Transposing partial components—An exercise on coalgebraic refinement

Luís S. Barbosa*, José N. Oliveira

Departamento de Informática, Di-CCTC, Universidade do Minho, Portugal

Abstract

A partial component is a process which fails or dies at some stage, thus exhibiting a finite, more ephemeral behaviour than expected. Partiality—which is the rule rather than exception in formal modelling—can be treated mathematically via totalization techniques. In the case of partial functions, totalization involves error values and exceptions.

In the context of a coalgebraic approach to component semantics, this paper argues that the behavioural counterpart to such functional techniques should extend behaviour with try-again cycles preventing from component collapse, thus extending totalization or transposition from the algebraic to the coalgebraic context.

We show that a refinement relationship holds between original and totalized components which is reasoned about in a coalgebraic approach to component refinement expressed in the pointfree binary relation calculus.

As part of the pragmatic aims of this research, we also address the factorization of every such totalized coalgebra into two coalgebraic components—the original one and an added front-end—which cooperate in a client-server style.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Partial components; Try-again cycles; Refinement; Coalgebra

1. Introduction

Partial functions (also termed simple relations) arise in mathematics and programming wherever their output is undefined for some input data. Programmers have learnt to deal with this situation by enriching the codomain of such functions with a special error mark indicating that nothing is output. In C/C++, for instance, this leads to functions which output pointers to values rather than just values. In functional languages such as Haskell [8], this has to do with functions which output Maybe-values rather than values, where Maybe is datatype Maybe \( a = \text{Nothing} | \text{Just} \ a \).

This effort towards functional totalization can be regarded as the addition of an interface shielding partial functions against inputs which lead to undefinedness. Instead of failing or dying, the evaluation of such shielded functions raises exceptions in a monadic style. From a data refinement perspective, every such totalized function can be regarded as a fully functional implementation arising from codomain pointer-reification [20].

The equivalent of this situation in a coalgebraic setting—partial coalgebras—leads to processes which fail or die at some stage, thus exhibiting a finite, more ephemeral behaviour than expected (e.g., operating system crash). Software

* Research supported by FCT (the Portuguese Foundation for Science and Technology), in the context of the PURE Project under contract POSI/CHS/44304/2002.

* Corresponding author. Tel.: +351 253604463; fax: +351 253604471.

E-mail addresses: lsb@di.uminho.pt (L.S. Barbosa), jno@di.uminho.pt (J.N. Oliveira).
designers have known to overcome process premature death by shielding services with *interfaces* which prevent from reaching dead states. Formally, this means to enrich the underlying coalgebra with a *try-again* behavioural alternative (signalling out some error message, in practice) such as is usual in e.g., command-line interpreters and human–computer interfacing.

In this paper, we approach such *try-again*-totalized coalgebras as behavioural counterparts of *maybe*-transposed-functions [21], thus extending totalization or transposition from the algebraic to the coalgebraic context.

We show that a refinement relationship holds between the original and the totalized components which can be expressed in the coalgebraic approach to component refinement developed by Meng and Barbosa [17,18]. In particular, the transposition of partial components arises as an example of *backward* refinement—a relation dual to the generalization of the more usual case of nondeterminism reduction studied in [18]. As an improvement, such a reasoning is carried out in the pointfree binary relation calculus [2,9], which is shown to compare favourably—for its elegance and effectiveness—with respect to its more widespread pointwise counterpart. Thanks to such agile notation and calculus, this paper provides calculations which come in support of the refinement preorders whose introduction in [17,18] was only intuitively motivated. Therefore, another main contribution of this paper is a significant extension and re-working of such an approach to component refinement.

As part of the pragmatic aims of this research, we address the bisimilarity between every such a shielded component and its factorization into two coalgebraic components—the original one and an added interface—which cooperate in a *client-server* style. This can be regarded as an abstract formulation of the *Seeheim principle* [23] (also known as the *separation principle*), which is consensual in up-to-date interactive software design.

**Paper structure:** The section which follows introduces the notion of a partial component and how it does arise in practice. Partial component transposing and the refinement relationship between partial and transposed components are presented in Section 3. The underlying theory of coalgebraic refinement is given in Section 4, and the proof of the main result of the paper is given in Section 5. The bisimilarity between every transposed component and its factorization into two coalgebraic components (client and server) are the main subjects of Section 6, which paves the way for the conclusions and pointers to future work.

2. What is a partial component?

**Partial modelling:** In the tradition of mathematical modelling in physics and other branches of science, constructive formal specification methods, such as VDM [16,11], Z [26] or B [1], are based on the notion of a software *formal model*. This is understood as a state-based abstract machine which models how a system reacts to input stimuli, changes state and yields output. Paying tribute to the nowadays widespread object-oriented programming principles, formal models in specification languages such as e.g., VDM++ [12] and Z++ [10] are encapsulated into abstract *objects*. These offer a number of services—e.g., PUSH and POP in the stack model of Fig. 1, written in VDM++ notation—through a public interface which provides limited access to a private state space—e.g., instance variable stack in the same model.

Regarded as state-based, dynamic systems, such formal models (objects, abstract machines or components) belong to the broad group of computing phenomena whose semantics are essentially *observational*, in the sense that all that can be traced of their evolution is their *interaction* with the environment. *Coalgebras* [25], i.e., functions of type \( \tau : \mathcal{U} \leftarrow \mathcal{U} \) for \( \tau \) a parametric datatype appear as suitable mathematical devices in *explaining* the semantics of such formal state-based models.

Coalgebra theory has been subject to recent, remarkable developments [25]. Refs. [5,6] present a coalgebraic approach to the semantics of state-based software components under the *components as coalgebras* slogan. Let us see the “approach at work” by (constructively) deriving a coalgebraic semantics for class stackObj of Fig. 1.

We first note that the semantics of PUSH is a function of type

\[
[\text{PUSH}] : S \times 1 \leftarrow S \times A
\]

where S abbreviates Stack and 1 (the singleton datatype) abbreviates () in VDM++. Similarly, the semantics of POP will exhibit signature

\[
[\text{POP}] : S \times A \leftarrow S \times 1
\]
class stackObj

types
  public Stack = seq of A ;
  public A = token :

instance variables
  stack : Stack := [];

operations

  public PUSH : A => ()
  PUSH(a) => stack := [a] ^ stack;

  public POP : () => A
  POP() => def r = hd stack
     in ( stack := tl stack;
         return r)
  pre stack <> [];

end stackObj

Fig. 1. VDM++ model of a stack.

class unOrdCol

types
  public Collection = set of A ;
  public A = token :

instance variables
  col : Collection := {};

operations

  public PUT : A => ()
  PUT(a) => col := {a} ^ union col;

  public GET : () => A
  GET() => let r in set col
     in ( col := col \ {r};
         return r)
  pre col <> {};

end unOrdCol

Fig. 2. VDM++ model of a collection.

However, \[\text{POP}\] is not a (total) function because of its precondition. We should thus interpret the arrows above as denoting partial functions.

In practice, reactive partiality is more the rule than the exception in formal modelling. But there is more. Fig. 2 models an unordered collection in terms of finite sets. The two models (stackObj and unOrdCol) are similar in shape. However, method GET (the counterpart of POP in Fig. 1) is not only partial but also nondeterministic, since sets are unordered and there is no such notion as the first or last element of a set. All in all, the arrows above have to be regarded as denoting binary relations, a concept which encompasses both total and partial functions as special cases [9].
We now focus on \texttt{unOrdCol}, whose nondeterministic semantics is more interesting than that of \texttt{stackObj}. The idea of \textit{packaging} methods \texttt{PUT} and \texttt{GET} together in a public interface is nicely captured by \textit{summing up} the two relations which capture their semantics,

\[
\| \text{PUT} \| + \| \text{GET} \| : S \times 1 + S \times A \leftarrow S \times A + S \times 1
\]

where \(+\) denotes relational \textit{coproduct} (vulg. datatype sum) and \(S\) is kept as the symbol denoting the model’s internal state. Thanks to the distributivity of \(\times\) through \(+\), we can factor out \(S\), leading to

\[
dr^\circ \cdot (\| \text{PUT} \| + \| \text{GET} \|) \cdot \text{dr} : S \times (1 + A) \leftarrow S \times (A + 1)
\]

where \(\text{dr}\) is the \textit{distribute-right} isomorphism and \(\text{dr}^\circ\) denotes its converse.\(^2\) Knowing that every binary relation \(R\) can be converted into a (set-valued) function \(AR\) via the \textit{power-transpose} isomorphism \([9,21]\) defined by

\[
f = A R \equiv (b Ra \equiv b \in f a)
\]

for all \(R : B \leftarrow A\) and \(f : \mathcal{P} B \leftarrow A\) (\(\mathcal{P} B\) denotes the set of all subsets of \(B\)), we convert relation \(\text{dr}^\circ \cdot (\| \text{PUT} \| + \| \text{GET} \|) \cdot \text{dr}\) into function

\[
A(\text{dr}^\circ \cdot (\| \text{PUT} \| + \| \text{GET} \|) \cdot \text{dr}) : \mathcal{P}(S \times (1 + A)) \leftarrow S \times (A + 1)
\]

which—finally—can be \textit{curried} into coalgebra

\[
A(\text{dr}^\circ \cdot (\| \text{PUT} \| + \| \text{GET} \|) \cdot \text{dr}) : \mathcal{P}(S \times (1 + A))^{(A + 1)} \leftarrow S
\]

where the bar over the coalgebra denotes the currying isomorphism which is such that, given binary function \(g\), equivalence

\[
f = \overline{g} \equiv (\forall a, b : (f a)b = g(a, b))
\]

holds.

In general, the semantics of a (nondeterministic) component \(p\) hiding internal state \(U_p\) and offering \(n\) methods \(M_{i=1,n}\) of public interface \(M_i : O_i \leftarrow I_i\) will be captured by coalgebra

\[
\overline{A \left( \text{dr}^\circ \cdot \left( \sum_{i=1}^n \| M_i \| \right) \cdot \text{dr} \right)}
\]

mapping \(U_p\) into \(TU_p = \mathcal{P}(U_p \times O)^I\), where \(O\) abbreviates \(\sum_{i=1}^n O_i\), \(I\) abbreviates \(\sum_{i=1}^n I_i\) and (for simplicity) \(\text{dr}\) is assumed extended to the \(n\)-ary case.

\textit{Going generic}: The above construction raises two observations. First, initialization statements such as \texttt{col := \{} in Fig. 2 have not been accounted for so far. They provide a specification of the initial value of the component’s state, i.e., the \textit{seed} from which all subsequent behaviour of the underlying coalgebra will be computed.

Second, a truly generic model for software components should not restrict itself to nondeterministic behaviour, as captured by the powerset construction above. Other components will exhibit different \textit{behaviour models}; as shown in [4–6], \textit{genericity} is achieved by replacing the \textit{powerset} monad by an arbitrary \textit{strong monad} \(^3\) \(B\). In this paper our

\footnotetext[1]{In the sequel, both functional and relational composition will be denoted by the same symbol \(\cdot\) given that the former is just a special case of the latter. Relational composition is defined in the usual way: \(b(R \times S)c\) holds wherever there exists some mediating \(a\) such that \(b Ra \land a Sc\) holds. In the case of functions, say \(f\) and \(g\) instead of \(R\) and \(S\), \(b(f \circ g)c\) equates \(b = f(g(c))\). Here—and elsewhere in this paper—we follow the fairly common notation standard of denoting (total) functions by lowercase letters \((f, g, \text{etc.})\) and all other relations by uppercase letters \((e.g., R, S)\).}

\footnotetext[2]{In general, the converse \(R^\circ\) of binary relation \(R\) is such that \(b Ra \equiv a R^\circ b\).}

\footnotetext[3]{A \textit{strong monad} is a monad \((B, \eta, \mu)\) where \(B\) is a strong functor and both \(\eta\) and \(\mu\) are strong natural transformations. \(B\) being strong means that there exist natural transformations \(\tau^B_l : B(\text{id} \times -) \leftarrow B \times -\) and \(\tau^B_r : B(- \times \text{id}) \leftarrow - \times B\) are called the right and left strength, respectively, subject to certain conditions. Their effect is to \textit{distribute} the free variable values in the context \(\langle - \rangle\) along functor \(B\). Strength \(\tau_l\), followed by \(\tau_l\) maps \(B I \times B J\) to \(B(I \times J)\), which can, then, be flattened to \(B(I \times J)\) via \(\mu\). In most cases, however, the \textit{order} of application is relevant for the outcome. The Kleisli composition of the right with the left strength, gives rise to a natural transformation whose component on objects \(I\) and \(J\) is given by \(\delta_{I,J} = \tau_{J,I} \circ \mu_{I,J}\) Dually, \(\delta_{I,J} = \tau_{I,J} \circ \mu_{I,J}\). Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of \(B\)-computations. Whenever \(\delta_l\) and \(\delta_l\) coincide, the monad is said to be \textit{commutative}.}
attention will be focused on behavioural models \( \mathcal{B} \) which incorporate a notion of possible failure, therefore modelling what we understand by a partial component. Such is trivially the case of the maybe monad. However, also note that \( \mathcal{P} A \cong 1 + \mathcal{P}_+ A \), where \( \mathcal{P}_+ A \) is the set of all nonempty subsets of \( A \).

Thus, we reach the generic notion of a component \( p \) with input interface \( I \) and output interface \( O \), denoted by \( p : O \leftarrow I \), as follows: it is specified as a (pointed) coalgebra in \( \text{Set} \)

\[
(\alpha_p \in U_p, \bar{\alpha}_p : \mathcal{B}(U_p \times O)^I \leftarrow U_p),
\]

where point \( \alpha_p \) is the ‘initial’ or ‘seed’ state. Therefore, the computation of an action will not simply produce an output and a continuation state, but a flow of events to a subset of inputs of the same component. Thus, arbitrary communication between components is achieved by first aggregating them via one of the tensors and then selecting the input and output points to be connected by the feedback operator. A particular case of feedback will be presented in Section 6.

A component calculus: Regarding software components as pointed coalgebras parametric on a behaviour model \( \mathcal{B} \), gives rise to a rich semantic framework where components become arrows in a (bicategorical) universe \( \mathcal{C}_p \) whose objects are sets, providing types to input/output parameters (the components’ interfaces). Component morphisms \( h : q \leftarrow p \), which impose a categorical structure on the corresponding homsets, amount basically to coalgebra morphisms. Such a framework was proposed in a series of papers beginning with [4], in the context of which a component calculus \([5,6]\) was developed also in a generic way, i.e., parametric on monad \( \mathcal{B} \).

This calculus involves component assembly patterns such as pipeline (\( ; \)) and three tensors capturing: respectively, external choice (\( \boxplus \)), parallel (\( \boxtimes \)) and concurrent (\( \boxplus \)) composition. For the purpose of this paper, it is enough to understand the semantics of pipelining and external choice. Let \( p : O \leftarrow I, q : R \leftarrow J \) and \( r : R \leftarrow O \) be components. Then \( p ; r : R \leftarrow I \) is a component whose coalgebra \( a_{p;r} : \mathcal{B}(U_p \times U_r \times R) \leftarrow U_p \times U_r \times I \) sequences the behaviour of \( a_p \) and \( a_r \) by feeding the latter with the output of the former, while the monadic effect is propagated (see Fig. 3). Concerning external choice, when interacting with \( p \boxplus q : O + R \leftarrow I + J \), the environment chooses either to input a value of type \( I \) or of type \( J \), which triggers the corresponding component (\( p \) or \( q \), respectively), producing the relevant output (see Fig. 3). The formal definition of the underlying coalgebra of type \( a_{p \boxplus q} : \mathcal{B}(U_p \times U_q \times (O + R)) \leftarrow U_p \times U_q \times (I + J) \), which captures this behaviour, can be found in [6].

Generalized interaction is catered through a sort of ‘feedback’ mechanism connecting a specified subset of outputs to a subset of inputs of the same component. Therefore, arbitrary communication between components is achieved by first aggregating them via one of the tensors and then selecting the input and output points to be connected by the feedback operator. A particular case of feedback will be presented in Section 6.

Finally, component adaptation is captured by a wrapping combinator and function lifting [6], a combinator which promotes a function \( f : B \leftarrow A \) to component \( f^{-} : B \leftarrow A \) whose coalgebra (over \( I \), the singleton type) is defined by \( a^{-} f^{-} = \eta \cdot (id \times f) \). This allows for component composition with arbitrary functions.
3. Transposing partial components

Once partial operations (such as POP or GET in the examples above) are offered by a given component, its overall behaviour will certainly include the possibility of failure. At component level, failure of a particular operation leads to the whole component collapsing. Moreover, as any other behavioural effect, failure propagates through any component network to which the failing component is Kleisli-composed (which is always the case because all component combinators in [5,6] involve forms of Kleisli-like composition).

However, as hinted earlier on in this paper, a more “positive” approach to behavioural partiality would be a try-again behaviour rather than overall collapse. This section discusses how a partial component, in the sense of Definition 3.1 below, can be transformed (or transposed) into a component with identical behaviour but for the possible failure cases, which are transformed into stuttering states: the transposed component will wait and remain accepting all invalid inputs while making no move, until a valid input is effectively processed.

Definition 3.1. A component $p$ such as given by (3) is said to be partial whenever the associated behaviour model $B$ can be decomposed in a maybe shape,

$$B \cong B_+ + 1$$

as witnessed by a given natural isomorphism $\zeta_B : B_+ + 1 \leftrightarrow B$. We denote by $R_p$ the simple relation (vulg. partial function) of type $B_+ (U_p \times O) \leftrightarrow U_p \times I$, of which $a_p$ is the maybe-transpose $[21]$, that is, $a_p = \Gamma R_p$ and $R_p = i_1^\top \cdot a_p$, where isomorphism $\Gamma$ is uniquely defined by universal property

$$f = \Gamma R \equiv R = i_1^\top \cdot f.$$  

(5)

For instance, $B_+$ is the identity monad $id$ for $B$ the “Maybe” monad and the nonempty powerset monad $P_+$ for $B = P$.

Let us now address the transposition of a partial component $p$ into a try-again (total) one, to be denoted by $p \uparrow$. Let us start with an example. Suppose $p$ is a nondeterministic component whose behaviour is expressed by powerset-coalgebra $a_p$. Its expected try-again counterpart will be

$$a_p\uparrow \left\langle u, i \right\rangle = \begin{cases}  
\text{if } a_p\langle u, i \rangle = \{\} & 
\text{then } \{\langle u, \text{Nothing} \rangle\} \\
\text{else } \{\langle u', \text{Just } o \rangle | \langle u', o \rangle \in a_p\langle u, i \rangle\}. 
\end{cases}$$

In general, the dynamics of $p$ is pre-composed with the diagonal function $\Delta a = \langle a, a \rangle$ which replicates the current value of the state-space. Whenever $a_p$ fails, $p$ is able to recover from such a value. Formally,

Definition 3.2. Given partial component $p : O \leftarrow I$, its try-again transpose $p \uparrow : O + 1 \leftarrow I$ is defined by a new coalgebra over the same state space whose dynamics is given by

$$a_p\uparrow = \begin{array}{c}
\begin{array}{c}
\Delta \times id \\
id \times a_p \\
i_{\top} \times B(U_p \times O) \\
i_{\top} \times \zeta_B \\
\pi_2 + id \\
i_{\top} (id \times i_1) + id \times i_2 \\

\end{array}
\end{array} U_p \times (U_p \times O + 1) \stackrel{dr}{\longrightarrow} (U_p \times U_p) \times I \stackrel{a}{\longrightarrow} U_p \times (U_p \times I) \stackrel{id \times ap}{\longrightarrow} U_p \times B(U_p \times O) \stackrel{id \times \zeta_B}{\longrightarrow} B_+(U_p \times O + 1) + U_p \times 1 \stackrel{\pi_2 + id}{\longrightarrow} B_+(U_p \times O) + U_p \times 1 \stackrel{B_+(id \times i_1) + id \times i_2}{\longrightarrow} B_+(U_p \times (O + 1)) + U_p \times (O + 1) \stackrel{[i_1, \zeta_B \cdot \eta_B]}{\longrightarrow} B_+(U_p \times (O + 1)) + 1 \stackrel{\zeta_B}{\longrightarrow} B(U_p \times (O + 1))$$

$^4$Further to $dr$ already introduced in this paper, the second step in the composition chain involves isomorphism $a$ which witnesses product association to the right. Function $id$ is the identity function such that $id \; a = a$ for all $a$. 


Note in this definition the role of \( dr \) in distributing the initial state across the \( \text{maybe-shape} \) is to be kept later (via \( \pi_2 + \text{id} \), where \( \pi_2 \) denotes the right projection of product) only on the failure side of this shape. The second-to-last step involves construction \([R, S]\) (read \( \text{either R or S} \)) which is given by closed formula,

\[
[R, S] = (R \cdot i_1^\alpha) \cup (S \cdot i_2^\beta),
\]

where \( i_1 \) and \( i_2 \) are the coproduct injections.

The signature extension, from \( p : O \leftarrow I \) to \( p \uparrow : O + 1 \leftarrow I \), resembles the \( \text{maybe-transpose} \) of partial functions [21]. At behaviour level, output of type \( 1 \) bears the informal meaning "please try again." In a sense, behavioural partiality is absorbed by an extension towards data-partiality at the (output) data level.

In general, \( p \) and \( p \uparrow \) are not bisimilar. Regarded as (generalized) transition systems, however, the underlying coalgebras have the same structure but for the presence, in the latter, of reflexive arrows at every \textit{partial state}, i.e., every state at which failure is a possibility at least for an argument. These correspond, as one would expect, to the \textit{try-again} extra-behavioural cycles.

**Lemma 3.1.** Component \( p \uparrow : O + 1 \leftarrow I \) is a backward refinement of \( p : O \leftarrow I \), with respect to the structural failure refinement order \( \leq F_T \) of [17].

The statement of this lemma calls for further explanation. First of all, it makes use of a notion of behaviour refinement proposed in a previous paper, [17], in which refinement is captured by the existence of some form of \textit{weak} coalgebra morphism (just as bisimulation amounts to the existence of a standard morphism) with respect to a particular refinement preorder. The latter, on its turn, exploits the structure of the coalgebra dynamics in a number of different ways (leading, correspondingly, to a number of refinement preorders). Finally, such preorders can be used in two dual ways referred to in [17] as \textit{forward} or \textit{backward} refinement. Ref. [17] is further expanded into a journal version [18] in which the component calculus of [5,6] is extended with a number of generic refinement laws. Both [17,18] are, however, mainly concerned with \textit{forward} refinement, which generalizes the usual axis of \textit{nondeterminism reduction} in a functorial way.

As we shall show in the sequel, \textit{backward} refinement corresponds to a similar functorial generalization of \textit{definition increase} and turns out to be the right way of characterizing the relationship between \( p \) and \( p \uparrow \) and proving Lemma 3.1. First, however, we have to recall the refinement theory in which this lemma lives. Such is the purpose of the following section which not only collects the main concepts from [17], but also reframes its main constructions in a more general \textit{pointfree} way. Such a reconstruction, on its turn, not only largely increases the calculational power, but also provides a formal justification of some definitions which were in [17] only intuitively motivated. This, which we regard as a main contribution of the present paper, can be appreciated in the (relational) proof of Lemma 3.1, which is deferred to Section 5.

### 4. Behavioural refinement by pointfree calculation

The starting point of [17] is that, just as transition systems can be coded back as coalgebras, any coalgebra \( \langle U, \alpha : T \leftarrow U \rangle \) specifies a \( (T\text{-shaped}) \) transition structure over its carrier \( U \). For extended polynomial \textit{Set} endofunctors, \(^5\) such a structure may be expressed as a binary relation \( \alpha \leftarrow : U \leftarrow U \), defined in terms of the \textit{structural membership} relation \( \in_T : U \leftarrow T \leftarrow U, \)

\[
u \alpha \leftarrow u \equiv u' \in_T \alpha \ u
\]

which can be written in less symbols as

\[
\alpha \leftarrow = \in_T \cdot \alpha
\]

---

\(^5\) This is the class inductively defined as the least collection of functors containing the identity \( \text{Id} \) and constant functors \( \text{K} \) for every object \( K \) in the category, closed by functor composition and finite application of product, coproduct, covariant exponential and finite powerset functors.
at pointfree level. Relation $\varepsilon_T$ is defined by induction on the structure of polynomial $T$ as follows:

\[
\begin{align*}
\varepsilon_{\text{Id}} &= \text{id} \\
\varepsilon_{\bot} &= \bot \\
\varepsilon_{T_1 \times T_2} &= (\varepsilon_{T_1} \cdot \pi_1) \cup (\varepsilon_{T_2} \cdot \pi_2) \\
\varepsilon_{T_1 + T_2} &= [\varepsilon_{T_1}, \varepsilon_{T_2}] \\
\varepsilon_{T_1 \cdot T_2} &= \varepsilon_{T_2} \cdot \varepsilon_{T_1} \\
\varepsilon_T^k &= \bigcup_{k \in K} \varepsilon_T \cdot \beta_k \text{ (where } \beta_k f = f k) \\
\varepsilon_p &= \varepsilon \text{ (set-theoretic membership)}
\end{align*}
\]

Some comments on notation follow. Since $\text{id}$ is the identity function and $\bot$ is the empty relation, $x \in_{\text{Id}} y \iff x = y$ is the pointwise counterpart of the first clause and $x \in_{\bot} y \iff \text{false}$ is that of the second. The pointwise expansion of the third clause is $x \in_{T_1 \cdot T_2} y \iff x \in_{T_1} \pi_1 y \lor x \in_{T_2} \pi_2 y$, where $\pi_1, \pi_2$ denote the left and right projections of relational product. Formula (6) and rule

\[
b(f \circ R \cdot g)a \iff (f b)R(g a)
\]

are useful in the pointwise-pointfree conversion of the fourth clause into

\[
x \in_{T_1 + T_2} y \iff \begin{cases} 
  y = i_1 y' \implies x \in_{T_1} y' \\
  y = i_2 y' \implies x \in_{T_2} y'
\end{cases}
\]

The fifth clause is self-explanatory. Finally, the exponentials clause generalizes the third, and the last clause brings in conventional set-theoretic membership.

Relation $\varepsilon_T$ is actually an instance of datatype membership defined by Hoogendijk [13] as a Galois connection, which entails two results which are of interest to this paper. First, that $\varepsilon_T$ satisfies the following naturality condition:

\[
h \cdot \varepsilon_T = \varepsilon_T \cdot T h
\]

for any function $h$. Second, that $\varepsilon_T$ can be further extended to the construction of recursive datatypes. In brief, this goes as follows: let $TA$ be the type functor induced by a given base polynomial bifunctor $B$, that is, $\text{in} : TA \leftarrow B(A, TA)$ is an isomorphism. Let $\varepsilon_1, \varepsilon_2$ be the two memberships associated to the two places of bifunctor $B$, that is, $\varepsilon_1 : X \leftarrow B(X, Y)$ and $\varepsilon_2 : Y \leftarrow B(X, Y)$. Relation $\text{Atroot} = \varepsilon_1 \cdot \text{in}^a$ (of type $\text{Atroot} : A \leftarrow TA$) can be understood as checking whether a particular $a$ of type $A$ can be found at root-level of a given “tree” $t$ of type $TA$, while $\text{Branch} : TA \leftarrow TA$, defined by $\text{Branch} = \varepsilon_2 \cdot \text{in}^a$, checks whether some other $t'$ is a branch of $t$. Rather elegantly, Hoogendijk calculates type functor membership, $\varepsilon_T : A \leftarrow TA$, as follows:

\[
\varepsilon_T = \text{Atroot} \cdot \text{Branch}^a
\]

where $\text{Branch}^a$ denotes the reflexive, transitive closure of $\text{Branch}$. So, $a \in_T t$ means that $a$ can be found at the root of either $t$ or any of its branches, at any depth.

4.1. Forward/backward refinement

The dynamics of a component $p : O \leftarrow I$ is based on functor $B(\text{Id} \times O)'$. Therefore, a possible (and intuitive) way of regarding component $p$ as a behavioural refinement of some other component $q : O \leftarrow I$ is to consider that $p$-transitions are simply preserved in $q$. For nondeterministic components, this is understood simply as set inclusion. But one may also want to consider additional restrictions. For example, to stipulate that if $p$ has no transitions from a given state, $q$ should also have no transitions from the corresponding state(s). Or one may adopt the dual point of view, requiring transition reflection instead of preservation. In any case the basic question remains: how can such a refinement situation be identified?

In data refinement, there is a “recipe” to identify a refinement situation: look for an abstraction function to witness it. In other words: look for a morphism in the relevant category, from the ‘concrete’ to the ‘abstract’ model such that the latter can be recovered from the former up to a suitable notion of equivalence, though, typically, not in a unique way.
In our components’ framework, however, things do not work this way. The reason is obvious: component morphisms are (seed preserving) coalgebra morphisms which are known (see e.g., [25]) to entail bisimilarity. Therefore, we have to look for a somewhat \textit{weaker} notion of a morphism between coalgebras.

Recall that a \(T\)-coalgebra morphism \(h: \alpha \hookrightarrow \beta\) is a function from the state space of \(\beta\) to that of \(\alpha\) such that

\[
T h \cdot \beta = \alpha \cdot h
\]

holds. Regarding \(\beta\) and \(\alpha\) as (generalised) transition systems, Eq. (18) becomes relational equality

\[
h \cdot \beta \hookrightarrow = \alpha \hookrightarrow \cdot h
\]

(thanks to (7), (16) and the fact that \((\in_T \bullet)\) is an isomorphism [21]), i.e., the conjunction of inclusions

\[
h \cdot \beta \hookrightarrow \subseteq \alpha \hookrightarrow \cdot h
\]

\[
\alpha \hookrightarrow \cdot h \subseteq h \cdot \beta \hookrightarrow
\]

By shunting,\(^6\) inclusion (20) is equivalent to

\[
\beta \hookrightarrow \subseteq h^\circ \alpha \hookrightarrow \cdot h
\]

Inequalities (22) and (21) take a more familiar shape once variables are introduced:

\[
v' \beta \hookrightarrow v \Rightarrow h v' \alpha \hookrightarrow h v
\]

\[
u' \alpha \hookrightarrow h v \Rightarrow \exists v' \in V \cdot v' \beta \hookrightarrow v \land u' = h v'
\]

They jointly state that, not only \(\beta\) dynamics, as represented by the induced transition relation, is \textit{preserved} by \(h\) (20, 23), but also \(\alpha\) dynamics is \textit{reflected} back over the same \(h\) (21, 24). Is it possible to weaken the morphism definition to capture only one of these aspects?

The answer is yes and resorts to the notion of a preorder \(\leq\) on a \(\text{Set}\) endofunctor \(T\). This is defined in [14] as a functor \(\leq\) which makes the following diagram commute:

\[
\xymatrix{
\text{Set} & \ar[l]_{\text{PreOrd}} \ar[r]^{(T V, \leq_{TV})} & \text{Set} \\
V \ar[u]_T & \ar[l]_{\leq} & TV \ar[u]_{T}
}
\]

This means that for any function \(h: U \hookrightarrow V\), \(Th\) preserves the order, i.e.,

\[
x_1 \leq_{TV} x_2 \Rightarrow (Th) x_1 \leq_{TU} (Th) x_2
\]

or, in a pointfree formulation,

\[
(Th) \cdot \leq_{TV} \subseteq \leq_{TU} \cdot (T h)
\]

In the definition which follows subscripts are dropped, e.g., \(\leq\) instead of \(\leq_{TV}\), for notation economy. Moreover, we denote by \(\leq\) the pointwise lifting of preorder \(\leq\) to the functional level, i.e.,

\[
f \leq g \equiv (\forall x : \cdot f x \leq g x)
\]

which can easily be shown to have the following pointfree-equivalent:

\[
f \leq g \equiv f \leq \leq \cdot g
\]

**Definition 4.1.** Let \(T\) be an extended polynomial functor on \(\text{Set}\) and consider two \(T\)-coalgebras \(\beta: TV \hookrightarrow V\) and \(\alpha: TU \hookrightarrow U\). A forward morphism \(h: \alpha \hookrightarrow \beta\) with respect to a preorder \(\leq\) is a function from \(V\) to \(U\) such that

\[
T h \cdot \beta \leq \alpha \cdot h
\]

---

\(^6\) In the relational calculus [3], Galois connections \(f \cdot R \subseteq S \equiv R \subseteq f^\circ \cdot S\) and \(R \cdot f^\circ \subseteq S \equiv R \subseteq S \cdot f\), involving function \(f\) and relations \(R\) and \(S\), are known as \textit{shunting rules}.
Dually, \( h \) is said to be a backward morphism if
\[
\pi \cdot h \preceq T h \cdot \beta
\]

The following lemma, a pointfree proof of which can be found in [18], shows that such morphisms compose and can be taken as witnesses of refinement situations:

**Lemma 4.1.** For \( T \) an endofunctor in \( \text{Set} \), \( T \)-coalgebras and forward (respectively, backward) morphisms define a category.

Such a split of a coalgebra morphism into two conditions makes it possible to capture separately transition preservation and reflection. Lemma 4.2 will state that forward morphisms preserve transitions whereas backward morphisms reflect them. To prove this, however, the following extra condition has to be imposed on preorder \( \preceq \) to express its compatibility with the membership relation: for all \( x \in X \) and \( x_1, x_2 \in TX \),
\[
x \in T x_1 \land x_1 \preceq x_2 \Rightarrow x \in T x_2
\]  
(29)

or, again in a pointfree formulation,
\[
\in_T \cdot \preceq \subseteq \in_T
\]  
(30)

A preorder \( \preceq \) on an endofunctor \( T \) satisfying inclusion (30) will be referred to, in the sequel, as a **refinement preorder**. Then,

**Lemma 4.2.** Let \( T \) be an extended polynomial functor in \( \text{Set} \), and \( \beta \) and \( \pi \) be two \( T \)-coalgebras as above. Let \( \beta \leftarrow \) and \( \pi \leftarrow \) denote the corresponding transition relations. A backward (respectively, forward) morphism \( h : \pi \leftarrow \beta \) reflects (respectively, preserves) such transition relations.

**Proof.** Let \( h \) be a backward morphism. Transition reflection, defined by Eq. (21), is established as follows:
\[
\pi \leftarrow \pi \cdot h
\]
\[
= \{ \text{definition (7)} \}
\]
\[
\in_T \cdot \pi \cdot h
\]
\[
\subseteq \{ \text{\( h \) backwards entails \( \pi \cdot h \preceq \cdot \pi \cdot T \cdot \beta \), monotonicity} \}
\]
\[
\in_T \cdot \pi \preceq \cdot \pi \cdot T h \cdot \beta
\]
\[
\subseteq \{ \text{compatibility with \( \in_T \) (30), monotonicity} \}
\]
\[
\in_T \cdot T h \cdot \beta
\]
\[
\equiv \{ \text{\( \in_T \) natural (16)} \}
\]
\[
h \cdot \in_T \cdot \beta
\]
\[
= \{ \text{definition (7)} \}
\]
\[
h \cdot \beta \leftarrow
\]

The forward case is documented in [18]. \( \square \)

The existence of a backward (forward) morphism connecting two components \( p \) and \( q \) (that is, a morphism between coalgebras \( a_p \) and \( a_q \)) witnesses a refinement situation whose symmetric closure coincides, as expected, with bisimulation and defines behaviour refinement by the existence of a backward morphism up to bisimulation. Formally,

**Definition 4.2.** Let \( T \) be the behaviour shape of components \( q = \langle u_q, \overline{a_q} \rangle \) and \( p = \langle u_p, \overline{a_p} \rangle \). Then \( q \) is said to be a backward refinement of \( p \)—written \( p \preceq_T q \)—if there is a (seed preserving) backward morphism \( p \leftarrow^h q \) that is,

\( ^7 \) A similar study is made in [18] on forward refinement.
such that
\[ h \text{ } u_q = u_p \]
\[ \alpha_p \cdot h \leq_T T h \cdot \alpha_q \]
Subscript $T$ is often dropped wherever it is clear from the context, as in e.g., the following obvious fact: $p \sim q \Rightarrow p \leq q$.

4.2. Calculating refinement preorders

The exact meaning of refinement assertion $p \leq q$ above depends, of course, on the concrete refinement preorder adopted. But what do we know about such preorders? Condition (30) equivales

\[ \leq \subseteq \in_{T \setminus T} \]

by direct application of the Galois connection which defines relational division,

\[ R \cdot X \subseteq S \iff X \subseteq R \setminus S \]

from which its pointwise meaning $x \ (R \setminus S) y \equiv (\forall y : yRx \Rightarrow ySz)$ can be inferred [2].

Clearly, (31) provides an upper bound for refinement preorders, the lower bound being $\text{id}$, the smallest preorder. It is well known (see e.g., [13,21]) that relation $\in_{T \setminus T}$ corresponds to the lifting of $\in_T$ to (structural) inclusion, i.e.,

\[ x \ (\in_{T \setminus T}) y \equiv (\forall e : e \in_T x : e \in_T y) \]

Clearly, $\in_{T \setminus T}$ always is a preorder. By (31), it is the largest refinement preorder. For $T = \text{id}$, (31) has only one solution which is easy to calculate:

\[
\begin{align*}
\leq_{\text{id}} & \subseteq \in_{\text{id}} \setminus \in_{\text{id}} \\
\equiv & \{ \text{(32)} \} \\
\in_{\text{id}} \cdot \leq_{\text{id}} & \subseteq \in_{\text{id}} \\
\equiv & \{ \text{membership definition } \in_{\text{id}} = \text{id} \} \\
\leq_{\text{id}} & \subseteq \text{id} \\
\equiv & \{ \text{as a preorder, } \leq_{\text{id}} \text{ is reflexive} \} \\
\leq_{\text{id}} & = \text{id}
\end{align*}
\]

Concerning case $T = K$, $\in_K \setminus \in_K = T$ (where $T$, the “topmost” relation of its type, is such that $x \ T y$ holds for any $x, y$), since $\in_K = \bot$. In our component model, however, such a preorder on the constant functor would make refinement based on $\in_T \setminus \in_T$ blind to the outputs produced. This suggests an additional requirement on the refinement preorders for $Cp$ components: their definition on a constant functor $K$ must be equality on set $K$, i.e., $\leq_K = \text{id}$, so as to leave transitions with different $O$-labels related.

In these two cases, we have chosen the unique and the smallest solutions to (31), respectively. For all other cases, there is a lot more freedom. Let us consider them in sequence.

4.2.1. Products

As above, we start by calculating upper-bound $\in_{T_1 \times T_2} \setminus \in_{T_1 \times T_2}$:

\[
\begin{align*}
\in_{T_1 \times T_2} \setminus \in_{T_1 \times T_2} \\
= & \{ \text{(10)} \} \\
(\in_{T_1} \cdot \pi_1 \cup \in_{T_2} \cdot \pi_2) \setminus \in_{T_1 \times T_2}
\end{align*}
\]

\[8\] Reflexivity: $R \setminus S$ is reflexive iff $R \subseteq S$; transitivity: $R \setminus S$ is transitive wherever $S \subseteq R$. 
In summary, this confirms that definition adopted in [18] is indeed a refinement preorder (the refinement preorder of a product is the product of its factors' refinements preorders).

4.2.2. Coproducts

By analogy with the above, we follow [18] in defining

\[ \leq_{T_1+T_2} \triangleq \leq_{T_1} + \leq_{T_2} \] (35)

but add the verification that such a sum of two membership-compatible preorders is membership-compatible:

\[ \leq_{T_1} + \leq_{T_2} \subseteq (\leq_{T_1+T_2}) \]

\[ \equiv \{ \text{Galos (32)} \} \]

\[ \leq_{T_1+T_2} \leq_{T_1} \leq_{T_2} \subseteq \leq_{T_1+T_2} \]

\[ \equiv \{ \text{membership definition} \} \]

\[ [\leq_{T_1}, \leq_{T_2}] \leq_{T_1} \leq_{T_2} \subseteq [\leq_{T_1}, \leq_{T_2}] \]

\[ \equiv \{ \text{+ fusion} \} \]

\[ [\leq_{T_1}, \leq_{T_2}] \leq_{T_1} \leq_{T_2} \subseteq [\leq_{T_1}, \leq_{T_2}] \]

\[ \equiv \{ \text{‘either’ is monotonic} \} \]

\[ \leq_{T_1} \leq_{T_1} \leq_{T_2} \leq_{T_2} \subseteq \leq_{T_1} \leq_{T_2} \]

\[ \equiv \{ \text{Galos (32) twice} \} \]

Note that \( \leq_{T_1} \leq_{T_2} \subseteq \leq_{T_1 \times T_2} \) holds (similarly for \( T_2, \pi_2 \)). From this we infer:

\[ \leq_{T_1} \leq_{T_2} \subseteq \leq_{T_1 \times T_2} \]

\[ \equiv \{ \text{monotonicity of upper adjoint (} \leq_{T} \text{) in (32)} \} \]

\[ \leq_{T_1} \leq_{T_2} \subseteq \leq_{T_1 \times T_2} \]

\[ \equiv \{ \text{monotonicity of } \leq_{T} \text{ and previous calculation} \} \]

\[ \leq_{T_1} \times \leq_{T_2} \subseteq \leq_{T_1 \times T_2} \]

In summary, this confirms that definition

\[ \leq_{T_1 \times T_2} \triangleq \leq_{T_1 \times T_2} \] (34)

adopted in [18] is indeed a refinement preorder (the refinement preorder of a product is the product of its factors' refinements preorders).
4.2.3. Functor composition

We have \(\leq_{T_1 \cdot T_2} = \leq_{T_2} \cdot \leq_{T_1}\), recall (12). Calculation of \(\leq_{T_1 \cdot T_2}\) proceeds by indirect inclusion:

\[
X \subseteq \leq_{T_1 \cdot T_2} \setminus \leq_{T_1 \cdot T_2} \\
\equiv \{ (12) \} \\\nX \subseteq (\leq_{T_2} \cdot \leq_{T_1}) \setminus \leq_{T_1 \cdot T_2} \\
\equiv \{ \text{Galois (32)} \} \\\n(\leq_{T_2} \cdot \leq_{T_1}) \cdot X \subseteq \leq_{T_1 \cdot T_2} \\
\equiv \{ \text{Galois and (12)} \}
\]

\[
\leq_{T_1} \cdot X \subseteq \leq_{T_2} \setminus (\leq_{T_2} \cdot \leq_{T_1}) \\
\leftarrow \{ \text{property } (R \setminus S) \cdot T \subseteq R \setminus (S \cdot T) \} \\
\leq_{T_1} \cdot X \subseteq (\leq_{T_2} \setminus \leq_{T_2}) \cdot \leq_{T_1} \\
\leftarrow \{ \text{assume } \leq_{T_2} \subseteq \leq_{T_1} \} \\
X \subseteq \leq_{T_1} \setminus (\leq_{T_2} \cdot \leq_{T_1}) \\
\therefore \{ \text{define } \leq_{T_1} \cdot \leq_{T_2} \triangleq \leq_{T_1} \setminus (\leq_{T_2} \cdot \leq_{T_1}) ; \text{indirection} \}
\]

\[
\leq_{T_1} \cdot \leq_{T_2} \triangleq \leq_{T_1} \setminus (\leq_{T_2} \cdot \leq_{T_1}) \\
\equiv \{ \text{Galois (32)} \} \\\nX \subseteq \leq_{T_1} \setminus (\leq_{T_2} \setminus \leq_{T_2}) \cdot \leq_{T_1} \\
\leftarrow \{ \text{choose } k' := k \} \\
X \subseteq \bigcap_{k \in K} (\leq_{T_1} \setminus \leq_{T_2} \cdot \leq_{T_1}) \\
\equiv \{ \text{since } f \cdot \leq_{T_1} \cdot \leq_{T_2} \triangleq \leq_{T_1} \setminus (\leq_{T_2} \cdot \leq_{T_1}) \} \\
X \subseteq \leq_T
\]

We have shown that should \(\leq_{T_2}\) be membership-compatible then so is\(^9\)

\[
\leq_{T_1} \cdot \leq_{T_2} \triangleq \leq_{T_1} \setminus (\leq_{T_2} \cdot \leq_{T_1}) \tag{36}
\]

4.2.4. Exponentials

Calculation of \(\leq_{T^K}\) proceeds by indirect inclusion:

\[
X \subseteq (\leq_{T^K} \setminus \leq_{T^K}) \\
\equiv \{ (13) ; (R \cup S) \cdot T = (R \setminus T) \cap (S \setminus T) \} \\\nX \subseteq \bigcap_{k \in K} ((\leq_{T} \cdot \beta_k) \setminus (\bigcup_{k' \in K} (\leq_{T} \cdot \beta_{k'})) \\
\leftarrow \{ \text{choose } k' := k \} \\
X \subseteq \bigcap_{k \in K} ((\leq_{T} \cdot \beta_k) \setminus (\leq_{T} \cdot \beta_k)) \\
\equiv \{ (R \setminus f) \cdot S = f^* \cdot (R \setminus S) \text{ and } R \setminus (S \cdot f) = (R \setminus S) \cdot f \cdot \text{ for } f := \beta_k \} \\
X \subseteq \bigcap_{k \in K} (\beta_k^2 \cdot (\leq_{T} \setminus \leq_{T})) \cdot \beta_k \\
\leftarrow \{ \text{assume } \leq_T \subseteq \leq_T \} \\
X \subseteq \bigcap_{k \in K} (\beta_k^2 \cdot \leq_T \cdot \beta_k) \\
\equiv \{ \text{since } f \cdot \leq_T \subseteq \leq_T \} \\
X \subseteq \leq_T
\]

So, \(\leq_T\) is membership-compatible wherever \(\leq_T\) is membership-compatible. Thus the definition

\[
\leq_{T^K} \triangleq \leq_T
\tag{37}
\]

chosen in [17].

The preorder definitions so far enable the following result.

**Lemma 4.3.** \(\leq_T = \text{id}\), for every polynomial functor \(T X = \sum_{i=0}^n C_i \times X^i\).

\(^9\) Another much stronger choice could have been

\[
\leq_{T_1} \cdot \leq_{T_2} \triangleq (\leq_{T_1} \setminus \leq_{T_2}) \cdot \leq_{T_1}.
\]
Proof.
\[
\leq \sum_{i=0}^{n} C_i \times X^i
\]
\[
= \{ \text{single out constant functor} \}
\]
\[
\leq C_0 + \sum_{i=1}^{n} C_i \times X^i
\]
\[
= \{ \text{choice } \leq_K = \text{id and } (35) \}
\]
\[
id + \sum_{i=1}^{n} C_i \times X^i
\]
\[
= \{ (34) \}
\]
\[
id + \sum_{i=1}^{n} (\leq C_i \times \leq X^i)
\]
\[
= \{ \text{as above and } (37) \}
\]
\[
id + \sum_{i=1}^{n} (id \times \leq X)
\]
\[
= \{ id = \text{id and } \times, +-\text{reflection} \}
\]
\[
id \square
\]

4.2.5. Powerset
As elsewhere \[21,17\], we define \(\leq_P\) as the maximum set-membership-compatible preorder, that is, set inclusion.

4.2.6. Recursive datatypes
Finally, let us solve (31) for recursive membership (17):
\[
\leq \subseteq \in_T \setminus \in_T
\]
\[
\equiv \{ (17) \text{ twice and } (32) \}
\]
\[
\text{Atroot} \cdot \text{Branch}^* \cdot \leq \subseteq \text{Atroot} \cdot \text{Branch}^*
\]
\[
\Leftarrow \{ \text{monotonicity of composition} \}
\]
\[
\text{Branch}^* \cdot \leq \subseteq \text{Branch}^*
\]
Clearly, \(\leq_T \triangleq \text{Branch}^*\) is a solution to the version of (31) just above: by substitution, one gets \(\text{Branch}^* \cdot \text{Branch}^* \subseteq \text{Branch}^*\), which holds since \(\text{Branch}^*\) is transitive. Altogether, by choosing this solution,
\[
\leq_T \triangleq \text{Branch}^*
\]
(38)

one obtains a quite obvious understanding of recursive datatype inclusion: \(t \leq_T t'\) holds wherever \(t\) is a subtree of \(t'\), at any depth.

4.2.7. Comments
Thanks to the results above, we are now able to easily calculate the compound refinement preorders, as the following calculation of \(\leq_{\mathcal{P}(\text{Id} \times O)}^t\) shows:
\[
\leq_{\mathcal{P}(\text{Id} \times O)}^t
\]
\[
= \{ (37) \}
\]
\[
\leq_{\mathcal{P}^* (\text{Id} \times O)}
\]
All in all, we have justified, by agile pointfree calculation, the following (pointwise) refinement preorder definitions proposed in [17]:

\[
\begin{align*}
  x \leq_{\text{id}} y & \iff x = y \\
  x \leq_{K} y & \iff x =_{K} y \\
  x \leq_{T_1 \times T_2} y & \iff \pi_1 x \leq_{T_1} \pi_1 y \land \pi_2 x \leq_{T_2} \pi_2 y \\
  x \leq_{T_1 + T_2} y & \iff \\
    \begin{cases}
      x = i_1 x' \land y = i_1 y' \Rightarrow x' \leq_{T_1} y' \\
      x = i_2 x' \land y = i_2 y' \Rightarrow x' \leq_{T_2} y'
    \end{cases} \\
  x \leq_{T^K} y & \iff \forall k \in K \cdot x k \leq_T y k \\
  x \leq_{P_T} y & \iff \forall e \in x \exists e' \in y \cdot e \leq_T e'
\end{align*}
\]

This preorder will be referred to in the sequel as \textit{structural inclusion}. Note that \textit{forward} refinement of nondeterministic components based on \(\leq_T\) captures the classical notion of \textit{nondeterminism reduction}.

However, \(\leq_T\) is inadequate for partial components, since via \(\leq_{T+1} = \leq_T + \text{id}\) refinement would collapse into bisimilarity instead of entailing an increase of definition on the implementation side. The alternative proposed in [17],

\[
x \leq^{F}_{T+1} y \iff \\
  \begin{cases}
    x = i_1 x' \land y = i_1 y' \Rightarrow x' \leq_{T} y' \\
    x = i_2 \ast \Rightarrow \text{true}
  \end{cases}
\]

(where \(F\) stands for ‘failure’) adds a \textit{maybe} clause and should take precedence over general sum. In order to reason about this alternative, we write it in pointfree notation:

\[
\leq^{F}_{T+1} \triangleq [i_1 \cdot \leq^{0}_{T}, T]^{0}
\]

The proof that this is an upper bound of \(\leq_{T+1}\) is immediate via converses:

\[
\begin{align*}
  (\leq^{T+1})^{0} & = \{ \text{(35) and converses} \} \\
  & \leq T^{0} + \text{id} \\
  & = \{ \text{relational coproduct} \} \\
  & [i_1 \cdot \leq^{0}_{T}, i_2] \\
  & \leq \{ i_2 \leq_T \} \\
  & [i_1 \cdot \leq^{0}_{T}, T] \\
  & = \{ \text{(39)} \} \\
  & \leq^{F}_{T+1})^{0}
\end{align*}
\]
That $\leq^F_{T+1}$ is membership-compatible can be proved in a similar way:

\[
\leq^F_{T+1} \subseteq \in_{T+1} \setminus \in_{T+1}
\]

\[
\equiv \{ \in_{T+1} = [\in_T, 1] = \in_T \cdot \xi_T^T ; \text{Galois (32)} \}
\]

\[
\in_T \cdot i_1^o \cdot [i_1 \cdot \leq_T^o, T]^o \subseteq \in_T \cdot i_1^o
\]

\[
\equiv \{ \text{converses} \}
\]

\[
[i_1 \cdot \leq_T^o, T] \cdot i_1 \cdot \in_T^o \subseteq i_1 \cdot \in_T^o
\]

\[
\equiv \{ +\text{-cancellation} \}
\]

\[
i_1 \cdot \leq_T^o \cdot \in_T^o \subseteq i_1 \cdot \in_T^o
\]

\[
\equiv \{ i_1 \text{ is an injection} \}
\]

\[
\leq_T \subseteq \in_T \setminus \in_T
\]

(Note the equivalence; this is not an implication.)

5. Proof of transposition as backward refinement

We are now ready to carry out the proof of Lemma 3.1. As explained in [17], behaviour refinement can only be discussed between components with the same interface. Therefore, for $p$ to be compared with $p \uparrow$, it needs to be postcomposed with a suitable embedding to extend its output interface from $O$ to $O + 1$. In this way, Lemma 3.1 is restated as refinement inequation:

\[
p, i_1 \sqsubseteq^F_T p \uparrow (40)
\]

whose likeness to its $\text{maybe}$-transpose counterpart

\[
i_1 \cdot R \subseteq \Gamma R
\]

arising from (5) is worth mentioning. Following Definition 4.2, (40) is established by finding a morphism $h$, such that

\[
a_{p, i_1^o} \cdot h \leq_T^o T h \cdot a_{p\uparrow}
\]

holds for $T X = B(X \times (O + 1))^T$. Knowing that the state space of both components is the same, we choose $h = id$. Then, thanks to (37) and (A.3), what we have to prove reduces to

\[
a_{p, i_1^o} \leq^F_{B(\sqcup \times (O + 1))} a_{p\uparrow}
\]

cf. diagram

\[
B(U_p \times (O + 1)) \xrightarrow{a_{p, i_1^o}} U_p \times I
\]

\[
B(id \times (id + id)) \xrightarrow{id} id
\]

\[
B(U_p \times (O + 1)) \xleftarrow{a_{p\uparrow}} U_p \times I
\]

or

\[
\xi_B \cdot a_{p, i_1^o} \leq^F_{B(\sqcup \times (O + 1)) + 1} \xi_B \cdot a_{p\uparrow}
\]

in order to bring the ‘failure’ version of the preorder into play. Recall that $a_{p, i_1^o} = B(id \times i_1) \cdot a_p$ and Definition 3.2. For notation economy, we factor the (long) chained composition which defines $a_{p\uparrow}$ as follows:

\[
a_{p\uparrow} = \xi_B^o \cdot [i_1, \xi_B \cdot \eta_B] \cdot (B_+ (id \times i_1) + (id \times i_2)) \cdot a'
\]

\[
a' = (\pi_2 + id) \cdot dr \cdot a''
\]

\[
a'' = (id \times \xi_B \cdot a_p) \cdot a \cdot (\Delta \times id)
\]
Then we calculate:

\[
\hat{c}_B \cdot a_p; \eta \leq_F B_{+}((O \times (O + 1)) + 1) \hat{c}_B \cdot a_p
\]

\[
\begin{align*}
\equiv & \quad \{ \text{(41): } \hat{c}_B \cdot \hat{c}_B = \text{id} \} \\
\equiv & \quad \{ \leq_F B_{+}((O \times (O + 1)) + 1) \text{ [i] } \hat{c}_B \cdot \eta \} \cdot (B_{+}(id \times i_1) + (id \times i_2)) \cdot a' \\
\equiv & \quad \{ \leq_F B_{+}((O \times (O + 1)) + 1) \text{ [i] } \hat{c}_B \cdot \eta \} \cdot (B_{+}(id \times i_1) + (id \times i_2)) \cdot a' \\
\equiv & \quad \{ \text{dropping subscripts } B \text{ and } B_{+} ; \hat{c}-natural ; +\text{-absorption (twice)} \} \\
\equiv & \quad \{ \text{defining } a' = (\hat{c}_B + id) \cdot \text{dr} \cdot a'' \text{ (42) and } \hat{c} \cdot a_p = \hat{c} \cdot a'' \} \\
\equiv & \quad \{ \text{monotonicity of } (\hat{c} \cdot a'' \hat{c}) \} \\
\equiv & \quad \{ \leq_F B_{+}(id \times i_1) , \hat{c} \cdot \eta \cdot (id \times i_2) ; \text{dr} \subseteq [i_1 \cdot \leq^0 \cdot B_{+}(id \times i_1) , \text{T}] \cdot \pi_2 \cdot \pi_2 \cdot a'' \} \\
\equiv & \quad \{ \text{shunting over } i_1^2 \text{ and } i_2^2 \text{; } +\text{-cancellation (twice)} ; \pi_2 \cdot \pi_2 \cdot \pi_2 = \text{id} \} \\
\equiv & \quad \{ i_1 \text{ is an injection ; every relation is at most } \text{T} \} \\
B_{+}(id \times i_1) \subseteq \leq^0 \cdot B_{+}(id \times i_1) \\
\equiv & \quad \{ \text{(39): } \leq \text{ is reflexive} \} \\
\end{align*}
\]

true

6. Factorization of transposed components

The transposition process described so far may be classified as internal or monolithic in the sense that the coalgebra which encodes the dynamics of \( p \) is modified. This may be a disadvantage in contexts where component \( p \) is offered by an external source and has to be deployed as-is. Typically, as in, e.g., Meyer’s design-by-contract approaches [15,19], such components are supplied with an interface which caters for any usage constraints \( p \) might have. From our modelling point of view, let \( \delta_{p} \) encode such an interface. Therefore component \( p \) splits into a server (the original \( p \)) and front-end \( \delta_{p} \) ("\( \delta_{p}^* \) after dialog) which validates inputs and activates the server only when the computation can be completed successfully. Clearly, the execution of \( p \) fails when and only when activated with pairs \( \langle u, i \rangle \) not in \( \text{dom} \ R_p \), the domain of \( R_p \) (recall Definition 3.1).

For the moment, however, consider the simpler case in which partiality of \( p \) depends only on the input values supplied and let \( \phi : 2 \leftarrow I \) be the test for valid inputs as recorded in the interface of component \( p \). Therefore, front-end \( \delta_{p} : I + 1 \leftarrow I \) is defined as the lifting \( \phi^\rightarrow \) of function

\[
\begin{align*}
\Phi & = I \xrightarrow{\phi^?} I + I \xrightarrow{id + !} I + 1 \\
\end{align*}
\]
Using $\delta_p$, we may now specify the try-again (total) version of component $p$ by the following server/front-end aggregation:

$$ \delta_p \mathbin{;} (p \uplus \text{idle}) : O + 1 \leftarrow I $$

(44)

where $\text{idle} = \gamma id_1 \gamma$, the lifting of identity over $I$, “absorbs” the “invalid” calls of $p$ (see Fig. 4).

The most general case, however, makes the validity of a component’s call depend not only on the input supplied but also on the current value of $p$’s state variable. Therefore, this value must be known to front-end $p$, which means that it should be made available by $p$ as a sort of attribute. It seems reasonable to assume such an attribute as private, i.e., available only when $p$ is intended to act as a server accessed through a validation front-end such as $p$. Formally, $p$ must be of shape

$$ p = p'; \gamma \pi_2 \gamma : O \leftarrow I $$

where $p' : U_p \times O \leftarrow I$, on completion of a service call, yields not only the corresponding output value but also the current value of its internal state. The role of $\gamma \pi_2 \gamma$ is, of course, that of hiding the latter on a stand-alone deployment of $p$.

Now, front-end $\delta_p$ has to maintain, as its own state space, the most recent value of $p$’s state space and offer an updating service, triggered by an input of type $U_p$. This adds to its main validation service, which makes use of both the supplied input (of type $I$) and the stored state information. Formally,

**Definition 6.1.** The front-end $\delta_p$ of a component $p$ is another component

$$ \delta_p : I + 1 \leftarrow I + U_p = \langle u_p \in U_p, \overline{a_{\delta_p}} \rangle $$

where

$$ a_{\delta_p} = U_p \times (I + U_p) \xrightarrow{\text{dr}} (U_p \times I) + (U_p \times U_p) \xrightarrow{\text{test+update}} (U_p \times (I + 1)) + U_p \xrightarrow{\eta_B \uplus [\text{id}, (id \uplus \text{id})]} B(U_p \times (I + 1)) $$

Service update is nothing but $\pi_2$: its purpose is to refresh the front-end state value, whereas

$$ \text{test} = U_p \times I \xrightarrow{a^* (\Delta \times \text{id})} U_p \times (U_p \times I) \xrightarrow{id \times \Gamma(\text{dom} R_p)} U_p \times (U_p \times I + 1) \xrightarrow{id \times (\pi_2 \uplus \text{id})} U_p \times (I + 1) $$

The definition of $a_{\delta_p}$ amounts to the definition of both services, pre- and post-composed with some housekeeping morphisms, among which $\eta_B$ is used to frame the front-end, which is always a purely deterministic component, in the behaviour model of $p$ so as to ensure correct composition.
Finally, the server/front-end architecture is defined through an aggregation pattern similar to (44) but with an additional step: on every execution of the server component, the computed value for its state is fed back to \( \delta_p \), using the corresponding update service. This is captured by component algebra expression

\[
(\delta_p ; (p' \boxplus \text{id}))|_{U_p} : O + 1 \leftarrow I
\]  

(45)

—see also Fig. 5—in which combinator \( p|_Z \), to be defined next, belongs to the family of feedback operators studied in [6].

**Definition 6.2.** The feedback combinator is defined, for each object \( Z \), as a family of functors which is the identity on arrows and maps each component \( p : Z \times O + P \leftarrow I + Z \) to

\[
p|_Z : O + P \leftarrow I = \langle u_p \in U_p, \bar{p}|_Z \rangle
\]

where

\[
a|_Z = U_p \times I \xrightarrow{id \times i_1} U_p \times (I + Z) \xrightarrow{a_p} B(U_p \times (Z \times O + P))
\]

\[
B((a^\circ \text{id}) \cdot \text{dr}) \xrightarrow{B((a^\circ \text{id}) \cdot \text{id})} B((U_p \times Z) \times O) + U_p \times P)
\]

\[
B((id \times i_2) \cdot \text{id}) \xrightarrow{B((id \times i_2) \cdot \text{id}) + \text{id}} B((U_p \times (I + Z)) \times O) + U_p \times P)
\]

\[
B((a_p \cdot \text{id}) + \text{id}) \xrightarrow{B(U_p \times (Z \times O + P)) \times O) + U_p \times P)
\]

\[
B((B_{\pi_1} \cdot \text{id}) + \text{id}) \xrightarrow{B(U_p \times O + U_p \times P)}
\]

\[
B((\tau_r + \eta) \cdot \text{id}) \xrightarrow{B(U_p \times O) + B(U_p \times P))}
\]

\[
B[B_{i_1}, B_{i_2}] \xrightarrow{BB((U_p \times O) + (U_p \times P))}
\]

\[
\mu \cdot BB\text{dr}^\circ \xrightarrow{B(U_p \times O + P))}
\]

Note that the output fragment \( Z \) to be fed back appears in a very general context—\( Z \times O + P \)—which explains the amount of *housekeeping* in the formal definition.

In Section 5, it was shown how transposed component \( p \uparrow \) could be regarded as a backward refinement of the original \( p \). Now that two alternative transposes for \( p \) have been introduced, through Definitions 3.2 and 6.2, their equivalence needs to be checked. This is formulated as a bisimulation equation:

**Lemma 6.1.** Let \( p : O \leftarrow I \) be a partial component. Then

\[
p \uparrow \sim (\delta_p ; (p' \boxplus \text{id}))|_{U_p}
\]  

(46)

**Proof.** In the style of [5,6] this equation is proved by the identification of a coalgebra morphism \( h : U_p \leftarrow U_p \times (U_p \times I) \) connecting the state-spaces of the underlying coalgebras. An obvious choice is \( h = \pi_1 \), whereby the commutativity of the homomorphism square is checked (see details in [7]). \( \square \)
7. Conclusions and future work

As mentioned in the Introduction, the context for this paper is a generic framework for composition and refinement of software components regarded as pointed coalgebras, parametric on a behavioural model [5,6,17]. In such a framework, our intention is to discuss formally how the behaviour of a partial component can be extended with try-again cycles preventing from eventual collapse. This leads to an extension of totalization (or transposition) techniques from the algebraic to the coalgebraic context. Note that the transposition process is generic in the sense that it can be applied to any component whose behavioural model, as captured by monad B, does not rule out the possibility of failure.

The transposition process was addressed in this paper as an exercise in coalgebraic refinement. In particular, it was shown that a backward refinement relation holds between the original partial component and the transposed one. In general, backward refinement reflects the dynamics of the abstract coalgebra back into the refined one and, for an appropriate refinement preorder ($\leq_F$), this seems to capture nicely the envisaged behavioural extension. This is actually a first published application of backward refinement: in previous publications (namely [17,18]) all emphasis has been placed on the dual forward form.

Regarding transposition as a refinement situation entailed the need to re-visit the theory in [17] in order to formally justify what seemed to be just intuitive decisions there. This, however, would lead to contrived proofs if performed at the (pointwise) level at which the refinement preorders are given in [17]. Following a similar approach adopted elsewhere in studying conventional operation refinement [22], it was decided to re-frame the theory of [17] in the pointfree relational calculus. The authors regard the outcome of this effort—a generic approach to coalgebraic refinement by pointfree calculation—as a major contribution of this paper. Moreover, this paves the way for the systematic study of the whole spectrum of refinement preorders for coalgebraic models, which, as shown in this paper, is larger than one would suspect at first sight.

Finally, in Section 6, we addressed the factorization (“fission”) of a totalized coalgebra into two coalgebraic components—the original one and an added front-end—which cooperate in a client–server style. In future work we intend to pursue the study of this sort of factorization which underlies the well-known “Seeheim” software architectural model. This raises an interesting question, leading to a further level of generalization, on factorization of software architectures as a formal approach to program understanding in-the-large. Current work on the application of slicing techniques to extract components and connector schemes from systems’s architectural information (see [24], a forthcoming PhD thesis) is a step in that direction.

Another topic for future work relates to the role of genericity, captured by abstracting typical behaviour models as strong monads, in a calculus of components and software architectures. Also related to this subject is the practical evidence given in [27] of the prominent role of a generic behaviour monad in flexibly capturing different evaluation modes in a formal language interpreter.

Appendix A. Lifting orderings to the functional level

Recall the pointwise lifting $\leq$ of a preorder $\leq$ to the functional level (27, 28). This construct enjoys a number of properties suitable for calculation (see e.g., [2]), of which we present only the ones relevant for this paper. Clearly, for any function $h$ one has

$$ f \leq g \Rightarrow f \cdot h \leq g \cdot h $$

(A.1)

The interplay between the lifted notation and converse is captured by equality

$$ (\leq) = (\leq)^\circ $$

(A.2)

whose proof is easy to carry out

$$ f \ (\leq)^\circ \ g $$

\[ \equiv \ \{ \text{(28)} \} \]

$$ f \leq \leq^\circ \cdot g $$

\[ \equiv \ \{ \text{converses} \} \]

$$ f^\circ \leq g^\circ \cdot \leq $$
\[\equiv \{ \text{shunting (twice)} \}\]
\[g \subseteq \leq \ast f\]
\[\equiv \{ (28) \}\]
\[g \leq f\]
\[\equiv \{ \text{converses} \}\]
\[f (\leq)^0 g\]

Finally, the interplay between the lifted notation and currying is captured by property

\[\overline{f} \leq g \equiv f \leq g\] (A.3)

References