Computation by interaction for space-bounded functional programming

Ugo Dal Lago, Ulrich Schöpp

Computation by Interaction for Space-Bounded Functional Programming

Ugo Dal Lago[a], Ulrich Schöpp[b]

[a] *Università di Bologna, Italy and INRIA, France*
[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. It can be understood as a programming language simplification of Stratified Bounded Affine Logic. We show that it captures the classes FLOGSPACE and NFLOGSPACE of the functions computable in deterministic logarithmic space and in non-deterministic logarithmic space, respectively. We illustrate the expressiveness of INTML by showing how typical graph algorithms, such a test for acyclicity in undirected graphs, can be represented in it.

1. Introduction

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

In this paper we study how a programming language can support the implementation of algorithms with sublinear space usage. Sublinear space algorithms are characterised by having less memory than they would need to store their input. They are useful for computing with large data that may not fit into memory. Examples of concrete application scenarios are streaming algorithms [35], web crawling, GPU computing, and statistical data mining.

We focus in particular on algorithms with logarithmic space usage. A common intuition of such algorithms is that they access externally stored large data using only a fixed number of pointers. A typical example is that of large graphs, where the pointers are references to graph nodes. To encode a pointer to a node of a graph with n nodes, one needs space $O(\log n)$. Logarithmic space graph algorithms are therefore often thought of as algorithms that work with only a constant number of pointers to graph nodes. Another typical example of large data are long strings. In this case the pointers are character positions.

While logarithmic space algorithms can only store a fixed number of pointers, they can produce large output that does not fit into memory. They do so by producing the output bit by bit. A program that outputs a graph would output the graph nodes one after the other, followed by a stream of the pairs of nodes that are connected by an edge. The program may not have enough memory to store the whole output all at once, but it can produce it bit by bit.

When writing programs with logarithmic space usage, one must often use special techniques for tasks that would normally be simple. Consider for example the composition of two programs. In order to remain in logarithmic space, one cannot run the two algorithms one after the other, as there may not be enough space to store the intermediate result. Instead, one needs to implement composition without storing the intermediate value at all. To compose two programs, one begins by running the second program and executes it until it tries to read a bit from the output of the first program. Only then does one start the first program to compute the requested bit, so that the execution of the second program can

be continued. Running the first program may produce more output than just the required bit, but this output is simply discarded. In this way, the two programs work as coroutines and the output of the first program is only computed on demand, one bit at a time.

Implementing such on-demand recomputation 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 a program that does not store the whole intermediate value.

In this paper we ask: How can the implementation of functions with logarithmic space usage be supported in a programming language? We follow the approach of viewing logarithmic space computation as computation with a fixed number of pointers. Algorithms that use a fixed number of pointer values are not hard to capture by first-order programming languages. Suppose we use a type `int` of unsigned integers to represent pointers. Then computation with a fixed number of pointers simply becomes computation with a fixed number of `int`-values. In imperative programming, this may be captured by a while-language [26]; in functional programming it may be captured by programs that are restricted to first-order data and tail recursion [27].

But how should we represent large data like externally stored graphs or large intermediate results that should only be recomputed in small parts on demand? First, we need some way to use the pointers to actually access some large data. If we work with a large graph, for example, and pointers are just `int`-values, then we need some way of finding out whether or not there is an edge between the nodes reference by two pointers. It would be easy to extend a programming language with primitive operations for this purpose if we only had a single fixed large input graph. But we would like to treat large data much like first-class values that can be passed around and transformed in the program. Therefore, we must explain how to access large data and how to construct it, not just for a the input and output, but also for intermediate results computed by the program itself.

In this paper we propose to model large data as interactive entities. The idea is to formalise the intuition that large data is never computed all at once, but that pieces of data are requested when they are needed. For example, a large graph may be represented by an entity that takes pairs of pointers to nodes as requests and that answers with true or false depending on whether or not there is an edge in the graph. Large strings can be represented similarly as entities that take requests for character positions and that answer with the character at that position.

Such interactive entities may be presented to the programmer as functions that map requests to answers. A large string can be represented by its length together with a function of type $\texttt{int} \to \texttt{char}$ that returns the character at the given position (if it is smaller than the length). Graphs can be represented by the number of nodes together with a function $\texttt{int} \times \texttt{int} \to \texttt{bool}$ that tells whether there is an edge between two nodes. Such functional representations of data are suitable for on-demand recomputation. In practical implementations of functional programming languages, the values of type $\texttt{int} \to \texttt{char}$ are not stored by the whole graph of the function. Rather, they are given by code that can compute a character for any given integer when required.

In a higher-order functional programming language, such functional representations of data can be used as first-class values. To represent strings, we may use the type $\texttt{String} := \texttt{int} \times (\texttt{int} \to \texttt{char})$ of pairs of the word length and a function mapping positions to characters. A predicate on strings then becomes a higher-order function of type $\texttt{String} \to \texttt{bool}$. To work with strings, one may define basic operations directly, e.g. $\texttt{cons}_c \colon \texttt{String} \to \texttt{String}$ for prepending a character c:

$$\texttt{cons}_c \;=\; \lambda x.\, \texttt{let}\ \langle n, w\rangle = x\ \texttt{in}\ \langle n+1, \lambda i.\, \texttt{if}\ i = 0\ \texttt{then}\ c\ \texttt{else}\ w\ (i-1)\rangle \;\;.$$

Having defined a suitable set of such operations, one does not need to know the encoding details of the string type anymore and can use it as if it were a normal data type with operations. It is then not visible that strings may be too large to fit into memory.

In this paper we show that this higher-order functional programming approach can be used for implementing algorithms with logarithmic space usage. We argue that the higher-order approach captures typical algorithms with logarithmic space in a natural way. Suppose we have a program p of type $\texttt{String} \to \texttt{String}$; a simple example would be $\lambda x.\, \texttt{cons}_c\ (\texttt{cons}_d\ x)$. Suppose we have a large externally stored string that we want to use as an input for this program. We can construct a value $\langle n, w\rangle$ of type `String`, where n is the length of the string and w is a function that when invoked with argument i has the effect of looking up the i-th character of the externally stored string (perhaps by

sending a network request to a database or similar). Now we can apply p to $\langle n, w\rangle$ and obtain a term of type `String`. We use it to compute the characters of the result string one by one by applying the returned function to 0, 1, 2, etc. Each time we ask for a character, the program will request from the input word the characters that it needs to compute the requested character. This corresponds well to the on-demand recomputation that one finds in algorithms with logarithmic space usage. But of course we have to be careful if we want to remain within logarithmic space. With higher-order functions the restriction to tail recursion no longer guarantees that programs only store a constant number of pointers during evaluation. Indeed, with a CPS-translation any recursive program can be brought into a tail recursive form.

In this paper we show that higher-order functions are nevertheless suitable for the implementation of functions with logarithmic space usage in a functional language. We show how to combine computation with a fixed number of pointers with higher-order functions in a functional programming language. We explain how one can extend a simple first-order language for programming with a fixed number of pointers with higher-order functions in such a way that programs are still guaranteed to use no more than a fixed number of pointers. The higher-order extensions will in fact be fully definitional in the sense that any higher-order program will be translated to a first-order program. Since our first-order programs cannot use more than a finite number of pointers, this property will then be true for higher-order programs as well.

In this paper we work this out in the form of a minimal functional programming language, INTML, for programming with logarithmic space. The INTML type system is conceptually simple, and, with the type inference procedure of [11], which is outlined in Section 2.3.1, it is also relatively easy to use. Nevertheless, we argue that the type system is strong enough to allow logarithmic space algorithms to be written in a natural way.

1.1. Organising Interactive Computation

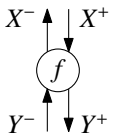
The heart of the definition of INTML is the construction of higher-order functions in a first-order language that computes with a fixed number of pointers only. This construction is an instance of the Int-construction [28], see Section 5.7. We begin with a simple first-order language that captures computation with a fixed number of pointers. We then study how this language can implement interactive processes, such as the logarithmic space algorithms that access their input by requesting it piece by piece. This model of interactive computation can interpret higher-order functions, which is what INTML makes available directly to the programmer. The point is that the whole construction is such that in the end all computation will happen in the first-order language. Even with higher-order functions, programs use no more than a fixed number of pointers and thus remain within logarithmic space.

For an informal explanation of the construction, the details of the first-order base language for computing with a fixed number of pointers are not very important. Assume given a first-order programming language with types type `int`, 1 (a unit type), $A + A$ and $A \times A$ that captures a suitable form of computation with a fixed number of pointers. Assume that it has the usual terms for these types and also a loop construct for possibly nonterminating iteration. A first-order program of type $A \to B$ is a program that takes a value of the first-order type A as input and that returns a value of the first-order type output B. Our first-order programs do not have state and always return the same answer for the same input. For now, the reader may think of programs of type $A \to B$ simply as partial functions from the values of type A to the values of type B. More details can be found in Section 2.1, but they should not be needed for now.

The construction of higher-order functions is based on a systematic analysis of on-demand interactive computation. The interface of an interactive entity X can be represented by a pair of first-order types (X^-, X^+), where X^- is the type of all the requests that may be sent to the entity and X^+ is the type of all possible answers. For a representation of long strings, one may choose $\texttt{String}^-$ to be a type that can encode the questions "What is the length of the string?" and "What is the i-th character?". The type $\texttt{String}^+$ would be chosen to encode the possible corresponding answers. An implementation of the interface X is a first-order program that maps X^- to X^+ and thus explains how to answer requests.

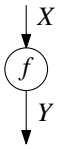
In computation with logarithmic space one often finds the situation that one has access to one interactive entity $X = (X^-, X^+)$ representing the input and one wants to implement another interactive entity $Y = (Y^-, Y^+)$ representing the output. This can be done without higher-order types by a first-order program of type $f\colon X^+ + Y^- \to X^- + Y^+$. 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 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 f to this answer. We repeat this until we get an answer in Y^+ (i.e. possibly forever).

Let us think of the first-order program function f as a message-passing node with two input wires and two output wires as drawn below:

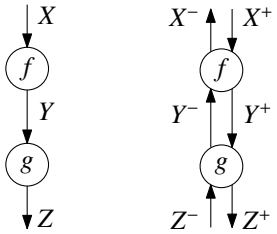


When an input value arrives on an input wire then f is applied to it and the resulting output value (if any) is passed along the corresponding output wire.

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 $X = (X^-, X^+)$ allows any message from X^+ to be passed in the forward direction and any message from X^- to be passed in the backwards direction:



Message passing nodes can be connected to form message-passing circuits. The composition $g \circ f \colon X \to Z$ of $f \colon X \to Y$ and $g \colon Y \to Z$ as shown on the left below is obtained simply by connecting the two nodes. The underlying first-order program of type $X^+ + Z^- \to Z^+ + X^-$ is most easily described in terms of message passing:



An input in X^+ is given to f and one in Z^- to g. If either f or g give an output in X^- or Z^+ then this is the output of $g \circ f$. If, however, f (resp. g) outputs a message on Y^+ (resp. Y^-), this message is given as an input to g (resp. f). This may lead to a looping computation and $g \circ f$ may be partial even when g and f are both total.

This simple model of interactive computation can be used to support higher-order functions.

Thunks. As base type, one may take the interfaces of the form $[A] := (1, A)$ for any type A. The single value of type 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.

Pairs. The interface $X \otimes Y := (X^- + Y^-, X^+ + Y^+)$ corresponds to a pair type. It may be used to represent entities that provide both an implementation of X and one of Y. The intention is that a query in X^- is answered with an answer in X^+, and likewise for Y. In conjunction with pairs, the empty interface $I := (\emptyset, \emptyset)$ is sometimes useful, as $X \otimes I$ and X can be considered as the same interface.

A message-passing node f of type $X \otimes Y \to Z$ can be understood as a message passing node with two inputs; we depict it as shown below. It corresponds to a program of type $(X^+ + Y^+) + Z^- \to (X^- + Y^-) + Z^+$. It is indicated on the right below, which type of messages should be passed in which direction along which wire.

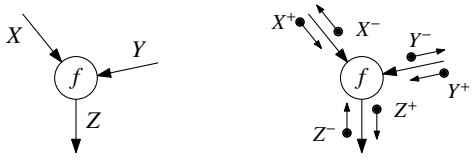


Figure 1: Types of messages

With this understanding, we can construct more complicated message passing circuits. For instance, given $h\colon X \to V \otimes U$, $g\colon V \otimes W \to I$ and $k\colon Y \otimes Z \to W$, we can construct the following circuit.

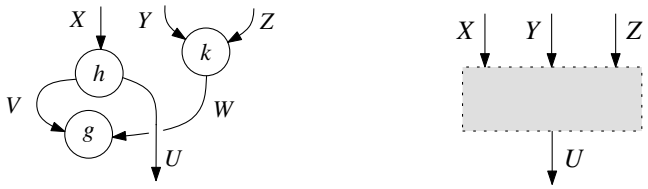


Figure 2: A circuit and its schematic notation

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 go into an infinite loop, in which case the whole computation will be blocked. Also, message passing inside the circuit may go on indefinitely in an infinite loop.

When only the interface of a circuit, but not its detailed implementation, is important, then we will only show a schematic notation of the circuit, as shown on the right in Fig. 2.

Functions. Importantly, it is possible to model higher-order functions as interactive entities. We can define an interface $X \multimap Y$ for functions from X to Y by letting $(X \multimap Y)^- := X^+ + Y^-$ and $(X \multimap Y)^+ := X^- + Y^+$. To understand this, suppose we have an implementation of this interface $X \multimap Y$ and implementation of interface X, as shown on the left in the figure below. Then we can connect them to obtain an implementation of interface Y as shown on the right below. In this figure, X^* denotes the interface obtained by exchanging question and answer types, i.e. $(X^-, X^+)^* := (X^+, X^-)$. The two unlabelled nodes are forwarding nodes. For each message arriving at either of these nodes, there is exactly one possible choice of outgoing wire of the same type, and the message is just sent along this wire. For example, the node of type $(X \multimap Y) \to X^* \otimes Y$ must be implemented by a program mapping $(X \multimap Y)^+ + (X^* \otimes Y)^- = (X^- + Y^+) + (X^+ + Y^-)$ to $(X \multimap Y)^- + (X^* \otimes Y)^+ = (X^+ + Y^-) + (X^- + Y^+)$ and there is just one canonical program of this type.

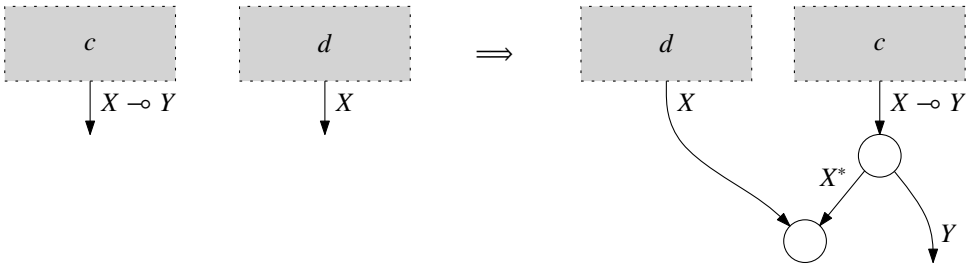


Figure 3: Application

One may think of $X \multimap Y$ as a way of formalising program parameterisation. A circuit implementing this interface has the property that when it is linked to a circuit implementing interface X as shown above, then the resulting circuit implements interface Y. In this sense, $X \multimap Y$ can be thought of as a type of functions from X to Y.

Another way of thinking about the interface $X \multimap Y = (X^+ + Y^-, X^- + Y^+)$ is in terms of interactive semantics, such as game semantics [2, 24] or the Geometry of Interaction [18]. 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.

Subexponentials. A final important construction in our simple model of interactive computation is that of *subexponentials*. The message-passing nodes that make up circuits are all stateless, being just programs of the stateless first-order language that we started with. They cannot remember any information from one request to the next.

This is a severe restriction on the expressiveness of functions. Suppose, for example, that *String* is an interface that allows us to access long strings by requesting the characters one by one. Predicates on such strings may be given the type $\mathit{String} \multimap [\mathtt{bool}]$, where `bool` is a type of booleans with values `true` and `false`. Suppose we want to implement such a predicate. Then we cannot store any information between one request to the string and the next. This means that we cannot even decide whether or not the first and the second character of the string are the same. We need to request both characters and there is no way to remember the first character while we request the second one.

Subexponentials are a way of storing first-order data across a request to some entity. For any interface X and any first-order type A, we define the subexponential $A \cdot X$ by $(A \times X^-, A \times X^+)$. A request $\langle a, m\rangle \in A \times X^-$ is intended to mean 'Please answer the request m and remind me of a when you give the answer'. An answer to $\langle a, m\rangle$ has the form $\langle a, p\rangle$, where p answers m. The invariant of subexponentials is that a is always returned unchanged. Because of this invariant, subexponentials can be used for data storage. The interface $A \cdot X$ in effect allows us to put any element of A to the side, make a request to X and retrieve the element of A when an answer arrives. This is much like local variables are preserved across function calls in machine code by using a call stack.

The above example of checking whether the first two characters of a string are the same, can be defined with type $((1 + \mathtt{char}) \cdot \mathit{String}) \multimap [\mathtt{bool}]$. The subexponential $1 + \mathtt{char}$ is chosen so that we do not store anything when making the first request to the string (hence the 1) and that we store the first returned character across the second request (hence the `char`).

One situation that will often arise is that we have a function of type $A \cdot X \multimap Y$ that we would like to apply to an argument of type X. Above we have only explained the application of $X \multimap Y$ to X. What we need is to turn an implementation of interface X into one of interface $A \cdot X$ in such a way that values of type A in the first component are always returned unchanged. On the level of circuits this is easy to do. We can change the circuit for X so that each wire of type Z becomes a wire of type $A \cdot Z$. The nodes pass on the value of type A unchanged and otherwise behave as before.

1.2. Overview

In this paper we develop the functional programming language INTML. It is constructed from a simple first-order programming language as a *base language*. INTML is obtained by adding to this base language a new class of types and terms that allows us to construct message passing circuits whose nodes are implemented in the base language. We call this the *interactive language*. The new types represent interfaces of wires in message-passing circuits, as outlined above. In INTML we use types of the form $[A]$, $X \otimes Y$ and $A \cdot X \multimap Y$. The terms of the interactive language, which are much like in a standard functional programming language, represent the circuits themselves.

By constructing INTML in this way we obtain a language that allows the programmer to work both in the base and in the interactive language. This reflects our experience with writing programs with logarithmic space usage. The interactive language allows the programmer to work with large data. The base language can be used work with small data and to manipulate pointers directly. It seems to be useful to keep such a distinction, as restrictions that need to be imposed on the manipulation of large data in order to remain in logarithmic space are not necessary for small values.

We introduce INTML in Section 2. The following Sections 3 and 4 contain programming examples and explain how logarithmic space graph algorithms can be represented in INTML. In Section 5 we show in detail how higher-order functions in INTML are translated to message-passing circuits and thus reduced to first-order programs. In Section 6 we use this reduction to prove soundness and completeness of INTML for the functions computable in logarithmic space.

This paper develops and expands the material first presented at ESOP 2010 [10]. It takes into account further developments from APLAS 2010 [11] and APLAS 2011 [43]. We discuss related work in Section 7.

1.3. 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 logarithmic 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 [32, 23], 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. [23], 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 [14]. Moreover, the proof of FLOGSPACE-completeness in this paper is completely straightforward. In earlier work, such as [36, 42], the completeness proofs were more involved. The expressiveness of IntML is further illustrated by the programming examples, such as in Section 4. Subexponentials not only give us control over space usage, they also afford an efficient treatment of controlled duplication. For example, Ghica and Smith [15] treat copying in the language by actual duplication of terms. Here we show how to allow copying by sharing 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.

Compared to [10], this paper includes full proofs of all main results presented there. Moreover, the characterisation of logarithmic space from [10] has been generalised to nondeterminism. The presence not only of higher-order functions but also the ability to define combinators, such as for call/cc or for block-scoped mutable variables like in Idealised Algol, is the main novelty of the language we present here. Finally, the relation with Joyal, Street & Verity's Int construction is analysed in more detail and precision. As such, this paper is not just a revised and extended version of previous work, but presents new contributions.

2. IntML

In this section we introduce IntML and its type system.

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

$$A \;::=\; \alpha \;\mid\; \mathtt{int} \;\mid\; 0 \;\mid\; 1 \;\mid\; A + A \;\mid\; A \times A \;.$$

They are built from type variables, a type of integers and product and coproduct types. We use lowercase Greek letters α, β, γ to range over type variables. The type $\mathtt{int}$ contains the unsigned integers of some fixed bit-width k. We define the interactive language without any assumptions on the choice of k. It shall be possible to evaluate any single program with respect to different choices of k. Indeed, we shall use $\mathtt{int}$ as a type of pointers to access large data, which means that we must choose k depending on the size of the data. In particular, we represent LOGSPACE-algorithms by programs that are evaluated with respect to different bit-width, depending on the size of the input; see Section 2.1.1.

The terms of the base language are parameterised by a signature of *constants* C. For any two types A and B, a signature specifies a set $C(A, B)$ of constants that take an argument of type A and that have type B. We assume that any signature contains constants for arithmetic operations $\{\mathtt{add}, \mathtt{sub}, \mathtt{mul}\} \subseteq C(\mathtt{int} \times \mathtt{int}, \mathtt{int})$ and for (in)equality tests on integers $\{\mathtt{eq}, \mathtt{lt}\} \subseteq C(\mathtt{int} \times \mathtt{int}, 1 + 1)$.

The terms of the base language are given by the grammar

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

in which x and y range over variables, c ranges over constants and n ranges over integer values. The base language is typed in a standard way. The typing judgement $\Gamma \vdash f : A$ states that term f has type A in context Γ. A context Γ is simply a finite mapping from variables to base types. The typing rules are given in Fig. 4. There are no linearity restrictions and the following weakening and contraction rules are admissible.

$$\frac{\Gamma \vdash f : B}{\Gamma,\, x\colon A \vdash f : B} \qquad \frac{\Gamma,\, x\colon A,\, y\colon A \vdash f : B}{\Gamma,\, z\colon A \vdash f[z/x, z/y] : B}$$

$$\frac{}{\Gamma,\, x\colon A \vdash x : A} \qquad \frac{}{\Gamma \vdash * : 1} \qquad \frac{c \in C(A,B) \qquad \Gamma \vdash f : A}{\Gamma \vdash c(f) : B}$$

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

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

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

$$\frac{\Gamma \vdash f : A \qquad \Gamma,\, x\colon A \vdash g : B + A}{\Gamma \vdash \mathtt{let}\ x = f\ \mathtt{loop}\ g : B}$$

Figure 4: Base Language Typing Rules

We explain the base language terms by defining its (unsurprising) operational semantics. It is given by a small-step reduction relation that is parameterised by a natural number k that represents the bit-width of all integer values. The set of *values* for bit-width k is defined by the following grammar

$$v, w \;::=\; x \;\big|\; * \;\big|\; n \;\big|\; \langle v, w\rangle \;\big|\; \mathtt{inl}(v) \;\big|\; \mathtt{inr}(w)\ ,$$

in which n is an integer that can be represented using k bits.

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

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

The special case ($\mathtt{let}\ x = f\ \mathtt{loop}\ \mathtt{inl}(g)$) evaluates the loop body exactly once. We write ($\mathtt{let}\ x = f\ \mathtt{in}\ g$) as an abbreviation for it. We shall also write 2 for 1 + 1 and use it as a type of booleans. We write $\mathtt{true}$ for $\mathtt{inl}(*)$, $\mathtt{false}$ for $\mathtt{inr}(*)$ and ($\mathtt{if}\ f\ \mathtt{then}\ g\ \mathtt{else}\ h$) for ($\mathtt{case}\ f\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g \mid \mathtt{inr}(y) \Rightarrow h$), where x and y are fresh variables.

The rewrite rules for products and coproducts are standard:

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

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

$$\begin{aligned}\mathtt{add}(\langle v, w\rangle) &\to_k \min(v + w, 2^k - 1)\\ \mathtt{sub}(\langle v, w\rangle) &\to_k \max(0, v - w)\\ \mathtt{mul}(\langle v, w\rangle) &\to_k \min(v * w, 2^k - 1)\end{aligned}$$

$$\begin{aligned}\mathtt{eq}(\langle v, v\rangle) &\to_k \mathtt{inl}(*)\\ \mathtt{eq}(\langle v, w\rangle) &\to_k \mathtt{inr}(*) \text{ if } v \neq w\\ \mathtt{lt}(\langle v, w\rangle) &\to_k \mathtt{inl}(*) \text{ if } v < w\\ \mathtt{lt}(\langle v, w\rangle) &\to_k \mathtt{inr}(*) \text{ if } v \geq w\end{aligned}$$

For integer constants we let $n \to_k 2^k - 1$ for any integer constant n that exceeds $2^k - 1$. Other integer constants are already values and have no reduction. Reduction is allowed in any *evaluation context*. These are contexts defined as

follows:

$$E, F, G \;::=\; [\cdot] \;\big|\; c(v_1, \ldots, v_n, E, f_1, \ldots, f_m) \;\big|\; \langle E, g\rangle \;\big|\; \langle v, E\rangle \;\big|\; \mathtt{fst}(E) \;\big|\; \mathtt{snd}(E)$$
$$\big|\; \mathtt{inl}(E) \;\big|\; \mathtt{inr}(E) \;\big|\; \mathtt{case}\ E\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g \mid \mathtt{inr}(y) \Rightarrow h$$
$$\big|\; \mathtt{let}\ x = E\ \mathtt{loop}\ g$$

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

Example 1. The following term f can be seen to compute the factorial function. For example, we have $f[4/x] \mapsto_k^* 24$, provided that k is large enough that that all numbers can be represented using k bits.

$$\begin{aligned}\mathtt{let}\ z = \langle x, 1\rangle\ \mathtt{loop}\ (\ &\mathtt{let}\ x = \mathtt{fst}(z)\ \mathtt{in}\\ &\mathtt{let}\ r = \mathtt{snd}(z)\ \mathtt{in}\\ &\mathtt{let}\ x' = \mathtt{sub}(\langle x, 1\rangle)\ \mathtt{in}\\ &\mathtt{case}\ \mathtt{eq}(\langle x', 0\rangle)\ \mathtt{of}\ \mathtt{inl}(*) \Rightarrow \mathtt{inl}(r)\\ &\qquad\qquad \mid \mathtt{inr}(*) \Rightarrow \mathtt{inr}(\langle x', \mathtt{mul}(\langle r, x\rangle)\rangle)))\end{aligned}$$

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, and any k, we have $f[\sigma] \mapsto_k^* v$ if and only if $g[\sigma] \mapsto_k^* v$.

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 s and r such that $\Gamma, x\colon A \vdash s : B$ and $\Gamma, y\colon B \vdash r : A$ are derivable and 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.

2.1.1. On Space Usage

Let us briefly discuss the space usage of base term reduction. In the rest of this paper we shall use values of type `int` to represent pointers into some given input, e.g. as pointers to graph nodes. We will vary the bit-width of base language integers with the size of the input, so that the integer type can represent such pointers. Given an input of size n, we will choose the bit-width to be $\lceil \log n \rceil$, so that `int` can count at least up to n. This is much like in finite model theory, where logics with counting have a universe as big as the size of the input [25].

With respect to space usage, it is therefore important to understand how much space is needed for term reduction as a function of the bit-width of the `int` type. Suppose the bit-width of `int` is k. Consider a term $x\colon \mathtt{int} \vdash f : A$ and a number n that can be represented with k bits. How much space does one need to reduce $f[n/x] \mapsto_k^* v$, i.e. to compute a value v? It is not hard to see that even in a naive implementation of the reduction relation $\mapsto_k$ by term rewriting the reduction needs not more than $O(k)$ space (see Lemma 22). The terms that can appear during reduction all have abstract syntax trees of constant depth, the constant depending only on the term f. They can therefore contain only a constant number of `int` values, each of which can be encoded using $O(k)$ bits. One can thus think of base term reduction as computing with a finite number of pointers.

2.2. Interactive Language

The base language is the basis of the definition of INTML. To complete the definition, we now introduce the interactive part of the language. This part adds higher-order functions to the base language. The interactive language can be understood as a way of constructing message-passing circuits, as outlined in the Introduction, in which each

message-passing node is implemented by a base language term. Nevertheless, the interactive language takes the form of a typed λ-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

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

in which A ranges over base language types.

The type $[A]$ represents computations that return a value of base type A if they terminate. One can think of $[A]$ much like a monadic type.

The type $X \otimes Y$ is a type of pairs. We use $\otimes$ instead of $\times$ to avoid confusion with the product types in the base language.

The type $A \cdot X \multimap Y$ is a type of functions from X to Y. It has an annotation A, a base language type, which we call a *subexponential annotation*. This annotation is important for space bound control. It can be understood as a stack shape annotation: Whenever the function makes a call to its argument, it puts a value of base type A on the stack. This value remains there until the call returns. The caller then retrieves the value from the stack for use in the rest of the computation. We write short $X \multimap Y$ for the type $1 \cdot X \multimap Y$. These are functions that do not need to reserve any stack space when making a call.

We believe that subexponential annotations are best considered as static space usage annotations that are inferred and tracked by a compiler. For a programmer it is usually sufficient to think of the function type $A \cdot X \multimap Y$ simply as $X \to Y$. Indeed, subexponential annotations do not appear explicitly in terms.

The terms of the interactive language are defined by the following grammar, in which f ranges over base language terms.

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

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:

- First, there are λ-abstraction and application ($\lambda x.\, t$ and $s\ t$, respectively), as well as standard terms for linear pairs ($\langle t, s \rangle$ and $\mathtt{let}\ \langle x, y \rangle = s\ \mathtt{in}\ t$).
- Base language terms can appear in interactive terms. First, the term $[f]$ denotes the computation that on request evaluates f to a value. The execution of a computation can be forced using the $\mathtt{let}$-term $\mathtt{let}\ [x] = s\ \mathtt{in}\ t$. The result of base-language computations can influence interactive terms. The $\mathtt{case}$ construct

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

  behaves as either $s[w/x]$ or $t[w/y]$, depending on whether v is $\mathtt{inl}(w)$ or $\mathtt{inr}(w)$. There is a third, more intricate, way of making use of the base language in the interactive language, namely the operator $\mathtt{hack}(x.f)$, on which we comment at the end of this section and in Section 3.
- The interactive calculus does not allow interactive variables to be used more than once. Controlled duplication is nevertheless possible by means of the term $\mathtt{copy}\ t\ \mathtt{as}\ x, y\ \mathtt{in}\ s$. This term captures a form of explicit sharing. There will be only one copy of t and any requests from s to either x or y will be handled by the same copy of t. It would be possible to formulate the type system without explicit $\mathtt{copy}$ terms, such that $\mathtt{copy}$ 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 type system for the interactive language derives typing judgements of the form

$$\Gamma \mid x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n \vdash t : Y \ . \tag{1}$$

In this judgement Γ is a base language context. The sequence $x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n$ is an *interactive context*. It is a list of variable declarations of the form $x_i : A_i \cdot X_i$. This declaration declares x_i to be a variable of the interactive type X_i.

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

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

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

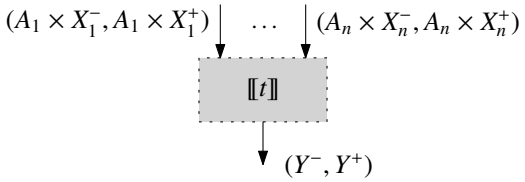
Figure 5: Interactive Typing: Structural Rules

The base language type A_i is a subexponential annotation that allows the type system to control the space usage of programs. An interactive context may contain at most one declaration for any variable. In the above typing judgement the variables $x_1, \ldots, x_n$ must be therefore be pairwise distinct.

To explain the subexponential annotations, we need to refer to message passing circuits that we have outlined in the Introduction and that will be defined in detail in Section 5. For each interactive type X, we define two base language types X^- and X^+ as follows. The informal meaning of these definitions are outlined in the Introduction.

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

The motivation of the interactive type system is that the term t in the typing judgement (1) will be translated to a message passing circuit $[\![t]\!]$ with the following interface.



Each of the message-passing nodes in this circuit is implemented by a base language term, which may make use of the variables from Γ. If the interactive term t is closed, then the circuit has no incoming wires and a single outgoing wire.

On the wire going out of the circuit we can send requests of base type Y^- and expect answers of base type Y^+. To compute the answer, the circuit may need to query any of the x_i. It does so by sending a request $\langle a, m \rangle : A_i \times X_i^-$ along the i-th input wire of the circuit. The circuit expects the answer to be of the form $\langle a, p \rangle : A_i \times X_i^+$, i.e. it expects the same a back with the answer. Since circuits are stateless and cannot store any information, this is useful for remembering the value a across a call to X_i.

We next introduce the typing rules. In them we write Φ, Ψ for the concatenation of two interactive contexts Φ and Ψ. Note that this is an interactive context only if no variable is declared in both Φ and Ψ. We write $A \cdot \Phi$ for the context obtained by replacing each declaration $x : B \cdot X$ in Φ with $x : (A \times B) \cdot X$. The structural rules in Fig. 5 allow one to manipulate the context Φ, which is treated *multiplicatively* in the logical rules. Rule Ax allows one to type a variable x if the context Φ consists of the sole declaration $x : 1 \cdot X$. The subexponential annotation 1 means that no additional space is needed when requests are sent to x. Weak and Exch are standard rules for weakening and context reordering. With rule Struct it is always possible to increase the subexponential annotations on a variable: in the premise of Struct, t relies on the guarantee that any value of type A that it sends together with a request of type X^- to x will be returned unchanged. If $A \triangleleft B$, then replacing A by B strengthens this guarantee. Any value of type A can be encoded into one of type B, be sent along with the request, and be decoded when it is returned. There may be many ways to encode values of type A into type B. It does not matter which one we choose. All that matters for intml is that by encoding and then decoding we get back the value we started from. The definition of $A \triangleleft B$ by a retraction is just enough to guarantee this property. Instead of the general semantic definition of $A \triangleleft B$ by retractions, one may also use syntactic approximations. We refer to [11] for an exploration of possible choices.

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

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

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

Figure 6: Interactive Typing: Introductions and Eliminations

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

Figure 7: Interactive Typing: Cross-Eliminations

The structural rules are set up so that no variable in an interactive context Φ may be used more than once. However, the calculus does support explicit duplication. A specific term construct serves this purpose, and can be typed by rule Copy. The reader may want to refer to Fig. 12 to get an idea of how the circuits for s and t are connected.

The introduction and elimination rules for the three type formers $[-]$, $\otimes$ and $\multimap$ appear in Fig. 6. These rules are fairly standard for a linear λ-calculus. Only perhaps rules [ ]I and [ ]E for embedding base language computations and for forcing their execution are of note. In rule [ ]E the term ($\mathtt{let}\,[x] = s\,\mathtt{in}\,t$) allows one to force the computation 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. 7 gives the rules for eliminating base language terms over interactive terms. With our simple base language, we only need a single rule $+\mathrm{E}^{\mathrm{c}}$, which 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 $\mathtt{let}\,[x] = s\,\mathtt{in}\,\mathtt{case}\ x\ \mathtt{of}\ \mathtt{inl}(y) \Rightarrow t \mid \mathtt{inr}(z) \Rightarrow u$. With the exception of Section 6.2, all results in this paper remain true verbatim also if one allows arbitrary base terms for v in rule $+\mathrm{E}^{\mathrm{c}}$.

The intml type system is completed by rule Hack in Fig. 8. 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 a single outgoing wire of type X and no other wires. 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 detailed knowledge of the runtime system.

The other terms can however be understood without any reference to circuits. Below we 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.

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

Figure 8: Interactive Typing: Hacking

2.3. Typing Examples

We give a first simple example to explain the features of the INTML type system. The term

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

defines a function that evaluates a thunk y, binds the returned value to variable x and then invokes a function f. When f asks for either of its arguments, the value x is returned as answer.

In the type, α, β, γ and δ are type variables that can be instantiated with arbitrary types. That β and γ are type variables means that no restriction is imposed on the subexponentials in the functions that may be given as argument for f. That the type of the argument f appears under $\alpha \cdot (\dots)$ tells us that a value of type α is sent along with any request to the function f, and this value is expected to be returned unchanged. In this case, this is the value of the variable x, which has type α.

The typing judgement of the term is derivable as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{- \mid y : 1 \cdot [\alpha] \vdash y : [\alpha]}\,\text{Ax} \quad \dfrac{}{x : \alpha \mid f : 1 \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]) \vdash f\ [x]\ [x] : [\delta]}\,\Pi}{- \mid y : 1 \cdot [\alpha], f : (\alpha \times 1) \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]) \vdash \texttt{let}\ [x] = y\ \texttt{in}\ f\ [x]\ [x] : [\delta]}\,[\,]\text{E}}{- \mid f : (\alpha \times 1) \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]), y : 1 \cdot [\alpha] \vdash \texttt{let}\ [x] = y\ \texttt{in}\ f\ [x]\ [x] : [\delta]}\,\text{Exch}}{- \mid f : (\alpha \times 1) \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]) \vdash \lambda y.\, \texttt{let}\ [x] = y\ \texttt{in}\ f\ [x]\ [x] : [\alpha] \multimap [\delta]}\,\multimap\text{I}}{- \mid f : \alpha \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]) \vdash \lambda y.\, \texttt{let}\ [x] = y\ \texttt{in}\ f\ [x]\ [x] : [\alpha] \multimap [\delta]}\,\textsc{Struct}}{- \mid - \vdash \lambda f.\, \lambda y.\, \texttt{let}\ [x] = y\ \texttt{in}\ f\ [x]\ [x] : \alpha \cdot (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta]) \multimap [\alpha] \multimap [\delta]}\,\multimap\text{I}$$

The sub-derivation Π is shown below. In it we abbreviate $(\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta])$ by T.

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \mid f : 1 \cdot T \vdash f : (\beta \cdot [\alpha] \multimap \gamma \cdot [\alpha] \multimap [\delta])}\,\text{Ax} \quad [\,]\text{I}\ \dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{x : \alpha \mid - \vdash [x] : [\alpha]}}{x : \alpha \mid f : 1 \cdot T \vdash f\ [x] : \gamma \cdot [\alpha] \multimap [\delta]}\,\multimap\text{E} \quad [\,]\text{I}\ \dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{x : \alpha \mid - \vdash [x] : [\alpha]}}{x : \alpha \mid f : 1 \cdot T \vdash f\ [x]\ [x] : [\delta]}\,\multimap\text{E}$$

Note that the similar term $\lambda f.\, \lambda y.\, f\ y\ y$ is not well-typed in INTML, as the interactive variable y is used twice. However, the following term can be typed for any X and Y; a typing derivation is constructed below.

$$\lambda f.\, \lambda y.\, \texttt{copy}\ y\ \texttt{as}\ y_1, y_2\ \texttt{in}\ f\ y_1\ y_2\ :\ (\beta \cdot X \multimap \gamma \cdot X \multimap Y) \multimap (\beta + \gamma) \cdot X \multimap Y$$

This term passes the same thunk y to both arguments of f. If f forces both its arguments one after the other, then y will be computed twice. This is in contrast to the above term $\texttt{let}\ [x] = y\ \texttt{in}\ f\ [x]\ [x]$, which first forces the thunk y, binds its return value to x and then passes to f the thunks that immediately return x. In this case the computation in y is executed only once.

2.3.1. Finding Subexponential Annotations

Let us use the typing judgement

$$\lambda f.\, \lambda y.\, \texttt{copy}\ y\ \texttt{as}\ y_1, y_2\ \texttt{in}\ f\ y_1\ y_2\ :\ (\beta \cdot X \multimap \gamma \cdot X \multimap Y) \multimap (\beta + \gamma) \cdot X \multimap Y$$

as an example to explain how subexponential annotations can be found.

The idea is to first use Hindley-Milner type inference to construct a skeleton of the typing derivation without subexponentials. If we leave out all subexponential annotations and use $X \to Y$ in place of $A \cdot X \multimap Y$, then we obtain the following skeleton of a derivation for the above term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{- \mid f : X \to X \to Y \vdash f : X \to X \to Y} \quad \dfrac{}{- \mid y_1 : X \vdash y_1 : X}}{- \mid f : X \to X \to Y, y_1 : X \vdash f\ y_1 : X \to Y} \quad \dfrac{}{- \mid y_2 : X \vdash y_2 : X}}{- \mid f : X \to X \to Y, y_1 : X, y_2 : X \vdash f\ y_1\ y_2 : Y}}{- \mid f : X \to X \to Y, y : X \vdash \texttt{copy}\ y\ \texttt{as}\ y_1, y_2\ \texttt{in}\ f\ y_1\ y_2 : Y}}{- \mid f : X \to X \to Y \vdash \lambda y.\, \texttt{copy}\ y\ \texttt{as}\ y_1, y_2\ \texttt{in}\ f\ y_1\ y_2 : X \to Y}}{- \mid - \vdash \lambda f.\, \lambda y.\, \texttt{copy}\ y\ \texttt{as}\ y_1, y_2\ \texttt{in}\ f\ y_1\ y_2 : (X \to X \to Y) \to X \to Y}$$

This is not a proper INTML typing derivation. To turn it into one we need to add subexponential annotations. To do so, add fresh type variables for all missing subexponential annotations. In order to obtain a correct typing derivation, we will need to insert rule STRUCT in a number of places. Without loss of generality, we can assume that STRUCT is being applied after every rule to each variable in the interactive context. Let us denote these applications of STRUCT by double lines. We obtain the following annotated skeleton of the derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{- \mid f : \alpha_1 \cdot (\beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y) \vdash f : \beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y} \quad \dfrac{}{- \mid y_1 : \alpha_2 \cdot X \vdash y_1 : X}}{- \mid f : \alpha_3 \cdot (\beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y),\, y_1 : \alpha_4 \cdot X \vdash f\ y_1 : \beta_2 \cdot X \multimap Y} \quad \dfrac{}{- \mid y_2 : \alpha_5 \cdot X \vdash y_2 : X}}{- \mid f : \alpha_6 \cdot (\beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y),\, y_1 : \alpha_7 \cdot X,\, y_2 : \alpha_8 \cdot X \vdash f\ y_1\ y_2 : Y}}{- \mid f : \alpha_9 \cdot (\beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y),\, y : \alpha_{10} \cdot X \vdash \mathtt{copy}\ y\ \mathtt{as}\ y_1, y_2\ \mathtt{in}\ f\ y_1\ y_2 : Y}}{- \mid f : \alpha_{11} \cdot (\beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y) \vdash \lambda y.\, \mathtt{copy}\ y\ \mathtt{as}\ y_1, y_2\ \mathtt{in}\ f\ y_1\ y_2 : \alpha_{10} \cdot X \multimap Y}}{- \mid - \vdash \lambda f.\, \lambda y.\, \mathtt{copy}\ y\ \mathtt{as}\ y_1, y_2\ \mathtt{in}\ f\ y_1\ y_2 : \alpha_{11} \cdot (\beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y) \multimap \alpha_{10} \cdot X \multimap Y}$$

Now we simply gather the constraints on the type variables α_i and β_i in the side conditions of all the rules in the derivation and we solve them. The side constraints in the above skeleton are:

- $1 \triangleleft \alpha_1$, $1 \triangleleft \alpha_2$ and $1 \triangleleft \alpha_5$ from the application of STRUCT after the axiom rules.
- $\alpha_1 \triangleleft \alpha_3$ and $(\beta_1 \times \alpha_2) \triangleleft \alpha_4$ from the application rule with conclusion $f\ y_1$. The conclusion of the application rule has context $f : \alpha_1 \cdot (\beta_1 \cdot X \multimap \beta_2 \cdot X \multimap Y), y_1 : (\beta_1 \times \alpha) \cdot X$ and applying STRUCT to each of the two variables gives the two constraints.
- $\alpha_3 \triangleleft \alpha_6$, $\alpha_4 \triangleleft \alpha_7$ and $(\beta_2 \times \alpha_5) \triangleleft \alpha_8$ from the application rule with conclusion $f\ y_1\ y_2$.
- $\alpha_6 \triangleleft \alpha_9$ and $(\alpha_7 + \alpha_8) \triangleleft \alpha_{10}$ from the copying rule.
- $\alpha_9 \triangleleft \alpha_{11}$ from the abstraction rule.

Notice that all constraints have the form $A \triangleleft \alpha$, where α is a type variable that does not appear in A. A number of such constraints $A_1 \triangleleft \alpha, \ldots, A_n \triangleleft \alpha$ can be solved simply by setting $\alpha := A_1 + \cdots + A_n$. It is not hard to see that $A_i \triangleleft (A_1 + \cdots + A_n)$ holds for $i = 1, \ldots, n$. This simple observation can be used to solve the constraints one by one. First we gather all the constraints with α_1 as an upper bound and solve for α_1, then we continue with α_2 and so on.

Using this simple idea, we can find the following solution to the constraints completely mechanically:

$$\begin{array}{llllll} \alpha_1 := 1 & \alpha_2 := 1 & \alpha_3 := 1 & \alpha_4 := \beta_1 \times 1 & \alpha_5 := 1 & \alpha_6 := 1 \\ \alpha_7 := \beta_1 \times 1 & \alpha_8 := \beta_2 \times 1 & \alpha_9 := 1 & \alpha_{10} := (\beta_1 \times 1) + (\beta_2 \times 1) & \alpha_{11} := 1 & \end{array}$$

The definition of $\triangleleft$ further allows us to apply simplifications, such as using β_1 instead of $\beta_1 \times 1$. We obtain a typing derivation with the claimed type.

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 outlined in the example above. It is described in more detail in [11].

We have found that in practice when writing programs one usually 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 $\lambda f.\, \lambda y.\, \mathtt{let}\ [x] = y\ \mathtt{in}\ f\ [x]\ [x]$. 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.

We note that with the general definition of $\triangleleft$, type inference is a hard problem. The simple and fast type inference procedure described above is incomplete. It may fail to find a type even when one exists. Moreover, even if it finds a typing derivation, it may not find the best one. Often, there will be many solutions for subexponential constraints. The

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

Figure 9: Interactive Language: Closed Reduction Semantics

choice of the solution will affect the space usage of compiled programs. If we use rule Struct with side condition $A \triangleleft B$, then in the final implementation the type B will be used to encode values of type A. Thus, the smaller we choose B, the less space will be used in the final program. This should illustrate why with the general definition of $\triangleleft$, one should expect the inference of the best type to be hard.

In [11] we study the hardness of type inference. We consider a number of possible syntactic approximations of $\triangleleft$ and study the computational hardness of type inference with such choices.

2.5. Operational Semantics

While the interactive language will be implemented using message passing circuits, we specify its meaning using a standard reduction-based operational semantics. We emphasise that the following operational semantics is a method of specification only.

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:

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

The evaluation contexts do not uniquely specify an evaluation order. Since we implement reduction only in the base language, it is not necessary to do so. We will show soundness for any choice of evaluation order.

The rewrite rules are given in Fig. 9, where s and t are assumed to be *closed* terms, i.e. terms not containing any free occurrence of any variable. Note that the `case`-terms is restricted to case distinction on values by the type system. A term t *reduces* to s, written $t \mapsto_k s$, if there are u, q such that $u \to_k q$, $t = E[u]$ and $s = E[q]$. As in the base language, the parameter k represents the bit-width of integers. In Section 5.6, we show that the translation of interactive terms to message passing circuits is sound with respect to reduction: $t \mapsto_k s$ 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 ($\texttt{let}\ [x] = t\ \texttt{in}\ f$) 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 16 in Section 5.6, which is why we have not included it in the definition.

2.6. On Large Data and Space Usage

Base language programs can be understood as programs that store a constant number of pointers. If the bit-width of integers is k, then any base language program evaluates in space $O(k)$. The same is true for the interactive language. We will show that the interactive part of the language can be compiled to the base language. Even though the interactive language allows higher-order functions, it can nevertheless be evaluated in space $O(k)$. Thus, IntML can be seen as a higher-order language for the computation with a finite number of pointers.

Let us outline with a simple concrete example how we use higher-order functions in INTML to implement sublinear space algorithms. We consider programs that work on long bit vectors, i.e. long sequences of 0s and 1s. The base language type $2 = 1 + 1$ can be seen as a type of bits, where $\mathtt{inl}(*)$ stands for 0 and $\mathtt{inr}(*)$ stands for 1. Bit vectors of length n can be represented as a base type value of type $2 \times \cdots \times 2$ (n factors), but this representation would not be space efficient. A value of this type has size $O(n)$, i.e. it will need linear space.

A higher-order representation of bit vectors is more space-efficient. If we choose the bit-width of the base language to be $\log n$, then the *positions* in the bit vector can be represented by values of type $\mathtt{int}$. Bit vectors of length n can then be represented by functions of type

$$A \cdot [\mathtt{int}] \multimap [2] \ .$$

(There may be more than n bits, but we can just ignore superfluous elements.)

First, we explain how large bit vectors can be given as input to INTML programs. Suppose we have a program of type $A \cdot (1 \cdot [\mathtt{int}] \multimap [2]) \multimap [2]$ that represents a predicate on bit vectors. We make no restriction on the subexponential and allow any type A (we cannot use a type variable as the program may require a type of a particular form, such as $1 + B$). Suppose that we have a concrete large vector (e.g. stored in some database) of length n that we would like to pass to the program. The program will translate to a circuit with one outgoing wire with interface (X^-, X^+).

$$\begin{array}{rcccccccccccc} X & = & A & \cdot & (& 1 & \cdot & [\mathtt{int}] & \multimap & [2] &) & \multimap & [2] \\ X^- & = & A & \times & (& 1 & \times & 1 & + & 2 &) & + & 1 \\ X^+ & = & A & \times & (& 1 & \times & \mathtt{int} & + & 1 &) & + & 2 \end{array}$$

To evaluate the program with the bit vector as input, we fix the bit-length of integers to be $\lceil \log n \rceil$ and then use the circuit to evaluate the result of the program. 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 $\mathtt{inr}(*) \colon X^-$ ('What is the value of the predicate?') to the circuit. A typical answer will be $\mathtt{inl}(\langle a, \mathtt{inr}(*)\rangle) \colon X^+$ ('I would like to know a bit of the input vector. Also, please remind me of a when you answer.'). We may answer by sending $\mathtt{inl}(\langle a, \mathtt{inl}(\langle *, *\rangle)\rangle) \colon X^-$ ('Which bit would you like to know? Also, here is the a you asked me to return.'). The reply will be $\mathtt{inl}(\langle a, \mathtt{inl}(\langle *, n\rangle)\rangle) \colon X^+$ ('I would like to know the i-th bit.'). We may then look up the i-th bit x of the large vector and return it by sending $\mathtt{inl}(\langle a, \mathtt{inr}(x)\rangle) \colon X^-$ ('The requested bit is x.'). This may continue for a while until we receive the final answer $\mathtt{inr}(p) \colon X^+$ ('The value of the predicate is p.'). Notice that in X^- and X^+ the summands for requests and answers line up with the interactive type in X. For instance, $\mathtt{inr}(*) \colon X^-$ and $\mathtt{inr}(a) \colon X^+$ both pertain the type [2].

If large data appears as the output of programs, then it can be requested in a similar way. Suppose we have a closed INTML term of type $B \cdot [\mathtt{int}] \multimap [2]$, again without restriction on the type B. The term will be translated to a circuit with a single outgoing wire with interface (Y^-, Y^+), in which Y^- and Y^+ are base language types:

$$\begin{array}{rcccccc} Y & = & B & \cdot & [\mathtt{int}] & \multimap & [2] \\ Y^- & = & B & \times & \mathtt{int} & + & 1 \\ Y^+ & = & B & \times & 1 & + & 2 \end{array}$$

To compute the bit vector we again interact with the circuit. We begin by sending it the message $\mathtt{inr}(*) \colon Y^-$ ('I would like to know the value of the thunk of type [2] that the function returns.'). The circuit would then typically reply with $\mathtt{inl}(b, *) \colon Y^+$ ('I need to know the argument of the function, i.e. the value of the thunk of type $[\mathtt{int}]$. When you answer this request, please also remind me of value b.'). Suppose we want to know the character in position $i \colon \mathtt{int}$. We then send the message $\mathtt{inl}(b, i) \colon Y^-$ ('The argument thunk has value i. And you asked me to remind you of value b.'). To this, the circuit will typically reply $\mathtt{inr}(p) \colon Y^+$ ('The thunk returned by the function has value p.'). The whole bit vector represented by the term of type $A \cdot [\mathtt{int}] \multimap [2]$ can now be reconstructed by repeatedly querying the circuit.

Notice that the messages that we send to and receive from the circuits are all base language values of size $O(k)$, where k is the bit-width of integers. The INTML type system guarantees that the same is true 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^k. It is this basic observation that leads to logarithmic space bounds.

Direct interaction with circuits is needed only to supply the input value and to read off the output value. Intermediate values can be defined using the terms of the interactive language. For example the exclusive or of two bit vectors can

be implemented as an interactive program

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

of type

$$(\alpha \cdot [\mathtt{int}] \multimap [2]) \multimap 2 \cdot (\beta \cdot [\mathtt{int}] \multimap [2]) \multimap (\alpha + 2 \times \beta) \cdot [\mathtt{int}] \multimap [2]\ .$$

In it, $\mathtt{xor}$ denotes a base language term $x\colon 2,\ y\colon 2 \vdash \mathtt{xor}(x, y) : 2$ that computes the exclusive or of individual bits, i.e. that satisfies $\mathtt{xor}(x, y)[a/x, b/y] \mapsto_k^* a \oplus b$ for all (encodings of) bits a and b. We omit the standard definition by case distinction. A typing derivation be found in Fig. 17 on page 51. The program for the exclusive or of bit vectors manipulates bit vectors of length 2^k. Its circuit, however, works by passing only messages of size $O(k)$. There are other ways of writing the exclusive or 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:

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

The type reflects the changed memory behaviour:

$$\mathtt{int} \cdot (\alpha \cdot [\mathtt{int}] \multimap [2]) \multimap (\mathtt{int} \times 2) \cdot (\beta \cdot [\mathtt{int}] \multimap [2]) \multimap [\mathtt{int}] \multimap [2].$$

A typing derivation be found in Fig. 17 on page 51.

3. Direct Definition of Combinators

The interactive language is a simple linear λ-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.

3.1. 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 $(s\ t) = [\mathtt{inr}(v)]$ and $(\mathtt{loop}\ s\ t) = \mathtt{loop}\ s\ [v]$ if $(s\ t) = [\mathtt{inl}(v)]$. We implement this combinator directly by $\mathtt{loop} := \mathtt{hack}(x.l)$ for a suitable base language term l of type $x\colon X^- \vdash l : X^+$, where:

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

The term $\mathtt{hack}(x.l)$ 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:

$$l[\mathtt{inr}(\mathtt{inr}(*))/x] \mapsto_k^* \mathtt{inr}(\mathtt{inl}(*)) \tag{2}$$

$$l[\mathtt{inr}(\mathtt{inl}(a))/x] \mapsto_k^* \mathtt{inl}(\langle a, \mathtt{inr}(*)\rangle) \tag{3}$$

$$l[\mathtt{inl}(\langle a, \mathtt{inr}(\mathtt{inr}(b))\rangle)/x] \mapsto_k^* \mathtt{inr}(\mathtt{inr}(b)) \tag{4}$$

$$l[\mathtt{inl}(\langle a, \mathtt{inr}(\mathtt{inl}(a'))\rangle)/x] \mapsto_k^* \mathtt{inl}(\langle a', \mathtt{inr}(*)\rangle) \tag{5}$$

$$l[\mathtt{inl}(\langle a, \mathtt{inl}(\langle g, *\rangle)\rangle)/x] \mapsto_k^* \mathtt{inl}(\langle a, \mathtt{inl}(\langle g, a\rangle)\rangle) \tag{6}$$

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 $\texttt{loop}\ s\ t$ 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 $\texttt{inr}(b)$, then we are done and give b as output (4). If s returns value $\texttt{inl}(a')$, then we overwrite the memory cell content with a' and restart the step function s by asking for its result (5).

In Section 5 we will make precise the circuit implementation of INTML and in Section 5.6 we show that is correctly implements the operational semantics. With the techniques of Section 5.6, one can show:

Proposition 2. *The following reduction is sound (in the sense of Theorem 13) in the interactive language.*

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

The proof goes by spelling out the circuit implementing the right-hand side of the reduction. 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 $x\colon A \vdash \mathit{islast}_A(x) : 2$, 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 $\texttt{inl}(*)$. Otherwise the value is always $\texttt{inr}(*)$. 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.

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

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

3.2. 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:

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

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

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

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

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

$$c[\texttt{inl}(\langle *, \texttt{inr}(a)\rangle)/x] \mapsto_k^* \texttt{inr}(a) \tag{8}$$

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

$$c[\texttt{inl}(\langle *, \texttt{inl}(g, \texttt{inl}(a))\rangle)/x] \mapsto_k^* \texttt{inr}(a) \tag{10}$$

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. [30]. As an example for `callcc`, we show the implementation of a *tail recursion combinator* `tailfix`.

$$\begin{aligned}&\texttt{tailfix}\colon \alpha \cdot (\gamma \cdot ([\alpha] \multimap [\beta]) \multimap (\delta \cdot [\alpha] \multimap [\beta])) \multimap [\alpha] \multimap [\beta]\\ &\texttt{tailfix} = \quad \lambda f.\, \texttt{loop}\ (\lambda x.\, \texttt{callcc}\ (\lambda k.\ \texttt{let}\ [w] = f\ (\lambda y.\, \texttt{let}\ [v] = y\ \texttt{in}\ k\ [\texttt{continue}(v)])\ x\ \texttt{in}\ [\texttt{return}(w)]))\end{aligned}$$

This combinator may be used to define functions by tail recursion. The tail recursive definition of factorial may then be defined as shown below.

$$\begin{aligned}&\texttt{fact}\colon [\texttt{int}] \multimap [\texttt{int}]\\ &\texttt{fact} = \quad \lambda x.\, \texttt{let}\ [v] = x\ \texttt{in}\ \texttt{fact\_aux}\ [\langle v, 1\rangle]\\ &\texttt{fact\_aux}\colon [\texttt{int} \times \texttt{int}] \multimap [\texttt{int}]\\ &\texttt{fact\_aux} = \quad \texttt{tailfix}\ (\lambda \mathit{fact\_aux}.\, \lambda x.\ \texttt{let}\ [\langle i, \mathit{acc}\rangle] = x\ \texttt{in}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{if}\ \texttt{eq}(\langle i, 0\rangle)\ \texttt{then}\ [\mathit{acc}]\ \texttt{else}\ \mathit{fact\_aux}\ [\langle \texttt{sub}(i, 1), \texttt{mul}(i, \mathit{acc})\rangle])\end{aligned}$$

The definition of `tailfix` using `callcc` may look complicated. It is also possible to define `tailfix` directly using `hack`, in which case the definition is much like that of `loop`.

4. Larger Examples

While the syntax of INTML is quite minimal, it is already possible to write meaningful programs. In this section we give examples to substantiate this claim.

4.1. Logarithmic Space Graph Algorithms

First we show how typical algorithms with logarithmic space usage can be implemented. We implement two classic LOGSPACE-complete decision problems [9]: the problem of deciding whether a given undirected graph has a cycle, and the problem of deciding whether two nodes in an undirected forest are connected by a path. We implement the first problem directly and solve the second one by reduction to the first. Section 4.1.2 is based on [11].

We use a higher-order representation of graphs, where a graph is given by its size together with a binary predicate that represents the set of edges. A pair of a number n and a predicate $\chi\colon \mathbb{N} \times \mathbb{N} \to 2$ represents the graph with vertex set $V = \{0, \ldots, n-1\}$ and edges $E = \{\langle m, n\rangle \in V^2 \mid \chi(m, n) = \texttt{true}\}$. In INTML such pairs can be represented by the following interactive type.

$$\mathit{Graph}_A = [\texttt{int}] \otimes (A \cdot [\texttt{int} \times \texttt{int}] \multimap [2])$$

In the rest of this section we will elide subexponential annotations. Our aim in this section is to show that FLOGSPACE functions can be implemented in INTML. For this purpose it is sufficient to know that subexponential annotations can be found. The programs in this section were written using a prototype implementation that infers subexponential annotations using the simple method outlined in Section 2.3.1, see also [11]. If one is interested in concrete space bounds, one can use type inference to compute the precise annotations, but in order to know that the implemented algorithm is in FLOGSPACE it is enough to know that annotations exist. Therefore, instead of Graph_A and $A \cdot X \multimap Y$ we write just *Graph* and $X \to Y$, with the understanding that it is possible to write down suitable annotations.

To define functions on graphs it is convenient to define boolean functions also on the interactive level. Case distinction *if*: $[2] \to X \to X \to X$ can be defined by $\lambda b.\, \lambda x.\, \lambda y.\, \texttt{let}\ [c] = b\ \texttt{in}\ \texttt{case}\ c\ \texttt{of}\ \texttt{inl}(t) \Rightarrow x \mid \texttt{inr}(f) \Rightarrow y$ for arbitrary X. Boolean functions for negation, conjunction and disjunction can similarly be defined with types *neg*: $[2] \to [2]$, *and*: $[2] \to [2] \to [2]$ and *or*: $[2] \to [2] \to [2]$.

In graph algorithms we frequently need to iterate over all nodes of a graph. It is possible to do so directly using the `loop` combinator. It may be used to define a fold-like function for this purpose, just like $fold_{\texttt{int}}$ in Section 3. In this section we need only a function *forall* that checks a given property p for all nodes of a graph. It can be defined concisely using $fold_{\texttt{int}}$.

$$
\begin{aligned}
&\mathit{forall}\colon [\texttt{int}] \to ([\texttt{int}] \to [2]) \to [2]\\
&\mathit{forall} = \lambda \mathit{size}.\, \lambda p.\ \texttt{let}\ [n] = \mathit{size}\ \texttt{in}\\
&\qquad\qquad \mathit{fold}_{\texttt{int}}\ (\lambda i.\, \lambda r.\ \texttt{let}\ [m] = i\ \texttt{in}\\
&\qquad\qquad\qquad\qquad (\mathit{and}\ r\ (\mathit{or}\ [\texttt{lt}(m,n)]\ (p\ [m])))\\
&\qquad\qquad\qquad [\texttt{true}]
\end{aligned}
$$

Using higher-order functions we can write functions to manipulate graphs. For example, a function that adds an edge to a graph can be defined as follows.

$$
\begin{aligned}
&\mathit{addEdge} : \mathit{Graph} \to [\texttt{int} \times \texttt{int}] \to \mathit{Graph}\\
&\mathit{addEdge} = \ \lambda \mathit{graph}.\, \texttt{let}\ \langle \mathit{size}, \mathit{edge}\rangle = \mathit{graph}\ \texttt{in}\\
&\qquad \lambda \mathit{newedge}.\, \langle \mathit{size}, \lambda p.\ \texttt{let}\ [\langle m, n\rangle] = p\ \texttt{in}\\
&\qquad\qquad\qquad\qquad \texttt{let}\ [\langle m', n'\rangle] = \mathit{newedge}\ \texttt{in}\\
&\qquad\qquad\qquad\qquad \mathit{or}\ (\mathit{edge}\ [\langle m, n\rangle])\\
&\qquad\qquad\qquad\qquad\quad [\texttt{if}\ \texttt{eq}(\langle m, m'\rangle)\ \texttt{then}\ \texttt{eq}(\langle n, n'\rangle)\ \texttt{else}\ \texttt{false}])
\end{aligned}
$$

In this definition we have allowed ourselves pattern matching notation to write $\texttt{let}\ [\langle m, n\rangle] = p\ \texttt{in}\ t$ instead of the more verbose $\texttt{let}\ [p'] = p\ \texttt{in}\ t[\texttt{fst}(p')/m, \texttt{snd}(p')/n]$.

4.1.1. Checking for Absence of Self-Loops

A first simple graph algorithm is a test that checks if a given graph contains a node with a self-loop, i.e. an edge to itself. It may be defined as follows:

$$
\begin{aligned}
&\mathit{noSelfLoop} : \mathit{Graph} \to [\texttt{int} \times \texttt{int}] \to \mathit{Graph}\\
&\mathit{noSelfLoop} = \lambda \mathit{graph}.\ \texttt{let}\ \langle \mathit{size}, \mathit{edge}\rangle = \mathit{graph}\ \texttt{in}\\
&\qquad\qquad\qquad\qquad \texttt{let}\ [n] = \mathit{size}\ \texttt{in}\\
&\qquad\qquad\qquad\qquad \mathit{forall}\ [n]\ (\lambda s'.\, \texttt{let}\ [s] = s'\ \texttt{in}\\
&\qquad\qquad\qquad\qquad\quad \mathit{forall}\ [n]\ (\lambda d'.\, \texttt{let}\ [d] = d'\ \texttt{in}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{neg}\ (\mathit{and}\ (\mathit{edge}\ [\langle s, d\rangle])\ [\texttt{eq}(\langle s, d\rangle)])))
\end{aligned}
$$

4.1.2. Checking for Absence of Undirected Cycles

The algorithm of Cook & McKenzie for checking acyclicity of undirected graphs can be explained using the notion of a right-hand walk. A right-hand walk is like a walk in a labyrinth where one always keeps his right hand on the wall. One imagines the graph edges to be corridors and the nodes to be junctions. The edges connected to each node are ordered as if the node were a junction of the arriving corridors. Cook & McKenzie's observation then is that an undirected graph is acyclic if and only if any right-hand walk will return to its starting point by traversing the edge over which it has left this point in the opposite direction. This observation leads to a FLOGSPACE algorithm, since this property can be checked by following right-hand walks, and for this one needs to keep in memory just a few graph nodes.

In order to implement right-hand walks, we first need a function

$$\mathit{nextEdge}\colon \mathit{Graph} \to [\texttt{int} \times \texttt{int}] \to [\texttt{int} \times \texttt{int}]$$

that takes an edge $\langle s, d\rangle$ and returns the next edge $\langle s, e\rangle$ emanating from the same source. To implement this function, we first define a base language term $x\colon \texttt{int} \vdash \mathit{next}(x) : \texttt{int}$ by

$$\mathit{next}(x) = \texttt{let}\ x' = \texttt{add}(\langle x, 1\rangle)\ \texttt{in}\ \texttt{if}\ \texttt{eq}(\langle x', x\rangle)\ \texttt{then}\ 0\ \texttt{else}\ x' \ .$$

It wraps around to 0 once the maximum `int`-value is exceeded.

Then we can define *nextEdge* as follows. Given $\langle s, d\rangle$, it starts from $\langle s, next(d)\rangle$ and applies *next* to the second component of the pair until it has found an edge.

$$
\begin{aligned}
&\mathit{nextEdge}\colon \mathit{Graph} \to [\texttt{int} \times \texttt{int}] \to [\texttt{int} \times \texttt{int}]\\
&\mathit{nextEdge} = \lambda \mathit{graph}.\ \texttt{let}\ \langle \mathit{size}, \mathit{edge}\rangle = \mathit{graph}\ \texttt{in}\\
&\qquad\qquad \texttt{loop}\ (\ \lambda \mathit{edge}.\\
&\qquad\qquad\qquad \texttt{let}\ [\langle \mathit{src}, \mathit{dst}\rangle] = \mathit{edge}\ \texttt{in}\\
&\qquad\qquad\qquad \texttt{let}\ [c'] = [\langle \mathit{src}, \mathit{next}\ \mathit{dst}\rangle]\ \texttt{in}\\
&\qquad\qquad\qquad \mathit{if}\ (\mathit{edge}\ [c'])\ [\texttt{return}(c')]\ [\texttt{continue}(c')])
\end{aligned}
$$

We can then write a function *checkPath*, which follows a right-hand-rule walk starting from some given edge and checks if this walk returns to its origin by walking the given edge in the opposite direction.

$$
\begin{aligned}
&\mathit{checkPath}\colon \mathit{Graph} \to [\texttt{int} \times \texttt{int}] \to [2]\ =\\
&\quad \lambda \mathit{graph}.\ \lambda \mathit{inputedge}.\ \texttt{copy}\ \mathit{graph}\ \texttt{as}\ \mathit{graph}_1, \mathit{graph}_2\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \langle \mathit{size}, \mathit{edge}\rangle = \mathit{graph}_1\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ [\langle s, d\rangle] = \mathit{inputedge}\ \texttt{in}\\
&\qquad\qquad \mathit{if}\ (\mathit{edge}\ [\langle s, d\rangle])\\
&\qquad\qquad\quad (\texttt{loop}\ (\lambda w.\ \texttt{let}\ [\langle s', d'\rangle] = w\ \texttt{in}\\
&\qquad\qquad\qquad\qquad \mathit{if}\ [\texttt{eq}(d', s)]\\
&\qquad\qquad\qquad\qquad\quad [\texttt{return}(\mathit{eq}(\langle s', d\rangle))]\\
&\qquad\qquad\qquad\qquad\quad (\texttt{let}\ [d] = \mathit{nextEdge}\ \mathit{graph}_2\ [\langle d', s'\rangle]\ \texttt{in}\ [\texttt{continue}(d)]))\ [\langle s, d\rangle])\\
&\qquad\qquad\quad [\mathit{true}]
\end{aligned}
$$

Now, writing a function *checkCycle* for testing for the presence of non-trivial cycles in undirected graphs is a simple matter of verifying *checkPath* for all edges in the graph.

$$
\begin{aligned}
&\mathit{checkCycle}\ \colon \mathit{Graph} \to [2]\ =\\
&\quad \lambda \mathit{graph}.\ \texttt{let}\ \langle \mathit{size}, \mathit{edge}\rangle = \mathit{graph}\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ [n] = \mathit{size}\ \texttt{in}\\
&\qquad\qquad \texttt{copy}\ \mathit{edge}\ \texttt{as}\ \mathit{edge}_1, \mathit{edge}_2\ \texttt{in}\\
&\qquad\qquad \mathit{forall}\ [n]\ (\lambda s'.\ \texttt{let}\ [s] = s'\ \texttt{in}\\
&\qquad\qquad\quad \mathit{forall}\ [n]\ (\lambda d'.\ \texttt{let}\ [d] = d'\ \texttt{in}\\
&\qquad\qquad\qquad\quad \mathit{if}\ (\mathit{edge}_1\ [\langle s, d\rangle])\\
&\qquad\qquad\qquad\qquad (\mathit{checkPath}\ \langle [n], \mathit{edge}_2\rangle\ [\langle s, d\rangle])\\
&\qquad\qquad\qquad\qquad [\mathit{true}]))
\end{aligned}
$$

Finally, an acyclicity check can be defined by combining this test with a test for the absence of trivial cycles, i.e. self-loops.

$$
\begin{aligned}
&\mathit{acyclic}\ \colon \mathit{Graph} \to [2]\ =\\
&\quad \lambda \mathit{graph}.\ \texttt{copy}\ \mathit{graph}\ \texttt{as}\ \mathit{graph}_1, \mathit{graph}_2\ \texttt{in}\\
&\qquad\qquad \mathit{and}\ (\mathit{noSelfLoop}\ \mathit{graph}_1)\ (\mathit{checkCycle}\ \mathit{graph}_2)
\end{aligned}
$$

4.1.3. Undirected Forest Accessibility

The problem of undirected forest accessibility is to decide whether two given nodes u and v in a given undirected forest are connected by a path. Cook & McKenzie [9] show that this is a LOGSPACE-complete decision problem. To show

that the problem is in LOGSPACE, they reduce it to the problem of deciding undirected acyclicity. Two nodes u and v are connected if and only if they are connected directly by an edge, or if adding an undirected edge between them makes the graph cyclic.

This reduction is easy to implement in INTML.

$$
\begin{aligned}
&\mathit{undirectedForestAccessible} : \mathit{Graph} \to [\mathtt{int} \times \mathtt{int}] \to \mathit{Graph}\\
&\mathit{undirectedForestAccessible} = \lambda \mathit{graph}.\ \mathtt{copy}\ \mathit{graph}\ \mathtt{as}\ \mathit{graph}_1, \mathit{graph}_2\ \mathtt{in}\\
&\qquad\qquad\qquad\qquad \mathtt{let}\ \langle \mathit{size}, \mathit{edge}\rangle = \mathit{graph}_1\ \mathtt{in}\\
&\qquad\qquad\qquad\qquad \lambda e.\ \mathtt{copy}\ e\ \mathtt{as}\ e_1, e_2\ \mathtt{in}\\
&\qquad\qquad\qquad\qquad\quad \langle \mathit{size}, \mathit{or}\ (\mathit{edge}\ e_1)\ (\mathit{neg}\ (\mathit{acyclic}\ (\mathit{addEdge}\ \mathit{graph}_2\ e_2)))\rangle
\end{aligned}
$$

This is a simple example of how the use of higher-order functions in INTML allows one to manipulate large data, in this case graphs, as if they were a value in memory.

4.2. Encoding Algol-like Languages

The definition of INTML was guided by the question of how programming with sublinear space can be supported in functional programming languages. However, the direct definition mechanism in INTML is expressive enough to express imperative programming constructs as in Idealised Algol as well.

Reynolds has introduced *Idealised Algol* as a synthesis of functional and imperative programming [41]. Its core is the simply-typed λ-calculus for functional programming, which is extended with constants for impertative programming constructs. Assume the simply-typed λ-calculus with the following types

$$U, V \ ::= \ \mathtt{com} \ \mid \ \mathtt{exp}_\tau \ \mid \ \mathtt{cell}_\tau \ \mid \ U \times V \ \mid \ U \to V \ ,$$

where $\tau \in \{\mathit{bool}, \mathit{int}\}$. The idea is that terms of type $\mathtt{com}$ are imperative commands that can be exectued and that may have side effects, terms of type $\mathtt{exp}_\tau$ are expressions that when executed return a value of type τ, and that $\mathtt{cell}_\tau$ repesents memory cells with content of type τ that may be updated destructively. In order to be able to construct programs of these types, one assumes suitable term constants. In the literature one can find a number of variants of Idealised Algol with various choices of constants. A typical selection is:

$$
\begin{aligned}
\mathtt{true} &: \mathtt{exp}_{\mathit{bool}} & \qquad \mathtt{skip} &: \mathtt{com}\\
\mathtt{neg} &: \mathtt{exp}_{\mathit{bool}} \to \mathtt{exp}_{\mathit{bool}} & \mathtt{seq} &: \mathtt{com} \times \mathtt{com} \to \mathtt{com}\\
\mathtt{or} &: \mathtt{exp}_{\mathit{bool}} \times \mathtt{exp}_{\mathit{bool}} \to \mathtt{exp}_{\mathit{bool}} & \mathtt{if} &: \mathtt{exp}_{\mathit{bool}} \times \mathtt{com} \times \mathtt{com} \to \mathtt{com}\\
0, 1, 2, \ldots &: \mathtt{exp}_{\mathit{int}} & \mathtt{while} &: \mathtt{exp}_{\mathit{bool}} \times \mathtt{com} \to \mathtt{com}\\
\mathtt{leq} &: \mathtt{exp}_{\mathit{int}} \times \mathtt{exp}_{\mathit{int}} \to \mathtt{exp}_{\mathit{bool}} & \mathtt{newvar}_\tau &: (\mathtt{cell}_\tau \to \mathtt{com}) \to \mathtt{com}\\
\mathtt{add}, \mathtt{sub}, \mathtt{times} &: \mathtt{exp}_{\mathit{int}} \times \mathtt{exp}_{\mathit{int}} \to \mathtt{exp}_{\mathit{int}} & \mathtt{asg}_\tau &: (\mathtt{cell}_\tau \times \mathtt{exp}_\tau) \to \mathtt{com}\\
& & \mathtt{der}_\tau &: \mathtt{cell}_\tau \to \mathtt{exp}_\tau
\end{aligned}
$$

The constants for expressions are a selection of standard operations on expressions. For example, $\mathtt{leq}\ \langle e_1, e_2\rangle$ represents the boolean expression $e_1 \le e_2$. The constants for commands represent imperative programs: $\mathtt{skip}$ is the command that does nothing; $\mathtt{seq}\ \langle c_1, c_2\rangle$ represents the sequential composition of c_2 after c_1; $\mathtt{if}$ and $\mathtt{while}$ are there for conditional execution and while loops. Finally, the constants $\mathtt{newvar}$, $\mathtt{asg}$ and $\mathtt{der}$ are there for the allocation of a memory cell, for assignment and for dereferencing. The term $\mathtt{newvar}_\tau\ (\lambda x.\, c)$ can be understood as follows. It first allocates a new memory cell x that contains some default value of type τ, then it executes the command c, which may make use of the cell x, and then it deallocates x. In Java, this corresponds block-scoped variables, as in `{int x; c}`. Using the term $\mathtt{asg}\ \langle x, e\rangle$ one can write the value of an expression to the cell x. The term $\mathtt{der}\ x$ is the expression that reads the value from x.

One may think of these constants as constructors for the abstract syntax of a while-language. For example, the Java program on the left below computes the factorial of an `int`-variable `x` (which is assumed to be declared before). It is represented by the λ-term on the right, in which `x` is a free variable of type $\mathtt{cell}_{\mathit{int}}$. Note how the constant $\mathtt{newvar}_{\mathit{int}}$

takes a function as argument and how this is used to represent that the variable y is bound by the declaration. Definitions of Idealised Algol typically come without the definition of a concrete syntax.

```
int y;
y = 1;
while (1 <= x) {
  y = y * x;
  x = x - 1
};
x = y
```

$$\begin{aligned}
&\mathtt{newvar}_{int}\ (\lambda \mathtt{y}.\\
&\quad \mathtt{seq}\,\langle \mathtt{asg}_{int}\,\langle \mathtt{y}, 1\rangle,\\
&\qquad \mathtt{seq}\,\langle \mathtt{while}\,\langle \mathtt{lt}\,\langle 1, \mathtt{der}_{int}\ \mathtt{x}\rangle,\\
&\qquad\qquad \mathtt{seq}\ \langle \mathtt{asg}_{int}\,\langle \mathtt{y}, \mathtt{mul}\,\langle \mathtt{der}_{int}\ \mathtt{y}, \mathtt{der}_{int}\ \mathtt{x}\rangle,\\
&\qquad\qquad\quad \mathtt{asg}_{int}\,\langle \mathtt{x}, \mathtt{sub}\,\langle \mathtt{der}_{int}\ \mathtt{x}, 1\rangle\rangle\rangle\rangle\\
&\qquad\quad \mathtt{asg}\,\langle \mathtt{x}, \mathtt{sub}\,\langle \mathtt{der}_{int}\ \mathtt{y}\rangle\rangle\rangle\rangle) : \ \mathtt{com}
\end{aligned}$$

In this section we outline how constants as in Idealised Algol can be defined in INTML. The aim is to implement the constants of Idealised Algol so that the abstract syntax representation of imperative programs has the intended behaviour. Since INTML does not allow unrestricted copying of variables in the interactive language, we will not be able to embed all of Idealised Algol, of course. However, there are interesting applications of Idealised Algol even without full copying. Ghica [14] uses a restricted variant of Idealised Algol, *Basic Syntactic Control of Interference (bSCI)*, as a language for hardware synthesis. Ghica translates bSCI-programs to hardware circuits in a way that is very similar to our translation of interactive terms to base language terms. Indeed, if one removes type variables from the base language, base language 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 expressiveness 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 [14], but we will show in Section 4.3 how to express them in INTML.

We now discuss how one may define an Algol-like language if in Reynolds' definition [41] one replaces the simply-typed λ-calculus with the interactive language of INTML. We define Algol-like program constants, the choice of which is guided by those in bSCI [14].

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

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

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. Expressions are modelled analogously, only that the returned value now represents a value of the expression type. 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 $[\alpha] \multimap \mathtt{com}$, which can be used to overwrite the cell with a new value.

Constants can be defined as follows. We define curried constants, as this allows us to give them more precise subexponential types.

- Logical operations op on booleans are represented by terms of type

  $$\mathtt{op}\colon \mathtt{exp}_2 \multimap 2 \cdot \mathtt{exp}_2 \multimap \mathtt{exp}_2\ .$$

  For instance, or can be defined as

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

  Other operations on integer expressions can be represented similarly.

- 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

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

- Branching is available by means of a term

$$\texttt{if}\colon \texttt{exp}_2 \multimap 2 \cdot \texttt{com} \multimap 2 \cdot \texttt{com} \multimap \texttt{com}\ .$$

Its definition is

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

- For iteration we use a term

$$\texttt{while}\colon 2 \cdot \texttt{exp}_2 \multimap 2 \cdot \texttt{com} \multimap \texttt{com}$$

defined by

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

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

$$\texttt{der}\colon \texttt{cell}_\alpha \multimap \texttt{exp}_\alpha\ , \qquad \texttt{asg}\colon \texttt{cell}_\alpha \multimap \texttt{exp}_\alpha \multimap \texttt{com}$$

for dereferencing a boolean cell and assigning to it can be defined simply by $\texttt{der} := \lambda m.\, \texttt{let}\ \langle x, y\rangle = m\ \texttt{in}\ y$ and $\texttt{asg} := \lambda m.\, \lambda e.\, \texttt{let}\ \langle x, y\rangle = m\ \texttt{in}\ x\ e$. More interesting is a term $\texttt{newvar}_A$, 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:

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

Notice that $\texttt{newvar}_A$ has a subexponential annotation A on its argument. The content of the memory cell will be encoded in the space given by this annotation. We implement $\texttt{newvar}_A$ using rule Hack. Even though we only need the cases where $A = 2$ of $A = \texttt{int}$, in order to make clear which assumptions on A are made, we give the implementation for an arbitrary type A with a given value $v_A\colon A$ that serves as the initial value of the memory cell. We define $\texttt{newvar}_A$ as $\texttt{hack}(x.n)$, where $x\colon X^- \vdash n : X^+$:

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

We implement n to satisfy the following reductions:

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

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

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

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

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

These can be understood as follows: Reduction (11) means that when the computation of $\texttt{newvar}_A\ c$ 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. Once c replies with a value u this value is placed in the memory cell by (15), overwriting the previous value, and termination of the update function is indicated to c.

- We note that it is also possible to define combinators $\mathtt{seq}'$ and $\mathtt{newvar}'$ that return an expression instead of a command.

$$\mathtt{seq}' : \mathtt{com} \multimap \mathtt{exp}_\tau \multimap \mathtt{exp}_\tau$$
$$\mathtt{newvar}'_A : A \cdot (\beta \cdot \mathtt{cell}_A \multimap \mathtt{exp}_\tau) \multimap \mathtt{exp}_\tau .$$

  One may be define them directly by modifying the above definition appropriately. They can also be defined from `seq` and `newvar` by using `callcc` to return an expression.

4.3. Kierstead Terms

To convince the reader that INTML allows proper higher-order functions, we consider as an example the Kierstead terms, which are a standard example from game semantics to illustrate the difficulties of higher-order functions (namely to explain the need for justification pointers [24]). The following two λ-terms are quite similar in structure, but not equivalent:

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

The terms can be turned into INTML terms typeable as

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

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

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

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

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

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

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

The base language term $x_1 : 2,\ x_2 : 2 \ \vdash\ \mathtt{nor}(x_1, x_2) : 2$ 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 $t_1\ s \mapsto_k^* [\mathtt{true}]$ and $t_2\ s \mapsto_k^* [\mathtt{false}]$.

5. 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. The construction does not depend on the bit-width of integers. In the following section we fix a bit-width k and consider all reduction with respect to this bit-width. We note that the translation from interactive terms to base terms does not depend on k, i.e. for any choice of k it produces the same base term.

5.1. Circuits

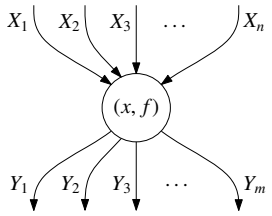
We start by defining the message passing circuits that serve as an intermediate language for the compilation. These circuits implement bidirectional message passing, as explained in the Introduction. Circuits may also be understood as a particular instance of string diagrams for monoidal categories [34]. They are also related to proof-nets, see [34] 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 $Y_1, Y_2, \ldots, Y_m$,

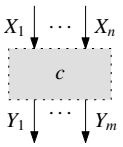


then the following base language typing judgement must be derivable:

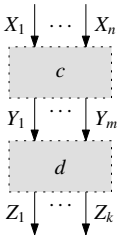
$$\Gamma,\ x\colon Y_1^- + \cdots + Y_m^- + X_1^+ + \cdots + X_n^+ \ \vdash\ f\ :\ Y_1^+ + \cdots + Y_m^+ + X_1^- + \cdots + X_n^-$$

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

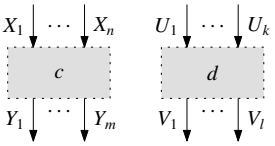
For each interactive type X, we have defined a pair of base language types X^- and X^+ in Section 2.2. We will use an interactive type X to stand for the edge label (X^-, X^+). It is useful to allow circuits to have a number of input and output ports. We formalise these by means of two distinguished 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.



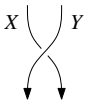
Given two circuits $c\colon [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ and $d\colon [Y_1, \ldots, Y_m] \to [Z_1, \ldots, Z_k]$, their sequential composition $d \circ c\colon [X_1, \ldots, X_n] \to [Z_1, \ldots, Z_k]$ is defined as depicted below: the i-th incoming edge to the sink of c and the i-th outgoing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

5.2. 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 with respect to bit-width k (recall the we have fixed k at the beginning of this section). 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 [33].

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

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

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

where we write $\mathtt{in}_k$ 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:

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

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 $c\colon [X_1,\dots,X_n] \to [Y_1,\dots,Y_m]$, 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 $[X_1,\dots,X_n] \to [Y_1,\dots,Y_m]$ 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.

5.3. Implementing Message Passing

The message passing behaviour of any circuit can be implemented by a base language term. For any circuit $c\colon [X_1,\dots,X_n] \to [Y_1,\dots,Y_m]$ 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 $\hat{c}[\sigma][v/x] \mapsto_k^* w$ 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 $x_1\colon A_1,\dots,x_n\colon A_n$. 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 $\hat{\chi}_c$.

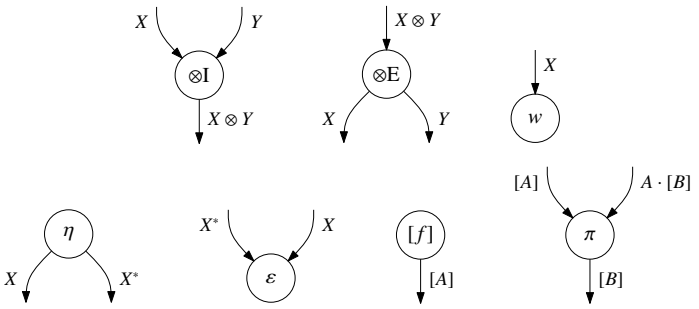
5.4. Constructing Circuits

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 3. *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 4. *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.

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

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

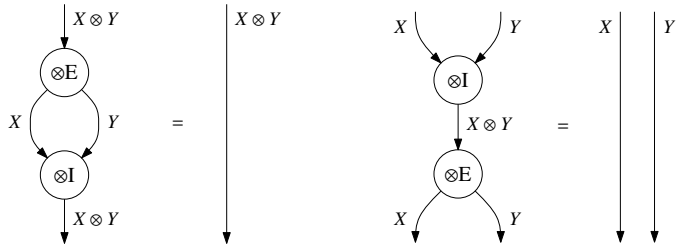
$$\chi_{[f]}(o_1, (\sigma, *, -)) = (o_1, (\sigma, v, +)) \text{ if } f[\sigma] \mapsto_k^* v \text{ and } v \text{ is a value}$$

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

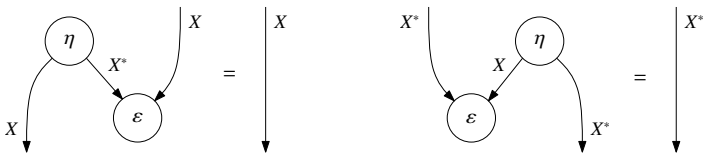
Informally, the nodes ⊗I and ⊗E serve for packing two message passing wires into one. Node w discards all messages; it is used for weakening. Nodes η and ε serve to exchange the direction of messages; we use them to implement functions. Node π is useful for sequencing thunks. If a question arrives at 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.

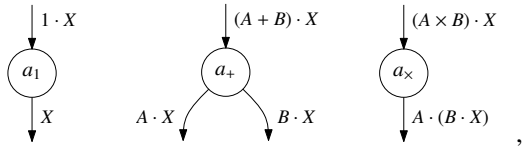
Lemma 5.



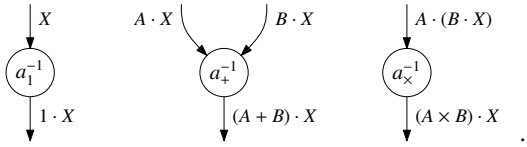
Lemma 6.



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

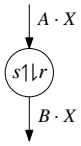


The behaviour of these nodes is given by the canonical terms of the respective type. We spell out the behaviour of a_+ and $a_\times$:

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

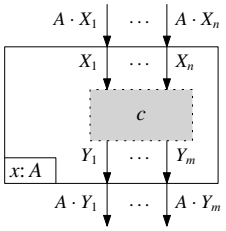
For the reader familiar with Linear Logic, nodes a_1, a_+ and $a_\times$ play a role similar to dereliction, contraction and digging nodes (respectively) in proof-nets for the exponentials.

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 node 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 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 $(x : A) \cdot c\colon [A \cdot X_1, \ldots, A \cdot X_n] \to [A \cdot Y_1, \ldots, A \cdot Y_m]$ 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 $(x : A) \cdot c$ can be specified as follows. Let $e_1, \ldots, e_{m+n}$ be the list of input and output edges of $(x : A) \cdot c$, 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 $(x : A) \cdot c$ is then specified by: $\hat{\chi}_{((x:A)\cdot c)}(e_i, (\sigma, (v, u), p)) = (e_j, (\rho, (v, w), r))$ 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 $(x : A) \cdot c$ 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 $(x : A) \cdot c = (y : A) \cdot c[y/x]$ for any fresh variable y. Here, $c[y/x]$ denotes substitution of y for x in each node in c. We write $A \cdot c$ instead of $(x : A) \cdot c$ if x does not appear free in any of the nodes in c.

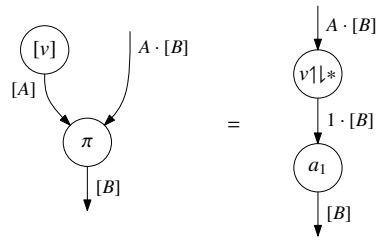
Boxes are easily seen to have the following properties:

Lemma 7. *The equations $A \cdot id = id$, $(x : A) \cdot (c \circ d) = ((x : A) \cdot c) \circ ((x : A) \cdot d)$ and $(x : A) \cdot (c \otimes d) = ((x : A) \cdot c) \otimes ((x : A) \cdot d)$ hold for all circuits c and d, all variables x and all types A for which the circuits in the equations are well-defined.*

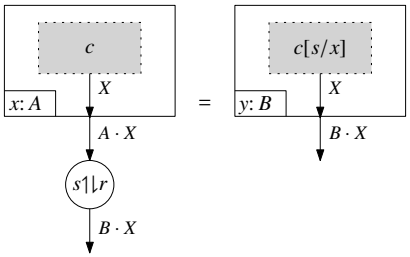
Having introduced all the building blocks for the construction of circuits, we state a few results that allow us to prove soundness of the translation from the interactive to the base language.

Lemma 8. *For any circuit c with one outgoing edge and no incoming edges, we have $a_1 \circ (1 \cdot c) = c$ and $a_+ \circ ((A + B) \cdot c) = (A \cdot c) \otimes (B \cdot c)$ and $a_\times \circ ((A \times B) \cdot c) = A \cdot (B \cdot c)$.*

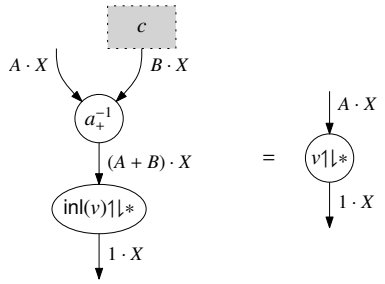
Lemma 9. *For any value $\vdash v : A$, the following equality of circuits holds.*



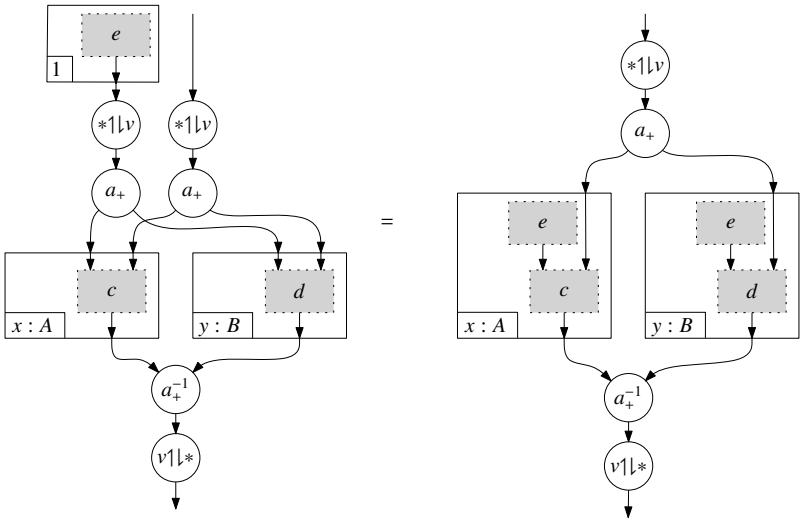
Lemma 10. *Let $\Gamma, y\colon B \vdash s : A$ and $\Gamma, x\colon A \vdash r : B$ be a section-retraction pair. For any circuit c over $\Gamma, x\colon A$ with no inputs and a single output labelled X we have:*



Lemma 11. *For any circuit $c\colon [\,] \to [B \cdot X]$ and any value $\vdash v : A$, the following equality of circuits holds.*



Lemma 12. *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 $(*\uparrow\downarrow v)$ and $(v\uparrow\downarrow *)$, such a message must have the form $(\sigma, (w_1, w_2), -)$ where $\mathtt{inl}(w_1) = v[\sigma]$. 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. □

5.5. 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 $[\![\delta]\!]\colon [A_1 \cdot X_1, \ldots, A_n \cdot X_n] \to [Y]$ 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. 10. 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 $\Phi = x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n$, we write Φ also for the list $[A_1 \cdot X_1, \ldots, A_n \cdot X_n]$ 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. 11–15. We give full details, including all edge labels, in the case of ⊗E in Fig. 11, 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. 16 the circuit that is obtained by interpreting the derivation of the typing judgement $\vdash \lambda f.\, \lambda y.\, \mathtt{let}\ [x] = y\ \mathtt{in}\ f\ [x]\ [x] : \alpha \cdot ([\alpha] \multimap [\alpha] \multimap [\beta]) \multimap [\alpha] \multimap [\beta]$ from Section 2.3.

Ax	$[\![\delta]\!] = a_1$
Struct	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Phi \otimes (s \mid r))$, 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	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Psi \otimes (a_+ \circ ((A + B) \cdot [\![\delta_s]\!]) \circ (a_\times)_\Phi))$
[ ]I	$[\![\delta]\!] = [f]$
[ ]E	$[\![\delta]\!] = \pi \circ ([\![\delta_s]\!] \otimes ((x : A \cdot [\![\delta_t]\!]) \circ (a_\times)_\Psi))$
⊗I	$[\![\delta]\!] = \otimes I \circ ([\![\delta_s]\!] \otimes [\![\delta_t]\!])$
⊗E	$[\![\delta]\!] = [\![\delta_t]\!] \circ (id_\Psi \otimes ((A \cdot [\![\delta_s]\!]) \circ (a_\times)_\Phi))$
—∘I	$[\![\delta]\!] = \otimes I \circ swap \circ ([\![\delta_t]\!] \otimes id_X) \circ (id_\Psi \otimes \eta)$
—∘E	$[\![\delta]\!] = (id_Y \otimes \varepsilon) \circ ((swap \circ \otimes E \circ [\![\delta_t]\!]) \otimes ((A \cdot [\![\delta_s]\!]) \circ (a_\times)_\Phi))$
+E^c	$[\![\delta]\!] = a_1 \circ (v \mid *) \circ a_+^{-1} \circ (((x : A) \cdot [\![\delta_s]\!]) \otimes ((y : B) \cdot [\![\delta_t]\!])) \circ (a_+)_\Phi \circ (* \mid v)_\Phi \circ (a_1^{-1})_\Phi$
Hack	$[\![\delta]\!]$ is the single node with label (x, f).

Figure 10: Interpretation of Interactive Derivations

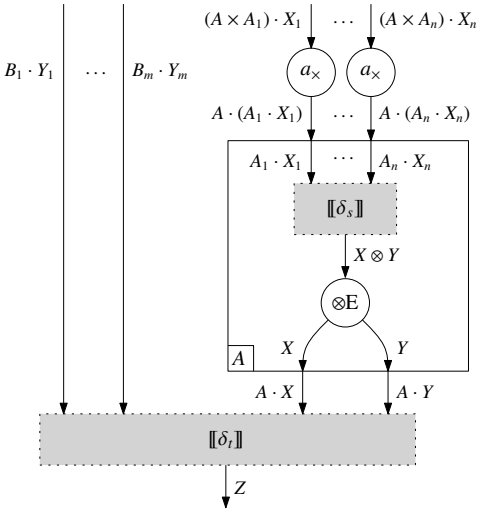


Figure 11: Circuit for ⊗E

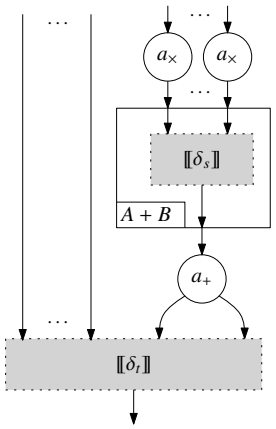


Figure 12: Circuit for Copy

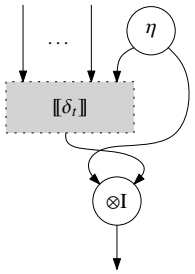


Figure 13: Circuit for ⊸I

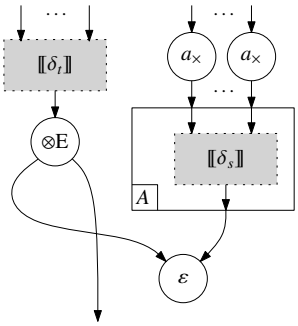


Figure 14: Circuit for ⊸E

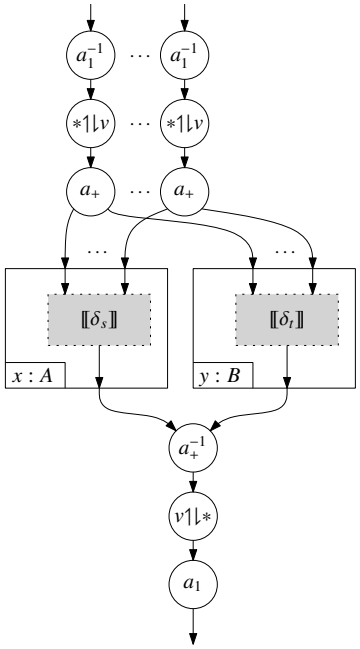


Figure 15: Circuit for $+E^c$

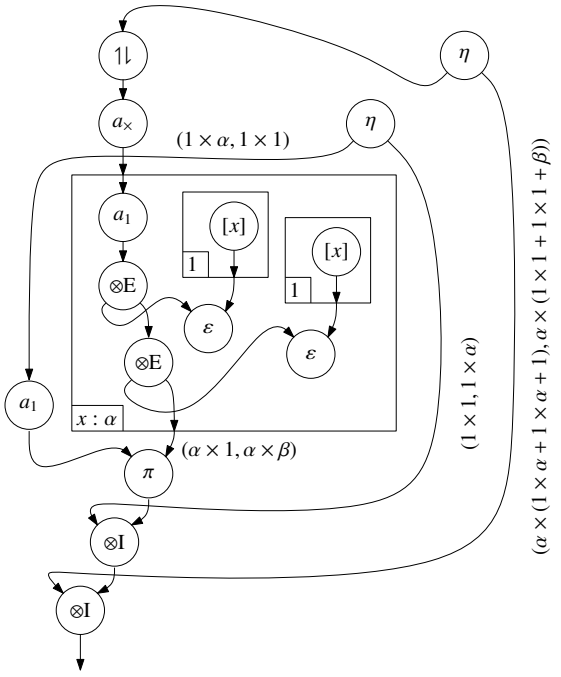


Figure 16: Example Circuit

5.6. 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 13 (Soundness)**.** *If δ derives $\cdot \mid \cdot \vdash s : X$ and $s \mapsto_k t$, then there exists a derivation ρ of $\cdot \mid \cdot \vdash t : X$ with $[\![\delta]\!] = [\![\rho]\!]$.*

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

Lemma 14 (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 $[\![\rho[\delta/x]]\!] = [\![\rho]\!] \circ (id_\Phi \otimes (A \cdot [\![\delta]\!]) \otimes id_\Psi)$.*

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 $[\![\rho[\delta/x]]\!] = [\![\delta]\!] = a_1 \circ (1 \cdot [\![\delta]\!]) = [\![\rho]\!] \circ (1 \cdot [\![\delta]\!])$, 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:

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

  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:

$$\begin{aligned}
[\![\rho[\delta/x]]\!] = [\![\rho_1[\delta/x]]\!] &= [\![\rho_1]\!] \circ (id_\Phi \otimes (B \cdot [\![\delta]\!])) && \text{by defn. and i.h.} \\
&= [\![\rho_1]\!] \circ (id_\Phi \otimes ((s \restriction r) \circ (A \cdot [\![\delta]\!]))) && \text{by Lemma 10} \\
&= [\![\rho_1]\!] \circ (id_\Phi \otimes (s \restriction r)) \circ (id_\Phi \otimes (A \cdot [\![\delta]\!])) \\
&= [\![\rho]\!] \circ (id_\Phi \otimes (A \cdot [\![\delta]\!]))
\end{aligned}$$

  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

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

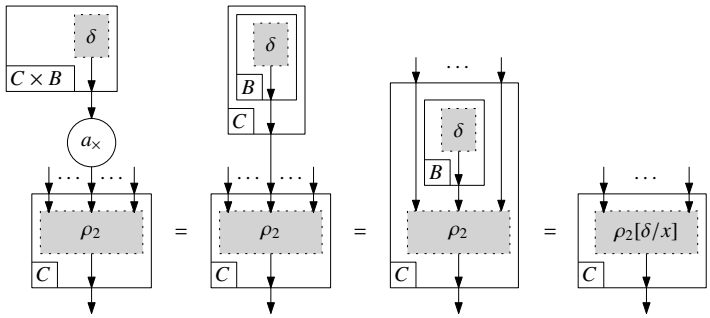
  as was required to show.

- 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 8 and 7 and the induction hypothesis. But now the leftmost circuit in the above equation appears in $[\![\rho]\!] \circ (id_\Phi \otimes (A \cdot [\![\delta]\!]) \otimes id_\Psi)$. 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 $+\mathrm{E}^{\mathsf{c}}$

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

then we use Lemma 12 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 12 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 $(r\restriction s) \circ (A \cdot (\otimes I)) = (B \cdot (\otimes I)) \circ ((r\restriction s) \otimes (r\restriction s))$ and $(B \cdot (\otimes E)) \circ (r\restriction s) = ((r\restriction s) \otimes (r\restriction s)) \circ (A \cdot (\otimes E))$.). Then, by induction hypothesis, the box contents are $\rho_1[\delta/x]$ and $\rho_2[\delta/x]$ respectively, which completes this case.

□

Lemma 15. *Suppose ρ derives $\Gamma,\ x\colon A \mid \Phi \vdash t : Y$ 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. □

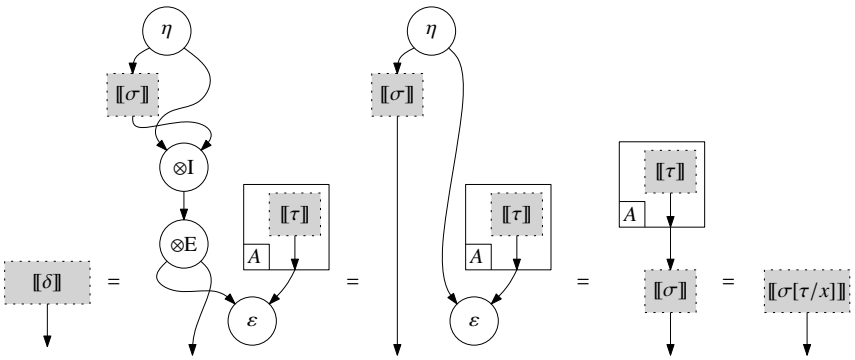
With these substitution lemmas, we can now prove the Soundness Theorem:

Proof of Theorem 13. For any reduction $s \mapsto_k t$ there exist decompositions $s = E[s']$ and $t = E[t']$ such that $s' \to_k t'$ is an instance of one of the basic reductions from Fig. 9.

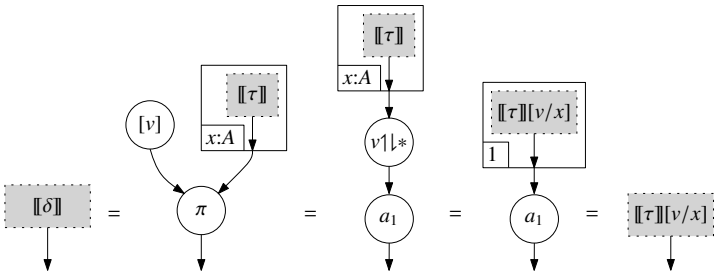
The proof then goes by induction on the structure of E. The base case, where E is empty, amounts to showing the assertion for the basic reductions. We spell out representative cases:

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

  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 $\cdot \mid \cdot \vdash p\colon Z$ of these rules. Then we use Lemmas 5 and 6 and the substitution lemma to calculate:

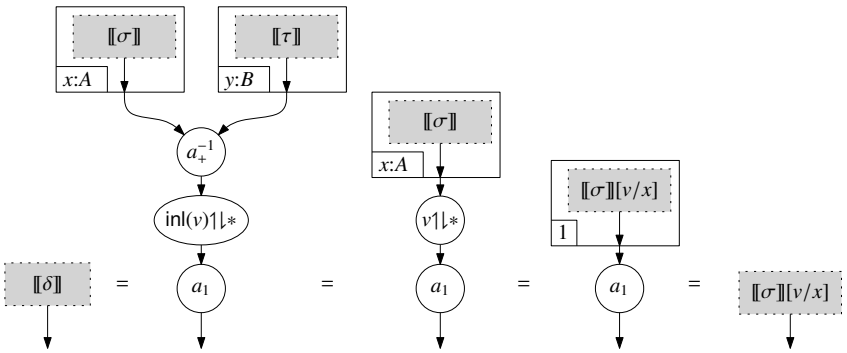


- $s \to_k t$ is $(\texttt{let}\ [x] = [v]\ \texttt{in}\ p) \to_k p[v/x]$, where v is a closed value. Using Lemmas 9, 10 and 8 we calculate:



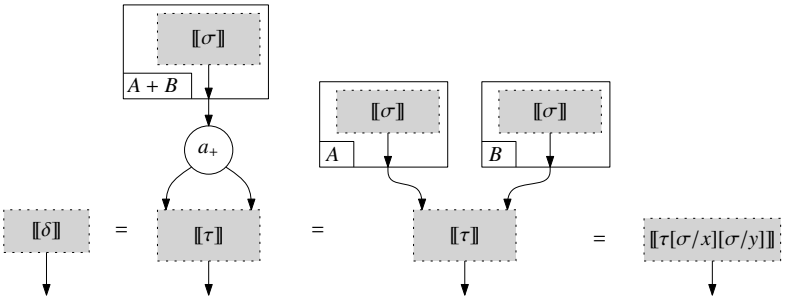
  Letting ρ be $\tau[v/x]$ and using Lemma 15 completes this case.

- $s \to_k t$ is $(\texttt{case}\ \texttt{inl}(v)\ \texttt{of}\ \texttt{inl}(x) \Rightarrow p \mid \texttt{inr}(y) \Rightarrow q) \to_k p[v/x]$, where v is a closed value. Using Lemmas 11, 8 and 10 we calculate:



  Again using Lemma 15, we can complete this case by letting ρ be $\sigma[v/x]$.

- $s \to_k t$ is $(\texttt{copy}\ p\ \texttt{as}\ x, y\ \texttt{in}\ q) \to_k q[p/x][p/y]$. In this case we use Lemma 8 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 16. *For any interactive term t and any base language term f there exists a base language term* ($\mathtt{let}\ [x] = t\ \mathtt{in}\ f$) *such that the typing rule*

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

is admissible and there are reductions

$$\mathtt{let}\ [x] = [v]\ \mathtt{in}\ f \ \mapsto_k^* \ f[v/x] \ \text{ if } v \text{ is a value}\ ,$$
$$\mathtt{let}\ [x] = s\ \mathtt{in}\ f \ \mapsto_k^* \ \mathtt{let}\ [x] = t\ \mathtt{in}\ f \ \text{ if } s \mapsto_k t\ .$$

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 ($\mathtt{let}\ [x] = t\ \mathtt{in}\ f$) to be ($\mathtt{let}\ y = \hat{t}[*/x]\ \mathtt{in}\ f$). □

5.7. Relation to the Int-Construction

At the beginning of this article we have a construction of message passing circuits from the base language. We have derived this construction 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 *within* 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 [13] and that of Call-by-Push-Value [31], see [43].

5.7.1. Complex Values

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

$$\begin{aligned} f, g \ ::=\ & x \mid * \mid \langle f, g\rangle \mid \mathtt{fst}(f) \mid \mathtt{snd}(f) \mid \mathtt{impossible}(f) \mid \mathtt{inl}(f) \mid \mathtt{inr}(g) \\ & \mid \mathtt{case}\ f\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g \mid \mathtt{inr}(y) \Rightarrow h \end{aligned}$$

These terms have the same meaning and typing rules as in the base language. The term $\mathtt{impossible}(f)$ is to be typed with the following elimination rule for type 0.

$$\frac{\Gamma \vdash f : 0}{\Gamma \vdash \mathtt{impossible}(f) : A}$$

In the base language, $\mathtt{impossible}(f)$ may be seen as an abbreviation for the looping term $\mathtt{let}\ x = *\ \mathtt{loop}\ \mathtt{inr}(*)$, which is why we have not included it in the definition of the base language.

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 17. *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.

5.7.2. 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. [29]. 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 $\mathbb{C}(A, B)_\Gamma$.

5.7.3. 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 is 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 18. *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 $\psi_\Gamma(E[\cdot], F[\cdot]) = \mathtt{case}\ [\cdot]\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow E[x] \mid \mathtt{inr}(y) \Rightarrow F[y]$. An inverse is $\psi_\Gamma^{-1}(E[\cdot]) = \langle E[\mathtt{inl}([\cdot])], E[\mathtt{inr}([\cdot])]\rangle$. □

The Int construction can be applied to any traced monoidal category [28]. 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 19. *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 $(\mathtt{let}\ y = [\cdot]\ \mathtt{loop}\ E[y])$. 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:

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

We have to show extensional equality of E_1 and E_2, that is for all closed values v and w, we have $E_1[v] \mapsto_k^* w$ if and only if $E_2[v] \mapsto_k^* w$. We show by well-founded induction on i that $E_1[v] \mapsto_k^i w$ implies $E_2[v] \mapsto_k^* w$ for all v and w. The reduction $E_1[v] \mapsto_k^* w$ takes one of the following two forms:

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

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

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

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

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

  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

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

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

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

    In this case we also have

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

    as was required to show.

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

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

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

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

□

Proposition 20. *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

$$F = \texttt{case}\ E[\texttt{inl}(x)]\ \texttt{of}\ \texttt{inl}(x) \Rightarrow x \mid \texttt{inr}(y) \Rightarrow \texttt{let}\ z = y\ \texttt{loop}\ E[\texttt{inr}(z)]$$

with $\Gamma,\ x\colon A \vdash F[x] : B$. This defines a uniform trace in each $\mathbb{C}_\Gamma$, see [20]. Naturality, i.e. closure under substitution with complex values, follows immediately. □

5.7.4. 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 [28]. 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 over Γ with interface $[X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ is a representation of a map of type $X_1 \otimes \cdots \otimes X_n \to Y_1 \otimes \cdots \otimes Y_m$ in $\mathrm{Int}(\mathbb{C})_\Gamma$. The compact closed structure of $\mathrm{Int}(\mathbb{C})$ is given in Lemmas 5 and 6. For further information on the categorical properties of the Int construction we refer to [28, 21], as the intention in this section is merely to relate the construction of circuits to the standard Int construction.

6. Space Bounds

The translation of interactive terms to circuits is a compilation method that allows us to establish interesting space bounds in a simple way. 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 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. Notice that this simple space bound can already be computed given only an INTML typing derivation.

If we restrict our attention to types not containing `int` or type variables, all these types will be finite, and we obtain finite size bounds. We immediately obtain:

Proposition 21 (Constant Space)**.** *If δ derives $\Gamma \mid \Phi \vdash t : X$ and δ does not contain any type variable of the type* `int`*, 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 [14]. As INTML is more expressive than Ghica's bSCI, it may be 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 (FLOGSPACE) and non-deterministic logarithmic space (NFLOGSPACE).

6.1. Logarithmic Space

We describe a precise correspondence between the class of functions representable in INTML and the class of functions computable in logarithmic space. We begin by analysing the space requirements of base language reduction.

We consider the space usage of a simple direct implementation of term reduction for the base language. The size needed to encode a term will be proportional to the size of its abstract syntax trees, provided that the size of `int`-constants is counted appropriately. We define the size of values to be $|x| = |*| = 1$, $|n| = \lceil \log n \rceil$, $|\langle v, w\rangle| = |v| + |w|$ and $|\mathtt{inl}(v)| = |\mathtt{inr}(v)| = 1 + |v|$. With this definition values can be encoded such that v needs space $O(|v|)$. This definition is extended to base terms in the evident way by counting any inner node in the abstract syntax tree as one, e.g. $|\mathtt{case}\ f\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g \mid \mathtt{inr}(y) \Rightarrow h| = 1 + |f| + |g| + |h|$, etc.

For the purposes of relating INTML to FLOGSPACE, it suffices to prove the following lemma on the space requirements of base language reduction. It states that the space needed for base language reduction is linear in bit-width of integers.

Lemma 22. *Let $x\colon A \vdash f : B$ be derivable for variable-free types A and B. There exist are constants $c \in \mathbb{N}$ and $d \in \mathbb{N}$ such that for any k and any closed value $v\colon A$ containing only k-bit integers, we have that $f[v/x] \mapsto_k^* g$ implies $|g| \leq c \cdot k + d$.*

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:

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

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

For any variable-free base type A, we can define an upper bound $|A|$ on the size of its values as follows: $|0| = |1| = 1$ and $|\mathtt{int}| = k$, $|A \times B| = |A + B| = 1 + |A| + |B|$. We have $|v| \leq |A|$ for any closed value v of type A that contains only k-bit integers.

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

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

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

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

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

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

$$pot(\Gamma \vdash \mathtt{inl}(f) : A + B) \leq 1 + pot(\Gamma \vdash f : A) \tag{21}$$

$$pot(\Gamma \vdash \mathtt{inr}(g) : A + B) \leq 1 + pot(\Gamma \vdash g : B) \tag{22}$$

$$\begin{aligned} pot(\Gamma \vdash \mathtt{case}\ f\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g \mid \mathtt{inr}(y) \Rightarrow h : C) \leq\ & 1 + pot(\Gamma \vdash f : A + B) \\ & + pot(\Gamma, x\colon A \vdash g : C) \\ & + pot(\Gamma, y\colon B \vdash h : C) \end{aligned} \tag{23}$$

$$\begin{aligned} pot(\Gamma \vdash \mathtt{let}\ x = f\ \mathtt{loop}\ g : A) \leq\ & 3 + pot(\Gamma \vdash f : B) \\ & + 2pot(\Gamma, x\colon B \vdash g : A + B) \end{aligned} \tag{24}$$

$$\tag{25}$$

We consider (23) as a representative case: Any reduction of $\mathtt{case}\ f[\sigma]\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g[\sigma] \mid \mathtt{inr}(y) \Rightarrow h[\sigma]$ must, by definition, start with reducing $f[\sigma]$. In this stage all intermediate terms have the form $\mathtt{case}\ f'\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g[\sigma] \mid \mathtt{inr}(y) \Rightarrow h[\sigma]$ for some f' with $f[\sigma] \mapsto_k^* f'$. 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 $\mathtt{inl}(v)$ or $\mathtt{inr}(w)$, 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.

To establish the bound claimed in the lemma, consider now a derivable typing judgement $\Gamma \vdash f : A$. 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 $c_1, \ldots, c_n, d$. But by definition of the size of types, this means that there exists some c, such that

$$pot(\Gamma \vdash f : A) \leq c \cdot k + d$$

holds for all k. By definition of the potential, this is 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 FLOGSPACE-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.

For the representation of words we will assume that the bit width of integers is chosen to be $\lceil \log n \rceil$, where n is the length of the input word. This means that the type `int` contains just enough values to point to any position in the input word. A little care needs to be taken with the trivial cases where $n \leq 1$, so we will choose the bit-width to be $k(n)$, defined as follows.

$$k(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ \lceil \log n \rceil & \text{otherwise.} \end{cases}$$

Since logarithmic space functions can produce output of polynomial length, we need to be able to represent numbers of up to n^m for any constant m. We do so in the standard way by using the type $\texttt{int}^m := \texttt{int} \times \cdots \times \texttt{int}$ (m factors). If n is the maximum value of type `int`, then $\texttt{int}^m$ can represent values up to n^m. We define the least value of $\texttt{int}^m$ to be the vector of all zeros and order the values lexicographically. The basic arithmetic operations on $\texttt{int}^m$ can be implemented in the base language. For any number i that can be encoded in $\texttt{int}^m$, we write $\langle i \rangle_m$ for its representation as a value of type $\texttt{int}^m$.

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 $\Sigma \uplus \{\square\}$.

Formally, we represent Σ-words using types of the form $\mathbb{B}^m := 1 \cdot [\texttt{int}^m] \multimap [\Sigma_\square]$:

Definition 8. A closed interactive term t of type $\mathbb{B}^m$ *represents a word* $w_0 w_1 \ldots w_n \in \Sigma^*$ *with respect to k-bit integers* if and only if n can be can be represented as a value of type $\texttt{int}^m$ (using k-bit integers) and we have

$$t\ \langle i \rangle_m \ \mapsto_k^*\ [c] \quad \iff \quad (i \leq n \wedge c = w_i) \vee (i > n \wedge c = \square)$$

for all numbers i representable in $\texttt{int}^m$ (using k-bit integers).

We have defined $\mathbb{B}^m$ to be $1 \cdot [\texttt{int}^m] \multimap [\Sigma_\square]$. The reader may wonder if the choice to 1 in the subexponential is a restriction. It is certainly not a restriction for a program that is given a variable of such a type as an input. The program can assume that the program does not need to store anything when it requests its argument. It is also not a restriction for programs defining values of this type, as the A can be pulled out. If t has type $A \cdot [B] \multimap X$, then $\lambda x.\, \texttt{let}\ [b] = x\ \texttt{in}\ t\ [b]$ has type $1 \cdot [B] \multimap X$. The value b of type B is computed up front and stored in the environment, so that when t requests its argument the value can be returned right away without having to query x again.

Lemma 23. *For any $w \in \Sigma^*$ there exists a closed interactive term $\langle w \rangle_m : \mathbb{B}^m$ that represents w with respect to $k(|w|)$-bit integers.*

Proof. We can define a suitable term by $\langle w \rangle_m = \lambda x.\, \texttt{let}\ [i] = x\ \texttt{in}\ [w(i)]$, where $w(i)$ is a base language term that implements the pseudo-code $\texttt{if}\ i = \langle 0 \rangle_m\ \texttt{then}\ w_0\ \texttt{else if}\ i = \langle 1 \rangle_m\ \texttt{then}\ w_1\ \ldots$. □

Definition 9. We say that a term $\vdash t : A \cdot \mathbb{B}^m \multimap \mathbb{B}^r$ *represents* a function $\varphi : \Sigma^* \to \Sigma^*$ if and only if for every word $w \in \Sigma^*$ the term $(t\ \langle w \rangle_m)$ encodes $\varphi(w)$ with respect to $k(|w|)$-bit integers.

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 $t\,\langle w\rangle_m\,\langle i\rangle_r$ 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 24 (FLOGSPACE Soundness)**.** *If $\varphi\colon \Sigma^* \to \Sigma^*$ is represented by t, then φ is computable in logarithmic space. Moreover, a* FLOGSPACE *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 $A \cdot \mathbb{B}^m \multimap \mathbb{B}^r$. By Lemma 22, reduction of f with respect to k-bit integers (i.e. computation of φ on strings of length up to $(2^k)^m$) can be implemented using space linear in k.

Using this observation, we now construct a logarithmic space Turing Machine M_φ for φ. Given input w, we choose the bit-width of integers to be $k(|w|)$. With this choice the machine simulates the circuit for the term $(t\,\langle w\rangle_m\,\langle i\rangle_r)\colon [\Sigma_\square]$ with respect to k-bit integers, 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 with all integers represented using $k := k(|w|)$ bits. It then reduces $f\ \mathtt{inr}(\mathtt{inr}(*))$ with respect to k-bit integers. This amounts to asking the function for a character. Depending on the result value, it continues as follows: (i) if the value is $\mathtt{inr}(\mathtt{inr}(c))$ (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 $\mathtt{inr}(\mathtt{inl}(v, *))$ (informally: 'Which character of $\varphi(w)$ should be computed?') it reduces $f\ \mathtt{inr}(\mathtt{inl}(v, \langle i\rangle_r))$, where i is the position of the output head, and continues with the same case distinction; (iii) if the value is $\mathtt{inl}(v, \mathtt{inr}(*))$ (informally: 'A character from the input is requested.'), then it reduces $f\ \mathtt{inl}(v, \mathtt{inl}(*, *))$ (informally: 'What is the position of the requested character?') and continues with the same case distinction; (iv) if the value is $\mathtt{inl}(v, \mathtt{inl}(*, \langle i\rangle_r))$ (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 $f\ \mathtt{inl}(v, \mathtt{inr}(w_i))$ (informally: 'The requested character is w_i.') and continues with the same case distinction. □

Theorem 25 (FLOGSPACE 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 logarithmic space Turing Machines in INTML.

Let M be a logarithmic space Turing Machine computing φ. We can assume that it has one work tape. The content of the work tape and the position of the work tape head can be encoded using two stacks that contain the characters that one would see when reading from the work tape head to the left and to the right respectively. With a binary encoding, these stacks can be encoded by two natural numbers. As the machine only uses logarithmic space, these numbers can be bounded by a polynomial in the size of the input, as can the output length of the machine.

We can assume that the bit width of integers is large enough, so that $\mathtt{int}$ has enough elements to encode the positions in an input word. The state of M can be encoded by values of a base language type of the form

$$S \;=\; Q \times \mathtt{int} \times \mathtt{int}^m \times \mathtt{int}^m \times \mathtt{int}^r,$$

where Q is the set of states of M. The component of type $\mathtt{int}$ encodes the position of the input head. The two components of type $\mathtt{int}^m$ encode the content of the work tape of M to the left and right of M's work head Finally, the component of type $\mathtt{int}^r$ encodes the position of the output head. Notice that m and r can always be found, as we have defined $k(n)$ so that $\mathtt{int}$ always has at least two elements.

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

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

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

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 $\mathtt{inl}(s')$ 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 $\mathtt{inr}(d)$ 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 is straightforward in the presence of iteration.

To simulate the computation of the Turing Machine 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

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

for suitable A and B:

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

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

$$\begin{aligned}&t_\varphi\colon A\cdot([\mathtt{int}] \multimap [\Sigma_\square]) \multimap ([\mathtt{int}^r] \multimap [\Sigma_\square])\\ &t_\varphi = \lambda w.\, \lambda n'.\, \mathtt{let}\ [n] = n'\ \mathtt{in}\ \mathtt{loop}\ \mathtt{step}(w,n)\ [\mathtt{init}_0]\end{aligned}$$

for some suitable A. □

6.2. Nondeterministic Logarithmic Space

The functions computable in non-deterministic logarithmic space (NFLOGSPACE) can be captured much in the same way as those in FLOGSPACE. 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.

The class NFLOGSPACE is defined to consist of all functions $f\colon \Sigma^* \to \Sigma^*$ for which there exists a non-deterministic Turing Machine M_f with logarithmic space usage and the following property: $f(w) = v$ if and only if M_f accepts input w and with v written on its output tape. Note that this implies that M_f is single-valued. Given input w, it may accept only with output $f(w) = v$; it must not accept with any other output. In the literature, the class of functions definable in non-determinism logarithmic space is also called FNL [3] and is defined to be $\text{FLOGSPACE}^{\text{NLOGSPACE}}$ – the set of functions that can be computed in logaritmic space with access to an NLOGSPACE-oracle. It is not hard to see that this definition is in fact equivalent to the definition of NFLOGSPACE above. A proof of the equivalence can be found in [4, Prop. 3.1].

We extend the base language with non-determinism by adding a new constant $\vdash$ `ndchoice` : 1 + 1: Its two reduction rules make `ndchoice` a non-deterministic choice operator:

$$\mathtt{ndchoice} \mapsto_k \mathtt{inl}(*) \qquad\qquad \mathtt{ndchoice} \mapsto_k \mathtt{inr}(*)$$

The reader should reconsider the definition of what it means for a term t of type $\vdash\ t : A \cdot \mathbb{B}^m \multimap \mathbb{B}^r$ to represent a function $\varphi\colon \Sigma^* \to \Sigma^*$ in the light of the non-determinism in the base language: For any word w and any $c \in \Sigma$, if we choose the bit-width of integers such that $|w|$ can be represented as an `int` value, then there exists a reduction sequence from $t\ \langle w\rangle_m\ \langle i\rangle_r$ 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 single-valued 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 26 (NFLOGSPACE 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 24 for LOGSPACE soundness. For the reduction of `ndchoice` we use non-deterministic choice. □

Theorem 27 (NFLOGSPACE 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 NFLOGSPACE Turing Machine that computes φ. We use the formulation of non-deterministic Turing Machines where a machine has two deterministic transition functions and decides non-deterministically in each step which of the two functions to apply.

We construct a term t_φ just as in the proof to Theorem 25. The state S of the machine is exactly as in Theorem 25. For the two step functions of M_φ we define terms $\mathtt{step}_0^1(c,s,n)$ and $\mathtt{step}_0^2(c,s,n)$ with

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

as in Theorem 25. From this we define the non-deterministic step function $\mathtt{step}_0(c,s,n)$ by:

$$c\colon \Sigma,\ n\colon P,\ s\colon S \ \vdash\ \mathtt{if}\ \mathtt{ndchoice}\ \mathtt{step}_0^1(c,s,n)\ \mathtt{else}\ \mathtt{step}_0^2(c,s,n) : S + \Sigma_\square$$

The rest of the construction of t_φ is exactly as in Theorem 25.

Non-deterministic choices are thus accounted for in the definition of $\mathtt{step}_0(c,s,n)$ by using the constant `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. □

7. Related Work

7.1. Linear Logic

Linear Logic [16] has played an important role in the implicit characterisation of complexity classes by functional programming languages. *Intuitionstic Linear Logic* has the formulae α, $X \otimes Y$, $X \multimap Y$, $!X$ and $\forall\alpha.\,X$. Under the Curry-Howard correspondence, these can be seen as types of a polymorphic λ-calculus. The resulting type system refines System F.

Girard, Scedrov and Scott [19] realised that by restricting the exponential $!X$, it is possible to carve out a class of System F terms that captures the functions computable in polynomial time. They introduced *Bounded Linear Logic* (BLL) [19], which replaces the exponential $!X$ with a bounded exponential $!_{x<p}X$, where p is a polynomial. While the unbounded exponential $!X$ represents an arbitrary number of copies of X, the bounded exponential $!_{x<p}X$ represents a polynomial number of copies of X, one for each number less than p.

If one further restricts universal quantification to $\forall\alpha \le p.\,X$, then one can identify the functions computable in logarithmic space. This has led to *Stratified Bounded Affine Logic* (SBAL) [42] with formulae of the form $\alpha(\vec{p})$, $X \otimes Y$, $X \multimap Y$, $!_{x<p}X$ and $\forall\alpha \le p.\,X$.

The definition of INTML can be seen as the result of a simplifying SBAL for programming purposes. The formulae of SBAL play the same role as the interactive types in INTML. In order to make the type system as simple as possible, INTML does not have polymorphism, and the resource polynomials have been enriched to become base language types.

The removal of polymorphism is motivated by programming language practice. In System F and the above-mentioned linear logics, data types such as natural numbers are defined using impredicative encodings [17]. In SBAL it is possible to represent the natural numbers $\mathbb{N}_p = \{0,\dots,p\}$ using a type of the form $\forall\alpha \le q.\,!_{x<p}(\alpha(x) \multimap \alpha(x+1)) \multimap \alpha(0) \multimap \alpha(p)$. In practical programming languages it is more common to find such data types as primitive base types. INTML removes polymorphism for simplicity and adds primitive data types instead. After removing polymorphism from SBAL, one is left with types of the form $X \otimes Y$, $X \multimap Y$ and $!_pX$. The now missing type of natural numbers may be added as a primitive type of the form $[\mathbb{N}_p]$, which is to be understood like `[int]` in INTML. The bounded exponential $!_pX$ corresponds to the subexponential $\mathbb{N}_p \cdot X$ in INTML. Indeed, if we consider $\mathbb{N}_p$ as a base language type, then t he interactive interpretation of $[\mathbb{N}_p]$, $X \otimes Y$, $X \multimap Y$ and $\mathbb{N}_p \cdot X$ described in this paper is essentially the same as the interactive interpretation of SBAL in [42].

But adding a base type $[\mathbb{N}_p]$ alone is not enough. We also need to add terms to be able to work with it. For example, one may expect there to be an addition constant $\mathtt{add}\colon [\mathbb{N}_p] \multimap \mathbb{N}_p \cdot [\mathbb{N}_p] \multimap [\mathbb{N}_{p+q}]$. But a good choice of primitive constants is not obvious. Since values of type $\mathbb{N}_p$ can be stored in space $\log p$ using a binary encoding, they can be stored fully in memory. The programmer should expect to be able to work with them without any language constraint. At the same time the language must guarantee that the use of large data does not lead beyond logarithmic space. Allowing unconstrained manipulation of small values that fully fit in memory was one motivation for making the base language visible to the programmer in INTML.

That INTML uses base types in place of polynomials is also a result of a simplification. While it is possible to formulate INTML just with types of the form $[\mathbb{N}_p]$, $X \otimes Y$, $X \multimap Y$ and $\mathbb{N}_p \cdot X$, the need for manipulation of polynomials would make the language more complicated. Irrelevant encoding details would become visible in the type system. For

example, the INTML type system works with $(A \times B) \cdot X$ instead of $A \cdot (B \cdot X)$. If we allow only base types of the form $\mathbb{N}_p$, then $\mathbb{N}_p \cdot (\mathbb{N}_q \cdot X)$ could become $\mathbb{N}_{p \cdot q} \cdot X$, as it is possible to represent pairs of numbers in $\mathbb{N}_p$ and $\mathbb{N}_q$ by a single number in $\mathbb{N}_{p \cdot q}$. But in an implementation one may like to use a fast and easy-to-implement pairing function that just concatenates the bits of the two numbers. Then one would need to replace $\mathbb{N}_{p \cdot q}$ by $\mathbb{N}_{r \cdot q}$, where r is an upper bound for next the power of two that is larger than p (it is not even obvious how to write a good bound as a polynomial). This means that small changes in the implementation would influence the type system. It seems more natural to separate concerns by writing $(\mathbb{N}_p \times \mathbb{N}_q) \cdot X$ in the type system and by handling the encoding details on the base language level.

Dual Light Affine Logic. The concrete definition of the interactive language of INTML is inspired by Baillot and Terui's Dual Light Affine Logic (DLAL) [6]. INTML treats the subexponential $A \cdot X$ 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 $A \cdot X \multimap Y$. The restriction to negative occurrences of $A \cdot X$ much simplifies the type system without being limiting in applications.

Subexponentials. We use the term subexponential by analogy with the subexponentials introduced by Nigam and Miller [37]. Nigam and Miller study a logic with labelled subexponentials. Depending on the label L, weakening and contraction may or may not be allowed for $!_L$ and it may be possible to go from two subexponentials with a certain labels $!_L!_K X$ to one with a third label $!_R X$.

In INTML, the subexponential $A \cdot X$ is labelled with a base type A. All INTML subexponentials allow weakening. Since our base language does not contain infinite types, they do not allow contraction. If one added a base type $\mathbb{N}$ of unbounded natural number, then $\mathbb{N} \cdot X$ would support contraction. Moreover, the phenomenon of combining two subexponentials into a third one appears in that $A \cdot (B \cdot X)$ becomes $(A \times B) \cdot X$. Rule STRUCT also allows conversion between different subexponentials.

7.2. Interaction Semantics

Geometry of Interaction Situations. Abramsky, Haghverdi & Scott [1] study an axiomatic framework for Girard's Geometry of Interaction (GoI) [18]. 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 case this would amount to assuming a type $\mathbb{N}$ of unbounded natural numbers with properties such as $\mathbb{N} + \mathbb{N} \triangleleft \mathbb{N}$. 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, see Proposition 21 and Theorems 24 and 26.

Game Semantics. The idea of viewing λ-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 [5, 12]. Indeed, our construction is quite similar to AJM games for multiplicative and exponential linear logic [2], the main difference being the way we treat the exponentials. While we use a simple construction of subexponentials $A \cdot X$, in AJM-games the exponentials are constructed from $\mathbb{N} \cdot X$ 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.

Geometry of Synthesis. The way to see λ-terms as circuits is similar to Ghica's Geometry of Synthesis [14], which is defined on bSCI as described in Section 4.2. 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 an 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 λ-calculus reduction [32]: 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 bounds on the space consumption of programs is are considered. Mackie's treatment of copying amounts to what happens when one adds to the base language a type of binary trees $T(\alpha)$:

$$\begin{aligned} \texttt{type}\ T(\alpha) \ &= \ \mathit{Empty}\ \texttt{of}\ 1 \\ &\mid \ \mathit{Leaf}\ \texttt{of}\ \alpha \\ &\mid \ \mathit{Node}\ \texttt{of}\ T(\alpha) \times T(\alpha) \end{aligned}$$

With such a type 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.

Defunctionalization. We have developed INTML directly from from an analysis of the demand-driven computation that one finds in logarithmic space algorithms. However, it turns out that the translation from the interactive language to the base language is closely related to the more traditional compilation techniques of CPS-translation [22] and defunctionalization [40]. These two techniques allow the translation of a higher-order functional language to a first-order language with tail recursion. Given a higher-order language, one may first apply a CPS-translation to fix the evaluation order and then use defunctionalization to reduce the resulting higher-order program to a first-order program.

In INTML we have used as the base language a simple first-order language with a loop construct. It is equally possible to use a first-order language with tail recursion instead. For each message passing node and each incoming wire we could define a first-order function that takes the incoming message as argument, that computes the output message and that then invokes the first-order function of the destination node with the resulting output message with a tail call. The translation from the higher-order interactive language to the first-order language with tail recursion that one obtains in this way is closely related to the program one obtains by call-by-name CPS-translation [22] followed by a particular defunctionalization procedure [7]. Details are described in [44].

7.3. Implicit Characterisations of Logarithmic Space

The work in this paper builds on earlier work on implicit characterisations of logarithmic space complexity classes. While implicit characterisations of complexity classes often focus on minimal formal systems, such as function algebras [36, 38, 39] or while-languages [26], the results are useful in obtaining a better understanding of the resource usage properties of programming languages. Bonfante [8] shows how to capture LOGSPACE by a first-order functional language. In INTML we work towards extending such results to higher-order functional programming languages. For example, we believe that Møller-Neergaard's function algebra $\mathrm{BC}^{-}_{\varepsilon}$ can be implemented in INTML. The LOGSPACE soundness of this function algebra depends on a complicated implementation of recursion with computational amnesia [36]. An INTML program for this can be found as an example in the experimental implementation of INTML[1]. An interesting direction for further work is to consider lazy data structures and corecursion in INTML and the relation to the work of Ramyaa and Leivant [39]. Where we use functions to represent long strings, Ramyaa and Leivant use streams.

8. Conclusion

We have found that an interactive approach to computation, as formalised by 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 FLOGSPACE. Practical experience with an experimental implementation of INTML suggests that, with suitable type inference, INTML can be made to be quite usable [11].

[1] Available from: `http://www.github.com/uelis/IntML`

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 [37].

References

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

[2] S. Abramsky, R. Jagadeesan, P. Malacaria, Full abstraction for PCF, Information and Computation 163 (2) (2000) 409–470.

[3] C. Àlvarez, J. L. Balcázar, B. Jenner, Functional oracle queries as a measure of parallel time, in: Symposium on Theoretical Aspects of Computer Science, STACS 1991, 1991, pp. 422–433.

[4] C. Àlvarez, B. Jenner, A note on logspace optimization, Computational Complexity 5 (2) (1995) 155–166.

[5] P. Baillot, Approches dynamiques en sémantique de la logique lineaire: jeux et géometrie de l'interaction, Ph.D. thesis, University Aix-Marseille II (1999).

[6] P. Baillot, K. Terui, Light types for polynomial time computation in lambda calculus, Information and Computation 207 (1) (2009) 41–62.

[7] A. Banerjee, N. Heintze, J. G. Riecke, Design and correctness of program transformations based on control-flow analysis, in: N. Kobayashi, B. C. Pierce (eds.), Theoretical Aspects of Computer Software, TACS 2001, vol. 2215 of Lecture Notes in Computer Science, Springer, 2001, pp. 420–447.

[8] G. Bonfante, Some programming languages for Logspace and Ptime, in: Algebraic Methodology and Software Technology, AMAST 2006, vol. 4019 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2006, pp. 66–80.

[9] S. Cook, P. McKenzie, Problems complete for deterministic logarithmic space, Journal of Algorithms 8 (3) (1987) 385–394.

[10] U. Dal Lago, U. Schöpp, Functional programming in sublinear space, in: European Symposium on Programming, ESPO 2010, vol. 6012 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 205–225.

[11] U. Dal Lago, U. Schöpp, Type inference for sublinear space functional programming, in: Asian Conference on Programming Languages and Systems, APLAS 2010, vol. 6461 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 376–391.

[12] V. Danos, H. Herbelin, L. Regnier, Game semantics and abstract machines, in: Logic in Computer Science, LICS 1996, IEEE, Los Alamitos, CA, 1996, pp. 394–405.

[13] J. Egger, R. E. Møgelberg, A. Simpson, Enriching an effect calculus with linear types, in: Computer Science Logic, CSL 2009, vol. 5771 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2009, pp. 240–254.

[14] D. R. Ghica, Geometry of synthesis: a structured approach to VLSI design, in: Principles of Programming Languages, POPL 2007, ACM, New York, NY, 2007, pp. 363–375.

[15] D. R. Ghica, A. Smith, Geometry of synthesis III: resource management through type inference, in: T. Ball, M. Sagiv (eds.), Principles of Programming Languages, POPL 2011, ACM, 2011, pp. 345–356.

[16] J. Girard, Linear logic, Theoretical Computer Science 50 (1987) 1–102.

[17] J. Girard, P. Taylor, Y. Lafont, Proofs and Types, Cambridge University Press, 1989.

[18] 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.

[19] 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.

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

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

[22] M. Hofmann, T. Streicher, Continuation models are universal for lambda-mu-calculus, in: Logic in Computer Science, LICS 1997, IEEE, Los Alamitos, CA, 1997, pp. 387–395.

[23] N. Hoshino, A modified GoI interpretation for a linear functional programming language and its adequacy, in: M. Hofmann (ed.), Foundations of Software Science and Computational Structures, FOSSACS 2011, vol. 6604 of Lecture Notes in Computer Science, Springer, 2011, pp. 320–334.

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

[25] N. Immerman, Descriptive complexity, Graduate texts in computer science, Springer, 1999.

[26] N. D. Jones, LOGSPACE and PTIME characterized by programming languages, Theoretical Computer Science 228 (1-2) (1999) 151–174.

[27] N. D. Jones, The expressive power of higher-order types or, life without CONS, Journal of Functional Programming 11 (1) (2001) 5–94.

[28] A. Joyal, R. Street, D. Verity, Traced monoidal categories, Mathematical Proceedings of the Cambridge Philosophical Society 119 (3) (1996) 447–468.

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

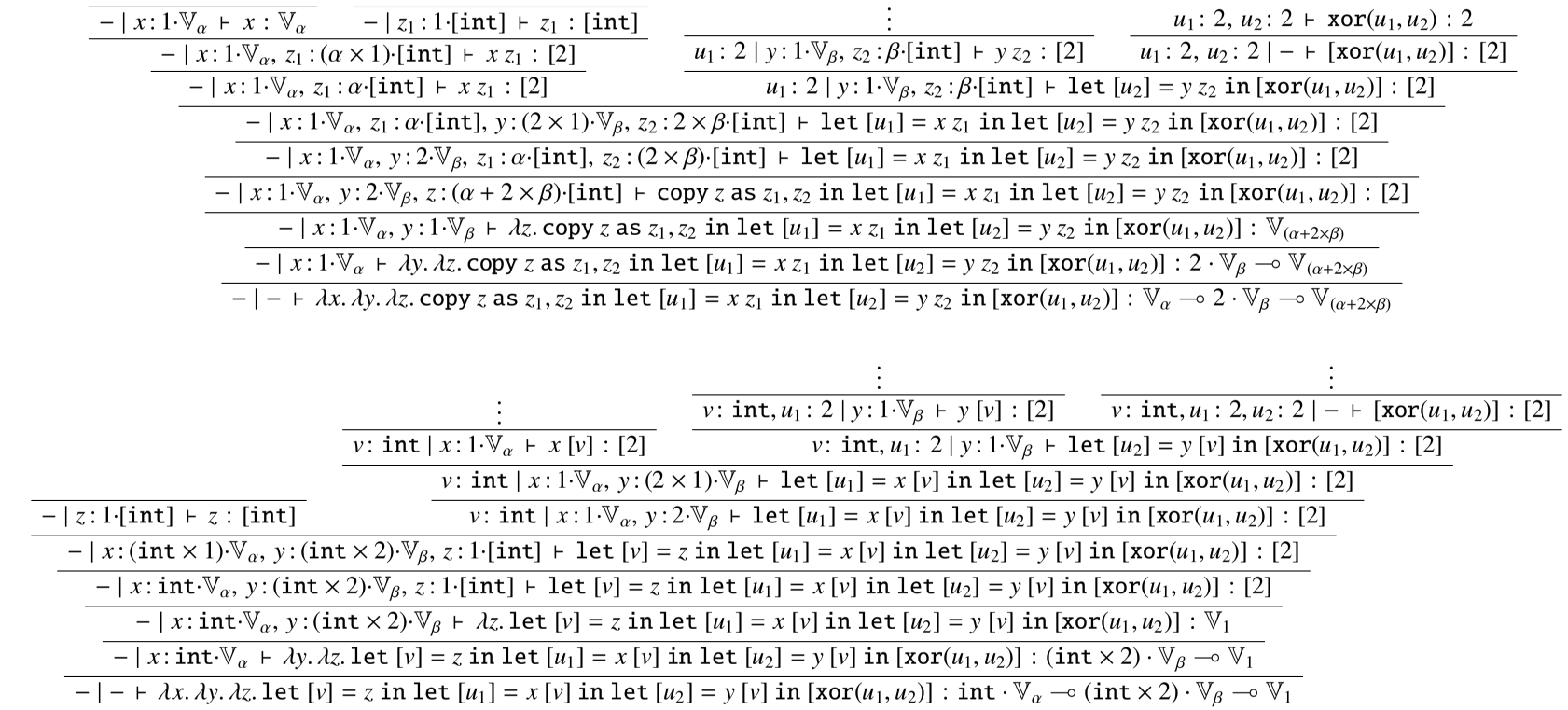
[30] J. Laird, Full abstraction for functional languages with control, in: Logic in Computer Science, LICS 1997, IEEE, Los Alamitos, CA, 1997, pp. 58–67.

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

[32] I. Mackie, The geometry of interaction machine, in: Principles of Programming Languages, POPL 1995, ACM, New York, NY, 1995, pp. 198–208.

[33] H. G. Mairson, From hilbert spaces to dilbert spaces: Context semantics made simple, in: Foundations on Software Technology and Theoretical Computer Science, FSTTCS 2002, Springer, Berlin, Heidelberg, 2002, pp. 2–17.

[34] P.-A. Melliès, Functorial boxes in string diagrams, in: Computer Science Logic, CSL 2006, vol. 4207 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2006, pp. 1–30.
[35] S. Muthukrishnan, Data streams: algorithms and applications, Foundations and trends in theoretical computer science, Now Publishers, Hanover, MA, 2005.
[36] P. M. Neergaard, A functional language for logarithmic space, in: Asian Symposium on Programming Languages and Systems, APLAS 2004, vol. 3302 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2004, pp. 311–326.
[37] V. Nigam, D. Miller, Algorithmic specifications in linear logic with subexponentials, in: Principles and Practice of Declarative Programming, PPDP 2009, ACM, New York, NY, 2009, pp. 129–140.
[38] I. Oitavem, Logspace without bounds., Frankfurt am Main: Ontos Verlag, 2010, pp. 355–362.
[39] R. Ramyaa, D. Leivant, Ramified corecurrence and logspace, Electronic Notes in Theoretical Computer Science 276 (2011) 247–261.
[40] J. C. Reynolds, Definitional interpreters for higher-order programming languages, in: Proceedings of the ACM annual conference - Volume 2, ACM '72, ACM, 1972, pp. 717–740.
[41] 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.
[42] U. Schöpp, Stratified bounded affine logic for logarithmic space, in: Logic in Computer Science, LICS 2007, IEEE, Los Alamitos, CA, 2007, pp. 411–420.
[43] U. Schöpp, Computation-by-interaction with effects, in: Asian Symposium on Programming Languages and Systems, APLAS 2011, vol. 7078 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2011, pp. 305–321.
[44] U. Schöpp, On the relation of interaction semantics to continuations and defunctionalization, Logical Methods in Computer Science 10 (4) (2014) 1–41.



Abbreviation: $\mathbb{V}_\alpha$ denotes $\alpha \cdot [\mathtt{int}] \multimap [2]$.
Applications of rule Exch have been hidden.

Figure 17: Typing Derivations for Judgements from Section 2.6