Transposing Partial Components — an Exercise on Coalgebraic Refinement ⋆

Luís S. Barbosa and José N. Oliveira [a],

[a] *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 (*e.g.* operating system crash). 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.

Key words: 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. Program-

⋆ Research supported by Fct (the Portuguese Foundation for Science and Technology), in the context of the PURe Project under contract POSI/ICHS/44304/2002.

Email address: {lsb,jno}@di.uminho.pt (Luís S. Barbosa and José N. Oliveira).

mers 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 [1], this has to do with functions which output *Maybe*-values rather than values, where *Maybe* is datatype $Maybe\ a = Nothing \mid Just\ a$.

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 [2].

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 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 (signaling 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 [3], 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 [4,5]. In particular, the transposition of partial components arises as an example of *backward* refinement — a relation dual to the generalisation of the more usual case of non determinism reduction studied in [5]. As an improvement, such a reasoning is carried out in the pointfree binary relation calculus [6,7], 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 [4,5] 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 formu-

```
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

lation of the *Seeheim principle* [8] (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 is presented in section 3. Once the underlying theory of coalgebraic refinement is given in section 4, 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) is the main subject of section 6, which paves the way to 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 [9,10], Z [11] or B [12], 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^{++} [13] and Z^{++} [14] 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.

```
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

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* [15], ie. functions of type $\alpha : \mathsf{T}U \longleftarrow U$ for T 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 [15]. References [16,17] 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 Figure 1.

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

$$[\![\texttt{PUSH}]\!] : S \times 1 \longleftarrow S \times A$$

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

$$[\![\texttt{POP}]\!] : S \times A \longleftarrow S \times 1$$

However, $[\![\texttt{POP}]\!]$ 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 [7].

We now focus on `unOrdCol`, whose nondeterministic semantics is more interesting than that of `stackObj`. The idea of *packaging* methods `PUT` and `GET` together in a public interface is nicely captured by *summing up* the two relations which capture their semantics,

$$[\![\texttt{PUT}]\!] + [\![\texttt{GET}]\!] : S \times 1 + S \times A \longleftarrow S \times A + S \times 1 \tag{1}$$

where $+$ denotes relational *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 [1]

$$\mathsf{dr}^\circ \cdot ([\![\texttt{PUT}]\!] + [\![\texttt{GET}]\!]) \cdot \mathsf{dr} : S \times (1 + A) \longleftarrow S \times (A + 1) \tag{2}$$

where dr is the *distribute-right* isomorphism and dr° denotes its *converse* [2]. Knowing that every binary relation R can be converted into a (set-valued) function ΛR via the *power-transpose* isomorphism [7,3] defined by

$$f = \Lambda\ R \equiv\ (bRa \equiv b \in f\ a)$$

for all $R : B \longleftarrow A$ and $f : \mathcal{P}B \longleftarrow A$ ($\mathcal{P}B$ denotes the set of all subsets of B), we convert relation $\mathsf{dr}^\circ \cdot ([\![\texttt{PUT}]\!] + [\![\texttt{GET}]\!]) \cdot \mathsf{dr}$ into function

$$\Lambda(\mathsf{dr}^\circ \cdot ([\![\texttt{PUT}]\!] + [\![\texttt{GET}]\!]) \cdot \mathsf{dr}) : \mathcal{P}(S \times (1 + A)) \longleftarrow S \times (A + 1)$$

which — finally — can be *curried* into coalgebra

$$\overline{\Lambda(\mathsf{dr}^\circ \cdot ([\![\texttt{PUT}]\!] + [\![\texttt{GET}]\!]) \cdot \mathsf{dr})} : \underbrace{\mathcal{P}(S \times (1 + A))^{(A+1)}}_{\mathsf{T}S} \longleftarrow S$$

where the bar over the coalgebra denotes the currying isomorphism which is

[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: $b(R \cdot S)c$ holds wherever there exists some mediating a such that $bRa \wedge aSc$ holds. In the case of functions, say f and g instead of R and S, $b(f \cdot g)c$ equivales $b = f(g\ c)$. Here — and elsewhere in this paper — we follow the fairly common notation standard of denoting (total) functions by lowercase letters (f, g, *etc*) and all other relations by uppercase letters (*e.g.* R, S).
[2] In general, the converse R° of binary relation R is such that bRa equivales $aR^\circ b$.

such that, given binary function g, equivalence

$$f = \overline{g} \;\equiv\; \langle \forall\ a, b\ ::\ (f\ a)b = g(a,b) \rangle$$

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 $M_i : O_i \longleftarrow I_i$ will be captured by coalgebra

$$\overline{\Lambda(\mathsf{dr}^{\circ} \cdot (\sum_{i=1}^{n} [\![M_i]\!]) \cdot \mathsf{dr})}$$

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

Going generic. The above construction raises two observations. First, initialization statements such as `col : Collection := {}` 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 *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 *behaviour models*: as shown in [18,16,17], *genericity* is achieved by replacing the *powerset* monad by an arbitrary *strong monad* [3] B. In this paper our attention will be focused on behavioural models 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 $p : O \longleftarrow I$, as follows: it is specified as

[3] A *strong monad* is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor and both η and μ are strong natural transformations. B being strong means there exist natural transformations $\tau_r^{\mathsf{B}} : \mathsf{B}(\mathsf{Id} \times -) \Longleftarrow \mathsf{B} \times -$ and $\tau_l^{\mathsf{B}} : \mathsf{B}(- \times \mathsf{Id}) \Longleftarrow - \times \mathsf{B}$ called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "$-$" along functor B. Strength τ_r, followed by τ_l maps $\mathsf{B}I \times \mathsf{B}J$ to $\mathsf{BB}(I \times J)$, which can, then, be flattened to $\mathsf{B}(I \times J)$ via μ. In most cases, however, the *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 $\delta r_{I,J} = \tau_{r_{I,J}} \bullet \tau_{l_{\mathsf{B}I,J}}$ Dually, $\delta l_{I,J} = \tau_{l_{I,J}} \bullet \tau_{r_{I,\mathsf{B}J}}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations. Whenever δ_r and δ_l coincide, the monad is said to be *commutative*.

a (pointed) coalgebra in Set

$$\langle u_p \in U_p, \overline{a}_p : \mathsf{B}(U_p \times O)^I \longleftarrow U_p \rangle \tag{3}$$

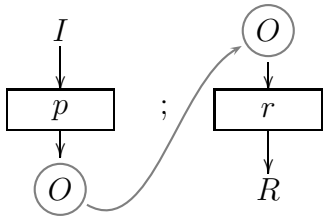
where point u_p is the 'initial' or 'seed' state. Therefore, the computation of an action will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations: unit (η) and multiplication (μ) provide, respectively, a value embedding and a 'flatten' operation to reduce nested behavioural annotations. Strength, either in its right (τ_r) or left (τ_l) version, caters for context information. Also notice that both I and O can be any datatype and are therefore not restricted to the datatype sums which arise in components built from model-oriented specifications, as illustrated above.

A component calculus. Regarding software components as *pointed coalgebras* parametric on a behaviour model B, gives rise to a rich semantic framework where components become *arrows* in a (bicategorical) universe Cp whose objects are sets, providing types to input/output parameters (the components' *interfaces*). Component morphisms $h : q \longleftarrow p$, 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 [18], in the context of which a component *calculus* [16,17] was developed also in a generic way, *i.e.*, parametric on monad B.

This calculus involves component assembly patterns such as *pipeline* ($;$) and three tensors capturing, respectively, *external choice* ($\boxplus$), *parallel* ($\boxtimes$) and *concurrent* ($\circledast$) composition. For the purpose of this paper, it is enough to understand the semantics of pipelining and external choice. Let $p : O \longleftarrow I$, $q : R \longleftarrow J$ and $r : R \longleftarrow O$ be components. Then $p \,;\, r : R \longleftarrow I$ is a component whose coalgebra $a_{p;r} : \mathsf{B}(U_p \times U_r \times R) \longleftarrow U_p \times U_r \times I$ 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 $p \boxplus q : O + R \longleftarrow I + J$, 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 $a_{p \boxplus q} : \mathsf{B}(U_p \times U_q \times (O + R)) \longleftarrow U_p \times U_q \times (I + J)$ which captures this behaviour can be found in [16].

Generalised *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.

Pipeline $p \,;\, r$:



Choice $p \boxplus q$:

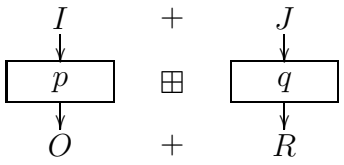


Fig. 3. Two combinators of the component algebra

Finally, component adaptation is captured by a *wrapping* combinator and function *lifting* [16], a combinator which promotes a function $f : B \longleftarrow A$ to component $\ulcorner f \urcorner : B \longleftarrow A$ whose coalgebra (over $\mathbf{1}$, the singleton type) is defined by $a_{\ulcorner f \urcorner} = \eta \cdot (id \times f)$. 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 [16,17] 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,*

$$\mathsf{B} \;\cong\; \mathsf{B}_{+} + \mathbf{1} \tag{4}$$

as witnessed by a given natural isomorphism $\xi_{\mathsf{B}} : \mathsf{B}_+ + \mathbf{1} \longleftarrow \mathsf{B}$. *We denote by* R_p *the simple relation (vulg. partial function) of type* $\mathsf{B}_+(U_p \times O) \longleftarrow U_p \times I$, *of which* a_p *is the* maybe-*transpose [3], that is,* $a_p = \Gamma\ R_p$ *and* $R_p = i_1^\circ \cdot a_p$, *where isomorphism* Γ *is uniquely defined by universal property*

$$f = \Gamma\ R \ \equiv\ R = i_1^\circ \cdot f \tag{5}$$

For instance, B_+ is the identity monad Id for B the "Maybe" monad and the non empty powerset monad $\mathcal{P}_+$ for $\mathsf{B} = \mathcal{P}$.

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 non-determinsitic component whose behaviour is expressed by powerset-coalgebra a_p. Its expected *try-again* counterpart will be

$$\begin{array}{l} a_{p\uparrow}\langle u, i\rangle \ = if\ a_p\langle u, i\rangle = \{\} \\ \qquad\qquad\quad then\ \{\langle u, Nothing\rangle\} \\ \qquad\qquad\quad else\ \{\langle u', Just\ o\rangle \mid \langle u', o\rangle \in a_p\langle u, i\rangle\} \end{array}$$

In general, the dynamics of p is pre-composed with the diagonal function $\triangle\ a = \langle a, a\rangle$ 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* $p : O \longleftarrow I$, *its* try-again *transpose* $p\uparrow : O + \mathbf{1} \longleftarrow I$ *is defined by a new coalgebra over the same state space whose dynamics is given by* [4]

$$\begin{array}{rl} a_{p\uparrow} = & U_p \times I \xrightarrow{\triangle \times id} (U_p \times U_p) \times I \xrightarrow{\mathsf{a}} U_p \times (U_p \times I) \\ & \xrightarrow{id \times a_p} U_p \times \mathsf{B}(U_p \times O) \\ & \xrightarrow{id \times \xi_{\mathsf{B}}} U_p \times (\mathsf{B}_+(U_p \times O) + \mathbf{1}) \xrightarrow{\mathsf{dr}} U_p \times \mathsf{B}_+(U_p \times O) + U_p \times \mathbf{1} \\ & \xrightarrow{\pi_2 + id} \mathsf{B}_+(U_p \times O) + U_p \times \mathbf{1} \\ & \xrightarrow{\mathsf{B}_+(id \times i_1) + id \times i_2} \mathsf{B}_+(U_p \times (O + \mathbf{1})) + U_p \times (O + \mathbf{1}) \\ & \xrightarrow{[i_1, \xi_{\mathsf{B}} \cdot \eta_{\mathsf{B}}]} \mathsf{B}_+(U_p \times (O + \mathbf{1})) + \mathbf{1} \xrightarrow{\xi_{\mathsf{B}}^\circ} \mathsf{B}(U_p \times (O + \mathbf{1})) \end{array}$$

Note in this definition the role of dr in distributing the initial state across the *maybe-shape*, later to be kept (via $\pi_2 + id$, where π_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 *either* R *or* S) which is given by

[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.

closed formula

$$[R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \tag{6}$$

where i_1 and i_2 are the coproduct injections.

The signature extension, from $p : O \longleftarrow I$ to $p\uparrow : O + \mathbf{1} \longleftarrow I$, resembles the *maybe*-transpose of partial functions [3]. At behaviour level, output of type $\mathbf{1}$ 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 (generalised) transition systems, however, the underlying coalgebras have the same structure but for the presence, in the latter, of reflexive arrows at every *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 *try-again* extra-behavioural cycles. Formally,

Lemma 3.1 *Component* $p\uparrow : O + \mathbf{1} \longleftarrow I$ *is a* backward *refinement of* $p : O \longleftarrow I$*, with respect to the* structural failure *refinement order* $\leq_{\mathsf{T}}^{F}$ *of [4].*

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, [4], in which refinement is captured by the existence of some form of *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 [4] as *forward* or *backward* refinement. Reference [4] is further expanded into a journal version [5] in which the component calculus of [16,17] is extended with a number of generic refinement laws. Both [4] and [5] are, however, mainly concerned with *forward* refinement, which generalises the usual axis of *non determinism reduction* in a functorial way.

As we shall show in the sequel, *backward* refinement corresponds to a similar functorial generalisation of *definition increase* and turns out to be the right way of characterising 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 [4], but also reframes its main constructions in a more general *pointfree* way. Such a reconstruction, on its turn, not only largely increases calculational power, but also provides a formal justification of some definitions which were in [4] 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 [4] is that, just as transition systems can be coded back as coalgebras, any coalgebra $\langle U, \alpha : \mathsf{T}U \longleftarrow U \rangle$ specifies a (T-shaped) transition structure over its carrier U. For extended polynomial Set endofunctors [5] such a structure may be expressed as a binary relation ${}_{\alpha}\longleftarrow : U \longleftarrow U$, defined in terms of the *structural membership* relation $\in_{\mathsf{T}} : U \longleftarrow \mathsf{T}\ U$,

$$u' \ {}_{\alpha}\longleftarrow u \quad \equiv \quad u' \in_{\mathsf{T}} \alpha\ u$$

which can be written in less symbols as

$${}_{\alpha}\longleftarrow \quad = \quad \in_{\mathsf{T}} \cdot \alpha \tag{7}$$

at pointfree level. Relation $\in_{\mathsf{T}}$ is defined by induction on the structure of polynomial T as follows:

$$\begin{aligned}
\in_{\mathsf{Id}} &= id && (8)\\
\in_{\mathsf{K}} &= \bot && (9)\\
\in_{\mathsf{T}_1 \times \mathsf{T}_2} &= (\in_{\mathsf{T}_1} \cdot \pi_1) \cup (\in_{\mathsf{T}_2} \cdot \pi_2) && (10)\\
\in_{\mathsf{T}_1 + \mathsf{T}_2} &= [\in_{\mathsf{T}_1}, \in_{\mathsf{T}_2}] && (11)\\
\in_{\mathsf{T}_1 \cdot \mathsf{T}_2} &= \in_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1} && (12)\\
\in_{\mathsf{T}^K} &= \bigcup_{k \in K} \in_{\mathsf{T}} \cdot \beta_k \ (\text{where } \beta_k f = f\ k) && (13)\\
\in_{\mathcal{P}} &= \in \quad (\text{set-theoretic membership}) && (14)
\end{aligned}$$

Some comments on notation follow. Since id is the identity function and $\bot$ is the empty relation, $x \in_{\mathsf{Id}} y \ \ \mathsf{iff} \ \ x = y$ is the pointwise counterpart of the first clause and $x \in_K y \ \ \mathsf{iff} \ \ \mathsf{false}$ is that of the second. The pointwise expansion of the third clause is $x \in_{\mathsf{T}_1 \times \mathsf{T}_2} y \ \ \mathsf{iff} \ \ x \in_{\mathsf{T}_1} \pi_1\ y \ \vee \ x \in_{\mathsf{T}_2} \pi_2\ y$, where π_1, π_2 dentote the left and right projections of relational product. Formula (6) and rule

$$b(f^{\circ} \cdot R \cdot g)a \ \ \mathsf{iff} \ \ (f\ b)R(g\ a) \tag{15}$$

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

$$x \in_{\mathsf{T}_1 + \mathsf{T}_2} y \quad \mathsf{iff} \quad \begin{cases} y = i_1\ y' & \Rightarrow x \in_{\mathsf{T}_1} y' \\ y = i_2\ y' & \Rightarrow x \in_{\mathsf{T}_2} y' \end{cases}$$

[5] This is the class inductively defined as the least collection of functors containing the identity Id and constant functors K for every object K in the category, closed by functor composition and finite application of product, coproduct, covariant exponential and finite powerset functors.

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

Relation $\in_{\mathsf{T}}$ is actually an instance of datatype membership defined by Hoogendijk [19] as a Galois connection, which entails two results which are of interest to this paper. First, that $\in_{\mathsf{T}}$ satisfies the following naturality condition

$$h \cdot \in_{\mathsf{T}} \;=\; \in_{\mathsf{T}} \cdot \mathsf{T}\, h \tag{16}$$

for any function h. Second, that $\in_{\mathsf{T}}$ can be further extended to the construction of *recursive* datatypes. In brief, this goes as follows: let $\mathsf{T}A$ be the type functor induced by a given *base* polynomial bifunctor B , that is, $in : \mathsf{T}A \longleftarrow \mathsf{B}(A, \mathsf{T}A)$ is an isomorphism. Let $\in_1, \in_2$ be the two memberships associated to the two places of bifunctor B, that is, $\in_1 \colon X \longleftarrow \mathsf{B}(X, Y)$ and $\in_2 \colon Y \longleftarrow \mathsf{B}(X, Y)$. Relation $Atroot = \in_1 \cdot in^\circ$ (of type $Atroot : A \longleftarrow \mathsf{T}A$ 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 $\mathsf{T}A$, while $Branch : \mathsf{T}A \longleftarrow \mathsf{T}A$, defined by $Branch = \in_2 \cdot in^\circ$, checks whether some other t' is a branch of t. Rather elegantly, Hoogendijk calculates type functor membership, $\in_{\mathsf{T}} \colon A \longleftarrow \mathsf{T}A$, as follows:

$$\in_{\mathsf{T}} = Atroot \cdot Branch^* \tag{17}$$

where $Branch^*$ denotes the reflexive, transitive closure of $Branch$. So, $a \in_{\mathsf{T}} t$ means that a can 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 $p : O \longleftarrow I$ is based on functor $\mathsf{B}(\mathsf{Id} \times O)^I$. Therefore a possible (and intuitive) way of regarding component p as a behavioural refinement of some other component $q : O \longleftarrow I$ is to consider that p-transitions are simply *preserved* in q. For non deterministic 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.* [15]) to entail bisimilarity. Therefore we have to look for a somewhat *weaker* notion of a morphism between coalgebras.

Recall that a T-coalgebra morphism $h : \alpha \longleftarrow \beta$ is a function from the state space of β to that of α such that

$$\mathsf{T}h \cdot \beta \;=\; \alpha \cdot h \tag{18}$$

holds. Regarding β and α as (generalised) transition systems, equation (18) becomes relational equality

$$h \cdot \;{}_{\beta}{\longleftarrow}\; \;=\; \;{}_{\alpha}{\longleftarrow}\; \cdot h \tag{19}$$

(thanks to (7), (16) and the fact that $(\in_{\mathsf{T}} \cdot)$ is an isomorphism [3]), *i.e.* the conjunction of inclusions

$$h \cdot \;{}_{\beta}{\longleftarrow}\; \subseteq \;{}_{\alpha}{\longleftarrow}\; \cdot h \tag{20}$$

$$\;{}_{\alpha}{\longleftarrow}\; \cdot h \subseteq h \cdot \;{}_{\beta}{\longleftarrow}\; \tag{21}$$

By *shunting* [6], inclusion (20) is equivalent to

$$\;{}_{\beta}{\longleftarrow}\; \subseteq h^{\circ} \cdot \;{}_{\alpha}{\longleftarrow}\; \cdot h \tag{22}$$

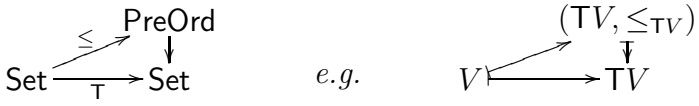
Inequalities (22) and (21) take a more familiar shape once variables are introduced:

$$v' \;{}_{\beta}{\longleftarrow}\; v \;\Rightarrow\; h\, v' \;{}_{\alpha}{\longleftarrow}\; h\, v \tag{23}$$

$$u' \;{}_{\alpha}{\longleftarrow}\; h\, v \;\Rightarrow\; \exists_{v' \in V}.\; v' \;{}_{\beta}{\longleftarrow}\; v \;\wedge\; u' = h\, v' \tag{24}$$

They jointly state that, not only β dynamics, as represented by the induced transition relation, is *preserved* by h (20, 23), but also α dynamics is *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 Set endofunctor T. This is defined in [21] as a functor $\leq$ which makes the following diagram commute:



[6] In the relational calculus [20], 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 *shunting rules*.

This means that for any function $h : U \longleftarrow V$, $\mathsf{T}h$ preserves the order, *i.e.*

$$x_1 \leq_{\mathsf{T}V} x_2 \;\;\Rightarrow\;\; (\mathsf{T}h)\; x_1 \leq_{\mathsf{T}U} (\mathsf{T}h)\; x_2 \tag{25}$$

or, in a pointfree formulation,

$$(\mathsf{T}h)\cdot \leq_{\mathsf{T}V} \;\subseteq\; \leq_{\mathsf{T}U} \cdot (\mathsf{T}\; h) \tag{26}$$

In the definition which follows subscripts are dropped, *e.g.* $\leq$ instead of $\leq_{\mathsf{T}V}$, for notation economy. Moreover, we denote by $\dot{\leq}$ the pointwise lifting of preorder $\leq$ to the functional level, *i.e.*

$$f \;\dot{\leq}\; g \;\equiv\; \langle \forall\; x \;::\; f\; x \leq g\; x \rangle \tag{27}$$

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

$$f \;\dot{\leq}\; g \;\equiv\; f \subseteq \leq \cdot g \tag{28}$$

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

$$\mathsf{T}\; h \cdot \beta \;\dot{\leq}\; \alpha \cdot h$$

Dually, h *is said to be a* backwards *morphism if*

$$\alpha \cdot h \;\dot{\leq}\; \mathsf{T}\; h \cdot \beta$$

The following lemma, a pointfree proof of which can be found in [5], shows that such morphisms compose and can be taken as witnesses of refinement situations:

Lemma 4.1 *For* T *an endofunctor in* 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 below will state that forward morphisms preserve transitions whereas backwards morphisms reflect them. To prove this, however, the following extra condition has to be imposed on preorder $\leq$ to express its compatibility with the membership relation: for all $x \in X$ and $x_1, x_2 \in \mathsf{T}X$,

$$x \in_{\mathsf{T}} x_1 \;\wedge\; x_1 \leq\; x_2 \;\;\Rightarrow\;\; x \in_{\mathsf{T}} x_2 \tag{29}$$

or, again in a pointfree formulation,

$$\in_{\mathsf{T}} \cdot \leq \ \subseteq \ \in_{\mathsf{T}} \tag{30}$$

A preorder $\leq$ 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* Set*, and β and α two T-coalgebras as above. Let ${}_{\beta}\longleftarrow$ and ${}_{\alpha}\longleftarrow$ denote the corresponding transition relations. A backward (respectively, forward) morphism $h : \alpha \longleftarrow \beta$ reflects (respectively, preserves) such transition relations.*

PROOF. Let h be a backward morphism. Transition reflection, defined by equation (21), is established as follows:

$$\begin{array}{ll}
& {}_{\alpha}\longleftarrow \cdot h \\
= & \quad \{ \text{ definition (7) } \} \\
& \in_{\mathsf{T}} \cdot \alpha \cdot h \\
\subseteq & \quad \{ \text{ } h \text{ backwards entails } \alpha \cdot h \ \subseteq \leq \cdot \mathsf{T}h \cdot \beta, \text{ monotonicity } \} \\
& \in_{\mathsf{T}} \cdot \leq \cdot \mathsf{T}h \cdot \beta \\
\subseteq & \quad \{ \text{ compatibility with } \in_{\mathsf{T}} \text{ (30), monotonicity } \} \\
& \in_{\mathsf{T}} \cdot \mathsf{T}h \cdot \beta \\
\equiv & \quad \{ \text{ } \in_{\mathsf{T}} \text{ natural (16) } \} \\
& h \cdot \in_{\mathsf{T}} \cdot \beta \\
= & \quad \{ \text{ definition (7) } \} \\
& h \cdot {}_{\beta}\longleftarrow
\end{array}$$

The forward case is documented in [5].

□

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 [7] and define *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*

[7] A similar study is made in [5] on *forward* refinement.

$p \leq_{\mathsf{T}} q$ — *if there is a (seed preserving) backward morphism* $p \xleftarrow{h} q$ *that is, such that*

$$\begin{array}{l} h\ u_q = u_p \\ \overline{a_p} \cdot h \dot{\leq}_{\mathsf{T}} \mathsf{T}h \cdot \overline{a_q} \end{array}$$

Subscript T is often dropped wherever 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

$$\leq \ \subseteq \ \in_{\mathsf{T}} \setminus \in_{\mathsf{T}} \tag{31}$$

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

$$R \cdot X \ \subseteq \ S \ \equiv \ X \ \subseteq \ R \setminus S \tag{32}$$

from which its pointwise meaning $x\ (R \setminus S)\ z \ \equiv \ \langle \forall\ y\ ::\ yRx \ \Rightarrow \ ySz \rangle$ can be inferred [6].

Clearly, (31) provides an upper bound for refinement preorders, the lower bound being *id*, the smallest preorder. It is well known (see *e.g.*, [19,3]) that relation $\in_{\mathsf{T}} \setminus \in_{\mathsf{T}}$ corresponds to the lifting of $\in_{\mathsf{T}}$ to (structural) inclusion, *i.e.*,

$$x\ (\in_{\mathsf{T}} \setminus \in_{\mathsf{T}})\ y \ \equiv \ \langle \forall\ e\ :\ e \in_{\mathsf{T}} x :\ e \in_{\mathsf{T}} y \rangle \tag{33}$$

Clearly, $\in_{\mathsf{T}} \setminus \in_{\mathsf{T}}$ always is a preorder [8]. By (31), it is the *largest* refinement preorder. For $\mathsf{T} = \mathsf{Id}$, (31) has only one solution which is easy to calculate:

$$\begin{array}{ll} & \leq_{\mathsf{Id}} \subseteq \in_{\mathsf{Id}} \setminus \in_{\mathsf{Id}} \\ \equiv & \quad \{\ (32)\ \} \\ & \in_{\mathsf{Id}} \cdot \leq_{\mathsf{Id}} \subseteq \in_{\mathsf{Id}} \\ \equiv & \quad \{\ \text{membership definition } \in_{\mathsf{Id}} = id\ \} \\ & \leq_{\mathsf{Id}} \subseteq\ id \end{array}$$

[8] Reflexivity: $R \setminus S$ is reflexive iff $R \subseteq S$; transitivity: $R \setminus S$ is transitive wherever $S \subseteq R$.

$$
\begin{array}{ll}
\equiv & \{ \text{ as a preorder, } \leq_{\mathsf{Id}} \text{ is reflexive } \} \\
& \leq_{\mathsf{Id}} = id
\end{array}
$$

Concerning case $\mathsf{T} = \mathsf{K}$, $\in_{\mathsf{K}} \setminus \in_{\mathsf{K}} = \top$ (where $\top$, the "topmost" relation of its type, is such that $x \top y$ holds for any x, y), since $\in_{\mathsf{K}} = \bot$. In our component model, however, such a preorder on the constant functor would make refinement based on $\in_{\mathsf{T}} \setminus \in_{\mathsf{T}}$ blind to the outputs produced. This suggests an additional requirement on refinement preorders for Cp components: their definition on a constant functor K must be equality on set K, *i.e.*, $\leq_{\mathsf{K}} = id$, 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 $\in_{\mathsf{T}_1 \times \mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}$:

$$
\begin{array}{ll}
& \in_{\mathsf{T}_1 \times \mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2} \\
= & \{ \text{ (10) } \} \\
& (\in_{\mathsf{T}_1} \cdot \pi_1 \cup \in_{\mathsf{T}_2} \cdot \pi_2) \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2} \\
= & \{ \text{ } (R \cup S) \setminus T = (R \setminus T) \cap (S \setminus T) \text{ } \} \\
& (\in_{\mathsf{T}_1} \cdot \pi_1 \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}) \cap (\in_{\mathsf{T}_2} \cdot \pi_2 \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}) \\
= & \{ \text{ } (R \cdot f) \setminus S = f^{\circ} \cdot (R \setminus S) \text{ } \} \\
& \pi_1^{\circ} \cdot (\in_{\mathsf{T}_1} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}) \cap \pi_2^{\circ} \cdot (\in_{\mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}) \\
= & \{ \text{ introduce combinator } \langle R, S \rangle = \pi_1^{\circ} \cdot R \cap \pi_2^{\circ} \cdot S \text{ } \} \\
& \langle \in_{\mathsf{T}_1} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}, \in_{\mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2} \rangle
\end{array}
$$

Note that $\in_{\mathsf{T}_1} \cdot \pi_1 \subseteq \in_{\mathsf{T}_1 \times \mathsf{T}_2}$ holds (similarly for T_2, π_2). From this we infer:

$$
\begin{array}{ll}
& \in_{\mathsf{T}_1} \cdot \pi_1 \subseteq \in_{\mathsf{T}_1 \times \mathsf{T}_2} \wedge \in_{\mathsf{T}_2} \cdot \pi_2 \subseteq \in_{\mathsf{T}_1 \times \mathsf{T}_2} \\
\Rightarrow & \{ \text{ monotonicity of upper adjoint } (R\setminus) \text{ in (32) } \} \\
& \in_{\mathsf{T}_1} \setminus (\in_{\mathsf{T}_1} \cdot \pi_1) \subseteq \in_{\mathsf{T}_1} \setminus (\in_{\mathsf{T}_1 \times \mathsf{T}_2}) \wedge \in_{\mathsf{T}_2} \setminus (\in_{\mathsf{T}_2} \cdot \pi_2) \subseteq \in_{\mathsf{T}_2} \setminus (\in_{\mathsf{T}_1 \times \mathsf{T}_2}) \\
\equiv & \{ \text{ } R \setminus (S \cdot f) = (R \setminus S) \cdot f \text{ (twice) } \} \\
& (\in_{\mathsf{T}_1} \setminus \in_{\mathsf{T}_1}) \cdot \pi_1 \subseteq \in_{\mathsf{T}_1} \setminus (\in_{\mathsf{T}_1 \times \mathsf{T}_2}) \wedge (\in_{\mathsf{T}_2} \setminus \in_{\mathsf{T}_2}) \cdot \pi_2 \subseteq \in_{\mathsf{T}_2} \setminus (\in_{\mathsf{T}_1 \times \mathsf{T}_2}) \\
\Rightarrow & \{ \text{ } \leq_{\mathsf{T}_i} \subseteq (\in_{\mathsf{T}_i} \setminus \in_{\mathsf{T}_i}), \text{ for } i := 1, 2 \text{ } \} \\
& \leq_{\mathsf{T}_1} \cdot \pi_1 \subseteq \in_{\mathsf{T}_1} \setminus (\in_{\mathsf{T}_1 \times \mathsf{T}_2}) \wedge \leq_{\mathsf{T}_2} \cdot \pi_2 \subseteq \in_{\mathsf{T}_2} \setminus (\in_{\mathsf{T}_1 \times \mathsf{T}_2}) \\
\Rightarrow & \{ \text{ monotonicity of } \langle R, S \rangle \text{ and previous calculation } \} \\
& \langle \leq_{\mathsf{T}_1} \cdot \pi_1, \leq_{\mathsf{T}_2} \cdot \pi_2 \rangle \subseteq \in_{\mathsf{T}_1 \times \mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \text{ introduce relational product } R \times S = \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \} \\
& \leq_{\mathsf{T}_1} \times \leq_{\mathsf{T}_2} \subseteq \in_{\mathsf{T}_1 \times \mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \times \mathsf{T}_2}
\end{array}
$$

In summary, this confirms that definition

$$
\leq_{\mathsf{T}_1 \times \mathsf{T}_2} \triangleq \leq_{\mathsf{T}_1} \times \leq_{\mathsf{T}_2} \tag{34}
$$

adopted in [5] 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 [5] in defining

$$
\leq_{\mathsf{T}_1 + \mathsf{T}_2} \triangleq \leq_{\mathsf{T}_1} + \leq_{\mathsf{T}_2} \tag{35}
$$

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

$$
\begin{array}{ll}
& \leq_{\mathsf{T}_1} + \leq_{\mathsf{T}_2} \subseteq (\in_{\mathsf{T}_1 + \mathsf{T}_2} \setminus \in_{\mathsf{T}_1 + \mathsf{T}_2}) \\
\equiv & \quad \{ \text{ Galois (32) } \} \\
& \in_{\mathsf{T}_1 + \mathsf{T}_2} \cdot (\leq_{\mathsf{T}_1} + \leq_{\mathsf{T}_2}) \subseteq \in_{\mathsf{T}_1 + \mathsf{T}_2} \\
\equiv & \quad \{ \text{ membership definition } \} \\
& [\in_{\mathsf{T}_1}, \in_{\mathsf{T}_2}] \cdot (\leq_{\mathsf{T}_1} + \leq_{\mathsf{T}_2}) \subseteq [\in_{\mathsf{T}_1}, \in_{\mathsf{T}_2}] \\
\equiv & \quad \{ \text{ +-fusion } \} \\
& [\in_{\mathsf{T}_1} \cdot \leq_{\mathsf{T}_1}, \in_{\mathsf{T}_2} \cdot \leq_{\mathsf{T}_2}] \subseteq [\in_{\mathsf{T}_1}, \in_{\mathsf{T}_2}] \\
\Leftarrow & \quad \{ \text{ 'either' is monotonic } \} \\
& \in_{\mathsf{T}_1} \cdot \leq_{\mathsf{T}_1} \subseteq \in_{\mathsf{T}_1} \wedge \in_{\mathsf{T}_2} \cdot \leq_{\mathsf{T}_2} \subseteq \in_{\mathsf{T}_2} \\
\equiv & \quad \{ \text{ Galois (32) twice } \} \\
& \leq_{\mathsf{T}_1} \subseteq \in_{\mathsf{T}_1} \setminus \in_{\mathsf{T}_1} \wedge \leq_{\mathsf{T}_2} \subseteq \in_{\mathsf{T}_2} \setminus \in_{\mathsf{T}_2}
\end{array}
$$

4.2.3 Functor composition

We have $\in_{\mathsf{T}_1 \cdot \mathsf{T}_2} = \in_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1}$, recall (12). Calculation of $\leq_{\mathsf{T}_1 \cdot \mathsf{T}_2}$ proceeds by indirect inclusion:

$$
\begin{array}{ll}
& X \subseteq \in_{\mathsf{T}_1 \cdot \mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \cdot \mathsf{T}_2} \\
\equiv & \quad \{ \ (12) \ \} \\
& X \subseteq (\in_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1}) \setminus \in_{\mathsf{T}_1 \cdot \mathsf{T}_2} \\
\equiv & \quad \{ \ \text{Galois (32)} \} \\
& (\in_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1}) \cdot X \subseteq \in_{\mathsf{T}_1 \cdot \mathsf{T}_2} \\
\equiv & \quad \{ \ \text{Galois and (12)} \ \} \\
& \in_{\mathsf{T}_1} \cdot X \subseteq \in_{\mathsf{T}_2} \setminus (\in_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1}) \\
\Leftarrow & \quad \{ \ \text{property } (R \setminus S) \cdot T \subseteq R \setminus (S \cdot T) \ \} \\
& \in_{\mathsf{T}_1} \cdot X \subseteq (\in_{\mathsf{T}_2} \setminus \in_{\mathsf{T}_2}) \cdot \in_{\mathsf{T}_1} \\
\Leftarrow & \quad \{ \ \text{assume } \leq_{\mathsf{T}_2} \subseteq \in_{\mathsf{T}_2} \setminus \in_{\mathsf{T}_2} \ \} \\
& X \subseteq \in_{\mathsf{T}_1} \setminus (\leq_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1}) \\
:: & \quad \{ \ \text{define } \leq_{\mathsf{T}_1 \cdot \mathsf{T}_2} \triangleq \in_{\mathsf{T}_1} \setminus (\leq_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1}) \ ; \ \text{indirection} \ \} \\
& \leq_{\mathsf{T}_1 \cdot \mathsf{T}_2} \subseteq \in_{\mathsf{T}_1 \cdot \mathsf{T}_2} \setminus \in_{\mathsf{T}_1 \cdot \mathsf{T}_2}
\end{array}
$$

We have shown that, should $\leq_{\mathsf{T}_2}$ be membership-compatible, than so is [9]

$$\leq_{\mathsf{T}_1 \cdot \mathsf{T}_2} \triangleq \in_{\mathsf{T}_1} \setminus (\leq_{\mathsf{T}_2} \cdot \in_{\mathsf{T}_1}) \tag{36}$$

4.2.4 Exponentials

Calculation of $\leq_{\mathsf{T}^K}$ proceeds by indirect inclusion:

$$
\begin{array}{ll}
& X \subseteq (\in_{\mathsf{T}^K} \setminus \in_{\mathsf{T}^K}) \\
\equiv & \quad \{ \ (13) \ ; \ (R \cup S) \setminus T = (R \setminus T) \cap (S \setminus T) \ \} \\
& X \subseteq \bigcap_{k \in K} ((\in_{\mathsf{T}} \cdot \beta_k) \setminus (\bigcup_{k' \in K} \in_{\mathsf{T}} \cdot \beta'_k)) \\
\Leftarrow & \quad \{ \ \text{choose } k' := k \ \} \\
& X \subseteq \bigcap_{k \in K} ((\in_{\mathsf{T}} \cdot \beta_k) \setminus (\in_{\mathsf{T}} \cdot \beta_k)) \\
\equiv & \quad \{ \ (R \cdot f) \setminus S = f^\circ \cdot (R \setminus S) \text{ and } R \setminus (S \cdot f) = (R \setminus S) \cdot f \text{ , for } f := \beta_k \ \}
\end{array}
$$

[9] Another — much stronger — choice could have been

$$\leq_{\mathsf{T}_1 \cdot \mathsf{T}_2} \triangleq (\in_{\mathsf{T}_1} \setminus \leq_{\mathsf{T}_2}) \cdot \in_{\mathsf{T}_1}$$

$$
\begin{array}{ll}
& X \subseteq \bigcap_{k\in K}(\beta_k^\circ \cdot (\in_{\mathsf{T}} \setminus \in_{\mathsf{T}}) \cdot \beta_k) \\
\Leftarrow & \quad \{ \text{ assume } \leq_{\mathsf{T}} \subseteq \in_{\mathsf{T}} \setminus \in_{\mathsf{T}} \ \} \\
& X \subseteq \bigcap_{k\in K}(\beta_k^\circ \cdot \leq_{\mathsf{T}} \cdot \beta_k) \\
\equiv & \quad \{ \text{ since } f \dot{\leq}_{\mathsf{T}} g \ \equiv \ \langle \forall\ k \in K \ ::\ f(\beta_k^\circ \cdot \leq_{\mathsf{T}} \cdot \beta_k) g \rangle \ \} \\
& X \subseteq \dot{\leq}_{\mathsf{T}}
\end{array}
$$

So, $\dot{\leq}_{\mathsf{T}}$ is membership-compatible wherever $\leq_{\mathsf{T}}$ is membership-compatible. Thus the definition

$$
\leq_{\mathsf{T}^K} \triangleq \dot{\leq}_{\mathsf{T}} \tag{37}
$$

chosen in [4].

The preorder definitions so far enable the following result.

Lemma 4.3 $\leq_{\mathsf{T}} = id$, *for every polynomial functor* $\mathsf{T}X = \sum_{i=0}^{n} C_i \times X^i$.

PROOF.

$$
\begin{array}{ll}
& \leq_{\sum_{i=0}^{n} C_i \times X^i} \\
= & \quad \{ \text{ single out constant functor } \} \\
& \leq_{C_0} + \leq_{\sum_{i=1}^{n} C_i \times X^i} \\
= & \quad \{ \text{ choice } \leq_K = id \text{ and } (35) \ \} \\
& id + \sum_{i=1}^{n} \leq_{C_i \times X^i} \\
= & \quad \{ \ (34) \ \} \\
& id + \sum_{i=1}^{n} (\leq_{C_i} \times \leq_{X^i}) \\
= & \quad \{ \text{ as above and } (37) \ \} \\
& id + \sum_{i=1}^{n} (id \times \dot{\leq}_X) \\
= & \quad \{ \ \dot{id} = id \text{ and } \times, + \text{-reflection } \} \\
& id
\end{array}
$$

4.2.5 Powerset

As elsewhere [3,4], we define $\leq_{\mathcal{P}}$ 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):

$$
\begin{array}{ll}
& \leq \subseteq \in_{\mathsf{T}} \setminus \in_{\mathsf{T}} \\
\equiv & \quad \{ \text{ (17) twice and (32) } \} \\
& Atroot \cdot Branch^* \cdot \leq \subseteq \ Atroot \cdot Branch^* \\
\Leftarrow & \quad \{ \text{ monotonicity of composition } \} \\
& Branch^* \cdot \leq \subseteq \ Branch^*
\end{array}
$$

Clearly, $\leq \ \triangleq \ Branch^*$ is a solution to the version of (31) just above: by substitution, one gets $Branch^* \cdot Branch^* \ \subseteq \ Branch^*$, which holds since $Branch^*$ is transitive. Altogether, by chosing this solution,

$$\leq_{\mathsf{T}} \triangleq Branch^* \tag{38}$$

one obtains a quite obvious understanding of recursive datatype inclusion: $t \leq_{\mathsf{T}} t'$ 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 compound refinement preorders, as the following calculation of $\leq_{\mathcal{P}(\mathsf{Id} \times O)^I}$ shows:

$$
\begin{array}{ll}
& \leq_{\mathcal{P}(\mathsf{Id} \times O)^I} \\
= & \quad \{ \text{ (37) } \} \\
& \dot{\leq}_{\mathcal{P} \cdot (\mathsf{Id} \times O)} \\
= & \quad \{ \text{ (36) } \} \\
& \overbrace{\in_{\mathcal{P}} \setminus (\dot{\leq}_{(\mathsf{Id} \times O)} \cdot \in_{\mathcal{P}})} \\
= & \quad \{ \ \mathsf{Id} \times O \text{ is polynomial } \} \\
& \in_{\mathcal{P}} \dot{\setminus} \in_{\mathcal{P}} \\
= & \quad \{ \text{ powerset maximal preorder } \} \\
& \dot{\subseteq}
\end{array}
$$

All in all, we have justified, by agile pointfree calculation, the following (pointwise) refinement preorder definitions proposed in [4]:

$$
\begin{array}{lll}
x \leq_{\mathsf{Id}} y & \mathsf{iff} & x = y \\
x \leq_K y & \mathsf{iff} & x =_K y
\end{array}
$$

$$
\begin{array}{rcl}
x \leq_{\mathsf{T}_1 \times \mathsf{T}_2} y & \text{iff} & \pi_1\ x \leq_{\mathsf{T}_1} \pi_1\ y\ \wedge\ \pi_2\ x \leq_{\mathsf{T}_2} \pi_2\ y \\
x \leq_{\mathsf{T}_1 + \mathsf{T}_2} y & \text{iff} & \begin{cases} x = i_1\ x' \wedge y = i_1\ y' & \Rightarrow x' \leq_{\mathsf{T}_1} y' \\ x = i_2\ x' \wedge y = i_2\ y' & \Rightarrow x' \leq_{\mathsf{T}_2} y' \end{cases} \\
x \leq_{\mathsf{T}^K} y & \text{iff} & \forall_{k \in K}.\ x\ k \leq_{\mathsf{T}} y\ k \\
x \leq_{\mathcal{P}\mathsf{T}} y & \text{iff} & \forall_{e \in x} \exists_{e' \in y}.\ e \leq_{\mathsf{T}} e'
\end{array}
$$

This preorder will be referred to in the sequel as *structural inclusion*. Note that *forward* refinement of non deterministic components based on $\leq_{\mathsf{T}}$ captures the classical notion of *non determinism reduction*.

However, $\leq_{\mathsf{T}}$ is inadequate for partial components, since via $\leq_{\mathsf{T}+1} = \leq_{\mathsf{T}} + id$ refinement would collapse into bisimilarity instead of entailing an increase of definition on the implementation side. The alternative proposed in [4],

$$
x \leq^F_{\mathsf{T}+\mathbf{1}} y \quad \text{iff} \quad \begin{cases} x = i_1\ x' \wedge y = i_1\ y' & \Rightarrow x' \leq_{\mathsf{T}} y' \\ x = i_2\ * & \Rightarrow \mathsf{true} \end{cases}
$$

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

$$
\leq^F_{\mathsf{T}+1} \triangleq [i_1 \cdot \leq_{\mathsf{T}}{}^\circ, \top]^\circ \tag{39}
$$

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

$$
\begin{array}{ll}
& (\leq_{\mathsf{T}+1})^\circ \\
= & \quad \{\ \text{(35) and converses}\ \} \\
& \leq_{\mathsf{T}}{}^\circ + id \\
= & \quad \{\ \text{relational coproduct}\ \} \\
& [i_1 \cdot \leq_{\mathsf{T}}{}^\circ, i_2] \\
\subseteq & \quad \{\ i_2 \subseteq \top\ \} \\
& [i_1 \cdot \leq_{\mathsf{T}}{}^\circ, \top] \\
= & \quad \{\ (39)\ \} \\
& (\leq^F_{\mathsf{T}+1})^\circ
\end{array}
$$

That $\leq^F_{\mathsf{T}+1}$ is membership-compatible can be proved in a similar way:

$$
\begin{array}{ll}
& \leq^F_{\mathsf{T}+1} \subseteq \in_{\mathsf{T}+1} \setminus \in_{\mathsf{T}+1} \\
\equiv & \quad \{\ \in_{\mathsf{T}+1} = [\in_{\mathsf{T}}, \bot] = \in_{\mathsf{T}} \cdot i_1^\circ\ ;\ \text{Galois (32)}\}
\end{array}
$$

$$
\begin{array}{ll}
& \in_{\mathsf{T}} \cdot i_1^\circ \cdot [i_1 \cdot \leq_{\mathsf{T}}{}^\circ, \top]^\circ \subseteq \in_{\mathsf{T}} \cdot i_1^\circ \\
\equiv & \quad \{ \text{ converses } \} \\
& [i_1 \cdot \leq_{\mathsf{T}}{}^\circ, \top] \cdot i_1 \cdot \in_{\mathsf{T}}{}^\circ \subseteq i_1 \cdot \in_{\mathsf{T}}{}^\circ \\
\equiv & \quad \{ +\text{-cancellation } \} \\
& i_1 \cdot \leq_{\mathsf{T}}{}^\circ \cdot \in_{\mathsf{T}}{}^\circ \subseteq i_1 \cdot \in_{\mathsf{T}}{}^\circ \\
\equiv & \quad \{ i_1 \text{ is an injection } \} \\
& \leq_{\mathsf{T}}{}^\circ \cdot \in_{\mathsf{T}}{}^\circ \subseteq \in_{\mathsf{T}}{}^\circ \\
\equiv & \quad \{ \text{ converses ; Galois (32)} \} \\
& \leq_{\mathsf{T}} \subseteq \in_{\mathsf{T}} \setminus \in_{\mathsf{T}}
\end{array}
$$

(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 [4], 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 post-composed with a suitable embedding to extend its output interface from O to $O + \mathbf{1}$. In this way, lemma 3.1 is restated as refinement inequation

$$p \,;\, \ulcorner i_1 \urcorner \;\leq^F_{\mathsf{T}}\; p\!\uparrow \tag{40}$$

whose likeness to its *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

$$\overline{a_{p;\ulcorner i_1 \urcorner}} \cdot h \;\dot{\leq}^F_{\mathsf{T}}\; \mathsf{T}h \cdot \overline{a_{p\uparrow}}$$

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

$$a_{p;\ulcorner i_1 \urcorner} \overbrace{\dot{\leq}^F_{\mathsf{B}(\_\times(O+\mathbf{1}))}} a_{p\uparrow}$$

cf. diagram

$$\begin{array}{ccc} \mathsf{B}(U_p \times (O+\mathbf{1})) & \xleftarrow{a_{p;\ulcorner i_1 \urcorner}} & U_p \times I \\ {\scriptstyle \mathsf{B}(id\times(id+id))}\uparrow & & \uparrow{\scriptstyle id} \\ \mathsf{B}(U_p \times (O+\mathbf{1})) & \xleftarrow[a_{p\uparrow}]{} & U_p \times I \end{array}$$

or

$$\xi_{\mathsf{B}} \cdot a_{p;\ulcorner i_1 \urcorner} \dot{\leq}^F_{\mathsf{B}_+(\_\times(O+\mathbf{1}))+\mathbf{1}} \xi_{\mathsf{B}} \cdot a_{p\uparrow}$$

in order to bring the 'failure' version of the preorder into play. Recall that $a_{p;\ulcorner i_1 \urcorner} = \mathsf{B}(id \times i_1) \cdot a_p$ and definition 3.2. For notation economy, we factor the (long) chained composition which defines $a_{p\uparrow}$ as follows

$$a_{p\uparrow} = \xi^\circ_{\mathsf{B}} \cdot [i_1, \xi_{\mathsf{B}} \cdot \eta_{\mathsf{B}}] \cdot (\mathsf{B}_+(id \times i_1) + (id \times i_2)) \cdot a' \tag{41}$$
$$a' = (\pi_2 + id) \cdot \mathsf{dr} \cdot a'' \tag{42}$$
$$a'' = (id \times \xi_{\mathsf{B}} \cdot a_p) \cdot \mathsf{a} \cdot (\triangle \times id) \tag{43}$$

Then we calculate:

$$\begin{array}{ll} & \xi_{\mathsf{B}} \cdot a_{p;\ulcorner i_1 \urcorner} \overbrace{\dot{\leq}^F_{\mathsf{B}_+(\_\times(O+\mathbf{1}))+\mathbf{1}}} \xi_{\mathsf{B}} \cdot a_{p\uparrow} \\ \equiv & \quad \{ \text{ (41) ; } \xi_{\mathsf{B}} \cdot \xi^\circ_{\mathsf{B}} = id \} \\ & \xi_{\mathsf{B}} \cdot \mathsf{B}(id \times i_1) \cdot a_p \overbrace{\dot{\leq}^F_{\mathsf{B}_+(\_\times(O+\mathbf{1}))+1}} [i_1, \xi_{\mathsf{B}} \cdot \eta_{\mathsf{B}}] \cdot (\mathsf{B}_+(id \times i_1) + (id \times i_2)) \cdot a' \\ \equiv & \quad \{ \leq^F_{\mathsf{B}_+(\_\times(O+\mathbf{1}))+1} = [i_1 \cdot \leq^\circ_{\mathsf{B}_+}, \top]^\circ \text{ thanks to (39) and lemma 4.3} \} \\ & \xi_{\mathsf{B}} \cdot \mathsf{B}(id \times i_1) \cdot a_p \overbrace{[i_1 \cdot \leq^\circ_{\mathsf{B}_+}, \top]^\circ} [i_1, \xi_{\mathsf{B}} \cdot \eta_{\mathsf{B}}] \cdot (\mathsf{B}_+(id \times i_1) + (id \times i_2)) \cdot a' \\ \equiv & \quad \{ \text{ (A.2) ; converses ; (28) } \} \\ & [i_1, \xi_{\mathsf{B}} \cdot \eta_{\mathsf{B}}] \cdot (\mathsf{B}_+(id \times i_1) + (id \times i_2)) \cdot a' \ \subseteq \ [i_1 \cdot \leq^\circ_{\mathsf{B}_+}, \top] \cdot \xi_{\mathsf{B}} \cdot \mathsf{B}(id \times i_1) \cdot a_p \\ \equiv & \quad \{ \text{ dropping subscripts B and B}_+ \text{ ; } \xi\text{-natural ; } +\text{-absorption (twice) } \} \\ & [i_1 \cdot \mathsf{B}_+(id \times i_1), \xi \cdot \eta \cdot (id \times i_2)] \cdot a' \ \subseteq \ [i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1), \top] \cdot \xi \cdot a_p \\ \equiv & \quad \{ \text{ definition 3.2 ; } a' = (\pi_2 + id) \cdot \mathsf{dr} \cdot a'' \text{ (42) and } \xi \cdot a_p = \pi_2 \cdot a'' \} \\ & [i_1 \cdot \mathsf{B}_+(id \times i_1), \xi \cdot \eta \cdot (id \times i_2)] \cdot (\pi_2 + id) \cdot \mathsf{dr} \cdot a'' \ \subseteq \ [i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1), \top] \cdot \pi_2 \cdot a'' \\ \Leftarrow & \quad \{ \text{ monotonicity of } (\cdot a'') \} \\ & [i_1 \cdot \mathsf{B}_+(id \times i_1), \xi \cdot \eta \cdot (id \times i_2)] \cdot (\pi_2 + id) \cdot \mathsf{dr} \ \subseteq \ [i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1), \top] \cdot \pi_2 \\ \equiv & \quad \{ \text{ } +\text{-absoption ; shunting on rightmost } \pi_2 \text{ ; } \mathsf{dr}^\circ = [id \times i_1, id \times i_2] \text{ ; converses ; } +\text{-fusion } \} \\ & [i_1 \cdot \mathsf{B}_+(id \times i_1) \cdot \pi_2, \xi \cdot \eta \cdot (id \times i_2)] \cdot [i_1 \cdot \pi_2, i_2 \cdot \pi_2]^\circ \ \subseteq \ [i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1), \top] \\ \equiv & \quad \{ \text{ fact } [R, S] \cdot [U, V]^\circ = R \cdot U^\circ \cup S \cdot V^\circ \text{ ; converses } \} \end{array}$$

$$
\begin{array}{ll}
& i_1 \cdot \mathsf{B}_+(id \times i_1) \cdot \pi_2 \cdot \pi_2^\circ \cdot i_1^\circ \cup \xi \cdot \eta \cdot (id \times i_2) \cdot \pi_2^\circ \cdot i_2^\circ \ \subseteq \ [i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1), \top] \\
\equiv & \quad \{ \cup\text{-universal property} \} \\
& \left\{ \begin{array}{l} i_1 \cdot \mathsf{B}_+(id \times i_1) \cdot \pi_2 \cdot \pi_2^\circ \cdot i_1^\circ \subseteq [i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1), \top] \\ \xi \cdot \eta \cdot (id \times i_2) \cdot \pi_2^\circ \cdot i_2^\circ \subseteq [i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1), \top] \end{array} \right. \\
\equiv & \quad \{ \text{ shunting over } i_1^\circ \text{ and } i_2^\circ \text{ ; } +\text{-cancellation (twice) ; } \pi_2 \cdot \pi_2^\circ = id \} \\
& \left\{ \begin{array}{l} i_1 \cdot \mathsf{B}_+(id \times i_1) \subseteq i_1 \cdot \leq^\circ \cdot \mathsf{B}_+(id \times i_1) \\ \xi \cdot \eta \cdot (id \times i_2) \cdot \pi_2^\circ \subseteq \top \end{array} \right. \\
\equiv & \quad \{ \text{ } i_1 \text{ is an injection ; every relation is at most } \top \} \\
& \mathsf{B}_+(id \times i_1) \subseteq \leq^\circ \cdot \mathsf{B}_+(id \times i_1) \\
\equiv & \quad \{ \text{ (39) ; } \leq \text{ is reflexive } \} \\
& \mathsf{true}
\end{array}
$$

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 p dynamics is modified. This may be a disadvantage in contexts where component p is offered by an external source and has to be deployed *as-it-is*. Typically, as in, *e.g.*, Meyer's *design-by-contract* approaches [22,23], such components are supplied with an interface which caters for any usage constraints p might have. From our modelling point of view, let δ_p encode such an interface. Component p is therefore split into a server (the original p) and front-end δ_p ("δ" 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 $dom\, R_p$, 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 $\phi : \mathbf{2} \longleftarrow I$ be the test for valid inputs as recorded in the interface of component p. Therefore, front-end $\delta_p : I + \mathbf{1} \longleftarrow I$ is defined as the lifting $\ulcorner \Phi \urcorner$ of function

$$\Phi \ = I \xrightarrow{\phi ?} I + I \xrightarrow{id + !} I + \mathbf{1}$$

Using δ_p we may now specify the *try again* (total) version of component p by the following server/front-end aggregation:

$$\delta_p \mathbin{;} (p \boxplus \mathsf{idle}) \ : O + \mathbf{1} \longleftarrow I \tag{44}$$

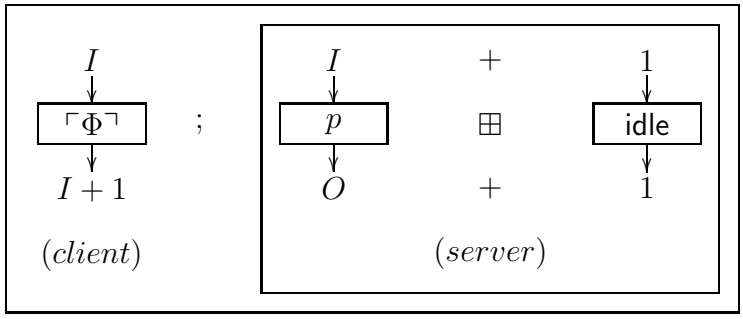


Fig. 4. Client-server "fission" of transposed component (idealized)

where $\mathsf{idle} = \ulcorner id_{\mathbf{1}} \urcorner$, the lifting of identity over $\mathbf{1}$, "absorves" 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 δ_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 δ_p. Formally, p must be of shape

$$p \;=\; p' \,;\ulcorner \pi_2 \urcorner \;\; : O \longleftarrow I$$

where $p' : U_p \times O \longleftarrow I$, 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 $\ulcorner \pi_2 \urcorner$ is, of course, that of hidding the latter on a *stand-alone* deployment of p.

Now, front-end δ_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 δ_p of a component p is another component*

$$\delta_p : I + \mathbf{1} \longleftarrow I + U_p \;=\; \langle u_p \in U_p, \overline{a}_{\delta_p} \rangle$$

where

$$\begin{array}{rcl} a_{\delta_p} \;=\; U_p \times (I + U_p) & \xrightarrow{\;\mathsf{dr}\;} & (U_p \times I) + (U_p \times U_p) \\ & \xrightarrow{\;\mathsf{test}+\mathsf{update}\;} & (U_p \times (I + \mathbf{1})) + U_p \\ & \xrightarrow{\;\eta_{\mathsf{B}} \cdot [\mathsf{id}, \langle \mathsf{id}, i_2 \cdot ! \rangle]\;} & \mathsf{B}(U_p \times (I + \mathbf{1})) \end{array}$$

Service update *is nothing but π_2: its purpose is to refresh the front-end state*

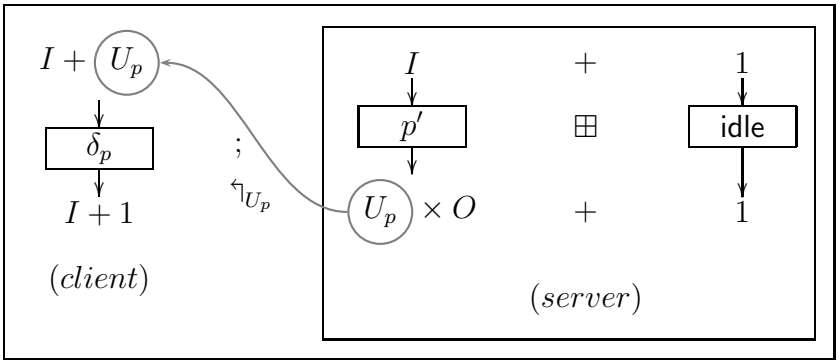


Fig. 5. Client-server "fission" of transposed component

value, whereas

$$\mathsf{test} \;=\; U_p \times I \;\xrightarrow{\mathsf{a}\cdot(\triangle\times\mathsf{id})}\; U_p \times (U_p \times I) \;\xrightarrow{id\times\Gamma(dom\, R_p)}\; U_p \times (U_p \times I + \mathbf{1}) \;\xrightarrow{id\times(\pi_2+\mathsf{id})}\; U_p \times (I + \mathbf{1})$$

The definition of a_{δ_p} amounts to the definition of both services, pre- and post-composed with some *housekeeping* morphisms, among which η_{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 δ_p, using the corresponding update service. This is captured by component algebra expression

$$(\delta_p \,;\, (p' \boxplus \mathsf{idle}))\,\leftarrow_{U_p} \;: O + \mathbf{1} \longleftarrow I \tag{45}$$

— see also Fig. 5 — in which combinator $p\,\leftarrow_Z$, to be defined next, belongs to the family of feedback operators studied in [16].

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 $p : Z \times O + P \longleftarrow I + Z$ to*

$$p\,\leftarrow_Z : O + P \longleftarrow I \;=\; \langle u_p \in U_p, \overline{a}_{p\leftarrow_Z}\rangle$$

where

$$\begin{aligned} a_{p\leftarrow_Z} \;= U_p \times I \;&\xrightarrow{id\times i_1}\; U_p \times (I + Z) \;\xrightarrow{a_p}\; \mathsf{B}(U_p \times (Z \times O + P)) \\ &\xrightarrow{\mathsf{B}((\mathsf{a}^\circ+\mathsf{id})\cdot\mathsf{dr})}\; \mathsf{B}((U_p \times Z) \times O) + U_p \times P) \\ &\xrightarrow{\mathsf{B}((id\times i_2)\times\mathsf{id})+\mathsf{id})}\; \mathsf{B}((U_p \times (I + Z)) \times O) + U_p \times P) \end{aligned}$$

$$
\begin{array}{l}
\xrightarrow{\mathsf{B}((a_p \times \mathsf{id})+\mathsf{id})} \mathsf{B}(\mathsf{B}(U_p \times (Z \times O + P)) \times O) + U_p \times P) \\
\xrightarrow{\mathsf{B}((\mathsf{B}\pi_1 \times \mathsf{id})+\mathsf{id})} \mathsf{B}(\mathsf{B}U_p \times O + U_p \times P) \\
\xrightarrow{\mathsf{B}(\tau_r+\eta)} \mathsf{B}(\mathsf{B}(U_p \times O) + \mathsf{B}(U_p \times P)) \\
\xrightarrow{\mathsf{B}[\mathsf{B}i_1,\mathsf{B}i_2]} \mathsf{BB}((U_p \times O) + (U_p \times P)) \\
\xrightarrow{\mu \cdot \mathsf{BB}\mathsf{dr}^\circ} \mathsf{B}(U_p \times (O + P))
\end{array}
$$

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 $p\uparrow$ can 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 $p : O \longleftarrow I$ be a partial component. Then*

$$p\uparrow \ \sim (\delta_p \,;\, (p' \boxplus \mathsf{idle}))^{\leftarrow}{}_{U_p} \tag{46}$$

PROOF. In the style of [16,17] this equation is proved by the identification of a coalgebra morphism $h : U_p \longleftarrow U_p \times (U_p \times \mathbf{1})$ 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 [24]).

□

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 [16,17,4]. In such a framework our intention was to discuss formally how the behaviour of a partial component could be extended with *try-again* cycles preventing from eventual collapse. This lead 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 [4,5]) 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 [4] 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 [4]. Following a similar approach adopted elsewhere in studying conventional operation refinement [25], it was decided to re-frame the theory of [4] 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 to 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 [26], 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.

References

[1] R. Bird, Functional Programming Using Haskell, Series in Computer Science, Prentice-Hall International, 1998.

[2] J. N. Oliveira, Calculate databases with 'simplicity', presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK. (September 2004).

[3] J. N. Oliveira, C. J. Rodrigues, Transposing relations: From *Maybe* functions to hash tables, in: D. Kozen (Ed.), 7th International Conference on Mathematics of Program Construction, Springer Lect. Notes Comp. Sci. (3125), 2004, pp. 334–356.

[4] S. Meng, L. S. Barbosa, On refinement of generic software components, in: C. Rettray, S. Maharaj, C. Shankland (Eds.), 10th Int. Conf. Algebraic Methods and Software Technology (AMAST), Springer Lect. Notes Comp. Sci. (3116), Stirling, 2004, pp. 506–520, best Student Co-authored Paper Award.

[5] S. Meng, L. S. Barbosa, Components as coalgebras: The refinement dimension, Theor. Comp. Sci. 351 (2006) 276–294.

[6] R. Backhouse, Mathematics of Program Construction, Univ. of Nottingham, 2004, draft of book in preparation. 608 pages.

[7] R. Bird, O. Moor, The Algebra of Programming, Series in Computer Science, Prentice-Hall International, 1997.

[8] G. Pfaff, P. Hagen, The Seeheim Workshop on User Interface Management Systems, Springer-Verlag, Berlim, 1985.

[9] C. B. Jones, Systematic Software Development Using VDM, Series in Computer Science, Prentice-Hall International, 1986.

[10] J. Fitzgerald, P. G. Larsen, Modelling Systems: Pratical Tools and Techniques in Software Development, Cambridge University Press, 1998.

[11] J. M. Spivey, The Z Notation: A Reference Manual (2nd ed), Series in Computer Science, Prentice-Hall International, 1992.

[12] J. R. Abrial, The B Book: Assigning Programs to Meanings, Cambridge University Press, 1996.

[13] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, M. Verhoef, Validated Designs for Object–oriented Systems, Springer, New York, 2005.

[14] J. P. Bowen, P. T. Breuer, K. C. Lano, Formal specifications in software maintenance: From code to Z^{++} and back again, Information and Software Technology 35 (11/12) (1993) 679–690.

[15] J. Rutten, Universal coalgebra: A theory of systems, Theor. Comp. Sci. 249 (1) (2000) 3–80, (Revised version of CWI Techn. Rep. CS-R9652, 1996).

[16] L. S. Barbosa, J. N. Oliveira, State-based components made generic, in: H. P. Gumm (Ed.), CMCS'03, Elect. Notes in Theor. Comp. Sci., Vol. 82.1, Elsevier, 2003.

[17] L. S. Barbosa, Towards a Calculus of State-based Software Components, Journal of Universal Computer Science 9 (8) (2003) 891–909.

[18] L. S. Barbosa, Components as processes: An exercise in coalgebraic modeling, in: S. F. Smith, C. L. Talcott (Eds.), FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems, Kluwer Academic Publishers, 2000, pp. 397–417.

[19] P. F. Hoogendijk, A generic theory of datatypes, Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology (1996).

[20] R. C. Backhouse, P. F. Hoogendijk, Elements of a relational theory of datatypes, in: B. Möller, H. Partsch, S. Schuman (Eds.), Formal Program Development, Springer Lect. Notes Comp. Sci. (755), 1993, pp. 7–42.

[21] B. Jacobs, J. Hughes, Simulations in coalgebra, in: H. P. Gumm (Ed.), CMCS'03, Elect. Notes in Theor. Comp. Sci., Vol. 82.1, Warsaw, 2003.

[22] B. Meyer, Object-Oriented Software Construction (2nd ed.), Series in Computer Science, Prentice-Hall International, 1997.

[23] H. Jifeng, L. Zhiming, L. Xiaoshan, A contract-oriented approach to component-based programming, in: Z. Liu (Ed.), Proc. of FACS'03, (*Formal Approaches to Component Software*), Pisa, 2003.

[24] L. S. Barbosa, J. N. Oliveira, On partial components, PURe-DI TR 2006:02:01, U. Minho, Portugal (2006).

[25] J. N. Oliveira, C. J. Rodrigues, Pointfree factorization of operation refinement, PURe Project technical report (submitted) (Feb. 2006).

[26] N. F. Rodrigues, Generic software slicing applied to the architectural reconstruction of legacy systems, Ph.D. thesis, (in preparation) DI, Universidade do Minho (2006).

[27] J. Visser, J. N. Oliveira, L. Barbosa, J. Ferreira, A. Mendes, Camila Revival: VDM meets Haskell , presented at the Overture Workshop, July 18, co-located with FM 2005: 13th International Symposium on Formal Methods, University of Newcastle upon Tyne, United Kingdom - July 18-22, 2005 (July 2005).

A Lifting orderings to the functional level

Recall the pointwise lifting $\dot{\leq}$ of a preorder $\leq$ to the functional level (27, 28). This construct enjoys a number of properties suitable for calculation (see *e.g.* [6]), of which we present only the ones relevant for this paper. Clearly, for any function h one has

$$f \mathrel{\dot{\leq}} g \;\Rightarrow\; f \cdot h \mathrel{\dot{\leq}} g \cdot h \tag{A.1}$$

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

$$(\dot{\leq}^{\circ}) = (\dot{\leq})^{\circ} \tag{A.2}$$

whose proof is easy to carry out

$$
\begin{array}{ll}
& f\ (\dot{\leq}^{\circ})\ g \\
\equiv & \quad \{\ (28)\ \} \\
& f \subseteq \leq^{\circ} \cdot g \\
\equiv & \quad \{\ \text{converses}\ \} \\
& f^{\circ} \subseteq g^{\circ} \cdot \leq \\
\equiv & \quad \{\ \text{shunting (twice)}\ \} \\
& g \subseteq \leq \cdot f \\
\equiv & \quad \{\ (28)\ \} \\
& g \dot{\leq} f \\
\equiv & \quad \{\ \text{converses}\ \} \\
& f\ (\dot{\leq})^{\circ}\ g
\end{array}
$$

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

$$\overline{f} \dot{\leq} \overline{g} \equiv f \dot{\leq} g \tag{A.3}$$