarrow (\mathcal{I}\mathbb{N} \multimap \mathcal{I}\Sigma) \tag{1}$$

in $\mathrm{Int}(\mathbf{Pfn})$. By definition, it is a partial function from $(\mathit{State} \times (\emptyset + \Sigma)) + (\mathbb{N} + \emptyset)$ to $(\mathit{State} \times (\mathbb{N} + \emptyset)) + (\emptyset + \Sigma)$, which, if we remove the superfluous empty sets, is the same as the type of an Offline Turing Machine, as described in Sec. 2.

As a morphism of type (1), an Offline Turing Machine is modelled simply as a map from inputs to outputs, which are both modelled as (linear) functions $\mathcal{I}\mathbb{N} \multimap \mathcal{I}\Sigma$. This encoding of words as functions reflects the fact that Offline Turing Machines do not have access to the whole input at once, but rather can read only a single character at a time. Reading the n-th character just corresponds to applying the input function of type $\mathcal{I}\mathbb{N} \multimap \mathcal{I}\Sigma$ to the natural number n. In $\mathrm{Int}(\mathbf{Pfn})$ we may therefore use the input as if it were a single function from natural numbers to characters, even though in reality the input can only be queried character by character. We argue that this is more convenient than the access to the input by means of explicit questions as in the direct implementation of Offline Turing Machines. For example we can use λ-calculus to manipulate the input functions.

3 A Functional Language for Logarithmic Space

In the rest of this paper we develop a functional programming language INTML that is based on understanding space-bounded computation in terms of the Int construction. For the definition of INTML we start with a standard functional programming language. In order to program sublinear space algorithms in this language, we would like to implement message passing networks in it and we would like to manipulate these networks using the structure that we have found in the Int construction. We obtain INTML by extending the original programming language with primitives for constructing and manipulating such message passing networks.

INTML provides a syntax for the structure that one obtains from applying the Int construction to a *term model* of the standard functional programming that we start with. In terms of the outline above, one should replace all the partial functions in $\mathbf{Pfn}$ with terms in the functional programming language. That means that the message passing networks captured by the Int construction are now not just partial functions but partial functions implemented in the functional programming language.

INTML is thus a typed functional programming language with two classes of terms and types: one for the functional language that we start with and one for the structure that we obtain by applying the Int construction to a term model of this language. We call the former the *working class* and the latter the *upper class*. The upper class part can be seen as a definitional extension of the working class part. Upper class terms do not compute themselves, but rather represent message passing networks that are implemented by working class terms.

In this paper, we choose for the working class part of INTML a simple first order functional language with finite types. We have chosen such a simple language because the main novelty of INTML is the upper class calculus. We stress that the working class calculus may be replaced by a more expressive language, like PCF for example.

The upper class provides constructs for space-bounded programming. It is a linear type system inspired by Dual Light Affine Logic (DLAL) [4]. INTML treats the indexed tensor $\bigotimes_A$ much like exponential modality ! is treated in DLAL. Just as the exponential modality may only appear in negative positions in DLAL, i.e. one can have $!X \multimap Y$ but not $X \multimap !Y$, INTML only accounts for types of the form $\bigotimes_A X \multimap Y$. In the syntax we write $A \cdot X \multimap Y$ for them. The restriction to negative occurrences of $\bigotimes_A$ simplifies the type system without being too limiting in applications. In fact, we do not know of an example where $\bigotimes_A$ in a positive position would be useful.

3.1 Type System

Working class types are first-order types with type variables. They may appear in the upper class types, which represent structure obtained from the Int construction.

$$\begin{array}{ll} \text{Working class} & A, B ::= \alpha \mid 1 \mid A \times B \mid A + B \\ \text{Upper class} & X, Y ::= [A] \mid X \otimes Y \mid A \cdot X \multimap Y \end{array}$$

Instead of $1 \cdot X \multimap Y$ we write just $X \multimap Y$. For working class types we define coherent type isomorphism to be the least congruence generated by $1 \times A \cong A \times 1 \cong A$ and $A \times (B + C) \cong (A \times B) + (A \times C)$ and $(B + C) \times A \cong (B \times A) + (C \times A)$.

The terms of INTML are formed by the grammars below. We write c, d for working class variables and use f, g, h to range over working class terms. Upper class variables are ranged over by x, y and upper class terms by s, t. The terms $\mathsf{loop}(c.f)(g)$ and $\mathsf{hack}(c.f)$ bind the variable c in f.

$$\begin{array}{lrl} \text{Working class} & f, g, h ::= & c \mid \mathit{min}_A \mid \mathit{succ}_A(f) \mid \mathit{eq}_A(f, g) \\ & & \mid \mathsf{inl}(f) \mid \mathsf{inr}(f) \mid \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow g \mid \mathsf{inr}(d) \Rightarrow h \\ & & \mid * \mid \langle f, g \rangle \mid \mathsf{fst}(f) \mid \mathsf{snd}(f) \mid \mathsf{loop}(c.f)(g) \mid \mathsf{unbox}(t) \\ \text{Upper class} & s, t ::= & x \mid \langle t, t \rangle \mid \mathsf{let}\ s\ \mathsf{be}\ \langle x, y \rangle\ \mathsf{in}\ t \mid \lambda x.\, t \mid s\ t \\ & & \mid [f] \mid \mathsf{let}\ s\ \mathsf{be}\ [c]\ \mathsf{in}\ t \mid \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inr}(d) \Rightarrow t \\ & & \mid \mathsf{copy}\ t\ \mathsf{as}\ x, y\ \mathsf{in}\ t \mid \mathsf{hack}(c.f) \end{array}$$

The working class terms include the standard terms for 1, $\times$ and $+$. In addition there are constants min_A, succ_A and eq_A for any type A. These constants provide a total ordering and decidable equality on any type. For example, the values of the type $(1+1) \times (1+1)$ can be ordered as $\langle \mathit{inl}(*), \mathit{inl}(*) \rangle$, $\langle \mathit{inr}(*), \mathit{inl}(*) \rangle$, $\langle \mathit{inl}(*), \mathit{inr}(*) \rangle$, $\langle \mathit{inr}(*), \mathit{inr}(*) \rangle$ and with *min* and *succ* we can access such an ordering generically for any type without having to define it by hand. Finally, there is a term $\mathsf{loop}(c.f)$ for iteration. It is a simple syntax for a trace operator with respect to $+$. The intended operational semantics of loop is $\mathsf{loop}(c.f)(\mathsf{inr}(v)) \longrightarrow \mathsf{loop}(c.f)(f[\mathsf{inr}(v)/c])$ and $\mathsf{loop}(c.f)(\mathsf{inl}(v)) \longrightarrow v$, where in both cases v has already been reduced to a value (see Section 4.1).

The typing rules for working class terms appear in Fig. 1. They derive working class sequents of the form $\Sigma \vdash f\colon A$ asserting that f has type A in context Σ. The context Σ assigns working class types to a finite number of working class variables. As usual, the comma in contexts corresponds to $\times$, even though loop corresponds to a trace with respect to $+$. The typing rules should be unsurprising, except perhaps that for unbox. One may think of a term $t{:}[A]$ as a thunk that can be evaluated with unbox.

$$\frac{}{\Sigma,\, c{:}A \vdash c\colon A} \qquad \frac{}{\Sigma \vdash \mathit{min}_A\colon A} \qquad \frac{\Sigma \vdash f\colon A}{\Sigma \vdash \mathit{succ}_A(f)\colon A} \qquad \frac{\Sigma \vdash f\colon A \qquad \Sigma \vdash g\colon A}{\Sigma \vdash \mathit{eq}_A(f,g)\colon 1+1}$$

$$\frac{}{\Sigma \vdash *\colon 1} \qquad \frac{\Sigma \vdash f\colon A \qquad \Sigma \vdash g\colon B}{\Sigma \vdash \langle f, g\rangle\colon A \times B} \qquad \frac{\Sigma \vdash f\colon A \times B}{\Sigma \vdash \mathsf{fst}(f)\colon A} \qquad \frac{\Sigma \vdash f\colon A \times B}{\Sigma \vdash \mathsf{snd}(g)\colon B}$$

$$\frac{\Sigma \vdash f\colon A}{\Sigma \vdash \mathsf{inl}(f)\colon A+B} \qquad \frac{\Sigma \vdash f\colon A+B \qquad \Sigma,\, c{:}A \vdash g\colon C \qquad \Sigma,\, d{:}B \vdash h\colon C}{\Sigma \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow g \mid \mathsf{inr}(d) \Rightarrow h\colon C}$$

$$\frac{\Sigma \vdash f\colon B}{\Sigma \vdash \mathsf{inr}(f)\colon A+B} \qquad \frac{\Sigma,\, c{:}A+B \vdash f\colon A+B \qquad \Sigma \vdash g\colon A+B}{\Sigma \vdash \mathsf{loop}(c.\, f)(g)\colon A} \qquad \frac{\Sigma \mid\ \vdash t\colon [A]}{\Sigma \vdash \mathsf{unbox}(t)\colon A}$$

Fig. 1. Working Class Typing Rules

Upper class terms denote message passing networks that will be implemented by working class terms. An upper class typing sequent has the form $\Sigma \mid \Gamma \vdash t\colon X$, where Σ is a working class context. The context Γ is a finite list of declarations of the form $x_1 : A_1{\cdot}X_1, \ldots, x_k : A_k{\cdot}X_k$. As usual, we assume that no variable is declared more than once in Γ. The term t denotes a network with a single (bidirectional) output edge of type X and (bidirectional) input edges of types $\bigotimes_{A_1} X_1, \ldots, \bigotimes_{A_k} X_k$. Informally, one may think of a declaration $x : A{\cdot}X$ as a declaration of A-many copies of a value in X, i.e. one copy for each value $v{:}A$. Having multiple copies is useful because our message passing networks are stateless. When we send a query to X and later receive an answer, then we may not know what to do with that answer, since we have forgotten all that was computed earlier. However, we can use $A \cdot X$ instead of X in order to remember a value of type A. If we want to remember a value $v{:}A$ then we simply query the v-th copy of X.

For any upper class context Γ and any working class type A, we define an upper class context $A \cdot \Gamma$ by $A \cdot \langle\rangle = \langle\rangle$ and $A \cdot (\Delta,\, x : B{\cdot}X) = (A \cdot \Delta),\, x : (A \times B){\cdot}X$.

The upper class typing rules appear in Fig. 2. While upper class terms denote message passing networks, at a first reading they may be understood without knowing precisely the networks they denote. In particular, the reader may wish to look at the reduction rules in Fig. 5, which are soundly implemented by the translation of terms to message passing networks. In Sec. 4 we describe which networks the terms denote.

The upper class rules represent a choice of the structure that the Int construction adds to the working class calculus. It does not capture this rich structure completely, however. Therefore we add the 'hacking' rule below that allows one to implement message passing nodes directly, much like one can use inline assembler in C.

$$(\textsc{Hack})\ \frac{\Sigma,\, c{:}X^- \vdash f\colon X^+}{\Sigma \mid \Gamma \vdash \mathsf{hack}(c.f)\colon X}$$

In this rule, X^- and X^+ denote the negative and positive parts of X defined by:

$$\begin{aligned} [A]^- &= 1 & [A]^+ &= A \\ (X \otimes Y)^- &= X^- + Y^- & (X \otimes Y)^+ &= X^+ + Y^+ \\ (A \cdot X \multimap Y)^- &= A \times X^+ + Y^- & (A \cdot X \multimap Y)^+ &= A \times X^- + Y^+ \end{aligned}$$

$$(\textsc{Weak})\ \frac{\Sigma \mid \Gamma \vdash s\colon Y}{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash s\colon Y} \qquad (\textsc{Exch})\ \frac{\Sigma \mid \Gamma,\, x : A{\cdot}X,\, y : B{\cdot}Y,\, \Delta \vdash s\colon Z}{\Sigma \mid \Gamma,\, y : B{\cdot}Y,\, x : A{\cdot}X,\, \Delta \vdash s\colon Z}$$

$$(\textsc{LWeak})\ \frac{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash s\colon Y}{\Sigma \mid \Gamma,\, x : (B \times A){\cdot}X \vdash s\colon Y} \qquad (\textsc{Congr})\ \frac{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash s\colon Y}{\Sigma \mid \Gamma,\, x : B{\cdot}X \vdash s\colon Y}\ A \cong B$$

$$(\textsc{Var})\ \frac{}{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash x\colon X} \qquad (\otimes\mathrm{I})\ \frac{\Sigma \mid \Gamma \vdash s\colon X \qquad \Sigma \mid \Delta \vdash t\colon Y}{\Sigma \mid \Gamma,\, \Delta \vdash \langle s, t\rangle\colon X \otimes Y}$$

$$(\otimes\mathrm{E})\ \frac{\Sigma \mid \Gamma \vdash s\colon X \otimes Y \qquad \Sigma \mid \Delta,\, x : A{\cdot}X,\, y : A{\cdot}Y \vdash t\colon Z}{\Sigma \mid \Delta,\, A \cdot \Gamma \vdash \mathsf{let}\ s\ \mathsf{be}\ \langle x, y\rangle\ \mathsf{in}\ t\colon Z}$$

$$(\multimap\mathrm{I})\ \frac{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash s\colon Y}{\Sigma \mid \Gamma \vdash \lambda x.\, s\colon A \cdot X \multimap Y} \qquad (\multimap\mathrm{E})\ \frac{\Sigma \mid \Gamma \vdash s\colon A \cdot X \multimap Y \qquad \Sigma \mid \Delta \vdash t\colon X}{\Sigma \mid \Gamma,\, A \cdot \Delta \vdash s\, t\colon Y}$$

$$(\textsc{Contr})\ \frac{\Sigma \mid \Gamma \vdash s\colon X \qquad \Sigma \mid \Delta,\, x : A{\cdot}X,\, y : B{\cdot}X \vdash t\colon Y}{\Sigma \mid \Delta,\, (A + B) \cdot \Gamma \vdash \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t\colon Y}$$

$$(\textsc{Case})\ \frac{\Sigma \vdash f\colon A + B \qquad \Sigma,\, c{:}A \mid \Gamma \vdash s\colon X \qquad \Sigma,\, d{:}B \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inr}(d) \Rightarrow t\colon X}$$

$$([\,]\mathrm{I})\ \frac{\Sigma \vdash f\colon A}{\Sigma \mid \Gamma \vdash [f]\colon [A]} \qquad ([\,]\mathrm{E})\ \frac{\Sigma \mid \Gamma \vdash s\colon [A] \qquad \Sigma,\, c{:}A \mid \Delta \vdash t\colon [B]}{\Sigma \mid \Gamma,\, A \cdot \Delta \vdash \mathsf{let}\ s\ \mathsf{be}\ [c]\ \mathsf{in}\ t\colon [B]}$$

Fig. 2. Upper Class Typing Rules

Examples. We give an example derivation to illustrate that while the upper class type system is linear, working-class variables can be copied arbitrarily:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\mid x : 1{\cdot}[\alpha] \vdash x\colon [\alpha]} \qquad \cfrac{\cfrac{\vdots}{c{:}\alpha \mid f : 1{\cdot}([\alpha] \multimap [\alpha] \multimap [\beta]) \vdash f\,[c]\colon [\alpha] \multimap [\beta]} \qquad \cfrac{\cfrac{}{c{:}\alpha \vdash c\colon \alpha}}{c{:}\alpha \mid\ \vdash [c]\colon [\alpha]}}{c{:}\alpha \mid f : 1{\cdot}([\alpha] \multimap [\alpha] \multimap [\beta]) \vdash f\,[c]\,[c]\colon [\beta]}}{f : (\alpha \times 1){\cdot}([\alpha] \multimap [\alpha] \multimap [\beta]),\, x : 1{\cdot}[\alpha] \vdash \mathsf{let}\ x\ \mathsf{be}\ [c]\ \mathsf{in}\ f\,[c]\,[c]\colon [\beta]}\ ([\,]\mathrm{E})}{f : \alpha{\cdot}([\alpha] \multimap [\alpha] \multimap [\beta]) \vdash \lambda x.\, \mathsf{let}\ x\ \mathsf{be}\ [c]\ \mathsf{in}\ f\,[c]\,[c]\colon [\alpha] \multimap [\beta]}\ (\textsc{Congr}), (\multimap\mathrm{I})}{\vdash \lambda f.\, \lambda x.\, \mathsf{let}\ x\ \mathsf{be}\ [c]\ \mathsf{in}\ f\,[c]\,[c]\colon\ \alpha \cdot ([\alpha] \multimap [\alpha] \multimap [\beta]) \multimap [\alpha] \multimap [\beta]}\ (\multimap\mathrm{I})$$

Even though upper class terms can be understood as if they were implemented by the reduction rules in Fig. 5, it is important to understand that they will be (in Sec. 4) compiled down to (large) working class terms that implement certain message passing networks. The network for the upper class term in the conclusion of the above derivation, for example, represents a message passing network with a single (bidirectional) output wire. It behaves as follows: if it receives a request for the value of the result in $[\beta]$, it first requests the value of x. Upon receipt of this value, the network will then ask the function f for its return value. Since f has a type of the form $\alpha \cdot X$, the network has access to α-many copies of f. It chooses the copy indexed by the value of x, so that, even though the network is stateless, the value of x will be available once an answer

from f arrives. If f answers with a value in $[\beta]$, then this answer is forwarded as the final answer of the whole network. If f answers with a request for one of its arguments, then the network gives the value of x as reply to f.

That being able to copy working class variables does not make copy superfluous can be seen in the following terms for conversion between $[\alpha \times \beta]$ and $[\alpha] \otimes [\beta]$.

$$\lambda y.\ \mathsf{copy}\ y\ \mathsf{as}\ y_1, y_2\ \mathsf{in}\ \langle \mathsf{let}\ y_1\ \mathsf{be}\ [c]\ \mathsf{in}\ [\mathsf{fst}(c)], \mathsf{let}\ y_2\ \mathsf{be}\ [c]\ \mathsf{in}\ [\mathsf{snd}(c)]\rangle \ : (\gamma + \delta) \cdot [\alpha \times \beta] \multimap [\alpha] \otimes [\beta]$$

$$\lambda z.\ \mathsf{let}\ z\ \mathsf{be}\ \langle x, y\rangle\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ [c]\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ [d]\ \mathsf{in}\ [\langle c, d\rangle] : \ \alpha \cdot ([\alpha] \otimes [\beta]) \multimap [\alpha \times \beta]$$

Useful Combinators. The upper class calculus is a simple linear lambda calculus for constructing message passing networks. It appears to be missing many constructs, such as loops, that are required to make it an expressive programming language. Such constructs can be defined as higher-order combinators using hack.

The most important example of such a combinator is a loop iterator that informally satisfies `loop` $f\ v = w$ if $f\ v = \mathsf{inr}(w)$ and `loop` $f\ v =$ `loop` $f\ w$ if $f\ v = \mathsf{inl}(w)$.

$$\mathtt{loop}\colon \alpha \cdot (\gamma \cdot [\alpha] \multimap [\alpha + \beta]) \multimap [\alpha] \multimap [\beta]$$

Before we define `loop`, we give a typical example for its use. We define fold_α, such that $\mathit{fold}_\alpha\ f\ y$ computes $f\ x_n\ (\ldots (f\ x_1\ (f\ x_0\ y)))$, where $x_0 = \mathit{min}_\alpha$ and $x_{i+1} = \mathit{succ}_\alpha(x_i)$ and x_n is the maximum element of α, i.e. the element with $x_n = \mathit{succ}_\alpha(x_n)$.

$$\mathit{fold}_\alpha : (\alpha \times \beta \times \alpha \times \beta) \cdot ([\alpha] \multimap [\beta] \multimap [\beta]) \multimap [\beta] \multimap [\beta]$$

$$\begin{aligned}\mathit{fold}_\alpha = \quad & \lambda f.\ \lambda y.\ \mathtt{loop}\ (\lambda w.\ \mathsf{let}\ w\ \mathsf{be}\ [e]\ \mathsf{in}\ \mathsf{let}\ f\ [\mathsf{fst}(e)]\ [\mathsf{snd}(e)]\ \mathsf{be}\ [z]\ \mathsf{in} \\ & \qquad \mathsf{case}\ \mathit{eq}_\alpha(\mathsf{fst}(e), \mathit{succ}_\alpha(\mathsf{fst}(e)))\ \mathsf{of}\ \mathsf{inl}(\mathit{true}) \Rightarrow [\mathsf{inr}(z)] \\ & \qquad\qquad \mid \mathsf{inr}(\mathit{false}) \Rightarrow [\mathsf{inl}(\langle \mathit{succ}_\alpha(\mathsf{fst}(e)), z\rangle)]) \\ & \quad (\mathsf{let}\ y\ \mathsf{be}\ [z]\ \mathsf{in}\ [\langle \mathit{min}_\alpha, z\rangle])\end{aligned}$$

The definition of `loop` uses hack and therefore makes explicit reference to the translation of upper class terms to message passing networks, which we describe in detail in Sec 4. The `loop`-combinator is defined by `loop` $= \mathsf{hack}(c.l)$, where l is a working class term of type $c{:}\alpha \times (\gamma \times 1 + (\alpha + \beta)) + (\alpha + 1) \vdash l\colon \alpha \times (\gamma \times \alpha + 1) + (1 + \beta)$ that implements the following mappings using a nested case expression: (i) $inr(inr(*)) \mapsto inl(inr(*))$; (ii) $inr(inl(a)) \mapsto inl(a, inr(*))$; (iii) $inl(a, inr(inr(b)) \mapsto inr(inr(b))$; (iv) $inl(a, inr(inl(a')) \mapsto inl(a', inr(*))$; and (v) $inl(a, inl(g, *)) \mapsto inl(a, inl(g, a))$. These assignments can be interpreted as follows: Mapping (i) says that when we get a request for the final value, we start by asking the base case for its value. When an answer from the base cases arrives, we put it in a memory cell (corresponding to $\alpha \cdot -$ in the type) and ask the step function for its result (ii). Whenever the step function asks for its argument, we supply the value from the memory cell (v). If the step function answers $inr(b)$, then we are done and give b as output (iii). If the step function answers $inl(a')$, then we overwrite the memory cell content with a' and restart the step function by asking for its result (iv).

A second useful combinator is a simple version of `callcc`. It can be given the following type for any upper class type X.

$$\mathtt{callcc}\colon \big(\gamma \cdot ([\alpha] \multimap X) \multimap [\alpha]\big) \multimap [\alpha]$$

This combinator is defined by $\texttt{callcc} = \mathsf{hack}(c.l)$, where l is a working-class term of type $c{:}(\gamma \times (\alpha + X^-) + \alpha) + 1 \vdash l\colon (\gamma \times (1 + X^+) + 1) + \alpha$ that implements the following mappings: (i) $inr(*) \mapsto inl(inr(*))$; (ii) $inl(inr(a)) \mapsto inr(a)$; (iii) $inl(inl(g, inr(x))) \mapsto inl(inl(g, inl(*)))$; and (iv) $inl(inl(g, inl(a))) \mapsto inr(a)$. These assignments implement `callcc` as follows: a request for the result becomes a request for the result of the argument function (i); When the argument function produces a result value, we forward it as the final result (ii). If the argument function ever uses its argument, i.e. calls the continuation, then the value passed to the continuation should be returned as the final result. This is done by assignments (iii) and (iv). Whenever the result of the continuation is requested, this request is turned into a request for the argument of the continuation (iii). Upon supply of the argument to the continuation, the computation is aborted and this argument is returned as the end result (iv).

3.2 Programming in INTML

We have introduced the upper class in INTML with the intention of helping the programmer to implement functions with sublinear space usage. Let us give a few examples of how we think the upper class features will be useful.

Consider for example binary words. For sublinear space computation, they are suitably modelled as functions of type $A \cdot [B] \multimap [3]$. With the constants *min* and *succ*, we can regard B as a type of numbers. We interpret 3 as a type containing characters '0', '1' and a blank symbol. Then, $A \cdot [B] \multimap [3]$ can represent words by functions that map the n-th element of B to the n-th character of the word (see Sec. 5 for a precise definition). Being a higher-order language, INTML allows the programmer both to define such words directly, but also to write higher-order combinators to manipulate them.

When working with $A \cdot X \multimap Y$, we have found that often we are not interested in the particular type A, only that some such type exists. Let us therefore in the following hide all such annotations and write just $X \to Y$ to mean $A \cdot X \multimap Y$ for some A.

Useful combinators for words encoded as functions are, e.g. $\texttt{zero}\colon ([\alpha] \to [3])$ for the empty word, $\texttt{succ}_0\colon ([\alpha] \to [3]) \to ([\alpha] \to [3])$ for appending the character 0 or $\texttt{if}\colon ([\alpha] \to [3]) \to ([\alpha] \to [3]) \to ([\alpha] \to [3]) \to ([\alpha] \to [3])$ for case distinction on the last character of a word. They allow one to work with words encoded as functions as if they were normal strings, even though these words do not even necessarily fit into memory. The combinators themselves can be implemented easily in INTML.

$$\begin{aligned} \texttt{succ}_0 := \lambda w.\, \lambda i.\ \mathsf{let}\ i\ \mathsf{be}\ [c]\ \mathsf{in}\ \mathsf{case}\ \mathit{eq}(c, \mathit{min})\ \mathsf{of}\ & \mathsf{inl}(\mathit{true}) \Rightarrow [\mathit{min}] \\ & \mid \mathsf{inr}(\mathit{false}) \Rightarrow w\ [\mathit{pred}\ c] \end{aligned}$$

Here *pred* denotes a working class predecessor term, which is easy to define.

$$\begin{aligned} \texttt{if} := \lambda w.\, \lambda w_0.\, \lambda w_1.\, \lambda i.\ & \mathsf{let}\ w\ [\mathit{min}]\ \mathsf{be}\ [c]\ \mathsf{in} \\ & \mathsf{case}\ c\ \mathsf{of}\ \mathsf{inl}(\mathit{blank}) \Rightarrow w_0\ i \\ & \qquad \mid \mathsf{inr}(z) \Rightarrow \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(\mathit{zero}) \Rightarrow w_0\ i \\ & \qquad\qquad\qquad\qquad\qquad \mid \mathsf{inr}(\mathit{one}) \Rightarrow w_1\ i \end{aligned}$$

These are simple examples, of course. We believe that nontrivial combinators can also be implemented. For example, Møller-Neergaard gives a LOGSPACE implementation of

safe recursion on notation by computational amnesia [14]. We believe that using `loop` and `callcc` a similar program can be implemented in INTML as a combinator `saferec` taking as arguments a base case $g\colon [\alpha] \to [3]$, two step functions $h_0, h_1\colon ([\alpha] \to [3]) \to ([\alpha] \to [3])$ and a word $w\colon [\alpha] \to [3]$ to recurse on and giving a word as output.

In this way the higher-order features of INTML can be used to abstract away details of a message-passing implementation of functions on words and to write LOGSPACE functions just like in $\mathrm{BC}_\varepsilon^-$ [14]. Moreover, INTML gives access to the implementation details, should the abstraction not be expressive enough. For instance, the proof of LOGSPACE completeness in Thm. 3 below goes by a straightforward encoding of a OTM. It is much simpler than the corresponding encoding in $\mathrm{BC}_\varepsilon^-$ [14] because INTML allows us to manipulate working class values directly.

The higher order approach also works for data types other than strings. For example, graphs can be represented by a type of the form $([\alpha] \to [2]) \otimes ([\alpha \times \alpha] \to [2])$, where the first component is a predicate that indicates which elements of α count as graph nodes and the second component is the edge relation.

4 Evaluation

In this section we present the evaluation mechanism for INTML. Evaluation of upper class terms is closely related to evaluation in [15], but also to Mackie's Interaction Abstract Machine [11] and to read-back from optimal reduction [12, 3]. Indeed, in his 1995 paper [11] Mackie speculates that this form of evaluation could have applications where space usage is important. With INTML we present a calculus that makes space usage analysis possible. INTML differs from the work in loc. cit. in that one can mix working class and upper class terms. Previously only the upper class part was considered.

4.1 Reduction of Working-Class Terms

The evaluation of INTML programs is done by reduction of working class terms. Before evaluation of an INTML program, all upper class terms are compiled into working class terms. Since, in particular, any occurrence of unbox will be removed, it suffices to define reduction only for unbox-free working class terms.

Working class values are defined by:

$$v, w := c \mid * \mid \langle v, w\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v)$$

The reduction of working class terms is explained by a small step reduction relation $\longrightarrow$ between closed unbox-free terms. Closedness here means the absence of both term variables and type variables (which could appear in the type annotations of constants like min_A). The relation $\longrightarrow$ formalises standard *eager* reduction, see e.g. [17]. We omit standard reduction rules and just explain here how loop and the constants min_A and succ_A are treated. Loops are unfolded by the rules $\mathsf{loop}(c.f)(\mathsf{inr}(v)) \longrightarrow v$ and $\mathsf{loop}(c.f)(\mathsf{inl}(v)) \longrightarrow \mathsf{loop}(c.f)(f[\mathsf{inl}(v)/c])$, in both of which v must be a value. Constants are unfolded on demand, guided by their type annotation. The minimum elements of all closed types are defined by:

$$\mathit{min}_1 \longrightarrow * \qquad \mathit{min}_{A+B} \longrightarrow \mathsf{inl}(\mathit{min}_A) \qquad \mathit{min}_{A\times B} \longrightarrow \langle \mathit{min}_A, \mathit{min}_B\rangle$$

In the implementation of succ_A, we must be a little careful that reducts do not become too large. We implement *succ* using a new constant *succmin* that informally denotes a function $\mathit{succmin}_A \colon A \to A + A$ with the following meaning: $\mathit{succmin}_A(x) = \mathsf{inl}(y)$ means that y is the successor of x; $\mathit{succmin}_A(x) = \mathsf{inr}(y)$ means that x has no successor and y is the minimum element of A. We use the following rules for *succmin*.

$$
\begin{aligned}
\mathit{succmin}_1(*) &\longrightarrow \mathsf{inr}(*) \\
\mathit{succmin}_{A+B}(\mathsf{inl}(v)) &\longrightarrow \mathsf{case}\ \mathit{succmin}_A(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\mathsf{inl}(x)) \\
&\qquad\qquad \mid\ \mathsf{inr}(y) \Rightarrow \mathsf{inl}(\mathsf{inr}(\mathit{min}_B)) \\
\mathit{succmin}_{A+B}(\mathsf{inr}(v)) &\longrightarrow \mathsf{case}\ \mathit{succmin}_B(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\mathsf{inr}(x)) \\
&\qquad\qquad \mid\ \mathsf{inr}(y) \Rightarrow \mathsf{inr}(\mathsf{inl}(\mathit{min}_A)) \\
\mathit{succmin}_{A\times B}(\langle v, w\rangle) &\longrightarrow \mathsf{case}\ \mathit{succmin}_A(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\langle x, w\rangle) \\
&\qquad\qquad \mid\ \mathsf{inr}(x) \Rightarrow \mathsf{case}\ \mathit{succmin}_B(w)\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow \mathsf{inl}(\langle x, y\rangle) \\
&\qquad\qquad\qquad\qquad \mid\ \mathsf{inr}(y) \Rightarrow \mathsf{inr}(\langle x, y\rangle)
\end{aligned}
$$

We use these rules instead of the evident rules for *succ* because they are linear in v, w, A and B, which is important for the proof of Prop. 1 below.

Because of the simplicity of the working class calculus, it is possible to give useful upper bounds on how large a term can become during the course of eager reduction directly by induction on the term structure. Essentially, we can bound the size of values in terms of their types and use this to derive a bound on the potential size of a term under reduction by looking at its variables and their types. We state this result in the following proposition, in which $|g|$ and $|C|$ denote the size of the abstract syntax trees of g and C respectively.

Proposition 1. *If* $c{:}A \vdash f \colon B$ *is derivable then there are constants* n *and* m *such that* $(f[C/\alpha])[v/c] \longrightarrow^* g$ *implies* $|g| \le n + m \cdot |C|$ *for every closed type* C *and every closed value* v *of type* $A[C/\alpha]$.

We will use for C types of the form $2 \times \cdots \times 2$ (k times), where 2 denotes $1 + 1$. This type represents the numbers from 0 to $2^k - 1$ in binary and we have $|C| \in O(k)$. A unary encoding is also possible with $C = 1 + \cdots + 1$ (2^k times), but we have $|C| \ge 2^k$ and values can indeed become as large as this.

4.2 Reducing Upper Class to Working Class

Now we explain how closed upper class terms are compiled down to working class terms, so that they can be reduced with the relation $\longrightarrow$ from the previous section. The compilation works by interpreting upper class terms as message passing circuits as in Sec. 2 and then implementing these circuits by working class terms.

We start by defining the message passing circuits we use in the compilation. These circuits may be understood as a particular instance of string diagrams for monoidal categories [13]. The are also related to proof nets, see [13] for a discussion. Circuits are directed graphs that represent networks in which messages are passed along edges. A node labelling allows us to use nodes with different message passing behaviour. Edges

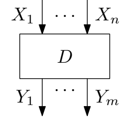
are labelled with types that tell which kind of messages can be passed along them. Edge labels are formed by the grammar below, in which A ranges over working class types.

$$X, Y ::= [A] \mid [A]^* \mid X \otimes Y \mid \bigotimes_A X$$

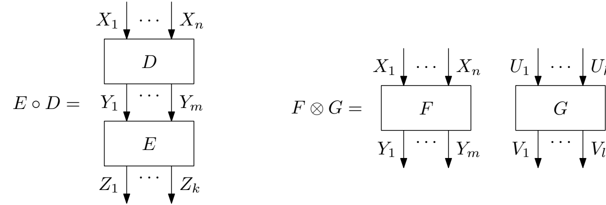
We define an operation $(-)^*$ on edge labels as follows. It maps $[A]$ to $[A]^*$ and $[A]^*$ to $[A]$ and is defined on compound expressions by $(X \otimes Y)^* = X^* \otimes Y^*$ and $(\bigotimes_A X)^* = \bigotimes_A X^*$. Note in particular that we have $X^{**} = X$ for any type label X.

Circuits are labelled directed graphs that are in addition equipped with a *two-way local ordering*. A two-way local ordering for a graph $G = (V, E)$ specifies for each node $v \in V$ a total ordering on both the set $(\{v\} \times V) \cap E$ of outgoing edges from v and on the set $(V \times \{v\}) \cap E$ of incoming edges to v. The need for a local ordering arises in the treatment of nodes such as $\otimes E$ below. This node has an incoming edge labelled with $X \otimes Y$ and two outgoing edges labelled with X and Y and we want to distinguish the two outgoing edges even if X and Y are the same.

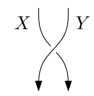
Furthermore, circuits have a number of input and output ports. We capture input and output edges by means of two distinguished nodes: a source and a sink. Edges from the distinguished source are input edges and edges to the sink are output edges. We write $D\colon (X_1, \dots, X_n) \to (Y_1, \dots, Y_m)$ for such a graph D with input edges of type $X_1, \dots, X_n$ and output edges of type $Y_1, \dots, Y_m$ (note that they are ordered because of the local ordering for source and sink). We usually draw D as shown on the right.



Given two graphs $D\colon (\boldsymbol{X}) \to (\boldsymbol{Y})$ and $E\colon (\boldsymbol{Y}) \to (\boldsymbol{Z})$, their sequential composition $E \circ D\colon (\boldsymbol{X}) \to (\boldsymbol{Z})$ is defined as depicted below: the i-th incoming edge to the sink of D and the i-th outgoing edge from the source of E are joined to a single edge. The sink of D and the source of E are removed. Furthermore, given $F\colon (\boldsymbol{X}) \to (\boldsymbol{Y})$ and $G\colon (\boldsymbol{U}) \to (\boldsymbol{V})$, we write $F \otimes G$ for the graph of type $(\boldsymbol{X}, \boldsymbol{U}) \to (\boldsymbol{Y}, \boldsymbol{V})$ obtained by putting F and G in parallel.



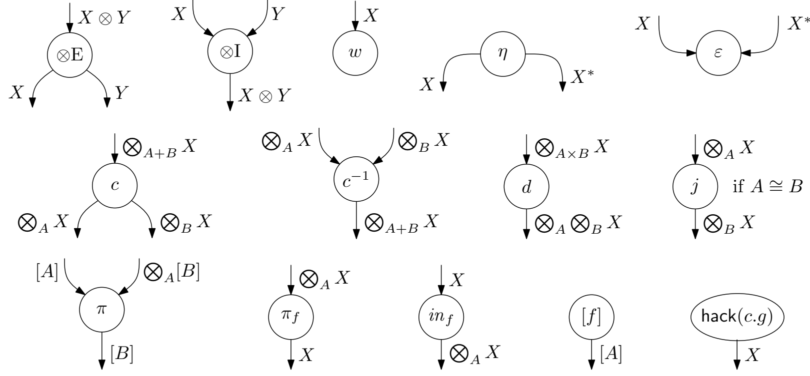
Notice that in a composition the local ordering on the edges labelled with $\boldsymbol{Y}$ disappears. For example, if we let $\mathit{swap}_{X,Y}\colon (X, Y) \to (Y, X)$ be the graph on the right, then $\mathit{swap}_{Y,X} \circ \mathit{swap}_{X,Y}$ is $id_X \otimes id_Y\colon (X, Y) \to (X, Y)$, where $id_X\colon (X) \to (X)$ is a single edge labelled X from input to output.



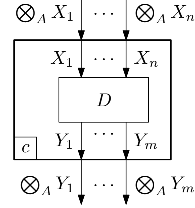
Definition 1 (Circuit on Σ). For any working class context Σ, we define the set of *circuits on Σ* to be the smallest set of two-way locally ordered graphs with input and output edges that satisfies the following conditions.

– For all X and Y both id_X and $\mathit{swap}_{X,Y}$ are are circuits on Σ.

– Each of the following single-node graphs is a circuit on Σ, where X and Y may be arbitrary type labels, A and B may be arbitrary working class types and f and g may be arbitrary terms with $\Sigma \vdash f\colon A$ and $\Sigma,\, c{:}X^- \vdash g\colon X^+$.



– If $D\colon (\boldsymbol{X}) \to (\boldsymbol{Y})$ and $E\colon (\boldsymbol{Y}) \to (\boldsymbol{Z})$ are circuits on Σ then so is $E \circ D$.
– If $D\colon (\boldsymbol{X}) \to (\boldsymbol{Y})$ and $E\colon (\boldsymbol{U}) \to (\boldsymbol{V})$ are circuits on Σ then so is $D \otimes E$.
– If $D\colon (X_1, \ldots, X_n) \to (Y_1, \ldots, Y_m)$ is a circuit on $\Sigma,\, c{:}A$ then the graph $\bigotimes_{c:A} D\colon\ (\bigotimes_A X_1, \ldots, \bigotimes_A X_n) \ \to\ (\bigotimes_A Y_1, \ldots, \bigotimes_A Y_m)$ constructed as follows is a circuit on Σ: each incoming edge of D is prepended with a node $\downarrow^c_{X_i}\colon (\bigotimes_A X_i) \to (X_i)$ and to each outgoing edge of D a node $\uparrow^c_{Y_i}\colon (Y_i) \to (\bigotimes_A Y_i)$ is appended. For better readability, we do not draw these nodes explicitly and draw a box around D instead, as depicted on the right.



Given a circuit D on $(\Sigma,\, c{:}A)$ and a term $\Sigma \vdash f : A$, we can form a circuit $D[f/c]$ on Σ by replacing each node π_g with $\pi_{g[f/c]}$, each node in_g with $in_{g[f/c]}$, each node $[g]$ with $[g[f/c]]$, and each node $\mathsf{hack}(d.g)$ with $\mathsf{hack}(d.g[f/c])$.

To each edge in a circuit D on Σ we assign a *level*, which is a working class context, by the following requirements. Input and output edges have level Σ. Any two edges incident to the same node have the same level, except if the node is $\downarrow^c_X\colon (\bigotimes_A X) \to (X)$ or $\uparrow^c_X\colon (X) \to (\bigotimes_A X)$. If the incoming edge of $\downarrow^c_X$ (resp. the outgoing edge of $\uparrow^c_X$) has level Σ' then the outgoing edge (resp. incoming edge) has level $\Sigma',\, c{:}A$.

Message Passing. Write $\mathcal{V}_A$ for the set of all closed working class values of type A. Write $\mathcal{E}_\Sigma$ for the set of Σ*-environments* consisting of all functions that map the variables in Σ to closed values of their declared types. The set $M_{\Sigma,X}$ of messages that can be passed along an edge labelled with X at level Σ is then defined by $M_{\Sigma,X} = \mathcal{E}_\Sigma \times (\mathcal{V}_{X^+} \times \{+\} \cup \mathcal{V}_{X^-} \times \{-\})$. A message $m \in M_{\Sigma,X}$ is either a *question* or an *answer* depending on whether its third component is '$-$' or '$+$'. Answers travel in the direction of the edge while questions travel in the opposite direction. Messages are essentially the same as the contexts in context semantics [12].

To define message passing for a circuit $D\colon (\boldsymbol{X}) \to (\boldsymbol{Y})$ on Σ, let the set M_D of messages on D consist of all pairs (e, m) of an edge e in D and a message $m \in$

$M_{\Sigma(e),X(e)}$, where $\Sigma(e)$ is the level of e and $X(e)$ is its label. Now we define how each node in D locally reacts to arriving messages. To this end we define for each node v a partial function φ_v by the assignments in Fig. 3. For example φ_π implements the behaviour of the term in ([ ]E), as sketched in Sec. 3.1. An initial question on edge o becomes a request for the value of type $[A]$ on edge i_1. Upon receipt of an answer $w{:}A$ from there, this answer is put into the memory cell provided by $\bigotimes_A$ and the value of type $[B]$ is queried on edge i_2. An answer to this request is then passed as final answer along edge o. In Fig. 3 we have omitted the cases for φ_ε and $\varphi_{c^{-1}}$, which are just like those for φ_η and φ_c. Notice that for different nodes v and w in D, the domains of φ_v and φ_w do not overlap.

The message passing behaviour of the whole circuit is then the partial function $\varphi_D \colon M_D \to M_D$ defined by repeatedly applying the local functions φ_v:

$$\varphi_D = Tr\left(\bigcup\nolimits_{v \in V(D)} \varphi_v\right) \qquad Tr(f)(m) = \begin{cases} Tr(f)(f(m)) & \text{if } f(m) \text{ defined} \\ m & \text{if } f(m) \text{ undefined} \end{cases}$$

It is not hard to see that messages cannot get stuck inside the circuit, i.e. $\varphi_D(e,m) = (e',m')$ implies that e' is an input or an output edge. In fact, we will forget about the internal structure of D and consider just the restriction of φ_D to messages on input or output edges of D. This restriction corresponds to a partial function Φ_D of type $\left(X_1^+ + \cdots + X_n^+ + Y_1^- + \cdots + Y_m^-\right) \times \mathcal{E}_\Sigma \longrightarrow Y_1^+ + \cdots + Y_m^+ + X_1^- + \cdots + X_n^-$. We call Φ_D *the behaviour of* D and consider circuits with the same behaviour to be equal. We write $D \sim E$ if D and E are circuits with the same interface and $\Phi_D = \Phi_E$.

Upper Class to Circuits. We now interpret upper class terms by circuits. To each derivation δ ending with sequent $\Sigma \mid x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n \vdash s \colon Y$ we assign $[\![\delta]\!]$, a circuit on Σ with one output wire of type Y and n input wires of type $(\bigotimes_{A_1} X_1, \ldots, \bigotimes_{A_n} X_n)$, where we identify $A \cdot X \multimap Y$ with $(\bigotimes_A X)^* \otimes Y$. That is, each variable declaration in Γ becomes an input wire of the translated circuit. Abusing notation slightly, we will write Γ also for the list of input types of this circuit.

The definition goes by induction on derivations and is given in the table in Fig. 4. In this table we denote the premises of δ by δ_s and δ_t depending on the term in the premise. We write just $\bigotimes_A$ for $\bigotimes_{c:A}$ if c does not appear anywhere. Given an upper class context Γ we write w_Γ for $w \otimes \cdots \otimes w \colon (\Gamma) \to ()$. Similarly, we write $id_\Gamma \colon (\Gamma) \to (\Gamma)$, $d_{A\cdot\Gamma} \colon (A \cdot \Gamma) \to (\bigotimes_A \Gamma)$, $(in_f)_\Gamma \colon (\Gamma) \to (\bigotimes_A \Gamma)$ and $c_\Gamma \colon (\Gamma) \to (\Gamma, \Gamma)$ for the analogous tensorings of id, d, in_f and c (the definition of c_Γ involves evident permutations of the outputs to arrive at the indicated type).

Implementing Message Passing. The compilation of upper class terms to message passing circuits almost completes the translation from upper class to working class. It just remains to implement message passing in the working class calculus.

That is, for any circuit D we construct a closed working class term that implements its behaviour Φ_D. In essence, the construction works in the same way as the definition of Φ_D above. First note that we can represent the set of messages $M_{\Sigma,X}$ by the working class type $(A_1 \times \cdots \times A_n) \times (X^- + X^+)$, where $\Sigma = x_1{:}A_1, \ldots, x_n{:}A_n$. With this we can represent M_D in the working class calculus in the form of a big sum type. Then,

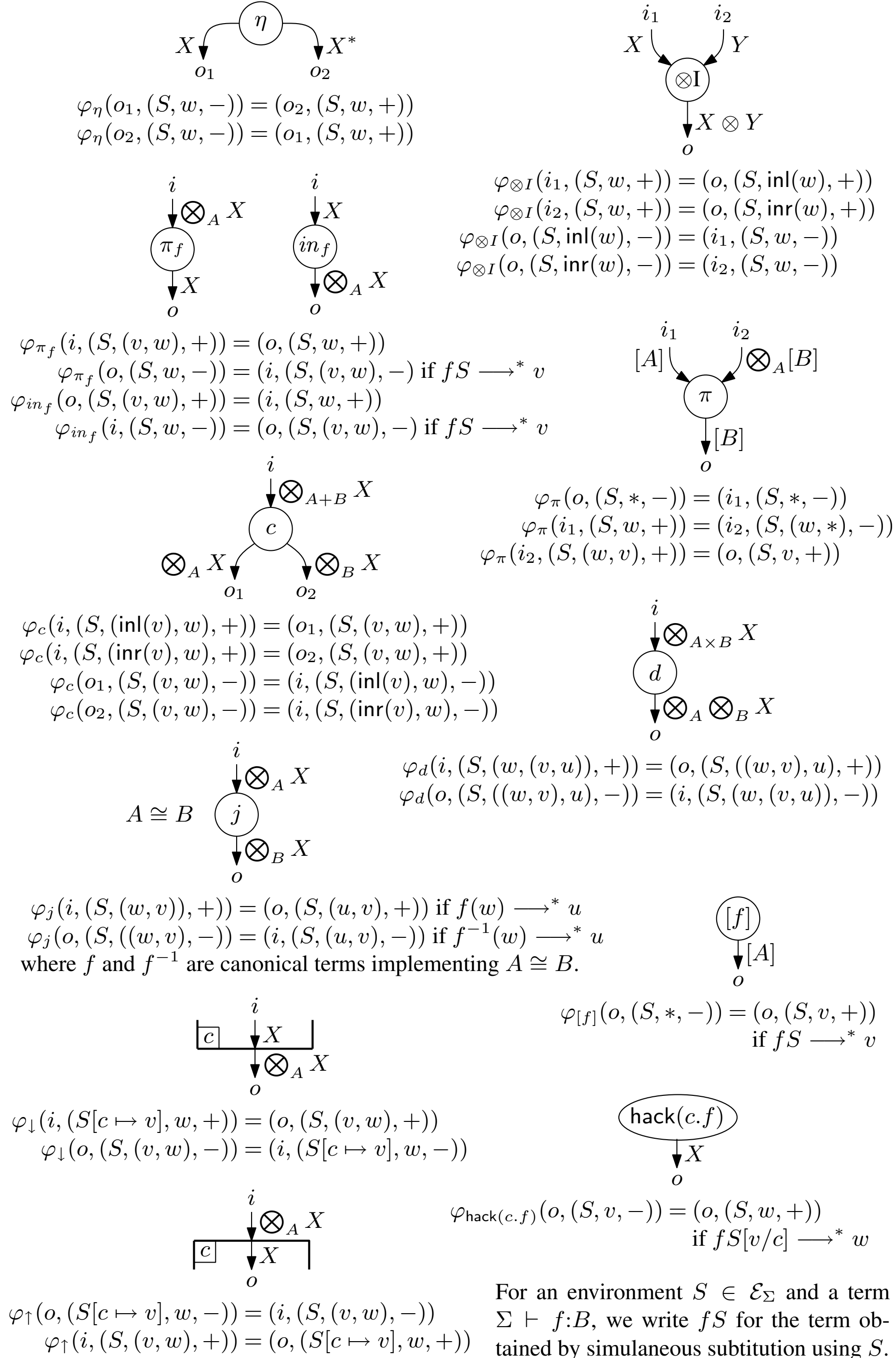


Fig. 3. Local message passing

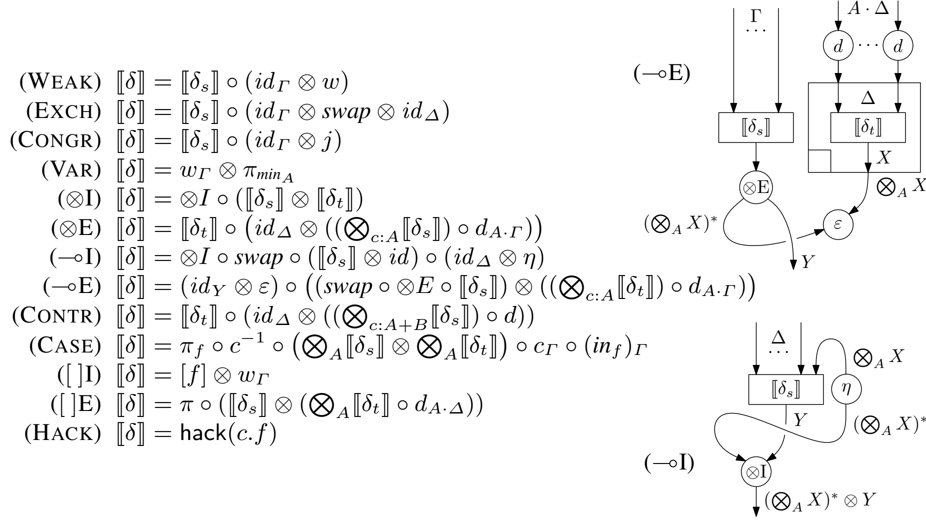


Fig. 4. Compilation to Circuits

$\bigcup_{v \in V(D)} \varphi_v \colon M_D \to M_D$ can be implemented easily by a big case distinction and it is easy to obtain φ_D from this using a single loop. From this, finally, we obtain Φ_D.

We have thus explained how each upper class term can be translated to a working class term. It remains to say how we deal with terms of the form $\Sigma \vdash \mathsf{unbox}(s) \colon A$ where $\Sigma \mid \vdash s \colon [A]$. Note that s is translated to a circuit $[\![\delta_s]\!]$ whose behaviour is a function of type $\Phi_{[\![\delta_s]\!]} \colon 1 \times \mathcal{E}_\Sigma \to A$. The working class term implementing this function is just what we need to interpret the term unbox.

4.3 Soundness

We have defined the evaluation of INTML by translation of upper class to working class, since this allows us to obtain sublinear space bounds (cf. Sec. 5). However, the translation is somewhat complicated and it is does not make it obvious just what it is that the upper class terms compute. In this section, we show that one may also consider the upper class as a functional programming language with standard reduction rules, which are then soundly implemented by the translation.

A notion of reduction can be formulated by the rules in Fig. 5. We include a rule for `loop` but note that we cannot give rules for hack in general. When we write $s \longrightarrow t$ we assume $\vdash s \colon X$ and $\vdash t \colon X$ for some X. We close these rules under evaluation contexts.

$$E, F ::= [\cdot] \mid \langle E, t\rangle \mid \langle t, E\rangle \mid \mathsf{let}\ E\ \mathsf{be}\ \langle x, y\rangle\ \mathsf{in}\ t \mid$$
$$E\ t \mid t\ E \mid \mathsf{let}\ E\ \mathsf{be}\ [c]\ \mathsf{in}\ t \mid \mathsf{copy}\ E\ \mathsf{as}\ x, y\ \mathsf{in}\ t$$

As usual, $E[s]$ is the term obtained by substituting s for the only occurrence of $[\cdot]$ in E.

The translation to working class terms is invariant under reduction:

Theorem 1 (Soundness). *If δ derives $\vdash s \colon X$ and $s \longrightarrow t$, then there exists a derivation ρ of $\vdash t \colon X$ that satisfies $[\![\delta]\!] \sim [\![\rho]\!]$.*

$$
\begin{aligned}
(\lambda x.s)t &\longrightarrow s[t/x] \\
\mathsf{let}\ \langle s,t\rangle\ \mathsf{be}\ \langle x,y\rangle\ \mathsf{in}\ u &\longrightarrow u[s/x][t/y] \\
\mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inr}(d) \Rightarrow t &\longrightarrow s[v/c] \quad \text{if } v \text{ is a value} \\
\mathsf{case}\ \mathsf{inr}(v)\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inr}(d) \Rightarrow t &\longrightarrow t[v/d] \quad \text{if } v \text{ is a value} \\
\mathsf{let}\ [v]\ \mathsf{be}\ [c]\ \mathsf{in}\ t &\longrightarrow t[v/c] \quad \text{if } v \text{ is a value} \\
\mathsf{copy}\ s\ \mathsf{as}\ x,y\ \mathsf{in}\ t &\longrightarrow t[s/x][s/y] \\
[f] &\longrightarrow [g] \quad \text{if } f \longrightarrow g \\
\mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inr}(d) \Rightarrow t &\longrightarrow \mathsf{case}\ g\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inr}(d) \Rightarrow t \quad \text{if } f \longrightarrow g \\
\mathsf{loop}\ s\ t &\longrightarrow \mathsf{let}\ t\ \mathsf{be}\ [c]\ \mathsf{in}\ \mathsf{let}\ s\ [c]\ \mathsf{be}\ [d]\ \mathsf{in} \\
&\qquad \mathsf{case}\ d\ \mathsf{of}\ \mathsf{inl}(d) \Rightarrow \mathsf{loop}\ s\ [d] \mid \mathsf{inr}(d) \Rightarrow [d]
\end{aligned}
$$

Fig. 5. Upper Class Reduction

For the proof we need substitution lemmas, which are proved by induction on ρ.

Lemma 1. *If $v \in \mathcal{V}_A$ and ρ derives $\Sigma, c{:}A \mid \Gamma \vdash t\colon X$, then there is a derivation $\rho[v/c]$ of $\Sigma \mid \Gamma \vdash t[v/c]\colon X$ such that $[\![\rho[v/c]]\!] \sim [\![\rho]\!][v/c]$ holds.*

Lemma 2. *If δ derives $\Sigma \mid \ \vdash s\colon X$ and ρ derives $\Sigma \mid \Gamma, x : A{\cdot}X, \Delta \vdash t\colon Y$, then there exists a derivation $\rho[\delta/x]$ of $\Sigma \mid \Gamma, \Delta \vdash t[s/x]\colon Y$ that satisfies $[\![\rho[\delta/x]]\!] \sim [\![\rho]\!] \circ (id_\Gamma \otimes \bigotimes_A [\![\delta]\!] \otimes id_\Delta)$.*

Proof (of Theorem 1). For any reduction $s \longrightarrow t$ there exist decompositions $s = E[s']$ and $t = E[t']$ such that $s' \longrightarrow t'$ is an instance of one of the reductions in Fig 5.

The proof then goes by induction on the structure of E. The base case where E is empty, amounts to showing the assertion for the basic reductions. For lack of space, we just spell out the first case where $s \longrightarrow t$ has the form $(\lambda x.\, q)\ p \longrightarrow q[p/x]$.

Since δ ends in a sequent with empty context, it cannot end with a structural rule. Hence, the last two rules in δ must be ($\multimap$E) after ($\multimap$I). Let σ and τ be the derivations of $x{:}B \cdot Z \vdash q\colon Y$ and $\vdash p\colon Z$ that derive the premises of these rules. Then we can use $\otimes E \circ \otimes I \sim id$ and $(id \otimes \varepsilon) \circ (\eta \otimes id) \sim id$, which are both easy to show directly, and the substitution lemma to calculate:

$$
\begin{aligned}
[\![\delta]\!] &= (id_Y \otimes \varepsilon) \circ ((\mathit{swap} \circ \otimes E \circ \otimes I \circ \mathit{swap} \circ ([\![\sigma]\!] \otimes id_{\bigotimes_B Z}) \circ \eta) \otimes \bigotimes\nolimits_B [\![\tau]\!]) \\
&\sim (id_Y \otimes \varepsilon) \circ ((([\![\sigma]\!] \otimes id_{\bigotimes_B Z}) \circ \eta) \otimes \bigotimes\nolimits_B [\![\tau]\!]) \\
&\sim [\![\sigma]\!] \circ ((id_{\bigotimes_B Z} \otimes \varepsilon) \circ (\eta \otimes id_{\bigotimes_B Z})) \circ \bigotimes\nolimits_B [\![\tau]\!] \\
&\sim [\![\sigma]\!] \circ \bigotimes\nolimits_B [\![\tau]\!] \sim [\![\sigma[\tau/y]]\!]
\end{aligned}
$$

Since $\sigma[\tau/y]$ derives $\vdash q[p/x]\colon Y$, this concludes this case.

The induction step follows straightforwardly from the induction hypothesis. If, for example, E is $\langle F, u\rangle$, then δ must end in rule ($\otimes$I) with two premises $\vdash F[s']\colon Y$ and $\vdash u\colon Z$, derived by σ and τ. We apply the induction hypothesis to σ to obtain σ'. Then we note that replacing σ with σ' in δ gives us a derivation ρ of the required term. The assertion then follows: $[\![\delta]\!] = \otimes I \circ ([\![\sigma]\!] \otimes [\![\tau]\!]) \sim \otimes I \circ ([\![\sigma']\!] \otimes [\![\tau]\!]) = [\![\rho]\!]$. □

We can conclude that the compilation in the previous section computes the same results as the standard reduction introduced here.

Corollary 1. *If δ derives $\vdash s\colon [A]$ and $s \longrightarrow^* [v]$ for some $v \in \mathcal{V}_A$, then $\Phi_{[\![\delta]\!]}$ is the function of type $\mathcal{V}_1 \times 1 \to \mathcal{V}_A$ that maps the unique element of its domain to v.*

This corollary follows by observing that $\vdash [v]\colon [A]$ must be derived by some ρ with ([ ]I) as last rule, that $\Phi_{[\![\rho]\!]}$ is by definition the function in the corollary and that $\Phi_{[\![\rho]\!]} \sim \Phi_{[\![\delta]\!]}$ follows from the theorem.

5 Logarithmic Space

In this section we describe a precise correspondence between the class of functions representable in INTML and the class of functions computable in logarithmic space.

First we must define how to represent functions from binary strings to binary strings in INTML. In principle, such functions can be programmed directly as terms of type $A \cdot [B] \multimap [B]$, where B is a working class type of binary strings. However, this would lead to linear space usage, so we use a more interactive type as discussed in Sec. 2.

With the constants *min* and *succ*, we can view any closed type $[A]$ as a type of natural numbers that can represent the numbers from 0 to $|\mathcal{V}_A| - 1$: the number i is encoded by the normal form of $[\mathit{succ}^i_A(\mathit{min}_A)]$, for which we write $\langle i\rangle_A$.

Then, for every closed type B, binary strings of length at most $|\mathcal{V}_B|$ can be represented as terms of type $\mathbb{B}_A(B) = A \cdot [B] \multimap [3]$: a binary string s is encoded by a function that when applied to $\langle i\rangle_B$ returns: b if the i-th symbol in s is b and 2 if i exceeds the length of s. Write $\langle s\rangle_{A,B}$ for the encoding of $s \in \{0,1\}^{\leq|\mathcal{V}_B|}$ in $\mathbb{B}_A(B)$.

Functions that take as inputs strings of arbitrary length can be represented using type variables. Let X be the upper class type $A \cdot \mathbb{B}_B(\alpha) \multimap \mathbb{B}_C(D)$, where α is a type variable and where A, B, C and D are arbitrary types that may also contain α, but not other type variables. If E is a closed type then $X[E/\alpha]$ is a type of functions from strings of length at most $|\mathcal{V}_E|$ to strings of length at most $|\mathcal{V}_{D[E/\alpha]}|$. We say that a term $\vdash t\colon X$ *represents* a function $\phi : \{0,1\}^* \to \{0,1\}^*$ if and only if:

- For every n and every $x \in \{0,1\}^n$, $|\phi(x)| \leq |\mathcal{V}_{D[n/\alpha]}|$;
- For every n, every $x \in \{0,1\}^n$ and every E with $|\mathcal{V}_E| \geq n$, if $\phi(x) = b_1 \dots b_m$ then for every $i \leq |\mathcal{V}_{D[n/\alpha]}|$, the term $t[E/\alpha]\langle s\rangle_{B[E/\alpha],E}\langle i\rangle_{D[E/\alpha]}$ reduces to $\langle b_i\rangle_3$ if $i \leq m$ and to $\langle 2\rangle_3$ if $i > m$.

We remark that one may also use an alternate definition, for which the following theorems are also valid, but which does not refer to upper class reduction: instead of requiring $t[E/\alpha]\langle s\rangle_{B[E/\alpha],E}\langle i\rangle_{D[E/\alpha]}$ to reduce to one of $\langle b_i\rangle_3$ or $\langle 2\rangle_3$, one may ask that the circuit for this term have the same behaviour as either $\langle b_i\rangle_3$ or $\langle 2\rangle_3$, depending on i.

Theorem 2 (Logspace Soundness). *If $\phi\colon \{0,1\}^* \to \{0,1\}^*$ is represented by t, then ϕ is computable in logarithmic space. Moreover, a* LOGSPACE *algorithm computing ϕ is given by* INTML*-evaluation of t.*

Proof. Compiling t to a working class term yields a term $x{:}X^- \vdash f\colon X^+$. We now choose $E_n = 2^{\lceil \log n\rceil}$ to substitute for α. Clearly, $|\mathcal{V}_{E_n}| \geq n$, but $|E_n| \leq 2(\log n + 1)$. By Prop. 1, evaluation of $f[E_n/\alpha]$ (i.e. computation of ϕ on strings of length up to n) can be implemented using space linear in $|2^{\lceil \log n\rceil}|$, thus logarithmic in n.

Theorem 3 (Logspace Completeness). *Any function $\phi\colon \{0,1\}^* \to \{0,1\}^*$ computable in logarithmic space is represented by some upper class term t_ϕ.*

The proof goes by an easy encoding of Offline Turing Machines in INTML. The step function of a given LOGSPACE OTM M can be mimicked by an upper class term $\mid x{:}[\alpha] \multimap [3] \vdash \mathtt{step}_M \colon [S(\alpha)] \multimap [S(\alpha) + S(\alpha)]$, where x represents the input string and where $S(\alpha)$ is a type with free variable α that can encode the state of M. The term t_M can be obtained by passing $\mathtt{step}_M$ to the `loop` combinator.

6 Conclusion

We have found that the Int construction is a good way of structuring space bounded computation that can help us to understand the principles of space bounded functional programming. This view has guided the design of INTML, a simple language capturing LOGSPACE. Initial experience with an experimental implementation of INTML suggests that, with suitable type inference, INTML can be made to be quite usable.

We hope that our systematic approach will be helpful for developing INTML further. For instance, recent results on capturing non-deterministic token machines by the Int construction [9] may perhaps be used to develop a NLOGSPACE-version of INTML.

References

1. Abramsky, S., Jagadeesan, R.: New Foundations for the Geometry of Interaction. Inf. Comput. **111**(1) (1994), 53–119
2. Abramsky, S., Haghverdi, E., Scott, P.J.: Geometry of interaction and linear combinatory algebras. Math. Struct. in Comput. Sci. **12**(5) (2002) 625–665
3. Asperti, A., Guerrini, S.: The optimal implementation of functional programming languages. Cambridge University Press (1998)
4. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda calculus. Inf. Comput. **207**(1) (2009) 41–62
5. Bonfante, G.: Some programming languages for Logspace and Ptime. In: AMAST. (2006) 66–80
6. Hasegawa, M.: On traced monoidal closed categories. Math. Struct. in Comput. Sci. **19**(2) (2009) 217–244
7. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Math. Proc. Cambridge Philos. Soc. **119**(3) (1996) 447–468
8. Katsumata, S.: Attribute grammars and categorical semantics. In: ICALP. (2008) 271–282
9. Katsumata, S., Hoshino, N.: Int construction and semibiproducts. RIMS-Technical Report 1676 (2009)
10. Kristiansen, L.: Neat function algebraic characterizations of LOGSPACE and LINSPACE. Computational Complexity **14** (2005) 72–88
11. Mackie, I.: The geometry of interaction machine. In: POPL. (1995) 198–208
12. Mairson, H.G.: From hilbert spaces to dilbert spaces: Context semantics made simple. In: FSTTCS. (2002) 2–17
13. Melliès, P.A.: Functorial boxes in string diagrams. In: CSL. (2006) 1–30
14. Neergaard, P.M.: A functional language for logarithmic space. In: APLAS. (2004) 311–326
15. Schöpp, U.: Stratified bounded affine logic for logarithmic space. In: LICS. (2007) 411–420
16. Szepietowski, A.: Turing Machines with Sublogarithmic Space. Springer LNCS 843. (1994)
17. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)