Transposing Partial Components — an Exercise on Coalgebraic Refinement

PURe Workshop ’05, 12 October 2005

LSB ⧆ JNO

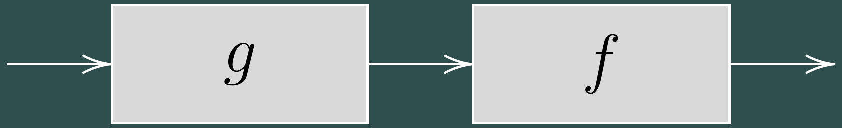
DI/U.Minho Braga, Portugal

Abstract

- By a **partial component** we mean a process which **fails** or **dies** at some stage, thus exhibiting (unexpected) ephemeral behaviour (eg. operating system crash).
- We deal with partial component **totalization** (or transposition) in a way similar to what is done wrt. **partial functions**, cf. exceptions.
- Behavioural transposition adds **try-again** cycles so as to prevent components from collapsing
- We address **client-server** fission of every **try-again** totalized coalgebra into two components — the original one and an added **front-end** — cf. the **“Seeheim (separation) principle”** (1985)

Why Software Components

Component-oriented design relies on **compositionality** — the true basis of software construction — for instance



Recall

- Unix pipes `g | f`
- Functional composition, $\lambda x.f(g(x))$
- etc

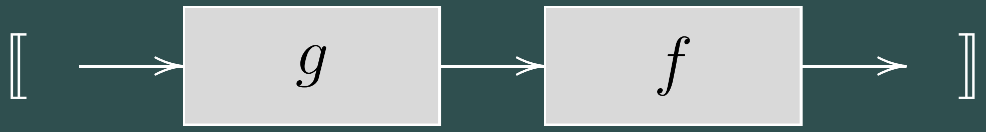
Why Software Components

Ideal world:

$$[\![\longrightarrow \boxed{g} \longrightarrow \boxed{f} \longrightarrow]\!] = [\![f]\!] \cdot [\![g]\!]$$

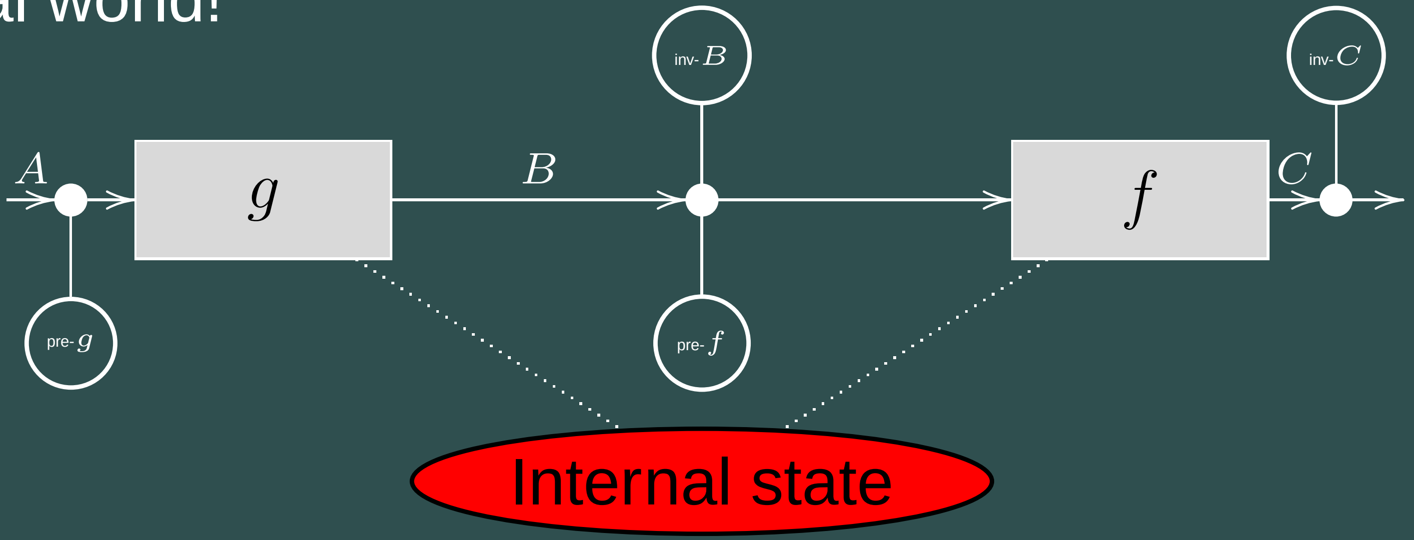
Why Software Components

Ideal world:



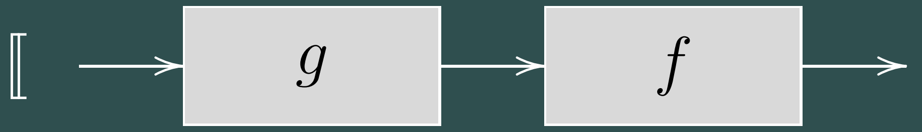
$$= [\![f]\!] \cdot [\![g]\!]$$

Real world!



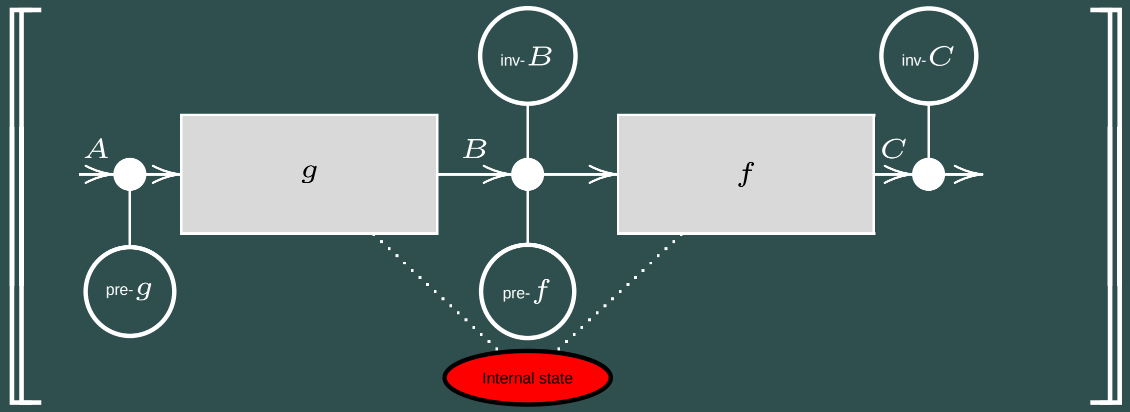
Why Software Components

Ideal world:



$= [\![f]\!] \cdot [\![g]\!]$

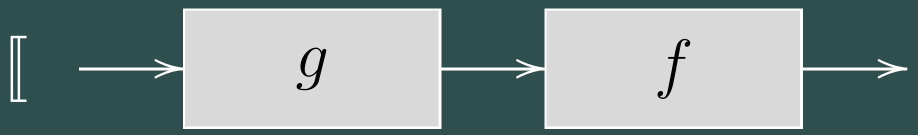
Semantics of real world ?



$= [\![f]\!] \; ? \; [\![g]\!]$

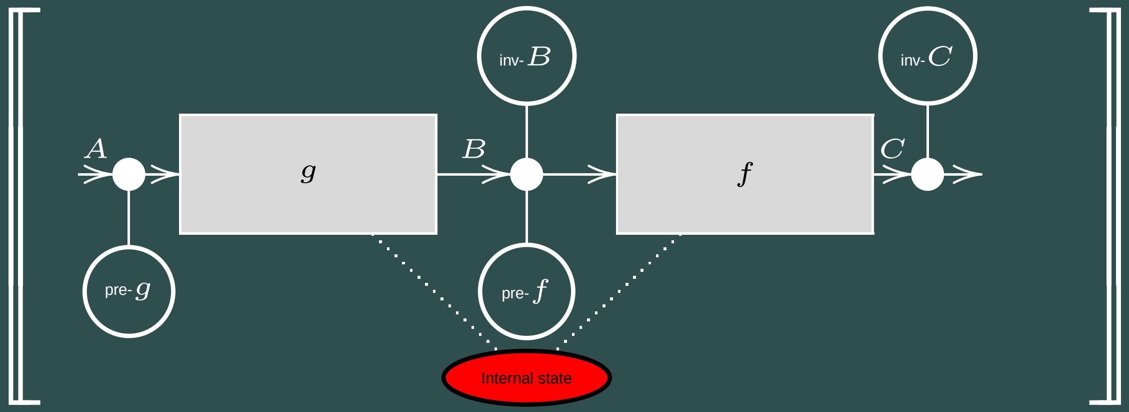
Why Software Components

Ideal world:



$$= [\![f]\!] \cdot [\![g]\!]$$

Semantics of real world ?



$$= [\![f]\!] \; ? \; [\![g]\!]$$

Monadic (Kleisli) $[\![f]\!]$. ! $[\![g]\!]$ replaces $[\![f]\!] \cdot [\![g]\!]$

Why monads

Compare:

$(f \cdot g)a \;=\;$ `let b = g(a) in f(b)`

with

$(f \;\texttt{.!}\; g)a \;=\;$ `do { b <- g(a); f(b) }`

Why monads

Compare:

$$(f \cdot g)a \;=\; \texttt{let b = g(a) in f(b)}$$

with

$$(f \;\texttt{.!}\; g)a \;=\; \texttt{do \{ b <- g(a); f(b) \}}$$

where types are, in the second case, as follows

$$\begin{tikzcd}
A \arrow[r, "g"'] & \mathsf{M}\ B \arrow[d, dotted, no head] & \\
 & B \arrow[r, "f"'] & \mathsf{M}\ C
\end{tikzcd}$$

Why monads

Compare:

$$(f \cdot g)a \;=\; \texttt{let b = g(a) in f(b)}$$

with

$$(f \;\mathtt{.!}\; g)a \;=\; \texttt{do \{ b <- g(a); f(b) \}}$$

In detail:

$$\begin{tikzcd}
A \arrow[r, "g"'] \arrow[rrr, bend left, "f \,\mathtt{.!}\, g"] & \mathsf{M}\;B \arrow[r, "\mathsf{M}\;f"'] \arrow[d, dash, dotted] & \mathsf{M}(\mathsf{M}\;C) \arrow[r, "\mu"'] \arrow[d, dash, dotted] & \mathsf{M}\;C \\
 & B \arrow[r, "f"'] & \mathsf{M}\;C &
\end{tikzcd}$$

Partiality and the `Error` monad

Which monad M ? A popular choice for handling partiality is

- datatype

```
data Error a = Err String | Ok a
```

- that is, monad

```
instance Monad Error where
    return b = Ok b
    (Err e) >>= f  =  Err e
    (Ok a)  >>= f  =  f a
```

First experiment

“Monadify” normal functions,

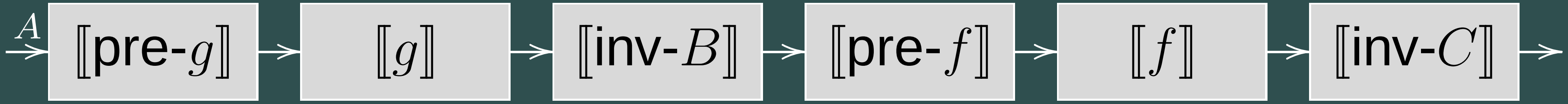
$$[\![f]\!] \;=\; Ok \cdot f$$

and convert conditions and invariants to monadic **partial identities**, eg.

$$\begin{aligned}[\![inv]\!]\; a \;=\;& if\ (inv\ a)\\ & then\ (Ok\ a\\ & else\ Err\ \text{''}Invariant\ violation\text{''}\end{aligned}$$

Back to the real world

In this way, we get a very simple, “pipelined” approach to composition



$$\xrightarrow{A} [\![\text{pre-}g]\!] \to [\![g]\!] \to [\![\text{inv-}B]\!] \to [\![\text{pre-}f]\!] \to [\![f]\!] \to [\![\text{inv-}C]\!] \to$$

where the arrows are `Error`-monadic — think of `(.!)` instead of $(\cdot)$ — that is

```
do { pre-g a;
     b <- g a;
     inv-B b; pre-f b;
     c <- f b;
     inv-C c
}
```

Changing the evaluation mode

See

Camila Revival: ***VDM*** *meets Haskell*

by

J. Visser et al (Overture Workshop last July,
Newcastle UK)

for alternatives to the error monad and a generic (type class based) way of commuting among them in a Haskell interpreter of VDM.

From functions to objects

```
class stackObj

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

instance variables
  stack : Stack := [];

operations

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

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

end stackObj
```

Method semantics

- Semantics of `PUSH` is a function of type

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

  (S abbreviates `Stack` and 1 abbreviates `()` in VDM^{++}.)

- Semantics of `POP` is of type

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

  However, $[\![\texttt{POP}]\!]$ is **not** a (total) function, because of its precondition.

- Reactive **partiality** is more the rule than the exception in formal modelling.

Nondeterministic objects

```
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 := tl \ {r};
                return r)
  pre s <> {};

end unOrdCol
```

Relational semantics

- `stackObj` and `unOrdCol` are similar in shape
- However, `GET` (the counterpart of `POP`) is not only partial but also **nondeterministic**
- All in all, the arrows above have to be regarded as denoting **binary relations**
- Let's **package** `PUT` and `GET` (or `PUSH` and `POP`) together:

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

Going coalgebraic

- Since $\times$ distributes over $+$, we can factor out S ,

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

  where dr is the **distribute-right** isomorphism and R° denotes the **converse** of R .

Going coalgebraic

- Since $\times$ distributes over $+$, we can factor out S ,

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

- Since every R (a relation) has a powerset transpose ΛR (a function),

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

Going coalgebraic

- Since $\times$ distributes over $+$, we can factor out S ,

$$\mathsf{dr}^\circ \cdot (\llbracket \mathsf{PUT} \rrbracket + \llbracket \mathsf{GET} \rrbracket) \cdot \mathsf{dr} : S \times (1 + A) \longleftarrow S \times (A + 1)$$

- ... we can convert the above relational semantics into

$$\Lambda(\mathsf{dr}^\circ \cdot (\llbracket \mathsf{PUT} \rrbracket + \llbracket \mathsf{GET} \rrbracket) \cdot \mathsf{dr})$$

Going coalgebraic

- Since $\times$ distributes over $+$, we can factor out S ,

$$\mathsf{dr}^\circ \cdot (\llbracket \mathsf{PUT} \rrbracket + \llbracket \mathsf{GET} \rrbracket) \cdot \mathsf{dr} : S \times (1 + A) \longleftarrow S \times (A + 1)$$

- . . . we can convert the above relational semantics into

$$\Lambda(\mathsf{dr}^\circ \cdot (\llbracket \mathsf{PUT} \rrbracket + \llbracket \mathsf{GET} \rrbracket) \cdot \mathsf{dr})$$

— a function of type $\mathcal{P}(S \times (1 + A)) \longleftarrow S \times (A + 1)$
which can — finally — be curried into coalgebra

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

In general

Given (nondeterministic) component p hiding internal state U_p and offering methods $M_{i=1,n}$ with public interface $M_i : O_i \longleftarrow I_i$ its semantics will be captured by coalgebra

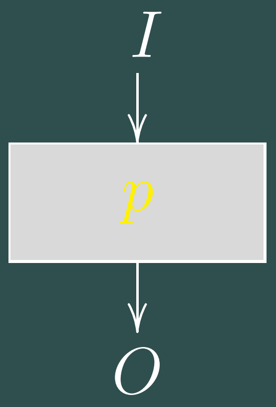
$$\overline{\Lambda(\mathsf{dr}^\circ \cdot (\sum_{i=1}^{n} [\![M_i]\!]) \cdot \mathsf{dr})}$$

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 $\sum_{i=1}^{n} I_i$ (For simplicity, dr is assumed extended to the n -ary case.)

Components as coalgebras

A (generic) component p with input interface I and output interface O

$$p : O \longleftarrow I$$



is a pair

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

where

- point u_p is the ‘initial’ or ‘seed’ state.
- B is an arbitrary **strong** monad.

Behavioural semantics

The semantics of p is the behaviour produced by starting at initial state u_p and **unfolding** over coalgebra $\overline{a}_p$:

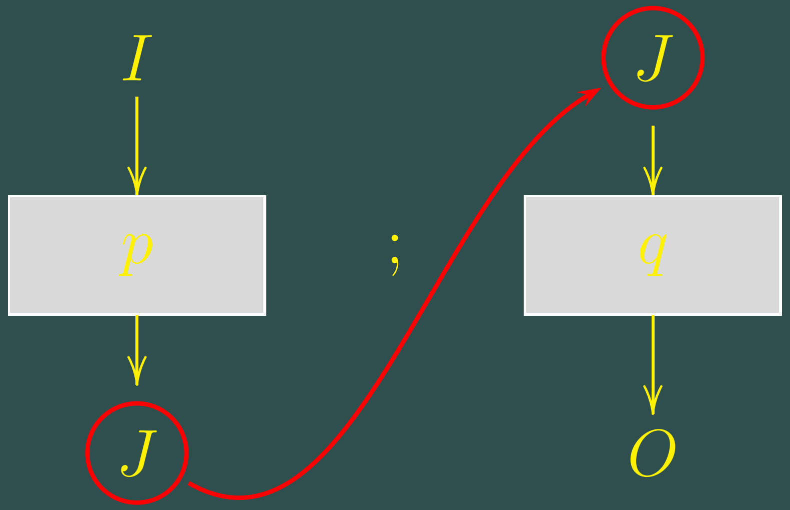
$$\llbracket p \rrbracket \;=\; [\!(\overline{a}_p)\!]\, u_p$$

$$\begin{tikzcd}
\mathsf{B}(\nu \times O)^I & \nu \arrow[l, "\omega"'] \\
\mathsf{B}(U_p \times O)^I \arrow[u, "\mathsf{B}({[\!(\overline{a}_p)\!]} \times O)^I"] & U_p \arrow[l, "\overline{a}_p"] \arrow[u, "{[\!(\overline{a}_p)\!]}"']
\end{tikzcd}$$

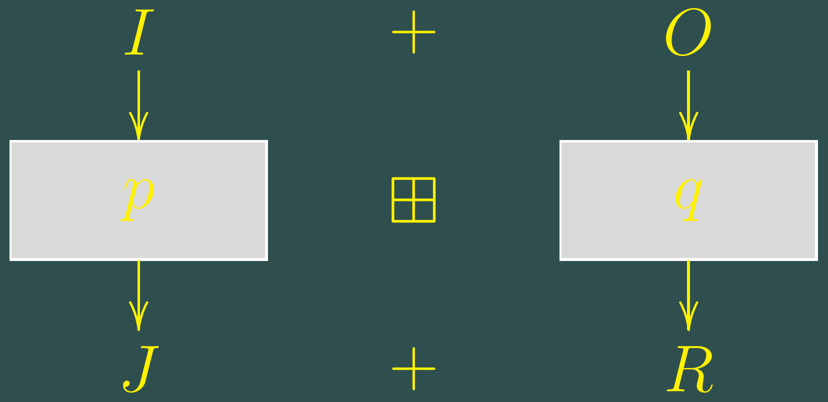
That is, an action will not simply produce an output and a continuation state, but a B -structure of such pairs.
Monad B's **unit** (η) and **multiplication** (μ) provide, respectively, a value embedding and a ‘flatten’ operation to unravel nested behavioural annotations.

Component combinator (algebra)

Pipeline $p \,;\, q$:



Choice $p \boxplus q$:



Behaviour partiality

Wherever B can be decomposed into a **maybe** shape,

$$\begin{tikzcd}\mathsf{B} \arrow[rr, bend left, "\xi_{\mathsf{B}}^{\circ}"] \arrow[rr, phantom, "\cong"] & & \mathsf{B}_{+} + \mathbf{1} \arrow[ll, bend left, "\xi_{\mathsf{B}}"]\end{tikzcd}$$

— eg. $\mathcal{P} \cong \mathcal{P}_{+} + 1$, $Maybe \cong \mathsf{Id} + \mathbf{1}$ — p will be referred to as a **partial component**: it may stop in presence of a precondition or invariant violation and its coalgebra is $Maybe$ -transposable into simple relation $\mathsf{B}_{+}(U_p \times O) \xleftarrow{R_p} U_p \times I$ such that $a_p = \xi_{\mathsf{B}} \cdot \Gamma\ R_p$ and $R_p = (\xi_{\mathsf{B}} \cdot \iota_1)^{\circ} \cdot a_p$.

Behaviour totalization

- Transpose partial component $p : O \longleftarrow I$ into $p\!\uparrow : O + \mathbf{1} \longleftarrow I$ such that
- output of type $\mathbf{1}$ bears the informal meaning **“please try again”**.
- Details about $a_{p\uparrow}$:

$$
\begin{aligned}
a_{p\uparrow} \;=\; & U_p \times I \xrightarrow{\triangle \times id} (U_p \times U_p) \times I \xrightarrow{\mathsf{a}} U_p \times (U_p \times I) \xrightarrow{\mathsf{id} \times \boxed{a_p}} U_p \times \mathsf{B}(U_p \times O) \\
& \xrightarrow{\mathsf{id} \times \xi_{\mathsf{B}}} U_p \times (\mathsf{B}_{+}(U_p \times O) + \mathbf{1}) \xrightarrow{\mathsf{dr}} U_p \times \mathsf{B}_{+}(U_p \times O) + (U_p \times \mathbf{1}) \\
& \xrightarrow{\pi_2 + \mathsf{id}} \mathsf{B}_{+}(U_p \times O) + (U_p \times \mathbf{1}) \\
& \xrightarrow{\mathsf{B}_{+}(id \times \iota_1) + id \times \iota_2} \mathsf{B}_{+}(U_p \times (O + \mathbf{1})) + U_p \times (O + \mathbf{1}) \\
& \xrightarrow{[\iota_1, \xi_{\mathsf{B}} \cdot \eta_{\mathsf{B}}]} \mathsf{B}_{+}(U_p \times (O + \mathbf{1})) + \mathbf{1} \xrightarrow{\xi_{\mathsf{B}}^{\circ}} \mathsf{B}(U_p \times (O + \mathbf{1}))
\end{aligned}
$$

Totalization as refinement

We have developed an equational (pointfree) proof for the following result:

> ***Lemma**: Component $p\!\uparrow\, : O + \mathbf{1} \longleftarrow I$ is a **backward** refinement of $p : O \longleftarrow I$, with respect to the **failure** refinement order $\leq^{F}_{\mathsf{T}}$, for $\mathsf{T} + 1 \cong \mathsf{B}(\mathsf{Id} \times O)$.*

We need to explain

- What “backward” refinement means
- The $\leq^{F}_{\mathsf{T}}$ failure refinement order.

Backward refinement

Let T be the behaviour shape of components $q = (u_q, \overline{a_q})$ and $p = (u_p, \overline{a_p})$ sharing the same state space U .

Then q is said to be a **backward** refinement of p wrt. preorder $\mathsf{T}U \xleftarrow{\leq_{\mathsf{T}}} \mathsf{T}U$ — written $p \ \leq_{\mathsf{T}} \ q$ — if

$$u_q = u_p$$

$$\overline{a_p} \ \dot{\leq}_{\mathsf{T}} \ \overline{a_q}$$

NB:

(a) this is a special case of a more general definition.

(b) $f \ \dot{\leq} \ g$ means $f \subseteq \leq \cdot \, g$ — that is, $f \ x \leq g \ x$ for all x .

Refinement preorders

Refinement preorders are membership-compatible preorders:

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

that is, such that

$$\in_{\mathsf{T}} \cdot \leq \;\subseteq\; \in_{\mathsf{T}}$$

One is free to choose $\leq$ in the range

$$id \;\subseteq \leq \subseteq\; \in_{\mathsf{T}} \setminus \in_{\mathsf{T}}$$

Our choice

By solving the above (in)equation we have arrived at the following preorder (defined by induction on the structure of T):

$$
\begin{aligned}
\leq_{\mathsf{Id}} &= id \\
\leq_{\mathsf{K}} &= id \\
\leq_{\mathsf{T}_1 \times \mathsf{T}_2} &= \leq_{\mathsf{T}_1} \times \leq_{\mathsf{T}_2} \\
\leq_{\mathsf{T}_1 + \mathsf{T}_2} &= \leq_{\mathsf{T}_1} + \leq_{\mathsf{T}_2} \\
\leq_{\mathsf{T}_1 \cdot \mathsf{T}_2} &= (\in_{\mathsf{T}_1} \setminus \leq_{\mathsf{T}_2}) \cdot \in_{\mathsf{T}_1} \\
\leq_{\mathsf{T}^K} &= \dot{\leq}_{\mathsf{T}} \\
\leq_{\mathcal{P}} &= \in_{\mathcal{P}} \setminus \in_{\mathcal{P}}
\end{aligned}
$$

Our choice

Pointwise equivalent:

$$
\begin{array}{rcl}
x \leq_{\mathsf{Id}} y & \equiv & x = y \\
x \leq_{K} y & \equiv & x =_{K} y \\
x \leq_{\mathsf{T}_1 \times \mathsf{T}_2} y & \equiv & \pi_1\ x \leq_{\mathsf{T}_1} \pi_1\ y \ \wedge\ \pi_2\ x \leq_{\mathsf{T}_2} \pi_2\ y \\
x \leq_{\mathsf{T}_1 + \mathsf{T}_2} y & \equiv & \begin{cases} x = \iota_1\ x' \wedge y = \iota_1\ y' & \Rightarrow x' \leq_{\mathsf{T}_1} y' \\ x = \iota_2\ x' \wedge y = \iota_2\ y' & \Rightarrow x' \leq_{\mathsf{T}_2} y' \end{cases} \\
x \leq_{\mathsf{T}^K} y & \equiv & \forall_{k \in K}.\ x\ k \leq_{\mathsf{T}} y\ k \\
x \leq_{\mathcal{P}\mathsf{T}} y & \equiv & \forall_{e \in x} \exists_{e' \in y}.\ e \leq_{\mathsf{T}} e'
\end{array}
$$

The failure refinement order

Increase in definition on the implementation side is ensured by extra clause

$$x \leq^{F}_{\mathsf{T}+1} y \quad \equiv \quad \begin{cases} x = \iota_1\ x' \ \wedge \ y = \iota_1\ y' & \Rightarrow x' \leq_{\mathsf{T}} y' \\ x = \iota_2\ * & \Rightarrow \textsc{True} \end{cases}$$

whose pointfree transform is

$$\leq^{F}_{\mathsf{T}+1} = [\iota_1 \cdot \leq_{\mathsf{T}}{}^{\circ}, \top]^{\circ}$$

So, wherever $a_p(u,i) = \iota_2 *$ and $\overline{a_p} \ \dot{\leq}^{F}_{\mathsf{T}+1} \ \overline{a_q}$ holds, then either $a_q(u,i) = a_p(u,i)$ or, for some y, $a_q(u,i) = \iota_1\ y$.

“Client-server fission”

Motivation:

- **“Seeheim principle”** (1985): separate partiality handler from (partial) server, typically

  Application = Client (GUI) + Server (IS)

- In out context, we want to split a given **try-again** totalized coalgebra into two coalgebraic components — the original one and an added **front-end**

“Client-server fission”

Motivation:

- **“Seeheim principle”** (1985): separate partiality handler from (partial) server, typically

  Application = Client (GUI) + Server (IS)

- Two versions:
  - Idealized situation first — an “oracle” tells the client when it is safe to invoke the server
  - Real situation — client interacts with the server before enabling a partial action

“Client-server fission” (idealized)

Recall how functions are “lifted” to components: $f : B \longleftarrow A$ becomes $\ulcorner f \urcorner : B \longleftarrow A$ over $\mathbf{1}$ such that

$$a_{\ulcorner f \urcorner} = \qquad \mathbf{1} \times A \xrightarrow{\eta \cdot (\mathsf{id} \times f)} \mathsf{B}(\mathbf{1} \times B)$$

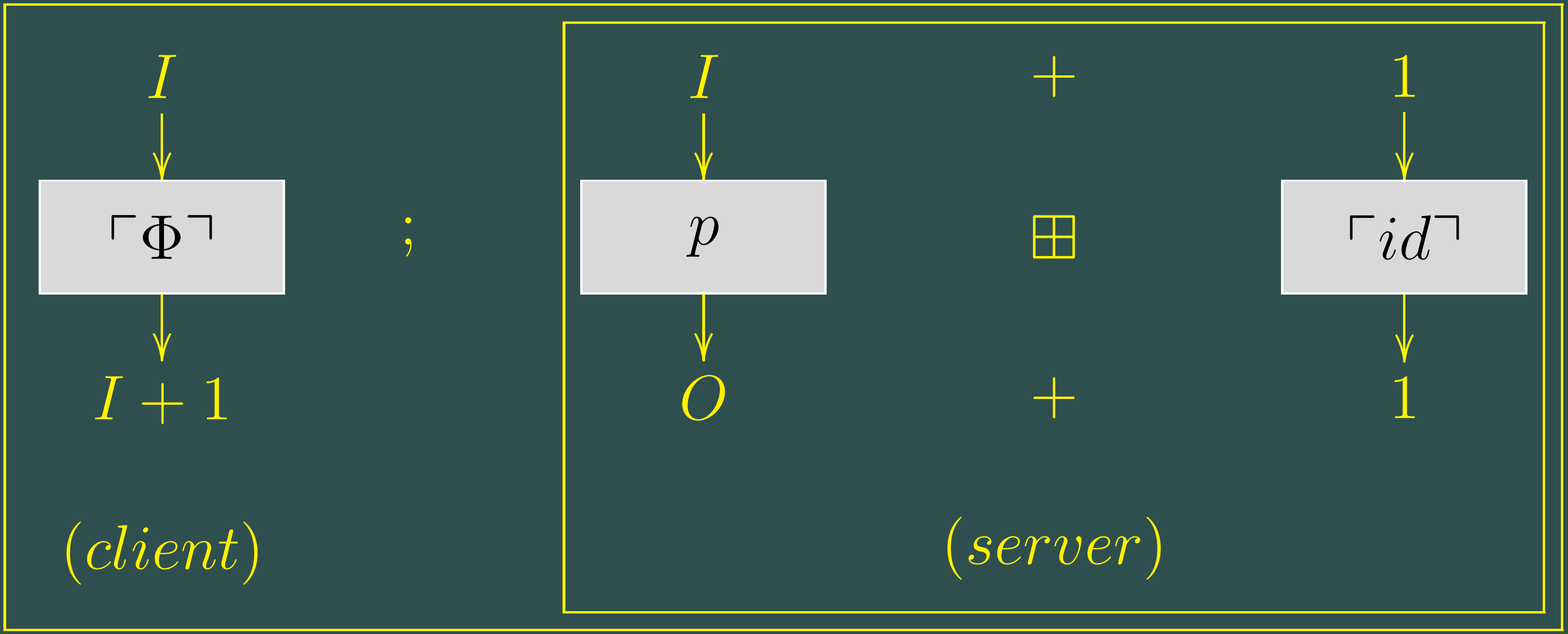
Then, given…

“Client-server fission” (idealized)

…“oracle”

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

telling which actions in I can be safely performed, **try-again** totalized component $p\uparrow$ would be bisimilar to



“Client-server fission” (idealized)

Thus the architectural expression

$$\mathsf{front\_end} \,;\, (p \boxplus \mathsf{idle}) \quad : O + \mathbf{1} \longleftarrow I$$

where $\mathsf{front\_end} = \ulcorner \Phi \urcorner$ and $\mathsf{idle} = \ulcorner \mathsf{id}_{\mathbf{1}} \urcorner$.

“Client-server fission” (idealized)

Thus the architectural expression

$$\mathsf{front\_end} \,;\, (p \boxplus \mathsf{idle}) \quad : O + \mathbf{1} \longleftarrow I$$

where $\mathsf{front\_end} = \ulcorner \Phi \urcorner$ and $\mathsf{idle} = \ulcorner \mathsf{id}_{\mathbf{1}} \urcorner$.

However — in reality — executability of a component’s call depends not only on the **input** supplied but also on the current value of p ’s **state** variable.

“Client-server fission” (realistic)

- As this value must be known to the front-end, it should be made available by p as a sort of attribute. It seems reasonable to assume such an attribute as **private**, ie, available only when p is intended to act as a server accessed through a validating front-end.
- So p must be of shape

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

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.

A new front-end for p

$$\begin{tikzcd} I + U_p \arrow[d] \\ \boxed{\mathsf{f\_end}_p} \arrow[d] \\ I + 1 \end{tikzcd} \quad = \quad (u_p \in U_p, \overline{a}_{\mathsf{f\_end}_p})$$

where

$$\boxed{\begin{aligned} a_{\mathsf{f\_end}_p} = U_p \times (I + U_p) & \xrightarrow{\mathsf{dr}} (U_p \times I) + (U_p \times U_p) \\ & \xrightarrow{\mathsf{test}+\mathsf{update}} (U_p \times (I + \mathbf{1})) + U_p \\ & \xrightarrow{\eta_{\mathsf{B}} \cdot [\mathsf{id}, (\mathsf{id}, \iota_2 \cdot !)]} \mathsf{B}(U_p \times (I + \mathbf{1})) \end{aligned}}$$

A new front-end for p

$$\begin{tikzcd} I + U_p \arrow[d] & & \\ \boxed{\mathsf{f\_end}_p} \arrow[d] & = & (u_p \in U_p, \overline{a}_{\mathsf{f\_end}_p}) \\ I + 1 & & \end{tikzcd}$$

where $\mathsf{update} = \pi_2$ and where

$$\boxed{\begin{aligned} \mathsf{test} \; = \; U_p \times I & \xrightarrow{\mathsf{a}\cdot(\Delta\times\mathsf{id})} U_p \times (U_p \times I) \\ & \xrightarrow{\mathsf{id}\times\Gamma(\mathit{dom}\ R_p)} U_p \times (U_p \times I + \mathbf{1}) \\ & \xrightarrow{\mathsf{id}\times(\pi_2+\mathsf{id})} U_p \times (I + \mathbf{1}) \end{aligned}}$$

“Client-server fission” (realistic)

Finally, the server/front-end architecture is defined through a similar aggregation pattern but with an additional step:

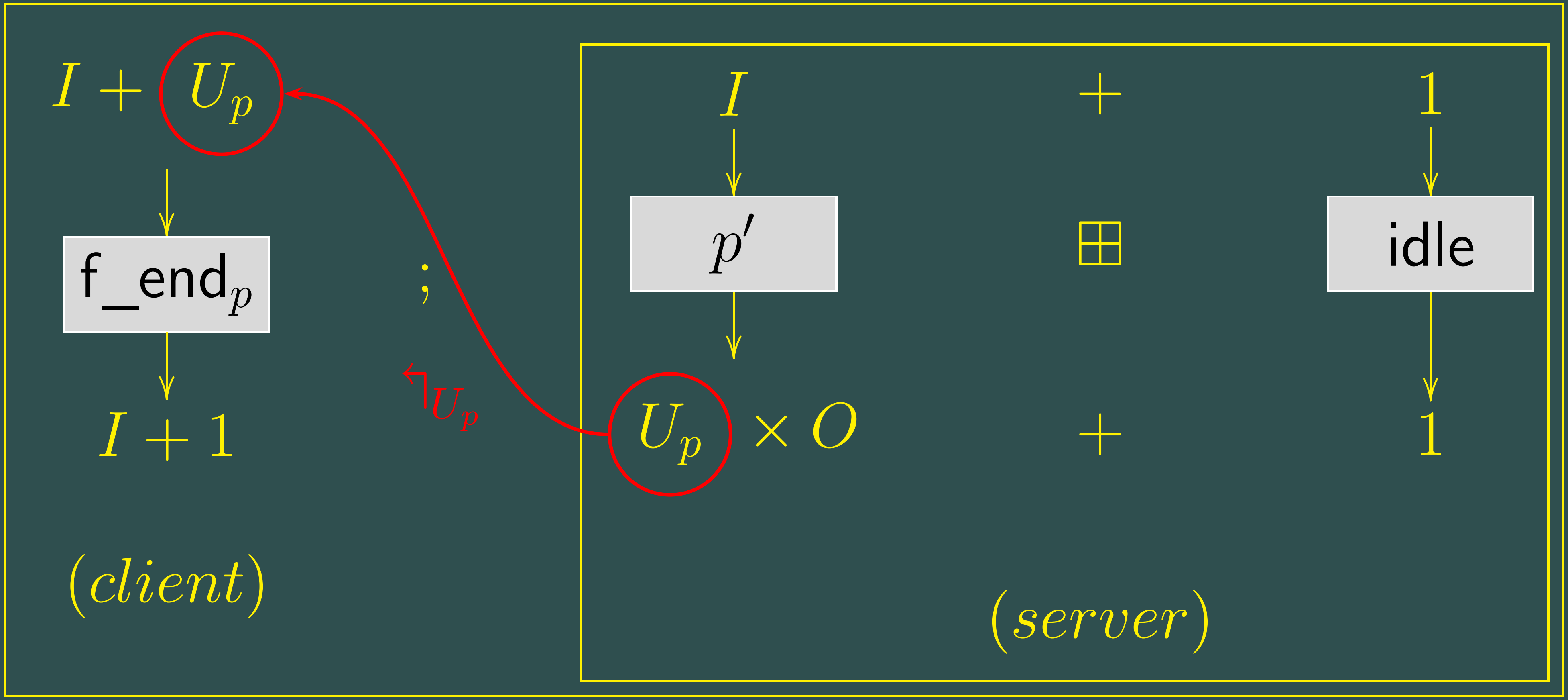
> *On every execution of the server component, the computed value for its state is fed back to* $\mathsf{f\_end}_p$ *, using the corresponding* `update` *service.*

Formally,

$$(\mathsf{f\_end}_p \,;\, (p' \boxplus \mathsf{idle})) \Lsh_{U_p} \; : O + \mathbf{1} \longleftarrow I$$

(See our draft paper for details about the $p \Lsh_X$ combinator)

“Client-server fission” (diagram)



Still missing but not essential: O also fed back to f_end$_p$ for **“beautification”**.

“Client-server fission” lemma

Fission is expressed by the following lemma:

> Given partial component p , its try-again-transpose $p{\uparrow}$ is bisimilar to $(\mathsf{f\_end}_p \mathbin{;} (p' \boxplus \mathsf{idle}))\Lsh_{U_p}$.

This is proved by identifying a coalgebra morphism $h : U_p \longleftarrow U_p \times (U_p \times \mathbf{1})$ connecting the state-spaces of the underlying coalgebras. The obvious choice is $h = \pi_1 \cdot \pi_2$.

Conclusions

- Regarding transposition as a refinement situation entailed the need to extend the combinator algebra ($p \overset{\leftarrow}{\urcorner}_{X}$ is new) and re-visit the underlying theory
- Formal justification of what seemed to be just intuitive
- Re-frame the theory in the **pointfree** relational calculus which makes effective calculations simple and elegant.
- Our calculations would require lengthy and contrived proofs had we resorted to classical pointwise reasoning

Hot topics

- Coalgebriaic refinement theory still “hot”, eg.
  - R_p instead of a_p ?
  - Lindsay Groves’ $\sqsubseteq = \sqsubseteq_{post} \cdot \sqsubseteq_{pre}$ factorization versus forward/backward refinement?
- Build software architecture catalog (eg. **client-server**, **pipe&filter**, **blackboard**, **pier-evolution**, etc) around canonical (generic) coalgebraic expressions (cf. “design patterns”)
- Use slicing, program analysis etc. to classify software systems wrt. to such a catalog
- Think of architectural transformation morphisms (software architecture refinement?)

Appendix: ASM refinement

ASM (=abstract state machines) refinement ordering:

Machine $\mathcal{P}A \xleftarrow{R} A$ implements machine $\mathcal{P}A \xleftarrow{S} A$ — written $S \vdash R$ iff

$$\langle \forall\, a \;:\; (S\, a) \supset \emptyset \;:\; \emptyset \subset (R\, a) \subseteq (S\, a) \rangle$$

where $S\ a$ means the set of states reachable (in machine S) from state a.

References