Lecture 4: Effect Handlers

Gordon Plotkin

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

NII Shonan Meeting No. 146
Programming and Reasoning with Algebraic Effects and Effect Handlers

Outline

Outline

Simple exception handler

- The construct is:

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

  assuming a finite set of exceptions $E =_{\mathrm{def}} \mathcal{M}[\![\mathrm{exc}]\!]$
- 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 $\mathrm{handle} : \varepsilon; 1, \mathrm{exc}$ say. Then we will not have:

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

Failure to be algebraic

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

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

and:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\models \mathrm{handle}(\mathcal{E}[M], x : \mathrm{exc}.\mathcal{E}[H(x)])} \\
= & \mathrm{handle}((\lambda y : \mathrm{nat}.\, \mathrm{raise}_{e_1})3, x : \mathrm{exc}.(\lambda y : \mathrm{nat}.\, \mathrm{raise}_{e_1})H(x)) \\
= & \mathrm{handle}(\mathrm{raise}_{e_1}, x : \mathrm{exc}.(\lambda y : \mathrm{nat}.\, \mathrm{raise}_{e_1})H(x)) \\
= & (\lambda y : \mathrm{nat}.\, \mathrm{raise}_{e_1})H(e_1) \\
=_{\mathrm{def}} & (\lambda y : \mathrm{nat}.\, \mathrm{raise}_{e_1})\mathrm{raise}_{e_2} \\
= & \mathrm{raise}_{e_2}
\end{array}
$$

and the two are different.

Understanding the Benton and Kennedy exception handler algebraically

Simple exception handler

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

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

Benton and Kennedy exception handler

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

Analysis of the semantics of the BK exception handler

- $M \in T_{\mathrm{Ax}}(\sigma) = \sigma + E$.
- $\{\mathrm{raise}_e \mapsto H_e^{A}\}_{e \in E}$ 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)

$$(\lambda x : \sigma.\, N(x))^{\dagger}(M)$$

The general algebraic situation

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

$$\begin{tikzcd} X \arrow[d, "\eta"'] \arrow[dr, "f"] & \\ T_{\mathrm{Ax}}(X) \arrow[r, "f^\dagger"] & A \end{tikzcd}$$

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

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

λ-calculus additions: Handlers for simple operations

- Handlers

$$H ::= \{\mathrm{op}(k_1 : T\tau, \ldots, k_n : T\tau) = H_{\mathrm{op}}\}_{\mathrm{op}:n}$$

where $T(\tau) =_{\mathrm{def}} \mathrm{unit} \to \tau$

$$\frac{\Gamma, k_1 : T\tau, \ldots, k_n : T\tau \vdash H_{\mathrm{op}} : \tau \quad (i = 1, n)}{\Gamma \vdash \{\mathrm{op}(k_1 : T\tau, \ldots, k_n : T\tau) = H_{\mathrm{op}}\}_{\mathrm{op}:n} : \tau\ \textbf{handler}}$$

- Handling

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

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

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

$$H ::= \{\mathrm{op}(k_1 : T\tau, \ldots, k_n : T\tau) = H_{\mathrm{op}}(k_1, \ldots, k_n)\}_{\mathrm{op}:n}$$

then

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

and

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

where

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

(where, for any term L, we set the thunk $[L] = \lambda x : \mathrm{unit}.\ L$)

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:

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

- 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:

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

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

How this handler works

The handler H_{ro} is:

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

It works as follows:

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

$$\begin{array}{l} = \mathrm{if}\ b\ \mathrm{then}\ [M_1\ \mathrm{handled\,with}\ H_{\mathrm{ro}}\ \mathrm{to}\ x : \sigma\ \mathrm{in}\ N](*) \\ \qquad \mathrm{else}\ [M_2\ \mathrm{handled\,with}\ H_{\mathrm{ro}}\ \mathrm{to}\ x : \sigma\ \mathrm{in}\ N](*) \end{array}$$

$$\begin{array}{l} = \mathrm{if}\ b\ \mathrm{then}\ M_1\ \mathrm{handled\,with}\ H_{\mathrm{ro}}\ \mathrm{to}\ x : \sigma\ \mathrm{in}\ N \\ \qquad \mathrm{else}\ M_2\ \mathrm{handled\,with}\ H_{\mathrm{ro}}\ \mathrm{to}\ x : \sigma\ \mathrm{in}\ N \end{array}$$

Additions to the λ-calculus: General operations

Handlers

$$H ::= \{\mathrm{op}_{\mathbf{x}:\mathbf{s}}(k_1 : \mathbf{s}_1 \to \tau, \ldots, k_n : \mathbf{s}_n \to \tau) = H_{\mathrm{op}}\}_{\mathrm{op}:\mathbf{s};\mathbf{s}_1,\ldots,\mathbf{s}_m}$$

$$\frac{\Gamma, \mathbf{x} : \mathbf{s}, k_1 : \mathbf{s}_1 \to \tau, \ldots, k_n : \mathbf{s}_n \to \tau \vdash H_{\mathrm{op}} : \tau \quad (i = 1, n)}{\Gamma \vdash \{\mathrm{op}_{\mathbf{x}:\mathbf{s}}(k_1 : \mathbf{s}_1 \to \tau, \ldots, k_n : \mathbf{s}_n \to \tau) = H_{\mathrm{op}}\}_{\mathrm{op}:\mathbf{s};\mathbf{s}_1,\ldots,\mathbf{s}_m} : \tau\ \textbf{handler}}$$

How these handlers work

Suppose H is

$$\{\mathrm{op}_{\mathbf{x}:\mathbf{s}}(k_1 : \mathbf{s}_1 \to \tau, \ldots, k_n : \mathbf{s}_n \to \tau) = H_{\mathrm{op}}(\mathbf{x}, k_1, \ldots, k_n)\}_{\mathrm{op}:\mathbf{s};\mathbf{s}_1,\ldots,\mathbf{s}_m}$$

then

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

where

$$K_i = \lambda \mathbf{x} : \mathbf{s}.\, M_i \text{ handled with } H \text{ to } x : \sigma \text{ in } N$$

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:

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

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

  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:

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

  where $e : \mathrm{exc} \vdash \mathrm{Roll} : \tau$.

- 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 the rollback computation Roll with the exception raised, we use:

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

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.

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

- Handlers

$$H ::= \{\mathrm{op}(k_1 : \pi \to \tau, \ldots, k_n, : \pi \to \tau) \mathbin{@} p : \pi = H_{\mathrm{op}}\}_{\mathrm{op}:n}$$

$$\frac{\Gamma, k_1 : \pi \to \tau, \ldots, k_n : \pi \to \tau, p : \pi \vdash H_{\mathrm{op}} : \tau \quad (\mathrm{op} : n)}{\Gamma \vdash \{\mathrm{op}(k_1 : \pi \to \tau, \ldots, k_n : \pi \to \tau) \mathbin{@} p : \pi = H_{\mathrm{op}}\}_{\mathrm{op}:n} : \pi \to \tau \ \textbf{handler}}$$

- Handling

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

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

How parameterised simple handlers work

Suppose H is

$$\{\mathrm{op}(k_1 : \pi \to \tau, \ldots, k_n, : \pi \to \tau) \mathbin{@} p : \pi = H_{\mathrm{op}}(p, k_1, \ldots, k_n)\}_{\mathrm{op}:n}$$

then

$$\begin{array}{l} \vdash \mathrm{op}(M_1, \ldots, M_n) \text{ handled with } H@P \text{ to } x : \sigma \text{ in } N \\ \quad = \text{let } p : \pi \text{ be } P \text{ in } H_{\mathrm{op}}(p, K_1, \ldots, K_n) \end{array}$$

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):

$$\{\mathrm{lookup}(k_1 : \mathrm{bool} \to \tau, k_2 : \mathrm{bool} \to \tau)\,@\,b : \mathrm{bool} \\ = \mathrm{if}\ b\ \mathrm{then}\ k_1(b)\ \mathrm{else}\ k_2(b)\} : \mathrm{bool} \to \tau\ \textbf{handler}$$

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

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

- Update So could define

$$\mathrm{update}_P(M) =_{\mathrm{def}} M\ \mathrm{handled\ with}\ H_{\mathrm{ro}}@P\ \mathrm{to}\ x : \sigma\ \mathrm{in}\ x$$

How this handler works

The handler H_{ro} is:

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

It works as follows:

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

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

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

Faking output and curtailing input

The handler H is

$$
\begin{array}{l}
\{\mathrm{input}(k : \mathrm{nat} \to (\mathrm{in} \to \tau))@\mathrm{limit} : \mathrm{nat} \\
\quad = \mathrm{if\ limit} > 0 \mathrm{\ then\ input}(y : \mathrm{in}.\, k(\mathrm{limit} - 1)(y)) \\
\qquad\qquad\qquad\quad \mathrm{else\ raise}_{\mathrm{input\_session\_finished}}(), \\
\ \mathrm{output}_{z:\mathrm{out}}(k : \mathrm{nat} \to \tau)@\mathrm{limit} : \mathrm{nat} \\
\quad = \mathrm{output}_{\mathrm{fake}}(k(\mathrm{limit}))\}
\end{array}
$$

It works as follows:

$$
\begin{array}{l}
\vdash \mathrm{input}(y : \mathrm{in}.\, M) \text{ handled with } H@\mathrm{limit} \text{ to } x : \sigma \text{ in } N \\
\quad = \mathrm{if\ limit} > 0 \\
\qquad \mathrm{then\ input}(y : \mathrm{in}.\, M \text{ handled with } H@(\mathrm{limit} - 1) \text{ to } x : \sigma \text{ in } N) \\
\qquad \mathrm{else\ raise}_{\mathrm{input\_session\_finished}}()
\end{array}
$$

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

A possible treatment of handlers for effects and types

- Effects

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

- Handling

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

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

- Handlers

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

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

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

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. Bauer and Pretnar's *Eff* language is based on this idea.

Outline

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*:

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

Nondeterminism as an algebraic effect

There is a natural equational theory, with signature $+ : 2 \to 1$, $\mathrm{NIL} : 0 \to 1$ and axioms:

$$\begin{array}{lrcl}
\textit{Associativity} & (x+y)+z & = & x+(y+z) \\
\textit{Commutativity} & x+y & = & y+x \\
\textit{Absorption} & x+x & = & x \\
\textit{Zero} & \mathit{NIL}+x & = & x
\end{array}$$

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

$$
\begin{array}{lll}
P & ::= & a.P\ (a \in \mathrm{Act}) \mid P + Q \mid \mathrm{NIL} \\
 & & \mid P \backslash b \mid P[b/c] \\
 & & \\
 & & \mid P|Q
\end{array}
$$

Effect Constructors
Unary Effect Deconstructors handling P
Binary Effect Deconstructors handling P and Q

Equational theory for the constructors

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

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

Modelling CCS

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

The Restriction Deconstructor

Restriction

$$-\backslash b : \mathrm{ST} \to \mathrm{ST}$$

is the unique homomorphism

$$-\backslash b : \mathrm{ST} \longrightarrow \mathcal{R}$$

where $\mathcal{R}$ is the algebra with carrier ST and operations given by:

$$\begin{array}{lcl} (a._{\mathcal{R}} u) & = & \left\{ \begin{array}{ll} \mathrm{NIL} & (a = b) \\ a.u & (a \neq b) \end{array} \right. \\ +_{\mathcal{R}}(u, v) & = & u + v \\ \mathrm{NIL}_{\mathcal{R}} & = & \mathrm{NIL} \end{array}$$

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:

$$
\begin{array}{lclcl}
(a.u)\backslash b & = & a._{\mathcal{R}}(u\backslash b) & = & \left\{\begin{array}{ll} \mathrm{NIL} & (a = b) \\ a.(u\backslash b) & (a \neq b) \end{array}\right. \\
(u+v)\backslash b & = & u\backslash b +_{\mathcal{R}} v\backslash b & = & u\backslash b + v\backslash b \\
\mathrm{NIL}\backslash b & = & \mathrm{NIL}_{\mathcal{R}} & = & \mathrm{NIL}
\end{array}
$$

The Restriction Deconstructor (cntnd.)

So we can just define restriction by:

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

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:

$$
\begin{array}{lcl}
(a.u)[b/c] & = & \left\{ \begin{array}{ll} b.(u[b/c]) & (a = c) \\ a.(u[b/c]) & (a \neq c) \end{array} \right. \\[2ex]
(u + v)[b/c] & = & u[b/c] + v[b/c] \\[2ex]
\mathrm{NIL}[b/c] & = & \mathrm{NIL}
\end{array}
$$

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

Interleaving

Consider the interleaving function

$$| \, : \mathrm{ST} \times \mathrm{ST} \longrightarrow \mathrm{ST}$$

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

$$u \,|\, v = u \,|^{\mathrm{l}}\, v + u \,|^{\mathrm{r}}\, v$$

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

Interleaving Defined

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

$$\begin{array}{rcl} \mathrm{NIL} \,|^{\mathrm{l}}\, z & = & \mathrm{NIL} \\ (x+y) \,|^{\mathrm{l}}\, z & = & (x \,|^{\mathrm{l}}\, z) + (y \,|^{\mathrm{l}}\, z) \\ a.x \,|^{\mathrm{l}}\, z & = & a.({\color{blue}x} \,|^{\mathrm{l}}\, z + {\color{red}x} \,|^{\mathrm{r}}\, z) \end{array}$$

and

$$\begin{array}{rcl} z \,|^{\mathrm{r}}\, \mathrm{NIL} & = & \mathrm{NIL} \\ z \,|^{\mathrm{r}}\, (x+y) & = & (z \,|^{\mathrm{r}}\, x) + (z \,|^{\mathrm{r}}\, y) \\ z \,|^{\mathrm{r}}\, a.x & = & a.(z \,|^{\mathrm{l}}\, {\color{red}x} + z \,|^{\mathrm{r}}\, {\color{blue}x}) \end{array}$$

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

$$\bar{l} : \mathrm{ST} \to \mathrm{ST}^{\mathrm{ST}} \qquad\qquad \bar{r} : \mathrm{ST} \times \mathrm{ST}^{\mathrm{ST}} \to \mathrm{ST}$$

as follows:

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

and

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

and then set:

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

Why this is a solution

Left Shuffle

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

Right Shuffle

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

(The idea was independently noted by Paul Levy.)

Dendriform dialgebras

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

$$\begin{array}{rcl} (x \lhd y) \lhd z & = & x \lhd (y \bowtie z) \\ (x \rhd y) \lhd z & = & x \rhd (y \lhd z) \\ x \rhd (y \rhd z) & = & (x \bowtie y) \rhd z \end{array}$$

  where

$$x \bowtie y =_{\mathrm{def}} x \lhd y + x \rhd y$$

- It is commutative (Shützenberger) if $x \lhd y = y \rhd x$ 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:

$$x \mid y = x \mid^{\mathrm{l}} y \; + \; x \mid^{\mathrm{s}} y \; + \; x \mid^{\mathrm{r}} y$$

  where the central $|^{\mathrm{s}}$ is for synchronisation.
- An NS algebra (Leroux, 2003) is a **k**-vector space equipped with three operations $\lhd$, $\rhd$, and $\bullet$ (respectively left linear, right linear, and bilinear) such that

$$\begin{array}{rcl}
(x \lhd y) \lhd z & = & x \lhd (y * z) \\
(x \rhd y) \lhd z & = & x \rhd (y \lhd z) \\
x \rhd (y \rhd z) & = & (x * y) \rhd z \\
(x * y) \bullet z + (x \bullet y) \lhd z & = & x \rhd (y \bullet z) + x \bullet (y * z)
\end{array}$$

  where $x * y =_{\mathrm{def}} x \lhd y + x \bullet y + x \rhd y$
- It is commutative if $\bullet$ is and $x \lhd y = y \rhd x$ 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 **k**-vector space, with three binary operations $\lhd$, $\rhd$, and $\bullet$ (with linearity as before) s.t.:

$$
\begin{array}{rcl}
(x \lhd y) \lhd z & = & x \lhd (y * z) \\
(x \rhd y) \lhd z & = & x \rhd (y \lhd z) \\
x \rhd (y \rhd z) & = & (x * y) \rhd z \\
\\
x \bullet (y \lhd z) & = & (x \bullet y) \lhd z \\
(x \rhd y) \bullet z & = & x \rhd (y \bullet z) \\
(x \lhd y) \bullet z & = & x \bullet (y \rhd z) \\
\\
(x \bullet y) \bullet z & = & x \bullet (y \bullet z)
\end{array}
$$

where $* =_{\mathrm{def}} \lhd + \bullet + \rhd$. 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 semigroup
- CCS Example
  $\langle \mathrm{Act}, \cdot \rangle$ where:

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

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

Defining concurrency with synchronisation

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

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

$$\begin{array}{rcl} \mathrm{NIL} \,|^{\mathrm{s}}\, z & = & \mathrm{NIL} \\ (x+y) \,|^{\mathrm{s}}\, z & = & x \,|^{\mathrm{s}}\, z \;+\; y \,|^{\mathrm{s}}\, z \\ a.x \,|^{\mathrm{s}}\, z & = & x \,|^{\mathrm{sr}}{}_{a} z \end{array}$$

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

- where:

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

Homomorphic definitions

Define

$$\bar{l} : \mathrm{ST} \to \mathrm{ST}^{\mathrm{ST}} \quad \bar{\mathrm{s}} : \mathrm{ST} \to \mathrm{ST}^{\mathrm{ST}} \quad \overline{\mathrm{sr}} : \mathrm{ST} \to \mathrm{ST}^{A \times \mathrm{ST}^{\mathrm{ST}}} \quad \bar{r} : \mathrm{ST} \to \mathrm{ST}^{\mathrm{ST}^{\mathrm{ST}}}$$

by

$$\begin{array}{rcl} \bar{l}(a.x) & = & \lambda z.\, a.(\bar{l}(x)(z) + \bar{\mathrm{s}}(x)(z) + \bar{r}(z)(\bar{l}(x) + \bar{\mathrm{s}}(x))) \\ \bar{\mathrm{s}}(a.x) & = & \lambda z.\, \overline{\mathrm{sr}}_a(z)(\lambda v.\, \bar{l}(x)(v) + \bar{\mathrm{s}}(x)(v) + \bar{r}(v)(\bar{l}(x) + \bar{\mathrm{s}}(x))) \\ \overline{\mathrm{sr}}_a(b.y) & = & \lambda f. \begin{cases} (a \cdot b).(f(y)) & (\text{if } a \cdot b \downarrow) \\ \mathrm{NIL} & (\text{otherwise}) \end{cases} \\ \bar{r}(a.y) & = & \lambda f.\, a.(f(y) + \bar{r}(y)(f)) \end{array}$$

then put

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

$$x \,|^{\mathrm{sr}}{}_a y = \overline{\mathrm{sr}}_a(y)(\lambda v : \mathrm{ST}.\, x \,|^{\mathrm{l}}\, v + x \,|^{\mathrm{s}}\, v + x \,|^{\mathrm{r}}\, v)$$

Prospects

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

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

- To some extent can use other theories for CCS such as Milner's for $(+, \mathrm{NIL}, \tau)$.
- 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.