Typed Connector Families*

José Proença[1,2] and Dave Clarke[3]

[1] HASLab/INESC TEC, Universidade do Minho, Portugal
[2] iMinds-DistriNet, Dept Computer Science, KU Leuven, Belgium
[3] Dept. Information Technology, Uppsala University, Sweden
jose.proenca@cs.kuleuven.be dave.clarke@it.uu.se

Abstract. Typed models of connector/component composition specify interfaces describing ports of components and connectors. Typing ensures that these ports are plugged together appropriately, so that data can flow out of each output port and into an input port. These interfaces typically consider the direction of data flow and the type of values flowing. Components, connectors, and systems are often parameterised in such a way that the parameters affect the interfaces. Typing such *connector families* is challenging. This paper takes a first step towards addressing this problem by presenting a calculus of connector families with integer and boolean parameters. The calculus is based on monoidal categories, with a dependent type system that describes the parameterised interfaces of these connectors. As an example, we demonstrate how to define n-ary Reo connectors in the calculus. The paper focusses on the structure of connectors—*well-connectedness*—and less on their behaviour, making it easily applicable to a wide range of coordination and component-based models. A type-checking algorithm based on constraints is used to analyse connector families, supported by a proof-of-concept implementation.

1 Introduction

Software product lines provide the flexibility of concisely specifying a family of software products, by identifying common features of functionality among these products and automatising the creation of products from a selection of relevant features. Interesting challenges in this domain include how to specify families and combinations of features, how to automatise the creation process, how to identify features from a collection of products, and how to reason about (e.g., verify) whole families of products.

This paper investigates such variability in coordination languages, i.e., it studies *connector families* that exogenously describe how (families of) components are connected. The key problem is that different connectors from a single family can have different interfaces, i.e., different ways of connecting to other connectors. Hence, specifying and composing such families of connectors while guaranteeing that interfaces still match becomes non-trivial.

* This research is supported by the FCT grant SFRH/BPD/91908/2012.

Consider, for example a component c that produces 3 values, and a family of connectors ∇_n that merge n values into a single output. We say the interface of c has 3 output ports, and the interface of each ∇_n has n input ports and 1 output port. This paper provides a calculus to compose such n-ary connectors while guaranteeing that all their ports can be properly connected. For example, "$c\,;\,\nabla_3$" denotes the sequential composition of c and a merger with 3 inputs, connecting the output ports of the first to the input ports of the second, resulting in a well-connected connector with 0 inputs and 1 outputs.

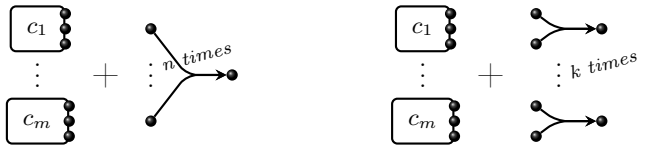


Fig. 1. Example of the composition of connectors.

Fig. 1 exemplifies more complex compositions of n-ary connectors. The left presents the composition of m parallel instances of the component c, written as c^m, with a merger with n inputs. This composition yields a new connector that, given some n and m values, produces a new connector with a single (output) port. This paper provides a type system that checks if such n and m values exist, and their relation: n must be 3 times larger than m. More formally, the connector is written as $\lambda m : \mathbb{N}, n : \mathbb{N} \cdot (c^m\,;\,\nabla_n)$, and the type system yields both the type $\forall m : \mathbb{N}, n : \mathbb{N} \cdot \mathsf{0} \to \mathsf{1}$ and the constraint $n = m * 3$. This means that both the connector and the type are parameterised by two numbers m and n, the connector has type $\mathsf{0} \to \mathsf{1}$, and $n = m * 3$ must hold for the connector to be well typed. The right example of Fig. 1 shows a variation of this example, where the instances of c are composed with k instances of a binary merger ∇_2. The type of the composed connector is $\forall m : \mathbb{N}, k : \mathbb{N} \cdot \mathsf{0} \to k$ constrained by $3 * m = 2 * k$, which means that $3 * m = 2 * k$ must hold for the connector to be well typed, yielding a connector with 0 inputs and k outputs. By writing this connector as $\lambda m : \mathbb{N}, k : \mathbb{N} \cdot (c^{2*m}\,;\,\nabla_2^k)$ the type becomes $\forall m : \mathbb{N}, k : \mathbb{N} \cdot \mathsf{0} \to 3 * m$, constrained by $k = 3 * m$.

To increase compositionality, parameterised connectors can also be composed. Hence $(\lambda m : \mathbb{N} \cdot c^m)\,;\,(\lambda n : \mathbb{N} \cdot \nabla_n)$ has the same type as the left composition of Fig. 1. Finally, extra constraints can be added to parameterised connectors. For example, $\lambda m : \mathbb{N} \cdot (c^m \mid_{m \leq 10})$ represents a parameterised connector that can have at most 10 instances of the connector c. We call *connector families* such connectors that can be parameterised, constrained, and composed.

Summarising, the main contributions of this paper are:

- a calculus for families of connectors with constraints;
- a type system to describe well-defined compositions of such families; and
- a constraint-based type-checking algorithm for this type system.

Connectors are defined incrementally. We start by defining a basic connector calculus for composing connectors inspired by Bruni et al.'s connector algebra [5,3] (Section 2). This calculus is then extended with parameters and expressions, over both integers and booleans (Section 3), being now able to specify connectors (and interfaces) that depend on input parameters. Both the basic and the extended calculus are accompanied by a type system; the latter is an extension of the former, allowing integer and boolean parameters (and effectively becoming a dependent type system). Section 4 introduces *connector families*, by explicitly incorporating constraints over the parameters, and by lifting the composition of connectors to the composition of constrained and parameterised connectors. Section 5 describes an algorithm to type-check connector families with untyped ports, i.e., when the type flowing over each port is not relevant, and presents our prototype implementation. This paper wraps up with related work (Section 6), conclusions and future work (Section 7).

2 Basic Connector Calculus

This section describes an algebraic approach to specify connectors (or components) with a fixed interface, that is, with a fixed sequence of input and output ports that are used to send and receive data. The main goal of this algebraic approach is to describe the structure of connectors and not so much their behaviour. We illustrate the usage of this algebra by using Reo connectors [2], which have well-defined semantics, although our approach can be applied to any connector-like model that connects entities with input and output interfaces.

We start by presenting an overview of how to specify connectors using our calculus. We then describe the syntax of the basic connector calculus and a type system to verify if connectors are well-connected, followed by a brief discussion on how to describe the semantics of connectors orthogonally to this calculus.

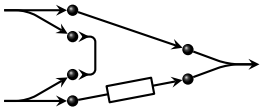
2.1 Overview

Our *basic connector calculus* is based on monoidal categories—more specifically on traced monoidal categories [13]—where connectors are morphisms, ";" is the composition of morphisms with identity id, and "$\otimes$" is the tensor product. The operator "$\otimes$" composes connectors in parallel, while the operator ";" connects the ports of the given connectors. Objects of this category are *interfaces*, which correspond to ports in our connectors and include the unit of the tensor product represented by $\mathsf{0}$. The commutativity of the tensor product is captured by a family of symmetries that swap the order of ports in parallel. Loops can be represented via *traces*, which plug part of the right interface to the left interface of the same connector.

The connector in Table 1 helps understanding the intuition behind our algebra of connectors. Our algebra is inspired by the graphical notation used for monoidal categories (see, e.g., Selinger's survey [13]), and by Bruni et al.'s connector algebra [5,3]. The Reo connector on the left is composed out of smaller

subconnectors, connected with each other via shared ports (●). The second column describes a possible representation of the same connector, writing the names of each subconnector parameterised by its ports. For example, the connector '⤙⤚' is written as $\mathsf{sdrain}(a, b)$ to mean that it has two ports named a and b. Composing connectors is achieved via the $\bowtie$ operator, which connects ports with the same names – this is the most common way to compose Reo connectors in the literature. In this paper we will use instead the algebraic representation on the right of Table 1, where port names are not necessary. The connector $\Delta \otimes \Delta$, for example, puts two duplicator channels in parallel, yielding a new connector with 2 input ports and 4 output ports. This can be composed via ";" with $\mathsf{id} \otimes \mathsf{sdrain} \otimes \mathsf{fifo}$ because this connector has 4 input ports: both the id and the fifo channels have one input port and the sdrain has 2 input ports.

Table 1. Specification of the alternator connector with port names and algebraically.

Graphical	With port names	Algebraic term
	$\Delta(a, a_1, a_2) \bowtie \Delta(b, b_1, b_2) \bowtie$ $\mathsf{sdrain}(a_2, b_1) \bowtie$ $\mathsf{id}(a_1, c_1) \bowtie \mathsf{fifo}(b_2, c_2) \bowtie$ $\nabla(c_1, c_2, c)$	$\Delta \otimes \Delta;$ $\mathsf{id} \otimes \mathsf{sdrain} \otimes \mathsf{fifo};$ ∇

2.2 Syntax

The syntax of connectors and interfaces of our basic connector calculus is presented in Fig. 2. Each connector has a signature $I \to J$ consisting of an input interface I and an output interface J. For example, the identity connector id_I has the same input and output interface I, written $\mathsf{id}_I : I \to I$. Ports of an interface are identified simply with a capital letter, such as A, which capture the type of messages that can be sent via that port. In our examples we assume that A can only be the type $\mathsf{1}$, which represents any port type. This more specific model is also exploited in our algorithm for constraint solving (later in Section 5).

$c ::= c_1 \,;\, c_2$	sequential composition	
$\mid c_1 \otimes c_2$	parallel composition	
$\mid \mathsf{id}_I$	identity connectors	
$\mid \gamma_{I,J}$	symmetries	
$\mid \mathsf{Tr}_I(c)$	traces	
$\mid p \in \mathcal{P}$	primitive connectors	

$p \in \mathcal{P} ::= \Delta_I$	duplicator with output I	
$\mid \nabla_I$	merger with input I	
$\mid \mathsf{sdrain}$	synchronous drain	
$\mid \mathsf{fifo}$	buffer	
$\mid \ldots$	user-defined connectors	
$I, J ::= I \otimes J$	tensor	
$\mid \mathsf{0}$	empty interface	
$\mid A$	port type	

Fig. 2. Connectors (left), primitive connectors (top-right), interfaces (bottom-right).

The intuition of these connectors becomes clearer with the visual representations exemplified in Fig. 3. All connectors are depicted with their input interface on the left side and the output interface on the right side. Each *identity* connector id_I has the same input and output interface I; each *symmetry* $\gamma_{I,J}$ swaps the top interface I with the bottom interface J, hence it has input interface $I \otimes J$ and output interface $J \otimes I$; and each *trace* $\mathsf{Tr}_I(c)$ creates a loop from the bottom output interface I of c with the bottom input interface I of c, hence if c has input interface $I' \otimes I$ and output interface $J' \otimes I$ then the trace has input and output interaces I' and J', respectively.

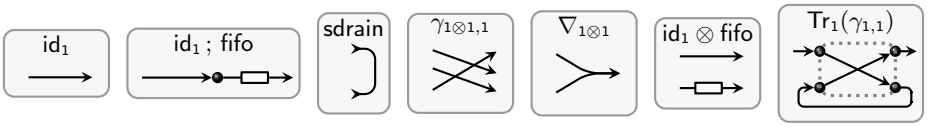


Fig. 3. Visual representation of simple connectors.

Parallelism is represented by tensor products, plugging of connectors by morphism composition, swapping order of parameters by symmetries, and loops by traces. Connectors and types obey a set of *Equations for Connectors* that allow their algebraic manipulation and capture the intuition behind the above mentioned representations. Fig. 4 presents *some* of these equations, which reflect properties of traced monoidal categories. For example, the fact that two symmetries in sequence with swapped interfaces are equivalent to the identity connector, or how the trace of the symmetry $\gamma_{1,1}$ is also equivalent to the identity.

$$
\begin{array}{rcl}
\mathsf{id}_I \;;\; c = c &=& c \;;\; \mathsf{id}_J \quad (\text{if } c : I \to J) \\
\gamma_{I,J} \;;\; \gamma_{J,I} &=& \mathsf{id}_{I \otimes J} \\
(c_1 \otimes c_2) \otimes c_3 &=& c_1 \otimes (c_2 \otimes c_3) \\
0 \otimes I = I &=& I \otimes 0 \\
(I_1 \otimes I_2) \otimes I_3 &=& I_1 \otimes (I_2 \otimes I_3)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{Tr}_I(\gamma_{I,I}) &=& \mathsf{id}_I \\
\mathsf{Tr}_0(c) &=& c \\
c_1 \;;\; \mathsf{Tr}_I(c_2) &=& \mathsf{Tr}_I(c_1 \otimes \mathsf{id}_I \;;\; c_2) \\
\mathsf{Tr}_I(c_1) \;;\; c_2 &=& \mathsf{Tr}_I(c_1 \;;\; c_2 \otimes \mathsf{id}_I) \\
\mathsf{Tr}_I(\mathsf{Tr}_J(c)) &=& \mathsf{Tr}_{I \otimes J}(c)
\end{array}
$$

Fig. 4. Equations for Connectors – based on properties of traced monoidal categories.

2.3 Type rules

Every connector c has an input interface I and an output interface J, written $c : I \to J$. We call these two interfaces the *type* of the connector. Every primitive has a fixed type, for example, $\mathsf{fifo} : 1 \to 1$ and $\nabla_{1\otimes 1} : 1 \otimes 1 \to 1$. The typing rules for connectors (Fig. 5) reflect the fact that two connectors can only be composed sequentially if the output interface of the first connector matches the input interface of the second one. A connector is well-connected if and only if it is well-typed.

$$\frac{\vdash c_1 : I_1 \to J \quad \vdash c_2 : J \to J_2}{\vdash c_1 \,;\, c_2 : I_1 \to J_2}\ (\text{sequence}) \qquad \frac{\vdash c_1 : I_1 \to J_1 \quad \vdash c_2 : I_2 \to J_2}{\vdash c_1 \otimes c_2 : I_1 \otimes I_2 \to J_1 \otimes J_2}\ (\text{parallel}) \qquad \frac{\vdash c : I_1 \otimes J \to I_2 \otimes J}{\vdash \mathsf{Tr}_J(c) : I_1 \to I_2}\ (\text{trace})$$

$$\frac{}{\vdash \gamma_{I,J} : I \otimes J \to J \otimes I}\ (\text{sym}) \qquad \frac{}{\vdash \mathsf{id}_I : I \to I}\ (\text{id}) \qquad \frac{p : I \to J \in \mathcal{P}}{\vdash p : I \to J}\ (\text{prim})$$

Fig. 5. Type rules for basic connectors.

For example, using these type rules it is possible to infer the type of the connector $\mathsf{Tr}_{1\otimes 1}(\gamma_{1\otimes 1,1}\,;\,(\mathsf{fifo}\otimes\mathsf{fifo}\otimes\mathsf{fifo}))$ to be $1 \to 1$, but no type could be inferred after removing one occurence of fifo. This connector is chaining in sequence 3 parallel fifo connectors.

The type rules from Fig. 5 rely on the syntactic comparison of interfaces, e.g., rule (sequence) allows c_1 and c_2 to be composed only if the output interface J of c_1 is syntactically equivalent to the input interface of c_2. To support more complex notions of interfaces we use the constraint-based type rules from Fig. 6, which explicitly compare interfaces that must be *provably equivalent* instead of syntactically comparing them. Rules (sym), (id), and (prim) remain the same, only with the context. The typing judgments now include a context $\Gamma \mid \phi$ consisting both of a set of typed variables Γ (that will only be used in the next section) and a set of constraints ϕ that must hold for the connector to be well-typed. The context must be always well-formed, i.e., Γ cannot have repeated variables and ϕ must have at least one solution, but for simplicity we do not include these global restrictions in the type rules.

$$\frac{\Gamma \mid \phi \vdash c_1 : I_1 \to J_1 \quad \Gamma \mid \phi \vdash c_2 : I_2 \to J_2}{\Gamma \mid \phi, J_1 = I_2 \vdash c_1 \;;\; c_2 : I_1 \to J_2}\ (\text{sequence}) \qquad \frac{\Gamma \mid \phi \vdash c_1 : I_1 \to J_1 \quad \Gamma \mid \phi \vdash c_2 : I_2 \to J_2}{\Gamma \mid \phi \vdash c_1 \otimes c_2 : I_1 \otimes I_2 \to J_1 \otimes J_2}\ (\text{parallel})$$

$$\frac{\Gamma \mid \phi \vdash c : J_1 \to J_2}{\Gamma \mid \phi, J_1{=}X_I{\otimes}I, J_2{=}X_J{\otimes}I \vdash \mathsf{Tr}_I(c) : X_I \to X_J}\ (\text{trace})$$

Fig. 6. Constraint-based type rules.

2.4 Connector behaviour

Semantics for the behaviour of connectors can be given in various ways. For this paper we use the Tile Model [7], as it aligns closely with the algebraic presentation of connectors. We also use the Reo coordination language—more specifically its context independent semantics [3]—as the behaviour of our primitive connectors, whose visual representation has been being used.

We use the same ideas from the Tile Model proposed for Reo [3], using a variation of the category used to describe connectors. Each connector in the Tile Model consists of a set of tiles, one for each possible behaviour, as exemplified in Fig. 7. Each of these tiles contains 4 objects of a double category (two categories with the same objects) and four morphisms between pairs of objects. Visually, a tile is depicted as a square with an object in each corner and with morphisms on the sides of this square. These morphisms go from left to right and from top to bottom: horizontal morphisms are from one category, describing the construction of a connector, and the vertical morphisms are from another category, describing the evolution in time of the connector. More specifically, horizontal morphisms are connectors as specified in Fig. 2, and objects are interfaces. Vertical morphisms are either flow, noFlow, or a tensor product of these, representing a step where data flows over the ports where the flow morphism is applied, and data does not flow over the ports where noFlow is applied.

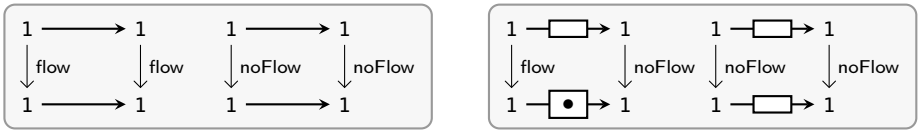


Fig. 7. Tiles for the behaviour of the id_1 (left) and the empty fifo (right) connectors.

Tiles can be composed vertically or horizontally when their adjacent morphisms match, or composed in parallel using the tensor product $\oplus$. Note that two morphisms being the same also implies that their domain and codomain must be the same (i.e., the source and destination of the arrows). The rest of this paper will focus on the horizontal composition of connectors, i.e., on the structural composition of connectors, and not on the behaviour of connectors—the vertical composition. This focus also makes the results presented here more easily applicable to any other coordination or component model where connectors or components have a set of interfaces that can be composed using our calculus.

3 Parameterised Connector Calculus

Connectors are now extended in two ways: (i) by adding integer and boolean expressions to control n-ary replication and conditional choice, and (ii) by adding free variables that can be instantiated with either natural numbers or booleans. These variables are also used in the connector types, previously written as $I \to J$, which are now given by the grammar:

$$T \quad ::= \quad I \to J \;\mid\; \forall x : P \cdot T$$

where x is a variable and $P \in \{\mathbb{N}, \mathbb{B}\}$ represents a primitive type that can be either the natural numbers ($\mathbb{N}$) or booleans ($\mathbb{B}$).

This section introduces the extended syntax and some of its properties, describes motivating examples, and extends the type rules for the connector types described above with boolean and integer parameters.

3.1 Syntax

The extended syntax of connectors and interfaces with integers and booleans is defined in Fig. 8. We write c^{α} instead of $c^{x \leftarrow \alpha}$ when x is not a free variable in c.

$c ::= \ldots$	connectors	$I ::= \ldots$	interfaces
$\mid\ c^{x \leftarrow \alpha}$	n-ary parallel replication	$\mid\ I^{\alpha}$	n-ary parallel replication
$\mid\ c_1 \oplus^{\phi} c_2$	conditional choice	$\mid\ I \oplus^{\phi} J$	conditional choice
$\mid\ \lambda x : P \cdot c$	parameterised connector		
$\mid\ c(\phi)$	bool-instantiation	α, β	integer expressions
$\mid\ c(\alpha)$	int-instantiation	ϕ, ψ	boolean expressions

Fig. 8. Extended syntax of connectors (left) and interfaces (right).

This paper does not formalise integer and boolean expressions with typed variables, since the details of these expressions are not relevant. The semantics of the n-ary parallel replication, the conditional choice, and the instantiation of parameters[4] is captured by the new Equations for Connectors in Fig. 9. These equations include a new notation—$c[v/x]$—that stands for the substitution of all variables x in c that appear freely (i.e., not bounded by a λ quantifier) by the expression v. This paper does not formalise free variables nor substitution, which follow the standard definitions.

$$
\begin{array}{rcl@{\qquad}rcl}
c^{x \leftarrow \alpha} & = & = c[0/x] \otimes \ldots \otimes c[\alpha{-}1/x] & I^{\alpha} & = & I \otimes \ldots \otimes I \quad (\alpha \text{ times}) \\
c_1 \oplus^{true} c_2 & = & c_1 & I_1 \oplus^{true} I_2 & = & I_1 \\
c_1 \oplus^{\phi} c_2 & = & c_2 \oplus^{\neg \phi} c_1 & I_1 \oplus^{\phi} I_2 & = & I_2 \oplus^{\neg \phi} I_1 \\
(\lambda x : P \cdot c)(v) & = & c[v/x] & & &
\end{array}
$$

Fig. 9. Equations for Connector – replication, choice, and instantiation.

Although this extension allows an n-ary composition in parallel of connectors and not in sequence, n-ary sequences of connectors can also be expressed by using traces, as exemplified in the general sequence of fifo connectors on the top-left corner of Fig. 10. We write expressions such as $n-1$ instead of the interface $\mathbf{1}^{n-1}$ for simplicity, when it is clear that these expressions represent interfaces. Observe that this example has been mentioned in the end of Section 2.3, for the specific case of 3 fifos in sequence, already defined using traces and parallel replication. The bottom example is more complex, and is based on the sequencer connector

[4] Known as β-reduction in lambda calculus.

found in Reo-related literature [2]. This connector forces n (synchronous) streams of data to alternate between which one has dataflow. It uses the zip and unzip connectors to combine γ connectors (symmetries) in order to regroup sequences of pairs into a pair of sequences and vice-versa. The top-right corner instantiates the zip connector to illustrate the overal idea; the visual representation unfolds the trace, used to produce a sequence of connectors (as in *seq-fifo*).

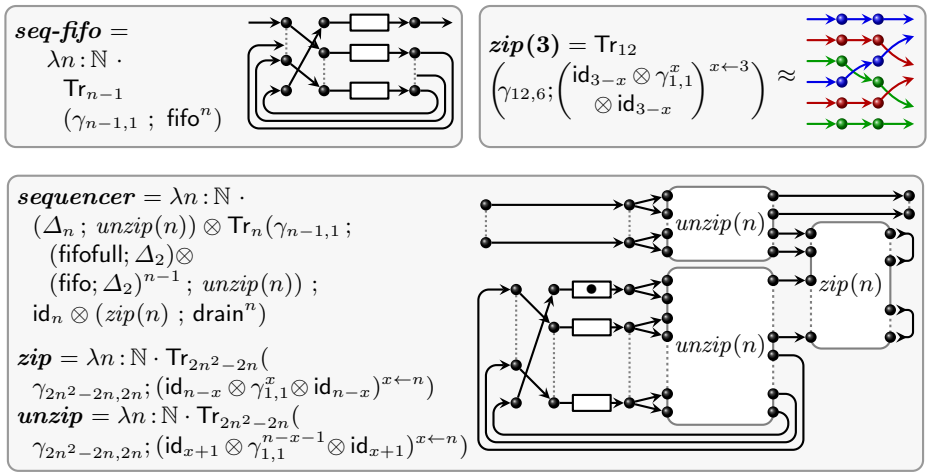


Fig. 10. A sequence of n fifo connectors (top-left), an instance of the *zip* connector (top-right), and an n-ary sequencer connector (bottom).

The details about the behaviour of the sequencer connector are out of the scope of this paper. However, observe that the visual representation is no longer precise enough, since the dotted lines only help to provide intuition but do not specify completely the connector. The parameterised calculus, on the other hand, precisely describes how to build a n-ary sequencer for any $n \geq 0$.

3.2 Parameterised type rules

The extended type rules are presented in Fig. 11, which now use the context Γ consisting of a set of variables and their associated primitive type ($\mathbb{B}$ or $\mathbb{N}$).

As mentioned before, the context cannot contain repeated variables, but this restriction is omitted from the type rules. The actual verification of the type of the boolean and integer variables is done during the type-checking of boolean and integer expressions, which is well known and not defined in this paper. Hence the new type rules have some gray premises, corresponding to the type rules for booleans and integer expressions. The typing judgment $\Gamma \mid \phi \vdash e : P$ for integer and boolean expressions means that $\Gamma \vdash e : P$ (i.e., the variables in the boolean or integer expression e have the type specified in Γ) in a context where ϕ is

$$\frac{\Gamma, x : P \mid \phi \vdash c : T \quad x \notin \phi}{\Gamma \mid \phi \vdash \lambda x : P \cdot c : \forall x : P \cdot T}\ \text{(parameterisation)} \qquad \frac{\Gamma \mid \phi \vdash v : P \qquad \Gamma \mid \phi \vdash c : \forall x : P \cdot T}{\Gamma \mid \phi \vdash c(v) : T[v/x]}\ \text{(instantiation)}$$

$$\frac{\begin{array}{c}\Gamma \mid \phi \vdash \alpha : \mathbb{N} \qquad \Gamma, x : \mathbb{N} \mid \phi \vdash c : I \to J \\ \phi_1 = \big(X_I = I[0/x] \otimes \ldots \otimes I[\alpha - 1/x]\big) \\ \phi_2 = \big(X_J = J[0/x] \otimes \ldots \otimes J[\alpha - 1/x]\big)\end{array}}{\Gamma \mid \phi, \phi_1, \phi_2 \vdash c^{x \leftarrow \alpha} : X_I \to X_J}\ \text{(replication)}$$

$$\frac{\begin{array}{c}\Gamma \mid \phi \vdash \psi : \mathbb{B} \\ \Gamma \mid \phi \vdash c_1 : I_1 \to J_1 \\ \Gamma \mid \phi \vdash c_2 : I_2 \to J_2\end{array}}{\Gamma \mid \phi \vdash c_1 \oplus^{\psi} c_2 : I_1 \oplus^{\psi} I_2 \to J_1 \oplus^{\psi} J_2}\ \text{(choice)}$$

Fig. 11. Parameterised type rules—$x \notin \phi$ means x does not appear in ϕ. Previous type rules remain unchanged.

satisfiable. The notation $I[e/x]$ denotes the substitution of free occurrences of x in I by the expression e, similarly to the substitution in connectors, also not formalised here. Observe that the constraint ψ in the (choice) rule does not influence the typing of c_1 and c_2. Intuitively, if ψ and $\neg\psi$ was to be added to the context when typing c_1 and c_2, respectively, then very likely one of these branches would have *false* in the context, meaning it could not be typed.

$$\begin{array}{l}\varnothing \mid 1 \otimes (n-1) = 1^n, (n-1) \otimes 1 = X_I \otimes (n-1), 1^n = X_J \otimes (n-1) \\ \quad \vdash\ \lambda n : \mathbb{N} \cdot \mathsf{Tr}_{n-1}(\gamma_{n-1,1}\ ;\ \mathsf{fifo}^n) : \forall n : \mathbb{N} \cdot X_I \to X_J \\ \text{parameterisation}\left[\begin{array}{l} x \notin \big(1 \otimes (n-1) = 1^n, (n-1) \otimes 1 = X_I \otimes (n-1), 1^n = X_J \otimes (n-1)\big) \\ n : \mathbb{N} \mid 1 \otimes (n-1) = 1^n, (n-1) \otimes 1 = X_I \otimes (n-1), 1^n = X_J \otimes (n-1) \\ \quad \vdash\ \mathsf{Tr}_{n-1}(\gamma_{n-1,1}\ ;\ \mathsf{fifo}^n) : X_I \to X_J \\ \text{trace}\left[\begin{array}{l} n : \mathbb{N} \mid 1 \otimes (n-1) = 1^n \\ \quad \vdash\ \gamma_{n-1,1}\ ;\ \mathsf{fifo} : (n-1) \otimes 1 \to 1^n \\ \text{sequence}\left[\begin{array}{l} n : \mathbb{N} \mid \varnothing\ \vdash\ \gamma_{n-1,1} : (n-1) \otimes 1 \to 1 \otimes (n-1) \\ n : \mathbb{N} \mid \varnothing\ \vdash\ \mathsf{fifo}^n : 1^n \to 1^n \end{array}\right. \end{array}\right. \end{array}\right.\end{array}$$

Fig. 12. Derivation tree for the *seq-fifo* connector; contexts are represented grey.

We illustrate the usage of these type rules by building the *derivation tree* for the *seq-fifo* connector (Fig. 12), where we illustrate how to calculate the type of this connector by consecutively applying type rules. At every step of this derivation tree the context is well-formed (Γ has no repeated variables and ϕ is always satisfiable). From the existence of this derivation tree one can conclude that the *seq-fifo* connector is well-typed, and by further analysing the constraints in the context it is possible to simplify the type to $\forall n : \mathbb{N} \cdot 1 \to 1$.

4 Connector families

This section introduces *connector families*: parameterised connectors that can (i) be *restricted* by given constraints ψ, written $c\,|_{\psi}$, and (ii) be *composed* with each

$$\frac{\Gamma \mid \phi \vdash \psi \quad \Gamma \mid \phi, \psi \vdash c : T}{\Gamma \mid \phi \vdash c\,|_{\psi} : T\,|_{\psi}}\ (\text{restriction}) \qquad \frac{\Gamma \mid \phi \vdash c_1 : I_1 \to J_1\,|_{\psi_1} \quad \Gamma \mid \phi \vdash c_2 : I_2 \to J_2\,|_{\psi_2}}{\Gamma \mid \phi \vdash c_1 \otimes c_2 : I_1 \otimes I_2 \to J_1 \otimes J_2\,|_{\psi_1, \psi_2}}\ (\text{parallel})$$

Fig. 13. Adding restrictions to types. Other rules remain almost the same, adapted in a similar way to the (parallel) rule.

other—sequentially, in parallel, via the choice or replication operators, or within traces. These restricted and composable connector families represent families in the same sense as software families in the context of software product lines (SPL) engineering [11]. The added constraints represent the family, which in the SPL community are often derived from feature models.

4.1 Restricted connectors and types

Connectors can now be written as $c\,|_{\psi}$, meaning that the connector c is restricted by the constraint ψ. For example, the connector with at most 5 fifo connectors in parallel can be written as $\lambda n : \mathbb{N} \cdot (\mathsf{fifo}^n\,|_{n \leq 5})$. The type of this connector is written similarly as $\forall n : \mathbb{N} \cdot n \to n\,|_{n \leq 5}$. More formally, types now include these constraints, following the following syntax.

$$T \quad ::= \quad I \to J \quad | \quad \forall x : P \cdot T \quad | \quad T\,|_{\psi}$$

The main type rules are presented in Fig. 13. The new rule (restriction) introduces a constraint ψ from the connector to the context. All other rules are adapted in a similar way to the (parallel) rule, by simply collecting the restriction constraints in the conclusions of the rules. For readability we write 'ψ_1, ψ_2' to denote '$\psi_1 \wedge \psi_2$'. A connector c is now *well-typed* if there is a derivation tree $\varnothing \mid \phi \vdash c : T\,|_{\psi}$ such that $\phi \wedge \psi$ is satisfiable, i.e., ψ has at least one solution that does not contradict at least one solution of ϕ.

The example with a parameterised sequence of fifos from Fig. 12 can be adapted to restrict to sequences of at most 5 fifos, yielding the typing judgment:

$$\varnothing \mid \mathbf{1} \otimes (n-1) = \mathbf{1}^n \ , \ (n-1) \otimes \mathbf{1} = X_I \otimes (n-1) \ , \ \mathbf{1}^n = X_J \otimes (n-1)$$
$$\vdash \lambda n : \mathbb{N} \cdot (\mathsf{Tr}_{n-1}(\gamma_{n-1,1}\ ;\ \mathsf{fifo}^n)\,|_{n \leq 5}) \quad : \quad \forall n : \mathbb{N} \cdot X_I \to X_J\,|_{n \leq 5}$$

The conjunction of the above constraints is satisfiable: the possible solutions map X_I and X_J to $\mathbf{1}$, and map n to any value between 0 and 5. Hence the connector is well-typed.

4.2 Family composition

Parameterised connectors (Section 3) allow integer and boolean expressions to influence the final connector. However, the existing type rules for composing connectors do not describe how to compose connectors with parameters. The type rules in Fig. 14 add support for composing connector families: the composition of two parameterised connectors produces a new connector parameterised by the

$$\text{(fam-parallel)}\quad \frac{\Gamma \mid \phi \vdash c_1 : \forall \overline{x_1 : T_1} \cdot I_1 \to J_1 \mid_{\psi_1} \quad \Gamma \mid \phi \vdash c_2 : \forall \overline{x_2 : T_2} \cdot I_2 \to J_2 \mid_{\psi_2} \quad \overline{x_1} \cap \overline{x_2} = \varnothing}{\Gamma \mid \phi \vdash c_1 \otimes c_2 : \forall \overline{x_1 : T_1}, \overline{x_2 : T_2} \cdot I_1 \otimes I_2 \to J_1 \otimes J_2 \mid_{\psi_1, \psi_2}}$$

$$\text{(fam-sequence)}\quad \frac{\Gamma \mid \phi \vdash c_1 : \forall \overline{x_1 : T_1} \cdot I_1 \to J_1 \mid_{\psi_1} \quad \Gamma \mid \phi \vdash c_2 : \forall \overline{x_2 : T_2} \cdot I_2 \to J_2 \mid_{\psi_2} \quad \overline{x_1} \cap \overline{x_2} = \varnothing}{\Gamma \mid \phi, J_1 = I_2 \vdash c_1 \ ; \ c_2 : \forall \overline{x_1 : T_1}, \overline{x_2 : T_2} \cdot I_1 \to J_2 \mid_{\psi_1, \psi_2}}$$

$$\text{(fam-replication)}\quad \frac{\begin{array}{c}\Gamma \mid \phi \vdash \alpha : \mathbb{N} \\ \Gamma, x : \mathbb{N} \mid \phi \vdash c : \forall \overline{x' : P} \cdot I \to J \mid_{\psi} \\ \phi_1 = \big(X_I = I[0/x] \otimes \ldots \otimes I[\alpha - 1/x]\big) \\ \phi_2 = \big(X_J = J[0/x] \otimes \ldots \otimes J[\alpha - 1/x]\big)\end{array}}{\begin{array}{l}\Gamma \mid \phi, \phi_1, \phi_2 \\ \quad \vdash \ c^{x \leftarrow \alpha} : \forall \overline{x' : P} \cdot X_I \to X_J \mid_{\psi}\end{array}}$$

$$\text{(fam-choice)}\quad \frac{\begin{array}{c}\Gamma \mid \phi \vdash \psi : \mathbb{B} \\ \Gamma \mid \phi \vdash c_1 : \forall \overline{x_1 : T_1} \cdot I_1 \to J_1 \mid_{\psi_1} \\ \Gamma \mid \phi \vdash c_2 : \forall \overline{x_2 : T_2} \cdot I_2 \to J_2 \mid_{\psi_2}\end{array}}{\begin{array}{l}\Gamma \mid \phi \vdash c_1 \oplus^{\psi} c_2 : \forall \overline{x_1 : T_1}, \overline{x_2 : T_2} \ \cdot \\ \qquad I_1 \oplus^{\psi} I_2 \to J_1 \oplus^{\psi} J_2 \mid_{\psi_1, \psi_2}\end{array}}$$

$$\text{(fam-trace)}\quad \frac{\Gamma \mid \phi \vdash c : \forall \overline{x : P} \cdot J_1 \to J_2 \mid_{\psi}}{\Gamma \mid \phi, I_1 = X_I \otimes I, I_2 = X_J \otimes I \vdash \mathsf{Tr}_I(c) : \forall \overline{x : P} \cdot X_I \to X_J \mid_{\psi}}$$

Fig. 14. Type rules for the lifted composition operators of connectors.

parameters of both connectors. We write $\forall \overline{x : P}$ to represent a (possibly empty) sequence of nested pairs $\forall x : P$. Note that connectors without parameters are specific instances of connector families; indeed, the new rules (fam-*) coincide with their simpler counterparts whenever the set of parameters is empty.

For example, both connectors below have the same type: $\forall x_1 : \mathbb{N}, x_2 : \mathbb{N}, x_3 : \mathbb{N} \cdot \mathbf{1}^{x_1} \to \mathbf{1}^{x_2} \otimes \mathbf{1}^{x_3}$, under a context where $\mathbf{1}^{x_1} = \mathbf{1}^{x_2} \otimes \mathbf{1}^{x_3}$. The first composes 3 connector families, while the second is a family that composes 3 connectors.

$$(\lambda x_1 : \mathbb{N} \cdot \mathsf{id}_1^{x_1}) \ ; \ (\lambda x_2 : \mathbb{N} \cdot \mathsf{id}_1^{x_2}) \otimes (\lambda x_3 : \mathbb{N} \cdot \mathsf{id}_1^{x_3}) \qquad \text{(composition of families)}$$

$$\lambda x_1 : \mathbb{N}, x_2 : \mathbb{N}, x_3 : \mathbb{N} \ \cdot \ (\mathsf{id}_1^{x_1} \ ; \ \mathsf{id}_1^{x_2} \otimes \mathsf{id}_1^{x_3}) \qquad \text{(family of compositions)}$$

Observe that the modularity gain with the composition of families is achieved by serialising all input arguments. As a consequence the tensor product $\otimes$ no longer obeys the property $(c_1 \otimes c_2); (c_3 \otimes c_4) = (c_1; c_3) \otimes (c_2; c_4)$ with connector families, since the serialisation of the arguments produces different orders.

5 Solving type constraints

This section describes an algorithm to check if the constraints produced by the type rules are satisfiable; if so, this algorithm also provides an assignment of variables to values or to other variables.

Constraint-based approaches to type-checking are well-known, for example, for the lambda calculus [10, Chapter 22], where constraints are solved using an unification algorithm. However, the unification algorithm used for the lambda calculus is not enough for our calculus, because interfaces can include complex

expressions that cannot be just syntactically compared. Hence our algorithm performs algebraic rewritings, uses an unification algorithm (for the simpler cases), and invokes a constraint solver (for the more complex cases).

We focus only on untyped ports, represented by `1`, which mean that any data can go through these ports. Consequently, interfaces are interpreted as integer expressions, denoting the number of ports, as we will shortly explain.

5.1 Overview

In our type-checking algorithm interfaces are interpreted as integers, by mapping constructors of interfaces to integer operations. For example, $(\!|I \otimes J|\!) = (\!|I|\!) + (\!|J|\!)$ and $(\!|I^\alpha|\!) = (\!|I|\!) * \alpha$, where $(\!|I|\!)$ represents the interpretation of I as an integer. Both the constraints that appear in the context and the constraints that appear in the type are combined, hence producing a type $\forall \overline{x : P} \cdot I \to J \,|_\psi$, where ψ represents the conjunction of these constraints.

We exemplify our approach using the *zip* connector (Fig. 10), restricted to when n is at least 5. The type rules produce the type $\forall n : \mathbb{N} \cdot x_3 \to x_4 \,|_\psi$, where ψ is as follows (after interpreting the interfaces as integer expressions).

$$x_3 + ((2{*}n) * (n{-}1)) = ((2{*}n) * (n{-}1)) + (2{*}n)\ ,\ x_4 + ((2{*}n) * (n{-}1)) = x_2\ ,$$
$$x_1 = \textstyle\sum_{0 \le x < n}(((n{-}x) + (2{*}x)) + (n{-}x))\ ,\ x_2 = \sum_{0 \le x < n}(((n{-}x) + (2{*}x)) + (n{-}x))\ ,$$
$$(2{*}n) + ((2{*}n) * (n{-}1)) = x_1\ \ ,\ \ n < 5$$

Using algebraic laws such as distributivity, commutativity, and associativity of sums and multiplications, the constraints are simplified as follows.

$$x_3 = 2n\ ,\ -2n + 2n^2 + x_4 = x_2\ ,\ x_1 = 2n^2\ ,\ x_2 = 2n^2\ ,\ 2n^2 = x_1\ ,\ n < 5$$

The unification algorithm then produces the sequence of substitutions below, leaving the $n < 5$ constraint to be handled in a later phase.

$$[2n/x_3]\ \circ\ [x_4 + 2n^2 - 2n/x_2]\ \circ\ [2n^2/x_1]\ \circ\ [2n/x_4]$$

The final step is to verify that the remaining constraint ($n < 5$) is satisfiable using a constraint solver, allowing us to conclude that the connector is well-typed. Furthermore, applying the substitution above to the type produced by the type rules gives the most general type $\forall n : \mathbb{N} \cdot 2n \to 2n \,|_{n<5}$. The constraint solver provides a solution, say $\{n \mapsto 0\}$, which can be used to produce an instance of the general type: $0 \to 0$.

5.2 Three-phase solver

This section explains in more detail the three-phase algorithm used to reason about constraints, exemplified in the previous subsection. These phases are performed in sequence, and consist of the *simplification* phase, the *unification* phase, and the *constraint-solving* phase, explained below.

Simplification This first phase prepares the constraints obtained by the type rules to be used by the unification phase. More specifically, it rewrites the con-

straints by applying algebraic laws of sums and multiplications, building a polinomial and manipulating the coefficients. For example, sums like $\sum_{n1 \leq x < n2}(5 * x)$, where $5 * x$ is linear on x, are rewritten into $(5 * n2 + 5 * n1) * (n2 - n1)/2$; to avoid integer divisions the denominator 2 is dropped and the other coefficients are multiplied by 2. Equalities are rewritten to match, if possible, the pattern $x = \alpha$, which is exploited by the unification phase.

Note that the type rules, apart from (restriction), only produce equalities of integer expressions. Our choice of rewrites included in the prototype implementation took into account the constraints generated by the type rules using a range of different connectors. These rewrites are able to simplify all the examples presented in this paper that do not use inequalities, most of which involve only linear expressions or are reduced to linear expressions, to a point where the constraint solving phase was not needed. Furthermore, other fast off-the-shelf technologies, such as computer algebra systems, could be used to quickly manipulate and simplify more complex expressions.

Unification The second phase consists of a traditional unification algorithm [10, Chapter 22] adapted to our type system, which produces both an *unification* and a set of constraints postponed to the constraint solving phase. An unification is formally a sequence of substitutions $\sigma_1 \circ \cdots \circ \sigma_n$, and applying a unification to a connector or interface t consists of applying the substitutions in sequence $((t\,\sigma_1)\ldots)\,\sigma_n$. For example, unifying the constraints $x = 2 + y, z = 3 + x, y = w$ produces the sequence of substitutions $[2 + y/x] \circ [3 + 2 + y/z] \circ [w/y]$. Applying this unification to an interface means first substituting x by $2 + y$, followed by the substitutions of z and y. The resulting interface is guaranteed to have no occurrences of x, y, nor z, and not to have w bound by any constraint.

The unification algorithm is described by the unify function (Fig. 15) that, given a set of constraints ϕ to be solved, returns a pair with a unification and a set of postponed constraints. The core of unify is defined in the right side of Fig. 15. For every equality $\alpha = \alpha'$, it first checks if they are syntactically equivalent (using $\equiv$). It then checks if either the left or the right side is a variable that does not occur on the other side; if so, it adds the equality to the resulting unification. If none of these cases apply, it postpones the analysis of the constraint for the third phase, by using the second argument of unify as an accumulator.

Constraint solving The last phase consists of collecting the constraints postponed by the unification phase and use an off-the-shelf constraint solver. This will tell us if the constraints are satisfiable, producing a concrete example of a substitution that satisfies the constraints. In the example of the sequence of

$$\mathsf{unify}(\phi) = \mathsf{unify}(\phi\,;\, \mathit{true})$$

$$\mathsf{unify}(\mathit{true}, \phi\,;\, \psi) = \mathsf{unify}(\phi\,;\, \psi)$$

$$\mathsf{unify}(\alpha = \alpha', \phi\,;\, \psi) = \begin{cases} \mathsf{unify}(\phi\,;\, \psi) & \text{if } \alpha \equiv \alpha' \\ \mathsf{unify}(\phi[\alpha'/x]\,;\, \psi) \circ [\alpha'/x] & \text{if } \alpha \equiv x \text{ and } x \notin \mathit{fv}(\alpha') \\ \mathsf{unify}(\phi[\alpha/x]\,;\, \psi) \circ [\alpha/x] & \text{if } \alpha' \equiv x \text{ and } x \notin \mathit{fv}(\alpha) \\ \mathsf{unify}(\phi\,;\, \psi, \alpha = \alpha') & \text{otherwise} \end{cases}$$

Fig. 15. Unification algorithm for constraints over boolean and integer variables.

fifos with at most 5 fifos (Section 5.1), a possible solution for the constraints is $\{n \mapsto 4, x_1 \mapsto 1, x_2 \mapsto 1\}$. This substitution, when applied to the type obtained for *seq-fifo*, yields a concrete type instance *seq-fifo* : $\mathtt{1} \to \mathtt{1}$. In this example the concrete type instance matches its general type ($\forall n : \mathbb{N} \cdot \mathtt{1} \to \mathtt{1}$), since the value of n does not influence the type of the connector.

Note that a wide variety of approaches for solving constraints exist. One can use, for example, numerical methods to find solutions, or SMT solvers over some specific theory. The expressive power supported by the constraint solver dictates the expressivity of the expressions α and ϕ used in the connector, which we are abstracting away in this paper. The choices made in our proof-of-concept implementation, briefly explained in the next subsection, are therefore not strict and can be rethought if necessary.

5.3 Implementation

We developed a proof-of-concept implementation in the Scala that covers all the examples described in this paper, which can be found online.[5] Listing 1 exemplifies the usage of this library—more examples can be also found online.

```
import paramConnectors.DSL._
val x = "x":I  ;  val n = "n":I  ;  val b = "b":B

//----- λx:ℕ · (fifo^x |_{x>5}) -----//
typeOf( lam(x, (fifo^x) | (x>5)) )
// returns ∀x:I . x -> x | x > 5
typeInstance( lam(x, (fifo^x) | (x>5)) )
// returns © 6 -> 6
typeSubstitution( lam(x, (fifo^x) | (x>5)) )
// returns © [x:I -> 6]

//----- seq-fifo -----//
typeOf(   lam(x, Tr(x-1, Sym(x-1,1) & (fifo^x))) )
// returns ∀x:I . 1 -> 1  [type obtained only after constraint solving]
typeTree( lam(x, Tr(x-1, Sym(x-1,1) & (fifo^x))) )
// returns ∀x:I . x1 -> x2 | ((x1 + (x - 1)) == ((x - 1) + 1))
//   & ((x2 + (x - 1)) == x) & ((1 + (x - 1)) == x) & (x1 >= 0) & (x2 >= 0)

//----- zip and sequencer -----//
val zip = ....
typeOf( zip )
// returns ∀n:I . 2 * n -> 2 * n
val sequencer = ....
typeOf( sequencer )
// returns ∀n:I . n -> n
```

Listing 1. Calculating the type of connectors using our tools.

[5] https://github.com/joseproenca/parameterised-connectors

This implementation includes a simple domain specific language to specify connectors, making them similar to the syntax used throughout this paper. It provides three main top-level functions: `typeTree`, `typeOf`, `typeInstance`, and `typeSubstitution`. The first creates the derivation tree (if it exists); `typeOf` simplifies the constraints, uses the unification algorithm, invokes the constraint solver, and returns the most general type found; and `typeInstance` and `typeSubstitution` perform the same steps as `typeOf`, but the former returns the result of the constraint solving phase (even if the type is not the most general one) and the latter returns the substitutions obtained by the unification and the constraint solver phases. Hence the result of `typeInstance` never includes constraints. The constraint solving phase uses the Choco solver[6] to search for solutions of the constraints.

Observe that the resulting type instance and substitution of the first connector start with ©—this means that the resulting type is a concrete instance of a type, i.e., the constraint solving phase found more than one solution for the variables of the inferred type (after unification). However, if we would ask for a type instance of $(\lambda x : \mathbb{N} \cdot \mathsf{fifo}^x | x > 5)(7)$, for example, the result would be also its (general) type $7 \to 7$, without the ©. Typing the connector $(\lambda x : \mathbb{N} \cdot \mathsf{fifo}^x | x > 5)(2)$ gives a type error, because the constraints are not satisfied.

6 Related Work

Algebras of connectors The usage of symmetric monoidal categories to represent Reo connectors (and others) has been introduced by Bruni et al. [5], where they introduce an algebra of stateless connectors with an operational semantics expressed using the Tile Model [7]. The authors focus on the behavioural aspects, exploiting normalisation and axiomatisation techniques. An extension of this work dedicated to Reo connectors [3] investigates more complex semantics of Reo (with context dependent connectors) using the Tile Model. Other extensions to connector algebras exist. For example, Sobocinski [14], and more recently Bonchi et al. [4], present stateful extensions to model and reason about the behaviour of Petri Nets and of Signal Flow Graphs, respectively. The latter also describes the usage of traces (Tr) as a possible way to specify loops in their algebra. In all these approaches, interfaces (objects of the categories) can be either input or output ports, independently of being on the left or right side of the connector (morphism), focusing on the behaviour of connectors instead of how to build families of these connectors.

In our work we do not distinguish input from output ports, assuming data always flows from left to right, and use traces to support loops and achieve the same expressivity. As a result, we found the resulting connectors to be easier to read and understand. For example the connector fifo has type $\bullet\circ \to 0$ in Bruni et al.'s algebra, meaning that the left side has 2 ports: an input $\bullet$ and an output $\circ$ one. Composing two fifos in sequence uses extra connectors (called nodes) and

[6] http://choco-solver.org

has type $0 \to \circ\bullet$—for a more complete explanation see [7]. Indeed, our algebra has stronger resemblances with lambda calculus (and with pointfree style in functional programming [8]), facilitating the extension to families of connectors, which is the main novelty of this work.

Analysis of software product lines In the context of software product lines Kästner et al. [9], for example, investigated how to lift a type-checking algorithm from programs to families of programs. They use featherweight Java annotated with constraints used during product generation, and present a type-checking approach that preserves types during this product generation. Their focus is on keeping the constraints being solved as small as possible, unlike previous approaches in the generative programming community (e.g., by Thaker et al. [15]) that compile a larger global set of constraints. Many other verification approaches for software product lines have been investigated [12,1,16,6]. Post and Sinz [12] verify families of Linux device drivers using the CBMC bouned model checker, and Apel et al. [1] verify more general families of C programs using the CPAchecker symbolic checker. More recently Thüm et al [16] presents an approach to use the KeY theorem prover to verify a feature-oriented dialect of Java with JML annotations. They encode such annotated families of Java programs into new (traditional) Java programs with new JML annotations that can be directly used by KeY to verify the family of products. Dimovski et al [6] take a more general view and provide a calculus for modular specification of variability abstractions, and investigate tradeoffs between precision and time when analysing software product lines and abstractions of them.

Our approach targets connector and component interfaces instead of typed languages, and explicitly uses parameters that influence the connectors. Consequently, feature models can contribute not only with feature selections but also with values used to build concrete connectors. Our calculus is simpler than other more traditional programming languages since it has no statements, no notion of heap or memory, nor tables of fields or methods.

7 Conclusion and Future Work

This paper formalises a calculus for connector families, i.e., for connectors (or components) with an open number of interfaces and restricted to given constraints. A dependant type system guarantees well-connectedness of such families, i.e., that interfaces of subconnectors can be composed as long as the parameters obey the constraints in the type. These constraints are reducible to non-linear constraints on integers when considering untyped ports (only the type $\mathtt{1}$), in which case arithmetic properties and integer constraint solvers can be used to check the constraints under which a connector family is well-connected.

In the future we will investigate connector families where the type of the data passing through the ports is also checked. Finally, we also plan to investigate how to reduce the size of the constraints being solved, by using the more dedicated contexts while building the type tree instead of collecting the constraints for a follow-up phase, similarly to the work of Kästner et al. [9].

References

1. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 372–375, Washington, DC, USA, 2011. IEEE Computer Society.
2. F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.
3. F. Arbab, R. Bruni, D. Clarke, I. Lanese, and U. Montanari. Tiles for Reo. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques*, pages 37–55. LNCS 5486, 2009.
4. F. Bonchi, P. Sobocinski, and F. Zanasi. Full abstraction for signal flow graphs. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, POPL '15, pages 515–526, New York, NY, USA, 2015. ACM.
5. R. Bruni, I. Lanese, and U. Montanari. A basic algebra of stateless connectors. *Theor. Comput. Sci.*, 366(1-2):98–120, 2006.
6. A. S. Dimovski, C. Brabrand, and A. Wasowski. Variability abstractions: Trading precision for speed in family-based analyses (extended version). *CoRR*, abs/1503.04608, 2015.
7. F. Gadducci and U. Montanari. Proof, language, and interaction. chapter The Tile Model, pages 133–166. MIT Press, Cambridge, MA, USA, 2000.
8. J. Gibbons. A pointless derivation of radix sort. *Journal of Functional Programming*, 9(3):339–346, 1999.
9. C. Kastner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 258–267, Washington, DC, USA, 2008. IEEE Computer Society.
10. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
11. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. 2005.
12. H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 2008 23rd International Conference on Automated Software Engineering*, ASE '08, pages 347–350. IEEE Computer Society, 2008.
13. P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer Berlin Heidelberg, 2011.
14. P. Sobocinski. Representations of petri net interactions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 554–568. Springer, 2010.
15. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 95–104. ACM, 2007.
16. T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 11–20, New York, NY, USA, 2012. ACM.