A Model of Cooperative Threads

Martín Abadi  Gordon Plotkin

Microsoft Research, Silicon Valley
LFCS, University of Edinburgh

Logic Seminar, Stanford, 2008
Introduction

A Language for Cooperative Threads

An Elementary Fully Abstract Denotational Semantics

An Algebraic View of the Semantics

Conclusions
Cooperative Threads and AME

- Cooperative Threads run without interruption until they yield control.
- Interest in such threads has increased recently with the introduction of Automatic Mutual Exclusion (AME) and the problem of programming multicore systems.
We describe a simple language for cooperative threads and give it a mathematically elementary fully abstract (may) semantics of sets of traces, being *transition sequences* of, roughly, the form:

\[ u = (\sigma_1, \sigma'_1) \ldots (\sigma_m, \sigma'_m) \]

à la Abrahamson, the authors, Brookes etc, but adapted to incorporate thread spawning.

Following the algebraic theory of effects, we characterise the semantics using a suitable inequational theory, thereby relating it to standard domain-theoretic notions of resumptions.
Syntax

\[ b \in B\text{Exp} = \ldots \]
\[ e \in N\text{Exp} = \ldots \]
\[ C, D \in \text{Com} = \text{skip} \]
\[ | \quad x := e \quad (x \in \text{Vars}) \]
\[ | \quad C; D \]
\[ | \quad \text{if } b \text{ then } C \text{ else } D \]
\[ | \quad \text{while } b \text{ do } C \]
\[ | \quad \text{async } C \]
\[ | \quad \text{yield} \]
\[ | \quad \text{block} \]
Example

```
async x := 0;
x := 1;
yield;
if x = 0 then x := 2 else block
```

This spawns the asynchronous execution of \( x := 0 \), and then executes \( x := 1 \) and yields. When it resumes it blocks unless the predicate \( x = 0 \) holds, when it executes \( x := 2 \).

With respect to safety properties, the conditional blocking amounts to awaiting that \( x = 0 \) holds. So the last line may be paraphrased as

```
await x = 0; x := 2
```
Operational Semantics

\[ \langle \sigma, T, \mathcal{E}[x := e] \rangle \]
\[ \langle \sigma, T, \mathcal{E}[\text{skip}; C] \rangle \]
\[ \langle \sigma, T, \mathcal{E}[(\text{if } b \text{ then } C \text{ else } D)] \rangle \]
\[ \langle \sigma, T, \mathcal{E}[\text{while } b \text{ do } C] \rangle \]
\[ \langle \sigma, T, \mathcal{E}[(\text{async } C)] \rangle \]
\[ \langle \sigma, T, \mathcal{E}[(\text{yield})] \rangle \]
\[ \langle \sigma, T.C.T', \text{skip} \rangle \]

\[ \rightarrow_a \quad \langle \sigma[x \mapsto \sigma(e)], T, \mathcal{E}[\text{skip}] \rangle \]
\[ \rightarrow_a \quad \langle \sigma, T, \mathcal{E}[C] \rangle \]
\[ \rightarrow_a \quad \langle \sigma, T, \mathcal{E}[C] \rangle \text{ (if } \sigma(b) = \text{true}) \]
\[ \rightarrow_a \quad \langle \sigma, T, \mathcal{E}[C; \text{while } b \text{ do } C] \rangle \text{ (if } \sigma(b) = \text{true}) \]
\[ \rightarrow_a \quad \langle \sigma, T.C, \mathcal{E}[\text{skip}] \rangle \]
\[ \rightarrow_a \quad \langle \sigma, T.\mathcal{E}[\text{skip}], \text{skip} \rangle \]
\[ \rightarrow_c \quad \langle \sigma, T.T', C \rangle \]
State Space

\[
\begin{align*}
\Gamma & \in \text{State} = \text{Store} \times \text{ThreadPool} \times \text{Com} \\
\sigma & \in \text{Store} = \text{Vars} \rightarrow \text{Val} \\
T & \in \text{ThreadPool} = \text{Com}^* 
\end{align*}
\]

where Vars is finite, and Val is countable.

Evaluation Contexts

\[
\mathcal{E} = [\ ] | \mathcal{E}; C
\]
Introduction

A Language for Cooperative Threads

An Elementary Fully Abstract Denotational Semantics

An Algebraic View of the Semantics

Conclusions

1. Introduction
2. A Language for Cooperative Threads
3. An Elementary Fully Abstract Denotational Semantics
   - Denotational Semantics
   - Adequacy and Full Abstraction
4. An Algebraic View of the Semantics
   - The Algebraic Theory of Effects
   - Resumptions Considered Algebraically
   - Asynchronous Processes Considered Algebraically
   - Processes Considered Algebraically
5. Conclusions
Transition Sequences

- Abrahamson used transition sequences of the form:
  \[ u = (\sigma_1, \sigma'_1) \ldots (\sigma_m, \sigma'_m) \]
- Perhaps we need hierarchical triples for thread spawning:
  \[ v = (\sigma_1, u_1, \sigma'_1) \ldots (\sigma_m, u_m, \sigma'_m) \]
- Miraculously, we only need 1 embedding to 1 level, roughly:
  \[ v = (\sigma_1, \sigma'_1) \ldots (\sigma_m, u, \sigma'_m) \]
- Precisely, so that prefix is the right partial order, and also to allow for totality, transition sequences are:
  \[ v = (\sigma_1, \sigma'_1) \ldots (\sigma_m, \sigma'_m)[(\sigma, \sigma' \text{ return})u] \]

where \[ u = (\bar{\sigma}_1, \bar{\sigma}'_1) \ldots (\bar{\sigma}_n, \bar{\sigma}'_n)[\text{done}] \] is a pure transition sequence (and \( m, n \geq 0 \)).
Proc, our domain of processes, is $\mathcal{I}_{\neq\emptyset,\omega}(TSeq)$ the $\omega$-cpo of all non-empty, countably-based ideals of transition sequences, i.e., all nonempty prefix-closed sets of transition sequences. We have:

$$\llbracket C \rrbracket \in \text{Proc}$$

Pool, our domain of thread pools, is $\mathcal{I}_{\neq\emptyset,\omega}(PSeq)$ the $\omega$-cpo of all non-empty, countably-based ideals of pure transition sequences, i.e., the $\omega$-cpo of all non-empty prefix-closed sets of pure transition sequences. We have:

$$\llbracket T \rrbracket \in \text{Pool}$$
Denotational Semantics of Commands

\[
\begin{align*}
\sem{\text{skip}} &= * \\
\sem{C; D} &= \sem{C} \circ \sem{D} \\
\sem{x := e} &= \{(\sigma, \sigma[x \mapsto v] \text{ return}) \text{ done} \mid \sigma \in \text{Store}, \sigma(e) = v\} \downarrow \\
\sem{\text{if } b \text{ then } C \text{ else } D} &= \{(\sigma, \tau)v \in \sem{C} \mid \sigma(b) = \text{true}\} \downarrow \cup \{(\sigma, \tau)v \in \sem{D} \mid \sigma(b) = \text{false}\} \downarrow \\
\sem{\text{while } b \text{ do } C} &= \bigcup_i \sem{(\text{while } b \text{ do } C)_i} \\
\sem{\text{async } C} &= \text{async}(\sem{C}^\circ) \\
\sem{\text{yield}} &= d(*) \\
\sem{\text{block}} &= \{\varepsilon\}
\end{align*}
\]
Sequential Composition of Processes

We give rules for composition, as it is easier to understand that way:

\[
\begin{align*}
\nu(\sigma, \sigma' \text{return})u & \in P & (\sigma', \tau)w & \in Q \\
\nu(\sigma, \tau)(u \triangleright w) & \subseteq P \circ Q \\
\nu & \in P \\
\nu & \in P \circ Q \\
\nu & \in P \circ Q
\end{align*}
\]

It is associative with two-sided unit:

\[
* = \{(\sigma, \sigma \text{ return}) \text{ done} \mid \sigma \in \text{Store}\} \downarrow
\]
The set of merges of a pure transition sequence $u$ and a (pure) transition sequence $w$ is given by:

$$u[\text{done}]^1 \triangleright w[\text{done}]^2 = (u \triangleright w)[\text{done}]^{1^\land 2}$$

where:
- the merge on the right is the standard merge of sequences
- the done on the right appears only if it appears both times on the left.
We define a continuous *delay* function $d : \text{Proc} \rightarrow \text{Proc}$ by:

$$d(P) = \{(\sigma, \sigma)v \mid \sigma \in \text{Store}, v \in P\} \downarrow$$

So that:

$$[\text{yield}] = d(*) = \{(\sigma, \sigma)(\sigma', \sigma' \text{ return})\text{done}\} \downarrow$$
Spawning Threads

Recall:

$$[[\text{async } C]] = \text{async}([[C]]^c)$$

Where:

- $$-^c : \text{Proc} \rightarrow \text{Pool}$$ is the extension to processes of the function $$-^c : \text{TSeq} \rightarrow \text{PSeq}$$ of the same name from transition sequences to pure transition sequences which removes the marker $$\text{return}$$

$$\text{async}(P) = \{ (\sigma, \sigma \text{ return})u \mid \sigma \in \text{Store}, u \in P \}$$

Note: $$\text{async}(P^c)$$ differs from $$d(P)$$ only in the placement of the marker $$\text{return}$$: the former replaces it at the beginning.
Denotational Semantics of Thread Pools

- $\bowtie$: Pool$^2 \rightarrow$ Pool is the extension to thread pools of the binary function on pure transition sequences of the same name.
- Together with $I \overset{\text{def}}{=} \{\text{done}\} \downarrow$ it forms a commutative monoid.
- The semantics of a thread pool $C_1, \ldots, C_n$ is given by:

$$[[C_1, \ldots, C_n]] = [[C_1]]^c \bowtie \ldots \bowtie [[C_n]]^c \quad (n \geq 0)$$

- Note that $[[\varepsilon]] = I$.
- Our domain of asynchronous processes $\text{AProc}$ is the sub-$\omega$-cpo of Pool none of whose elements contain $\text{done}$.
- We always have $[[C]]^c \in \text{AProc}$.
Outline

1. Introduction
2. A Language for Cooperative Threads
3. An Elementary Fully Abstract Denotational Semantics
   - Denotational Semantics
   - Adequacy and Full Abstraction
4. An Algebraic View of the Semantics
   - The Algebraic Theory of Effects
   - Resumptions Considered Algebraically
   - Asynchronous Processes Considered Algebraically
   - Processes Considered Algebraically
5. Conclusions
Adequacy Theorem for Pure Transition Sequences

Define:

\[ \Gamma \Rightarrow \Gamma' \text{ iff } \Gamma \xrightarrow{a^*} c \Gamma' \]

and

\[ \llbracket T, C \rrbracket = \text{async}(\llbracket T \rrbracket) \circ \llbracket C \rrbracket \]

Theorem

The following are equivalent:

1. \((\sigma_1, \sigma'_1) \ldots (\sigma_n, \sigma'_n)\) done \( \in \llbracket T_1, C_1 \rrbracket^c \) (\( n > 0 \))

2. There are \( T_i, C_i \), \( (i = 2, n) \) such that
   
   \[ \langle \sigma_i, T_i, C_i \rangle \Rightarrow \langle \sigma'_i, T_{i+1}, C_{i+1} \rangle, \text{ for } 1 \leq i \leq n - 1 \]
   
   \[ \langle \sigma_n, T_n, C_n \rangle \xrightarrow{a^*} \langle \sigma'_n, \varepsilon, \text{skip} \rangle. \]

There is an analogous statement for \((\sigma_1, \sigma'_1) \ldots (\sigma_n, \sigma'_n) \in \llbracket T, C \rrbracket^c\)
To account for uninterrupted running, we define, for \( P \in \text{Pool} \):

\[
\text{runs}(P) = \{ \sigma_1 \ldots \sigma_n[\text{done}] \mid (\sigma_1, \sigma_2)(\sigma_2, \sigma_3)\ldots(\sigma_{n-1}, \sigma_n)[\text{done}] \in P \}
\]

*These runs are our observables.*

**Corollary**

The following are equivalent:

1. \( \sigma_1 \ldots \sigma_n[\text{done}] \in \text{runs}([ [T_1, C_1] ]) \quad (n \geq 2) \)

2. There are \( T_i, C_i \), \( (i = 2, n - 1) \) such that:
   \[
   \langle \sigma_1, T_1, C_1 \rangle \Rightarrow \ldots \Rightarrow \langle \sigma_{n-1}, T_{n-1}, C_{n-1} \rangle \rightarrow_a^* \langle \sigma_n, \varepsilon, \text{skip} \rangle
   \]

There is an analogous statement for \( \sigma_1 \ldots \sigma_n \in \text{runs}([ [T_1, C_1] ]) \)
Inequational Full Abstraction

**Theorem**

The following are equivalent, for any commands $C$ and $D$:

1. $[C] \subseteq [D]$

2. For every context $C$, $\text{runs}([C[C]]^c) \subseteq \text{runs}([C[D]]^c)$. 
Following Moggi we are interested in a monadic point of view, here using a continuous monad $T(P)$ over $\omega Cpo$ to model the set of computations for elements of $P$. We seek such a $T_{Proc}$ with:

$$\text{Proc} = T_{Proc}(1)$$

To this end we seek a computationally interesting, and mathematically natural, equational theory $L_{Proc}$ such that $T_{Proc}$ is the corresponding free algebra (better, free model) monad.

This theory will be a variant of the theory for the classical resumptions monad and so we will also see how the trace model described above fits in with standard notions.
Introduction

A Language for Cooperative Threads

An Elementary Fully Abstract Denotational Semantics

An Algebraic View of the Semantics

Conclusions

The Algebraic Theory of Effects

Resumptions Considered Algebraically

Asynchronous Processes Considered Algebraically

Processes Considered Algebraically

Outline

1. Introduction
2. A Language for Cooperative Threads
3. An Elementary Fully Abstract Denotational Semantics
   - Denotational Semantics
   - Adequacy and Full Abstraction
4. An Algebraic View of the Semantics
   - The Algebraic Theory of Effects
   - Resumptions Considered Algebraically
   - Asynchronous Processes Considered Algebraically
   - Processes Considered Algebraically
5. Conclusions
Inequational Theories

These are:

\[ Th = (\Sigma, \text{InEq}) \]

where the operation arities \( f : n \to 1 \) are given by \( \Sigma \) and \( \text{InEq} \) is a set of inequations

\[ t \leq u \]

over terms formed from these operation symbols.

One then has the usual notion of \( \Sigma \)-algebra in \( \omega \text{Cpo} \) and the free model—meaning modelling the inequations—monad over \( \omega \text{Cpo} \) is written \( T_{Th} \).
Two Examples

**Example** Nontermination: the theory $L_\Omega$

\[ \Omega \leq x \]

Here $T_\Omega$ is the usual lifting monad.

**Example** Hoare (Lower) Powerdomain: the theory $L_H$

\[
x \leq x \cup y \quad y \leq x \cup y \quad z \cup z \leq z
\]

Here $T_H$ is the lower powerdomain monad in $\omega\text{Cpo}$; $T_H(P)$ is the free $\omega$-semilattice over $P$ (meaning all countable sups) and it consists of of all countably generated Scott closed sets.
The Side-Effects Monad Considered Algebraically

Monad

\[ T_S(P) = (\text{Store} \times P)^\text{Store} \]

Signature

\[ \text{lookup} : \text{Val} \rightarrow \text{Vars} \quad \text{update} : 1 \rightarrow \text{Vars} \times \text{Val} \]

Equivalently, for all \( l \in \text{Vars} \) and \( v \in \text{Val} \),

\[ \text{lookup}_l : \text{Val} \quad \text{update}_{l,v} : 1 \]

The corresponding generics are the functions:

\[ ! : \text{Vars} \rightarrow T_S(\text{Val}) \quad := : \text{Vars} \times \text{Val} \rightarrow T_S(1) \]
Sample Equations for the Side-Effects Theory $SE$

\[
\text{update}_{l,v}(\text{update}_{l',v'}(x)) = \text{update}_{l',v'}(\text{update}_{l,v}(x)) \quad (\text{if } l \neq l')
\]

\[
\text{lookup}_l(\ldots \text{update}_{l,v}(x) \ldots) = x
\]

which last can be written in a finitary way as:

\[
\text{lookup}_l((v : \text{val}).\text{update}_{l,v}(x)) = x
\]
Countably Infinitary Continuous Algebra

- **Signature** \( \Sigma = \{ f : \vec{l}_1, \ldots, \vec{l}_m \rightarrow O_1, \ldots, O_n \} \), with \( \vec{l}_1, \ldots, \vec{l}_m \) countably infinite sets, and \( O_1, \ldots, O_n \) parameter spaces, being \( \omega \)-cpos, giving:
  - Function symbols \( f_{o_1, \ldots, o_n} \) (for \( o_j \in O_j \)), indexed by:
    \[
    O = \text{def } O_1 \times \ldots \times O_n
    \]
    of arity
    \[
    I = \text{def } (\prod \vec{l}_1) \times \ldots \times (\prod \vec{l}_m)
    \]
  - Infinitary terms
    \[
    f_{\vec{o}}(\langle t_{\vec{i}_1}, \ldots, t_{\vec{i}_m} \rangle_{\vec{i}_1, \ldots, \vec{i}_m})
    \]
    Note the indexed arguments.

- **Inequations** \( \text{InEq} \) consists of inequations \( t \leq u \) between the (possibly) infinitary terms formed from the function symbols.
Models

- **Algebras** Carriers, being $\omega$-cpos $A$, equipped with continuous maps

  \[ f_A : A^I \rightarrow A^O \]

  equivalently

  \[ f_A : O \times A^I \rightarrow A \]

  Models are such satisfying the inequations.

- **Free Algebra Monad** We obtain $T_{Th}$ giving the free such model; it is an $\omega$Cpo-monad.

  **Remark** There is a useful finitary notation for such infinitary theories.
Outline

1. Introduction
2. A Language for Cooperative Threads
3. An Elementary Fully Abstract Denotational Semantics
   - Denotational Semantics
   - Adequacy and Full Abstraction
4. An Algebraic View of the Semantics
   - The Algebraic Theory of Effects
   - Resumptions Considered Algebraically
   - Asynchronous Processes Considered Algebraically
   - Processes Considered Algebraically
5. Conclusions
The Theory for Resumptions

We define:

\[ L_{\text{Res}} = L_H \otimes ((L_S \otimes L_\Omega) + L_d) \]

where:

- \( L_d \) is the theory of a unary operator, \( d \), with no axioms.
- The axioms of \( L + L' \) are those of \( L \) and \( L' \) (we assume the operation symbols are disjoint).
- The axioms of \( L \otimes L' \) are those of \( L + L' \) together with the commutativity of the operations of the one over the operations of the other (again assuming disjointness).
Basic Q-Transition Sequences

- $Q$ a poset
- $Q$-transition: $(\sigma, \sigma' x)$ where $x \in Q$
- basic $Q$-transition sequence: $(\sigma_1, \sigma_1), \ldots, (\sigma_n, \sigma_n)[(\sigma, \sigma' x)]$
- $Q$-BTrans is the partial order of $Q$-transition sequences where $u \leq v$ holds iff:
  
  either $u \leq_p v$
  
  or else $\exists w, x \leq y. u \leq_p w(\sigma, \sigma' x) \land v = w(\sigma, \sigma' y)$
Characterisation Theorem for Resumptions

**Theorem**

1. **Viewed as an** $L_{Res}$**-model,** $\mathcal{I}_\omega(Q\text{-BTrans})$ **is** $T_{Res}(\mathcal{I}_\omega(Q))$.

2. **As a semilattice with a zero this is the solution in** $\omega SL$ **of the**
   ‘domain equation’

   $$R \cong (S \times (R_\bot + Id_\omega(Q)))^S$$

   **equivalently**

   $$R \cong (S \times S) \times (R_\bot + Id_\omega(Q))$$

3. **Q-BTrans is the solution in** $Pos$ **of:**

   $$T \cong (S \times S) \times (T_\bot + Q)$$
Asynchronous Processes

$L_{\text{AProc}}$ is $L_{\text{Res}}$ extended by a new constant $\text{halt}$ with the axiom:

$$\text{d}(\Omega) \leq \text{halt}$$

**Theorem**

1. $\text{AProc}$ is the initial $L_{\text{AProc}}$-model, i.e., it is $T_{\text{AProc}}(0)$.
2. As a semilattice with a zero this is the solution in $\omega\text{SL}$ of the ‘domain equation’

$$R \cong (S \times S) \times (R + 1)_\perp$$
Outline

1. Introduction
2. A Language for Cooperative Threads
3. An Elementary Fully Abstract Denotational Semantics
   - Denotational Semantics
   - Adequacy and Full Abstraction
4. An Algebraic View of the Semantics
   - The Algebraic Theory of Effects
   - Resumptions Considered Algebraically
   - Asynchronous Processes Considered Algebraically
   - Processes Considered Algebraically
5. Conclusions
The Theory for Processes

This is:

\[ L_{\text{Proc}} = L_{\text{Res}} + L_{\text{Spawn}} \]

where \( L_{\text{Spawn}} \) is the theory for **spawning** whose signature is that for \( L_{\text{Res}} \) together with two new operation symbols:

- \( \text{async} : 1 \rightarrow AProc \)
- \( \text{yield\_to} : 1 \rightarrow AProc \)

We write

- \( P > t \) for \( \text{async}_P(t) \)
- \( P < t \) for \( \text{yield\_to}_P(t) \)

thinking of ‘two halves of a left action.’
For $P \in \text{AP roc}$ and $Q \in \text{Proc}$ we define:

$$P >_{\text{Proc}} Q = \text{async}(P) \circ Q = \bigcup \{ (\sigma, \tau)u \gg w \mid u \in P, (\sigma, \tau)w \in Q \} \downarrow$$

$$P <_{\text{Proc}} Q = \bigcup \{ (\sigma, \sigma')u \gg v \mid (\sigma, \sigma')u \in P, v \in Q \} \downarrow$$
First Group of Equations

These concern commutation with \( \cup \):

\[
(P \cup_{AProc} P' ) > x = P > x \cup P' > x
\]
\[
P > (x \cup y) = P > x \cup P > y
\]
\[
(P \cup_{AProc} P' ) < x = (P < x) \cup (P' < x)
\]
\[
P < (x \cup y) = P < x \cup P < y
\]
Equations for `async`

\[
\begin{align*}
P > \text{update}
_\text{v}(x) & = \text{update}
_\text{v}(P > x) \\
P > \text{lookup}
_\text{v} (\langle x \rangle) & = \text{lookup}
_\text{v} (\langle P > x \rangle) \\
P > \Omega & = \Omega \\
P > d(x) & = d(P \gg x) \\
P > (P' > x) & = (P \gg P') > x \\
P > (P' < x) & = P' < (P \gg x)
\end{align*}
\]

writing \( P \gg x \) for the ‘left action’ \( P > x \cup P < x \)

The last equation is redundant.
Equations for \texttt{yield\_to}

\[
\begin{align*}
\text{(update}_{A\text{Proc}})_{l,v}(P) < x &= \text{update}_{l,v}(P < x) \\
\text{(lookup}_{A\text{Proc}})_{l}(f) < x &= \text{lookup}_{l}((f(v) < x)_v) \\
\Omega_{A\text{Proc}} < x &= \Omega \\
d_{A\text{Proc}}(P) < x &= d(P \triangleright x) \\
\text{halt}_{A\text{Proc}} < x &= d(x)
\end{align*}
\]
Transition Sequences for Processes

\[ Q\text{-Trans} \overset{\text{def}}{=} (Q \times \text{PSeq})\text{-BTrans} \]

Its elements have the form:

\[
(\sigma_1, \sigma'_1) \ldots (\sigma_m, \sigma'_m)[(\sigma, \sigma'\langle x, (\overline{\sigma}_1, \overline{\sigma}'_1) \ldots (\overline{\sigma}_n, \overline{\sigma}'_n)[\text{done}]\rangle)]
\]
Characterisation Theorems for Processes

**Theorem**

1. **Viewed as an** $L_{Proc}$-**model**, $I_\omega(Q\text{-Trans})$ **is the free model over** $I_\omega^\uparrow(Q)$.
2. **So as a** Res-algebra:
   
   $$T_{Proc}(I_\omega^\uparrow(Q)) \cong T_{Res}(\text{Pool} \times I_\omega^\uparrow(Q))$$
3. **There is an isomorphism of posets**, $\theta : Q\text{-Trans} \cong T_{Seq}\backslash\{\varepsilon\}$, **where** $Q = \{\text{return}\}$ **and so, as a** Proc-algebra:
   
   $$\text{Proc} \cong I_\omega(T_{Seq}\backslash\{\varepsilon\}) \cong T_{Proc}(1)$$
Some Algebraic Reflections

- This is applied domain theory where one is interested in particular models and, particularly, their algebraic structure.
- Having free algebras is a condition on a (generalised) domain theory.
- Here, some structure, particularly the semilattice structure, is ‘nice’ mathematically; the actions are less so.
- Still, parallel constructs are typically not even algebraic operations.
- In the Proc characterisation theorem, Part 2, we do not get the correct left action structure, though there is a wrong structure as Pool is a (commutative) monoid.
- Perhaps a Hopf shuffle algebra would help for a ‘rational algebraic analysis’
Loday's dendriform dialgebras

**Dendriform dialgebras** These are modules $A$ with two binary bilinear operations $<$ and $>$ such that, for all $x, y, z \in A$:

\[
\begin{align*}
(x < y) < z &= x < (y \bowtie z) \\
x > (y > z) &= (x \bowtie y) > z \\
(x > y) < z &= x > (y < z)
\end{align*}
\]

where $x \bowtie y \overset{\text{def}}{=} x < y + y > x$.

**Example** (Day57, Shützenberger58) Languages, i.e., sets of strings, with $<$ = left shuffle, and $>$ = right shuffle. This is *commutative*, meaning that $x < y = y > x$.

**Remark** $(A, \bowtie)$ is a semigroup (in the category of modules), commutative if $A$ is.
Foissy’s dendriform $A$-modules

Given a dendriform algebra $A$ these are modules $M$ with two binary bilinear operations $>, <: A \times M \rightarrow M$ such that, for all $a, b \in A$ and $x \in M$:

$$(a < b) < x = a < (b \triangleright x)$$

$$a > (b > x) = (a \triangleright b) > x$$

$$(a > b) < x = a > (b < x)$$

where $\triangleright: A \times M \rightarrow M$ is given by: $a \triangleright x = a < x + a > x$.

**Example** $\text{AProc}$ is a commutative dendriform algebra, and $\text{Proc}$ is a dendriform module.

**Remark** $\triangleright: A \times M \rightarrow M$ is an action of $(A, \triangleright)$ on $M$. 
Possible Future Work

- Must semantics (compact sets of transition sequences)
- Add variable declaration: a challenge, at the least, for the algebraic part.
- Add higher-types. Can do as have monad, but full abstraction is another matter.
- Change notion of observations: runs with stuttering or mumbling.
- Fairness: all threads in the pool will eventually be chosen in any infinite run.
- Lower level semantics, with block treated as an exception causing a rollback; can then do \texttt{C orelse C}'.
- What equations hold not involving side-effects, conditionals or while loops? Example:
  \[
  \llbracket (\text{async } (C; \text{async } (D))) \rrbracket = \llbracket \text{async } (C; \text{yield}; D) \rrbracket
  \]