The Geometry of Interaction as a Module System

Ulrich Schöpp

April 15, 2018

Abstract

The Geometry of Interaction (GOI) was originally introduced by Girard in the context of Linear Logic. Many of its recent applications concern the interpretation and analysis of functional programming languages, with applications ranging from hardware synthesis to quantum computation. In this paper we argue that for such programming-language applications it is useful to understand the GOI as a module system. We show that the structure of particle-style GOI models can naturally be expressed by a standard ML-like module system. This provides a convenient, familiar formalism for working with the GOI that abstracts from inessential implementation details. With this view, the GOI becomes a method of implementing ML-style module systems for a wide range of programming languages. It can be considered as a higher-order generalisation of system-level linking. The relation between the GOI and the proposed module system is established by developing a linear version of the F-ing modules approach of Rossberg, Russo and Dreyer that uses as new decomposition of the exponential rules of Linear Logic. We illustrate the utility of the modular view of the GOI with examples on game semantics and parallel evaluation.

1 Introduction

Modularity is very important for software construction. To make the development of large-scale systems manageable, it is important to be able split them into small modules that can be specified, developed and verified independently. Virtually all programming languages have support for some kind of modular programming, the module system of Standard ML being a particularly expressive example. For the effective application of formal methods, modularity is also becoming increasingly important at a small scale. To make complex formal methods scale to programs of realistic size, it is desirable to decompose even small programs into tiny fragments that are easier to treat with formal methods than larger ones. To support such kinds of modular programming and reasoning, a good understanding of modularity and module systems for programming languages is essential.

This paper connects the Geometry of Interaction (GOI) to ML-style module systems. The GOI was originally proposed by Girard in the context of Linear Logic [9]. It has since found many applications in programming languages, especially in situations where one wants to design higher-order programming languages for some restricted computational model. Examples are hardware circuits [8], LOGSPACE-computation [5], quantum computation [12], distributed systems [7], etc. These applications use the particle-style variant of the GOI, which constructs a model of higher-order programming languages in terms of dialogues between simple interacting entities (see Section 1.2 for an outline). If one lets the interacting entities be programs from the restricted computational model, then such models can be used for compilation. Interpretation of higher-order programs in the model becomes translation to the restricted computational setting.

This paper is about expressing the model constructions of the GOI in simple familiar terms. We relate the GOI to a fairly standard ML-like module system, which allows us to express GOI-constructions in terms of the structures, signatures and functors that are familiar from Standard ML and OCaml.

From a practical point of view, the message of this paper is that the GOI constructs an ML-style module system even for very simple programming languages. The construction requires few assumptions and applies in particular to low-level languages that capture computation in some restricted universe of computation. The module system constructed in this paper is higher-order in the sense that modules can be parameterised over modules that themselves may depend on external modules. This higher-order structure is useful, for example to implement higher-order programming languages like Idealized Algol efficiently in

computational models that lack higher-order functions. The existing applications of the GOI in loc. cit. provide more examples.

From a theoretical point of view, the message of this paper is that an ML-style module system is a convenient formalism for working with the constructions of the GOI. Presentations of the GOI usually start from scratch by giving a low-level implementation of constructions that are essentially standard. One such presentation is outlined in Section 1.2 below. There are many similar presentations of the GOI that differ only in low-level implementation choices. For non-experts interested in programming language applications of the GOI, such construction present a hurdle to overcome before they can get to the actual applications of the GOI. For experts, the low-level details are not very interesting. We suspect that even the authors of such work would prefer to not read or write them anymore. By capturing the constructions of the GOI in terms of a standard module system, we provide a formalism that abstracts from arbitrary implementation details and that is already familiar to many programmers.

In the literature, it is common to use variants of System F to abstract from implementation details in the GOI. To understand its use in applications of the GOI, one needs to understand a particular GOI-interpretation of System F. This may still be unfamiliar to non-experts and we believe that the intuituion for this interpretation can be expressed more naturally using a module system. But this is a minor point that may be just a matter of taste. More important is that System F is not very convenient for working with abstract types via existentials, as can be seen in e.g. [20, 22]. Indeed, the Standard ML module system can be seen as a mode of use for System F_ω [19] that makes programming with abstract types much more convenient for the programmer.

From a technical point of view, the contribution of this paper is to work out the technical details of an implementation of a module system that can be seen as a higher-order generalisation of systems-level linking. It has zero overhead in applications that just need standard systems-level linking, but it can also handle higher-order linking. We use the approach of F-ing modules [19] for the definition of a module system for the structure of the GOI, somewhat like 1-ML [17] does for System F_ω. We also show that F-ing is suitable for practical implementation.

Let us now give a more concrete outline of the paper's content.

1.1 From Systems-Level Linking to Higher-Order Modules

The first way to read this paper is from a practical point of view, as a generalisation of systems-level linking to a proper higher-order ML-style module system.

We outline the idea using a simple generic first-order language. It has first-order types (integers, pairs, disjoint unions, ...) and allows (mutually recursive) function definitions. A simple example program should be enough to understand the features of this language (details appear in Sections 2 and 3, but are not really important at this point):

```
fn fact_aux(x: int, acc: int) → int {
  if x = 0 then return acc
  else let x = fact_aux(x - 1, acc * x) in return x
}
fn fact(x: int) → int {
  let y = fact_aux(x, 1) in return y
}
```

In practical applications, such languages are rarely used without some form of module system. At the very least, one would like to split larger programs into several files. A standard approach to do this is to use the linker provided by standard operating systems. Individual files compile to object files, which contain the information which functions are defined in them and which external functions are being used in them. One can think of the interface X of an object file as consisting of two sets X^- and X^+ that contain the function names defined in the module and the external functions used in the module respectively. The system linker uses this information to glue several object files together. It can be seen as a weak first-order module system.

This paper may be read as a direct generalisation of this kind of linking to a proper higher-order ML-style module system. The paper constructs such a module system for the kind of first-order systems-level programming languages that we have just considered. The construction is definitional in the sense that is works without extending the first-order language. We believe that it should be particularly useful for

low-level languages, like LLVM-IR, that offer few high-level programming features to support modular programming.

In essence, we define a basic ML-style module system as syntactic sugar on top of the first-order language. First, we have *structures*, which are records that collect function definitions from the first-order language. A simple example structure appears on the left below. The type of a structure is a *signature* that records the types of all (public) components. The type of the structure is shown on the right below.

```
struct                                          sig
  fn fact_aux(x:int, a:int) → int {...}           fact_aux: int×int → int
  fn fact(x: int) → int {...}                     fact: int → int
  S = struct                                      S: sig f: A → B end
    fn f(x: A) → B {...}                        end
  end
end
```
(1)

Next, *functors* are parameterised structures. In the simplest case, a functor allows one to write a program that uses external functions, for example:

```
functor(X: sig f: int → int end) →
    struct
      fn f(x: int) → int { let y = X.f(x+1) in return y }
      fn g(x: bool) → int { if x then X.f(0) else return 0 }
    end
```
(2)

The type of a functor is a functor signature. In this particular case it is:

```
functor(X: sig f: int → int end) →
    sig
      f: int → int
      g: bool → int
    end
```
(3)

The kinds of structures and functors defined so far can be seen as simple syntactic sugar for what one can already do with systems-level linking. The structure (1) would be sugar for a first-order program of the form

```
fn fact_aux(x: int, acc: int) → int { ... }
fn fact(x: int) → int { ... }
fn S.f(x: A) → B { ... }
```

in which `S.f` is just a function name (the dot being a distinguished character). Sub-structures are included by taking all the functions from the sub-structure and prefixing them with "`S.`". The above functor (2) can be seen as syntactic sugar for the first-order program

```
fn res.f(x: int) → int { let y = arg.f(x+1) in return y }
fn res.g(x: bool) → int { if x then arg.f(0) else return 0 }
```

where the bodies of the functions contain calls to the external function `arg.f: int→ int` belonging to the functor argument. The functions belonging to the functor argument are prefixed with "`arg.`" to avoid name-clashes with the functions for the functor result, which are prefixed with "`res.`".

If one only allows first-order functors, i.e. functors that take structures (but not functors or structures containing functors) as argument, this simple module system is just a way of formulating systems-level linking. Functor application works by renaming the argument to add the prefix "`arg.`" to all function names, concatenating it with the body of the functor and then removing the prefix "`res.`" from the functions for the result. Suppose, for example, that we want to apply the functor (2) to a structure that represents the following first-order program:

```
fn f(x: int) → int { return (x+1) }
```

We would just rename `f` to `arg.f`, add the resulting function definition to the first-order definitions from the functor, and remove "`res.`"-prefixes. The result is

```
fn f(x: int) → int { let y = arg.f(x+1) in return y }
fn g(y: bool) → int { if x then arg.f(0) else return 0 }
fn arg.f(x: int) → int { return (x+1) }
```

which amounts to systems-level program linking.

This approach works very well for first-order functors, but it is too simple to handle the higher-order case, where functor arguments can be functors too. Consider, for example the following functor definition, in which FS abbreviates the functor signature from (3).

```
functor (F: FS) →
  struct
    A1 = struct f = fn(x:int) → int { return x+1 } end
    A2 = struct f = fn(x:int) → int { return x+2 } end
    h = fn (x:int) → int {
      let y1 = F(A1).g(x) in let y2 = F(A2).g(x) in
      return y1+y2
    }
  end
```
(4)

If we try to see it as syntactic sugar like above, then it its unclear how to handle the two functor applications F(A1) and F(A2):

```
fn res.A1.f(x: int) → int { return x+1 }
fn res.A2.f(x: int) → int { return x+2 }
fn res.h(x: int) → int {
  let y1 = arg.g(x) /  ???  / in let y2 = arg.g(x) /  ???  / in
  return y1+y2
}
```

In the first call to g, the function arg.arg.f should be bound to res.A1.f, while in the second it should be bound to res.A2.f.

There are several ways of solving this problem. One can duplicate the actual argument for F upon application [6], one can use closures to implement higher-order functors, and so on. But suppose one wants to maintain the view of the module system as a formalisation of program linking, where functor application is implemented by linking functor program and argument. The GOI provides interesting applications for such an approach, such as [8, 23]. It may be interesting to consider in conjunction with recent work on link-time-optimisation [14].

If one wants to maintain the view of functor application as program linking, then one can use *indices* to solve the problem with higher-order application. One may assume that to implement a functor application, the linker adds a new first argument (of a suitable type index) to all functions in the actual functor argument F. In the case of the problematic application above, this would mean that the linker adds the following function definitions to the functor code:

```
fn arg.f(i:index, x:int) → int {let y = arg.arg.f(i, x+1) in return y}
fn arg.g(i:index, x:bool) → int {if x then arg.arg.f(i, 0) else return 0}
```

With such an assumption, the higher-order functor can be implemented as follows:

```
fn res.A1.f(x: int) → int { return x+1 }
fn res.A2.f(x: int) → int { return x+2 }
fn res.h(x: int) → int {
  let y1 = arg.g(1, x) in let y2 = arg.g(2, x) in
  return y1+y2
}
fn arg.arg.f(i: index, x: int) → int {
  case i of 1 → res.A1.f(x) | 2 → res.A2.f(x)
}
```

In the calls to arg.g, we pass an index that identifies which instantiation of the argument functor is being used. When the argument functor calls a function from its own argument, which in this case must be a call to arg.arg.f, then we dispatch on the index to call the function in the correct actual functor argument.

This simple idea of having the linker add an index can be made to work in general for higher-order modules. Functor application remains linking in the sense that the code coming from functor and argument are just concatenated. The only difference is that linking may now involve adding additional index arguments

of suitable type. At higher-order, such arguments may need to be added both to the argument and to the functor. In practice, one would want to add arguments only when necessary, one would like to choose a precise as possible type for `index` (in this case `unit + unit`), and one would like to reduce case distinction as much as possible. This is much like in defunctionalisation, which is related to the approach [21]. There is evidence that the approach can be used for efficient compilation even when the higher-order structure is used extensively [23].

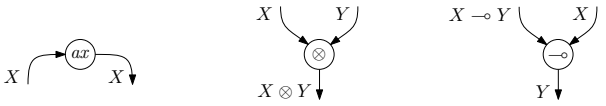
The module system introduced in this paper implements functors in this way. Of course, it also allows type definitions and type abstractions. These will be described later.

1.2 Particle-Style Geometry of Interaction

A second way to read this paper is as a programming-language presentation of the constructions of the particle-style Geometry of Interaction. The GOI has its origins in logic and can be seen as the construction of a model of Linear Logic. We give an outline of the most basic constructions, in order to explain how the module system in this paper is a syntax for the GOI. It will not be important to understand in what way the constructions produce a model of Linear Logic.

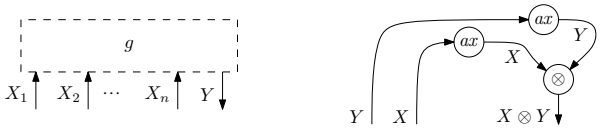
The particle-style GOI can be seen as modelling Linear Logic in terms of message-passing graphs. We outline a formulation with directed graphs that have labelled nodes and edges. Think of the nodes as stateless processes that pass messages along edges. Each edge has a label X that defines two sets X^- and X^+. These sets specify what messages may be passed along the edge: elements of type X^+ may be passed with the direction of the edge, and elements of type X^- against the direction of the edge. The nodes are passive until they receive a message along one of the edges connected to them. They then process the incoming message, construct a new outgoing message, which they finally send along an edge of their choice. Nodes have extremely simple behaviour, such as 'upon receipt of v on one port, send out $\mathtt{inl}(v)$ one another port'. The node label determines the behaviour of nodes.

To model Linear Logic, one builds message-passing graphs from a fixed set of standard nodes that corresponds to the proof rules. For Intuitionistic Linear Logic (with $\otimes$ and $\multimap$) one can use three kinds of nodes



which are defined for arbitrary X and Y. The edge labels are formulae. The labels $X \otimes Y$ and $X \multimap Y$ determine sets $(X \otimes Y)^- = X^- + Y^-$, $(X \otimes Y)^+ = X^+ + Y^+$, $(X \multimap Y)^- = X^+ + Y^-$ and $(X \multimap Y)^+ = X^- + Y^+$, where $+$ stands for disjoint union. The same nodes are also available with all edge directions reversed (this presentation has some redundancy). The message-passing behaviour of these nodes is essentially already determined by the edge labels. In all cases, there is just one canonical way of passing on a message of a particular type. For example, if node $\multimap$ receives message $\mathtt{inr}(v)$ through the edge labelled with $X \multimap Y$, then $v \in Y^+$ and the only reasonable action is to pass v along the edge labelled with Y. The above nodes treat all other messages analogously in the canonical way. The graphs are assumed to be locally ordered, so that the two edges going into $\otimes$ can be distinguished even if, for instance, X and Y are the same.

With this choice of nodes, one can interpret proofs of Intuitionistic Linear Logic. A proof of a sequent $X_1, \ldots, X_n \vdash Y$ in Linear Logic is translated to a graph g as shown on the left below.

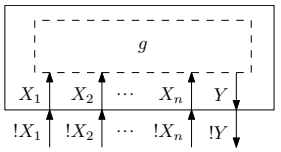


Proof rules are implemented in the canonical way using the above nodes. For example, a $\otimes$-introduction rule for going from $\Gamma \vdash X$ and $\Delta \vdash Y$ to $\Gamma, \Delta \vdash X \otimes Y$ is interpreted by taking the union of the two graphs and adding a $\otimes$-node to connect the two edges labelled with X and Y to one edge labelled $X \otimes Y$. The canonical proof of $Y,\ X \vdash X \otimes Y$ would lead to the graph on the right above.

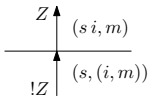
One obtains a message-passing interpretation of Intuitionistic Linear Logic. The message-passing behaviour of the graph obtained from a proof identifies the proof. The approach can also be seen as a presentation of Abramsky-Jagadeesan-Malacaria game semantics [2].

It is not hard to extend this approach to cover also the exponential $!X$. To implement the exponentials, one uses an edge label $!X$ with $(!X)^- = \mathbb{N} \times X^-$ and $(!X)^+ = \mathbb{N} \times X^+$. The formula $!X$ represents infinitely many copies of X and the new first component in $(!X)^-$ and $(!X)^+$ represents the number of the copy that the message is intended for. It is usually called the *index*. To keep track of indices, messages are now extended with a stack. This means that messages now have the form (s, m), where m is a message as before and $s \in \mathbb{N}^*$ is a list of indices. Where one could previously pass message m, one now passes message (s, m). The graph nodes have essentially the same behaviour as before. They ignore the new stack component and just pass it on unchanged. For example, if node $\multimap$ receives a message of the form $(s, \mathtt{inr}(v))$ on its edge labelled with $X \multimap Y$, then it reacts by sending message (s, v) on its edge labelled with Y.

The stack in messages is only ever modified by a new box construction. This construction allows one to construct a new graph node by putting a graph inside a box, as shown on the left below.

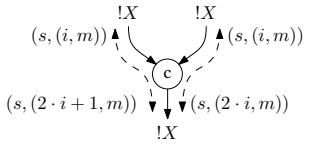


Message-passing:



Each incoming edge of the new node corresponds to an input edge into the boxed graph, and each outgoing edge from the new node corresponds to an output edge of the boxed graph. The only difference is that all labels change from X to $!X$ when stepping outside of the box. In terms of message passing, crossing the border of the box corresponds to pushing to and popping from the message stack. This is shown on the right in the figure above. When $(s, (i, m))$ arrives outside the box, $(s\,i, m)$ is passed inside and then message passing is performed inside the box. When $(s\,i, m)$ arrives at the inside border of a box, the message $(s, (i, m))$ is passed to the outside.

To implement the structural rules for the exponentials, one uses corresponding new message nodes that manipulate the index in messages. For example, contraction may be implemented using a node with the following message-passing behaviour. It is based on one particular isomorphism $\mathbb{N} \simeq \mathbb{N} + \mathbb{N}$; any other choice would be fine as well.



This very coarse outline covers the bare essentials of the standard constructions that one finds in many articles on the Geometry of Interaction, such as [13, 4, 5]. We have already mentioned at the beginning of the Introduction that there are a number of programming language applications of this structure. Many such applications are based on the observation that message-passing nodes are very simple and can be implemented easily even in very restricted computational settings. With the GOI one can build a model of a higher-order programming language in such settings in the form of message-passing graphs.

1.3 From Interaction to Modules

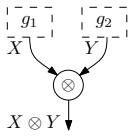
How do the constructions from the GOI relate to the module system outlined in Section 1.1?

To explain the correspondence, it is useful to outline how one can implement message-passing graphs by first-order programs. We have explained that a graph edge with label X allows elements of X^- and X^+ to be passed as messages. One way of implementing message-passing in a first-order programming language is by means of function calls. To each edge e in the graph, we associate two function labels, one for each end of the edge. The function $\mathtt{send}_e^- : X^- \to \mathtt{empty}$ is invoked to pass a message in the direction of the edge, and $\mathtt{send}_e^+ : X^+ \to \mathtt{empty}$ is invoked to pass a message in the other directions. In both cases, the return type is `empty`, as message passing cedes control to the recipient of the message. With this approach,

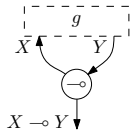
a graph node is implemented simply by defining the functions for the ends of the edges that are connected to it. For example, any *ax*-node would be implemented simply by two functions **fn** $\mathtt{send}^-_{e_1}(x)$ { $\mathtt{send}^-_{e_2}(x)$ } and **fn** $\mathtt{send}^+_{e_2}(x)$ { $\mathtt{send}^-_{e_1}(x)$ }, where e_1 is the left edge connected to the node and e_2 is the other edge.

If one implements all the nodes in a graph in this way, then only the functions for the ends of edges that have no recipient in the graph remain undefined. A graph g as shown on the right becomes a program that defines a function $\mathtt{send}^- : X^- \to \mathtt{empty}$ (in addition to internally used functions) and that may call an external function $\mathtt{send}^+ : X^+ \to \mathtt{empty}$.

The definition of $X \otimes Y$ amounts to linking two programs together. Consider the case shown on the right. The resulting program defines a function $\mathtt{send}^- : X^- + Y^- \to \mathtt{empty}$ and may call an external function $\mathtt{send}^+ : X^+ + Y^+ \to \mathtt{empty}$. Notice that a function of type $X^- + Y^- \to \mathtt{empty}$ is in one-to-one correspondence with two functions of types $X^- \to \mathtt{empty}$ and $Y^- \to \mathtt{empty}$. In particular, $\mathtt{send}^-$ corresponds to two such functions, let us call them $\ell_1.\mathtt{send}^-$ and $\ell_2.\mathtt{send}^-$. The external function $\mathtt{send}^+$ can be replaced by externals $\ell_1.\mathtt{send}^+$ and $\ell_2.\mathtt{send}^+$ in the same way. Then, the full graph corresponds to the syntactic sugar for **struct** ℓ_1 = g_1, ℓ_2 = g_2 **end** outlined in Section 1.1.



In Section 1.1, we have implemented functors simply by treating the definitions from the functor arguments as external functions that will be linked later. In the message-passing graphs, this corresponds to treating functor arguments as incoming edges that will be connected later, as for g in the figure on the right. Adding the node $\multimap$ to this graph simply amounts to packing up the two edges into one. In this case, $\mathtt{send}^-$ corresponds to $\mathtt{res}.\mathtt{send}^- : Y^- \to \mathtt{empty}$ and $\mathtt{arg}.\mathtt{send}^+ : X^+ \to \mathtt{empty}$. The external $\mathtt{send}^+$ amounts to externals $\mathtt{res}.\mathtt{send}^+ : Y^+ \to \mathtt{empty}$ and $\mathtt{arg}.\mathtt{send}^- : X^- \to \mathtt{empty}$. Notice how this agrees with the informal implementation of functors in Section 1.1.



This gives a rough outline of how the GOI corresponds to the implementation of modules in Section 1.1. The indices used there to implement higher-order functors correspond to the exponentials in the GOI. One minor restriction is that the GOI uses only has non-returning functions (with return type empty), but this is not hard to lift.

The simple module system from Section 1.1 may be considered a syntax for the constructions for the constructions of the GOI. This view should make the GOI more accessible to readers familiar with functional programming but not the GOI. More importantly, the view is more than just a notational reformulation. Once one gets to more complicated features of module systems, such as type definition and type abstraction, as defined in Section 4, using a module system provides a real benefit. This is discussed in Section 7. In effect, when presented as in Section 1.2, the GOI defines a module system *and* its low-level implementation. It is clear that the latter becomes more complicated once one gets to more sophisticated features of the module system. It also means that work on the GOI is usually formulated for ease of presentation rather than for efficient implementation.

1.4 Paper Outline

In Section 2 and 3 we fix the syntax of a core programming language. It captures what we assume about the core programming language that we want to equip with a module system. In Section 4, we then define a module system over the core language. In Section 5, we show how the constructions of the particle-style GOI allow us to view the module system as a definitional extension of the core language. To do so, we develop a linear form of F-ing [19]. In Section 6 we show how to deal with linearity in type inference. Finally, in Section 7 we outline how viewing the GOI as a module system is useful in applications.

2 Core Expressions

We fix a very basic language of computational expressions as the basis for all further constructions. Modules will organise these kinds of expressions. They also form the core expressions of first-order systems-level

programs.

$$
\begin{array}{rrcl}
\text{Core types} & A & ::= & \texttt{int} \mid \texttt{unit} \mid A \times A \mid \texttt{empty} \mid A + A \\
\text{Core values} & v, w & ::= & x \mid n \mid () \mid (v, w) \mid \texttt{inl}(v) \mid \texttt{inr}(v) \\
\text{Core expressions} & e & ::= & \texttt{return}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let}\ x = op(v)\ \texttt{in}\ e \\
 & & & \mid \texttt{let}\ (x, y) = v\ \texttt{in}\ e \mid \texttt{case}\ v\ \texttt{of}\ \texttt{inl}(x) \Rightarrow e;\ \texttt{inr}(x) \Rightarrow e
\end{array}
$$

In this grammar, n ranges over integers and op ranges over primitive operations, such as add, sub and mul. It is possible to have effectful operations, such as $print$ for I/O, or put and get for global state, as required, but we do not need to assume any specific operations in this paper. The type `int` is an example of a base type; let us assume that it represents fixed-width integers.

3 Systems-level Programs

There is hardly any programming language that consists of core expressions alone. In practice, one needs a bit of infrastructure to organise them into programs. In this section we define a simple programming language that uses first-order functions to organise core expressions. We call it a systems-level programming language, because it roughly corresponds to a very small core of systems-level programming languages like `C` or LLVM-IR.

The language may alternatively be seen as defining essentially the minimum of what one needs to implement the particle-style interaction of the GOI as outlined in Section 1.3. Suppose the behaviour of each node in an interaction graph is given by a core expression. To implement the message-passing process in the whole graph, one needs more than core expressions. For example, message-passing in a graph may lead to a looping computation that may not be implemented by core expressions alone. Systems-level programs extend core expressions just enough to implement particle-style interaction. They can be seen as a simple syntactic formulation of the mathematical assumptions in GOI-situations [1].

Systems-level programs are simple first-order programs built from core expressions. For the purposes of this paper, there is a lot of freedom in the definition of this language; here we define a simple generic language that is easy to instantiate to concrete situations.

$$
\begin{array}{rrcl}
\text{Systems-level types} & B & ::= & \text{core type constructors} \mid \texttt{raw}_k \text{ where } k \in \{0, 1, 2, \ldots, \infty\} \\
\text{Systems-level expressions} & e & ::= & \text{core expression constructors} \mid f(v) \\
 & & & \mid \texttt{let}\ x = \texttt{coerc}_{B,\texttt{raw}_k}(v)\ \texttt{in}\ e \mid \texttt{let}\ \texttt{coerc}_{B,\texttt{raw}_k}(x) = v\ \texttt{in}\ e \\
\text{Systems-level programs} & P & ::= & \text{empty} \mid \texttt{fn}\ f(x_1 \colon B_1, \ldots, x_n \colon B_n) \to B\ \{e\}\ \ P
\end{array}
$$

The phrase 'core type constructor' means that we include all cases from the grammar for core types, only now with B in place of A.

In contrast to the other calculi in this paper, the type system of systems-level programs is not intended to capture interesting correctness properties. It is nevertheless useful for implementation purposes, e.g. to statically know the size of values for efficient compilation.

A program consists of a list of function definitions of the form $\texttt{fn}\ f(x_1 \colon B_1, \ldots, x_n \colon B_n) \to B\ \{e\}$. This defines a function named f with arguments $x_1, \ldots, x_n$ of types $B_1, \ldots, B_n$, return type B and body e. The grammar of core expressions has been extended with a new case $f(v)$ for function calls. Functions are allowed to be mutually recursive.

The new types $\texttt{raw}_k$ are types of unstructured, raw data. The intention is that $\texttt{raw}_k$ is a type that provides at least k bits of storage. Type $\texttt{raw}_\infty$ can store values of arbitrary size and can be thought of like `void*` in `C`. We write just `raw` for $\texttt{raw}_\infty$.

We assume that there is a function `size` from types to $\{0, 1, 2, \ldots, \infty\}$ with the intention that $\texttt{size}(B) = k$ means that if one takes the raw underlying data of any value of type B, then it can be represented using $\texttt{raw}_k$. Of course, the function `size` can be defined only if one knows the low-level encoding for all values. We do not want to fix a concrete encoding, as there many reasonable choices. We only make the assumption that the size function is such that values of any type with size $\leq k$ can be cast without loss into values of type $\texttt{raw}_k$, as described in the following paragraph.

Values of type $\mathtt{raw}_k$ can only be constructed by casting values of some other type into this type. Using the new term $\mathtt{let}\ x = \mathtt{coerc}_{B,\mathtt{raw}_k}(v)\ \mathtt{in}\ e$ one can cast a value v of any type B into its raw underlying data x of type $\mathtt{raw}_k$. The term $\mathtt{let}\ \mathtt{coerc}_{B,\mathtt{raw}_k}(y) = w\ \mathtt{in}\ e$ allows one to interpret a raw value w as a value y of type B. This term should be read like a pattern matching operation that matches w against $\mathtt{coerc}_{B,\mathtt{raw}_k}(y)$. The term binds the variable y in e. We assume that $\mathtt{let}\ x = \mathtt{coerc}_{B,\mathtt{raw}_k}(v)\ \mathtt{in}\ \mathtt{let}\ \mathtt{coerc}_{B,\mathtt{raw}_k}(y) = x\ \mathtt{in}\ e$ behaves in the same way as $e[x \mapsto v]$ whenever $\mathtt{size}(B) \leq k$. This means that coercing a value into $\mathtt{raw}_k$ and back gives us back the original value, as long as $\mathtt{size}(B) \leq k$ holds. There are no guarantees about any other casts, such as $\mathtt{let}\ x = \mathtt{coerc}_{B_1,\mathtt{raw}_k}(v)\ \mathtt{in}\ \mathtt{let}\ \mathtt{coerc}_{B_2,\mathtt{raw}_k}(y) = x\ \mathtt{in}\ e$ where B_1 and B_2 are different types.

Note that our assumptions on the types $\mathtt{raw}_k$ can be achieved in many ways. For example, it would be sound, if perhaps inefficient, to implement all types $\mathtt{raw}_k$ as $\mathtt{raw}_\infty$. In work on the GOI, one often uses a type of unbounded natural numbers to implement raw.

Notation "x as B". To work with the coerc-terms, it is convenient to introduce some notation. For example, if $x : \mathtt{int} \times (\mathtt{unit} + \mathtt{unit})$, then we can coerce x into a value of type $x : \mathtt{int} \times \mathtt{raw}$ by coercing the second component of the pair. Writing down such terms is tedious, however. Instead, define the binary relation $\lhd$ on types to be the congruence relation generated by $B \lhd \mathtt{raw}_k$ for all B and k with $\mathtt{size}(B) \leq k$. Whenever $B_1 \lhd B_2$, then we can cast a variable x of type B_1 into one of type B_2. We write $x\ \mathtt{as}\ B_2$ for an expression computing this value. A value of type B_2 can be cast back into one of type B_1. We also write $x\ \mathtt{as}\ B_1$ for such an expression. If $B_1 \lhd B_2$ and $x : B_1$, then we know that $(x\ \mathtt{as}\ B_2)\ \mathtt{as}\ B_1$ returns just x.

Interfaces. As is usual in systems-level programming, we consider programs that are incomplete in the sense that they contain calls to external functions. An *interface* $(I; O)$ for a program consists of two sets I and O of function signatures. The set I contains the functions signatures $f : (B_1, \ldots, B_n) \to B$ of all functions defined in the program. Any function that is called but not defined in the program must appear with an appropriate type in O. The set O is allowed to contain more definitions.

4 Module System

We define a module system for core expressions that elaborates into systems-level programs. Elaboration works by mapping modules to the structure of the GOI, which is then implemented by system programs. The module system is intentionally kept fairly standard in order to express the GOI in terms that are familiar to anyone familiar with ML.

The module system has the following types.

$$\begin{array}{rrcl}
\text{Paths} & p & ::= & X \mid p.\ell \\
\text{Base types} & C & ::= & \text{core type constructors} \mid p \\
\text{Module types} & \Sigma & ::= & \mathrm{M}C \mid \mathtt{type} \mid \mathtt{type}{=}C \mid \mathtt{sig}\ \overline{\ell_i(X_i) : \Sigma_i}\ \mathtt{end} \mid \mathtt{functor}(X : \Sigma) \to \Sigma \mid C \to \Sigma \mid B{\cdot}\Sigma
\end{array}$$

In paths, X ranges over an infinite supply of module variables. These variables are distinct from the value variables that may appear in core values. Base types are core types with an additional base case for paths, as usual, so that one can write types like $\mathtt{int} \times X.t$.

Most of the cases for module types are familiar from ML-like languages. In the module types, as in the rest of this paper, we use the notation $\overline{x_i}$ for a vector $x_1, \ldots, x_n$. Signatures therefore have the form $\mathtt{sig}\ \ell_1(X_1) : \Sigma_1, \ldots, \ell_n(X_n) : \Sigma_n\ \mathtt{end}$. In such a signature, the ℓ_i are *labels* for referring to the components from the outside using paths and the X_i are *identifiers* for referring to the components from within the signature. In a programming language, one would typically use write only labels, i.e. write $\mathtt{sig}\ \ell_1 : \Sigma_1, \ldots, \ell_n : \Sigma_n\ \mathtt{end}$. However, since labels may be used to access parts of the signature from the outside, they cannot be α-renamed, which causes problems with name capture in type-checking. For this reason, one introduces the additional identifiers, which can be α-renamed without harm. Signatures are an example of translucent sums [11].

Type declarations come in two forms: `type` and `type=`C. The former declares *some* base type, while the latter is a *manifest type* that is known to be the same as C. For example, one can write **`sig`** `t:` **`type`** `= int, f: Mt` **`end`**, which means that `t` is the type `int`. We shall allow ourselves to write **`type`** `t` and **`type`** `t =` **`int`** as syntactic sugar for `t:` **`type`** and `t:` **`type=int`**.

The type MC is a base case for computations. One should think of M(−) as the monad implicit in core expressions and of MC as a computation that returns a value of type C. We make the monad explicit to make clear where computational effects may happen. Note that the module system does not allow value declarations. This simplifies the development, as we would otherwise need to fix an evaluation order for modules to make sure that evaluation of values happens in the right order. Without value declarations, this is not necessary. All possible effects are accounted for by MC.

The module system nevertheless allows parameterisation over values by means of the type $C \to \Sigma$, which can only be applied to (already computed) values. The typical use of this type is in first-order function types $C_1 \to \mathrm{M}C_2$.

Finally, the type $B{\cdot}\Sigma$ corresponds to a slight generalisation of the exponentials from Linear Logic. The exponential $!\Sigma$ appears as the special case $\mathtt{raw}{\cdot}\Sigma$. The annotation states that the module has an index variable of type B, as outlined in Section 1.1. The programmer may consider $B{\cdot}\Sigma$ as having the same meaning as Σ; the annotation B will be computed automatically by type inference. It is nevertheless important to make it explicit in the types, since it needs to be known for linking.

Module terms are defined by the following grammar.

$$\begin{array}{lll} \text{Module terms} & M & ::= \; p \mid \mathtt{type}\ C \mid \mathtt{struct}\ \overline{\ell_i(X_i) = M_i}\ \mathtt{end} \mid \text{core expression constructors} \\ & & \quad \mid \mathtt{functor}(X : \Sigma) \to M \mid M\ X \mid \mathtt{fn}(x{:}C) \to M \mid M\ v \mid M :\!> \Sigma \end{array}$$

They can contain both value and module variables. Value variables x, y, z can appear in core values. They are bound by core expressions and by the new construct $\mathtt{fn}(x{:}C) \to M$ that allows abstraction over values. The corresponding application is $M\ v$. Module variables X, Y, Z may appear in paths. They are bound in structure and functor definitions.

Most module terms should be familiar from other module systems, particularly type declarations, signatures, functors and type sealing. For example, if M has type **`sig type`** `t =` **`int`**`, f: Mt` **`end`**, then sealing $M :\!> \Sigma$ allows one to abstract the signature to $\Sigma =$**`sig type`** `t, f: Mt` **`end`**.

What is perhaps less standard is that module terms are closed under the term formers for core expressions. Core expressions may not only be used for terms of type MC. One can use $\mathtt{let}\ (x, y) = v\ \mathtt{in}\ M$ and $\mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow M_1;\ \mathtt{inr}(y) \Rightarrow M_2$ for M, M_1 and M_2 of arbitrary module type. The case distinction is implemented by dynamic dispatch on v. Its typing rule requires M_1 and M_2 to have the same type. This is a simplification that makes type-checking easier. In the case where the types of M_1 and M_2 match, but are not equal, one must use sealing explicitly to make their types equal.

4.1 Examples

We give a few very simple examples to illustrate the features of the module system. The following signature `Stream` may be used as an interface for infinite streams of `int`s. The structure `Nats` implements the stream $0, 1, 2, \ldots$. We allow ourselves some syntactic sugar, e.g. for arithmetic expressions.

```
Stream := sig                          Nats := struct
            t: type,                             t = type int,
            init: Mt,                            init = return 0,
            next: t → M(int * t)                 next = fn(x: t) →
          end                                      return (x, x+1)
                                               end :> Stream
```

Without sealing, one could also give `Nats` the type with `t:` **`type=int`** instead of `t:` **`type`**.

An example of a functor is a module that multiplies a given stream with $1, -1, 1, -1, \ldots$

```
A := functor(X: Stream) →
       struct
         t = type (int * X.t),
         init = let x = X.init in return (1, x),
         next = fn (c: t) → let (s, x) = c in
```

```
                                let (i, x') = X.next(x) in
                                let c' = (s * (-1), x') in
                                return (s * i, c')
    end :> Stream
```

The following example shows that modules are not completely independent from core computation. It is possible to define a module by case-distinction, for example.

```
G := fn(b: unit+unit) → case b of inl() → A(Nats) | inr() → Nats
```

It has type `(unit + unit)→ Stream`. Computationally, a call to `G(v).next(x)` will first perform a case distinction on `v` and then dispatch to either of the two implementations of `next` from the two branches.

To give an example for the use of exponentials, consider again the higher-order functor from (4). It can be written as `functor(F: (unit + unit)·FS) →  struct` */ as before /* `end`. Its type is:

```
functor(F: (unit + unit)·FS) →
  sig
    A1: sig f: int → int end,
    A2: sig f: int → int end,
    h: int → int
  end
```

The exponential `(unit + unit)·FS` is essential because it tells the linker to insert an index argument of type `unit + unit` before application. The functor cannot be typed without it, as the parameter `F` is used twice in its body. We can see from the type that the functor `A` for alternating streams above does not require an index argument and can be applied using standard linking. This difference is the reason why it is essential to record exponentials explicitly in the types of modules.

5 Typing and Elaboration

We now define the module type system and the elaboration to systems-level programs. To manage all the details of type definitions, it is convenient to define elaboration using the F-ing modules approach of Rossberg, Russo and Dreyer [19]. This means that elaboration becomes a two-step process: an elaboration from modules into a linear variant of System F, followed by a GOI-translation into systems-level programs. The first step removes type declarations and replaces them by appropriately quantified type variables. The second step uses the constructions of the GOI for the systems-level implementation of modules.

5.1 Flexible Types for Interaction

We begin describing the linear variant of System F that captures the constructions of the GOI in a way that is suitable for elaboration into systems-level programs. It is a development of the calculus INT from [23].

The main new feature of the calculus in this section is the new form of contexts that allows a flexible management of the scope of value variables and of exponentials. This is important for the translation of a module calculus that allows a combination of paths and exponentials that is hard to treat efficiently using the standard rules from Linear Logic. The new formulation allows type inference to be simple and type-directed, even where exponentials are involved. One does not need to guess when to apply the rules for exponentials, but can simply deal with them eagerly. At the same time, it produces an efficient systems-level implementation of modules.

A technical contribution of this section is to develop an elaboration of the calculus that implements the informal idea of Section 1.1 with a systems-level language that has returning functions as target. In loc. cit. only non-returning functions were considered.

The calculus has the following types and terms.

Base types	D	$::=$	systems-level type constructors $\mid \alpha$
Interaction types	S, T	$::=$	$\mathrm{M}D \mid \{\overline{\ell_i : S_i}\} \mid S \multimap T \mid D \to S \mid \forall\alpha.\, S \mid \exists\alpha.\, S \mid D{\cdot}S$
Interaction terms	s, t	$::=$	core expression constructors $\mid \{\overline{\ell_i = t_i}\} \mid \mathsf{let}\ \{\overline{\ell_i = X_i}\} = s\ \mathsf{in}\ t$
			$\mid \lambda X : S.t \mid s\ t \mid \mathsf{fn}(x : D) \to t \mid t\ v \mid \Lambda\alpha.\, M \mid t\ D$
			$\mid \mathsf{pack}(D, t) \mid \mathsf{let}\ \mathsf{pack}(\alpha, X) = s\ \mathsf{in}\ t$

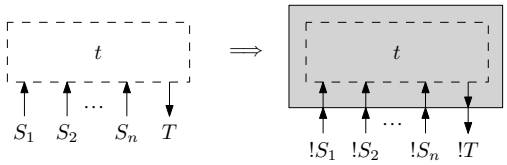
The grammar for core types is now extended with type variables, which can be bound by universal and existential quantifiers. Interaction types S and T correspond to the structure of the GOI. Instead of $S \otimes T$, we use a labelled record type $\{\ell_1 : S, \ell_2 : T\}$. This is convenient for the elaboration of structures later. As before, we write $!S$ for $\mathtt{raw} \cdot S$. We thus have $S \otimes T$, $S \multimap T$ and $!S$, as in the informal outline of the GOI in the Introduction. The types $\mathrm{M}D$ and $D \to S$ have the same role as in the module system above. The terms should be unsurprising to anyone familiar with System F.

In the rest of this section, we define the type system for this calculus and its elaboration into systems-level programs. We begin by explaining contexts and structural rules.

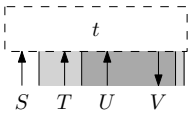
5.1.1 Contexts and Structural Rules

The type system uses a new kind of contexts and structural rules that allow for a simple and flexible management of exponentials. A context Γ is a finite list of variable declarations, of which there are three kinds: module declarations $X : S$, value declarations $x : D$ and type declarations α, as defined in Fig. 1 (ignore the parts after $\rightsquigarrow$ for now). We identify contexts up to the equivalence induced by $\Gamma, X : S, Y : T, \Delta = \Gamma, Y : T, X : S, \Delta$. This means that any two module declarations may be reordered. The order of value declarations is important, however. They may not be reordered, neither with module variable declarations nor with other value variable declarations. The order of type declarations is not important, but we do not need to reorder them and it is easiest to treat them like value variables. Value variable declarations thus partition contexts into zones.

The meaning of contexts is perhaps best explained using graphical notation like in Section 1.2. In standard type systems for Linear Logic, the promotion rule takes $\Gamma \vdash t : S$ to $!\Gamma \vdash t : !S$ (perhaps with digging in the context). In graphical notation, it amounts to putting the whole graph for t into a box, as shown below. Traditionally, the variables in the context Γ all live at the same nesting depth of boxes.



Our notion of contexts enables a flexible construction of such exponential boxes. Boxes can be constructed incrementally rather than all at once with promotion. Any value declaration in a context can be understood as the left border of an exponential box. All the following declarations are inside this box. The right border of the box appears at the end of the sequent. For example, a derivation of $X : S, x : \mathtt{raw}, Y : T, y : \mathtt{raw}, Z : U \vdash t : V$ represents the following situation (note that the boxes are not closed at the bottom).



One should think of each box as representing the scope of a value variable. Each box has an associated value that is available everywhere within the box and that never changes inside the box (it corresponds to the indices of the GOI). This value can be accessed with the value variable declared in the context. The variable x is available in the lightly shaded box, which contains the darker box. Variable y is available only in the dark box. All boxes are properly nested, as is standard in graphical representations of Linear Logic [10]. They are just not closed (yet) at the bottom.

The structural rules allow one to construct the boxes in a more gradual manner than the traditional promotion rule. The two rules for this purpose are CLOSE-L and CLOSE-R in Fig. 4. These rules are written with a double line, which means that they can be used both from bottom to top and from top to bottom. Used from top to bottom, they have the effect of (partially) closing a box. Rule CLOSE-L moves the border of the box past one of the variables, see the image on the left below. Since its edge now enters a box, its type must be decorated with an exponential. Rule CLOSE-R is applicable only in a context that ends with a value variable. The image on the right below depicts its effect on the premise $X : S, x : \mathtt{raw}, Y : T, y : \mathtt{raw} \vdash t : V$.

$$\frac{}{empty \rightsquigarrow (); \emptyset; \emptyset} \qquad \frac{\Gamma \rightsquigarrow \overline{B_i};\ I;\ O \qquad \Gamma \vdash S \rightsquigarrow J;\ P \qquad X \text{ not declared in } \Gamma}{\Gamma,\ X : S \rightsquigarrow \overline{B_i};\ I \cup J[X];\ O \cup P[X]}$$

$$\frac{\Gamma \rightsquigarrow \overline{B_i};\ I;\ O \qquad \Gamma \vdash D \rightsquigarrow B \qquad x \text{ not declared in } \Gamma}{\Gamma,\ x : D \rightsquigarrow \overline{B_i}, B;\ I;\ O} \qquad \frac{\Gamma \rightsquigarrow \overline{B_i};\ I;\ O \qquad \alpha \text{ not declared in } \Gamma}{\Gamma,\ \alpha \rightsquigarrow \overline{B_i};\ I;\ O}$$

Figure 1: Contexts: Well-Formedness and Elaboration

Using these two rules upside-down allows one to open boxes.

It is not hard to see that the standard promotion rule can be implemented with these new box-handling rules. The main advantage of the new rules is that they allow for a fully type-directed type inference method. One does not have to guess when to apply the promotion rule, but can just apply rule CLOSE-R eagerly whenever one encounters an exponential type.

The price to pay for the more flexible box-management rules is that context concatenation becomes a little more difficult. In multiplicative rules, we cannot just concatenate contexts, but we must combine them zone by zone, so that the level of all variables remains the same in the combined context. We define a partial operation of joining two contexts Γ and Δ into a single context $\Gamma + \Delta$ as follows:

$$(X : S,\ \Gamma_1) + \Gamma_2 := X : S,\ (\Gamma_1 + \Gamma_2) \qquad \Gamma_1 + (X : S, \Gamma_2) := X : S,\ (\Gamma_1 + \Gamma_2)$$
$$(x : A,\ \Gamma_1) + (x : A,\ \Gamma_2) := x : A,\ (\Gamma_1 + \Gamma_2) \qquad (\alpha,\ \Gamma_1) + (\alpha,\ \Gamma_2) := \alpha,\ (\Gamma_1 + \Gamma_2)$$

The use of the context operation + in the typing rules has the effect of treating module variables linearly (or multiplicatively) and all other variables non-linearly (or additively). Note that this operation is well-defined only by the identification up to reordering of module variable declarations. For example, $(X : \texttt{Mint},\ x : \texttt{int}) + (Y : \texttt{Mbool},\ x : \texttt{int})$ may be either $(X : \texttt{Mint},\ Y : \texttt{Mbool},\ x : \texttt{int})$ or $(Y : \texttt{Mbool},\ X : \texttt{Mint},\ x : \texttt{int})$, but we consider these contexts as being the same. The operation + is partial, since both contexts must contain the same value and type declarations in the same order.

5.1.2 Type System

As is standard in work on module systems, we formulate the type system so that the typing rules already contain the elaboration of types and terms to systems-level interfaces and programs in the part after the "$\rightsquigarrow$".

$$\Gamma \rightsquigarrow \overline{B_i};\ I;\ O \qquad \Gamma \vdash D \rightsquigarrow B \qquad \Gamma \vdash S \rightsquigarrow I;\ O \qquad \Gamma \vdash t :\ S \rightsquigarrow m$$

The first three judgements express well-formedness of contexts, base-types and interaction types, and the last one states that t has type S. Before coming to the elaboration, we briefly discuss the typing rules themselves. Figs. 1–3 are well-formedness rules for contexts and types. The structural rules in Fig. 4 are standard except for CLOSE-L, CLOSE-R and DUPLICATION, the former two we have already explained above. The latter allows unrestricted contraction on variables of certain type. These are the types that are implemented by systems-level programs that may not call any external function. With this rule, any first-order module can be typed without using any exponentials. The typing rules for terms appear in Figs. 5 and 6. With the new form of contexts, these should be unsurprising. We state them mainly for their elaboration part, so that the reader can follow the translation from modules to systems-level programs.

$$\frac{}{\Gamma,\, \alpha,\, \Delta \vdash \alpha \rightsquigarrow \mathtt{raw}} \qquad \frac{B \in \{\mathtt{empty}, \mathtt{unit}, \mathtt{int}, \mathtt{raw}_k\}}{\Gamma \vdash B \rightsquigarrow B} \qquad \frac{\Gamma \vdash D_1 \rightsquigarrow B_1 \qquad \Gamma \vdash D_2 \rightsquigarrow B_2 \qquad \mathtt{op} \in \{+, \times\}}{\Gamma \vdash D_1 \,\mathtt{op}\, D_2 \rightsquigarrow B_1 \,\mathtt{op}\, B_2}$$

Figure 2: Base Types: Well-Formedness and Elaboration

$$\frac{\Gamma \rightsquigarrow \overline{B_i};\ \ldots;\ \ldots \qquad \Gamma \vdash D \rightsquigarrow B}{\Gamma \vdash \mathrm{M}D \rightsquigarrow \{\square.\mathtt{main} : \overline{B_i} \to B\};\ \emptyset} \qquad \frac{\Gamma \vdash S_k \rightsquigarrow I_k;\ O_k \text{ for } k = 1, \ldots, n \qquad \overline{\ell_i} \text{ pairwise distinct}}{\Gamma \vdash \{\overline{\ell_i : S_i}\} \rightsquigarrow \bigcup_{i=1}^{n} I_i[\square.\ell_i];\ \bigcup_{i=1}^{n} O_i[\square.\ell_i]}$$

$$\frac{\Gamma \vdash S \rightsquigarrow I;\ O \qquad \Gamma \vdash T \rightsquigarrow J;\ P}{\Gamma \vdash S \multimap T \rightsquigarrow O[\square.\mathtt{arg}] \cup J[\square.\mathtt{res}];\ I[\square.\mathtt{arg}] \cup P[\square.\mathtt{res}]} \qquad \frac{\Gamma,\, x\colon D \vdash S \rightsquigarrow I;\ \ldots \qquad \Gamma \vdash S \rightsquigarrow \ldots;\ O}{\Gamma \vdash D \to S \rightsquigarrow I;\ O}$$

$$\frac{\Gamma,\, x\colon D \vdash S \rightsquigarrow I;\ O}{\Gamma \vdash D{\cdot}S \rightsquigarrow I;\ O} \qquad \frac{\Gamma,\, \alpha \vdash S \rightsquigarrow I;\ O}{\Gamma \vdash \forall\alpha.\, S \rightsquigarrow I;\ O} \qquad \frac{\Gamma,\, \alpha \vdash S \rightsquigarrow I;\ O}{\Gamma \vdash \exists\alpha.\, S \rightsquigarrow I;\ O}$$

Figure 3: Interaction Types: Well-Formedness and Elaboration

$$\textsc{close-l}\ \frac{\Gamma,\, x\colon D,\, X\colon S,\, \Delta \vdash t\colon T \rightsquigarrow m}{\Gamma,\, X\colon D{\cdot}S,\, x\colon D,\, \Delta \vdash t\colon T \rightsquigarrow m} \qquad \textsc{close-r}\ \frac{\Gamma,\, x\colon D \vdash t\colon S \rightsquigarrow m}{\Gamma \vdash t\colon D{\cdot}S \rightsquigarrow m}\ x \notin FV(t)$$

$$\textsc{weak}\ \frac{\Gamma \vdash t\colon T \rightsquigarrow m \qquad \Gamma \vdash S \rightsquigarrow I;\ O}{\Gamma,\, X\colon S \vdash t\colon T \rightsquigarrow m,\ \mathtt{fn}\, O(\overline{x})\ \{\, O(\overline{x})\,\}}$$

$$\textsc{dereliction}\ \frac{\Gamma,\, X\colon S,\, \Delta \vdash t\colon T \rightsquigarrow m \qquad \Gamma \vdash S \rightsquigarrow I;\ O}{\Gamma,\, X\colon D{\cdot}S,\, \Delta \vdash t\colon T \rightsquigarrow m[X \mapsto X'],\ \mathrm{der}_{I,O}(X, X')}\ B \text{ NON-EMPTY},\ X' \text{ FRESH}$$

$$\textsc{duplication}\ \frac{\Gamma,\, X\colon D{\cdot}S,\, \Delta \vdash t\colon T \rightsquigarrow m \qquad \Gamma \vdash S \rightsquigarrow I;\ \emptyset}{\Gamma,\, X\colon S,\, \Delta \vdash t\colon T \rightsquigarrow m[X \mapsto X'],\ \mathtt{fn}\, I[X'](\gamma, i, \overline{x})\ \{\, I[X](\gamma, \overline{x})\,\}}\ X' \text{ FRESH}$$

$$\textsc{contraction}\ \frac{\Gamma,\, X_1\colon D_1{\cdot}S,\, X_2\colon D_2{\cdot}S,\, \Delta \vdash t\colon T \rightsquigarrow m \qquad \Gamma \vdash S \rightsquigarrow I;\ O}{\Gamma,\, X\colon (D_1 + D_2){\cdot}S,\, \Delta \vdash t[X_1 \mapsto, X_2 \mapsto X]\colon T \rightsquigarrow m,\ \mathrm{contr}_{I,O}(X, X_1, X_2)}\ X \notin \{X_1, X_2\}$$

$$\textsc{digging}\ \frac{\Gamma,\, X\colon D_1{\cdot}(D_2{\cdot}S),\, \Delta \vdash t\colon T \rightsquigarrow m \qquad \Gamma \vdash S \rightsquigarrow I;\ O}{\Gamma,\, X\colon (D_1 \times D_2){\cdot}S,\, \Delta \vdash t\colon T \rightsquigarrow m[X \mapsto X'],\ \mathrm{dig}_{I,O}(X, X')}\ X' \text{ FRESH}$$

$$\textsc{subtyping}\ \frac{\Gamma,\, X\colon D_1{\cdot}S,\, \Delta \vdash t\colon T \rightsquigarrow m \qquad \Gamma \vdash D_2{\cdot}S \rightsquigarrow I;\ O \qquad \Gamma \vdash D_1{\cdot}S \rightsquigarrow J;\ P}{\Gamma,\, X\colon D_2{\cdot}S,\, \Delta \vdash t\colon T \rightsquigarrow m[X \mapsto X'],\ \mathrm{coerc}_{I,O,J,P}(X, X')}\ D_1 \lhd D_2$$

Figure 4: Structural Typing Rules

$$\text{VAR}\ \frac{\Gamma \vdash S \rightsquigarrow I;\ O \qquad \Gamma \rightsquigarrow \ldots;\ \ldots;\ O'}{\Gamma,\ X\colon S \vdash X\colon\ S \rightsquigarrow \text{forward}_{I,O}(X, \square),\ \mathtt{fn}\ O'(\overline{x})\ \{\ O'(\overline{x})\ \}}$$

$$\multimap\text{-I}\ \frac{\Gamma,\ X\colon S \vdash t\colon\ T \rightsquigarrow m}{\Gamma \vdash \lambda X\colon S.t\colon\ S \multimap T \rightsquigarrow m[\square \mapsto \square.\mathtt{res}, X \mapsto \square.\mathtt{arg}]}$$

$$\multimap\text{-E}\ \frac{\Gamma \vdash s\colon\ S \multimap T \rightsquigarrow m \qquad \Delta \vdash t\colon\ S \rightsquigarrow n}{\Gamma + \Delta \vdash s\ t\colon\ T \rightsquigarrow n[\square \mapsto \square.\mathtt{arg}],\ m,\ \text{forward}_{I,O}(\square.\mathtt{res}, \square)}$$

$$\otimes\text{-I}\ \frac{\Gamma_i \vdash t_i\colon\ S_i \rightsquigarrow m_i \qquad \overline{\ell_i}\ \text{pairwise distinct}}{\Gamma_1 + \cdots + \Gamma_n \vdash \{\overline{\ell_i = t_i}\}\colon\ \{\overline{\ell_i\colon S_i}\} \rightsquigarrow \overline{m_i[\square.\ell_i]}}$$

$$\otimes\text{-E}\ \frac{\Gamma \vdash s\colon\ \{\overline{\ell_i\colon S_i}\} \rightsquigarrow m \qquad \Delta,\ \overline{X_i\colon S_i} \vdash t\colon\ T \rightsquigarrow m'}{\Gamma + \Delta \vdash \mathtt{let}\ \{\overline{\ell_i = X_i}\} = s\ \mathtt{in}\ t\colon\ T \rightsquigarrow m[\overline{\square.\ell_i \mapsto X_i}],\ m'}$$

$$\text{FN-I}\ \frac{\Gamma,\ x\colon D \vdash t\colon\ S \rightsquigarrow m \qquad \Gamma \vdash S \rightsquigarrow I;\ O}{\Gamma \vdash \mathtt{fn}\,(x\colon D) \to t\colon\ D \to S \rightsquigarrow m[X],\ \mathtt{fn}\ I\,(\overline{x})\ \{\ I[X](\overline{x})\ \}, \mathtt{fn}\ O[X](\gamma, x, \overline{x})\ \{\ O[X](\gamma, \overline{x})\ \}}\ X\ \text{FRESH}$$

$$\text{FN-E}\ \frac{\Gamma \vdash t\colon\ D \to S \rightsquigarrow m \qquad \Gamma \vdash v\colon\ D \qquad \Gamma \vdash S \rightsquigarrow I;\ O}{\Gamma \vdash t\ v\colon\ S \rightsquigarrow m[X],\ \mathtt{fn}\ I\,(\gamma, \overline{x})\ \{\ I[X](\gamma, v, \overline{x})\ \},\ \mathtt{fn}\ O(\overline{x})\ \{\ O[X](\overline{x})\ \}}\ X\ \text{FRESH}$$

$$\text{ALL-I}\ \frac{\Gamma \vdash t\colon\ S \rightsquigarrow m \qquad \Gamma \vdash S \rightsquigarrow I;\ O}{\Gamma \vdash \Lambda\alpha.\, t\colon\ \forall\alpha.\, S \rightsquigarrow m}\ \alpha\ \text{NOT IN}\ \Gamma$$

$$\text{ALL-E}\ \frac{\Gamma \vdash t\colon\ \forall\alpha.\, S \rightsquigarrow m \qquad \Gamma \vdash S[\alpha \mapsto D] \rightsquigarrow I;\ O \qquad \Gamma,\ \alpha \vdash S \rightsquigarrow J;\ P}{\Gamma \vdash t\ D\colon\ S[\alpha \mapsto D] \rightsquigarrow m[X],\ \text{coerc}_{I,O,J,P}(X, \square)}\ X\ \text{FRESH}$$

$$\text{EXISTS-I}\ \frac{\Gamma \vdash t\colon\ S[X \mapsto D] \rightsquigarrow m \qquad \Gamma \vdash S[\alpha \mapsto D] \rightsquigarrow I;\ O \qquad \Gamma,\ \alpha \vdash S \rightsquigarrow J;\ P}{\Gamma \vdash \mathsf{pack}(D, t)\colon\ \exists\alpha.\, S \rightsquigarrow m[X],\ \text{coerc}_{I,O,J,P}(X, \square)}\ X\ \text{FRESH}$$

$$\text{EXISTS-E}\ \frac{\Gamma \vdash s\colon\ \exists\alpha.\, S \rightsquigarrow m \qquad \Gamma,\ \alpha,\ X\colon S \vdash t\colon\ T \rightsquigarrow n}{\Gamma,\ \alpha \vdash \mathtt{let}\ \mathsf{pack}(\alpha, X) = s\ \mathtt{in}\ t\colon\ T \rightsquigarrow m[X],\ n}\ X\ \text{FRESH}$$

Figure 5: Typing Rules for Interaction Terms

$$\text{RETURN}\ \frac{\Gamma \rightsquigarrow \ldots;\ \ldots;\ O \qquad \Gamma \vdash v\colon\ D}{\Gamma \vdash \mathtt{return}\ v\colon\ \mathrm{M}D \rightsquigarrow \mathtt{fn}\ \square.\mathtt{main}(\gamma)\ \{\ \mathtt{return}\ v\ \}, \mathtt{fn}\ O(\overline{x})\ \{\ O(\overline{x})\ \}}$$

$$\text{SEQ}\ \frac{\Gamma \vdash e_1\colon \mathrm{M}B \rightsquigarrow m \qquad \Delta,\ x\colon B \vdash e_2\colon \mathrm{M}C \rightsquigarrow n}{\Gamma + \Delta \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\colon \mathrm{M}C \rightsquigarrow m[X_1],\ n[X_2],\ \mathtt{fn}\ \square.\mathtt{main}(\gamma)\ \{\ \mathtt{let}\ x = X_1.\mathtt{main}(\gamma)\ \mathtt{in}\ X_2.\mathtt{main}(\gamma, x)\ \}}\ X_1, X_2\ \text{FRESH}$$

$$\text{PAIR-E}\ \frac{\Gamma \vdash v\colon\ A \times B \qquad \Gamma \vdash S \rightsquigarrow I;\ O \qquad \Gamma,\ x\colon A,\ y\colon B \vdash t\colon\ S \rightsquigarrow m}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ (x, y) = v\ \mathtt{in}\ t\colon\ S \rightsquigarrow m[X],\ \mathtt{fn}\ I\,(\gamma, \overline{x})\ \{\ \mathtt{let}\ (x, y) = v\ \mathtt{in}\ I[X](\gamma, x, \overline{x})\ \},\\ \qquad\qquad \mathtt{fn}\ O[X](\gamma, x, \overline{x})\ \{\ O(\gamma, \overline{x})\ \}\end{array}}\ X\ \text{FRESH}$$

$$\text{SUM-E}\ \frac{\begin{array}{l}v \vdash A_1 + A_2\colon \qquad \Gamma \vdash S \rightsquigarrow I;\ O \qquad \Gamma,\ x_i\colon A_i \vdash t_i\colon\ S \rightsquigarrow m_i\ \text{for}\ i \in \{1, 2\}\end{array}}{\begin{array}{l}\Gamma \vdash\ \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x_1) \Rightarrow t_1;\ \mathtt{inr}(x_2) \Rightarrow t_2\colon\ S\\ \qquad \rightsquigarrow\ m_1[X_1],\ m_1[X_1],\ \mathtt{fn}\ O[X_1](\gamma, x, \overline{x})\ \{\ O(\gamma, \overline{x})\ \},\ \mathtt{fn}\ O[X_2](\gamma, x, \overline{x})\ \{\ O(\gamma, \overline{x})\ \},\\ \qquad\quad \mathtt{fn}\ I\,(\gamma, \overline{x})\ \{\ \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x_1) \Rightarrow I[X_2](\gamma, x_1, \overline{x});\ \mathtt{inr}(x_2) \Rightarrow I[X_1](\gamma, x_2, \overline{x})\ \},\end{array}}\ X_1, X_2\ \text{FRESH}$$

Figure 6: Typing Rules for Expression Terms

Elaboration We now discuss the elaboration of the calculus into systems-level programs, i.e. the parts of the form $\rightsquigarrow \ldots$ in the typing judgements.

The rules for the base type elaboration judgement $\Gamma \vdash D \rightsquigarrow B$ obtain B from D essentially by replacing any occurrence of a type variable α with `raw`. Polymorphism is implemented by casting actual values to their raw underlying data.

The judgement $\Gamma \vdash S \rightsquigarrow I;\ O$ is defined in Fig. 3. It produces two sets I and O of function declarations, which are the systems-level interface of module of type S. The elaboration of types corresponds to the definitions of $(X \otimes Y)^-$, $(X \otimes Y)^+$, $(X \multimap Y)^-$ and $(X \multimap Y)^+$ from Section 1.2 as outlined in Section 1.3. The interface $(I;\ O)$ depends only on the value variables in Γ.

The function labels in I and O are words generated by the grammar $L ::= \square \mid X \mid L.\ell$, in which X ranges over module variables and where $\square$ represents a hole. The labels therefore are essentially paths considered as function labels. This is convenient to avoid arbitrary encodings into strings. To define the elaboration of modules, we often need to substitute labels for $\square$. We write short $(-)[L]$ for the substitution operation $(-)[\square \mapsto L]$.

The judgement $\Gamma \vdash S \rightsquigarrow I;\ O$ defines a systems-level interface in which all function labels have the prefix "$\square$.". This prefix is intended to be substituted with a base name. For instance, for a variable $X : S$ in a context we will use the interface $I[X];\ O[X]$.

We comment on the case for the type MD, because it shows the systems-level treatment of value variables. The premise $\Gamma \rightsquigarrow \overline{B_i};\ \ldots;\ \ldots$ in its rule states that the list of the types of value declarations in Γ elaborates to $\overline{B_i}$. Thus, MD elaborates to a single function declaration that takes the values of the value variables and returns a value of the type that D elaborates to. This means that on the systems-level, we make value variables available by passing them explicitly as additional arguments. This corresponds to the stack in the GOI. The management of values is implemented by the structural rules: When one enters a box, one must provide a new value for this variable as an argument to the called function. Inside the box, the variable is passed on unchanged. Upon leaving a box, its value is discarded.

The elaboration of $D{\cdot}S$ is also based on this way of managing value variables. The systems-level interface of $D{\cdot}S$ differs from that of S in that all function declarations have an additional argument of the type that D elaborates to. This is useful because we enforce the following *callee-save-invariant*: A systems-level program implementing the interface $D{\cdot}S$ must treat the additional variable like a variable from the context, i.e. it may not modify its value and must pass it on unchanged. We shall additionally enforce that the behaviour of the program may not even depend on the value of the additional variable. With these invariants, we can use the added value of type D like an index in Section 1.1.

The module elaboration judgement $\Gamma \vdash M : S \rightsquigarrow m$ translates the module term M to a systems-level program m. The program m defines all the functions in I and it may call the functions from O, where I and O are defined by $\Gamma \vdash S \rightsquigarrow I;\ O$. But, of course, m may also make use of the modules that are declared in Γ. So, if Γ is $\Delta,\ X : T,\ \Delta'$ and $\Delta \vdash T \rightsquigarrow J;\ P$, then m may assume that the module X is available as a systems-level program with interface $(J[X];\ P[X])$. This means that m may also invoke the functions from $J[X]$. In return, it must define all functions from $P[X]$.

In the elaboration part of rules for terms, we use some notation. Given any set I of definitions, we write short $\texttt{fn}\ I[X](\overline{x})\ \{\ I[Y](\overline{x})\ \}$ for the set $\left\{\texttt{fn}\ f[X](\overline{x})\ \{\ f[Y](\overline{x})\ \} \mid f \in I\right\}$. The length of the vector $\overline{x}$ is determined in each case by the type of f. In the rules, we use comma for set union. If Γ is a context, then we write γ for the list of value variables declared in it (in this order). We use the following abbreviations for sets of function definitions, which depend (through the use of γ) on the context Γ of the rule in which they are used.

- $\text{forward}_{I,O}(X, Y)$ abbreviates $\texttt{fn}\ I[X](\gamma, \overline{x})\{\ I[Y](\gamma, \overline{x})\ \},\ \texttt{fn}\ O[Y](\gamma, \overline{x})\{\ O[X](\gamma, \overline{x})\ \}$.
- $\text{dig}_{I,O}(X, Y)$ abbreviates

  $$\texttt{fn}\ I[X](\gamma, x, \overline{y})\{\ \texttt{let}\ (x_1, x_2){=}x\ \texttt{in}\ I[X](\gamma, x_1, x_2, \overline{y})\ \},$$
  $$\texttt{fn}\ O[Y](\gamma, x_1, x_2, \overline{y})\{\ O[X](\gamma, (x_1, x_2), \overline{y})\ \}.$$

- $\text{der}_{I,O}(X, Y)$ abbreviates $\texttt{fn}\ I[X](\gamma, \overline{y})\{\ I[Y](\gamma, v, \overline{y})\ \},\ \texttt{fn}\ O[Y](\gamma, x, \overline{y})\ \{\ O[X](\gamma, \overline{y})\ \}$.
- $\text{contr}_{I,O}(X, X_1, X_2)$ abbreviates

```
fn I[X1](γ, x1, ȳ){ I[X](γ, inl(x1), ȳ) }, fn I[X2](γ, x2, ȳ){ I[X](γ, inr(x2), ȳ) },
fn O[X](γ, x, ȳ){case x of inl(x1) → O[X1](γ, x1, ȳ)
                         | inr(x2) → O[X2](γ, x2, ȳ) }.
```

- $\mathrm{coerc}_{I,O,J,K}(X, Y)$ abbreviates the set of all functions defined as follows: For each f such that $f : \overline{A} \to C \in I$ and $f : \overline{B} \to C \in J$, there is the function

  ```
  fn f[Y](x̄ᵢ : Aᵢ){ let y1=x1 as B1 in ... let yn=xn as Bn in f[X](ȳᵢ) }.
  ```

  For each g such that $g : \overline{A} \to C \in P$ and $g : \overline{B} \to C \in O$, there is the function

  ```
  fn g[X](x̄ᵢ : Aᵢ){ let y1=x1 as B1 in ... let yn=xn as Bn in g[Y](ȳᵢ) }.
  ```

With these definitions, the structural rules should not be hard to understand. Perhaps rule WEAK deserves comment: It defines the functions from O trivially. Since no function from I is ever invoked, neither will be any function from O. But an implementation of the unused module X could contain a call to some external function in O, and we must define them, so that the program does not contain calls to undefined functions. Rules VAR and RETURN have similar definitions.

The rules for terms are such that the rules for records and $\multimap$-functions implement the informal implementation of signatures and functors outlined in the Introduction. Signatures just collect all the definitions with appropriate name-prefixes. Functors implement system-level program linking. The implementation of functors appears to be similar to that in MixML [18].

The rules for `fn`-functions bring an additional value variable into scope. On the level of systems-level program, this just amounts to adding a new formal parameter to all functions. The rules for `fn` refer to an obvious typing judgement for values $\Gamma \vdash v : D$, which is omitted.

In elaboration, the quantifiers become just coercions and have no essential effect at all. The type `raw` takes the place of the values of the concrete type. In particular, the packing and unpacking of existential types that will be frequent in the next section amounts to nothing but casts.

5.1.3 Correctness

We end this section by explaining in which sense elaboration provides a correct implementation of calculus. It is not completely obvious how to state correctness, as we have not given an operational semantics of the interaction calculus. Here we define set-theoretic semantics for the interaction calculus that interprets types as sets and both $\multimap$ and $\to$ simply as functions. This semantics ignores exponentials, which can be seen as an implementation detail that does not affect the meaning of programs. We then show that elaborated programs correctly implement this interpretation.

To define the set-theoretic interpretation, we assume a monad M on **Sets** that is sufficient to interpret the systems-level language. In the simplest case, this will be just the non-termination monad $MX = X + \{\bot\}$, which accounts for possible non-termination.

We assume a standard set-theoretic semantics for the systems-level language. The interpretation $[\![B]\!]$ of a systems-level type B is defined to be the set of closed values of type B. A systems-level function $f : (B_1, \ldots, B_n) \to B$ is interpreted as a function $[\![B_1]\!] \times \cdots \times [\![B_n]\!] \to M[\![B]\!]$. A systems-level program m is interpreted as $[\![m]\!]_\sigma$, where σ is an environment that maps function signatures like $f : (B_1, \ldots, B_n) \to B$ to corresponding functions $[\![B_1]\!] \times \cdots \times [\![B_n]\!] \to M[\![B]\!]$. The semantics of the program $[\![m]\!]_\sigma$ is then a mapping from function signatures (the ones defined in m) to corresponding functions (of the same format as in σ). We do not spell out this interpretation. It is essentially a standard semantics of an imperative language, see e.g. [24].

We use the following notion of systems-level program equality. We write $m =_{I,O} m'$ if, for any environment σ that defines all functions in O, the interpretations $[\![m]\!]_\sigma$ and $[\![m']\!]_\sigma$ agree on all functions from I.

With these definitions, we can define a set-theoretic semantics for the interaction calculus. For any

closed interaction type S, we define the set $[\![S]\!]$ as follows:

$$[\![\mathrm{M}D]\!] = M[\![D]\!] \qquad [\![B \cdot S]\!] = [\![S]\!]$$
$$[\![\{\overline{\ell_i : S_i}\}]\!] = \{f \mid \mathrm{dom}(f) = \{\overline{\ell_i}\},\ f(\ell_i) \in [\![S_i]\!]\} \qquad [\![\forall\alpha.\, S]\!] = \prod_{B \text{ systems-level type}} [\![S[\alpha \mapsto B]]\!]$$
$$[\![S \multimap T]\!] = [\![S]\!] \to [\![T]\!] \qquad [\![\exists\alpha.\, S]\!] = \sum_{B \text{ systems-level type}} [\![S[\alpha \mapsto B]]\!]$$
$$[\![D \to S]\!] = [\![D]\!] \to [\![S]\!]$$

We omit the interpretations of terms.

Elaboration correctly implements this set-theoretic semantics. To make this statement precise, we define a relation $m \sim_S f$, which expresses that the systems-level program m implements the semantic value $f \in [\![S]\!]$. The relation is defined such that it can hold only if m is a systems-level program of the interface $(I;\ O)$ determined by $\vdash S \rightsquigarrow I;\ O$. The definition is given by induction on S in the following cases. To simplify the notation, we write m_R for the systems-level program in the conclusion of rule R. For example, $m_{\otimes\text{-I}}$ is $\overline{m_i[\square.\ell_i]}$.

- $m \sim_{\mathrm{M}D} f$ iff: $[\![m]\!](\square.\mathtt{main})() = f$.
- $m \sim_{\overline{\{\ell_i : S_i\}}} f$ iff: there exist $\overline{m_i}$ and $\overline{f_i}$ such that $m =_{I,O} m_{\otimes\text{I}}$ and $\overline{f(\ell_i) = f_i}$ and $\overline{m_i \sim_{S_i} f(\ell_i)}$.
- $m \sim_{S \multimap T} f$ iff: whenever $n \sim_S g$, then $m_{\multimap\text{-E}} \sim_T f(g)$.
- $m \sim_{D \to S} f$ iff: for all $d \in [\![D]\!]$, we have $m_{\text{FN-E}} \sim_T f(d)$ for some/any fresh X.
- $m \sim_{B \cdot S} f$ iff: there exists n with with $n \sim_S f$ and $m =_{I,O} B \cdot n$, where $B \cdot n$ denotes the program obtained from n by adding a new first formal parameter of type B to all functions, which is passed as first argument in all function calls.
- $m \sim_{\forall\alpha.\, S} f$ iff: we have $m_{\text{ALL-E}} \sim_{S[\alpha \mapsto B]} f(B)$ for all B, where J and P in $m_{\text{ALL-E}}$ are determined by $\vdash S[\alpha \mapsto B] \rightsquigarrow J;\ P$.
- $m \sim_{\exists\alpha.\, S} f$ iff: there exist B, n, g, such that f is (B, g) and $m =_{I,O} m_{\text{EXISTS-I}}$ and $n \sim_{S[\alpha \mapsto B]} g$.

With these definitions, we can state the correctness result:

Proposition 1. *If $\vdash t :\ S \rightsquigarrow m$, then $m \sim_S [\![t]\!]$.*

The proof is a straightforward, if lengthy, induction on typing derivations. To deal with typing sequents with non-empty contexts, it uses a logical extension of $\sim$ to terms in context, which amounts to abstracting the whole context and using closed $\sim$.

5.2 Elaboration of the Module System

We now translate the module system into the interaction calculus. Records and $\multimap$-functions can already directly represent structures and functors. It remains to account for type declarations, their abstraction and the use of paths to access them.

F-ing [19] translates an ML-style module system into System F_ω. Here we adapt it to translate our module system into the interaction calculus. Module types translate to interaction types and module terms translate to interaction terms. In essence, structures translate to records, functors translate to $\multimap$-functions, and any type declaration `type` or `type=`D in a module type is replaced by the unit type $\{\}$ (the empty record). As unit types elaborate to an empty systems-level interface, this means that type declarations are compiled out completely and are only relevant for type checking. The declared types become type variables and are thus implemented using `raw`.

While one wants to remove type declarations in the elaboration process, type information is needed for type checking, of course. In order to be able to express elaboration and type-checking in one step, it is useful to use labelled unit types that still record the erased type information. We define the type $[{=}D]$ as a copy of the unit type $\{\}$, labelled with D. This type could be made a primitive type, but it can also be defined as the type $[{=}D] := D \to \{\}$ with inhabitant $\star_D := \mathtt{fn}(x : D) \to \{\}$. Note that $[{=}D]$ still elaborates to

an empty systems-level interface. The labelling can now be used to track correct usage of types: `type`$=D$ becomes $[=D]$ and `type` becomes $[=\alpha]$ for a new, existentially quantified, type variable α. For example, the signature **sig** `s: type, t: type, f: Mt, g: Ms` **end** becomes $\exists\alpha,\beta.\{\mathtt{s}\colon[=\alpha],\ \mathtt{t}\colon[=\beta],\ \mathtt{f}\colon \mathrm{M}\beta,\ \mathtt{g}\colon \mathrm{M}\alpha\}$. The elaborated type still contains the information that `f` returns a value of type `t`, which would have been lost had we used $\{\}$ instead of $[=\alpha]$. Elaborated types thus contain all information that is needed for type-checking.

In the rest of this section, we define an F-ing translation of our module system to the linear type system for the Geometry of Interaction. The definitions follow [19]; the main difference is the treatment of linearity in module and interaction types. The translation uses five kinds of judgements for base types, module types, module terms, module subtyping and matching:

$$\Gamma \vdash C \rightsquigarrow D \qquad \Gamma \vdash \Sigma \rightsquigarrow \Xi \qquad \Gamma \vdash M : \Xi \rightsquigarrow t \qquad \Gamma \vdash \Xi \leq \Xi' \rightsquigarrow t \qquad \Gamma \vdash S \leq \Xi' \rightsquigarrow \overline{D}, t$$

In these judgements, S and Ξ are interaction types defined by the following grammar.

$$\begin{array}{lcl} S & ::= & [=D] \mid \mathrm{M}D \mid \{\overline{\ell_i : S}\} \mid \forall\overline{\alpha}.\, S \multimap \Xi \mid D \to S \mid D{\cdot}S \\ \Xi & ::= & \exists\overline{\alpha}.\, S \end{array}$$

The context Γ in the judgements is a context as in the previous section. However, all module variable declarations in it must have the form $X : S$, where S is generated by the above grammar.

The judgement $\Gamma \vdash \Sigma \rightsquigarrow \Xi$ formalises module type elaboration. For example, if Σ is the signature **sig** `t: type, f: Mt` **end**, then Ξ will be $\exists\alpha.\{\mathtt{t}\colon[=\alpha],\ \mathtt{f}\colon \mathrm{M}\alpha\}$.

The judgement $\Gamma \vdash M : \Xi \rightsquigarrow t$ expresses that M is a module term whose type elaborates to Ξ and that the module itself elaborates to the interaction term t with $\Gamma \vdash t : \Xi$.

The judgement $\Gamma \vdash \Xi \leq \Xi' \rightsquigarrow t$ is for subtyping. In it, t is a coercion term from Ξ to Ξ' that satisfies $\Gamma,\ X : \Xi \vdash t\ X : \Xi'$.

Finally, $\Gamma \vdash S \leq \Xi \rightsquigarrow \overline{D}, t$ is a matching judgement. By definition, Ξ has the form $\exists\overline{\alpha}.\, S'$. The matching judgement produces a list of types $\overline{D}$ and a term t, such that $\Gamma \vdash S \leq S'[\overline{\alpha} \mapsto \overline{D}] \rightsquigarrow t$.

In all judgements, the context records the *already elaborated* type of variables. Labelled unit types record enough information to perform type checking with elaborated types only.

Module type elaboration in Fig. 8 implements the idea of translating structures to records, functors to $\multimap$-functions and to replace type declarations by labelled unit types. The variable case in the translation of base types in Fig. 7 gives a first example of where one uses the label of a unit type. Functors are modelled generatively. If the argument elaborates to $\exists\alpha.\, S$ and the result to $\exists\beta.\, T$, then the functor elaborates to $\forall\alpha.\, S \multimap \exists\beta.\, T$. This means that the type β may be different for each application of the functor. To cover existing applications, such as [23], generative functors were a natural choice (indeed, types of the form $\forall\alpha.\, S \multimap \exists\beta$ already appear in loc. cit.); in the future, applicative functors may also be useful.

The elaboration rules for terms are shown in Figs. 9 and 10. In addition to these rules, we also assume that the structural rules from Fig. 4 are available in the rules for module elaboration.

The module system is linear in the sense that a module may be used more than once only if it appears under a suitable exponential. We make exponentials explicit in the types, because they add a little overhead in the systems-level implementation of modules, and it is desirable to have control over them. Already in the Introduction we have seen that non-linear use of module variables has an effect on the elaboration of modules. In the higher-order example in Section 1.1, the variable `F` was used twice, once in `F(A1)` and once in `F(A2)`, and this duplication led to the need to insert an index parameter in the elaboration. This additional parameter introduces a little overhead in the systems-level program. Had we used `F` only once, then we could have done without it. In the type system, this difference appears in the presence or absence of an exponential on the type.

Defining what it means for a module to be used more than once is a little subtle. Suppose we have a variable X of the (already elaborated) type $\{\ell_1 : S,\ \ell_2 : T\}$. A module that uses both $X.\ell_1$ and $X.\ell_2$ should be considered as using X linearly, even though the variable X appears twice. As each component of the module is used only once, elaboration works without the need for an exponential. A module like `struct` $A = X.\ell,\ B = X$ `end` that uses both $X.\ell_1$ and X cannot be considered as using X linearly, however. An exponential is needed.

$$\frac{}{\Gamma,\, X : [=\alpha],\, \Delta \vdash X \rightsquigarrow \alpha} \qquad \frac{B \in \{\mathtt{empty}, \mathtt{unit}, \mathtt{int}\}}{\Gamma \vdash B \rightsquigarrow B} \qquad \frac{\Gamma \vdash C_1 \rightsquigarrow D_1 \qquad \Gamma \vdash C_2 \rightsquigarrow D_2 \qquad \mathsf{op} \in \{+, \times\}}{\Gamma \vdash C_1 \,\mathsf{op}\, C_2 \rightsquigarrow D_1 \,\mathsf{op}\, D_2}$$

Figure 7: Base Type Elaboration

$$\frac{\Gamma \vdash C \rightsquigarrow D}{\Gamma \vdash \mathrm{M}C \rightsquigarrow \mathrm{M}D} \qquad \frac{}{\Gamma \vdash \mathtt{type} \rightsquigarrow \exists\alpha.\, [=\alpha]} \qquad \frac{\Gamma \vdash C \rightsquigarrow D}{\Gamma \vdash \mathtt{type}{=}C \rightsquigarrow [=D]} \qquad \frac{}{\Gamma \vdash \mathtt{sig\ end} \rightsquigarrow \{\}}$$

$$\frac{\Gamma \vdash \Sigma \rightsquigarrow \exists\overline{\alpha}.\, S \qquad \Gamma,\, \overline{\alpha},\, X : X \vdash \mathtt{sig}\ \overline{D}\ \mathtt{end} \rightsquigarrow \exists\overline{\beta}.\, \{\overline{E}\}}{\Gamma \vdash \mathtt{sig}\ \ell(X) : \Sigma,\, \overline{D}\ \mathtt{end} \rightsquigarrow \exists\overline{\alpha}, \overline{\beta}.\, \{\ell : S,\, \overline{E}\}} \qquad \frac{\Gamma \vdash \Sigma \rightsquigarrow \exists\overline{\alpha}.\, S \qquad \Gamma \vdash \mathrm{T} \rightsquigarrow \exists\overline{\beta}.\, T}{\Gamma \vdash \mathtt{functor}(X :\ \Sigma) \to \mathrm{T} \rightsquigarrow \forall\overline{\alpha}.\, S \multimap \exists\overline{\beta}.\, T}$$

$$\frac{\Gamma,\, x : B \vdash \Sigma \rightsquigarrow \exists\overline{\alpha}.\, S \qquad \Gamma \vdash C \rightsquigarrow B}{\Gamma \vdash C \to \Sigma \rightsquigarrow \exists\overline{\alpha}.\, B \to S} \qquad \frac{\Gamma \vdash \Sigma \rightsquigarrow \exists\overline{\alpha}.\, S}{\Gamma \vdash B{\cdot}\Sigma \rightsquigarrow \exists\overline{\alpha}.\, B{\cdot}S}$$

Figure 8: Module Type Elaboration

To capture a suitable notion of linearity, the elaboration procedure manages paths differently from other approaches [19, 15]. In rule VAR, the base case of term elaboration is defined only for variables, not for arbitrary paths. However, rule SIG-E allows one to reduce paths in a term. For example, to derive $X : \{f : S,\, g : T\} \vdash X.f :\ S$, one can first use SIG-E to reduce the goal to $f : S,\, g : T \vdash f :\ S$. The advantage of this approach using SIG-E over standard treatments of paths is that it allows us to control linear use of modules, even in cases where a module variable syntactically appears twice in a term. For example, if $X : \{f : S,\, g : T\}$ then one can give a type to a module of the form $\mathtt{struct}\ A = X.\ell_1,\, B = X.\ell_2\ \mathtt{end}$, but not (in general) one of the form $\mathtt{struct}\ A = X.\ell,\, B = X\ \mathtt{end}$. The fact that SIG-E removes variable X from the context is good, because is disallows the latter case where one uses both $X.\ell$ and X alone (which may lead to an indirect second use of $X.\ell$ via B later). If one needs X again, then one needs to use contraction beforehand.

The variable rule is further unusual in that it requires the variable to be declared last in the context. Recall that contexts do not allow arbitrary reordering of module variables. To type a variable in a larger context, one must use rules WEAK and CLOSE-R to reduce the context before applying VAR. The following example illustrates this:

$$\text{WEAK}\ \frac{\text{CLOSE-R}\ \dfrac{\text{VAR}\ \dfrac{}{\Gamma,\, X : (D{\cdot}S) \vdash X :\ D{\cdot}S}}{\Gamma,\, X : (D{\cdot}S),\, x : D \vdash X :\ S}}{\Gamma,\, X : (D{\cdot}S),\, x : D,\, Y : T \vdash X :\ S}$$

The example also shows that applying the variable rule in a context containing a value variable is possible only if the variable has a suitable exponential.

Finally, the rules for subtyping and matching appear in Figure 11. From a technical point of view, they are very similar to the rules in [19]. However, since type variables only range over base types here, they are actually much simpler to work with. Subtyping can be decided easily, for example.

Proposition 2. *If* $\Gamma \vdash M :\ \Xi \rightsquigarrow m$, *then* $\Gamma \vdash m :\ \Xi$ *in the interaction type system.*

Together with the set-theoretic semantics for interaction types from the previous section, the elaboration can be seen as the definition of a semantics for the module system. Prop. 1 states that this semantics is implemented correctly.

6 Type Checking

For practical applications, we want to implement module type checking and inference. Compared to [19], the main difficulty with our module system is that the elaboration rules are not obviously syntax-directed, mainly because the structural rules from Fig. 4 can be applied in many ways. Indeed, there are many

$$\textsc{var}\ \frac{}{\Gamma,\, X : S \vdash X :\ S \rightsquigarrow X} \qquad \textsc{type}\ \frac{\Gamma \vdash C \rightsquigarrow D}{\Gamma \vdash \mathtt{type}\ C :\ [{=}D] \rightsquigarrow \star_D} \qquad \textsc{sig-i1}\ \frac{}{\Gamma \vdash \mathtt{struct\ end} :\ \{\} \rightsquigarrow \{\}}$$

$$\textsc{sig-i2}\ \frac{\Gamma \vdash M :\ \exists \overline{\alpha}.\, S \rightsquigarrow t \qquad \Delta,\, \overline{\alpha},\, X : S \vdash \mathtt{struct}\ D\ \mathtt{end} :\ \exists \overline{\beta}.\, \{\overline{E}\} \rightsquigarrow s \qquad \ell \text{ not defined in } D}{\begin{array}{l}\Gamma + \Delta \vdash\ \mathtt{struct}\ \ell(X) = M,\ D\ \mathtt{end} :\ \exists \overline{\alpha}, \overline{\beta}.\, \{\ell : S,\ \overline{E}\} \\ \qquad \rightsquigarrow \mathtt{let\ pack}(\overline{\alpha}, x) = t\ \mathtt{in\ let\ pack}(\overline{\beta}, \{\overline{b}\}) = s\ \mathtt{in\ pack}(\overline{\alpha}\overline{\beta}, \{\ell = x, \overline{b}\})\end{array}}$$

$$\textsc{sig-e}\ \frac{\Gamma,\, \overline{Y_i : S_i},\, \Delta \vdash M : \Xi \rightsquigarrow t}{\Gamma,\, X : \{\overline{\ell_i : S_i}\},\, \Delta \vdash M[\overline{Y_i \mapsto X.\ell_i}] : \Xi \rightsquigarrow \mathtt{let}\ \{\overline{\ell_i = Y_i}\} = X\ \mathtt{in}\ t}$$

$$\textsc{functor-i}\ \frac{\Gamma \vdash \Sigma \rightsquigarrow \exists \overline{\alpha}.\, S \qquad \Gamma,\, \overline{\alpha},\, X : S \vdash M : \Xi \rightsquigarrow t}{\Gamma \vdash \mathtt{functor}(X :\ \Sigma) \to M :\ \forall \overline{\alpha}.\, S \multimap \Xi \rightsquigarrow \lambda \overline{\alpha}.\, \lambda X : S.\, t}$$

$$\textsc{functor-e}\ \frac{\Gamma \vdash M :\ \forall \overline{\alpha}.\, S \multimap \Xi \rightsquigarrow t \qquad \Delta \vdash Y :\ S' \rightsquigarrow s \qquad \Gamma + \Delta \vdash S' \leq \exists \overline{\alpha}.\, S \rightsquigarrow \overline{D}, f}{\Gamma + \Delta \vdash M\ Y :\ \Xi[\overline{\alpha \mapsto D}] \rightsquigarrow t\ \overline{D}\ (f\ s)}$$

$$\textsc{fn-i}\ \frac{\Gamma,\, x : D \vdash M : \Xi \rightsquigarrow t \qquad \Gamma \vdash C \rightsquigarrow D}{\Gamma \vdash \mathtt{fn}\,(x : C) \to M :\ D \to \Xi \rightsquigarrow \mathtt{fn}\,(x : D) \to t} \qquad \textsc{fn-e}\ \frac{\Gamma \vdash M :\ D \to \Xi \rightsquigarrow t \qquad \Gamma \vdash v :\ D}{\Gamma \vdash M\ v : \Xi \rightsquigarrow t\ v}$$

$$\textsc{seal}\ \frac{\Gamma \vdash M : \Xi \rightsquigarrow t \qquad \Gamma \vdash \Sigma \rightsquigarrow \Xi' \qquad \Gamma \vdash \Xi \leq \Xi' \rightsquigarrow c}{\Gamma \vdash M :> \Sigma : \Xi' \rightsquigarrow c\ t}$$

Figure 9: Module Term Elaboration

$$\textsc{return}\ \frac{\Gamma \vdash v :\ D}{\Gamma \vdash \mathtt{return}\ v :\ \mathrm{M}D \rightsquigarrow \mathtt{return}\ v} \qquad \textsc{seq}\ \frac{\Gamma \vdash e_1 :\ \mathrm{M}D_1 \rightsquigarrow t \qquad \Delta,\, x : D_1 \vdash e_2 :\ \mathrm{M}D_2 \rightsquigarrow s}{\Gamma + \Delta \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :\ \mathrm{M}D_2 \rightsquigarrow \mathtt{let}\ x = t\ \mathtt{in}\ s}$$

$$\textsc{pair-e}\ \frac{\Gamma \vdash v :\ D_1 \times D_2 \qquad \Gamma,\, x : D_1,\, y : D_2 \vdash M : \Xi \rightsquigarrow t}{\Gamma \vdash \mathtt{let}\ (x, y) = v\ \mathtt{in}\ M : \Xi \rightsquigarrow \mathtt{let}\ (x, y) = v\ \mathtt{in}\ t}$$

$$\textsc{sum-e}\ \frac{\Gamma \vdash v :\ D_1 + D_2 \qquad \Gamma,\, x_i : D_i \vdash M_i : \Xi \rightsquigarrow t_i \qquad \text{for } i \in \{1, 2\}}{\Gamma \vdash \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x_1) \Rightarrow M_1;\ \mathtt{inr}(x_2) \Rightarrow M_2 : \Xi \rightsquigarrow \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x_1) \Rightarrow t_1;\ \mathtt{inr}(x_2) \Rightarrow t_2}$$

Figure 10: Module Expression Elaboration

$$\frac{}{\Gamma \vdash [{=}D] \leq [{=}D] \rightsquigarrow \lambda x.\, x} \qquad \frac{}{\Gamma \vdash \mathrm{M}D \leq \mathrm{M}D \rightsquigarrow \lambda x.\, x} \qquad \frac{\Gamma,\, x : D \vdash S \leq T \rightsquigarrow c}{\Gamma \vdash D \to S \leq D \to T \rightsquigarrow \mathtt{fn}\,(x : D) \to c\ x}$$

$$\frac{\Gamma,\, \overline{\beta} \vdash S_2 \leq \exists \overline{\alpha}.\, S_1 \rightsquigarrow \overline{D}, c \qquad \Gamma,\, \overline{\beta} \vdash \Xi_1[\overline{\alpha \mapsto D}] \leq \Xi_2 \rightsquigarrow d}{\Gamma \vdash (\forall \overline{\alpha}.\, S_1 \multimap \Xi_1) \leq (\forall \overline{\beta}.\, S_2 \multimap \Xi_2) \rightsquigarrow \lambda f.\, \Lambda \overline{\beta}.\, \lambda x.\, d\ (f\ \overline{D}\ (c\ x))} \qquad \frac{\Gamma,\, x : D \vdash S \leq T \rightsquigarrow c}{\Gamma \vdash D \cdot S \leq D \cdot T \rightsquigarrow c}$$

$$\frac{}{\Gamma \vdash \{\} \leq \{\} \rightsquigarrow \lambda x.\, x} \qquad \frac{\Gamma \vdash \{\overline{E}\} \leq \{\overline{F}\} \rightsquigarrow c}{\Gamma \vdash \{\ell : S,\, \overline{E}\} \leq \{\overline{F}\} \rightsquigarrow \lambda x.\, \mathtt{let}\ \{\ell = y,\, \overline{b}\} = x\ \mathtt{in}\ c\ \{\overline{b}\}}$$

$$\frac{\Gamma \vdash S \leq T \rightsquigarrow c \qquad \Gamma \vdash \{\overline{E}\} \leq \{\overline{F}\} \rightsquigarrow d}{\Gamma \vdash \{\ell : S,\, \overline{E}\} \leq \{\ell : T,\, E\} \rightsquigarrow \lambda x.\, \mathtt{let}\ \{\ell = y,\, \overline{b}\} = x\ \mathtt{in\ let}\ \{\overline{b_1}\} = d\ \{\overline{b}\}\ \mathtt{in}\ \{\ell = (c\ y),\, \overline{b_1}\}}$$

$$\frac{\Gamma \vdash S \leq T[\overline{\alpha \mapsto D}] \rightsquigarrow c}{\Gamma \vdash S \leq \exists \overline{\alpha}.\, T \rightsquigarrow \overline{D}, c} \qquad \frac{\Gamma,\, \overline{\alpha} \vdash S \leq \Xi \rightsquigarrow \overline{D}, c}{\Gamma \vdash \exists \overline{\alpha}.\, S \leq \Xi \rightsquigarrow \lambda x.\, \mathtt{let\ pack}(\overline{\alpha}, y) = x\ \mathtt{in\ pack}(\overline{D}, c\ x)}$$

Figure 11: Module Subtyping and Matching

ways to deal with the exponentials and we want to construct a typing derivation that elaborates modules to efficient systems-level programs. For example, suppose we have a module term that contains a module variable X such that $X.\ell_1$ is used once and $X.\ell_2$ is used twice. This is possible in a situation where $X : !\{\ell_1 : \Sigma_1, \ell_2 : \Sigma_2\}$ is in the context. However, the placement of the exponential is not ideal. It would be better to use $X : \{\ell_1 : \Sigma_1, \ell_2 : !\Sigma_2\}$. In the former case, the whole module X would get an additional index parameter in elaboration, while in the second case only the ℓ_2-part gets one. Furthermore, even when exponentials are unavoidable, $(\mathtt{unit} + \mathtt{unit})\cdot S$ is preferable to $!S$, as a value of type $\mathtt{unit} + \mathtt{unit}$ is likely to be compiled more efficiently than one of type `raw` at the systems level.

One problem is to decide where to place exponentials. But notice that exponentials of the form $D\cdot S$ provide more fine-grained control over duplication than just $!S$. In particular, $\mathtt{unit}\cdot S$ is essentially as good as S. After elaboration, these two types differ only in the presence of an additional argument of type `unit`, which we can expect to be optimised away later. We therefore restrict our attention to *normal form types*, which are defined by the following grammar.

$$\begin{array}{lcl} S & ::= & [=D] \mid \mathrm{M}D \mid D\cdot\{\overline{\ell_i : S}\} \mid D\cdot(\forall\overline{\alpha}.\, S \multimap \Xi) \mid D\cdot(D \to S) \\ \Xi & ::= & \exists\overline{\alpha}.\, S \end{array}$$

The cases for S that do not appear under a bang are types where DUPLICATION is applicable.

The new structural rules allow us to deal with the exponentials in a syntax-directed manner. For normal form types, the placement of exponentials is determined by the types. For example, a term of the form $\mathtt{functor}(X : \Sigma_1) \to \Sigma_2$ must have a type of the form $D\cdot(\forall\overline{\alpha}.\, S \multimap \Xi)$. To construct a typing derivation for a term of this type, we can always just start by applying rule CLOSE-R first. The applications of other structural rules on the left can be deferred, e.g. to until we get to the leaves. These uses are captured by the following lemma.

Lemma 1. *Suppose Δ is a context that declares value variables of the types $D_1, \ldots, D_n$. Then the following rules are admissible.*

$$\frac{D_1 \times \cdots \times D_n \lhd D}{\Gamma,\, X : (D\cdot S),\, \Delta \vdash X : S \rightsquigarrow \ldots} \qquad \frac{\Gamma,\, X_1 : (D'\cdot S),\, X_2 : (D''\cdot S),\, \Delta \vdash M : \Xi \rightsquigarrow \ldots \qquad D' + D'' \lhd D}{\Gamma,\, X : (D\cdot S),\, \Delta \vdash M[X_1 \mapsto X, X_2 \mapsto X] : \Xi \rightsquigarrow \ldots}$$

Proof Sketch. To derive the rule for variables, one first (starting from the conclusion) removes the context Δ using rules WEAK and CLOSE-R. Each application of CLOSE-R adds an exponential $D_i\cdot(-)$ to the type after the turnstile. After removing Δ one thus has an exponential for each value variable in Δ. Using DIGGING, SUBTYPING and VAR, one can then complete the derivation.

The following example illustrates the approach (omitting the elaboration-part):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\, X : D_1\cdot(D_2\cdot S) \vdash X : D_1\cdot(D_2\cdot S)}\,\textsc{var}}{\Gamma,\, X : D\cdot S \vdash X : D_1\cdot(D_2\cdot S)}\,\textsc{digging,subtyping}}{\Gamma,\, X : D\cdot S,\, x : D_1 \vdash X : D_2\cdot S}\,\textsc{close-r}}{\Gamma,\, X : D\cdot S,\, x : D_1,\, Y : T \vdash X : D_2\cdot S}\,\textsc{weak}}{\Gamma,\, X : D\cdot S,\, x : D_1,\, Y : T,\, y : D_2 \vdash X : S}\,\textsc{close}$$

□

Notice that one can always take `raw` for D in the lemma.

The task of type checking a module term M is now to construct a derivation that gives M a normal form type. In general, there will be many possible choices for the exponentials in the type of M. In order to explain the construction of the typing derivation, we first consider the trivial choice, where all exponentials are `raw`. This will be the basis for a better choice of exponentials below. We use the term *copyable types* for normal form types in which all exponentials are `raw`.

For normal form types, typing is essentially syntax directed. If one considers only copyable types, then the types are determined by the terms:

Lemma 2. *For any well-formed context Γ (see Fig. 1), we have:*

- *For any C, there exists at most one D such that $\Gamma \vdash C \rightsquigarrow D$.*
- *For any Σ, there exists at most one S such that $\Gamma \vdash \Sigma \rightsquigarrow S$.*
- *For any M, there exists at most one copyable Ξ with $\Gamma \vdash M : \Xi \rightsquigarrow m$.*

The proof is a straightforward induction.

Proposition 3. *There is an algorithm, which, given Γ and M, computes a copyable type Ξ and a program m such that $\Gamma \vdash M : \Xi \rightsquigarrow m$, if such Ξ and m exist, and rejects otherwise.*

Proof Sketch. If one restricts attention to copyable Ξ, then positions where an exponential can appear in the type are uniquely determined, cf. Lemma 2. This makes the elaboration rules essentially syntax-directed. For example, a term of the form $\mathtt{functor}(X : \Sigma_1) \to \Sigma_2$ must have a type of the form $!(\forall\overline{\alpha}.\, S \multimap \Xi)$.

Only rule SIG-E remains as a source of non-syntax-directedness. To deal with it, we first prove the property of the proposition under the assumption that Γ does not contain variables of record type. The general assertion follows, since such variables can be eliminated first using SIG-E. The proof of the property goes by induction on the term M. We show representative cases:

- M is $\mathtt{functor}(X : \Sigma) \to N$. The type must have the form $!(\forall\overline{\alpha}.\, S \multimap \Xi)$. By applying rule CLOSE-R, the goal reduces to deriving $M : \forall\overline{\alpha}.\, S \multimap \Xi$ in context $\Gamma,\ x : \mathtt{raw}$, for fresh x. Apply rule FUNCTOR-I. It now remains to derive $N : \Xi$ in context $\Gamma,\ x : \mathtt{raw},\ \overline{\alpha},\ X : S$. If S is a record type, then we use SIG-E in conjunction with DERELICTION and, if X is needed more than once, also CONTRACTION to bring the context in a form where we can apply the induction hypothesis. Then we can complete the derivation using the induction hypothesis.
- M is $N\ Y$. We can apply rule FUNCTOR-E. By applying the induction hypothesis to N, we get a type, which should be of the form $!(\forall\overline{\alpha}.\, S \multimap \Xi)$. If it is not, we reject. Otherwise, dereliction gives us that N also has type $\forall\overline{\alpha}.\, S \multimap \Xi$. We now check the type of Y using the variable rule from Lemma 1 and verify the matching hypothesis of rule FUNCTOR-E. For the latter, note that subtyping is not hard to decide, since variable range over base types only.

□

It is now not hard to compute a better choice of exponentials for the derivations with copyable types. If one replaces each exponential types $!S$ with $\alpha{\cdot}S$ for fresh α, then the rules from Lemma 1 lead to a number of $\lhd$-side conditions that just need to be solved. If α is a type variable and $D_1 \lhd \alpha, \ldots, D_n \lhd \alpha$ are all the constrains with α as an upper bound, then it is correct to let $\alpha := \mathtt{raw}_k$ where $k \geq \mathtt{size}(D_i)$ for all i, and where the size of type variables is defined to be ∞. By solving the constraints in the right order, one obtains a simple way of computing more precise exponentials. It amounts to a type-based flow analysis and appears to work well in experiments with a prototype implementation.

For the above example of a module term that uses a module variable X such that $X.\ell_1$ is used once and $X.\ell_2$ twice, the outlined approach would give the type $\mathtt{raw}_0{\cdot}\{\ell_1 : \mathtt{raw}_0{\cdot}\Sigma_1,\ \ell_2 : \mathtt{raw}_1{\cdot}\Sigma_2\}$ to X. One may identify $\mathtt{raw}_0$ with $\mathtt{unit}$.

7 Example Applications and Directions

Having motivated the module system as a convenient formalism for programming language applications of the GOI, we ought to outline intended applications of the module system and potential benefits of its use. In this section, we assume that all module types have exponentials placed like in normal form types. Since these are intended to be computed automatically, we do not show them and treat them as if they were written with invisible ink.

The first example, which uses all the features of the module system, explains why we believe that the module system will make working with GOI-constructions easier. Suppose we want to define a GOI-interpretation for a higher-order functional programming language with call-by-value semantics. For instance, we may want to use a call-by-value language for hardware synthesis. An efficient GOI-implementation of call-by-value is not completely straightforward.

Let us outline how to do it for the simply-typed λ-calculus with types $X, Y ::= \mathbb{N} \mid X \to Y$. We want to implement call-by-value evaluation efficiently. To be able to see the evaluation strategy, let us assume that the λ-calculus has a constant `print`$: \mathbb{N} \to \mathbb{N}$ for outputting numbers, and that there is a corresponding effectful operation in the systems-level language.

To implement call-by-value evaluation, one can translate a closed λ-term $t : X$ to a module of type $\mathcal{M}[\![X]\!]$, as defined below, where $\mathcal{I}[\![X]\!]$ is defined by induction in the type X:

```
M[[X]] := sig
  T: I[[X]],
  eval: M(T.t)
end
```

```
I[[N]] := sig
    type t = int
  end
```

```
I[[X → Y]] := sig
  type t, / abstract /
  T: functor (X: [[X]]) → sig
       T: [[Y]],
       apply: t × X.t → M(T.t)
     end
end
```

Thus, a term of type $\mathbb{N}$ is translated essentially just to a computation `eval : M`**`int`** that computes the number and performs the effects of term t. A term of type $\mathbb{N} \to \mathbb{N}$ is translated to a computation `eval : Mt`, which computes the abstract function value and performs the effects of doing so, and a function `apply : t × `**`int`** `→ M`**`int`** for function application. In the higher-order case, where X is a function type, the function `apply` can make calls to `X.apply`. If Y is also a function type, then the module `T :` $[\![Y]\!]$ defines the `apply`-function for the returned function. The idea comes from [23], where the translation is described in more detail.

Defining the translation from a term $t : X$ to a module of type $\mathcal{M}[\![X]\!]$ is verbose, but essentially straightforward. Examples for application and the constant `print`$: \mathbb{N} \to \mathbb{N}$ are shown below.

```
[[print : N → N]] =
  struct
    type t = unit,
    T: functor (X: I[[N]]) → sig
      T = struct type t = int end
      fn apply(f:t, x:int) →
        let y = print(x) in return y
     end
    eval = return ()
  end :> M[[N → N]]
```

```
[[t1 t2 : Y]] =
  struct
    T1 = [[t1 : X → Y]],
    T2 = [[t2 : X]],
    S = T1.T(T2),
    T = S.T,
    eval = let f = T1.eval in
           let x = T2.eval in
           S.apply(f, x)
  end :> M[[Y]]
```

It should not be hard for a reader familiar with ML-like languages to fill in the rest of the details. If one adds a fixed-point combinator to the module system, then this approach can be extended to a full programming language.

A direct translation of the same translation to the GOI requires quite a bit more effort, however [23]. It is also possible to use an interaction calculus like in Section 5.1. This is described in [22, 23]; the work reported there amounts to working directly with the elaboration of the above modules. For example, $\mathcal{M}[\![\mathbb{N} \to \mathbb{N}]\!]$ elaborates to $\exists\alpha.\, \mathrm{M}\alpha \otimes\, !\,(\forall\beta.\, [=\beta] \multimap \exists\gamma.\, [=\gamma] \otimes (\alpha \times \beta \to \mathrm{M}\gamma))$. Working with such types is rather cumbersome, however, as one needs to pack and unpack existentials often. By allowing type declarations and type abstraction, the module system takes care of this task. Also, we hope that the above module type $\mathcal{M}[\![\mathbb{N} \to \mathbb{N}]\!]$ is easier to understand than its elaboration.

7.0.1 Directions

Having given an example where the module system simplifies existing work, we conclude by outlining where it may be useful as a basis for new work.

Parallelism. Dal Lago et al. [4] have given a parallel implementation of call-by-value in a multi-token GOI, where there is not just a single message being passed around a message-passing graph, but many at the same time.

A similar effect can be achieved using the par-monad [16] in the above modular implementation of call-by-value. Suppose the monad of the systems-level language integrates the par-monad for parallel computation. This monad provides write-once references that can be used to synchronise parallel processes.

For any type A there is a new type $\mathtt{IVar}(A)$ of a write-once reference that may be empty or contain a value of type A. There is an operation $\mathtt{new} : \mathrm{M}(\mathtt{IVar}(A))$ for creating new reference cells. An operation $\mathtt{put} : A \to \mathtt{IVar}(A) \to \mathtt{Munit}$ allows one to write a value into the cell. An operation $\mathtt{get} : \mathtt{IVar}(A) \to \mathrm{M}A$ reads from the reference. If `get` cannot read a value because the cell is empty, then it will wait until a value has been written into the cell by another process. Finally, there is an operation $\mathtt{fork}(e)$ that forks off a new process that executes e.

With the par-monad, the above types $\mathcal{M}[\![X]\!]$ and $\mathcal{I}[\![X]\!]$ can be modified to allow parallel computation. Change the types of `eval` and `apply` to $\mathtt{eval} : \mathtt{IVar}(T.t)$ and $\mathtt{apply} : \mathtt{IVar}(t) \times \mathtt{IVar}(X.t) \to \mathtt{IVar}(T.t)$. Executing the `eval`-computation for $[\![t : X]\!]$ now only enables the computation of t. In essence, `eval` creates an `IVar` for each subterm of t and forks processes that will fill these `IVars` with the value of the subterm, once all the values required to do so have appeared in their `IVars`. The return value of `eval` is not the value of the term, but an `IVar` in which the value will appear. To illustrate the idea, let us define `eval` for a primitive addition operation $[\![t_1 + t_2 : \mathbb{N}]\!]$:

```
eval = let c1 = [[t1 : N]].eval in let c2 = [[t2 : N]].eval in let res = new in
       fork (let v1 = get c1 in let v2 = get c2 in put (v1 + v2) res);
       return res
```

This approach to parallel computation seems similar to the multi-token computation of [4], if one thinks of the multiple tokens as values materialising in `IVars`. The effect of `eval` is implicitly built into the model. While the detailed correspondence remains to be worked out, it is an example of the kind of questions one can study with the modular formulation of the GOI.

Game semantics. It is also interesting the use the modular formulation to study game semantics from an implementation-oriented perspective. Suppose we have a module of type $\mathcal{I}[\![X]\!]$ that we can interact with. What is the trace of actions that can appear in the interaction? If X is $\mathbb{N}$, then we can evaluate `eval` and we obtain an integer i as result. If X is a function $\mathbb{N} \to \mathbb{N}$, then we can evaluate `eval`, but we cannot call `apply` right away, as the type `t` is abstract and we do not have a value of it yet. But evaluating `eval` gives us some value of type `t`, let us call it $*$ (we cannot inspect it because of abstraction). With this value, we can call $\mathtt{apply}(*, i)$ with any argument i number. The result will be the return value j of the implemented function.

So, in the case of $X = \mathbb{N}$, the traces of interaction have the form $\mathtt{eval?}, \mathtt{eval!}(i)$, where "`eval?`" represents the action of starting the evaluation of `eval` and "`eval!`" the answer. In the case of $X = \mathbb{N} \to \mathbb{N}$, the traces of interaction have the from $\mathtt{eval?}, \mathtt{eval!}(*), \mathtt{apply?}(*, i), \mathtt{apply!}(j)$. In both cases, interleavings of such traces are possible as well. Note that type abstraction enforces a particular ordering of messages in the case of functions. When we compare the traces to Abramsky and McCusker's call-by-value games [3], we find that the traces correspond to the plays in the arenas of these games. This seems to remain true for arbitrary X. So, the module system may allow for a natural formulation of game semantics in a way that is close to actual implementations.

8 Conclusion

We have used the structure of the Geometry of Interaction to show how to extend first-order programming languages with an ML-style module system. The module system can be seen as a natural higher-order generalisation of systems-level linking. Even at higher order, functor application remains essentially linking. The module system is an improvement over standard systems-level linking. It supports standard first-order linking without overhead, but can also handle higher-order modules. The module system was formulated to be suitable for implementation. Its definitional, first-order nature may make the module system particularly suitable in conjunction with link-time optimisations, see e.g. [14]. Only the choice of `raw` for abstract types is a simplification for simplicity. This can be improved by using $\mathtt{raw}_k$ instead. To do so, one can extend the module system to keep track of the sizes of types. Experience with an implementation of the application from Section 7 suggests [23] that this will be enough for an efficient implementation of higher-order modules.

The module system captures the central structure of the GOI in simple programming-language terms. It abstracts from low-level implementation details. Especially for programming-language applications, where

one is interested in efficient implementation, it is not a small task to account for them. We believe that this is a limiting factor in research on applications of the GOI, as one needs to expend a good amount of effort to work out such details every time. We hope that the module system can help to address these limitations better than this has been the case with variants of System F so far and make applications of the GOI easier and more accessible. We would like to start future work at the point of Section 7 rather than having to begin from Section 1.2 again.

References

[1] Samson Abramsky, Esfandiar Haghverdi, and Philip J. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[3] Samson Abramsky and Guy McCusker. Call-by-value games. In *Computer Science Logic, CSL 1997*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1997.

[4] Ugo Dal Lago, Claudia Faggian, Ichiro Hasuo, and Akira Yoshimizu. The geometry of synchronization. In *Computer Science Logic – Logic in Computer Science, CSL-LICS 2014*, pages 35:1–35:10, 2014.

[5] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space bounded functional programming. Information and Computation, 2016.

[6] Martin Elsman. Static interpretation of modules. In *International Conference on Functional Programming, ICFP 1999*, pages 208–219, New York, NY, USA, 1999. ACM.

[7] Olle Fredriksson and Dan R. Ghica. Seamless distributed computing from the geometry of interaction. In Catuscia Palamidessi and Mark Dermot Ryan, editors, *Trustworthy Global Computing, TGC 2012*, volume 8191 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2012.

[8] Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *Principles of Programming Languages, POPL 2007*, pages 363–375. ACM, 2007.

[9] Jean-Yves Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 69–108. American Mathematical Society, 1989.

[10] Jean-Yves Girard. Proof-nets: The parallel syntax for proof-theory. In Aldo Ursini and Paulo Agliano, editors, *Logic and Algebra (Pontignano, 1994)*, Lecture Notes in Pure and Applied Mathematics, pages 97–124. CRC Press, 1996.

[11] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Principles of Programming Languages, POPL 1994*, pages 123–137, New York, NY, USA, 1994. ACM.

[12] Ichiro Hasuo and Naohiko Hoshino. Semantics of higher-order quantum computation via geometry of interaction. In *Logic in Computer Science, LICS 2011*, pages 237–246. IEEE, 2011.

[13] Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *Computer Science Logic – Logic in Computer Science, CSL-LICS 2014*. ACM, 2014.

[14] T. Johnson, M. Amini, and X. David Li. Thinlto: Scalable and incremental lto. In *Code Generation and Optimization, CGO 2017*, pages 111–121, 2017.

[15] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[16] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Symposium on Haskell, Haskell 2011*, pages 71–82, New York, NY, USA, 2011. ACM.

[17] Andreas Rossberg. 1ml – core and modules united (f-ing first-class modules). In *International Conference on Functional Programming, ICFP 2015*, pages 35–47, New York, NY, USA, 2015. ACM.

[18] Andreas Rossberg and Derek Dreyer. Mixin’ up the ml module system. *ACM Trans. Program. Lang. Syst.*, 35(1):2:1–2:84, 2013.

[19] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.

[20] Ulrich Schöpp. Call-by-value in a basic logic for interaction. In Jacques Garrigue, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2014*, volume 8858 of *Lecture Notes in Computer Science*, pages 428–448. Springer, 2014.

[21] Ulrich Schöpp. On the relation of interaction semantics to continuations and defunctionalization. *Logical Methods in Computer Science*, 2014.

[22] Ulrich Schöpp. From call-by-value to interaction by typed closure conversion. In *Asian Symposium on Programming Languages and Systems, APLAS 2015*, volume 9458 of *Lecture Notes in Computer Science*, pages 251–270. Springer, 2015.

[23] Ulrich Schöpp. Defunctionalisation as modular closure conversion. In *Principles and Practice of Declarative Programming, PPDP 2017*, New York, NY, 2017. ACM.

[24] Glynn Winskell. *The Formal Semantics of Programming Languages*. MIT Press, 1993.