

Lecture 4: Effect Handlers

Gordon Plotkin

Laboratory for the Foundations of Computer Science, School of Informatics,
University of Edinburgh

Beihang University
October, 2015

Outline

1 Effect Deconstructors

2 Concurrency

Outline

1 Effect Deconstructors

2 Concurrency

Simple exception handler

- The construct is:

$$M^\sigma \text{ handled with } \{\text{raise}_e \mapsto N_e^\sigma\}_{e \in E} : \sigma$$

assuming a finite set of exceptions $E =_{\text{def}} \mathcal{M}[\text{exc}]$

- This evidently does not arise from a definable $1 + |E|$ -ary operation using the exceptions theory.
- Even worse, it cannot be an operation of **any** algebraic theory.
- For suppose we have a suitable operation $\text{handle} : \varepsilon; 1, \text{exc}$ say. Then, in general, we will **not** have:

$$\mathcal{E}[\text{handle}(M, x : \text{exc}.N(x))] = \text{handle}(\mathcal{E}[M], x : \text{exc}.\mathcal{E}[N(x)])$$

Failure to be algebraic

Take $\mathcal{E} = (\lambda y : \text{nat. raise}_{e_1})[\cdot]$, $M = 3$, and $N(x) = \text{raise}_{e_2}$, where $e_1 \neq e_2$. Then we have:

$$\begin{aligned} \models \mathcal{E}[\text{handle}(M, x : \text{exc. } N(x))] &=_{\text{def}} (\lambda y : \text{nat. raise}_{e_1})\text{handle}(3, x : \text{exc. } N(x)) \\ &= (\lambda y : \text{nat. raise}_{e_1})3 \\ &= \text{raise}_{e_1} \end{aligned}$$

and:

$$\begin{aligned} \models \text{handle}(\mathcal{E}[M], x : \text{exc. } \mathcal{E}[N(x)]) &= \text{handle}((\lambda y : \text{nat. raise}_{e_1})3, x : \text{exc. } (\lambda y : \text{nat. raise}_{e_1})N(x)) \\ &= \text{handle}(\text{raise}_{e_1}, x : \text{exc. } (\lambda y : \text{nat. raise}_{e_1})N(x)) \\ &= (\lambda y : \text{nat. raise}_{e_1})N(e_1) \\ &=_{\text{def}} (\lambda y : \text{nat. raise}_{e_1})\text{raise}_{e_2} \\ &= \text{raise}_{e_2} \end{aligned}$$

and the two are **different**.

Understanding the Benton and Kennedy exception handler algebraically

Simple exception handler

$M^{\sigma+E}$ handled with $\{\text{raise}_e \mapsto N_e^{\sigma+E}\}_{e \in E} : \sigma + E$

with E finite. (We are mixing syntax and semantics.)

Benton and Kennedy exception handler

$M^{\sigma+E}$ handled with $\{\text{raise}_e \mapsto N_e^A\}_{e \in E}$ to $x : \sigma$ in $N(x) : A$

Analysis of the semantics of the BK exception handler

- $M \in T_{Ax}(\sigma) = \sigma + E$.
- $\{\text{raise}_e \mapsto N_e^A\}_{e \in E}$ specifies a model of Ax with carrier A (any algebra is!).
- $\sigma \xrightarrow{x:\sigma. N(x)} A$
- The semantics of the BK exception handler is (that of)

$$(x : \sigma. N(x))^\dagger(M)$$

The general algebraic situation

The free model principle, that, for any algebra \mathcal{A} over A satisfying equational axioms A_X , and for any $f : X \rightarrow \mathcal{A}$ there exists a unique homomorphism $f^\dagger : T_{A_X}(X) \rightarrow \mathcal{A}$ such that the following diagram commutes

$$\begin{array}{ccc}
 X & & \\
 \eta \downarrow & \searrow f & \\
 T_{A_X}(X) & \xrightarrow{f^\dagger} & \mathcal{A}
 \end{array}$$

suggests a syntax for, and an interpretation of, *effect deconstructors*. Continuing to mix syntax and semantics, we write:

$M^{T_{A_X}(X)}$ handled with \mathcal{A} to $x : X$ in $N(x) : \mathcal{A}$

Additions to the λ -calculus: Handlers for simple operations

- **Handlers**

$$H ::= \{\text{op}(k_1 : T\tau, \dots, k_n : T\tau) = B_{\text{op}}\}_{\text{op}:n}$$

where $T(\tau) =_{\text{def}} \text{unit} \rightarrow \tau$

$$\frac{\Gamma, k_1 : T\tau, \dots, k_n : T\tau \vdash B_{\text{op}} : \tau \quad (i = 1, n)}{\Gamma \vdash \{\text{op}(k_1 : T\tau, \dots, k_n : T\tau) = B_{\text{op}}\}_{\text{op}:n} : \tau \text{ handler}}$$

- **Handling**

$$M ::= M \text{ handled with } H \text{ to } x : \sigma \text{ in } N$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash H : \tau \text{ handler} \quad \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash M \text{ handled with } H \text{ to } x : \sigma \text{ in } N : \tau}$$

Warning!! Not all handlers are *correct*, i.e., define models, so semantics of a λ -calculus term may not be defined.

How the handlers work

Suppose

$$H ::= \{\text{op}(k_1 : T\tau, \dots, k_n : T\tau) = B_{\text{op}}(k_1, \dots, k_n)\}_{\text{op}:n}$$

then

$$x : \sigma \vdash x \text{ handled with } H \text{ to } x : \sigma \text{ in } N = N$$

and

$$\vdash \text{op}(M_1, \dots, M_n) \text{ handled with } H \text{ to } x : \sigma \text{ in } N = B_{\text{op}}(K_1, \dots, K_n)$$

where

$$K_i = [M_i \text{ handled with } H \text{ to } x : \sigma \text{ in } N]$$

(and $[M] = \lambda x : \text{unit}. M$)

An example: changing the contents of a read-only memory, holding a boolean

Assume there is only one location, storing booleans.

- **The handler** A “temporary state” handler H_{ro} is given by:

$$b : \text{bool} \vdash \{ \text{lookup}(k_1 : T(\tau), k_2 : T(\tau)) \\ = \text{if } b \text{ then } k_1(*) \text{ else } k_2(*) \} : \tau \textbf{ handler}$$

- **Handling** To evaluate a computation $\vdash M : \sigma$, continuing with $x : \sigma \vdash N : \tau$ and forcing any lookup’s to give a value b we use:

$$b : \text{bool} \vdash M \text{ handled with } H_{\text{ro}} \text{ to } x : \sigma \text{ in } N : \tau$$

One may prefer a syntax allowing parametric handlers parameterised on arbitrary types.

How this handler works

The handler H_{ro} is:

$$\{\text{lookup}(k_1 : T(\tau), k_2 : T(\tau)) = \text{if } b \text{ then } k_1(*) \text{ else } k_2(*) \}$$

It works as follows:

$$\begin{aligned} &\vdash \text{lookup}(M_1, M_2) \text{ handled with } H_{\text{ro}} \text{ to } x : \sigma \text{ in } N \\ &= \text{if } b \text{ then } [M_1 \text{ handled with } H_{\text{ro}} \text{ to } x : \sigma \text{ in } N](*) \\ &\quad \text{else } [M_2 \text{ handled with } H_{\text{ro}} \text{ to } x : \sigma \text{ in } N](*) \\ &= \text{if } b \text{ then } M_1 \text{ handled with } H_{\text{ro}} \text{ to } x : \sigma \text{ in } N \\ &\quad \text{else } M_2 \text{ handled with } H_{\text{ro}} \text{ to } x : \sigma \text{ in } N \end{aligned}$$

Additions to the λ -calculus: General operations

Handlers

$$H ::= \{\text{op}_{\mathbf{x}:\mathbf{s}}(k_1 : \mathbf{s}_1 \rightarrow \tau, \dots, k_n : \mathbf{s}_n \rightarrow \tau) = B_{\text{op}}\}_{\text{op}:\mathbf{s};\mathbf{s}_1,\dots,\mathbf{s}_m}$$

$$\frac{\Gamma, \mathbf{x} : \mathbf{s}, k_1 : \mathbf{s}_1 \rightarrow \tau, \dots, k_n : \mathbf{s}_n \rightarrow \tau \vdash B_{\text{op}} : \tau \quad (i = 1, n)}{\Gamma \vdash \{\text{op}_{\mathbf{x}:\mathbf{s}}(k_1 : \mathbf{s}_1 \rightarrow \tau, \dots, k_n : \mathbf{s}_n \rightarrow \tau) = B_{\text{op}}\}_{\text{op}:\mathbf{s};\mathbf{s}_1,\dots,\mathbf{s}_m} : \tau \text{ handler}}$$

How these handlers work

Suppose H is

$$\{\text{op}_{\mathbf{x}:\mathbf{s}}(k_1 : \mathbf{s}_1 \rightarrow \tau, \dots, k_n : \mathbf{s}_n \rightarrow \tau) = B_{\text{op}}(\mathbf{x}, k_1, \dots, k_n)\}_{\text{op}:\mathbf{s};\mathbf{s}_1,\dots,\mathbf{s}_m}$$

then

$$\begin{aligned} \vdash \text{op}_{\mathbf{A}}(\mathbf{x}_1 : \mathbf{s}_1. M_1, \dots, \mathbf{x}_n : \mathbf{s}_n. M_n) \text{ handled with } H \text{ to } x : \sigma \text{ in } N \\ = \text{let } \mathbf{x} : \mathbf{s} \text{ be } \mathbf{A} \text{ in } B_{\text{op}}(\mathbf{x}, K_1, \dots, K_n) \end{aligned}$$

where

$$K_j = \lambda \mathbf{x} : \mathbf{s}. M_j \text{ handled with } H \text{ to } x : \sigma \text{ in } N$$

An example: rollback

When a computation raises an exception while modifying the memory, e.g., when a connection drops during a database transaction, we may want to revert all modifications made during the computation. This behaviour is termed **rollback**.

- **Signature** The (disjoint) union of that for (global) state and exceptions.
- **Axioms** The union of the two sets of equations for global state and for exceptions, together with two commutation equations:

$$\text{lookup}_l(m : \text{nat. raise}_e) = \text{raise}_e$$

$$\text{update}_{l,v}(\text{raise}_e) = \text{raise}_e$$

of which the first is redundant.

- **Monad**

$$T(X) = ((S \times X) + E)^S$$

Exception handler for rollback

Assume there is only one location l_0 .

- **The handler** A “rollback to n ” handler H_{rollback} is given by:

$$n : \text{nat} \vdash \text{raise}_{e:\text{exc}} = \text{update}_{l_0, n}(N_{\text{raise}}(e)) : \tau \text{ handler}$$

where $e : \text{exc} \vdash N_{\text{raise}} : \tau$.

- **Handling** To evaluate a computation $\vdash M : \sigma$, continuing with $x : \sigma \vdash N : \tau$ if no exception is raised, and otherwise rolling back to the initial state and executing N_{raise} with the exception raised, we use:

$$\vdash \text{lookup}_{l_0}(n : \text{nat}. M \text{ handled with } H_{\text{rollback}} \text{ to } x : \sigma \text{ in } N) : \tau$$

Note: One again may prefer a syntax allowing parametric handlers parameterised on arbitrary types.

In the above one would then take n as a parameter, rather than a free variable, and replace N_{raise} by a parameter of type $\text{exc} \rightarrow \tau$.

Additions to the λ -calculus: Handlers with parameters for simple operations

- **Handlers**

$$H ::= \{\text{op}(k_1 : \pi \rightarrow \tau, \dots, k_n : \pi \rightarrow \tau) @ p : \pi = B_{\text{op}}\}_{\text{op}:n}$$

$$\frac{\Gamma, k_1 : \pi \rightarrow \tau, \dots, k_n : \pi \rightarrow \tau, p : \pi \vdash B_{\text{op}} : \tau \quad (\text{op} : n)}{\Gamma \vdash \{\text{op}(k_1 : \pi \rightarrow \tau, \dots, k_n : \pi \rightarrow \tau) @ p : \pi = B_{\text{op}}\}_{\text{op}:n} : \pi \rightarrow \tau \text{ handler}}$$

- **Handling**

$$M ::= M \text{ handled with } H @ P \text{ to } x : \sigma \text{ in } N$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash H : \pi \rightarrow \tau \text{ handler} \quad \Gamma \vdash P : \pi \quad \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash M \text{ handled with } H @ P \text{ to } x : \sigma \text{ in } N : \tau}$$

How parameterised simple handlers work

Suppose H is

$$\{\text{op}(k_1 : \pi \rightarrow \tau, \dots, k_n : \pi \rightarrow \tau) @ p : \pi = B_{\text{op}}(p, k_1, \dots, k_n)\}_{\text{op}:n}$$

then

$$\begin{aligned} \vdash \text{op}(M_1, \dots, M_n) \text{ handled with } H @ P \text{ to } x : \sigma \text{ in } N \\ = \text{let } p : \pi \text{ be } P \text{ in } B_{\text{op}}(p, K_1, \dots, K_n) \end{aligned}$$

where

$$K_i = \lambda p : \pi. M_i \text{ handled with } H @ p \text{ to } x : \sigma \text{ in } N$$

A parameterised handler example: changing the contents of a boolean read-only memory

Assume there is only one location, storing booleans.

- **The handler** The handler H_{ro} is (now):

$$\begin{aligned} & \{\text{lookup}(k_1 : \text{bool} \rightarrow \tau, k_2 : \text{bool} \rightarrow \tau) @ b : \text{bool} \\ & \quad = \text{if } b \text{ then } k_1(b) \text{ else } k_2(b)\} : \text{bool} \rightarrow \tau \textbf{ handler} \end{aligned}$$

- **Handling** To evaluate a computation $\vdash M : \sigma$, continuing with $x : \sigma \vdash N : \tau$ and forcing any lookup's to give a value P we use:

$$\vdash M \text{ handled with } H_{ro} @ P \text{ to } x : \sigma \text{ in } N : \tau$$

How this handler works

The handler H_{ro} is:

$$\{\text{lookup}(k_1 : \text{bool} \rightarrow \tau, k_2 : \text{bool} \rightarrow \tau) @ b : \text{bool} \\ = \text{if } b \text{ then } k_1(b) \text{ else } k_2(b)\}$$

It works as follows:

$\vdash \text{lookup}(M_1, M_2)$ handled with $H_{ro}@true$ to $x : \sigma$ in N

$= \text{if true then } (\lambda b : \text{bool}. M_1 \text{ handled with } H_{ro}@b \text{ to } x : \sigma \text{ in } N)(\text{true}) \\ \text{else } (\lambda b : \text{bool}. M_2 \text{ handled with } H_{ro}@b \text{ to } x : \sigma \text{ in } N)(\text{true})$

$= M_1$ handled with $H_{ro}@true$ to $x : \sigma$ in N

Faking output and curtailing input

The handler H is

$$\begin{aligned} & \{\text{input}(k : \text{nat} \rightarrow (\text{in} \rightarrow \tau))@limit : \text{nat} \\ & \quad = \text{if } limit > 0 \text{ then } \text{input}(y : \text{in}. k(limit - 1)(y)) \\ & \quad \quad \quad \text{else } \text{raise}_{\text{input_session_finished}}(), \\ & \quad \text{output}_{z:\text{out}}(k : \text{nat} \rightarrow \tau)@limit : \text{nat} \\ & \quad = \text{output}_{\text{fake}}(k(limit))\} \end{aligned}$$

It works as follows:

$$\begin{aligned} & \vdash \text{input}(y : \text{in}. M) \text{ handled with } H@limit \text{ to } x : \sigma \text{ in } N \\ & \quad = \text{if } limit > 0 \\ & \quad \quad \text{then } \text{input}(y : \text{in}. M \text{ handled with } H@(limit-1) \text{ to } x : \sigma \text{ in } N) \\ & \quad \quad \text{else } \text{raise}_{\text{input_session_finished}}() \end{aligned}$$

$$\begin{aligned} & \vdash \text{output}_3(M) \text{ handled with } H@limit \text{ to } x : \sigma \text{ in } N \\ & \quad = \text{output}_{\text{fake}}(M \text{ handled with } H@limit \text{ to } x : \sigma \text{ in } N) \end{aligned}$$

A possible treatment of handlers for effects and types

• Effects

$$\alpha \subseteq_{\text{fin}} \Sigma$$

• Handling

$$\frac{M ::= M \text{ handled with } H \text{ to } x : \sigma \text{ in } N \quad \Gamma \vdash M : \sigma! \alpha \quad \Gamma \vdash H : \alpha \text{ to } \tau! \beta \text{ handler} \quad \Gamma, x : \sigma \vdash N : \tau! \beta}{\Gamma \vdash M \text{ handled with } H \text{ to } x : \sigma \text{ in } N : \tau! \beta}$$

• Handlers

$$H ::= \{\text{op}(x_1 : T_\beta(\tau), \dots, x_n : T_\beta(\tau)) \mapsto B_{\text{op}}\}_{\text{op}:n \in \alpha}$$

where $T_\beta(\tau) =_{\text{def}} \text{unit} \xrightarrow{\beta} \tau$

$$\frac{\Gamma, x_1 : T_\beta(\tau), \dots, x_n : T_\beta(\tau) \vdash B_{\text{op}} : \tau! \beta \quad (i = 1, n)}{\Gamma \vdash \{\text{op}(x_1 : T_\beta(\tau), \dots, x_n : T_\beta(\tau)) \mapsto B_{\text{op}}\}_{\text{op}:n \in \alpha} : \alpha \text{ to } \tau! \beta \text{ handler}}$$

Discussion

- The above language is maximal in that arbitrary handlers can be defined. These define interpretations, but not necessarily models. It is up to the programmer to not write meaningless programs.
- One might instead add a proof requirement, à la type theory, so that a program is not well-formed unless a proof has been supplied.
- One might instead consider a two-level version in which only the compiler writers write handlers. Plotkin and Pretnar, ESOP.
- One might consider restricting the handlers that can be written, so that only meaningful programs can be written. Buneman et al comprehension syntax for database programming on collections (= bags = elements of free commutative monoids).
- If one works only with free algebras, so not "real" effects, then all programs are correct and one has an operational semantics. Might be handy programming idiom. Bauer and Pretnar *Eff*

Outline

1 Effect Deconstructors

2 Concurrency

Finite Nondeterminism with deadlock

Working in **Set** we take $T(X) = \mathcal{F}(X)$ the collection of finite subsets of X to model nondeterminism, including an “empty” choice (deadlock).

To create the effects we add two *effect constructors*:

$$\frac{M : \sigma \quad N : \sigma}{M + N : \sigma} \quad \text{NIL} : \sigma$$

Nondeterminism as an algebraic effect

There is a natural equational theory, with signature $+ : 2 \rightarrow 1$, $NIL : 0 \rightarrow 1$ and axioms:

$$\textit{Associativity} \quad (x + y) + z = x + (y + z)$$

$$\textit{Commutativity} \quad x + y = y + x$$

$$\textit{Absorption} \quad x + x = x$$

$$\textit{Zero} \quad NIL + x = x$$

The evident algebra on $\mathcal{F}(X)$ satisfies these equations, interpreting $+$ as \cup , and NIL as \emptyset .

Further: \mathcal{F} is the free algebra monad.

CCS

Syntax

$$\begin{aligned}
 P ::= & a.P \ (a \in \text{Act}) \mid P + Q \mid \text{NIL} && \text{Effect Constructors} \\
 & \mid P \setminus a \mid P[a/b] && \text{Unary Effect Deconstructors} \\
 & && \text{handling } P \\
 & \mid P \mid Q && \text{Binary Effect Deconstructors} \\
 & && \text{handling } P \text{ and } Q
 \end{aligned}$$

Equational theory for the constructors

Signature: $a.- : 1 \rightarrow 1$, for $a \in \text{Act}$, $+ : 2 \rightarrow 1$, $\text{NIL} : 0 \rightarrow 1$

Axioms: That $+$, NIL forms a commutative semilattice, as per finite nondeterminism with deadlock.

Modelling CCS

We model CCS terms as elements of $\text{ST} \stackrel{\text{def}}{=} T_{\text{CCS}}(\emptyset)$; these are just the finite synchronisation trees.

The Restriction Deconstructor

Restriction

$$-\backslash a : ST \rightarrow ST$$

is the unique homomorphism

$$-\backslash a : ST \longrightarrow A$$

where A is the algebra with carrier ST and operations given by:

$$\begin{aligned} (b \cdot_A u) &= \begin{cases} \text{NIL} & (b = a) \\ b \cdot u & (b \neq a) \end{cases} \\ +_A(u, v) &= u + v \\ \text{NIL}_A &= \text{NIL} \end{aligned}$$

Note This evidently defines a CCS-algebra.

The Restriction Deconstructor (cntnd.)

More intuitively, one can simply define restriction by a kind of primitive recursion.

We have:

$$\begin{aligned}
 (b.u)\backslash a &= b.A(u\backslash a) &= \begin{cases} \text{NIL} & (b = a) \\ b.(u\backslash a) & (b \neq a) \end{cases} \\
 (u + v)\backslash a &= u\backslash a +_A v\backslash a &= u\backslash a + v\backslash a \\
 \text{NIL}\backslash a &= \text{NIL}_A &= \text{NIL}
 \end{aligned}$$

The Restriction Deconstructor (cntnd.)

So we can just define restriction by:

$$\begin{aligned}(b.u)\backslash a &= \begin{cases} \text{NIL} & (b = a) \\ b.(u\backslash a) & (b \neq a) \end{cases} \\ (u + v)\backslash a &= u\backslash a + v\backslash a \\ \text{NIL}\backslash a &= \text{NIL}\end{aligned}$$

But one needs also to verify that the implicit algebra on ST is a CCS-Algebra.

Remark: This restriction is not exactly that of CCS. It is an exercise to correct the definition.

The Renaming Deconstructor

This is defined recursively by:

$$(c.u)[a/b] = \begin{cases} a.(u[a/b]) & (c = b) \\ c.(u[a/b]) & (c \neq b) \end{cases}$$

$$(u + v)[a/b] = u[a/b] + v[a/b]$$

$$\text{NIL}[a/b] = \text{NIL}$$

(and, as before, a correction needs to be made to get CCS renaming).

Interleaving

Consider the interleaving function

$$| : ST \times ST \longrightarrow ST$$

Following the rather natural Dutch ACP approach, we write it as the sum of left interleaving and right interleaving operations:

$$u | v = u |^l v + u |^r v$$

where $|^l$ has first action that of its first argument and then becomes a regular interleaving, and $|^r$ rather favours its second argument.

Interleaving Defined

There is a natural "mutually recursive" definition:
The left and right operators satisfy the following defining equations:

$$\begin{aligned} \text{NIL} \mid^l z &= \text{NIL} \\ (x + y) \mid^l z &= (x \mid^l z) + (y \mid^l z) \\ a.x \mid^l z &= a.(x \mid^l z + x \mid^r z) \end{aligned}$$

and

$$\begin{aligned} z \mid^r \text{NIL} &= \text{NIL} \\ z \mid^r (x + y) &= (z \mid^r x) + (z \mid^r y) \\ z \mid^r a.x &= a.(z \mid^l x + z \mid^r x) \end{aligned}$$

But these equations do not fit with the homomorphic view (even accommodating it to allow parameters and mutually recursive definitions). The problem is the switch from **recursion variable** to **parameter**.

A homomorphic solution to the defining equations

Define

$$\bar{l} : \text{ST} \rightarrow \text{ST}^{\text{ST}} \quad \bar{r} : \text{ST} \times \text{ST}^{\text{ST}} \rightarrow \text{ST}$$

as follows:

$$\begin{aligned} \bar{l}(\text{NIL}) &= \lambda z : \text{ST}. \text{NIL} \\ \bar{l}(x + y) &= \lambda z : \text{ST}. \bar{l}(x)(z) + \bar{l}(y)(z) \\ \bar{l}(a.x) &= \lambda z : \text{ST}. a.(\bar{l}(x)(z) + \bar{r}(z, \bar{l}(x))) \end{aligned}$$

and

$$\begin{aligned} \bar{r}(\text{NIL}, f) &= \text{NIL} \\ \bar{r}(x + y, f) &= \bar{r}(x, f) + \bar{r}(y, f) \\ \bar{r}(a.x, f) &= a.(f(x) + \bar{r}(x, f)) \end{aligned}$$

and then set:

$$x \mid^l z = \bar{l}(x)(z) \quad x \mid^r z = \bar{r}(z, \bar{l}(x))$$

Why this is a solution

Left Shuffle

$$a.x \mid^l z = \bar{l}(a.x)(z) = a.(\bar{l}(x)(z) + \bar{r}(z, \bar{l}(x))) = a.(x \mid^l z + x \mid^r z)$$

Right Shuffle

$$x \mid^r a.z = \bar{r}(a.z, \bar{l}(x)) = a.(\bar{l}(x)(z) + \bar{r}(z, \bar{l}(x))) = a.(x \mid^l z + x \mid^r z)$$

(The idea was independently noted by Paul Levy.)

Dendriform dialgebras

- A **dendriform dialgebra** (Loday, 1993) is a \mathbf{k} -vector space $\langle A, + \rangle$ equipped with two binary operations, \triangleleft and \triangleright such that, for all $x, y, z \in A$:

$$\begin{aligned}(x \triangleleft y) \triangleleft z &= x \triangleleft (y \bowtie z) \\ (x \triangleright y) \triangleleft z &= x \triangleright (y \triangleleft z) \\ x \triangleright (y \triangleright z) &= (x \bowtie y) \triangleright z\end{aligned}$$

where

$$x \bowtie y =_{\text{def}} x \triangleleft y + x \triangleright y$$

- It is **commutative** (Shützenberger) if $x \triangleleft y = y \triangleright x$ always holds.
- Then \bowtie is an associative operation; it is commutative if the dialgebra is.

Concurrency with synchronisation

- Again following the ACP tradition, split $|$ into **three** parts:

$$x | y = x |^l y + x |^s y + x |^r y$$

where the central $|^s$ is for **synchronisation**.

- An **NS algebra** (Leroux, 2003) is a \mathbf{k} -vector space equipped with three bilinear operations, \triangleleft (left-linear), \triangleleft (right-linear), and \bullet such that

$$\begin{aligned} (x \triangleleft y) \triangleleft z &= x \triangleleft (y * z) \\ (x \triangleright y) \triangleleft z &= x \triangleright (y \triangleleft z) \\ x \triangleright (y \triangleright z) &= (x * y) \triangleright z \\ (x * y) \bullet z + (x \bullet y) \triangleleft z &= x \triangleright (y \bullet z) + x \bullet (y * z) \end{aligned}$$

where $x * y =_{\text{def}} x \triangleleft y + x \bullet y + x \triangleright y$

- It is **commutative** if \bullet is and $x \triangleleft y = y \triangleright x$ always holds.
- Then $*$ is an associative bilinear operation; it is commutative if the NS-algebra is.

Dendriform trialgebra, 1984

A dendriform trialgebra (Loday and Ronco, 2004) consists of a \mathbf{k} -vector space with three, binary operations $\triangleleft, \triangleright$, and \bullet s.t.:

$$\begin{aligned} (x \triangleleft y) \triangleleft z &= x \triangleleft (y * z) \\ (x \triangleright y) \triangleleft z &= x \triangleright (y \triangleleft z) \\ x \triangleright (y \triangleright z) &= (x * y) \triangleright z \end{aligned}$$

$$\begin{aligned} x \bullet (y \triangleleft z) &= (x \bullet y) \triangleleft z \\ (x \triangleright y) \bullet z &= x \triangleright (y \bullet z) \\ (x \triangleleft y) \bullet z &= x \bullet (y \triangleright z) \end{aligned}$$

$$(x \bullet y) \bullet z = x \bullet (y \bullet z)$$

where $* =_{\text{def}} \triangleleft + \bullet + \triangleright$. It is automatically an NS-algebra.

These equations appear already in Bergstra and Klop, 1984

Concurrency definition

- Synchronisation algebra
 $\langle A, \cdot \rangle$ a commutative partial monoid
- CCS Example
 $\langle \text{Act}, \cdot \rangle$ where:

$$a \cdot b = \begin{cases} \tau & (\bar{a} = b \neq \tau) \\ \uparrow & (\text{otherwise}) \end{cases}$$

Note: We use Roman a , etc, rather than Greek α for CCS actions.

Defining concurrency with synchronisation

- Define $|^l$, $|^r$, and $|^s$, together with $|^{sr} : A \times ST \times ST \longrightarrow ST$
- by:

$$a.x|^l z = a.(x|^l z + x|^s z + x|^r z), \text{ etc}$$

$$\text{NIL} |^s z = \text{NIL}$$

$$(x + y) |^s z = x|^s z + y|^s z$$

$$a.x|^s z = x|^{\text{sr}} a z$$

$$z|^r a.y = a.(z|^l y + z|^s y + z|^r y), \text{ etc}$$

- where:

$$z|^{\text{sr}} a \text{NIL} = \text{NIL}$$

$$z|^{\text{sr}} a(x + y) = z|^{\text{sr}} a x + z|^{\text{sr}} a y$$

$$z|^{\text{sr}} a b.y = \begin{cases} (a \cdot b).(z|^l y + z|^s y + z|^r y) & (\text{if } a \cdot b \downarrow) \\ \text{NIL} & (\text{otherwise}) \end{cases}$$

Homomorphic definitions

Define

$$\bar{l} : \text{ST} \rightarrow \text{ST}^{\text{ST}} \quad \bar{s} : \text{ST} \rightarrow \text{ST}^{\text{ST}} \quad \bar{s}\bar{r} : \text{ST} \rightarrow \text{ST}^{A \times \text{ST}^{\text{ST}}} \quad \bar{r} : \text{ST} \rightarrow \text{ST}^{\text{ST}^{\text{ST}}}$$

$$\text{by } \bar{l}(a.x) = \lambda z. a.(\bar{l}(x)(z) + \bar{s}(x)(z) + \bar{r}(z)(\bar{l}(x) + \bar{s}(x)))$$

$$\bar{s}(a.x) = \lambda z. \bar{s}\bar{r}_a(z)(\lambda v. \bar{l}(x)(v) + \bar{s}(x)(v) + \bar{r}(v)(\bar{l}(x) + \bar{s}(x)))$$

$$\bar{s}\bar{r}_a(b.y) = \lambda f. \begin{cases} (a \cdot b).(f(y)) & (\text{if } a \cdot b \downarrow) \\ \text{NIL} & (\text{otherwise}) \end{cases}$$

$$\bar{r}(a.y) = \lambda f. a.(f(y) + \bar{r}(y)(f))$$

then put

$$x \mid^l y = \bar{l}(x)(y) \quad x \mid^s y = \bar{s}(x)(y) \quad x \mid^r y = \bar{r}(y)(\bar{l}(x) + \bar{s}(x))$$

$$x \mid^{\text{sr}} a y = \bar{s}\bar{r}_a(y)(\lambda v : \text{ST}. x \mid^l v + x \mid^s v + x \mid^r v)$$

Prospects

- Can generalise the CCS deconstructors to all free algebras $T_{\text{CCS}}(X)$, eg:

$$| : T_{\text{CCS}}(X) \times T_{\text{CCS}}(Y) \longrightarrow T_{\text{CCS}}(X \times Y)$$

- To some extent can use other theories for CCS such as Milner's for $(+, \text{NIL}, \tau)$.
- **Prospect I**: a principled combination of process algebra and functional programming.
- Examples: CSP (with van Glabbeek); INRIA join calculus; pi-calculus (Stark).
- Questions: Operational semantics? Logic?
- **Prospect II**: integration of process calculus theory with the theory of effects.