Functional BIP: Embedding Connectors in Functional Programming Languages

Romain Edelmann, Simon Bliudze, and Joseph Sifakis

École polytechnique fédérale de Lausanne
Rigorous System Design laboratory
EPFL IC IIF RiSD, Station 14, CH-1015, Lausanne, Switzerland
<name.surname>@epfl.ch

NOTE TO REVIEWERS / EDITORS:

The body of this paper spans 18 pages. For the reviewers' convenience, some benchmarks from [13] are reproduced as an appendix on pages 19–22.

Abstract This paper presents a theoretical foundation for functional language implementations of Behavior-Interaction-Priority (BIP). We introduce a set of connector combinators describing synchronization, data transfer, priorities and dynamicity in a principled way. A static type system ensures the soundness of connector semantics.
Based on this foundation, we implemented BIP as an embedded domain specific language (DSL) in Haskell and Scala. The DSL embedding allows programmers to benefit from the full expressive power of high-level languages. The clear separation of behavior and coordination inherited from BIP leads to systems that are arguably simpler to maintain and reason about, compared to other approaches.

Keywords: BIP, connectors, combinators, dynamicity, static typing, functional programming, Haskell, Scala

1 Introduction

This paper proposes a lightweight and elegant implementation of the BIP (Behavior-Interaction-Priority) [2,3] in functional languages. BIP is a framework for component-based design of correct-by-construction applications. It provides a simple, but powerful mechanism for coordination of concurrent components by superposing three layers: Behavior, Interaction, and Priority.

Systems are composed of atoms (atomic components) that have communication *ports* used for coordination. Atoms have disjoint state spaces; their behavior is specified as a system of transitions labeled with ports. Interactions between components are defined by hierarchically structured connectors [5,7]. A connector description consists of three parts:

1. A control part specifying a relation between a set of strongly synchronized bottom ports—at most one per atomic component—and a single top port. The latter can be used as bottom ports of higher level connectors.
2. A data-flow part specifying the computation associated with the interaction. The computation can affect variables associated with the ports. It consists of an upstream computation followed by a downstream computation. The former is specified by a function that takes as arguments the values exported

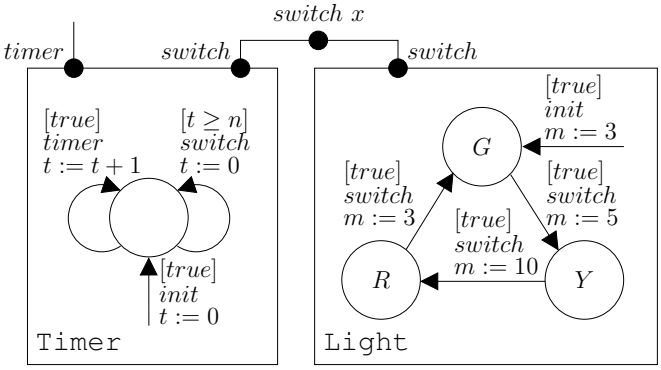


Figure 1: Traffic light in BIP

through the ports involved in the interaction. The computed value is exported through the top port. The downstream computation produces values associated with the synchronized ports from the value received at the top port. This allows bidirectional exchange of information upon synchronisations among components.

3. A Boolean guard determining the enabledness of an interaction depending on the values of the provided data: the interaction is only enabled if the data provided by the components satisfies the guard [7].

Finally, priorities are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled simultaneously. Interaction and Priority layers are collectively called *Glue*.

Figure 1 shows a traffic light controller system modeled in BIP. It is composed of two atomic components `Timer` and `Light`, modelling, respectively, a timer and the light switching behavior. `Timer` uses the variable t—initialised to 0—to keep track of the time spent since the last change of color. `Light` determines the color of the traffic light and the duration that the light must stay in a specific state (i.e., a specific color of the light). The system has two connectors: a singleton connector with one port *timer* and no data transfer and a binary connector, synchronising the ports *switch* of the two components. The first, singleton connector is necessary, since, in BIP, only ports that belong to at least one connector can fire. The second connector has a top port, also called *switch*, and an associated data variable x used for transfer. It is defined by the following *interaction expression* [7]:

$$(switch \leftarrow \texttt{Timer}.switch\ \texttt{Light}.switch).[true : x := \texttt{Timer}.m // \texttt{Light}.n := x],$$

where $(switch \leftarrow \texttt{Timer}.switch\ \texttt{Light}.switch)$ is the control part, the guard is the constant predicate *true*, the upward and downward dataflows are defined, respectively, by the assignments $x := \texttt{Timer}.m$ and $\texttt{Light}.n := x$. Thus, upon each synchronisation, `Light` informs `Timer` about the amount of time to spend in the next location. Notice that, when $t \geq n$, both transitions, *timer* and *switch*, of the `Timer` atom are enabled. Since all other guards in the system are constant

predicates *true*, this means that both connectors can fire. Imposing the priority $timer < switch$ resolves this choice, so that switching is performed whenever possible. In general, it is not necessary to impose priorities in all conflict situations: according to the BIP semantics, one of the enabled maximal priority interactions is chosen non-deterministically [5].

The strict separation between behavior—i.e., stateful components—and coordination—i.e., stateless connectors and priorities—leads to systems that are simpler to understand, test and maintain. Furthermore, hierarchical combination of interactions and priorities allows universal expressiveness [1,6]. The BIP language has been implemented as a coordination language for C. It is supported by a toolset including translators from various programming models into BIP, source-to-source transformers as well as compilers for generating code executable by dedicated engines.[1]

This paper adapts the BIP coordination mechanisms to the context of functional programming languages: systems are composed of atoms, ports and a *single static* connector. As in the original BIP implementations, atoms have their own private memory, which is not shared with other atoms. Thus, atoms can communicate only by interacting through the ports of the system. The main difference is that, in Functional BIP, ports are not directly associated with atoms. On the contrary, they are globally accessible entities used for communication and coordination. We introduce the combinator `Bind` to attach a port to a specific atom.

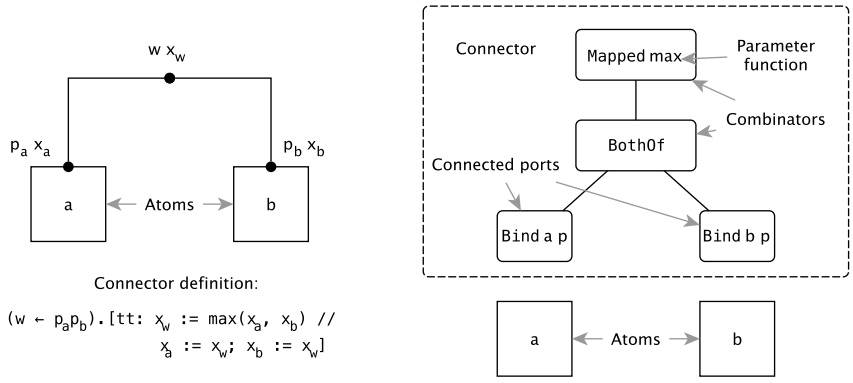
Such dissociation of ports from atoms allows us to introduce forms of dynamicity, which are not currently available in any of the original BIP implementations: some atoms are created at the system initiation, while others may be spawned by existing atoms at runtime; we introduce the combinator `Dynamic`, which binds all existing atoms, including those created at runtime, to a given port.

Ports offer a very simple interface which consists of a single operation, called `await`. When invoking `await`, the atom sends a value to the port and blocks until the port sends a value back to the atom. Atoms may wait on multiple ports simultaneously. Ports are typed to describe the type of values that can be sent and received through them.

The connector of a system links atom-port pairs to define the possible interactions. An interaction describes a set of strongly synchronized components which exchange values via ports. Several interactions may be enabled at the same time. The connector is unique and fixed throughout the system lifespan.

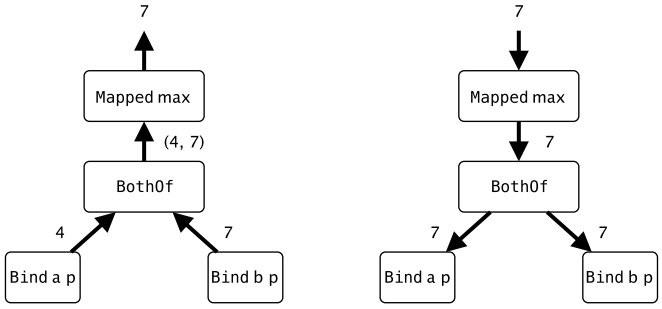
We introduce additional combinators to describe connectors. Connector combinators are used to hierarchically build complex connectors, starting from a set of basic ones [8]. Figure 2 shows a simple BIP system and its equivalent in Functional BIP. The system is composed of two atoms which can communicate by sending a value on the port p. When both components have done so, an interaction is possible and both components receive the maximum of the two values sent.

[1] `http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html`



(a) In BIP (b) In Functional BIP

Figure 2: View of a system in BIP and in Functional BIP



(a) Upwards phase (b) Downwards phase

Figure 3: Data transfer within connectors

In Functional BIP, the connector is represented as a tree, with port-atom bindings at the leaves and combinators in the nodes. The arity of each node depends on the corresponding combinator.

Data transfer within the connector occurs in two phases: upward and downward. During the upwards phase, the values sent through the ports by the atoms are collected and propagated up through the connector hierarchy. Once a value reaches the top of the hierarchy, it is transferred downwards, back to the atoms involved in the interaction. Both up- and downward data transfers at each node of the tree are defined by the corresponding combinator. Figure 3 illustrates the data transfer within the connector in Fig. 2, assuming that the atoms a and b have sent, respectively, the values 4 and 7 on the port p.

A connector ensures that atoms may only receive a value on the ports on which they are waiting. To maintain this invariant, we ensure that only the

combinators which provided a value during the upwards computation phase may receive a value during the downwards phase.

The contributions of this paper are the following:

- We present a set of combinators to build connectors which can describe synchronization, data transfer, priorities and dynamicity [9]. We introduce the formal semantics and typing rules of those combinators and present some of their algebraic properties.
- We present implementations of the concepts developed in this paper in two functional programming languages, Haskell [22] and Scala [23]. In each case, we show how the concepts can be transposed using well known idioms of the language. The two resulting frameworks are released under open source licenses and are freely available for download and use[2][3].

The proofs of all the results in this paper, as well as additional examples and implementation details, can be found in the Master thesis of first author [13].

The rest of this paper is structured as follows. Section 2 introduces the typing and semantic framework for connector combinators. Section 3 introduces the core, data, priority and dynamic connector combinators. Section 4 discusses their algebraic properties. Section 5 describes an implementation of the presented concepts in Haskell and Scala. Section 6 discusses the related work. Section 7 concludes and summarises the paper.

2 Semantic Framework

Before we introduce the connector combinators, we introduce notation and concepts that will be used to describe their semantics and types.

Partial Functions We denote by $A \rightharpoonup B$ the set of partial functions from the set A to the set B. We will sometimes use the fact that partial functions can be represented as sets of pairs from $A \times B$, in particular when we will construct such partial functions. For instance, we will denote the empty partial function by $\emptyset$ and the singleton partial function that maps x to y by $\{(x, y)\}$ or even $\{x \mapsto y\}$, whichever is clearer in the context.

Universe of Values We denote by $\mathcal{V}$ the set of values which can be handled and exchanged by the atoms. We assume that this set contains at least the Boolean values `true` and `false`, integers, all pairs of values and all total functions from values to values. Connectors are also part of the universe of values.

Types The values are equipped with a polymorphic type system, similar to Haskell's type system. We denote by $v : t$ the fact that a value $v \in \mathcal{V}$ has type t. Booleans are given the type `bool`, integers the type `int` while pairs of values of type a and b are given the type $a \times b$. Total functions from values of type a to values of type b are given the type $a \to b$.

Connectors propagating values of type u during the upwards phase and receiving values of type d during the downwards phase have the type $u \uparrow\downarrow d$. The type system dictates how the connector combinators can be used.

[2] Haskell version: `https://github.com/redelmann/bip-in-haskell`
[3] Scala version: `https://github.com/redelmann/bip-in-scala`

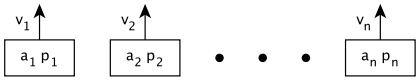


Figure 4: System state

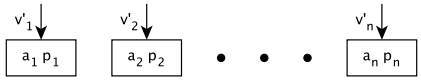


Figure 5: Assignment

Atoms We denote by $\mathcal{A}$ the set of atom identifiers of a system.

Ports We denote by P the set of ports of a system. Each port $p \in P$ is associated with two types: $\mathit{sendType}(p)$ and $\mathit{receiveType}(p)$. Atoms can only send values of type $\mathit{sendType}(p)$ through the port p and are guaranteed that the value eventually received from the port, if any, has type $\mathit{receiveType}(p)$.

Connectors We denote by $\mathcal{C}$ the set of connectors c for which there exists at least two types u and d such that $c : u \uparrow\downarrow d$.

System States A system state is as partial function, mapping atom-port pairs to corresponding values sent by the atom through the port. The set of states is:

$$\mathcal{S} = \{f \in (\mathcal{A} \times P) \rightharpoonup \mathcal{V} \mid \forall\big((a,p) \mapsto x\big) \in f\,,\ x : \mathit{sendType}(p)\}\,.$$

In a given system state, all atoms are waiting on a (possibly empty) subset of ports. When a port is active for an atom, a value of the appropriate type has been sent through it. Figure 4 shows a system state where atoms a_i have sent values v_i on ports p_i.

Assignments Similarly, an assignment is a partial function, mapping atom-port pairs to corresponding values received by the atom on the port. The set of assignments is:

$$\mathcal{R} = \{f \in (\mathcal{A} \times P) \rightharpoonup \mathcal{V} \mid \forall\big((a,p) \mapsto x\big) \in f\,,\ x : \mathit{receiveType}(p)\}\,.$$

An assignment maps each atom-port pair to at most one value. The value assigned, if any, is the value received by the atom on the given port. Figure 5 represents an assignment where atoms a_i receive values v_i on the ports p_i.

Open Interactions We denote by $\mathcal{O} = \mathcal{V} \times (\mathcal{V} \rightharpoonup \mathcal{R})$ the set of open interactions. An open interaction consists of an *upwards value* in $\mathcal{V}$ and a *downwards function* in $\mathcal{V} \rightharpoonup \mathcal{R}$ that, given a *downwards value* may return an assignment. Those interactions are called *open* as the values exchanged are exposed. Figure 6 shows the schematic representation of an open interaction.

When the upwards value is in the domain of the downwards function, the open interaction can be closed to obtain a valid assignment. In this case, the upwards value is used as a downwards value:

$$\begin{aligned} &\mathit{close} : \mathcal{O} \rightharpoonup \mathcal{R} \\ &\mathit{close}((v,g)) = g(v),\ \text{if } g \text{ is defined at } v\,. \end{aligned} \tag{1}$$

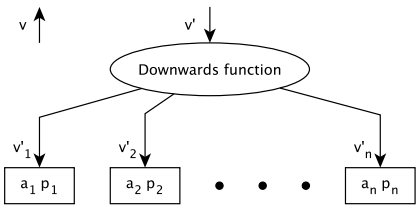


Figure 6: Open interaction

Downwards Compatibility We consider two assignments $R_1, R_2 \in \mathcal{R}$ to be *downwards compatible* if and only if:

$$\{a \mid \exists p \in P.(a,p) \in domain(R_1)\} \cap \{a \mid \exists p \in P.(a,p) \in domain(R_2)\} = \emptyset\,.$$

Intuitively, two assignments are downwards compatible if they involve distinct sets of atoms. We extend this notion to downwards functions. Downwards functions g_1, g_2 are downwards compatible if and only if they produce downwards compatible assignments for all possible downwards values. Downward compatibility ensures that two interactions can be fused [5] without violating the requirement that atoms are only involved once in an interaction.

3 Connector Combinators

Using the semantic framework introduced in Sect. 2, we can now introduce the set of combinators used to describe connectors. For the sake of clarity, we will introduce combinators progressively, in related groups. First, we introduce core combinators, which correspond to the Algebra of Interactions [5]. Then, we introduce combinators describing data manipulation, priorities and dynamicity.

For each connector combinator, we will provide the type inference rule defining the type of the resulting connector from the types of the parameters and children connectors. We also provide formal semantics of each combinator. To this end, we define the following semantic function:

$$[\cdot] : \mathcal{C} \to \mathcal{S} \to 2^{\mathcal{O}}$$

This semantic function gives, for each connector and system state, the set of possible open interactions. We will define this function progressively as we encounter the different set of connector combinators. We will define it recursively for each of the possible combinators.

3.1 Core Combinators

`Bind` This combinator takes a port and an atom and binds them together. It is equivalent to the port of an atomic component in BIP. The type of the resulting connector depends on the send and receive types of the port:

$$\frac{}{\texttt{Bind}\ a\ p : sendType(p) \uparrow\downarrow receiveType(p)}$$

The resulting connector provides a single interaction if the atom is currently waiting on the port and none otherwise. The value propagated upwards by this connector is the value, if any, that was sent through the port by the atom. When this connector receives a value during the downwards phase, it is transmitted to the atom via the given port. Expressed in terms of the open semantics function, the behavior of the resulting connector is defined as:

$$[\texttt{Bind}\ a\ p](S) = \begin{cases} \{(S(a,p), \{d \mapsto \{(a,p) \mapsto d\} \mid d : \mathit{receiveType}(p)\})\}\,, \\ \qquad \text{if } (a,p) \in \mathit{domain}(S)\,, \\ \emptyset\,, \qquad \text{otherwise.} \end{cases}$$

If the system state has an entry for the given atom-port pair, then a single open interaction is possible. The upwards value of the open interaction is the value sent by the port, while the downwards function propagates the downwards value to the atom and port.

Success and Failure These two combinators do not involve any atoms and ports. They have arity zero and can only be found at the leaves of the connector tree.

Regardless of the system state, Success v always provides a single open interaction, whose upwards value is v. Failure, on the other hand, represents a connector that is never enabled. While by themseleves, these combinators do not present any practical interest, they can be combined with others to build more useful connectors. They correspond to the elements 1 and 0 of the Algebra of Interactions [5]. The types of connectors defined by Success and Failure are given by the following inference rules:

$$\frac{x : u}{\texttt{Success}\ x : u \uparrow\downarrow d}\,, \qquad \frac{}{\texttt{Failure} : u \uparrow\downarrow d}\,.$$

Notice that Success is polymorphic in the downwards type and Failure in both the upwards and downwards types. Expressed using the open semantics function, their behavior is defined as follows:

$$[\texttt{Success}\ v](S) = \{(v, \{d \mapsto \emptyset \mid d \in \mathcal{V}\})\}\,,$$
$$[\texttt{Failure}](S) = \emptyset\,.$$

OneOf This combinator expresses the non-deterministic choice between two connectors. The connector OneOf c_1 c_2 behaves as either c_1 or c_2. This combinator corresponds to the union operation in the Algebra of Interactions [5]. The type and open interaction semantics of OneOf c_1 c_2 are defined as follows:

$$\frac{c_1 : u \uparrow\downarrow d \qquad c_2 : u \uparrow\downarrow d}{\texttt{OneOf}\ c_1\ c_2 : u \uparrow\downarrow d}\,, \qquad [\texttt{OneOf}\ c_1\ c_2](S) = [c_1](S) \cup [c_2](S)\,.$$

Thus, the interactions possible in this connector are all interactions that are possible in either c_1 or c_2.

BothOf This last core combinator represents the fusion of two connectors. It corresponds to the fusion operator in the Algebra of Interactions [5]. The corresponding type inference rule is

$$\frac{c_1 : u_1 \uparrow\downarrow d \qquad c_2 : u_2 \uparrow\downarrow d}{\texttt{BothOf}\ c_1\ c_2 : (u_1 \times u_2) \uparrow\downarrow d}\,.$$

Thus, $\texttt{BothOf}\ c_1\ c_2$ propagates upwards the pair of upwards values coming from c_1 and c_2. In the presence of non-determinism, this connector returns all possible combinations of interactions from c_1 and c_2:

$$\begin{aligned}[\texttt{BothOf}\ c_1\ c_2](S) = \\ \Big\{\big((x_1, x_2), \{d \mapsto g_1(d) \cup g_2(d) \mid d \in \mathit{domain}(g_1) \cap \mathit{domain}(g_2)\}\big) \\ \Big| \ (x_1, g_1) \in [c_1](S)\,, (x_2, g_2) \in [c_2](S) \\ \text{and } g_1 \text{ and } g_2 \text{ are downwards compatible}\Big\}\,.\end{aligned}$$

Interactions that would lead to atoms receiving more than one value during the downwards phase are filtered out by the downwards compatibility check.

3.2 Data Combinators

We now introduce the combinators for data manipulation in the connectors.

Mapped and ContraMapped These two combinators apply a parameter function to, respectively, the upwards and downwards values propagated by the underlying connector. The types of the resulting connectors are defined by the inference rules:

$$\frac{f : u \to v \qquad c : u \uparrow\downarrow d}{\texttt{Mapped}\ f\ c : v \uparrow\downarrow d}\,, \qquad \frac{f : e \to d \qquad c : u \uparrow\downarrow d}{\texttt{ContraMapped}\ f\ c : u \uparrow\downarrow e}\,.$$

The two combinators never modify the number of possible interactions:

$$\begin{aligned}[\texttt{Mapped}\ f\ c](S) &= \{(f(x), g) \mid (x, g) \in [c](S)\}\,, \\ [\texttt{ContraMapped}\ f\ c](S) &= \{(x, g \circ f) \mid (x, g) \in [c](S)\}\,.\end{aligned}$$

Guarded This combinator is used to filter out open interactions whose upwards values fail to satisfy a predicate passed as the argument. The connector type is defined by the inference rule

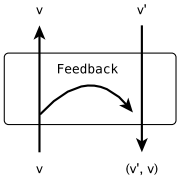
$$\frac{f : u \to \texttt{bool} \qquad c : u \uparrow\downarrow d}{\texttt{Guarded}\ f\ c : u \uparrow\downarrow d}\,.$$

This combinator allows restricting non-determinism, ensuring that upwards values all satisfy a given predicate:

$$[\texttt{Guarded}\ f\ c](S) = \{(x, g) \in [c](S) \mid f(x) = \texttt{true}\}\,.$$

Feedback This last data manipulation combinator allows feeding the upwards value back into the downwards propagation phase. The typing inference rule and the semantics are defined by putting

$$\frac{c : u \uparrow\downarrow (d, u)}{\texttt{Feedback}\ c : u \uparrow\downarrow d}\,,$$
$$[\texttt{Feedback}\ c](S) = \{(x, g \circ tag_x) \mid (x, g) \in [c](S)\}\,,$$



where $tag_x(y) \stackrel{def}{=} (y, x)$ (see the figure to the right).

3.3 Priority Combinators

In this sub-section, we introduce the combinators used to specify priorities. Priorities are used to inhibit the execution of certain interactions when interactions of higher priority are possible. Contrary to the classical BIP syntax, where priorities are defined by a separate syntactic construction and can be applied only at the top level of a connector within a compound component, priority combinators can be applied at any level in the connector hierarchy.

FirstOf This combinator imposes fixed-order priority among the sub-connectors. In the connector $\texttt{FirstOf}\ c_1\ c_2$, interactions from c_1 will be preferred over interactions from c_2, whenever the former are available. The type and semantics of the resulting connector are defined as follows

$$\frac{c_1 : u \uparrow\downarrow d \qquad c_2 : u \uparrow\downarrow d}{\texttt{FirstOf}\ c_1\ c_2 : u \uparrow\downarrow d}\,, \quad [\texttt{FirstOf}\ c_1\ c_2](S) = \begin{cases} [c_1](S) & \text{if } [c_1](S) \neq \emptyset\,, \\ [c_2](S) & \text{otherwise}\,. \end{cases}$$

Maximal Given a partial ordering on the upwards data domain encoded by a predicate f, the connector $\texttt{Maximal}\ f\ c$ returns all interactions from the connector c whose upwards values are maximal. The parameter function f should return `true` when its first parameter is strictly less than its second parameter, and `false` otherwise. The type of the resulting connector is defined by the inference rule

$$\frac{f : (u \times u) \to \texttt{bool} \qquad c : u \uparrow\downarrow d}{\texttt{Maximal}\ f\ c : u \uparrow\downarrow d}\,,$$

whereas its semantics is given by

$$[\texttt{Maximal}\ f\ c](S) = \{(u, g) \in [c](S) \mid \nexists (v, h) \in [c](S).f(u, v) = \texttt{true}\}\,.$$

Notice that Maximal is a strictly more powerful combinator than FirstOf: the latter can be defined by combining Mapped, OneOf and Maximal:

$$\texttt{FirstOf}\ c_1\ c_2 = \texttt{Mapped}\ untag\ \big(\texttt{Maximal}\ cmp_2 \big(\texttt{OneOf}\ (\texttt{Mapped}\ tag_2\ c_1)\ (\texttt{Mapped}\ tag_1\ c_2)\big)\big)\,,$$

where the function $untag$, cmp_2 and tag_n are defined as by putting

$$\begin{aligned} untag(x, n) &= x\,, \\ cmp_2((x, n), (y, m)) &= n < m\,, \\ tag_n(x) &= (x, n)\,. \end{aligned}$$

3.4 Dynamic Combinators

Finally, we introduce a set of combinators that allow dynamicity, i.e., creation and deletion of atoms at run time and dynamic reconfiguration of the set of possible interactions among the atoms. Since the connector of a system is static, such dynamicity cannot be realised with the combinators introduced so far and requires additional ones.

`Dynamic` Given a port p, the `Dynamic` combinator, binds all existing atoms to the port p. Atoms that are created at run time are also bound to the port. This combinator can be thought of as `Bind` with an atom chosen non-deterministically. Hence, the corresponding type inference rule and semantics are the following:

$$\frac{}{\texttt{Dynamic}\ p : sendType(p) \uparrow\downarrow receiveType(p)}\,,$$

$$[\texttt{Dynamic}\ p](S) = \Big\{ \big(S(a,p), \{d \mapsto \{(a,p) \mapsto d\} \mid d : receiveType(p)\}\big) \;\Big|\; (a,p) \in domain(S) \Big\}\,.$$

`Joined` This last combinator that we introduce allows dynamic connector reconfiguration by accepting connectors as values. The connector `Joined` c acts as a placeholder for connectors passed as upwards values by the connector c. The type of this connector is thus given by the inference rule

$$\frac{c : (u \uparrow\downarrow d) \uparrow\downarrow d}{\texttt{Joined}\ c : u \uparrow\downarrow d}\,.$$

The connector is named after the natural transformation μ of monads, called *join* or *multiply*, in the context of category theory and `join` in Haskell. The corresponding semantic function is defined by

$$\begin{aligned} &[\texttt{Joined}\ c](S) = \\ &\quad \Big\{ \big(u_2, \{d \mapsto g_1(d) \cup g_2(d) \mid d \in domain(g_1) \cap domain(g_2)\}\big) \\ &\qquad \Big| \; (u_1, g_1) \in [c](S), (u_2, g_2) \in [u_1](S) \\ &\qquad\qquad \text{and } g_1 \text{ and } g_2 \text{ are downwards compatible} \Big\}\,. \end{aligned}$$

This connector is very expressive and can be used to derive some of the combinators that we have introduced previously, such as `Guarded` and `BothOf`.

3.5 Closed Semantic Function

Based on the semantic function we have defined throughout this section, we now introduce the partial function $[\![\cdot]\!] : \mathcal{C} \rightharpoonup \mathcal{S} \to 2^{\mathcal{R}}$ that associates to a connector and a system state, the set of possible assignments defined by the interactions allowed by the connector (see (1) for the definition of *close*):

$$[\![c]\!](S) \stackrel{def}{=} \{close(o) \mid o \in [c](S)\}\,, \text{ if } c : a \uparrow\downarrow a \text{ for some type } a\,.$$

This function gives to a connector its meaning as a function from system states to possible assignments.

The following propositions guarantee that the closed semantics of any well-typed connector is well defined and the resulting behaviour satisfies the BIP consistency constraints.

Proposition 1. *For any type a, the function $[\![\cdot]\!]$ is well-defined on connectors of type $a \uparrow\downarrow a$.*

Proposition 2. *For any type a, connector $c \in \mathcal{C}$ of type $a \uparrow\downarrow a$, system state $S \in \mathcal{S}$ and assignment $R \in [\![c]\!](S)$, the inclusion $domain(R) \subseteq domain(S)$ holds.*

Proposition 2 means that assignments resulting from closing the connector only send values to atoms through the ports that these are waiting on.

Proposition 3. *For any type a, connector $c \in \mathcal{C}$ of type $a \uparrow\downarrow a$ and system state $S \in \mathcal{S}$, all assignments in $[\![c]\!](S)$ involve each of the atoms at most once.*

Notice that the closed semantics of connectors allows simulating the `Feedback` combinator. This can be achieved by modifying all the combinators higher in the connector tree hierarchy to propagate the corresponding value upwards then downwards back to the connector, to which `Feedback` has to be applied. Therefore, the `Feedback` combinator is not, strictly speaking primitive. However, it greatly simplifies this construction, by allowing feeding values back locally.

4 Algebraic Properties

Algebraic structures often form powerful abstractions in functional programming languages. Such structures, such as monoids, functor and monads [24], as well as many others[25], have been successfully used in the context of functional programming.

In Haskell, those algebraic structures form typeclasses such as `Monoid`, `Functor` and `Monad` and can be used very effectively to build high-level expressions. The `Monad` typeclass is so prevalent that syntactic sugar (the *do notation*) is present in the language to make its use easier.

In Scala, those structures are not explicitly present. However, many methods, such as `map` and `flatMap` mirror the functor and monad operations. Syntactic sugar (the *for-notation*) is present in the language and, as in Haskell, makes the use of multiple `map` and `flatMap` operations syntactically lighter.

In this section, we show that connector combinators are instances of such algebraic structures. The fact that connector combinators follow these algebraic

structures, allows connectors to be made instances of the corresponding type-classes. This allows programmers to use well known abstractions and syntactic sugar to build connectors.

The following two lemmata show that connector combinators form two monoidal structures.

Lemma 1. $(\mathcal{C}, \texttt{OneOf}, \texttt{Failure})$ *is a commutative monoid, that is* `OneOf` *is associative and commutative, with the identity element* `Failure`*.*

Lemma 2. *Given a commutative monoid with the binary operation* $\times$ *and an identity element* 1*, the structure* $\big(\mathcal{C}, \texttt{Mapped}\ (\cdot\times\cdot)\ \texttt{BothOf}, \texttt{Success}\ 1\big)$ *is also a commutative monoid, that is* `Mapped` $(\cdot\times\cdot)$ `BothOf` *is associative and commutative, with the identity element* `Success` 1*.*

Notice that the values $\mathcal{V}$ and types introduced in Sect. 2 form a category, whose objects are types, whereas arrows between objects a and b are functions $f : a \to b \in \mathcal{V}$. We denote this category T.

Lemma 3. *For any type* d*, the mapping* F^d_{up}*, defined, for each type* a *and function* $f : a \to b$*, by* $F^d_{up}(a) = a \uparrow\downarrow d$ *and* $F^d_{up}(f) = \texttt{Mapped}\ f : (a \uparrow\downarrow d) \to (b \uparrow\downarrow d)$*, is a functor from the category* T *to itself, that is*

$$\begin{aligned} \texttt{Mapped}\ \mathit{identity}_a &= \mathit{identity}_{(a\uparrow\downarrow d)}\,, \\ \texttt{Mapped}\ (g \circ f) &= \texttt{Mapped}\ g \circ \texttt{Mapped}\ f\,. \end{aligned}$$

Lemma 4. *For any type* u*, the mapping* F^u_{down}*, defined, for each type* d *and function* $f : a \to b \in \mathcal{V}$*, by* $F^u_{down}(d) = u \uparrow\downarrow d$ *and* $F^u_{down}(f) = \texttt{ContraMapped}\ f : u \uparrow\downarrow b \to u \uparrow\downarrow a$ *is a contravariant functor from the category* T *to itself, that is*

$$\begin{aligned} \texttt{ContraMapped}\ \mathit{identity}_a &= \mathit{identity}_{(a\uparrow\downarrow d)}\,, \\ \texttt{ContraMapped}\ (g \circ f) &= \texttt{ContraMapped}\ f \circ \texttt{ContraMapped}\ g\,. \end{aligned}$$

Lemma 5. *For any type* d*, the functor* F^d_{up} *along with* `Success` *and* `Joined` *form a monad, i.e., the following four properties hold:*

- `Success` *is a natural transformation from* $1_{\mathcal{V}}$ *to* F^d_{up}*:*

$$\texttt{Success} \circ f = \texttt{Mapped}\ f \circ \texttt{Success}\,,$$

- `Joined` *is a natural transformation from* $F_{up} \circ F_{up}$ *to* F_{up}*:*

$$\texttt{Joined} \circ \texttt{Mapped}\ (\texttt{Mapped}\ f) = \texttt{Mapped}\ f \circ \texttt{Joined}\,,$$

- *First monad law:*

$$\texttt{Joined} \circ \texttt{Mapped Joined} = \texttt{Joined} \circ \texttt{Joined}\,,$$

- *Second monad law:*

$$\texttt{Joined} \circ \texttt{Mapped Success} = \texttt{Joined} \circ \texttt{Success} = \mathit{identity}\,.$$

```
data Connector s d u
```
(a) Connector type in Haskell

```
class Connector[-D, +U]
```
(b) Connector type in Scala

Figure 7: The Connector type in Haskell and Scala

5 Implementation

We have implemented the concepts presented in this paper in two functional programming languages, Haskell[4] and Scala[5]. These implementations allow programmers to build concurrent systems following the principles of BIP in very expressive high-level languages. Thus programmers can separately describe the behavior and coordination of their system. The connector combinators we have introduced in Sect. 3, along with a library of derived combinators can be used to describe connectors.

Both implementations allow programmers to describe concurrent systems with a set of instructions, which are used to declare the atoms and ports of the system. Special instructions, such as `await` and `spawn` are provided for atoms to respectively wait on ports and spawn children atoms.

The connector of a system is described using the connector combinators we have introduced in this paper and some others derived combinators. The connector type indicates the types of upwards and downwards values propagated by the connector. In both cases, connectors are encoded as Generalised Algebraic Data Types (GADTs) as illustrated in Fig. 7. Notice that the Haskell connector type has a type parameter `s`. This parameter is a phantom type, used to ensure that identifiers do not escape the scope of the system in which they are defined.

As shown in Sect. 4, connectors follow the laws of many interesting algebraic structures. This allows us to make connectors instances of many Haskell typeclasses, such as `Monoid`, `Functor`, `ProFunctor`, `Applicative`, `Alternative`, `Monad` and `MonadPlus`. These typeclasses allow programmers to build connectors using concepts and functions they already are familiar with.

In Scala, the Connector class implements common methods such as `map`, `flatMap` and `filter`. These methods often mirror those of Haskell typeclasses.

The execution engine is composed of three distinct parts:

The pool of threads is used to concurrently execute the behavior of atoms.

The system state records information about waiting atoms. It is encoded as a mapping from atoms and ports to:

- nothing, if the atom is not currently waiting on the port;
- the value sent by the atom on the port, along with the continuation of the atom, if the atom is waiting on the port.

The continuations of atoms are stored to avoid needlessly blocking the execution thread of waiting atoms.

The engine core computes possible interactions from the system state and connector. This computation normally takes place when all atoms are either

[4] https://github.com/redelmann/bip-in-haskell

[5] https://github.com/redelmann/bip-in-scala

done with their execution or waiting on ports. In Edelmann's Master thesis [13], we investigate a way to execute some of those interactions earlier, while some atoms are still executing.

We illustrate the implementations of Functional BIP by a concurrent system composed of a producer and 20 consumers. The producer repeatedly produces values that are then sent on the `send` port of the system. Consumers repeatedly wait to receive a value on the `receive` port and then directly consume it. The connector of the system states that the values sent by the producer on the `send` port may be be transmitted to any atom waiting on the `receive` port. Figure 8 shows a Haskell and a Scala implementations.

6 Related Work

We show that an expressive component framework such as BIP can be defined as a DSL based on general purpose functional programming languages. Writing DSL's in functional languages allows their programmers to use powerful modeling concepts and tools with minimal effort. Similar work can be found in [15,19]. Approaches such as [12] also use combinators to describe coordination, but do so locally at each process. These approaches have no notion of a global coordination object such as the connectors we present.

Dynamicity in BIP has been studied by several authors [9,10,11]. In [9], the authors present the Dy-BIP framework that allows dynamic reconfiguration of connectors among the ports of the system. They use *history variables* to allow sequences of interactions with the same instance of a given component type. Functional BIP can emulate history variables using data. Dymanic combinators allow reconfiguration of connectors, but also creation and deletion of atoms, thereby extending the expressivity with respect to Dy-BIP.

In [11], the authors revisit the BIP expressiveness, by introducing simple behaviour, such as prefixing, in the BIP glue operators. They show that such minor modifications can rapidly lead to Turing completeness of glue. In contrast to [11] and other frameworks, such as [16,20], Functional BIP relies on a clear distinction between behavior and coordination expressed by memoriless connectors obtained by combinator composition. From this perspective, the approach in [10] is closest to the one we adopted in Functional BIP. Although, the expressiveness of the two approaches seems very close, we leave the formal investigation for future work.

In contrast to other formalisms such as [21] our framework supports full dynamism and is rooted in rigorous abstract semantics. In [18], dynamic architectures are defined as a set of global transitions between global configurations. These transitions are expressed in a first order logic extended with architecture-specific predicates. The same logic is used in [14,17] but global configurations are computed at runtime from the local constraints of each component. [4] provides an operational semantics based on the composition of global configurations from local ones. These express three forms of dependencies between services (mandatory, optional and negative). Nonetheless, dynamism is supported only at the installation phase.

```
main = runSystem $ do

  -- Creation of the two ports.
  -- Port on which to send the values
  -- produced.
  send <- newPort

  -- Port on which to receive the values
  -- to consume.
  receive <- newPort

  -- Creation of 20 consumers.
  replicateM 20 $ newAtom $ forever $ do
    value <- await receive ()
    consume value

  -- Creation of the producer.
  producer <- newAtom $ forever $ do
    value <- produce
    await send value

  -- The connector of the system
  registerConnector $
    -- The producer on its send port.
    bind producer send
    <*
    -- Any atom on the request port.
    dynamic receive
```

(a) Haskell

```
object ProducerConsumers {
  def main(args: Array[String]) {
    // Creation of the system.
    val system = new System

    // Creation of the two ports of the system.
    val send    = system.newPort[Any, Int]
    val receive = system.newPort[Int, Unit]

    // Creation of the consumers.
    for (_ <- 1 to 20) yield system.newAtom {
      def act() {
        await(receive) { (value: Int) =>
          consume(receive)
          act()
      } }
      act()
    }

    // Creation of the producer.
    val producer = system.newAtom {
      def act() {
        val value = produce()
        await(send, value) { (_: Any) =>
          act()
      } }
      act()
    }

    // The connector of the system.
    system.registerConnector {
      producer.bind(send).
            andLeft(dynamic(receive))
    }
    system.run()
} }
```

(b) Scala

Figure 8: Producer-Consumers example implemented in Haskell and Scala

7 Conclusion

The paper shows how the BIP component framework can be embedded in functional host languages. The embedding consists in defining sets of connector combinators that can describe the coordination mechanisms of BIP. The definition is progressive and incremental. We first define combinators to express synchronization and associated data transfer between the components of a system. Then, we have introduced combinators for the application of priority policies allowing conflict resolution between enabled interactions. Finally, we have presented combinators to deal with the dynamic creation of components and connectors.

We have shown that the set of the defined combinators enjoy interesting algebraic properties and form well-known algebraic structures which are of particular importance for the implementation of connectors in functional programming languages.

The two implementations show how these concepts can be transposed, respectively, into Haskell and Scala. For both languages, we have released an open source framework which programmers can use to build concurrent systems using the high-level coordination primitives offered by BIP.

References

1. Baranov, E., Bliudze, S.: Offer semantics: Achieving compositionality, flattening and full expressiveness for the glue operators in BIP. Science of Computer Programming 109(0), 2–35 (2015)
2. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Software Engineering and Formal Methods 2006. pp. 3–12. IEEE (2006)
4. Belguidoum, M., Dagnat, F.: Dependency management in software component deployment. Electronic Notes in Theoretical Computer Science 182, 17–32 (2007)
5. Bliudze, S., Sifakis, J.: The algebra of connectors—structuring interaction in BIP. IEEE Transactions on Computers 57(10), 1315–1330 (2008)
6. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: CONCUR 2008, pp. 508–522. Springer (2008)
7. Bliudze, S., Sifakis, J., Bozga, M.D., Jaber, M.: Architecture internalisation in BIP. In: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering. pp. 169–178. CBSE '14, ACM (2014)
8. Bliudze, S., Sifakis, J., Bozga, M.D., Jaber, M.: Architecture internalisation in BIP. In: CBSE '14. pp. 169–178. ACM (2014)
9. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: Software Composition. pp. 1–16. Springer (2012)
10. Bruni, R., Melgratti, H.C., Montanari, U.: Behaviour, interaction and dynamics. In: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi. LNCS, vol. 8373, pp. 382–401. Springer (2014), http://dx.doi.org/10.1007/978-3-642-54624-2_19
11. Di Giusto, C., Stefani, J.B.: Revisiting glue expressiveness in component-based systems. In: COORDINATION 2011. pp. 16–30. Springer (2011), http://dx.doi.org/10.1007/978-3-642-21464-6_2

12. Donnelly, K., Fluet, M.: Transactional events. SIGPLAN Not. 41(9), 124–135 (Sep 2006), http://doi.acm.org/10.1145/1160074.1159821
13. Edelmann, R.: Behaviour-Interaction-Priority in Functional Programming Languages: Formalisation and Implementation of Concurrency Frameworks in Haskell and Scala. Master's thesis, Ecole polytechnique fédérale de Lausanne (EPFL) (Jan 2015), http://infoscience.epfl.ch/record/215767
14. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: Self-healing systems. pp. 33–38. ACM (2002)
15. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2), 202–220 (2009)
16. Inverardi, P., Wolf, A.L.: Formal specification and analysis of software architectures using the chemical abstract machine model. Software Engineering, IEEE Transactions on 21(4), 373–386 (1995)
17. Kacem, M.H., Jmaiel, M., Kacem, A.H., Drira, K.: Evaluation and comparison of ADL based approaches for the description of dynamic of software architectures. In: ICEIS (3). pp. 189–195 (2005)
18. Kim, J.S., Garlan, D.: Analyzing architectural styles. Journal of Systems and Software 83(7), 1216–1235 (2010)
19. Launchbury, J., Elliott, T.: Concurrent orchestration in Haskell. SIGPLAN Not. 45(11), 79–90 (Sep 2010), http://doi.acm.org/10.1145/2088456.1863534
20. Le Métayer, D.: Describing software architecture styles using graph grammars. Software Engineering, IEEE Transactions on 24(7), 521–533 (1998)
21. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21(6), 3–14 (Oct 1996), http://doi.acm.org/10.1145/250707.239104
22. Marlow, S. (ed.): Haskell 2010 language report. Haskell.org (2010), https://www.haskell.org/definition/haskell2010.pdf
23. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An Overview of the Scala Programming Language. Tech. rep., EPFL (2004), http://infoscience.epfl.ch/record/52656
24. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, pp. 24–52. Springer (1995)
25. Yorgey, B.: The typeclassopedia. The Monad.Reader 13, 17–68 (2009)

A Benchmarks

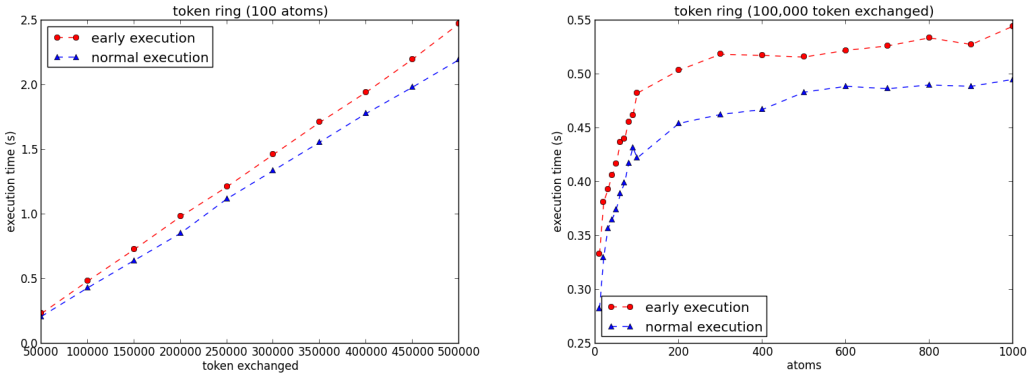
A.1 Token Ring

We developed a token ring implementation in Haskell using the functional BIP implementation. We benchmarked its performance with respect to a varying number of atoms and a varying number of token exchanges.

```
-- | Describes a token ring.
--
--   * @n@ is the number of atoms.
--
--   * @m@ is the number of token exchanges.
--
--   * @display@ indicates if atoms should print to standard output or not.
tokenRing :: Int -> Int -> Bool -> System s ()
tokenRing n m display = mdo

   -- Port on which to send the token and its recipient.
   send <- newPort

   -- Port on which to receive the token.
   receive <- newPort

   -- Vector of atom identifiers
   let v = fromList as

   -- Returns the identifier of the atom that is
   -- supposed to receive message from the i^th atom.
   let next i | i == pred n = a
              | otherwise = v ! i

   -- Behavior of the i^th atom.
   let atom i = do
         -- Awaiting to receive a value.
         n <- await receive ()
         -- Printing the value received.
         when display $ liftIO $ do
            putStrLn $ "Atom " ++ show i ++ " received " ++ show n
         -- Computing the next value.
         let !n' = succ n
         -- Sending the value to the next atom.
         when (n' <= m) $ do
            await send (n', next i)
            atom i

   -- Creating the first atom.
```

```
a <- newAtom $ do
    -- Printing the first message.
    when display $ liftIO $ putStrLn "Atom 0 ready to send."
    -- Sending the first message.
    await send (1, next 0)
    -- From now on, acts as a "normal" atom.
    atom 0

-- Creating the n - 1 next atoms.
as <- forM [ 1 .. pred n ] $ \ i ->
    newAtom $ atom i


-- Connector of the system.
registerConnector $ do
    -- Accepts any atom on the port send.
    (value, destination) <- dynamic send
    -- Bind the receiver on the receive port
    -- and send it the given value.
    bind destination receive # receiving value
```

Figure 9 shows the execution time for a varying number of token exchanges and for a varying number of atoms. We tested both the normal execution mode and the early execution mode, in which interactions may be executed even when the system hasn't fully reached a stable state.



(a) Varying the number of exchanges (b) Varying the number of atoms

Figure 9: Performance of the Haskell token ring implementation

Figure 9a show that the execution time scales linearly with the number of token exchanges, as is to be expected. Figure 9b shows that performance decreases more rapidly for the first few atoms, and then stabilizes. This behavior is exhibited because of the data structures used in the engine and because of the underlying

Haskell concurrent runtime system. In both cases, the early interaction execution mode showcases worse performance due to its overhead. In this particular system, stable states are immediately reached and thus no interactions may be executed in advance.

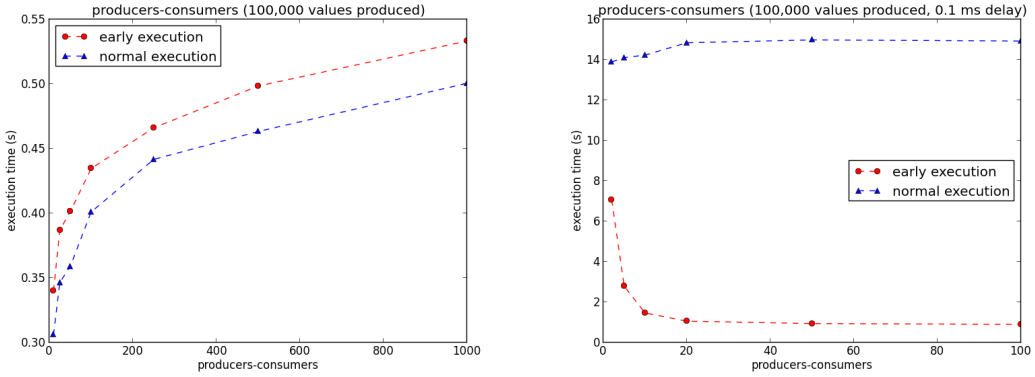
A.2 Producers-Consumers

To showcase the importance of the early interaction execution mode, we implemented a system of multiple producers and consumers and added a varying time cost for producing a value.

```
-- | Producers-consumers system.
--
--   * @n@ is the number of producers and consumers.
--
--   * @k@ is the number of values produced.
--     It should be a multiple of @n@.
--
--   * @s@ is the cost in microseconds to produce a value.
producersConsumers :: Int -> Int -> System s ()
producersConsumers n k s = do
   -- Creation of the two ports.
   send    <- newPort  -- Port on which to send values produced.
   request <- newPort  -- Port on which to receive values.

   -- Creation of the producers.
   replicateM n $ newAtom $ replicateM (k `div` n) $ do
      when (s > 0) $ liftIO $ threadDelay s
      await send ()

   -- Creation of the consumers.
   replicateM n $ newAtom $ forever $ await request ()

   -- The connector of the system
   registerConnector $
      dynamic send  -- Picks any producer
      <*
      dynamic request  -- Picks any consumer
```

Figure 10 shows the execution time for varying number of producers and consumers. We tested both the normal execution mode and the early execution mode. In each case, we tested both without any delay to produce a value and with a delay of 1ms.

Figure 9a exhibits the same behavior as in the previous benchmark. Figure 9b shows that performance can be increased tremendously by having the early interaction execution mode enabled. As the interactions described by this particular connector consist of a single producer and a single consumer, the



(a) Values produced without delay

(b) Values produced with 1ms delay

Figure 10: Performance of the Haskell producers-consumers implementation

system exhibits little to no parallelism in normal execution mode. On the other hand, in early execution mode, the execution time initially decreases with the number of producers and consumers, showing the benefit of parallel execution.