Computation by Interaction for Space Bounded Functional Programming

Ugo Dal Lago[a], Ulrich Schöpp[b]

[a] *Università di Bologna, Italy*
[b] *Ludwig-Maximilians-Universität München, Germany*

Abstract

We consider the problem of supporting sublinear space programming in a functional programming language. 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 modelling 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 core functional language INTML 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 the complexity classes LOGSPACE and NLOGSPACE.

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. The programming language should give him or her the means to delegate inessential implementation details to the computer. This, of course, requires advanced compilation techniques, given that machine languages are too low level to be convenient.

In this paper we consider the particular question of how algorithms with strongly limited memory space can be supported in functional programming. We consider sublinear space algorithms, which are characterised by having less memory than would be needed to store their input. Such algorithms are useful for computing with large data that may not fit into memory. Examples of concrete application scenarios in which the situation above manifests are streaming algorithms [1], web crawling, GPU computing, and statistical data mining.

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 the two algorithms sequentially, 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 as they are needed. Doing so, however, is tedious and error-prone. We believe that programming language support should be very useful for the implementation of algorithms that rely on on-demand recomputation of intermediate values. Instead of implementing composition with on-demand recomputation by hand, the programmer should be able to write function composition in the usual way and have a compiler generate the complicated, interactive, program.

The possibilities for programming language support for sublinear space computation have been explored in work on implicit characterisations of the complexity class LOGSPACE of those functions which can be computed by algorithms with logarithmic space usage. A number of characterisations of this complexity class have been explored in terms of function algebras [2] and linear logics [3] (there is more work for LOGSPACE-predicates, e.g. [4, 5], but we focus on functions here). However, these characterisations are still far away from being real programming languages.

In this paper we work towards the goal of making the abstractions explored in this line of work usable in functional programming. In contrast to previous work [2, 3], we aim to enrich a given language with constructs for working with on-demand recomputation conveniently, rather than hiding this recomputation completely. This is in line with the

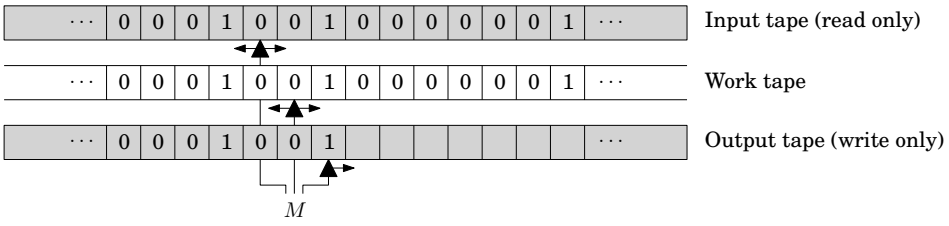


Figure 1: An Offline Turing Machine M

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 work out the details of a minimal functional programming language, INTML, for programming with sublinear space. This language is derived from an instance of the Int construction [6]. 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. The thus obtained INTML type system is conceptually simple, and, with the type inference procedure described in [7], it is also relatively easy to use. Nevertheless, the type system is strong enough to allow LOGSPACE algorithms to be written in a natural way, see [7].

Being able to write programs in an interactive style has a nice byproduct: many useful programming constructs, like iteration and control operators are expressible in INTML (see Section 4). Moreover, INTML can be easily tuned as to capture not only deterministic LOGSPACE computation but also nondeterministic LOGSPACE computation and constant space computation, as described in Section 5.

This paper develops and expands the material first presented at ESOP 2010 [8]. It takes into account further developments from APLAS 2010 [7] and APLAS 2011 [9].

1.1. Space Bounded Computation

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

In the definition of space complexity classes, in particular sublinear space complexity classes, one uses *Offline Turing Machines* (OTMS). These 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. See Fig. 1 for a graphical representation of an OTM with one working tape. The intention of the special treatment of the input and output tapes is that their content is not stored in memory. The input is thought to be externally stored data with random access and the output is given 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 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 $\{\mathit{inl}(x) \mid x \in A\} \cup \{\mathit{inr}(y) \mid y \in B\}$. 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

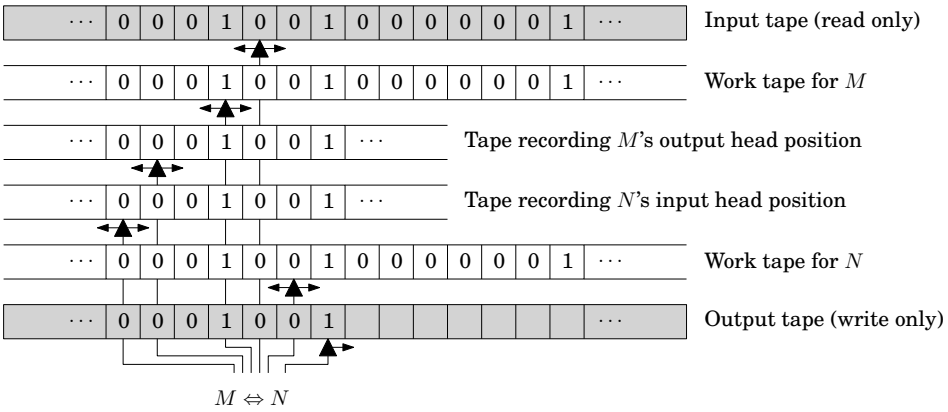


Figure 2: Composition of two Offline Turing machines M and N

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. If the space usage is at least logarithmic then storing the head positions does not change the space usage asymptotically. 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" [10].

While at first sight, the step from Turing Machines to Offline Turing Machines looks like a technicality, it is a step from unidirectional to bidirectional computation. While standard Turing Machines are composed just by running 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. A Turing Machine can implement this dialogue with the tapes shown in Fig. 2.

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 doing this is to study space-bounded computation in terms of the *Int construction* of Joyal, Street & Verity [6, 11]. 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 [12, 13] and has many applications, e.g. to Attribute Grammars [14].

1.2. Structuring Space Bounded Computation

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.

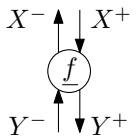
The idea is to start from a computational model, e.g. the partial computable functions as formalised by Turing Machines, and then apply the Int construction to this model. The result is a model that still contains the original one, but that in addition also captures bidirectional (or interactive) computation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

In its general form, the Int construction takes a traced monoidal category $\mathbf{B}$ and gives a category Int($\mathbf{B}$) that represents bidirectional computation in $\mathbf{B}$. For the aims of this introduction, consider the special case where $\mathbf{B}$ is the category **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 **Pfn** with respect to tagged disjoint union as monoidal structure, then we obtain the following category Int(**Pfn**). Its objects are pairs (X^-, X^+) of sets X^- and X^+. The object (X^-, X^+) can be thought of as the interface of an entity that can be interacted with. The set X^- contains all the requests that may be sent to the entity and X^+ contains all possible answers. A morphism f from $X = (X^-, X^+)$ to $Y = (Y^-, Y^+)$ is a partial function $\underline{f} \colon Y^- + X^+ \to Y^+ + X^-$. It explains how to answer requests to interface Y, given that one can answer requests to interface X. To answer a request in Y^-, we apply $\underline{f}$ to it. If the result is in Y^+, then this is our answer. If it is in X^-,

then we can answer that request with an element of X^+ and then apply $\underline{f}$ to this answer. We repeat this until we get an answer in Y^+ (i.e. possibly forever). Notice the similarity to the behaviour of an OTM.

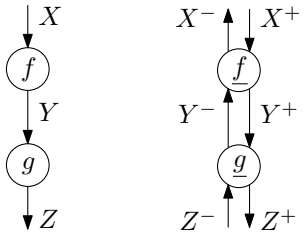
Morphisms capture bidirectional computation. Let us think of the partial function $\underline{f}$ as a message-passing node with two input wires and two output wires as drawn below:



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 draw the two edges for X^- and X^+ as a single edge in which messages may travel both ways (and likewise for Y). Thus we obtain the node depicted below, 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 Int(**Pfn**) connects nodes using a form of message passing. 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 in Int(**Pfn**) as morphisms of type

$$(State \times \mathbb{N}, State \times \Sigma) \to (\mathbb{N}, \Sigma) \ .$$

This follows immediately from the definition of morphisms and the discussion in Section 1.1.

This view of OTMs as morphisms in Int(**Pfn**) is useful because we can use the well-known structure of this category for constructing and manipulating OTMs. Firstly, composition of OTMs in Int(**Pfn**) agrees with the standard composition of such machines. But Int(**Pfn**) has much more useful structure. We briefly outline the structure that we use in this paper.

Monoidal Structure. Int(**Pfn**) has a monoidal structure $\otimes$ that on objects is defined by $X \otimes Y = (X^- + Y^-, X^+ + Y^+)$ and whose unit is $I = (\emptyset, \emptyset)$. The object $X \otimes Y$ represents the joint interface of two entities with interfaces X and Y respectively. With the monoidal structure, one can easily model message passing nodes with more than one output or input. For example, a morphism of type $f \colon X \otimes Y \to Z$ can be understood as a message passing node with two inputs, see the figure below. It is indicated on the right, which type of messages be passed in which direction along the wires.

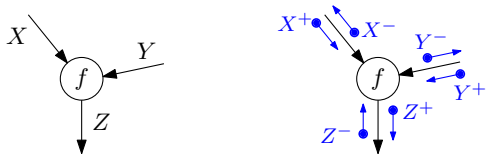


Figure 3: Types of messages

With this understanding, the morphisms of Int(**Pfn**) can be considered as message passing circuits. For instance, given $h\colon X \to V \otimes U$, $g\colon V \otimes W \to I$ and $k\colon Y \otimes Z \to U$, the circuit in Fig. 4 below corresponds to $(g \otimes U) \circ (id_V \otimes swap_{W,U}) \circ (h \otimes k)$.

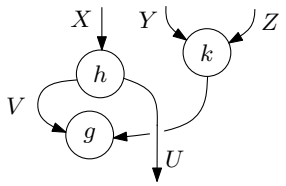


Figure 4: A circuit

In this circuit, an element of U^- may be passed to h against the direction of the output wire. With this input, the node h may then perhaps decide to pass an element of V^+ along the wire to g or to return an element of X^- to the environment. It may also decide not to do anything, in which case the whole computation will be blocked. Also, message passing inside the circuit may go on indefinitely in an infinite loop.

Higher-order Functions. It is well known that the operation $(-)^*$ defined to exchange question and answer sets, i.e. $(X^-, X^+)^* = (X^+, X^-)$, makes Int(**Pfn**) compact closed, see e.g. [11]. Therefore, we obtain a linear function space by letting $X \multimap Y = X^* \otimes Y$. The interface $X \multimap Y = (X^+ + Y^-, X^- + Y^+)$ represents functions that are computed interactively like in game semantics [15, 16]. If one wants to know the result of the function, one sends it a query from Y^- for the result. The function may directly answer this request with an answer from Y^+. Or, it may respond with a request in X^- for its argument. The environment can answer this request by passing the answer in X^+ as a new 'question' to the function.

Thunks. Also useful is the object $[A] = (1, A)$, where 1 is a singleton set $1 = \{*\}$. The single element $*$ of 1 signals an explicit request for a value of type A, which may be provided as an answer. Thus, one may think of $[A]$ as a type of thunks that are evaluated on demand.

Subexponentials. Of particular importance is what we call subexponentials. By this we mean objects $X^{\otimes A}$ (where A is finite) that are isomorphic to $X \otimes \cdots \otimes X$ ($|A|$ times). They are defined by $(X^{\otimes A})^- = A \times X^-$ and $(X^{\otimes A})^+ = A \times X^+$. The first component in the messages indicates which component of the tensor product we are communicating with.

Embedding **Pfn**. Finally, we note that in moving from **Pfn** to Int(**Pfn**) we do not lose anything. 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.

To see how the structure of Int(**Pfn**) is useful for working with OTMs, consider a morphism of type

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

in Int(**Pfn**). By definition, it is a partial function from $(State \times (\emptyset + \Sigma)) + (\mathbb{N} + \emptyset)$ to $(State \times (\mathbb{N} + \emptyset)) + (\emptyset + \Sigma)$, which, if we remove the superfluous empty sets, is the same as the type of an Offline Turing Machine, as described above. As a morphism of type (1), an Offline Turing Machine is modelled simply as a map from input words to output words, where words are encoded as (linear) functions of type $\mathcal{I}\mathbb{N} \multimap \mathcal{I}\Sigma$. This encoding 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 Int(**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 lambda-calculus to manipulate the input functions.

1.3. Constructing IntML

In this paper we develop the functional programming language IntML based on understanding space-bounded computation in terms of the Int construction. We start with a simple first-order programming language and apply the Int construction to a *term model* of it. This gives us a structure of message passing circuits, very similar to Int(**Pfn**) above. The only difference is that **Pfn** is now replaced with the term model: sets become types and partial functions become terms.

The language IntML provides syntax for this structure of message passing circuits. It is a typed λ-calculus that extends the simple first order language with higher-order primitives for a convenient construction and manipulation of circuits. IntML has two classes of terms and types: one for the language we started with and one for constructing message passing circuits. We call the former the *base language* and the latter the *interactive language*.

The interactive language in IntML is inspired by Dual Light Affine Logic (DLAL) [17]. IntML treats the subexponential $X^{\otimes A}$ much like DLAL treats the exponential modality $!X$. 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 $X^{\otimes A} \multimap Y$. In the syntax, we write $A \cdot X \multimap Y$ for them. The restriction to negative occurrences of $X^{\otimes A}$ much simplifies the type system without being too limiting in applications.

1.4. Contribution

The main contribution of this paper is to identify a higher-order functional language and a translation to circuits that can be evaluated using sublinear space. The idea of translating programs to circuits by means of interpretation in an interactive model is not new. In the literature one can find a number of higher-order functional languages, such as [18, 19], that are translated to circuits using this approach. However, without further restrictions one cannot prove that circuits evaluate in sublinear space. Even in languages without recursion, e.g. [19], one may encounter messages of exponential size during circuit evaluation. What is new in this paper is that we show how to refine language and model in order to obtain an expressive language for sublinear space programming. The main issue is the control of exponentials, which are responsible for size increases in message passing. In order to obtain space bounds, the exponentials must therefore be restricted. The difficulty is to do so without the programming language becoming too weak. In this paper we argue that IntML with its subexponentials represents a good solution to this problem.

Subexponentials make IntML an expressive higher-order language. For example, we show that it can type the Kierstead terms, which cannot be typed in similar linear type systems, such as [28]. Moreover, the proof of LOGSPACE-completeness in this paper is completely straightforward. In earlier work, such as [2, 3], the completeness proofs were more involved. The expressiveness of IntML is further illustrated by the extended examples in [7].

Subexponentials not only give us control over space usage, they also afford an efficient treatment of controlled duplication. For example, Ghica and Smith [20] treat copying in the language by actual duplication of terms. Here we show how to allow such copying without the need to duplicate parts of the program.

Finally, the language IntML presented in this paper is the first higher-order language with logarithmic space bounds that allows one to use (and define!) higher-order combinators such as for tail recursion and call with current continuation.

2. IntML

In this section we give the definition of IntML and its type system, as yet without reference to circuits or space complexity.

$$\frac{}{\Gamma,\, x\colon A \vdash x : A} \qquad \frac{}{\Gamma \vdash * : 1} \qquad \frac{c \in C(B_1, \ldots, B_n; A) \qquad \{\Gamma \vdash f_i : B_i\}_{i=1,\ldots,n}}{\Gamma \vdash c(f_1, \ldots, f_n) : A}$$

$$\frac{\Gamma \vdash f : A \qquad \Gamma \vdash g : B}{\Gamma \vdash \langle f, g\rangle : A \times B} \qquad \frac{\Gamma \vdash f : A \times B}{\Gamma \vdash \mathsf{fst}(f) : A} \qquad \frac{\Gamma \vdash f : A \times B}{\Gamma \vdash \mathsf{snd}(f) : B}$$

$$\frac{\Gamma \vdash f : A}{\Gamma \vdash \mathsf{inl}(f) : A + B} \qquad \frac{\Gamma \vdash f : B}{\Gamma \vdash \mathsf{inr}(f) : A + B}$$

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

$$\frac{\Gamma \vdash f : 0}{\Gamma \vdash \mathsf{image}(f) : A} \qquad \frac{\Gamma \vdash f : A \qquad \Gamma,\, x\colon A \vdash g : B + A}{\Gamma \vdash \mathsf{let}\ x = f\ \mathsf{loop}\ g : B}$$

Figure 5: Base Language Typing Rules

2.1. Base Language

We begin by fixing the details of the base language. We emphasise that this language was chosen mainly for simplicity and that richer languages can be chosen without affecting the construction of INTML. The types of the base language are generated by the grammar

$$A ::= \alpha \mid 0 \mid 1 \mid A + A \mid A \times A \ .$$

They are built from type variables using product and coproduct types.

The terms of the base language are parametrised by a signature of *constants* C, which specifies for each constant c an arity n, the types $B_1, \ldots, B_n$ of its parameters and finally the type A of the constant itself. The set of constants with type A and parameter types $B_1, \ldots, B_n$ is written $C(B_1, \ldots, B_n, A)$. Formally, the terms of the base language are generated by the grammar

$$\begin{aligned} f, g, h \ ::= \ & x \mid c(f_1, \ldots, f_n) \mid * \mid \langle f, g\rangle \mid \mathsf{fst}(f) \mid \mathsf{snd}(f) \\ & \mid \mathsf{image}(f) \mid \mathsf{inl}(f) \mid \mathsf{inr}(f) \mid \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g \mid \mathsf{inr}(y) \Rightarrow h \\ & \mid \mathsf{let}\ x = f\ \mathsf{loop}\ g \ , \end{aligned}$$

in which x and y range over variables and c ranges over constants.

The typing rules are given in Fig. 5. There are no linearity restrictions and the contexts in these rules are simply finite mappings from variables to types. Thus, weakening and contraction rules are admissible.

The base language is a fairly standard call-by-value language. As a consequence, its operational semantics crucially relies on the notion of a *value*:

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

The loop construct $\mathsf{let}\ x = f\ \mathsf{loop}\ g$ is a simple way of capturing iteration in the language. Its operational semantics is defined by the rewrite rule

$$\begin{aligned} \mathsf{let}\ x = v\ \mathsf{loop}\ g \ \rightarrow \ & \mathsf{case}\ g[v/x]\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow y \\ & \mid \mathsf{inr}(z) \Rightarrow (\mathsf{let}\ x = z\ \mathsf{loop}\ g) \ . \end{aligned}$$

The only other nonstandard operator in the based language is image for which no rewrite rule exists (no closed values

of type 0 can be produced). The other rewrite rules are standard:

$$\begin{array}{rcl} \mathsf{fst}(\langle v, w\rangle) & \to & v \\ \mathsf{snd}(\langle v, w\rangle) & \to & w \\ \mathsf{case}\ (\mathsf{inl}(v))\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow f \mid \mathsf{inr}(y) \Rightarrow g & \to & f[v/x] \\ \mathsf{case}\ (\mathsf{inr}(v))\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow f \mid \mathsf{inr}(y) \Rightarrow g & \to & g[v/y] \end{array}$$

The operational semantics of constants depends on the signature C. We assume that rewrite rules are specified as part of the signature.

Evaluation contexts are defined as follows:

$$\begin{array}{rcl} E, F, G & ::= & [\cdot] \mid c(v_1, \ldots, v_n, E, f_1, \ldots, f_m) \mid \langle E, g\rangle \mid \langle v, E\rangle \mid \mathsf{fst}(E) \mid \mathsf{snd}(E) \\ & & \mid \mathsf{image}(E) \mid \mathsf{inl}(E) \mid \mathsf{inr}(E) \mid \mathsf{case}\ E\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g \mid \mathsf{inr}(y) \Rightarrow h \\ & & \mid \mathsf{let}\ x = E\ \mathsf{loop}\ g \end{array}$$

A term f is said to *reduce* to g in one step, written $f \mapsto g$ if and only if there are an evaluation context E and two terms h, p such that $h \to p$, $f = E[h]$ and $g = E[p]$. We write $\mapsto^*$ for the reflexive transitive closure of $\mapsto$, as usual.

Definition 1. Two terms $\Gamma \vdash f : A$ and $\Gamma \vdash g : A$ are *extensionally equal*, if, for any substitution σ that assigns to each variable $x\colon B$ in Γ a closed value of type B, and any closed value v of type A, we have $f[\sigma] \mapsto^* v$ if and only if $g[\sigma] \mapsto^* v$.

We shall consider a type A to be smaller than a type B when any closed value of type A can be encoded into a closed value of type B, and encoding and decoding terms can be written in the base language. Formally, this is defined by means of section-retraction pairs.

Definition 2. A *section-retraction pair* between types A and B in context Γ is a pair of terms $\Gamma, x\colon A \vdash s : B$ and $\Gamma, y\colon B \vdash r : A$ such that $\Gamma, x\colon A \vdash r[s/y] : A$ and $\Gamma, x\colon A \vdash x : A$ are extensionally equal.

Definition 3. Type A is a *retract* of B, written $A \triangleleft B$, if there exists a *section-retraction pair* between A and B in the empty context.

2.2. Interactive Language

In Section 1.3 we have explained the interactive language as a means for constructing and manipulating message-passing circuits, which are themselves implemented by base language terms. Nevertheless, the interactive language takes the form of a typed lambda calculus that can be described without reference to circuits and that allows one to write programs following a handier functional paradigm.

Interactive types are defined by the grammar

$$X, Y \quad ::= \quad [A] \mid X \otimes Y \mid A \cdot X \multimap Y \ ,$$

in which A ranges over base language types.

The shape of interactive types already tells you two key aspects of the interactive language, even if its concrete form has not been described yet:

- Interactive objects can incorporate base language objects via the type construction $[A]$: an interactive term of such a type is meant to be a thunk that on demand returns a base language value of type A.
- One can build higher-order functions, and those higher-functions are typed according to a linear type discipline. The type $A \cdot X \multimap Y$ consists of functions from X to Y, which use their argument at most 'A times'. In this way, the affine function space becomes $1 \cdot X \multimap Y$. We write short $X \multimap Y$ for $1 \cdot X \multimap Y$. The usual, nonlinear function space *would* be $\omega \cdot X \multimap Y$, if we had allowed a type ω of natural numbers in the base language. It is important for establishing space bounds, however, that the function space is more controlled.

$$\textsc{Ax}\ \frac{}{\Gamma \mid x : 1 \cdot X \vdash x : X} \qquad \textsc{Struct}\ \frac{\Gamma \mid \Phi,\ x : A \cdot X \vdash t : Y}{\Gamma \mid \Phi,\ x : B \cdot X \vdash t : Y}\ A \triangleleft B$$

$$\textsc{Weak}\ \frac{\Gamma \mid \Phi \vdash t : Y}{\Gamma \mid \Phi,\ x : A \cdot X \vdash t : Y} \qquad \textsc{Exch}\ \frac{\Gamma \mid \Phi,\ x : A \cdot X,\ y : B \cdot Y,\ \Psi \vdash t : Z}{\Gamma \mid \Phi,\ y : B \cdot Y,\ x : A \cdot X,\ \Psi \vdash t : Z}$$

$$\textsc{Copy}\ \frac{\Gamma \mid \Phi \vdash s : X \qquad \Gamma \mid \Psi,\ x : A \cdot X,\ y : B \cdot X \vdash t : Z}{\Gamma \mid \Psi,\ (A + B) \cdot \Phi \vdash \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t : Z}$$

Figure 6: Interactive Typing: Structural Rules

The set of interactive terms is defined as follows:

$$\begin{aligned} s, t \ ::= \ & x \mid \langle t, s \rangle \mid \mathsf{let}\ \langle x, y \rangle = s\ \mathsf{in}\ t \mid \lambda x.\, t \mid s\ t \mid [f] \mid \mathsf{let}\ [x] = s\ \mathsf{in}\ t \\ & \mid \mathsf{image}(f) \mid \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow s \mid \mathsf{inr}(y) \Rightarrow t \\ & \mid \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t \mid \mathsf{hack}(x.f) \end{aligned}$$

In this grammar f ranges over base language terms.

Although the meaning of the various constructs will be clear only when typing rules and reductions are introduced, we can already introduce a classification of the interactive operators:

- There is an affine lambda calculus with pairs, first of all. So we have λ-abstraction and application ($\lambda x.\, t$ and $s\ t$, respectively), as well as standard terms for pairs ($\langle t, s \rangle$ and $\mathsf{let}\ \langle x, y \rangle = s\ \mathsf{in}\ t$).
- Base language terms can appear in interactive terms. First, the term $[f]$ denotes a thunk that computes f on demand. Such thunks can be forced using the operator $\mathsf{let}\ [x] = s\ \mathsf{in}\ t$. Another way of embedding base language terms into interactive terms is the construct

  $$\mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow s \mid \mathsf{inr}(y) \Rightarrow t\ ,$$

  which behaves as either $s[v/x]$ or $t[w/y]$, depending on whether the outcome of the evaluation of f is $\mathsf{inl}(v)$ or $\mathsf{inr}(w)$. There is a third, more intricate, way of making use of the base language in the interactive language, namely the operator $\mathsf{hack}(x.f)$, on which we comment at the end of this section and in Section 4.
- The interactive calculus is affine and does not allow arbitrary copying. Controlled copying is nevertheless possible by means of the term $\mathsf{copy}\ t\ \mathsf{as}\ x, y\ \mathsf{in}\ s$. It would be possible to formulate the type system without explicit copy terms, such that copy is inserted implicitly when a variable is used more than once. While this may indeed be preferable for practical programming, we use explicit terms here for conceptual clarity.

The interactive type system derives sequents of the form $\Gamma \mid \Phi \vdash t : Y$, where Γ is a base language context and the *interactive context* Φ is a sequence $x_1 : A_1 \cdot X_1, \ldots, x_1 : A_n \cdot X_n$ of assignments of variables to expressions of the form $A_i \cdot X_i$ (called *subexponential types*), in which A_i is a base language type and X_i is an interactive type. The declaration $x_i : A_i \cdot X_i$ means that x_i stands for an A_i-fold number of copies of a value of type X_i (one copy for each closed value of type A_i). The presence of the two contexts in the typing judgement reflects that base language terms and their variables can appear within interactive terms.

Before we gradually introduce the typing rules, we fix the notation $A \cdot \Phi$, which stands for the context obtained by replacing each declaration $x : B \cdot X$ in Φ with $x : (A \times B) \cdot X$.

The structural rules (see Fig. 6) allow one to manipulate the context Φ, which is treated *multiplicatively* in the logical rules. Weak and Exch are standard rules for weakening and context reordering. Rule Ax allows one to type a variable x, if the context Φ consists of the sole declaration $x : 1 \cdot X$. Informally, this means that x denotes a single copy of X. With rule Struct it is always possible to increase the multiplicity of a variable: if A copies of x suffice for t, and $A \triangleleft B$, then it is safe to declare that B copies of x are enough. Rule Struct is similar to a subtyping rule. The side condition $A \triangleleft B$ is here defined to mean that A is a retract of B. Instead of this quite general semantic side condition, one may also use syntactic approximations of the notion of a retract. We refer to [7] for an exploration of possible choices.

$$[\,]\mathrm{I}\ \frac{\Gamma \vdash f : A}{\Gamma \mid \cdot \vdash [f] : [A]} \qquad [\,]\mathrm{E}\ \frac{\Gamma \mid \Phi \vdash s : [A] \qquad \Gamma,\, x\colon A \mid \Psi \vdash t : [B]}{\Gamma \mid \Phi,\, A \cdot \Psi \vdash \mathsf{let}\,[x] = s\,\mathsf{in}\,t : [B]}$$

$$\otimes\mathrm{I}\ \frac{\Gamma \mid \Phi \vdash s : X \qquad \Gamma \mid \Psi \vdash t : Y}{\Gamma \mid \Phi,\, \Psi \vdash \langle s, t\rangle : X \otimes Y} \qquad \otimes\mathrm{E}\ \frac{\begin{array}{c}\Gamma \mid \Phi \vdash s : X \otimes Y \\ \Gamma \mid \Psi,\, x\colon A\cdot X,\, y\colon A\cdot Y \vdash t : Z\end{array}}{\Gamma \mid \Psi,\, A \cdot \Phi \vdash \mathsf{let}\,\langle x, y\rangle = s\,\mathsf{in}\,t : Z}$$

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

Figure 7: Interactive Typing: Introductions and Eliminations

$$0\mathrm{E}^{\mathsf{c}}\ \frac{\Gamma \vdash f : 0}{\Gamma \mid \cdot \vdash \mathsf{image}(f) : X}$$

$$+\mathrm{E}^{\mathsf{c}}\ \frac{\Gamma \vdash v : A + B \qquad \Gamma,\, x\colon A \mid \Phi \vdash s : X \qquad \Gamma,\, y\colon B \mid \Phi \vdash t : X}{\Gamma \mid \Phi \vdash \mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow s \mid \mathsf{inr}(y) \Rightarrow t : X}\ v \text{ is a value}$$

Figure 8: Interactive Typing: Cross-Eliminations

The structural rules are set up so that variables declared in an interactive context Φ must be used linearly. However, the calculus does support explicit duplication. A specific term construct serves this purpose, and can be typed by rule Copy.

The introduction and elimination rules for the three type formers $[-]$, $\otimes$ and $\multimap$ appear in Fig. 7. These rules are fairly standard for a linear λ-calculus. Only perhaps rules [ ]I and [ ]E for building a thunk from a base language term and for forcing it are of note. In rule [ ]E the term ($\mathsf{let}\,[x] = s\,\mathsf{in}\,t$) allows one to force a thunk s and bind its value to a variable x in the base language context of t. The value of the thunk thus becomes available in t and, as base language contexts are treated additively, this value may be used many times in t.

Fig. 8 gives rules for eliminating base language terms over interactive terms. For example, $+\mathrm{E}^{\mathsf{c}}$ allows one to define interactive terms by case distinction over a base language term. In this rule v is restricted to be a value and it will typically be a variable. The typical use of the case-term is of the form $\mathsf{let}\,[x] = s\,\mathsf{in}\,\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow t \mid \mathsf{inr}(z) \Rightarrow u$. With the exception of Section 5.2, all results in this paper remain true verbatim also if one allows arbitrary base terms for v in rule $+\mathrm{E}^{\mathsf{c}}$.

The intml type system is completed by rule Hack in Fig. 9. In this rule X is an arbitrary interactive type and X^- and X^+ denote the base types defined by:

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

Rule Hack is the only rule that requires a detailed understanding of the translation of interactive terms to message passing circuits. This rule allows one to define an interactive term of type X by giving explicitly a base language term f that implements a message passing node with interface X. Therefore, rule Hack can only be explained properly once the compilation to circuits that are implemented in the base language has been explained in detail. For now, the reader should consider rule Hack like a method for inlining C code in an ML program, whose understanding requires a

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

Figure 9: Interactive Typing: Hacking

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

Figure 10: Interactive Language: Closed Reduction Semantics

detailed knowledge of the runtime system.

The other terms can however be understood without any reference to circuits. We next define an operational semantics for the interactive terms which specifies their meaning. The translation of interactive terms to circuits will then be a space-efficient implementation of this specification.

The operational semantics of the interactive language is formulated as a set of rewrite rules that are applicable in any evaluation context derivable from the following grammar:

$$
\begin{aligned}
E, F \quad ::= \quad & [\cdot] \mid \langle E, t\rangle \mid \langle t, E\rangle \mid \mathsf{let}\ \langle x, y\rangle = E\ \mathsf{in}\ t \\
& \mid E\ t \mid t\ E \mid \mathsf{let}\ [x] = E\ \mathsf{in}\ t \mid \mathsf{copy}\ E\ \mathsf{as}\ x, y\ \mathsf{in}\ t
\end{aligned}
$$

The rewrite rules are given in Fig. 10, where s and t are assumed to be *closed* terms, i.e. terms not containing any free occurrence of any variable. A term t *reduces* to s, written $t \mapsto s$, if there are u, q such that $u \to q$, $t = E[u]$ and $s = E[q]$. In Section 3.6, we show that the translation of interactive terms to message passing circuits is sound with respect to reduction: $t \mapsto s$ implies that the circuits for t and s have the same message passing behaviour (even if they are not the same).

This completes the definition of the interactive language of INTML. The reader may wonder if in addition to the various ways of embedding base language terms into interactive terms, it is also possible to embed interactive terms into base terms. It is indeed possible to add to the base language a term ($\mathsf{let}\ [x] = t\ \mathsf{in}\ f$) that evaluates a closed interactive thunk t and uses the result in a base language term f. In fact, such a term is already admissible, see Corollary 14 in Section 3.6, which is why we have not included it in the definition.

2.3. Examples

2.3.1. Kierstead Terms

The following two λ-terms, attributed to Kierstead, are quite similar in structure, but not equivalent:

$$
\lambda f.f(\lambda x.f(\lambda y.y)) \qquad\qquad \lambda f.f(\lambda x.f(\lambda y.x))
$$

They are often cited in the context of Hyland-Ong games [16], where the two strategies interpreting them would not be distinguishable without pointers. The terms can be turned into INTML terms typeable as

$$
\begin{aligned}
t_1 &= \lambda f.\,\mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\ f_2\ (\lambda y.\ y)) :\ (1+\alpha)\cdot(\alpha\cdot(X \multimap X) \multimap X) \multimap X\ , \\
t_2 &= \lambda f.\,\mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\ f_2\ (\lambda y.\ x)) :\ (1+\alpha)\cdot(\alpha\cdot X \multimap X) \multimap X) \multimap X\ .
\end{aligned}
$$

An INTML type derivation for t_1 has the following form (we abbreviate $(\alpha \cdot (X \multimap X) \multimap X)$ by T):

$$
\dfrac{\dfrac{\dfrac{\text{Ax}\ \dfrac{}{\mid f : 1\cdot T \vdash f : T} \qquad \Pi}{\mid f : ((1+\alpha)\times 1)\cdot T \vdash \mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\ f_2\ (\lambda y.\ y)) : X}\ \textsc{Copy}}{\mid f : (1+\alpha)\cdot T \vdash \mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\ f_2\ (\lambda y.\ y)) : X}\ \textsc{Struct}}{\cdot \mid \cdot \vdash t_1 : (1+\alpha)\cdot T \multimap X}\ \multimap\text{I}
$$

In the right-hand branch, Π stands for the following derivation.

$$\dfrac{\dfrac{\text{Ax}\ \dfrac{}{\mid f_1 : 1\cdot T \vdash f_1 : T} \qquad \multimap\text{E}\ \dfrac{\dfrac{\text{Ax}\ \dfrac{}{\mid f_2 : 1\cdot T \vdash f_2 : T} \qquad \multimap\text{E}\ \dfrac{\dfrac{\text{Ax}\ \dfrac{}{\mid y : 1\cdot X \vdash y : X}}{\cdot \mid \cdot \vdash \lambda y.\, y : X \multimap X}}{\mid f_2 : 1\cdot T \vdash f_2\ (\lambda y.\, y) : X}}{\text{Weak}\ \dfrac{\mid f_2 : 1\cdot T \vdash f_2\ (\lambda y.\, y) : X}{\mid f_2 : 1\cdot T,\ x : 1\cdot X \vdash f_2\ (\lambda y.\, y) : X}} \qquad \multimap\text{I}\ \dfrac{\mid f_2 : 1\cdot T,\ x : 1\cdot X \vdash f_2\ (\lambda y.\, y) : X}{\mid f_2 : 1\cdot T \vdash \lambda x.\, f_2\ (\lambda y.\, y) : X \multimap X}}{\mid f_1 : 1\cdot T,\ f_2 : (\alpha \times 1)\cdot T \vdash f_1\ (\lambda x.\, f_2\ (\lambda y.\, y)) : X}}{\text{Struct}\ \dfrac{\mid f_1 : 1\cdot T,\ f_2 : (\alpha \times 1)\cdot T \vdash f_1\ (\lambda x.\, f_2\ (\lambda y.\, y)) : X}{\mid f_1 : 1\cdot T,\ f_2 : \alpha\cdot T \vdash f_1\ (\lambda x.\, f_2\ (\lambda y.\, y)) : X}}$$

To see that terms t_1 and t_2 can be distinguished in INTML, consider the term

$$s = \lambda g.\, \mathsf{copy}\ g\ \mathsf{as}\ g_1, g_2\ \mathsf{in}\ \mathsf{let}\ [x_1] = g_1\ [\mathtt{false}]\ \mathsf{in}\ \mathsf{let}\ [x_2] = g_2\ [\mathtt{true}]\ \mathsf{in}\ [\mathsf{nor}(x_1, x_2)]\ .$$

In it we consider the base language type $2 = 1 + 1$ as a type of booleans with $\mathtt{false} = \mathsf{inl}(*)$ and $\mathtt{true} = \mathsf{inr}(*)$. The base language term $x_1 \colon 2,\ x_2 \colon 2 \vdash \mathsf{nor}(x_1, x_2) : 2$ defines the usual logical NOR, i.e. $\neg(x_1 \vee x_2)$. We omit its straightforward definition by case distinction on x_1 and x_2.

The term s can be given the two types

$$(1 + 2) \cdot ([2] \multimap [2]) \multimap [2]\ , \qquad\qquad (1 + 2) \cdot ((1 + 2) \cdot [2] \multimap [2]) \multimap [2]\ .$$

As a consequence, we can apply both t_1 and t_2 to s. However, we have $t_1\ s \mapsto^* [\mathtt{true}]$ and $t_2\ s \mapsto^* [\mathtt{false}]$.

2.3.2. Copying Values

A second simple example shows that even though interactive variables may not be copied, base language variable can be used many times. The term

$$\lambda f.\, \lambda y.\, \mathsf{let}\ [x] = y\ \mathsf{in}\ f\ [x]\ [x] :\ \alpha \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]) \multimap [\alpha] \multimap [\delta]$$

shows how after a thunk y has been evaluated, its return value may be used more than once. In the type, α, β, γ and δ are type variables that can be instantiated with arbitrary types. That the type of the argument f is $\alpha \cdot (\dots)$ tells us that f is used in a context where the base language context contains another variable of type α, namely x in this case.

2.4. Type Inference

We note that the subexponential annotations in the above examples and all the examples in the rest of this paper were found by the type inference procedure described in [7]. We have found that in practise when writing programs one does not have to be aware of particular subexponentials in the type.

In the above example on copying values, for instance, it is useful to ignore subexponential annotations on functions at first and think of the type as $([\alpha] \to [\alpha] \to [\delta]) \to [\alpha] \to [\delta]$. Keeping the restrictions on variable duplication and the ways of introducing and eliminating [ ]-types, it is straightforward to come up with the term $\lambda f.\, \lambda y.\, \mathsf{let}\ [x] = y\ \mathsf{in}\ f\ [x]\ [x]$. The precise type $\alpha \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]) \multimap [\alpha] \multimap [\delta]$ with subexponential annotations can then be computed using type inference.

2.5. On Circuits and Space Usage

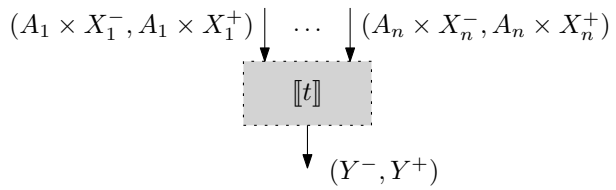
While we have defined INTML without reference to circuits, the application of INTML to sublinear space computation makes essential use of a translation of interactive terms to circuits. Here we give a brief overview of how the interactive language, circuits and the space usage analysis are related. The details are then developed in the rest of the paper.

In relation to message passing circuits, interactive types can be understood as the interfaces of entities that receive and send messages. To each interactive type X we have assigned two base language types X^- and X^+ that represent the values that may be received and sent respectively.

Interactive terms themselves represent circuits. A term of type

$$\Gamma \mid x_1 : A_1 \cdot X_1, \dots, x_n : A_n \cdot X_n \vdash t : Y$$

represents a circuit $[\![t]\!]$ with the following interface.



The circuit nodes in $[\![t]\!]$ are implemented by base language terms, which may contain the variables from Γ. If the interactive term t is closed, then the circuit has no incoming wires and a single outgoing wire.

Let us explain on an example how we get sublinear space bounds from the circuit-based evaluation of INTML. We write a simple INTML-function that computes the bitwise exclusive-or ($\oplus$) of bit vectors (of the same length). We do so in a way that the space usage of the program is sublinear in the length of the vectors.

The base language type $2 = 1 + 1$ can be seen as a type of bits, where $\mathsf{inl}(*)$ stands for 0 and $\mathsf{inr}(*)$ stands for 1. Of course, on individual bits the boolean function $\oplus$ can be programmed easily as a base language term $x\colon 2, y\colon 2 \vdash \mathsf{xor}(x, y) : 2$ that satisfies $\mathsf{xor}(x, y)[a/x, b/y] \mapsto^* a \oplus b$ for all (encodings of) bits a and b.

Let now 2^n be the base language type defined by

$$\underbrace{2 \times 2 \times \ldots \times 2}_{n \text{ times}} \ .$$

This type represents n-bit unsigned integers with a standard binary encoding. Observe how there are exponentially many closed values of 2^n, each having size linear in n. This is *not*, however, a good way to represent bit vectors when we are interested in sublinear space bounds. For this purpose we would use the interactive functional type

$$A \cdot [2^n] \multimap [2] \ .$$

In other words, a bit vector is encoded as a function that maps a position to the bit in this position. Observe how this type has room for bit vectors of length 2^n.

A closed term of type $A \cdot [2^n] \multimap [2]$ will correspond to a circuit with a single outgoing wire with interface (X^-, X^+), in which X^- and X^+ are base language types:

$$\begin{array}{rcccccc} X & = & A & \cdot & [2^n] & \multimap & [2] \\ X^- & = & A & \times & 2^n & + & 1 \\ X^+ & = & A & \times & 1 & + & 2 \end{array}$$

The values of type X^- are the messages that we can send to the circuit, the values of type X^+ are the possible responses. A typical dialogue with this circuit will be as follows: We send it the message $\mathsf{inr}(*)\colon X^-$ ('I would like to know the value of the thunk of type [2] that the function returns.'). The circuit would then typically reply with $\mathsf{inl}(a, *)\colon X^+$ ('I need to know the argument of the function, i.e. the value of the thunk of type $[2^n]$. When you answer this request, please also remind me of value a.'). Suppose we want to know the character in position $p\colon 2^n$. We then send the message $\mathsf{inl}(a, p)\colon X^-$ ('The argument thunk has value p. And you asked me to remind you of value a.'). To this, the circuit will typically reply $\mathsf{inr}(b)\colon X^+$ ('The thunk returned by the function has value b.').

Notice that in X^- and X^+ the summands for requests and answers line up with the interactive type in X. For instance, $\mathsf{inr}(*)\colon X^-$ and $\mathsf{inr}(b)\colon X^+$ both pertain [2].

It should be clear that any bit vector can be reconstructed from its representation as a function of type $A \cdot [2^n] \multimap [2]$ by repeatedly querying the circuit. Notice that the messages that we send to and receive from the circuit are all base language values of length proportional to n. We will see that this is also the case for the messages exchanged internally in the circuit. The space needed for messages is thus logarithmic in the length of the bit vectors that are represented, which is 2^n. It is this basic observation that leads to logarithmic space bounds.

The exclusive-or of two bit vectors can be implemented as an interactive program of type

$$(\alpha \cdot [2^n] \multimap [2]) \multimap (\beta \cdot [2^n] \multimap [2]) \multimap (\alpha + \beta) \cdot [2^n] \multimap [2] \ .$$

Let us spell out the type: given a bit vector of length 2^n (which uses its argument α times) and another bit vector of length 2^n (which uses its argument β times), the term produces a third bit vector of the same length (which uses its

argument $\alpha + \beta$ times). Here is our term:

$$\lambda x.\, \lambda y.\, \lambda z.\, \mathsf{copy}\ z\ \mathsf{as}\ z_1, z_2\ \mathsf{in}\ \mathsf{let}\ [u_1] = x\ z_1\ \mathsf{in}\ \mathsf{let}\ [u_2] = y\ z_2\ \mathsf{in}\ [\mathsf{xor}(u_1, u_2)].$$

This program manipulates bit vectors of length 2^n. Its circuit, however, works by passing only messages of size *proportional* to n, just as outlined for type $A \cdot [2^n] \multimap [2]$ above.

Interestingly, there are other ways of writing the same function. In the above program the thunk z will typically be evaluated twice. It is also possible to evaluate it only once up front and keep its value in memory:

$$\lambda x.\, \lambda y.\, \lambda z.\, \mathsf{let}\ [v] = z\ \mathsf{in}\ \mathsf{let}\ [u_1] = x\ [v]\ \mathsf{in}\ \mathsf{let}\ [u_2] = y\ [v]\ \mathsf{in}\ [\mathsf{xor}(u_1, u_2)].$$

The type reflects the changed memory behaviour:

$$2^n \cdot (\alpha \cdot [2^n] \multimap [2]) \multimap 2^n \cdot (\beta \cdot [2^n] \multimap [2]) \multimap \gamma \cdot [2^n] \multimap [2].$$

3. Compiling Interactive Terms to Base Terms

Now we explain in detail how closed interactive terms are compiled down to base language terms, so that whole INTML-programs can be evaluated using base language reduction alone.

The compilation works by interpreting interactive terms as message passing circuits and then implementing these circuits by base language terms.

3.1. Circuits

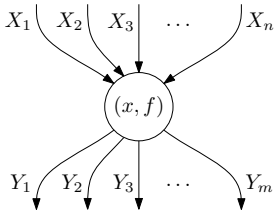
We start by defining the message passing circuits that serve as an intermediate language for the compilation. These circuits implement bidirectional message passing, as explained in the Introduction. Circuits may also be understood as a particular instance of string diagrams for monoidal categories [21]. They are also related to proof-nets, see [21] for a discussion.

Circuits are directed graphs that represent message passing networks. The edges are communication channels along which base language values can be passed. Each edge is labelled with a pair (X^-, X^+) of base language types. This label specifies that values of type X^+ may be passed from source to target of the edge, while messages of type X^- may be passed in the opposite direction.

The nodes of a circuit implement the processing and passing on of messages. At any time, only a single message travels in a circuit. When it arrives at a node, it is processed there and as a result a new message may be sent from this node to continue the message passing process. To define the behaviour of the nodes of a circuit, each node is labelled with a base language term that is used to process incoming messages. When a base language value arrives as a message along some edge, the term is used to compute a response, which is then passed on over some edge.

For the formal definition of circuits, we need the notion of a *locally ordered graph*. A 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 incoming edges to v and on the set $(\{v\} \times V) \cap E$ of outgoing edges from v. The local ordering is used to distinguish edges with the same label, for example in nodes like $\otimes E$ below.

Definition 4. A *circuit over* Γ is a locally ordered directed graph in which each node is labelled with a pair (x, f) of a base language term f and a variable x and each edge is labelled with a pair of base language types. This labelling must be such that if any node is labelled with (x, f), its incoming edges are labelled with $X_1, X_2, \ldots, X_n$ and its outgoing edges are labelled with $Y_1, Y_2, \ldots, Y_m$,



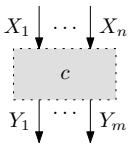
then the following base language typing judgement must be derivable:

$$\Gamma,\ x\colon Y_1^- + \cdots + Y_m^- + X_1^+ + \cdots + X_n^+ \ \vdash\ f\ :\ Y_1^+ + \cdots + Y_m^+ + X_1^- + \cdots + X_n^-$$

Here, X^- and X^+ denote the first and second component of the pair X respectively.

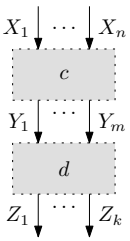
For each interactive type X, we have defined a pair of base language types X^- and X^+ in Section 2.2. We will use an interactive type X to stand for the edge label (X^-, X^+).

It is useful to allow circuits to have a number of input and output ports. We formalise these 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 $c\colon [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ for such a circuit c with input edges of type $X_1, \ldots, X_n$ and output edges of type $Y_1, \ldots, Y_m$ (note that they are ordered because of the local ordering for source and sink). We usually draw c as shown below.



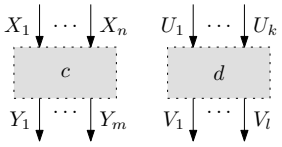
Given two circuits $c\colon [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ and $d\colon [Y_1, \ldots, Y_m] \to [Z_1, \ldots, Z_k]$, their sequential composition $d \circ c\colon [X_1, \ldots, X_n] \to [Z_1, \ldots, Z_k]$ is defined as depicted below: the i-th incoming edge to the sink of c and the i-th outgoing edge from the source of d are joined to a single edge. The sink of c and the source of d are removed.

$d \circ c =$

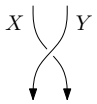


Given $c\colon [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ and $d\colon [U_1, \ldots, U_k] \to [V_1, \ldots, V_l]$, we write $c \otimes d$ for the circuit obtained by putting c and d in parallel.

$c \otimes d =$



Notice that in a composition the local ordering on the intermediate edges (the ones labelled above with $Y_1, \ldots Y_m$) disappears. For example, if we let $swap_{X,Y}\colon [X, Y] \to [Y, X]$ be the circuit



,

then $swap_{Y,X} \circ swap_{X,Y}$ is $id_X \otimes id_Y\colon [X, Y] \to [X, Y]$, where $id_X\colon [X] \to [X]$ is a single edge from input to output with label X.

3.2. Message Passing

Let us now make precise the message passing in circuits. Write $\mathcal{V}_A$ for the set of all closed base language values of type A. Write $\mathcal{E}_\Gamma$ for the set of Γ-*environments* consisting of all functions that map the variables in Γ to closed values of

their declared types. Given $\sigma \in \mathcal{E}_\Gamma$ and a term $\Gamma \vdash f : A$, we write $f[\sigma]$ for the closed term obtained by simultaneous substitution with the assignments in σ.

The set $M_{\Gamma,X}$ of messages that can be passed along an edge labelled with X in a circuit over Γ is defined by $M_{\Gamma,X} = \mathcal{E}_\Gamma \times (\mathcal{V}_{X^+} \times \{+\} \cup \mathcal{V}_{X^-} \times \{-\})$. A message $w \in M_{\Gamma,X}$ 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 [22].

To define message passing for a circuit c on Γ, let the set M_c of messages on c consist of all pairs (e, w) of an edge e in c and a message $w \in M_{\Gamma,X(e)}$, where $X(e)$ is the label of e. First, we define how each node in c locally reacts to arriving messages. For each node v we define a partial function $\chi_v \colon M_c \to M_c$ as follows: Let (x, f) be the label of v and let $i_1, \ldots, i_n$ and $o_1, \ldots, o_m$ be the incoming and outgoing edges connected to v respectively. Recall that f must by definition have type

$$\Gamma,\, x \colon X(o_1)^- + \cdots + X(o_m)^- + X(i_1)^+ + \cdots + X(i_n)^+ \vdash \\ f \colon X(o_1)^+ + \cdots + X(o_m)^+ + X(i_1)^- + \cdots + X(i_n)^- \ .$$

The function χ_v is then defined by

$$\chi_v(o_k, (\sigma, w, -)) = \begin{cases} (o_{k'}, (\sigma, w', +)) & \text{if } f[\sigma][\mathsf{in}_k(w)/x] \mapsto^* \mathsf{in}_{k'}(w') \text{ and } k' \leq m, \\ (i_{k'-m}, (\sigma, w', -)) & \text{if } f[\sigma][\mathsf{in}_k(w)/x] \mapsto^* \mathsf{in}_{k'}(w') \text{ and } k' > m, \end{cases}$$

$$\chi_v(i_k, (\sigma, w, +)) = \begin{cases} (o_{k'}, (\sigma, w', +)) & \text{if } f[\sigma][\mathsf{in}_{m+k}(w)/x] \mapsto^* \mathsf{in}_{k'}(w') \text{ and } k' \leq m, \\ (i_{k'-m}, (\sigma, w', -)) & \text{if } f[\sigma][\mathsf{in}_{m+k}(w)/x] \mapsto^* \mathsf{in}_{k'}(w') \text{ and } k' > m, \end{cases}$$

where we write in_k as a short-hand for the injection into the k-th summand of a sum type.

The message passing behaviour of the circuit as a whole is the partial function $\chi_c \colon M_c \to M_c$ defined by repeatedly applying the local functions χ_v for all nodes v:

$$\chi_c = Tr\left(\bigcup\nolimits_{v \in V(c)} \chi_v\right) \qquad Tr(f)(w) = \begin{cases} Tr(f)(f(w)) & \text{if } f(w) \text{ defined}, \\ w & \text{if } f(w) \text{ undefined}. \end{cases}$$

It is not hard to see that message passing cannot end at an internal edge, i.e. $\chi_c(e, w) = (e', w')$ implies that e' is an input or an output edge.

In fact, we will forget about the internal structure of c and consider just the restriction of χ_c to messages on input or output edges of c. For a circuit $c \colon [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$, this restriction yields a partial function $\hat{\chi}_c$ of type

$$\mathcal{E}_\Gamma \times \mathcal{V}_{Y_1^- + \cdots + Y_m^- + X_1^+ + \cdots + X_n^+} \longrightarrow \mathcal{V}_{Y_1^+ + \cdots + Y_m^+ + X_1^- + \cdots + X_n^-} \ .$$

We call $\hat{\chi}_c$ *the behaviour of* c and consider circuits with the same behaviour to be equal.

Definition 5 (Equality of Circuits)**.** Two circuits c and d of type $[X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ over Γ are *equal* if their behaviour is the same, i.e. if $\hat{\chi}_c = \hat{\chi}_d$ holds. We write $c = d$ for equality of circuits.

3.3. Implementing Message Passing

The message passing behaviour of any circuit can be implemented by a base language term. For any circuit $c \colon [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ we can construct a base language term $\hat{c}$ that implements the behaviour $\hat{\chi}_c$ in the following sense. Its type is

$$\Gamma,\, x \colon Y_1^- + \cdots + Y_m^- + X_1^+ + \cdots + X_n^+ \vdash \hat{c} : Y_1^+ + \cdots + Y_m^+ + X_1^- + \cdots + X_n^-$$

and $\hat{\chi}_c(\sigma, v) = w$ holds if and only if $\hat{c}[\sigma][v/x] \mapsto^* w$ does.

In essence, the construction of $\hat{c}$ works in the same way as the definition of $\hat{\chi}_c$ above. First note that we can represent the set of messages $M_{\Gamma,X}$ by the base language type $(A_1 \times \cdots \times A_n) \times (X^- + X^+)$, where Γ is $x_1 \colon A_1, \ldots, x_n \colon A_n$. Then, M_c can be represented in the base language by a big sum type and the function $\bigcup_{v \in V(c)} \chi_v \colon M_c \to M_c$ can be implemented easily by a big case distinction. Using a single loop one can construct from this a base language term that implements $\hat{\chi}_c$.

3.4. Constructing Circuits

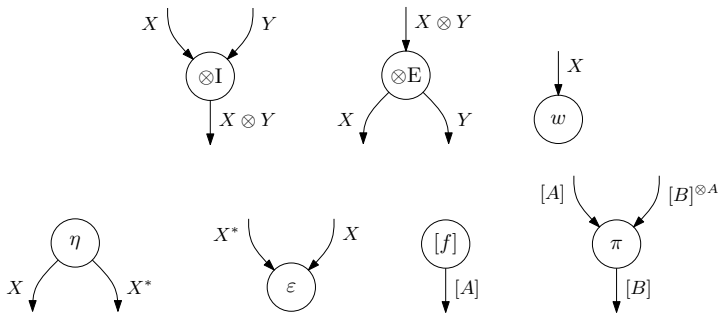
To organise the compilation of interactive terms to circuits and the soundness proof, we next identify some structure of circuits.

First we observe that sequential and parallel composition have the expected properties. Recall that the equalities in these lemmas mean that the circuits have the same message-passing behaviour, cf. Definition 5.

Lemma 1. *The equations $c = c \circ id = id \circ c$ and $(c \circ d) \circ e = c \circ (d \circ e)$ hold for all circuits c, d and e for which the terms in these equations are defined.*

Lemma 2. *The equations $id \otimes id = id$ and $(c \otimes d) \circ (c' \otimes d') = (c \circ c') \otimes (d \circ d')$ hold for all circuits c, c', d and d' for which the terms in these equations are defined.*

For the construction of circuits, we use a set of predefined nodes. We define the following nodes



over Γ for any X, Y, A and B and all f with $\Gamma \vdash f : A$. Rather than spelling out the base language terms for these nodes explicitly, we just specify their (very simple) message-passing behaviour and note that it is easily implemented. The behaviour of the above nodes is specified as the least functions satisfying the following equations. In these equations we write i_1 and i_2 for the incoming edges of a node (ordered from left to right) and likewise o_1 and o_2 for the outgoing edges (in the same order). There is no equation for χ_w, as this must be the empty function.

$$\chi_{\otimes I}(i_1, (\sigma, v, +)) = (o_1, (\sigma, \mathsf{inl}(v), +)) \qquad \chi_{\otimes E}(i_1, (\sigma, \mathsf{inl}(v), +)) = (o_1, (\sigma, v, +))$$
$$\chi_{\otimes I}(i_2, (\sigma, v, +)) = (o_1, (\sigma, \mathsf{inr}(v), +)) \qquad \chi_{\otimes E}(i_1, (\sigma, \mathsf{inr}(v), +)) = (o_2, (\sigma, v, +))$$
$$\chi_{\otimes I}(o_1, (\sigma, \mathsf{inl}(v), -)) = (i_1, (\sigma, v, -)) \qquad \chi_{\otimes E}(o_1, (\sigma, v, -)) = (i_1, (\sigma, \mathsf{inl}(v), -))$$
$$\chi_{\otimes I}(o_1, (\sigma, \mathsf{inr}(v), -)) = (i_2, (\sigma, v, -)) \qquad \chi_{\otimes E}(o_2, (\sigma, v, -)) = (i_1, (\sigma, \mathsf{inr}(v), -))$$

$$\chi_{\eta}(o_1, (\sigma, v, -)) = (o_2, (\sigma, v, +)) \qquad \chi_{\varepsilon}(i_1, (\sigma, v, +)) = (i_2, (\sigma, v, -))$$
$$\chi_{\eta}(o_2, (\sigma, v, -)) = (o_1, (\sigma, v, +)) \qquad \chi_{\varepsilon}(i_2, (\sigma, v, +)) = (i_1, (\sigma, v, -))$$

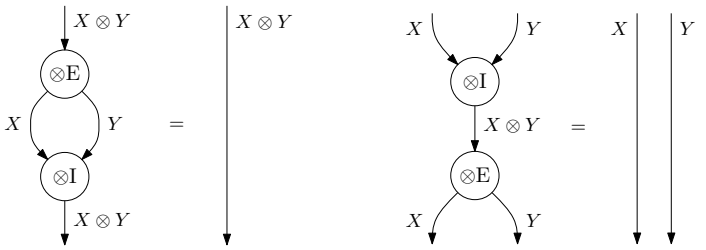
$$\chi_{[f]}(o_1, (\sigma, *, -)) = (o_1, (\sigma, v, +)) \text{ if } f[\sigma] \mapsto^* v \text{ and } v \text{ is a value}$$

$$\chi_{\pi}(i_1, (\sigma, v, +)) = (i_2, (\sigma, (v, *), -))$$
$$\chi_{\pi}(i_2, (\sigma, (v, w), +)) = (o_1, (\sigma, w, +))$$
$$\chi_{\pi}(o_1, (\sigma, *, -)) = (i_1, (\sigma, *, -))$$

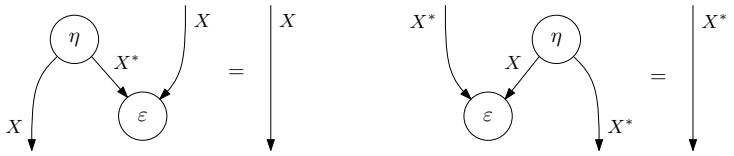
Informally, the nodes ⊗I and ⊗E serve for packing two message passing wires into one. Node w discards all messages; it is used for weakening. Nodes η and ε serve to exchange the direction of messages; we use them to implement functions. Node π is useful for sequencing thunks. If a question arrives at its output, this node queries its left input. If an answer v arrives from this input, node π queries its second input and passes the value v along with the query. The second input thus gets access to the result of the first input.

The following lemmas are immediate.

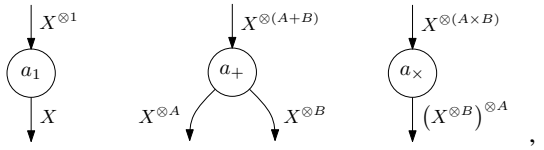
Lemma 3.



Lemma 4.

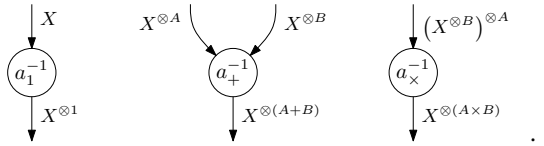


In addition to the general nodes defined above, we also need *administrative nodes* to deal with the subexponentials. A subexponential can be understood as an iterated multiplication. We have administrative nodes that correspond to the familiar high-school identities for exponentiation $x^1 = x$, $x^{a+b} = x^a \cdot x^b$ and $x^{a \cdot b} = (x^b)^a$



,

along with their inverses



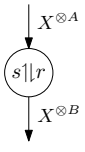
.

The behaviour of these nodes is given by the canonical terms of the respective type. We spell out the behaviour of a_+ and $a_\times$:

$$\begin{aligned}
\chi_{a_+}(i_1, (\sigma, (\mathsf{inl}(v), w), +)) &= (o_1, (\sigma, (v, w), +)) \\
\chi_{a_+}(i_1, (\sigma, (\mathsf{inr}(v), w), +)) &= (o_2, (\sigma, (v, w), +)) \\
\chi_{a_+}(o_1, (\sigma, (v, w), -)) &= (i_1, (\sigma, (\mathsf{inl}(v), w), -)) \\
\chi_{a_+}(o_2, (\sigma, (v, w), -)) &= (i_1, (\sigma, (\mathsf{inr}(v), w), -)) \\
\chi_{a_\times}(i_1, (\sigma, ((u, v), w), +)) &= (o_1, (\sigma, (u, (v, w)), +)) \\
\chi_{a_\times}(o_1, (\sigma, (u, (v, w)), -)) &= (i_1, (\sigma, ((u, v), w), -))
\end{aligned}$$

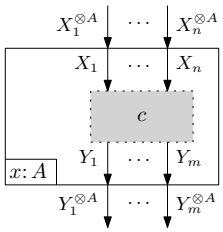
For the reader familiar with Linear Logic, nodes a_1, a_+ and $a_\times$ play a role similar to dereliction, contraction and digging nodes (respectively) in proof-nets for the exponentials.

For the manipulation of subexponentials, we use a reindexing node. For any two terms $\Gamma, b\colon B \vdash s : A$ and $\Gamma, a\colon A \vdash r : B$ we have a node



over Γ. This nodes modifies just the index in the subexponential. When a message travels from top to bottom, r is applied to the index; in the other direction s is applied. We shall use the reindexing node in particular to interpret rule Struct. In this case s and r will in fact be a section-retraction pair.

Finally, we use a box construction that allows us to bind base language variables in circuits. Given a circuit $c\colon [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ over $(\Gamma,\ x\colon A)$, we define a circuit $c^{\otimes x:A}\colon [{X_1}^{\otimes A}, \ldots, {X_n}^{\otimes A}] \to [{Y_1}^{\otimes A}, \ldots, {Y_m}^{\otimes A}]$ over Γ, which we depict graphically as follows:



Informally, when a value (v, w) arrives from the outside to the box, the value v is bound to the variable x and the message w is passed into the box. If a message w' reaches the border from inside, then the value v is retrieved again by looking up the variable x in the environment and then (v, w') travels out from the box. The value of x does not change inside the box.

More formally, the behaviour of $c^{\otimes x:A}$ can be specfied as follows. Let $e_1, \ldots, e_{m+n}$ be the list of input and output edges of $c^{\otimes x:A}$, in which first all the input edges appear in their order and then the output edges. Let $e'_1, \ldots, e'_{m+n}$ be the input and output edges of c, listed in the same way. The behaviour of $c^{\otimes x:A}$ is then specified by: $\hat{\chi}_{(c^{\otimes x:A})}(e_i, (\sigma, (v, u), p)) = (e_j, (\rho, (v, w), r))$ if and only if $\hat{\chi}_c(e'_i, (\sigma[v/x], u, p)) = (e'_j, (\tau, w, r))$ and ρ is the restriction of τ to variables appearing in Γ. Concretely, such a circuit $c^{\otimes x:A}$ can be constructed simply as a single node, obtained from implementation of the behaviour $\hat{\chi}_c$.

The variable x should be considered being bound in the above box; we have the usual α-conversion law $c^{\otimes x:A} = c[y/x]^{\otimes y:A}$ for any fresh variable y. Here, $c[y/x]$ denotes substitution of y for x in each node in c. We write $c^{\otimes A}$ instead of $c^{\otimes x:A}$ if x does not appear free in any of the nodes in c.

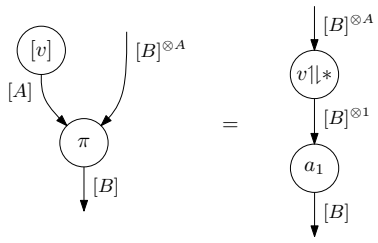
Boxes are easily seen to have the following properties:

Lemma 5. *The equations $id^{\otimes A} = id$, $(c \circ d)^{\otimes x:A} = c^{\otimes x:A} \circ d^{\otimes x:A}$, $(c \otimes d)^{\otimes x:A} = c^{\otimes x:A} \otimes d^{\otimes x:A}$ hold for all circuits c and d, all variables x and all types A for which the circuits in the equations are well-defined.*

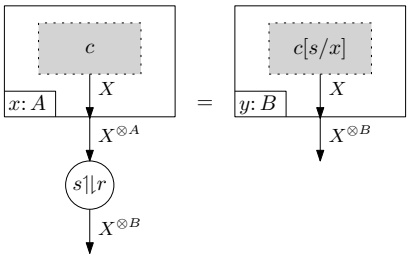
Having introduced all the building blocks for the construction of circuits, we state a few results that allow us to prove soundness of the translation from the interactive to the base language.

Lemma 6. *For any circuit c with one outgoing edge and no incoming edges, we have $a_1 \circ c^{\otimes 1} = c$ and $a_+ \circ c^{\otimes(A+B)} = c^{\otimes A} \otimes c^{\otimes B}$ and $a_\times \circ c^{\otimes(A\times B)} = \left(c^{\otimes B}\right)^{\otimes A}$.*

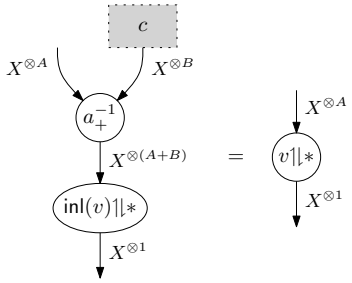
Lemma 7. *For any value $\vdash v : A$, the following equality of circuits holds.*



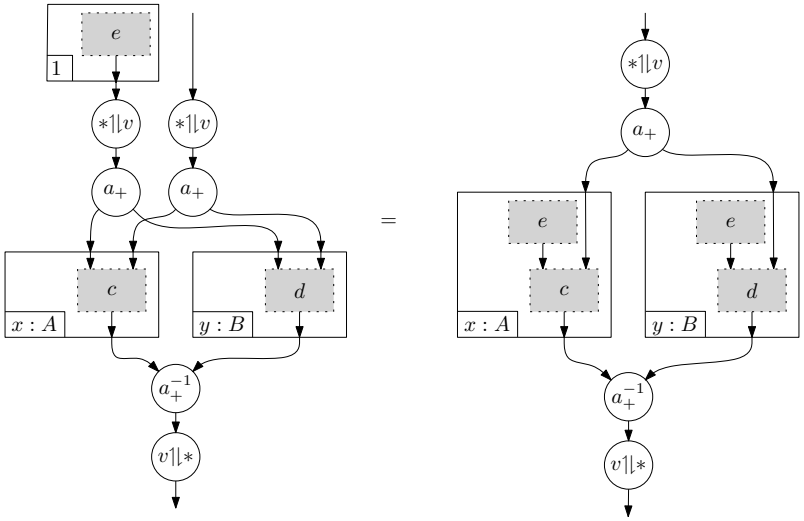
Lemma 8. *Let $\Gamma,\ y\colon B \vdash s : A$ and $\Gamma,\ x\colon A \vdash r : B$ be a section-retraction pair. For any circuit c over $\Gamma,\ x\colon A$ with no inputs and a single output labelled X we have:*



Lemma 9. *For any circuit* $c\colon [\,] \to [X^{\otimes B}]$ *and any value* $\vdash v : A$*, the following equality of circuits holds.*



Lemma 10. *For all circuits* $c\colon [X, Y] \to [Z]$ *over* Γ*,* $x\colon A$ *and* $d\colon [X, Y] \to [Z]$ *over* Γ*,* $y\colon B$ *and* $e\colon [\,] \to X$ *over* Γ*, and all values* $\Gamma \vdash v : A + B$*, the following is true:*



Proof. Consider what form a message can have that leaves the box containing c over an incoming edge. Because all input and output edges of the circuit are guarded by $(*\rangle|\langle v)$ and $(v\rangle|\langle *)$, such a message must have the form $(\sigma, (w_1, w_2), -)$ where $\mathsf{inl}(w_1) = v[\sigma]$. In this case, the message $(\sigma, w_2, -)$ will be passed to the circuit e. An answer $(\sigma, w_3, +)$ from e will result in $(\sigma, (w_1, w_2), +)$ being passed into the box containing c. This process can be shortcut by moving e inside the box containing c. A similar argument applies to the box containing d. □

3.5. Translating Interactive Terms to Circuits

We now interpret interactive terms by circuits. To each derivation δ ending with sequent $\Gamma \mid x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n \vdash s : Y$ we assign a circuit $[\![\delta]\!]\colon [X_1{}^{\otimes A_1}, \ldots, X_n{}^{\otimes A_n}] \to [Y]$ on Γ. Each variable declaration in the interactive context thus becomes an input wire of the translated circuit.

The definition of $[\![\delta]\!]$ goes by induction on derivations and is given in Fig. 11. Therein we use the following notation: We denote the premises of δ by δ_s and δ_t depending on the term in the premise. Given an interactive context

Ax	$[\![\delta]\!] = a_1$
Struct	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Phi \otimes (s\restriction r))$, where $\Gamma,\ x\colon A \vdash s : B$ and $\Gamma,\ y\colon B \vdash r : A$ are an *arbitrarily chosen* section-retraction pair witnessing $A \triangleleft B$.
Weak	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Phi \otimes w)$
Exch	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Phi \otimes swap \otimes id_\Psi)$
Copy	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Psi \otimes (a_+ \circ [\![\delta_s]\!]^{\otimes(A+B)} \circ (a_\times)_\Phi))$
[ ]I	$[\![\delta]\!] = [f]$
[ ]E	$[\![\delta]\!] = \pi \circ ([\![\delta_s]\!] \otimes ([\![\delta_t]\!]^{\otimes x:A} \circ (a_\times)_\Psi))$
$\otimes$I	$[\![\delta]\!] = \otimes I \circ ([\![\delta_s]\!] \otimes [\![\delta_t]\!])$
$\otimes$E	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Psi \otimes ([\![\delta_s]\!]^{\otimes A} \circ (a_\times)_\Phi))$
$\multimap$I	$[\![\delta]\!] = \otimes I \circ swap \circ ([\![\delta_t]\!] \otimes id_X) \circ (id_\Psi \otimes \eta)$
$\multimap$E	$[\![\delta]\!] = (id_Y \otimes \varepsilon) \circ ((swap \circ \otimes E \circ [\![\delta_t]\!]) \otimes ([\![\delta_s]\!]^{\otimes A} \circ (a_\times)_\Phi))$
0E^c	$[\![\delta]\!]$ is the single node with label $(x, \mathsf{image}(f))$.
+E^c	$[\![\delta]\!] = a_1 \circ (v\restriction *) \circ a_+^{-1} \circ ([\![\delta_s]\!]^{\otimes x:A} \otimes [\![\delta_t]\!]^{\otimes y:B}) \circ (a_+)_\Phi \circ (*\restriction v)_\Phi \circ (a_1^{-1})_\Phi$
Hack	$[\![\delta]\!]$ is the single node with label (x, f).

Figure 11: Interpretation of Interacitve Derivations

$\Phi = x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n$, we write Φ also for the list $[X_1{}^{\otimes A_1}, \ldots, X_n{}^{\otimes A_n}]$ of input/output wires of a circuit. We write id_Φ for $(id \otimes \cdots \otimes id)\colon \Phi \to \Phi$ and use similar notation for the evident circuits $(a_+)_\Phi\colon (A + B) \cdot \Phi \to (A \cdot \Phi, B \cdot \Phi)$, $(a_\times)_\Phi\colon (A \times B) \cdot \Phi \to A \cdot B \cdot \Phi$, etc.

To make it easier to read the interpretation of the rules, we spell out the more complicated cases in graphical form in Figs. 12–16. We give full details, including all edge labels, in the case of $\otimes$E in Fig. 12, and omit the edge labels otherwise.

For well-definedness of the interpretation, notice that the side condition on rule Struct ensures that it is always possible to choose a section-retraction pair.

As an example we show in Fig. 17 the circuit that is obtained by interpreting the canonical derivation of the typing judgment $\vdash \lambda f.\, \lambda y.\, \mathsf{let}\ [x] = y\ \mathsf{in}\ f\ [x]\ [x] : \alpha \cdot ([\alpha] \multimap [\alpha] \multimap [\beta]) \multimap [\alpha] \multimap [\beta]$.

3.6. Soundness

Having defined a translation from interactive terms to base language ones by translation to circuits and their implementation, it now remains to show soundness of the translation with respect to the reduction semantics from Section 2.2. We prove the following result:

Theorem 11 (Soundness)**.** *If δ derives $\cdot \mid \cdot \vdash s\colon X$ and $s \mapsto t$, then there exists a derivation ρ of $\cdot \mid \cdot \vdash t\colon X$ with $[\![\delta]\!] = [\![\rho]\!]$.*

For the proof of the soundness theorem we need substitution lemmas.

Lemma 12 (Substitution)**.** *If ρ derives $\Gamma \mid \Phi,\ x : A \cdot X,\ \Psi \vdash t : Y$ and δ derives $\Gamma \mid \cdot \vdash s : X$, then there exists a derivation $\rho[\delta/x]$ of $\Gamma \mid \Phi,\ \Psi \vdash t[s/x] : Y$ that satisfies $[\![\rho[\delta/x]]\!] = [\![\rho]\!] \circ (id_\Phi \otimes [\![\delta]\!]^{\otimes A} \otimes id_\Psi)$.*

Proof. The proof goes by induction on ρ. We consider representative cases:

- Ax: If ρ ends in an application of Ax and thus derives $x : 1 \cdot Y \vdash x : Y$, then we have $[\![\rho]\!] = a_1$. We let $\rho[\delta/x] = \delta$. We have $[\![\rho[\delta/x]]\!] = [\![\delta]\!] = a_1 \circ [\![\delta]\!]^{\otimes 1} = [\![\rho]\!] \circ [\![\delta]\!]^{\otimes 1}$, as required.
- Struct: If ρ ends with an application of Struct, two cases are possible:
  1. The context Ψ is empty and ρ has the form:

$$\textsc{Struct}\ \frac{\rho_1 : \Gamma \mid \Phi,\ x : B \cdot X \vdash t : Y}{\Gamma \mid \Phi,\ x : A \cdot X \vdash t : Y}\ B \triangleleft A$$

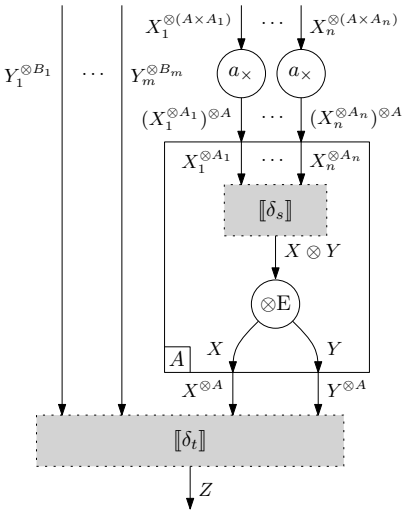


Figure 12: Circuit for ⊗E

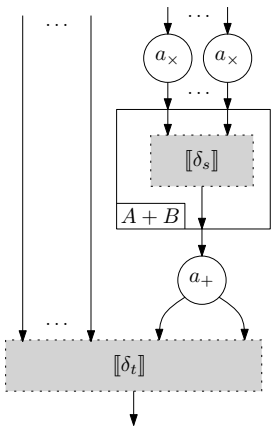


Figure 13: Circuit for Copy

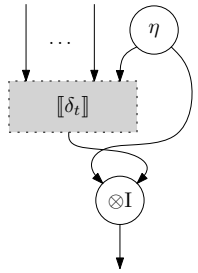


Figure 14: Circuit for ⊸I

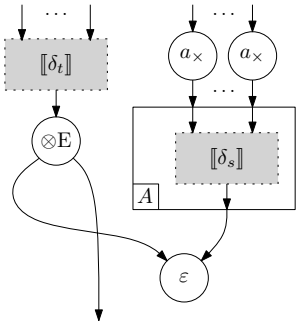


Figure 15: Circuit for ⊸E

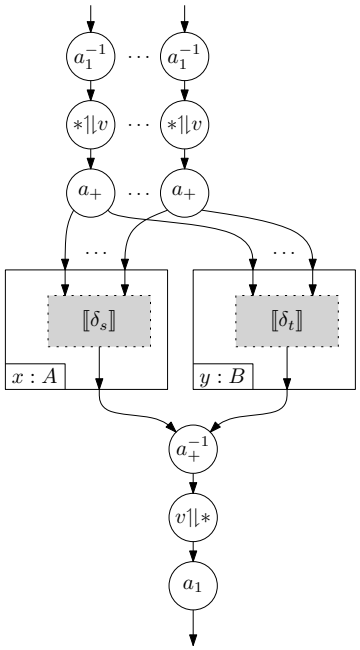


Figure 16: Circuit for +E$^{\mathsf{c}}$

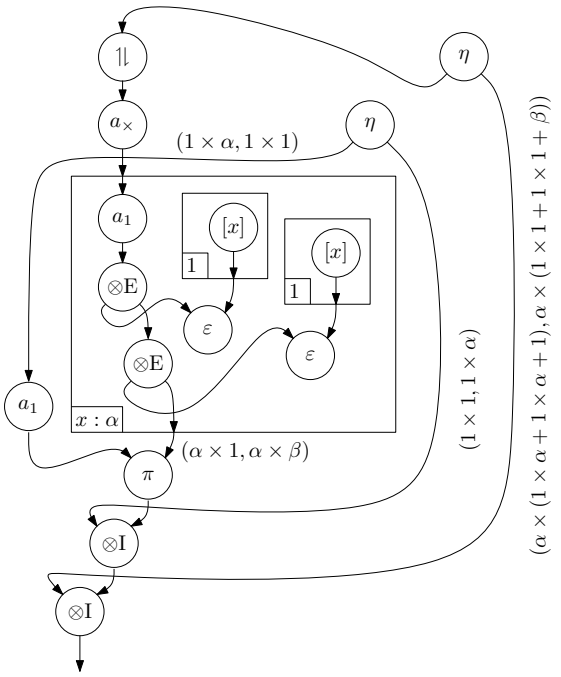


Figure 17: Example Circuit

In this case $[\![\rho]\!]$ is $[\![\rho_1]\!] \circ (id_\Phi \otimes (s\restriction r))$ for some section-retraction pair s and r from B to A. We let $\rho[\delta/x]$ be $\rho_1[\delta/x]$. We have:

$$\begin{aligned}
[\![\rho[\delta/x]]\!] &= [\![\rho_1[\delta/x]]\!] = [\![\rho_1]\!] \circ \left(id_\Phi \otimes [\![\delta]\!]^{\otimes A}\right) && \text{by defn. and i.h.} \\
&= [\![\rho_1]\!] \circ \left(id_\Phi \otimes \left((s\restriction r) \circ [\![\delta]\!]^{\otimes B}\right)\right) && \text{by Lemma 8} \\
&= [\![\rho_1]\!] \circ (id_\Phi \otimes (s\restriction r)) \circ \left(id_\Phi \otimes [\![\delta]\!]^{\otimes B}\right) \\
&= [\![\rho]\!] \circ \left(id_\Phi \otimes [\![\delta]\!]^{\otimes B}\right)
\end{aligned}$$

2. The context Ψ has the form $\Psi_1,\ y : D{\cdot}Z$ and ρ has the form:

$$\textsc{Struct}\ \frac{\rho_1 : \Gamma \mid \Phi,\ x : A{\cdot}X,\ \Psi_1,\ y : C{\cdot}Z \vdash t : Y}{\Gamma \mid \Phi,\ x : A{\cdot}X,\ \Psi_1,\ y : D{\cdot}Z \vdash t : Y}\ C \triangleleft D$$

In this case we let $\rho[\delta/x]$ be:

$$\textsc{Struct}\ \frac{\rho_1[\delta/x] : \Gamma \mid \Phi,\ \Psi_1,\ y : C{\cdot}Z \vdash t[s/x] : Y}{\Gamma \mid \Phi,\ \Psi_1,\ y : D{\cdot}Z \vdash t[s/x] : Y}\ C \triangleleft D$$

With this choice we have

$$\begin{aligned}
[\![\rho[\delta/x]]\!] &= [\![\rho_1[\delta/x]]\!] \circ (id_{(\Phi,\,\Psi_1)} \otimes (s\restriction r)) \\
&= [\![\rho_1]\!] \circ (id_\Phi \otimes [\![\delta]\!] \otimes id_{(\Psi_1,\,y:C\cdot Z)}) \circ (id_{(\Phi,\,\Psi_1)} \otimes (s\restriction r)) \\
&= [\![\rho_1]\!] \circ (id_{(\Phi,\,x:A\cdot X,\,\Psi_1)} \otimes (s\restriction r)) \circ (id_\Phi \otimes [\![\delta]\!] \otimes id_\Psi) \\
&= [\![\rho]\!] \circ (id_\Phi \otimes [\![\delta]\!] \otimes id_\Psi),
\end{aligned}$$

as was required to show.

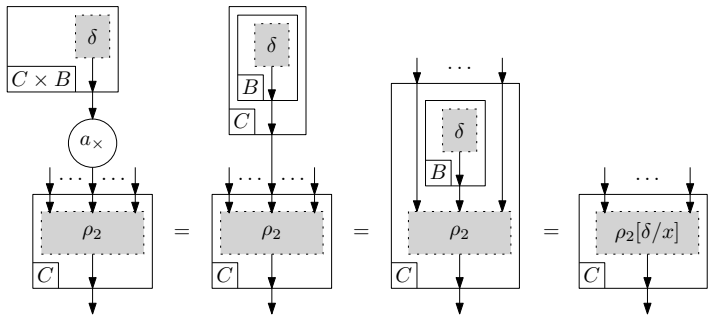
- The cases for the other structural rules follow by similar arguments.

- $\multimap$E: Suppose ρ ends in an application of rule $\multimap$E:

$$\multimap\text{E}\ \frac{\rho_1 : \Gamma \mid \Phi_1 \vdash t : C \cdot Z \multimap Y \qquad \rho_2 : \Gamma \mid \Phi_2 \vdash s : Z}{\Gamma \mid \Phi_1, C \cdot \Phi_2 \vdash t\ s : Y}$$

  The context $\Phi_1, C \cdot \Phi_2$ must be the same as Φ, $x : A \cdot X$, Ψ. If the declaration $x : A \cdot X$ appears in Φ_1, then the assertion follows directly by applying the induction hypothesis to ρ_1.

  Suppose then that $x : A \cdot X$ appears in $C \cdot \Phi_2$. In this case, $x : B \cdot X$ appears in Φ_2 for some B and we have $A = C \times B$. We define $\rho[\delta/x]$ to be the derivation obtained by applying $\multimap$E to ρ_1 and $\rho_2[\delta/x]$. We have



  by Lemmas 6 and 5 and the induction hypothesis. But now the leftmost circuit in the above equation appears in $[\![\rho]\!] \circ (id_\Phi \otimes [\![\delta]\!]^{\otimes A} \otimes id_\Psi)$. If we replace it by the rightmost circuit in the equation, then we obtain just $[\![\rho[\delta/x]]\!]$. We have thus shown the required equality.

- The cases for rules $\multimap$I, $\otimes$I, $\otimes$E and [ ]E follow by similar arguments.

- The rules [ ]-I, 0E and Hack cannot be the conclusion of ρ, as they all have an empty interactive context.

- If ρ ends with an application of $+\text{E}^{\text{c}}$

$$+\text{E}^{\text{c}}\ \frac{\Gamma \vdash f : B + C \qquad \rho_1 : \Gamma,\, x\colon B \mid \Theta \vdash s : Y \qquad \rho_2 : \Gamma,\, y\colon C \mid \Theta \vdash t : Y}{\Gamma \mid \Theta \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow s \mid \mathsf{inr}(y) \Rightarrow t : Y}$$

  then we use Lemma 10 to move $[\![\delta]\!]$ into the two inner boxes for ρ_1 and ρ_2, similarly to the case for $\multimap$E above. (Strictly speaking, we have formulated Lemma 10 only with one output and two input wires. It can be used if c and d has a number of input and output wires by packing all these wires into a single one, which is possible because of $(r\restriction s) \circ (\otimes I)^{\otimes A} = (\otimes I)^{\otimes B} \circ ((r\restriction s) \otimes (r\restriction s))$ and $(\otimes E)^{\otimes B} \circ (r\restriction s) = ((r\restriction s) \otimes (r\restriction s)) \circ (\otimes E)^{\otimes A}$.). Then, by induction hypothesis, the box contents are $\rho_1[\delta/x]$ and $\rho_2[\delta/x]$ respectively, which completes this case.

□

Lemma 13. *Suppose ρ derives Γ, $x\colon A \mid \Phi \vdash t : Y$ and v is a closed value of type A. Then there exists a derivation $\rho[v/x]$ of $\Gamma \mid \Phi \vdash t[v/x] : Y$ with $[\![\rho[v/c]]\!] = [\![\rho]\!][v/x]$.*

Proof. A straightforward induction on ρ. □

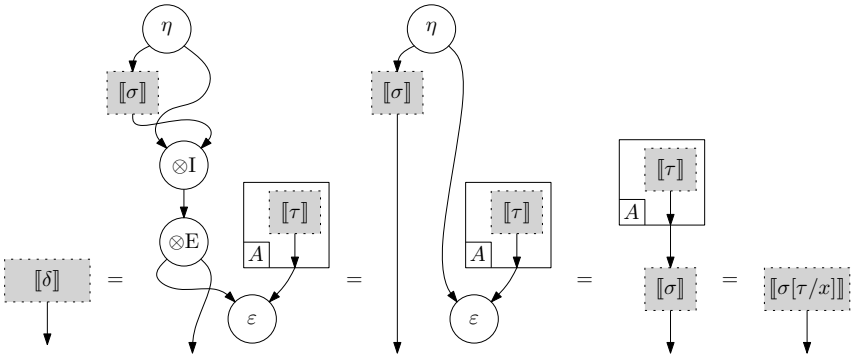
With these substitution lemmas, we can now prove the Soundness Theorem:

Proof of Theorem 11. For any reduction $s \mapsto t$ there exist decompositions $s = E[s']$ and $t = E[t']$ such that $s' \to t'$ is an instance of one of the basic reductions from Fig. 10.

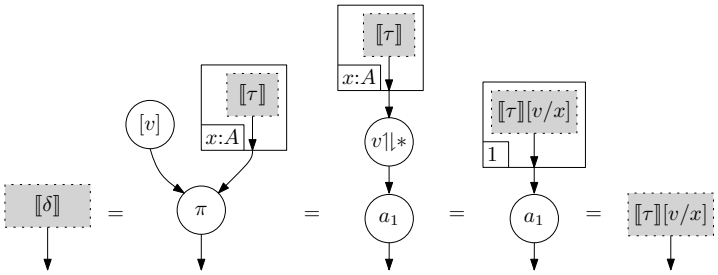
The proof then goes by induction on the structure of E. The base case, where E is empty, amounts to showing the assertion for the basic reductions. We spell out representative cases:

- $s \to t$ is $(\lambda x.\, q)\ p \to q[p/x]$.

  Since the derivation δ of s ends in a sequent with an 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 the premises $x : B \cdot Z \vdash q : Y$ and $\cdot \mid \cdot \vdash p : Z$ of these rules. Then we use Lemmas 3 and 4 and the substitution lemma to calculate:

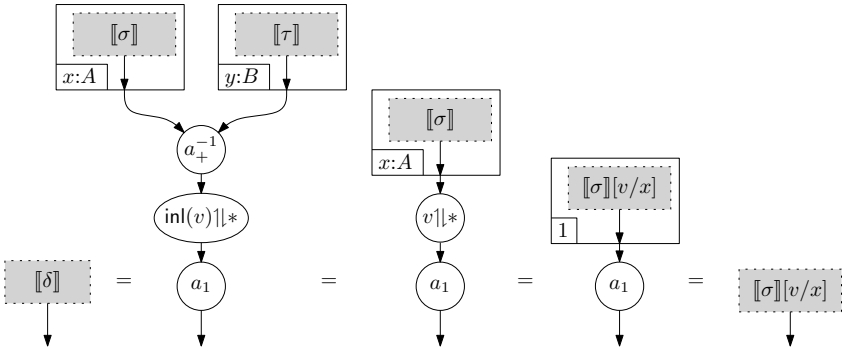


- $s \to t$ is $(\mathsf{let}\ [x] = [v]\ \mathsf{in}\ p) \to p[v/x]$, where v is a closed value. Using Lemmas 7, 8 and 6 we calculate:



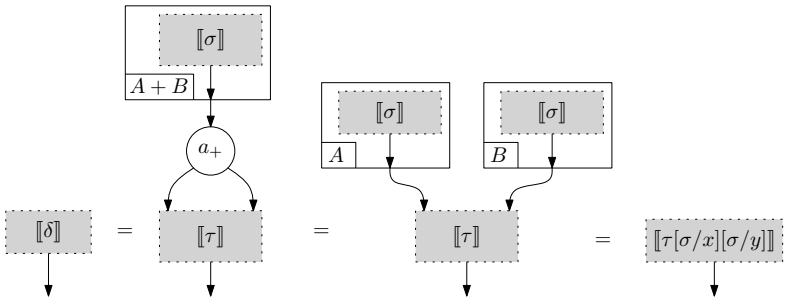
  Letting ρ be $\tau[v/x]$ and using Lemma 13 completes this case.

- $s \to t$ is $(\mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow p \mid \mathsf{inr}(y) \Rightarrow q) \to p[v/x]$, where v is a closed value. Using Lemmas 9, 6 and 8 we calculate:



  Again using Lemma 13, we can complete this case by letting ρ be $\sigma[v/x]$.

- $s \to t$ is $(\mathsf{copy}\ p\ \mathsf{as}\ x, y\ \mathsf{in}\ q) \to q[p/x][p/y]$. In this case we use Lemma 6 and the substitution lemma to show that $\rho = \tau[\sigma/x][\sigma/y]$ is a suitable choice.



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 $\cdot \mid \cdot \vdash F[s']\colon Y$ and $\cdot \mid \cdot \vdash u\colon Z$, 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 because we have $[\![\delta]\!] = \otimes I \circ ([\![\sigma]\!] \otimes [\![\tau]\!]) = \otimes I \circ ([\![\sigma']\!] \otimes [\![\tau]\!]) = [\![\rho]\!]$. □

Corollary 14. *For any interactive term t and any base language term f there exists a base language term* (let $[x] = t$ in f) *such that the typing rule*

$$[\,]\mathrm{E}^{\mathsf{c}}\ \frac{\Gamma \mid \cdot \vdash t : [A] \qquad \Gamma,\, x\colon A \vdash f : B}{\Gamma \vdash \mathsf{let}\ [x] = t\ \mathsf{in}\ f : B}$$

is admissible and there are reductions

$$\mathsf{let}\ [x] = [v]\ \mathsf{in}\ f \ \mapsto^* \ f[v/x] \ \textit{ if } v \textit{ is a value}\ ,$$
$$\mathsf{let}\ [x] = s\ \mathsf{in}\ f \ \mapsto^* \ \mathsf{let}\ [x] = t\ \mathsf{in}\ f \ \textit{ if } s \mapsto t\ .$$

Proof. Given $\Gamma \mid \cdot \vdash t : [A]$, the term $\hat{t}$ implementing the message passing circuit $[\![t]\!]$ has type $\Gamma,\, x\colon 1 \vdash \hat{t} : A$. We can then define the term (let $[x] = t$ in f) to be (let $y = \hat{t}[*/x]$ in f), where we write (let $z = g$ in h) as an abbreviation for (let $z = g$ loop inl(h)). □

3.7. Relation to the Int-Construction

At the beginning of this article we have outlined how INTML was derived from the categorical structure obtained by applying the Int construction to a term model of a basic functional language. We have spelled out the details of INTML without reference to this categorical structure, as we believe this presentation to be more broadly accessible. Nevertheless, we would like to explain how INTML and the compilation from interactive to base terms can be understood as a model construction with the Int construction.

In this section we explain in which sense the circuits defined earlier can be understood as morphisms in a category Int($\mathbb{C}$) obtained by Int construction from a suitable term model $\mathbb{C}$. The main difficulty with this is that circuits may contain free variables and so must appear in a suitably indexed variant of the Int construction, rather than the standard one. In this section we show how to capture this indexing in a way that remains close to the original Int construction.

We start by organising the base language into a term model of computations indexed by (complex) values. This structure is derived from the structure of the Enriched Effect Calculus [23] and that of Call-by-Push-Value [24], see [9].

3.7.1. Complex Values

Definition 6. A *complex value* is any base language term that can be formed by the following grammar:

$$f, g \ ::= \ x \ \mid\ * \ \mid\ \langle f, g\rangle \ \mid\ \mathsf{fst}(f) \ \mid\ \mathsf{snd}(f) \ \mid\ \mathsf{image}(f) \ \mid\ \mathsf{inl}(f) \ \mid\ \mathsf{inr}(g)$$
$$\mid\ \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g \mid \mathsf{inr}(y) \Rightarrow h$$

Complex values can be organised into a category $\mathbb{V}$: the objects are the base language types; the set $\mathbb{V}(A, B)$ of morphisms from A to B consists of complex values of type $x\colon A \vdash f : B$, identified up to extensional equality of terms and up to renaming of x.

Proposition 15. *The category $\mathbb{V}$ has finite products and coproducts. Moreover, finite products distribute over finite coproducts, i.e. the canonical map $A \times B + A \times C \to A \times (B + C)$ has an inverse.*

We define *complex evaluation contexts* by the grammar obtained by modifying the grammar for base language evaluation contexts such that complex values may be used instead of values.

3.7.2. Enriched Categories

For the rest of this section we assume that the reader is familiar with the basic notions of enriched category theory, see e.g. [25].

Let $\widehat{\mathbb{V}}$ be the category $\mathbf{Set}^{\mathbb{V}^{\mathrm{op}}}$ of functors from $\mathbb{V}^{\mathrm{op}}$ to **Set**. For a functor $F\colon \mathbb{V}^{\mathrm{op}} \to \mathbf{Set}$ we write F_A for the set it assigns to A; given $f \in \mathbb{V}(B, A)$ we write $(-)[f]$ for the function $Ff\colon F_A \to F_B$. With this notation, the functoriality laws for F become $x[id] = x$ and $x[f \circ g] = x[f][g]$ for all $x \in F_A$, $f \in \mathbb{V}(B, A)$ and $g \in \mathbb{V}(C, B)$.

We work with $\widehat{\mathbb{V}}$-enriched categories. Recall that a $\widehat{\mathbb{V}}$-enriched category $\mathbb{C}$ is given by a collection of objects, a hom-object $\mathbb{C}(A, B)$ in $\widehat{\mathbb{V}}$ for any two objects A and B, and maps in $\widehat{\mathbb{V}}$ for identity $id_A\colon 1 \to \mathbb{C}(A, A)$ and composition $(-) \circ (-)\colon \mathbb{C}(A, B) \times \mathbb{C}(B, C) \to \mathbb{C}(A, C)$ satisfying the usual laws. The enrichment is thus taken with respect to the cartesian product on $\widehat{\mathbb{V}}$.

Since the enrichment is in $\widehat{\mathbb{V}} = \mathbf{Set}^{\mathbb{V}^{\mathrm{op}}}$, the above data specifies an ordinary category $\mathbb{C}_\Gamma$ for each object Γ of $\mathbb{V}$: The objects of $\mathbb{C}_\Gamma$ are those of $\mathbb{C}$ and the set of morphisms $\mathbb{C}_\Gamma(A, B)$ is defined to be $\mathbb{C}(A, B)_\Gamma$.

3.7.3. Computations

The $\widehat{\mathbb{V}}$*-category of computations* $\mathbb{C}$ has the same objects as $\mathbb{V}$. The set of morphisms $\mathbb{C}(A, B)_\Gamma$ is defined to be the set of all complex evaluation contexts E, for which $\Gamma,\ x\colon A \vdash E[x] : B$ is derivable, identified up to extensional equality. The functor action of $\mathbb{C}(A, B)$ is given by substitution. The composition of two maps given by the complex evaluation contexts $\Gamma,\ x\colon A \vdash E[x] : B$ and $\Gamma,\ x\colon B \vdash F[x] : C$ is defined to be the map given by $\Gamma,\ x\colon A \vdash F[E[x]] : C$. The identity morphism is given by $[\cdot]$.

Proposition 16. *The $\widehat{\mathbb{V}}$-category $\mathbb{C}$ has $\widehat{\mathbb{V}}$-enriched finite coproducts.*

Proof. We need to define an isomorphism $\psi\colon \mathbb{C}(A, C) \times \mathbb{C}(B, C) \to \mathbb{C}(A + B, C)$ that is $\widehat{\mathbb{V}}$-natural in C. It can be defined by $\psi_\Gamma(E[\cdot], F[\cdot]) = \mathsf{case}\ [\cdot]\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow E[x] \mid \mathsf{inr}(y) \Rightarrow F[y]$. An inverse is $\psi_\Gamma^{-1}(E[\cdot]) = \langle E[\mathsf{inl}([\cdot])], E[\mathsf{inr}([\cdot])]\rangle$. □

The Int construction can be applied to any traced monoidal category [6]. We next show that $\mathbb{C}$ is traced with respect to coproducts.

Definition 7. A uniform $\widehat{\mathbb{V}}$-Conway operator is a map

$$(-)^\dagger\colon \mathbb{C}(A, B + A) \to \mathbb{C}(A, B)$$

satisfying the following equations:

1. (Fixpoint) $[id_B, f^\dagger] \circ f = f^\dagger$ for all $f \in \mathbb{C}(A, B + A)_\Gamma$.
2. (Naturality) $g \circ f^\dagger = ((g + id_A) \circ f)^\dagger$ for all $f \in \mathbb{C}(A, B + A)_\Gamma$ and all $g \in \mathbb{C}(B, C)_\Gamma$.
3. (Dinaturality) $([inl, g] \circ f)^\dagger = [id_C, ([inl, f] \circ g)^\dagger] \circ f$ for all $f \in \mathbb{C}(A, C + B)_\Gamma$ and all $g \in \mathbb{C}(B, C + A)_\Gamma$.
4. (Diagonal) $(f^\dagger)^\dagger = ([id_B, inr] \circ f)^\dagger$ for all $f \in \mathbb{C}(A, (C + A) + A)_\Gamma$.
5. (Uniformity) For all $g \in \mathbb{C}(A, C + A)_\Gamma$, $h \in \mathbb{C}(B, C + B)_\Gamma$ and $k \in \mathbb{C}(A, B)_\Gamma$, if $h \circ k = (id_C + k) \circ g$ then $g^\dagger = h^\dagger \circ k$.

Lemma 17. *The $\widehat{\mathbb{V}}$-category $\mathbb{C}$ has a $\widehat{\mathbb{V}}$-Conway operator.*

Proof. The operator is defined by mapping $E \in \mathbb{C}(A, B + A)_\Gamma$ to $(\mathsf{let}\ y = [\cdot]\ \mathsf{loop}\ E[y])$.

We verify the dinaturality law, leaving the other cases to the reader. The two morphism $([inl, g] \circ f)^\dagger$ and $[id_C, ([inl, f] \circ g)^\dagger \circ f$ in $\mathbb{C}$ are given by the following complex evaluation contexts E_1 and E_2:

$$\begin{aligned} E_1 &:= \mathsf{let}\ x = [\cdot]\ \mathsf{loop}\ \mathsf{case}\ F[x]\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow \mathsf{inl}(z) \mid \mathsf{inr}(y) \Rightarrow G[y] \\ E_2 &:= \mathsf{case}\ F\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \mid \mathsf{inr}(y) \Rightarrow E_2'[y] \\ &\qquad \text{where } E_2' := \mathsf{let}\ y = [\cdot]\ \mathsf{loop}\ \mathsf{case}\ G[y]\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(x) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{inr}(y) \Rightarrow F[y] \end{aligned}$$

We have to show extensional equality of E_1 and E_2, that is for all closed values v and w, we have $E_1[v] \mapsto^* w$ if and only if $E_2[v] \mapsto^* w$.

We show by well-founded induction on i that $E_1[v] \mapsto^i w$ implies $E_2[v] \mapsto^* w$ for all v and w. The reduction $E_1[v] \mapsto^* w$ takes one of the following two forms:

- The reduction of $E_1[v]$ has the following form, in which $F[v]$ reduces to $\mathsf{inl}(w)$:

$$\begin{aligned} E_1[v] &\mapsto \mathsf{case}\ (\mathsf{case}\ F[v]\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow \mathsf{inl}(z) \mid \mathsf{inr}(y) \Rightarrow G[y])\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \\ &\qquad\qquad \mid \mathsf{inr}(y) \Rightarrow E_1[y] \\ &\mapsto^* \mathsf{case}\ (\mathsf{case}\ \mathsf{inl}(w)\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow \mathsf{inl}(z) \mid \mathsf{inr}(y) \Rightarrow G[y])\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \\ &\qquad\qquad \mid \mathsf{inr}(y) \Rightarrow E_1[y] \\ &\mapsto \mathsf{case}\ \mathsf{inl}(w)\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \mid \mathsf{inr}(y) \Rightarrow E_1[y] \\ &\mapsto w \end{aligned}$$

  In this case, we get $E_1[v] \mapsto^* w$ and immediately also $E_2[v] \mapsto^* w$, which is what we had to prove.

- The reduction of $E_1[v]$ starts as follows, where $F[v]$ reduces to $\mathsf{inr}(u)$:

$$\begin{aligned} E_1[v] &\mapsto \mathsf{case}\ (\mathsf{case}\ F[v]\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow \mathsf{inl}(z) \mid \mathsf{inr}(y) \Rightarrow G[y])\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \\ &\qquad\qquad \mid \mathsf{inr}(y) \Rightarrow E_1[y] \\ &\mapsto^* \mathsf{case}\ (\mathsf{case}\ \mathsf{inr}(u)\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow \mathsf{inl}(z) \mid \mathsf{inr}(y) \Rightarrow G[y])\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \\ &\qquad\qquad \mid \mathsf{inr}(y) \Rightarrow E_1[y] \\ &\mapsto^* \mathsf{case}\ G[u]\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \mid \mathsf{inr}(y) \Rightarrow E_1[y] \end{aligned}$$

  The reduction then continues by reduction of $G[u]$. We next make a case distinction on the value $G[u]$ reduces to. But note first that we also have

$$E_2[v] \mapsto E_2'[u] \mapsto^* \mathsf{case}\ G[u]\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(x) \mid \mathsf{inr}(y) \Rightarrow E_2'[y]\ .$$

  – $G[u]$ is reduced to $\mathsf{inl}(w)$, i.e. the reduction of $E_1[v]$ continues as follows:

$$\begin{aligned} \mathsf{case}\ G[u]\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \mid \mathsf{inr}(y) \Rightarrow E_1[y] &\mapsto^* \mathsf{case}\ \mathsf{inl}(w)\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \mid \mathsf{inr}(y) \Rightarrow E_1[y] \\ &\mapsto w \end{aligned}$$

    In this case we also have

$$\begin{aligned} E_2[v] &\mapsto^* \mathsf{case}\ G[u]\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(x) \mid \mathsf{inr}(y) \Rightarrow E_2'[y] \\ &\mapsto^* \mathsf{case}\ \mathsf{inl}(w)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(x) \mid \mathsf{inr}(y) \Rightarrow E_2'[y] \\ &\mapsto w\ , \end{aligned}$$

    as was required to show.

  – $G[u]$ is reduced to $\mathsf{inr}(u')$, i.e. the reduction of $E_1[v]$ continues as follows:

$$\begin{aligned} \mathsf{case}\ G[u]\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \mid \mathsf{inr}(y) \Rightarrow E_1[y] &\mapsto^* \mathsf{case}\ \mathsf{inr}(u')\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow z \mid \mathsf{inr}(y) \Rightarrow E_1[y] \\ &\mapsto E_1[u'] \end{aligned}$$

    The rest of the reduction then is $E_1[u'] \mapsto^j w$ for some $j < i$. By induction hypothesis we get $E_2[u'] \mapsto^* w$. We can then conclude this case by observing the following reduction sequence:

$$\begin{aligned} E_2[v] &\mapsto^* \mathsf{case}\ G[u]\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(x) \mid \mathsf{inr}(y) \Rightarrow E_2'[y] \\ &\mapsto^* \mathsf{case}\ \mathsf{inr}(u')\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(x) \mid \mathsf{inr}(y) \Rightarrow E_2'[y] \\ &\mapsto E_2'[u'] \mapsto E_2[u'] \mapsto^* w \end{aligned}$$

□

Proposition 18. *The $\widehat{\mathbb{V}}$-category $\mathbb{C}$ has a uniform trace with respect to coproducts. Precisely, this means that for all A, B and C there is a map*

$$Tr(A,B,C)\colon \mathbb{C}(A+C,B+C) \to \mathbb{C}(A,B)$$

such that, for all Γ, $(Tr_{A,B,C})_\Gamma$ is a uniform trace in $\mathbb{C}_\Gamma$ in the standard sense.

Proof. The natural transformation is given such that $Tr(A,B,C)_\Gamma$ maps E with $\Gamma,\ x\colon A+C \vdash E[x] : B+C$ to

$$\begin{aligned} F = \mathsf{case}\ E[\mathsf{inl}(x)]\ \mathsf{of}\ \mathsf{inl}(x) &\Rightarrow x \\ \mid \mathsf{inr}(y) &\Rightarrow \mathsf{let}\ z = y\ \mathsf{loop}\ E[\mathsf{inr}(z)] \end{aligned}$$

with $\Gamma,\ x\colon A \vdash F[x] : B$.

This defines a uniform trace in each $\mathbb{C}_\Gamma$, see [26]. Naturality, i.e. closure under substitution with complex values, follows immediately. □

3.7.4. Int-Construction

We can now explain in which sense the circuits defined earlier in this section appear as morphisms in an instance of the Int construction.

We use the following evident generalisation of the Int construction to $\widehat{\mathbb{V}}$-enriched categories. The $\widehat{\mathbb{V}}$-category $\mathrm{Int}(\mathbb{C})$ has as objects pairs (X^-, X^+) of $\mathbb{C}$-objects. The hom-objects are defined by

$$\mathrm{Int}(\mathbb{C})((X^-,X^+),(Y^-,Y^+)) = \mathbb{C}(X^+ + Y^-, Y^+ + X^-)\ .$$

Composition and identity are uniquely determined by requiring that $\mathrm{Int}(\mathbb{C})_\Gamma$ agrees with the usual Int construction on $\mathbb{C}_\Gamma$.

In this $\widehat{\mathbb{V}}$-enriched variant one finds the structure well-known from the Int construction, see [6]. For example, a compact closed structure is given on objects by $X \otimes Y = (X^- + Y^-, X^+ + Y^+)$ and $(X^-,X^+)^* = (X^+,X^-)$. We do not spell out this structure in detail here, because we have already defined it in terms of circuits earlier in this section.

A circuit with interface $[X_1,\dots,X_n] \to [Y_1,\dots,Y_m]$ is a representation of a map of type $X_1 \otimes \cdots \otimes X_n \to Y_1 \otimes \cdots \otimes Y_m$ in $\mathrm{Int}(\mathbb{C})$. The compact closed structure of $\mathrm{Int}(\mathbb{C})$ is given in Lemmas 3 and 4. For further information on the categorical properties of the Int construction we refer to [6, 11], as the intention in this section is merely to relate the construction of circuits to the standard Int construction.

4. Direct Definition of Combinators

The interactive language a simple linear lambda calculus with restricted copying. As such it is quite weak. However, with rule Hack a programmer can define interactive terms directly by giving an implementation of the message passing nodes that implement them. The main intended use of this is to allow the programmer to enrich the interactive language by implementing useful combinators of higher-order type that can then be used as if they were built-in constants. In this section we give examples of how to implement a few useful combinators.

4.1. Iteration

The most important example of such a combinator is an iterator

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

that informally satisfies $(\mathtt{loop}\ s\ t) = [v]$ if $(s\ t) = [\mathsf{inr}(v)]$ and $(\mathtt{loop}\ s\ t) = \mathtt{loop}\ s\ [v]$ if $(s\ t) = [\mathsf{inl}(v)]$.

We implement this combinator directly by $\mathtt{loop} := \mathsf{hack}(x.l)$ for a suitable base language term l of type $x\colon X^- \vdash l : X^+$, where:

$$\begin{array}{lllllllllllllllll} X & = & \alpha & \cdot & (& \gamma & \cdot & [\alpha] & \multimap & [\alpha+\beta] &) & \multimap & (& [\alpha] & \multimap & [\beta] &) \\ X^- & = & \alpha & \times & (& \gamma & \times & 1 & + & (\alpha+\beta) &) & + & (& \alpha & + & 1 &) \\ X^+ & = & \alpha & \times & (& \gamma & \times & \alpha & + & 1 &) & + & (& 1 & + & \beta &) \end{array}$$

Recall that $\mathsf{hack}(x.l)$ defines a single message passing node with an outgoing wire labelled X. It can therefore receive messages of type X^- and may reply with messages of type X^+.

The term $x\colon X^- \vdash l : X^+$ implements the message passing node. It works by case distinction on x and gives the following result in the five possible cases:

$$l[\mathsf{inr}(\mathsf{inr}(*))/x] \mapsto^* \mathsf{inr}(\mathsf{inl}(*)) \tag{2}$$
$$l[\mathsf{inr}(\mathsf{inl}(a))/x] \mapsto^* \mathsf{inl}(\langle a, \mathsf{inr}(*)\rangle) \tag{3}$$
$$l[\mathsf{inl}(a, \mathsf{inr}(\mathsf{inr}(b))/x] \mapsto^* \mathsf{inr}(\mathsf{inr}(b)) \tag{4}$$
$$l[\mathsf{inl}(a, \mathsf{inr}(\mathsf{inl}(a'))/x] \mapsto^* \mathsf{inl}(\langle a', \mathsf{inr}(*)\rangle) \tag{5}$$
$$l[\mathsf{inl}(a, \mathsf{inl}(g, *))/x] \mapsto^* \mathsf{inl}(\langle a, \mathsf{inl}(\langle g, a\rangle)\rangle) \tag{6}$$

We leave it as an easy exercise to write out a concrete term l that implements this specification.

The term l implements the following strategy: (2) says that a request for the value of `loop` $s\ t$ results in the value of t being requested. When an answer from t arrives, we put it in a memory cell (corresponding to $\alpha \cdot -$ in the type) and ask s for its return value (3). Whenever s asks for its argument, we supply the value from the memory cell (6). If s returns value $\mathsf{inr}(b)$, then we are done and give b as output (4). If s returns value $\mathsf{inl}(a')$, then we overwrite the memory cell content with a' and restart the step function s by asking for its result (5).

Proposition 19. *The following reduction is sound in the interactive language.*

$$\begin{aligned}\mathsf{loop}\ s\ t \quad \to \quad & \mathsf{let}\ [x] = t\ \mathsf{in}\ \mathsf{let}\ [y] = s\ [x]\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inl}(z) \Rightarrow \mathsf{loop}\ s\ [z] \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{inr}(z) \Rightarrow [z]\end{aligned}$$

It is an instructive exercise to spell out the interaction strategy implementing the right-hand side of the reduction in the proposition. The direct implementation of `loop` above is an explicit description of this strategy.

We give a typical example for a use of `loop`. Assume A is a base language type that we can iterate over in the following sense: there are terms $\vdash \mathit{first}_A : A$ and $x\colon A \vdash \mathit{next}_A(x) : A$ and $x\colon A \vdash \mathit{islast}_A(x) : 1 + 1$, such that first_A is the first element of A and, by repeatedly evaluating next_A, we get all values of A. The term islast_A indicates whether we have arrived at the last element, in which case its value is $\mathsf{inl}(*)$. Otherwise the value is always $\mathsf{inr}(*)$. For any variable-free type A such terms are easily defined.

With these assumptions, we define a combinator fold_A with the property that $\mathit{fold}_A\ s\ t$ computes $s\ x_n\ (\ldots (s\ x_1\ (s\ x_0\ t)))$, where $x_0 = \mathit{first}_A$ and $x_{i+1} = \mathit{next}_A(x_i)$ and x_n is the last element in the enumeration of A.

$$\begin{aligned}\mathit{fold}_A &: (A \times \beta \times A \times \beta) \cdot ([A] \multimap [\beta] \multimap [\beta]) \multimap [\beta] \multimap [\beta] \\ \mathit{fold}_A = \quad & \lambda f.\, \lambda y.\, \mathtt{loop}\ (\lambda p.\, \mathsf{let}\ [\langle x, y\rangle] = p\ \mathsf{in} \\ & \qquad \mathsf{let}\ [y'] = f\ [x]\ [y]\ \mathsf{in} \\ & \qquad \mathsf{case}\ \mathit{islast}_A(x)\ \mathsf{of}\ \mathsf{inl}(\mathit{true}) \Rightarrow \mathsf{return}(y') \\ & \qquad\qquad\qquad \mid \mathsf{inr}(\mathit{false}) \Rightarrow \mathsf{continue}(\langle \mathit{next}_A(x), y'\rangle) \\ &)\ (\mathsf{let}\ [y_0] = y\ \mathsf{in}\ [\langle \mathit{first}_A, y_0\rangle])\end{aligned}$$

In this program we write $\mathsf{let}\ [\langle x, y\rangle] = p\ \mathsf{in}\ t$ as syntactic sugar for the more verbose $\mathsf{let}\ [p'] = p\ \mathsf{in}\ t[\mathsf{fst}(p')/x, \mathsf{snd}(p')/y]$. We use the suggestive abbreviations $\mathsf{return} := \mathsf{inr}$ and $\mathsf{continue} := \mathsf{inl}$.

4.2. Call with Current Continuation

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

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

Its definition is again $\mathtt{callcc} := \mathsf{hack}(x.c)$, where c is a base language term of type $x\colon Y^- \vdash c : Y^+$:

$$\begin{array}{rcccccccccccccccc} Y & = & 1 & \cdot & (& \gamma & \cdot & (& [\alpha] & \multimap & X &) & \multimap & [\alpha] &) & \multimap & [\alpha] \\ Y^- & = & 1 & \times & (& \gamma & \times & (& \alpha & + & X^- &) & + & \alpha &) & + & 1 \\ Y^+ & = & 1 & \times & (& \gamma & \times & (& 1 & + & X^+ &) & + & 1 &) & + & \alpha \end{array}$$

We implement c by a nested case distinction, so that we have the following reductions:

$$c[\mathsf{inr}(*)/x] \mapsto^* \mathsf{inl}(\langle *, \mathsf{inr}(*)\rangle) \tag{7}$$

$$c[\mathsf{inl}(*, \mathsf{inr}(a))/x] \mapsto^* \mathsf{inr}(a) \tag{8}$$

$$c[\mathsf{inl}(*, \mathsf{inl}(g, \mathsf{inr}(v)))/x] \mapsto^* \mathsf{inl}(\langle *, \mathsf{inl}(\langle g, \mathsf{inl}(*)\rangle)\rangle) \tag{9}$$

$$c[\mathsf{inl}(*, \mathsf{inl}(g, \mathsf{inl}(a)))/x] \mapsto^* \mathsf{inr}(a) \tag{10}$$

These assignments implement `callcc` as follows: a request for `callcc` k results by (7) in a request for the return value of k. If k produces a result value, then (8) applies and the request is forwarded as the return value of `callcc` k . If k ever uses its argument, which amounts to calling the continuation, then the value passed to the continuation should be returned as the final result. This is done by assignments (9) and (10). Whenever k requests the result of the continuation, this request is turned into a request to k for the argument of the continuation (9). Upon supply of the argument to the continuation, the computation is aborted and this argument is returned as the end result (10).

The implementation is similar to the interpretation of `callcc` in game semantics, see e.g. [27]. As an example for `callcc`, we show the implementation of a *tail recursion combinator* `tailfix`.

$$\texttt{tailfix}\colon \alpha \cdot (\gamma \cdot ([\alpha] \multimap [\beta]) \multimap (\delta \cdot [\alpha] \multimap [\beta])) \multimap [\alpha] \multimap [\beta]$$

$$\texttt{tailfix} = \quad \lambda f.\, \texttt{loop}\ (\lambda x.\, \texttt{callcc}\ (\lambda k.\ \mathsf{let}\ [w] = f\ (\lambda y.\, \mathsf{let}\ [v] = x'\ \mathsf{in}\ k\ [\mathsf{continue}(v)])\ x\ \mathsf{in}\ [\mathsf{return}(w)]))$$

This combinator may be used to define functions by tail recursion. Consider for example the factorial function on n-bit integers. We may use the base type 2^n to represent such integers and it is straightforward to define the required operations, such as multiplication, on it. The tail recursive definition of factorial may then be defined as shown below, where we use syntactic sugar for operations on 2^n and for if-then-else.

$$\texttt{fact}\colon \alpha \cdot [2^n] \multimap [2^n]$$

$$\texttt{fact} = \quad \lambda x.\, \mathsf{let}\ [v] = x\ \mathsf{in}\ \texttt{fact\_aux}\ [\langle v, 1\rangle]$$

$$\texttt{fact\_aux}\colon \alpha \cdot [2^n \times 2^n] \multimap [2^n]$$

$$\texttt{fact\_aux} = \quad \texttt{tailfix}\ (\lambda \mathit{fact\_aux}.\, \lambda x.\, \mathsf{let}\ [\langle i, \mathit{acc}\rangle] = x\ \mathsf{in}\ \mathsf{if}\ i = 0\ \mathsf{then}\ [\mathit{acc}]\ \mathsf{else}\ \mathit{fact\_aux}\ [\langle i - 1, i * \mathit{acc}\rangle])$$

4.3. Encoding Algol-like Languages

In this section we outline how an Algol-like language with block structured state can be embedded in INTML. We embed a language that is modelled on *Basic Syntactic Control of Interference (bSCI)*, which is used by Ghica for hardware synthesis. We chose bSCI, as Ghica's translation of bSCI-programs to hardware circuits is similar to our translation of interactive terms to base language terms. Indeed, if one removes type variables from the base language, its programs become finite functions, which can be encoded directly in hardware. Thus, INTML can also be used as a language for hardware synthesis. In this section we show that its expressivity is comparable to bSCI. We note that the fine-grained control of copying in INTML allows one to go beyond bSCI, for example the Kierstead terms cannot be typed in bSCI [28], but we have shown in Section 2.3.1 how to express them in INTML.

We follow Reynolds' approach [29] of representing an Algol-like language by a simply-typed lambda calculus with a number of constants for imperative constructs. We take INTML as the underlying lambda-calculus and define constants for the Algol-like program constructs. The particular choice of constructs is guided by those in bSCI [28].

First, we define type for commands, expressions and memory cells by

$$\texttt{com} := [1]\ , \qquad \texttt{exp} := [2]\ , \qquad \texttt{cell}_\alpha := ([\alpha] \multimap \texttt{com}) \otimes [\alpha]\ .$$

A command is thus represented by a thunk. Evaluation starts upon request of the thunk and termination is indicated by receipt of the value of type 1. Boolean expressions are modelled analogously, only that the returned value now represents a boolean. A memory cell is modelled as a pair of a thunk of type $[\alpha]$, from which the content of the cell can be requested, and an update function of type $[\alpha] \multimap \texttt{com}$, which can be used to overwrite the cell with a new value.

In Reynolds approach to representing Algol one now adds constants that allow one to work with basic types. Next we show how such constants can be defined in INTML.

- Logical operations $\mathtt{op}$ on booleans are represented by terms of type

$$\mathtt{op}\colon \mathtt{exp} \multimap 2 \cdot \mathtt{exp} \multimap \mathtt{exp} \ .$$

For instance, $\mathtt{or}$ can be defined as

$$\lambda b_1.\, \lambda b_2.\, \mathsf{let}\ [x_1] = b_1\ \mathsf{in}\ \mathsf{let}\ [x_2] = b_2\ \mathsf{in}\ \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{inl}(\mathit{true}) \Rightarrow [\mathsf{inl}(*)] \mid \mathsf{inr}(\mathit{false}) \Rightarrow [x_2] \ .$$

- For commands there are terms for a skip-command and sequencing:

$$\mathtt{skip}\colon \mathtt{com}\ , \qquad \mathtt{seq}\colon \mathtt{com} \multimap \mathtt{com} \multimap \mathtt{com}\ .$$

These are defined simply by $\mathtt{skip} := [*]$ and

$$\mathtt{seq} := \lambda c_1.\, \lambda c_2.\, \mathsf{let}\ [x] = c_1\ \mathsf{in}\ \mathsf{let}\ [y] = c_2\ \mathsf{in}\ \mathtt{skip}\ .$$

We find it convenient to use let-notation directly instead of $\mathtt{seq}$.

- Branching is available by means of a term

$$\mathtt{if}\colon \mathtt{exp} \multimap 2 \cdot \mathtt{com} \multimap 2 \cdot \mathtt{com} \multimap \mathtt{com}\ .$$

Its definition is

$$\begin{aligned}\mathtt{if} := \lambda b.\, \lambda c_1.\, \lambda c_2.\, \mathsf{let}\ [x] = b\ \mathsf{in}\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(\mathit{true}) &\Rightarrow c_1 \\ \mid \mathsf{inr}(\mathit{false}) &\Rightarrow c_2\ .\end{aligned}$$

- For iteration we use a term

$$\mathtt{while}\colon 2 \cdot \mathtt{exp} \multimap 2 \cdot \mathtt{com} \multimap \mathtt{com}$$

defined by

$$\begin{aligned}\lambda b.\, \lambda c.\, \mathtt{loop}\ (\lambda s.\quad & \mathsf{let}\ [x] = b\ \mathsf{in} \\ & \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(\mathit{true}) \Rightarrow \mathsf{return}(*) \\ & \qquad \mid \mathsf{inr}(\mathit{false}) \Rightarrow \mathsf{let}\ [y] = c\ \mathsf{in}\ \mathsf{continue}(y) \\ )\ [*]\ .\end{aligned}$$

- It remains to define the terms for working with the type $\mathtt{cell}_\alpha$ of memory cells. Recall that, by definition, a value of type $\mathtt{cell}_\alpha$ consists of a pair of a function of type $[\alpha] \multimap \mathtt{com}$ for writing to the cell and a thunk of type $[\alpha]$ for reading from it. Thus, terms

$$\mathtt{der}\colon \mathtt{cell}_2 \multimap \mathtt{exp}\ , \qquad \mathtt{asg}\colon \mathtt{cell}_2 \multimap \mathtt{exp} \multimap \mathtt{com}$$

for dereferencing a boolean cell and assigning to it can be defined simply by $\mathtt{der} := \lambda m.\, \mathsf{let}\ \langle x, y\rangle = m\ \mathsf{in}\ y$ and $\mathtt{asg} := \lambda m.\, \lambda e.\, \mathsf{let}\ \langle x, y\rangle = m\ \mathsf{in}\ x\ e$.

More interesting term is a term $\mathtt{newvar}_A$, which actually implements a memory cell of type A. It allocates a new memory cell of type A and makes it available to a given block:

$$\mathtt{newvar}_A\colon A \cdot (\beta \cdot \mathtt{cell}_A \multimap \mathtt{com}) \multimap \mathtt{com}$$

Notice that $\mathtt{newvar}_A$ may use A-many copies of its argument. The content of the memory cell will be encoded in the number of the argument that is really being used.

We implement $\mathtt{newvar}_A$ using rule Hack. Even though we will use only the case $A = 2$, in order to make clear which assumptions on A are made, we give the implementation for an arbitrary type A with a given value $v_A : A$ that serves as the initial value of the memory cell. We define $\mathtt{newvar}_A$ as $\mathsf{hack}(x.n)$, where $x\colon X^- \vdash n : X^+$:

$$\begin{array}{lllllllllllllllllll} X & = & A & \cdot & (& \beta & \cdot & (& ([A] & \multimap & \mathtt{com}) & \otimes & [A] &) & \multimap & \mathtt{com}) & \multimap & \mathtt{com} \\ X^- & = & A & \times & (& \beta & \times & (& (A & + & 1) & + & 1 &) & + & 1) & + & 1 \\ X^+ & = & A & \times & (& \beta & \times & (& (1 & + & 1) & + & A &) & + & 1) & + & 1 \end{array}$$

We implement n to satisfy the following reductions:

$$n[\mathsf{inr}(*)/x] \mapsto^* \mathsf{inl}(\langle v_A, \mathsf{inr}(*)\rangle) \tag{11}$$

$$n[\mathsf{inl}(\langle v, \mathsf{inr}(*)\rangle)/x] \mapsto^* \mathsf{inr}(*) \tag{12}$$

$$n[\mathsf{inl}(\langle v, \mathsf{inl}(\langle w, \mathsf{inr}(*)\rangle)\rangle)/x] \mapsto^* \mathsf{inl}(\langle v, \mathsf{inl}(\langle w, \mathsf{inr}(v)\rangle)\rangle) \tag{13}$$

$$n[\mathsf{inl}(\langle v, \mathsf{inl}(\langle w, \mathsf{inl}(\mathsf{inr}(*))\rangle)\rangle)/x] \mapsto^* \mathsf{inl}(\langle v, \mathsf{inl}(\langle w, \mathsf{inl}(\mathsf{inl}(*))\rangle)\rangle) \tag{14}$$

$$n[\mathsf{inl}(\langle v, \mathsf{inl}(\langle w, \mathsf{inl}(\mathsf{inl}(u))\rangle)\rangle)/x] \mapsto^* \mathsf{inl}(\langle u, \mathsf{inl}(\langle w, \mathsf{inl}(\mathsf{inr}(*))\rangle)\rangle) \tag{15}$$

These can be understood as follows: Reduction (11) means that when the computation of $\mathtt{newvar}_A\ c$ is started, then value v_A is put into the new memory cell and computation c is started. If c terminates, then, by (12), the whole computation terminates. If, during the evaluation of c the content of the memory cell is being requested, then (13) applies and the content v of the memory cell is being returned. A call of c to the update function results by (14) in an immediate request to c for the value that should be written into the cell. Finally, once c replies with a value u then by (15) that value is placed in the memory cell, overwriting the previous value, and termination of the update function is indicated to c.

5. Space Bounds

The translation of interactive terms to circuits is a compilation method for whose output interesting space bounds can be established with simple means.

How much space does the circuit implementation of an interactive term use? Recall that a circuit implements message passing. Messages travel from node to node; they are transformed by each node they pass. Our circuit implementation takes any message that may be passed into the circuit and traces it through the circuit, until it is returned to the environment. The space used by this implementation is proportional to the sum of the size of the circuit and the maximum size that a message can attain. Both sizes are easy to estimate in our case. The circuit size depends only on the term and will be fixed. The maximum message size too can be read off from the circuit. As circuit wires are labelled with types, we know in advance all the possible (base language) types that the messages can have. So, a simple bound on the maximum size of a message is the maximum of the sizes of all the values of any type that appears in the circuit. Once we instantiate all the type variables, all these types will be finite, and we thus obtain finite size bounds. We immediately obtain:

Proposition 20 (Constant Space)**.** *If δ derives $\Gamma \mid \Phi \vdash t : X$ and δ does not contain any type variables, then the behaviour of the circuit $[\![\delta]\!]$ can be implemented in constant space.*

This proposition may perhaps seem trivial, but the constant space circuit implementation of INTML terms is close to Ghica's method of hardware synthesis [28]. As INTML is more expressive than Ghica's bSCI, we believe that it is interesting to study the constant space variant of INTML, e.g. for hardware synthesis.

In this section we show that this simple idea of analysing the space usage of circuit evaluation can be used to obtain interesting space bounds for INTML programs. We show that INTML characterises the classes of functions computable in logarithmic space (LOGSPACE) and non-deterministic logarithmic space (NLOGSPACE).

5.1. Logarithmic Space

We describe a precise correspondence between the class of functions representable in INTML and the class of functions computable in logarithmic space.

In order to be able to implement all LOGSPACE functions in INTML generically, we extend the base language with constants for enumerating the values of any type A.

$$C(\cdot; A) = \{\mathsf{first}_A\} \qquad\qquad C(A; A + A) = \{\mathsf{next}_A\}$$

The intention is that first_A is the first value in the enumeration of A and that $\mathsf{next}_A(v)$ evaluates to $\mathsf{inl}(w)$ if w comes after v in the enumeration, or to $\mathsf{inr}(\mathsf{first}_A)$, if v is the last value.

Clearly, for any variable-free type A one can define such terms first_A and $\mathsf{next}_A(v)$ easily. The point is, however, that the constants are available also for type variables and for types containing variables. As a result, type variables now range over finite types with a total ordering.

We extend the reduction relation to account for the new constants. While it is useful to be able to refer the total ordering on types with variables that have not been instantiated yet, the reduction itself can only be implemented on variable-free types. In the following reduction rules, A and B range over arbitrary variable-free base language types, N ranges over *non-empty* variable-free types (i.e. types for which some closed value $v\colon N$ exists), E ranges over empty variable-free types (i.e. all types without any closed value), and finally v and w range over values.

$$\begin{aligned} \mathsf{first}_1 &\to * & \mathsf{first}_{N+A} &\to \mathsf{inl}(\mathsf{first}_N) \\ \mathsf{first}_{A\times B} &\to \langle \mathsf{first}_A, \mathsf{first}_B\rangle & \mathsf{first}_{E+B} &\to \mathsf{inr}(\mathsf{first}_B) \end{aligned}$$

The reduction rules for next enumerate the values of a sum type $A + B$ so that first the values of type A are enumerated and then those of type B. The values of a product type $A \times B$ are enumerated lexicographically.

$$\begin{aligned} \mathsf{next}_1(*) &\to \mathsf{inr}(*) \\ \mathsf{next}_{A+E}(\mathsf{inl}(v)) &\to \mathsf{case}\ \mathsf{next}_A(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\mathsf{inl}(x)) \mid \mathsf{inr}(y) \Rightarrow \mathsf{inr}(\mathsf{inl}(y)) \\ \mathsf{next}_{A+N}(\mathsf{inl}(v)) &\to \mathsf{case}\ \mathsf{next}_A(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\mathsf{inl}(x)) \mid \mathsf{inr}(y) \Rightarrow \mathsf{inl}(\mathsf{inr}(\mathsf{first}_N)) \\ \mathsf{next}_{E+B}(\mathsf{inr}(v)) &\to \mathsf{case}\ \mathsf{next}_B(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\mathsf{inr}(x)) \mid \mathsf{inr}(y) \Rightarrow \mathsf{inr}(\mathsf{inr}(y)) \\ \mathsf{next}_{N+B}(\mathsf{inr}(v)) &\to \mathsf{case}\ \mathsf{next}_B(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\mathsf{inr}(x)) \mid \mathsf{inr}(y) \Rightarrow \mathsf{inr}(\mathsf{inl}(\mathsf{first}_N)) \\ \mathsf{next}_{A\times B}(\langle v, w\rangle) &\to \mathsf{case}\ \mathsf{next}_A(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\langle x, w\rangle) \\ &\qquad \mid \mathsf{inr}(x) \Rightarrow \mathsf{case}\ \mathsf{next}_B(w)\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow \mathsf{inl}(\langle x, y\rangle) \\ &\qquad\qquad \mid \mathsf{inr}(y) \Rightarrow \mathsf{inr}(\langle x, y\rangle) \end{aligned}$$

We note that it is easy to distinguish empty from non-empty types. A variable-free type is empty exactly if, when interpreted as an arithmetic expression, it equals zero. For instance the type $(1 + 1) \times (1 + 1)$ is non-empty, since as an arithmetic expression it denotes 4. The type $(1 + 1) \times (0 + 0)$ is empty, however. In an implementation of the above reduction rules, it is thus possible to determine which case matches in time and space linear in the size of the term under consideration.

With this extension to the base language, we can now prove that INTML captures the complexity class LOGSPACE. We start by analysing the space requirements of base language reduction.

We assume that term reduction is implemented directly on a machine. The size needed to encode terms and types will thus be proportional to the sizes of the corresponding abstract syntax trees. Concretely, for a base language type A, we write $|A|$ for the size of the abstract syntax tree representing it: $|\alpha| = |0| = |1| = 1$ and $|A + B| = |A \times B| = 1 + |A| + |B|$. Likewise, we write $|f|$ for the size of the abstract syntax tree of a term f, i.e. $|\mathsf{first}_A| = 1 + A$ and $|\mathsf{next}_A(f)| = 1 + A + |f|$ and $|\mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g \mid \mathsf{inr}(y) \Rightarrow h| = 1 + |f| + |g| + |h|$, etc.

For the purposes of relating INTML to LOGSPACE, it suffices to prove the following lemma on the space requirements of base language reduction. It states that the space needed for the reduction is linear in the size of the types chosen for the type variables.

Lemma 21. *Let $x\colon A \vdash f : B$ be derivable, where A and B may contain a type variable α. Then there are constants $c \in \mathbb{N}$ and $d \in \mathbb{N}$ such that $f[C/\alpha][v/x] \mapsto^* g$ implies $|g| \le c \cdot |C| + d$ for every variable-free type C and every closed value $v\colon A[C/\alpha]$.*

Proof. Define the *potential* of a term $\Gamma \vdash f : A$ to be the size of the largest term that can appear when we first substitute closed values for the free variables in f and then apply an arbitrary number of reduction steps:

$$pot(\Gamma \vdash f : A) = \max\{\, |g| \mid f[\sigma] \mapsto^* g \text{ for some value-substitution } \sigma \in \mathcal{E}_\Gamma\}$$

Notice that $pot(\Gamma \vdash f : A)$ is thus defined if and only if $\Gamma \vdash f : A$ is derivable.

Next we show that each of the following inequalities holds, given that the terms on both of its sides are defined.

$$pot(\Gamma \vdash x : A) \leq |A| \tag{16}$$

$$pot(\Gamma \vdash * : 1) \leq 1 \tag{17}$$

$$pot(\Gamma \vdash \langle f, g\rangle : A \times B) \leq 1 + pot(\Gamma \vdash f : A) + pot(\Gamma \vdash g : B) \tag{18}$$

$$pot(\Gamma \vdash \mathsf{fst}(f) : A) \leq 1 + pot(\Gamma \vdash f : A \times B) \tag{19}$$

$$pot(\Gamma \vdash \mathsf{snd}(f) : B) \leq 1 + pot(\Gamma \vdash f : A \times B) \tag{20}$$

$$pot(\Gamma \vdash \mathsf{image}(f) : A) \leq 1 + pot(\Gamma \vdash f : 0) \tag{21}$$

$$pot(\Gamma \vdash \mathsf{inl}(f) : A + B) \leq 1 + pot(\Gamma \vdash f : A) \tag{22}$$

$$pot(\Gamma \vdash \mathsf{inr}(g) : A + B) \leq 1 + pot(\Gamma \vdash g : B) \tag{23}$$

$$\begin{aligned} pot(\Gamma \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g \mid \mathsf{inr}(y) \Rightarrow h : C) \leq\ & 1 + pot(\Gamma \vdash f : A + B) \\ & + pot(\Gamma, x\colon A \vdash g : C) \\ & + pot(\Gamma, y\colon B \vdash h : C) \end{aligned} \tag{24}$$

$$\begin{aligned} pot(\Gamma \vdash \mathsf{let}\ x = f\ \mathsf{loop}\ g : A) \leq\ & 3 + pot(\Gamma \vdash f : B) \\ & + 2pot(\Gamma, x\colon B \vdash g : A + B) \end{aligned} \tag{25}$$

$$pot(\Gamma \vdash \mathsf{first}_A : A) \leq 2|A| + 1 \tag{26}$$

$$pot(\Gamma \vdash \mathsf{next}_A(f) : A) \leq 14|A| + pot(\Gamma \vdash f : A) \tag{27}$$

We consider representative cases:

- Case (16). This case is immediate, as we have $|v| \leq |A|$ for any value v of type A. This is easily shown by induction on A.

- Case (24). Any reduction of $\mathsf{case}\ f[\sigma]\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g[\sigma] \mid \mathsf{inr}(y) \Rightarrow h[\sigma]$ must, by definition, start with reducing $f[\sigma]$. In this stage all intermediate terms have the form $\mathsf{case}\ f'\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g[\sigma] \mid \mathsf{inr}(y) \Rightarrow h[\sigma]$ for some f' with $f[\sigma] \mapsto^* f'$. Clearly, the size of this term is bounded by $1 + pot(\Gamma \vdash f : A + B) + pot(\Gamma, x\colon A \vdash g : C) + pot(\Gamma, y\colon B \vdash h : C)$. Once $f[\sigma]$ has been reduced to a value $\mathsf{inl}(v)$ or $\mathsf{inr}(w)$, reduction continues with a contraction of the case and then with either $g[\sigma][v/x]$ or $h[\sigma][w/y]$. In either case, the size of the intermediate terms are bounded by the claimed bound.

- Case (27). Here we do a structural induction on the type A.

  Notice first that the reduction of $\mathsf{next}_A(f[\sigma])$ must start with a reduction of $f[\sigma]$ to a value u. The possible size increases therein are covered by $pot(\Gamma \vdash f : A)$. In the next reduction step, the constant next is unfolded once. The reduct has size at most $14 + |A| + |v|$, the worst possible case being that where A is a product $A_1 \times A_2$.

  Let us consider only the case for $A_1 \times A_2$, as the other cases are similar. In this case $u = \langle v, w\rangle$ and the reduct is

$$\begin{aligned} &\mathsf{case}\ \mathsf{next}_{A_1}(v)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow \mathsf{inl}(\langle x, w\rangle) \\ &\qquad \mid \mathsf{inr}(x) \Rightarrow \mathsf{case}\ \mathsf{next}_{A_2}(w)\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow \mathsf{inl}(\langle x, y\rangle) \\ &\qquad\qquad \mid \mathsf{inr}(y) \Rightarrow \mathsf{inr}(\langle x, y\rangle)\ . \end{aligned}$$

  As A is $A_1 \times A_2$, we can use the induction hypothesis to obtain the inequalities $pot(\Gamma \vdash \mathsf{next}_{A_1}(v) : A_1) \leq 14|A_1| + pot(\Gamma \vdash v : A_1)$ and $pot(\Gamma \vdash \mathsf{next}_{A_2}(w) : A_2) \leq 14|A_2| + pot(\Gamma \vdash w : A_2)$.

  Notice now that the reduction of this term must proceed first to reduce $\mathsf{next}_{A_1}(v)$ to a value. From the bound on the potential of $\mathsf{next}_{A_1}(v)$, we obtain that in this reduction, all intermediate terms can have size at most $14|A_1| + pot(\Gamma \vdash v : A_1) + (14 + |A_2| + |w|)$, where $(14 + |A_2| + |w|)$ is the size of the context in which $\mathsf{next}_{A_1}(v)$ appears. From $|A| = 1 + |A_1| + |A_2|$ we obtain $14|A| \geq 14 + 14|A_1| + |A_2|$. As $\langle v, w\rangle$ is the value that $f[\sigma]$ reduced to, it is clear that $pot(\Gamma \vdash f : A) \geq pot(\Gamma \vdash v : A_1) + |w|$ holds. Thus, the size of all intermediate terms is bounded by $14|A| + pot(\Gamma \vdash f : A)$.

Now that $\mathsf{next}_{A_1}(v)$ has been reduced to a value, the reduction must continue to reduce the case itself, and then to reduce the appropriate branch. If the second case is to be reduced, then reduction continues with $\mathsf{next}_{A_2}(w)$. As in the case for $\mathsf{next}_{A_1}(v)$ above, it follows from the induction hypothesis that all reducts have size no greater than $14|A| + pot(\Gamma \vdash f : A)$.

To establish the bound claimed in the lemma, consider now a derivable sequent $\Gamma \vdash f : A$. By unfolding the above inequality repeatedly as long as possible, we obtain an inequality $pot(\Gamma \vdash f : A) \leq c_1 \cdot |A_1| + c_2 \cdot |A_2| + \cdots + c_n \cdot |A_n| + d$ for some suitable types $A_1, \ldots, A_n$ that all appear in the derivation of $\Gamma \vdash f : A$ and some natural numbers $c_1, \ldots, c_n, d$. Since derivations are closed under type substitution, we get

$$pot(\Gamma[C/\alpha] \vdash f[C/\alpha] : A[C/\alpha]) \leq c_1 \cdot |A_1[C/\alpha]| + \cdots + c_n \cdot |A_n[C/\alpha]| + d$$

for any type C. But noting that $|A_i[C/\alpha]| \leq |A_i| \cdot |C|$ holds for all $i \in \{1, \ldots, n\}$, this means that there is a number c (depending only on the derivation of $\Gamma \vdash f : A$) such that

$$pot(\Gamma[C/\alpha] \vdash f[C/\alpha] : A[C/\alpha]) \leq c \cdot |C| + d$$

holds for all types C. But, by definition of the potential, this amounts to what we have to prove. □

Having proved a linear space bound for reduction in the base language, we now show that the interactive language can capture the functions computable in logarithmic space. Let Σ be a fixed, finite alphabet. We show that INTML captures the LOGSPACE-computable functions $\Sigma^* \to \Sigma^*$.

First, we must define how to represent functions from Σ^* to Σ^* in INTML. We choose an encoding that represents functions in a space efficient way, making essential use of the interactive nature of INTML-computation. Any finite set can be represented as a variable-free base language type, e.g. by a sum $1 + \cdots + 1$ with one summand for each element. We shall use finite sets as if they were base language types, with the understanding that such an encoding has been applied implicitly. With the constants first_A and next_A, any base language type can be viewed as an initial segment of the natural numbers. A variable-free type A can represent the numbers from 0 to $|\mathcal{V}_A| - 1$: the number i is encoded by the normal form of $succ_A^i(\mathsf{first}_A)$, where $succ_A(f) := \mathsf{case}\ \mathsf{next}_A(f)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow x \mid \mathsf{inr}(y) \Rightarrow f$. We write $\langle i \rangle_A$ for this value.

Now, words over the alphabet Σ can be represented as functions that map positions to characters. That is, the word $w = w_0 w_1 \ldots w_n$ is represented by a function mapping i to w_i. Any position that goes beyond the end of the word is mapped to a special blank symbol $\square$. We write $\Sigma_\square$ for $\Sigma \uplus \{\square\}$. As each type represents an initial segment of the natural numbers, we can view any type as a type of positions in a word.

Formally, for every variable-free type B, we represent Σ-words of length at most $|\mathcal{V}_B|$ as terms of the interactive type $\mathbb{B}_A(B) := A \cdot [B] \multimap [\Sigma_\square]$:

Definition 8. A closed interactive term t of type $\mathbb{B}_A(B)$ *represents a word* $w_0 w_1 \ldots w_n$, where $w_0, \ldots, w_n \in \Sigma$, if and only if $n < |\mathcal{V}_B|$ and the equivalence

$$(t\ \langle i \rangle_B) \mapsto^* [c] \quad \Longleftrightarrow \quad (i \leq n \wedge c = w_i) \vee (i > n \wedge c = \square)$$

holds for any $i < |\mathcal{V}_B|$.

Lemma 22. *For every word $w \in \Sigma^*$ and all variable-free types A and B satisfying $|w| \leq |\mathcal{V}_B|$, there exists a closed interactive term $\langle w \rangle_{A,B}$ that represents w.*

Proof. We can define a suitable term by $\langle w \rangle_{A,B} = \lambda x.\, \mathsf{let}\ [i] = x\ \mathsf{in}\ [w(i)]$, where $w(i)$ is a base language term that with nested case distinctions implements the pseudo-code $\mathsf{if}\ i = \langle 0 \rangle_B\ \mathsf{then}\ w_0\ \mathsf{else}\ \mathsf{if}\ i = \langle 1 \rangle_B\ \mathsf{then}\ w_1\ \ldots$. Notice that using next we can define an equality function on B: two elements are equal if and only if it takes the same number of applications of next until we reach the end of the enumeration. The return value of next tells us whether the end of the enumeration has been reached. □

Functions that take as inputs words of unbounded length can be represented using type variables. Let X be the interactive 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 types that may also contain α, but not other type variables. If E is a variable-free type, then $X[E/\alpha]$ is a type of functions from words of length at most $|\mathcal{V}_E|$ to words of length at most $|\mathcal{V}_{D[E/\alpha]}|$.

Definition 9. We say that a term $\vdash t : A \cdot \mathbb{B}_B(\alpha) \multimap \mathbb{B}_C(D)$ *represents* a function $\varphi : \Sigma^* \to \Sigma^*$ if and only if for every word $w \in \Sigma^*$ and every variable-free type E with $|\mathcal{V}_E| \geq |w|$, the term $(t[E/\alpha]\ \langle w \rangle_{B[E/\alpha],E})$ encodes $\varphi(w)$.

We remark that one may also use an alternate definition that does not refer to reduction in the interactive language, without affecting the truth of the following theorem. Instead of requiring $t[E/\alpha]\ \langle w \rangle_{B[E/\alpha],E}\ \langle i \rangle_{D[E/\alpha]}$ to reduce to one of c_i or $\square$, one may ask for the circuit for this term to have the same behaviour as either $[c_i]$ or $[\square]$, depending on i.

Theorem 23 (Logspace Soundness)**.** *If $\varphi\colon \Sigma^* \to \Sigma^*$ is represented by t, then φ is computable in logarithmic space. Moreover, a* logspace *algorithm computing φ is given by circuit-evaluation of t.*

Proof. Compiling t to a base language term yields a term $x\colon X^- \vdash f : X^+$, where X has the form $A \cdot \mathbb{B}_B(\alpha) \multimap \mathbb{B}_C(D)$. We now choose E_n to be the type

$$E_n \;=\; \underbrace{(1+1) \times \cdots \times (1+1)}_{\lceil \log n \rceil \text{ times}} \;.$$

Clearly, $|\mathcal{V}_{E_n}| \geq n$, but $|E_n| \leq 2(\log n + 1)$. By Lemma 21, reduction of $f[E_n/\alpha]$ (i.e. computation of φ on strings of length up to n) can be implemented using space linear in $|E_n|$ and so overall using space $O(\log n)$.

Using this observation, we now construct a logspace otm M_φ for φ. Given input w, this machine simulates the circuit for the term $(t\ \langle w \rangle_{B[E_n/\alpha],E_n}\ \langle i \rangle_D)\colon [\Sigma_\square]$, first for $i = 0$, then $i = 1$, etc. This will result in a sequence of characters, which the machine writes on its output tape, as they are returned from the circuit. The machine stops once a blank symbol is returned.

The machine M_φ works as follows. Given input word w, it writes down the term $f[E_n/\alpha]$, where $n = |w|$. It then reduces $f[E_n/\alpha]\ \mathsf{inr}(\mathsf{inr}(*))$, which amounts to asking the function for a character. Depending on the result value, it continues as follows: (i) if the value is $\mathsf{inr}(\mathsf{inr}(c))$ (informally: 'The requested character is c.') then it outputs c, advances the output head and if c was not a blank symbol it begins from the start; (ii) if the value is $\mathsf{inr}(\mathsf{inl}(v, *))$ (informally: 'Which character of $\varphi(w)$ should be computed?') it reduces $f[E_n/\alpha]\ \mathsf{inr}(\mathsf{inl}(v, \langle i \rangle_{D[E_n/\alpha]}))$, where i is the position of the output head, and continues with the same case distinction; (iii) if the value is $\mathsf{inl}(v, \mathsf{inr}(*))$ (informally: 'A character from the input is requested.'), then it reduces $f[E_n/\alpha]\ \mathsf{inl}(v, \mathsf{inl}(\mathsf{first}_B, *))$ (informally: 'What is the position of the requested character?') and continues with the same case distinction; (iv) if the value is $\mathsf{inl}(v, \mathsf{inl}(w, \langle i \rangle_{E_n}))$ (informally: 'It is the i-th input character that is requested.'), then it moves its input head to position i of the input tape, reads the character w_i there and reduces $f[E_n/\alpha]\ \mathsf{inl}(v, \mathsf{inr}(w_i))$ (informally: 'The requested character is w_i.') and continues with the same case distinction. □

Theorem 24 (Logspace Completeness)**.** *Any function $\varphi\colon \Sigma^* \to \Sigma^*$ computable in logarithmic space is represented by some interactive term t_φ.*

Proof. The proof goes by a simple encoding of Offline Turing Machines in intml.

Let M be a logspace otm computing φ. Assuming that α is a type with enough elements to encode the positions in an input word, the state of M can be encoded by a base language type of the form

$$S(\alpha) \;=\; Q \times \alpha \times T(\alpha) \times T(\alpha) \times P(\alpha),$$

where Q is the set of states of M and $T(\alpha)$ and $P(\alpha)$ are types containing the type variable α. The component of type α encodes the position of the input head. The two components of type $T(\alpha)$ encode the content of the work tape of M to the left and right of M's work head in a standard way. Note that if for α we choose a type with n values, then the number of elements of $T(\alpha)$ will be polynomial in n. We can therefore encode a tape of logarithmic length (and not more). Finally, the component of type $P(\alpha)$ encodes the position of the output head.

The initial configuration of the machine M is readily encoded as a base language value $\vdash \mathtt{init}_0 : S(\alpha)$.

The step function of M is implemented as a base language term

$$c\colon \Sigma,\ n\colon P(\alpha),\ s\colon S(\alpha) \vdash \mathtt{step}_0(c, s, n) : S(\alpha) + \Sigma_\square$$

that formalises the step of M, given that the currently scanned input character is c, the current state is s and that the aim is to compute the character at position n on the output tape. The result of $\mathtt{step}_0(c, s, n)$ is $\mathsf{inl}(s')$ if M moves from s to s' in one step and the output head of M has not moved beyond the position represented by n. The result of $\mathtt{step}_0(c, s, n)$

is $\mathsf{inr}(d)$ if in this step M writes the character d the position addressed by n on the output tape. We omit the details of the definition of $\mathtt{step}_0$, which in the presence of iteration and the constants first and next is a straightforward exercise.

To simulate the computation of OTM M, we must read the character c from the input word, which is encoded as an function in the interactive language. We define the step function as an interactive term

$$n\colon P(\alpha) \mid w\colon A \cdot [\alpha] \multimap [\Sigma_\square] \vdash \mathtt{step}(w,n) : B \cdot [S(\alpha)] \multimap [S(\alpha) + \Sigma_\square] \ ,$$

for suitable A and B:

$$\begin{aligned} \mathtt{step}(w,n) = \lambda s. \quad & \mathsf{let}\ [\langle q, pos, tapeleft, taperight, outpos\rangle] = s\ \mathsf{in} \\ & \mathsf{let}\ [c] = w\ [pos]\ \mathsf{in} \\ & [\mathtt{step}_0(c, \langle q, pos, tapeleft, taperight, outpos\rangle, n)] \end{aligned}$$

With these definitions, a term t_φ representing φ can be defined by

$$\begin{aligned} & t_\varphi\colon C\cdot(D\cdot[\alpha] \multimap [\Sigma_\square]) \multimap (E\cdot[P(\alpha)] \multimap [\Sigma_\square]) \\ & t_\varphi = \lambda w.\, \lambda n'.\, \mathsf{let}\ [n] = n'\ \mathsf{in}\ \mathtt{loop}\ \mathtt{step}(w,n)\ [\mathtt{init}_0] \ . \end{aligned}$$

□

Being able to encode Turing Machines does not say much about the intensional expressivity of INTML, however. We refer to [7] for larger program examples, which demonstrate that INTML is useful as a structured programming languages for LOGSPACE.

5.2. Nondeterministic Logarithmic Space

The functions computable in non-deterministic logarithmic space (NLOGSPACE) can be captured much in the same way as those in LOGSPACE. Just as for Turing Machines one moves from deterministic to non-deterministic transition functions, one can in INTML allow non-determinism in the base language and capture non-deterministic logarithmic space in this way.

We extend the base language with non-determinism by adding a new constant $\vdash\ \mathsf{ndchoice} : 1 + 1$ in addition to first and next: Its two reduction rules make $\mathsf{ndchoice}$ a non-deterministic choice operator:

$$\mathsf{ndchoice} \mapsto \mathsf{inl}(*) \qquad\qquad \mathsf{ndchoice} \mapsto \mathsf{inr}(*)$$

The reader should reconsider the definition of what it means for a term t of type $\vdash\ t : A \cdot \mathbb{B}_B(\alpha) \multimap \mathbb{B}_C(D)$ to represent a function $\varphi\colon \Sigma^* \to \Sigma^*$ in the light of the non-determinism in the base language: For any word w, any variable-free base language type E with $|w| < |\mathcal{V}_E|$, any $i < |\mathcal{V}_E|$ and any $c \in \Sigma$, there exists a reduction sequence from $t\ \langle w\rangle_{B[E/\alpha],E}\ \langle i\rangle_{D[E/\alpha]}$ to $[c]$ if and only if c is the character at position i in the word $\varphi(w)$ (when considered padded with blank symbols). This corresponds to the computation of a non-deterministic Turing Machine: the character at position i on its output tape is c if and only if it has a run that outputs c on this position.

Theorem 25 (NLOGSPACE Soundness)**.** *If $\varphi\colon \Sigma^* \to \Sigma^*$ is represented by t, then φ is computable in non-deterministic logarithmic space.*

Proof. The proof goes just like that of Theorem 23 for LOGSPACE soundness. For the reduction of $\mathsf{ndchoice}$ we use non-deterministic choice. □

Theorem 26 (NLOGSPACE Completeness)**.** *Any function $\varphi\colon \Sigma^* \to \Sigma^*$ computable in non-deterministic logarithmic space is represented by some interactive term t_φ.*

Proof. Let M_φ be a NLOGSPACE Turing Machine that computes φ. We construct a term t_φ just as in the proof to Theorem 24. Non-deterministic choices are accounted for in the definition of $\mathtt{step}_0(c, s, n)$ by using the constant $\mathsf{ndchoice}$. That is, we define $\mathtt{step}_0(c, s, n)$ such that, when given concrete values for c, s and n, there exist reduction sequences leading to each possible successor configuration, and only to those configurations. □

6. Related Work

Geometry of Synthesis. The way to see λ-terms as circuits is similar to Ghica's Geometry of Synthesis [28], which is defined on bSCI as described in Section 4.3. The main difference to our approach is the treatment of duplication. In INTML, duplication is handled through subexponential types, similarly to what happens with indexes in Abramsky-Jagadeesan-Malacaria (AJM) games. This does not require circuits to have internal states, but forces tokens to be more complex: whenever you interact with an object which could possibly be duplicated, you need to specify which specific copy of it you are querying. In Ghica's work, on the other hand, the target language consists of so-called handshake circuits, which do have an internal state. This has of course the advantage of keeping tokens simple, but has a price: the implementation of non-linear λ-terms is not always possible, Kierstead's terms being a prominent example.

Geometry of Implementation. Many years before the advent of the Geometry of Synthesis, Mackie had the idea of turning the Geometry of Interaction into an abstract machine implementing lambda calculus reduction [18]: the source language was PCF, while the object language took the form of a minimal assembly language for an abstract register machine. Mackie's translation is similar to ours, but no considerations about space consumption of programs is made. Mackie's treatment of copying amounts to what happens when one adds to the base language a type of binary trees $T(\alpha)$ with the constructors

$$\mathit{Empty}\colon T(\alpha)\ , \qquad \mathit{Leaf}\colon \alpha \to T(\alpha)\ , \qquad \mathit{Node}\colon T(\alpha) \to T(\alpha) \to T(\alpha)\ .$$

With such a type and a suitable destructor one has $T(\alpha) \times T(\alpha) \triangleleft T(\alpha)$ and $T(\alpha) + T(\alpha) \triangleleft T(\alpha)$. Then, one can use $T(1)$ for all subexponentials and use the STRUCT to maintain this form after each rule application. Thus, arbitrary copying can be allowed. Of course, the type system then does not yield interesting space bounds anymore. In this paper we have shown that this approach to copying can be refined and controlled, so that space bounds can be proven.

Geometry of Interaction Situations. Abramsky, Haghverdi & Scott [13] study an axiomatic framework for Girard's Geometry of Interaction (GoI) [30]. They introduce *Geometry of Interaction situations* to capture the structure of the GoI abstractly. In particular, they show that the Int construction gives rise to one such GoI situation. The main difference to our work in this paper is that in a GoI situation one assumes enough structure to model the exponentials of Linear Logic. In our notation, the added structure would amount to a base language type ω of natural numbers with properties such as $\omega + \omega \triangleleft \omega$. We consider it a main contribution of INTML that we do not need to make such assumptions, as this enables programming with strongly limited space, such as constant or logarithmic space.

Game Semantics. The idea of viewing lambda-terms as message passing circuits which communicate with their environment by exchanging questions and answers is also reminiscent of game semantics. This comes with no surprise, given the strong relation existing between the latter and geometry of interaction [31, 32]. Indeed, our construction is quite similar to AJM games for multiplicative and exponential linear logic [15], the main difference being the way we treat the exponentials. While we use a simple construction of subexponentials $X^{\otimes A}$, in AJM-games the exponentials are constructed from $X^{\otimes \mathbb{N}}$ by identifying a set of possible plays and by identifying different plays that represent the same computation. In INTML there is no provision, such as the plays in AJM-games, that restricts the possible interactions with a circuit. One may see INTML as a programming language for implementing the (possibly ill-formed) raw strategies of AJM-games.

Stratified Bounded Affine Logic. Indexing is not a new idea: Girard, Scedrov and Scott's Bounded Linear Logic [33] is one of the very first characterisations of polynomial time computable functions as a subsystem of Linear Logic, and is based on exactly the same idea: $!A$ does not stand for an *arbitrary* number of copies of A, but for a bounded one, where the bound takes the form of a polynomial p which explicitly appears in formula, e.g. $!_pA$. The second author has introduced a restriction on Bounded Linear Logic, namely Stratified Bounded Affine Logic (SBAL), which captures logarithmic space [3]. SBAL can be understood as a variant of INTML, in which only the interactive language is visible and the base language is hidden from the programmer. While access to the base language makes INTML simpler, SBAL has features, such as bounded quantification, that have not yet been considered in the context of INTML.

7. 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, expressive language capturing LOGSPACE. Practical experience with an experimental implementation of INTML suggests that, with suitable type inference, INTML can be made to be quite usable [7].

We believe that INTML is not only interesting as a language for sublinear space computation. The interactive computation model appears to have a wide range of applications – we have already commented on the connection to hardware synthesis, for example – and the INTML type system is general enough to be instantiated for different applications. Regarding the INTML type system itself, we intend to investigate the relation to the use of subexponentials in the proof theory of linear logic, e.g. in [34].

References

[1] S. Muthukrishnan, Data streams: algorithms and applications, Foundations and trends in theoretical computer science, Now Publishers, Hanover, MA, 2005.

[2] P. M. Neergaard, A functional language for logarithmic space, in: Proceedings of Programming Languages and Systems: Second Asian Symposium (APLAS'04), Vol. 3302 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2004, pp. 311–326.

[3] U. Schöpp, Stratified bounded affine logic for logarithmic space, in: Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS'07), IEEE, Los Alamitos, CA, 2007, pp. 411–420.

[4] L. Kristiansen, Neat function algebraic characterizations of LOGSPACE and LINSPACE, Computational Complexity 14 (2005) 72–88.

[5] G. Bonfante, Some programming languages for Logspace and Ptime, in: Proceedings of Algebraic Methodology and Software Technology (AMAST'06), Vol. 4019 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 66–80.

[6] A. Joyal, R. Street, D. Verity, Traced monoidal categories, Math. Proc. Cambridge Philos. Soc. 119 (3) (1996) 447–468.

[7] U. Dal Lago, U. Schöpp, Type inference for sublinear space functional programming, in: Proceedings of the 8th Asian conference on Programming languages and systems (APLAS'10), Vol. 6461 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 376–391.

[8] U. Dal Lago, U. Schöpp, Functional programming in sublinear space, in: Proceedings of the 19th European Symposium on Programming (ESOP'10), Vol. 6012 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 205–225.

[9] U. Schöpp, Computation-by-interaction with effects, in: Proceedings of the 9th Asian conference on Programming languages and systems (APLAS'11), Vol. 7078 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 305–321.

[10] A. Szepietowski, Turing Machines with Sublogarithmic Space, LNCS 843, Springer-Verlag, Berlin, Heidelberg, 1994.

[11] M. Hasegawa, On traced monoidal closed categories, Mathematical Structures in Computer Science 19 (2) (2009) 217–244.

[12] S. Abramsky, R. Jagadeesan, Games and full completeness for multiplicative linear logic, J. Symbolic Logic 59 (1994) 543–574.

[13] S. Abramsky, E. Haghverdi, P. J. Scott, Geometry of interaction and linear combinatory algebras, Mathematical Structures in Computer Science 12 (5) (2002) 625–665.

[14] S. Katsumata, Attribute grammars and categorical semantics, in: In Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08), Vol. 5126 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 271–282.

[15] S. Abramsky, R. Jagadeesan, P. Malacaria, Full abstraction for PCF, Inf. Comput. 163 (2) (2000) 409–470.

[16] J. M. E. Hyland, C.-H. L. Ong, On full abstraction for PCF: I, II, and III, Inf. Comput. 163 (2000) 285–408.

[17] P. Baillot, K. Terui, Light types for polynomial time computation in lambda calculus, Inf. Comput. 207 (1) (2009) 41–62.

[18] I. Mackie, The geometry of interaction machine, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), ACM, New York, NY, 1995, pp. 198–208.

[19] N. Hoshino, A modified goi interpretation for a linear functional programming language and its adequacy, in: M. Hofmann (Ed.), FOSSACS, Vol. 6604 of Lecture Notes in Computer Science, Springer, 2011, pp. 320–334.

[20] D. R. Ghica, A. Smith, Geometry of synthesis iii: resource management through type inference, in: T. Ball, M. Sagiv (Eds.), POPL, ACM, 2011, pp. 345–356.

[21] P.-A. Melliès, Functorial boxes in string diagrams, in: Proceedings of the 25th International Workshop on Computer Science Logic (CSL'06), Vol. 4207 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 1–30.

[22] H. G. Mairson, From hilbert spaces to dilbert spaces: Context semantics made simple, in: Proceedings of the 22nd Conference Foundations on Software Technology and Theoretical Computer Science (FSTTCS'02), Springer-Verlag, Berlin, Heidelberg, 2002, pp. 2–17.

[23] J. Egger, R. E. Møgelberg, A. Simpson, Enriching an effect calculus with linear types, in: In Proceedings of the 23rd international Workshop on Computer Science Logic (CSL'09), Vol. 5771 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 240–254.

[24] P. B. Levy, Call-By-Push-Value: A Functional/Imperative Synthesis, Vol. 2 of Semantics Structures in Computation, Springer-Verlag, Berlin, Heidelberg, 2004.

[25] G. M. Kelley, Basic Concepts of Enriched Category Theory, Cambridge University Press, Cambridge, UK, 1982.

[26] M. Hasegawa, Models of Sharing Graphs (A Categorical Semantics of Let and Letrec), Springer-Verlag, Berlin, Heidelberg, 1999.

[27] J. Laird, Full abstraction for functional languages with control, in: In Proceedings of the 12th Symposium on Logic in Computer Science (LICS'97), IEEE, Los Alamitos, CA, 1997, pp. 58–67.

[28] D. R. Ghica, Geometry of synthesis: a structured approach to VLSI design, in: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), ACM, New York, NY, 2007, pp. 363–375.

[29] J. C. Reynolds, Syntactic control of interference, in: Proceedings of the Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'78), ACM, New York, NY, 1978, pp. 39–46.
[30] J.-Y. Girard, Geometry of interaction 1: Interpretation of System F, in: In Proceedings of Logic Colloquium '88, Vol. 127 of Studies in Logic and the Foundations of Mathematics, Elsevier, Amsterdam, 1989, pp. 221 – 260.
[31] P. Baillot, Approches dynamiques en sémantique de la logique lineaire: jeux et géometrie de l'interaction, Ph.D. thesis, University Aix-Marseille II (1999).
[32] V. Danos, H. Herbelin, L. Regnier, Game semantics and abstract machines, in: Proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS'96), IEEE, Los Alamitos, CA, 1996, pp. 394–405.
[33] J.-Y. Girard, A. Scedrov, P. J. Scott, Bounded linear logic: a modular approach to polynomial-time computability, Theoretical Computer Science 97 (1992) 1–66.
[34] V. Nigam, D. Miller, Algorithmic specifications in linear logic with subexponentials, in: Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'09), ACM, New York, NY, 2009, pp. 129–140.