# Algebraic semantics

Laws traditionally play an important role in explaining CSP.

It is, after all, a *process algebra.*

They can either be viewed as *explaining* the denotational semantics, or as *constraining* what abstract semantics are allowed. (Depending which you believe are more important.)

An *algebraic semantics* is a formalisation of the latter: an attempt to capture all equivalences between processes using a set of laws and related rules.

At one level any set of laws gives an algebraic semantics: two processes are equivalent iff provably equivalent using the laws. But to make sense we really need a clearer characterisation of what equivalences they prove. You must always be aware of the twin dangers of proving too many, or too few, processes equal.

## Our aims

We will concentrate on finding algebraic semantics congruent to $\mathcal{N}$. In other words we will hope to be able to prove two processes equivalent algebraically iff identified by this semantics.

We will deal only with finitely nonndeterministic CSP over finite $\Sigma$.

Sequencing and $SKIP$ need special treatment: in these lectures we will deal only with the language without them (but see the book).

Relatively few changes are needed for $\mathcal{T}$ and $\mathcal{F}$, and in general you would expect the sets of laws for other equivalences to vary only slightly........

## Approaching $\mathcal{N}$ via algebra

It is easy to tell you haven't equated too many processes when working with an established conguence like $\mathcal{N}$: all you have to do is show all laws (and any rules) are valid. This *soundness* thus requires a (probably simple) lemma for each.

*Though see some exercises in the book for what can go wrong when you don't respect a model properly!*

But how do we know we have *enough* laws? Proving *completeness* is likely to be significantly more difficult: we need to show that any pair of programs that happen to map to the same value in $\mathcal{N}$ can be proved equal using laws, perhaps supplemented by some rules.

Almost all the laws required can be found in Part I of the book. Our effort will be concentrated on developing a procedure to demonstrate completeness. We will devise and transform to a *normal form*: specialised syntax capturing the semantics.

This will be done in several stages.....

## Operational semantics via algebra

Suppose we have a procedure for reducing any process to the form

$$\bigsqcap_{i=1}^{N} ?x : A_i \rightarrow P_i(x)$$

Then we can work out choices it has for initial offers to the environment.

Once one of these choices has been selected, and if a communication then occurs, we can start all over again.

You can regard this as an operational semantics, since it gives a formalised recipe for a possible implementation.

Following functional programming, let's call the above *head normal form* (hnf).

## Laws needed for transformation to hnf

There are four types of laws required to do this:

- the *distributive* laws, for removing nondeterministic choices to the outside;
- the *step* laws, which calculate the first-step actions of constructs like $P \oplus Q$ in terms of those of $P$ and $Q$;
- the unwinding law of recursion; and
- manipulations of the expressions within processes.

We will, for simplicity, just assume the right to all true equations between expressions, thus glossing over the last category (which tell us nothing about communicating behaviour).

## Transforming to hnf

- $?x : A \rightarrow P'$, is already in hnf, so we need do nothing.
- For $P' \sqcap P''$, reduce each of $P'$ and $P''$. The result is in hnf. *We will assume the basic properties of* $\sqcap$ *and* $\bigsqcap$.
- If $P$ has the form $P' \oplus P''$, where $\oplus$ is any other standard binary operator (i.e., $\Box$, $\underset{X}{\|}$, $|||$ or ${}_X\|_Y$), then first reduce each of $P'$ and $P''$, to $P^\dagger$ and $P^\ddagger$, say.
  Next, use the distributive laws of $\oplus$ to move all the top-level nondeterminism from $P^\dagger$ and $P^\ddagger$ to the outside. The result then has the form
  $$\bigsqcap_{i=1}^{N} (?x : A_i \rightarrow P_i^\dagger) \oplus (?x : B_i \rightarrow P_i^\ddagger)$$
  The respective step law ($\langle\Box$-step$\rangle$, $\langle\underset{X}{\|}$-step$\rangle$, $\langle|||$-step$\rangle$ or $\langle{}_X\|_Y$-step$\rangle$) transforms this into the required form.
- If $P$ has the form $P'[\![R]\!]$, then very much the same procedure (using $\langle[\![R]\!]$-dist$\rangle$ and $\langle[\![R]\!]$-step$\rangle$) is followed.

## Reducing hiding

If $P$ has the form $P' \setminus X$, then applying the same procedure (using $\langle$hide-dist$\rangle$ and $\langle$hide-step$\rangle$) will result in a process of the form $\bigsqcap_{i=1}^{N} Q_i$ where the $Q_i$ either have the form $?x : A_i \rightarrow (Q_i' \setminus X)$, which is what we want, or

$$(?x : A_i \rightarrow (Q_i' \setminus X)) \triangleright \bigsqcap\{P''_{i,j} \setminus X \mid j \in \{1, \ldots, M_i\}\}$$

which is not.

The second case means some of the initial actions of $P'$ are hidden and so do not create initial *visible* actions of $P$.

The strategy in the second case is to reduce the processes $P''_{i,j} \setminus X$, and then use the definition of $P \triangleright Q = (P \mathbin{\Box} Q) \sqcap Q$ and $\langle\Box$-step$\rangle$ to organise the result into the correct shape.

## Recursion and regression

If $P$ is recursively defined, simply apply $\langle\mu$-unwind$\rangle$ (or the equational definition of the process) and reduce the result.

Unfortunately we have a problem: the clauses for reducing $P \setminus X$ and $\mu\, p.P$ can both result in infinite regression: consider

$$(\mu\, p.a \to p) \setminus \{a\} \quad \text{and} \quad \mu\, p.p$$

In these cases, the reduction-based operational semantics diverges, which is actually just as well! Since $\mathbf{div}$ is not equivalent to any term in hnf, it follows that our strategy must diverge for any such term.

This is fine from the point of view of operational semantics, but it is unfortunate that we can't handle these terms in transforming to normal form.

## $\mathbf{div}$ as an hnf

The solution to this problem is:

(a) Add the term $\mathbf{div}$ as a further sort of hnf.

(b) Use the various *zero* laws to reduce all terms where $\mathbf{div}$ appears significantly during the calculation of their hnf to $\mathbf{div}$. (For example, $P \,\Box\, (Q\ {}_X\|_Y\ \mathbf{div}) = \mathbf{div}$ by $\langle\Box\text{-zero}^{\mathcal{N}}\rangle$ and $\langle {}_X\|_Y\text{-zero}^{\mathcal{N}}\rangle$.)

(c) Add perhaps *infinitary* rules for reducing diverging terms to $\mathbf{div}$.

## Dealing algebraically with divergence

There can be no finitary and complete rule for the last of these, but there are still useful finitary rules, such as

**Loop**$^{\mathcal{N}}$ If, during an attempt to reduce a process $P$ to head normal form the (syntactic) process $P$ is encountered other than at the root of the reduction, then $P$ is divergent and may be replaced by $\mathbf{div}$.

(A complete but infinitary version is that *if* the reduction of $P$ to hnf fails to terminate, *then* $P =_{FD} \mathbf{div}$.)

Let's assume, for the time being, that all processes we come across reduce finitely (perhaps using **Loop**$^{\mathcal{N}}$ and similar) to hnf.

## What do we want of a normal form?

- A pair of normal form programs are only semantically equal if they are either syntactically identical, or differ in some trivial way.
- Every program is equivalent to one in normal form.
- This equivalence can be demonstrated by the algebraic theory.

*(A good example is full DNF for propositional formulae.)*

Modulo the infinitary treatment of divergence, hnf meets the last two of these, but not the first, *even for the first step of its behaviour*.

There are still far too many ways to present most possibilities.

We will refine hnf into a full normal form, while showing how to transform hnf into it.

## Characterising a process' behaviour

We must be able to tell the following from a normal form:

- Does $P$ diverge immediately? If not,
- what is $P$'s set of initial actions $initials(P)$, and
- what are $P$'s maximal refusals?
- how does it behave after each event *independently of how the event arose*?

The $\mathbf{div}$ case of hnf captures the first, but the others need to be worked on.

## Ambiguity in first step behaviour

A non-$\mathbf{div}$ hnf takes the form

$$\bigsqcap_{i=1}^{N} ?x : A_i \to P_i$$

The set of *acceptances* $\{A_1, \ldots, A_N\}$ determines initial behaviour, but

all of the following choices of action

$$\{\{a, b, c\}, \{a\}, \{b\}\}$$
$$\{\{b, c\}, \{a\}, \{b\}\}$$
$$\{\{a, b, c\}, \{a, b\}, \{a\}, \{b\}\}$$
$$\{\{a, b, c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a\}, \{b\}\}$$

give the same result, and are not the only possibilities for this one.

We need a standard way to represent a given initials/refusals combination in these terms.

## Allowed patterns of acceptances

Let's stipulate that the set $\{A_1, \ldots, A_N\}$

- has the property that $A_1 \supset A_j$ for all $j > 1$ (i.e., $A_1$ is the union of all $A_i$, including itself, and represents $initials(P)$), and
- if $i, j > 1$ are distinct then $A_i \not\subseteq A_j$ (i.e., all these $A_i$ represent minimal acceptances, and hence their complements, the maximal refusals).

From the list on the last slide, we would only allow the first:

$$\{\{a, b, c\}, \{a\}, \{b\}\}$$

We need a procedure for turning an arbitrary hnf into this form, *and simultaneously making the result of any action independent of the $A_i$ it comes from.*

## Manipulating acceptances

Suppose we are starting with any $\{A_1, \ldots, A_n\}$.

$\langle$input-dist$\rangle$ can eliminate any repeats amongst the $A_j$.

If there is only one remaining, there is nothing left to do, *otherwise*

We need to ensure that the collection contains the union and the rest are minimal. Note that:

$$
\begin{aligned}
P \sqcap Q &= (P \sqcap Q) \,\Box\, (P \sqcap Q) \\
&\phantom{=}\ \text{by } \langle\Box\text{-idem}\rangle \\
&= (P \,\Box\, Q) \sqcap P \sqcap Q \\
&\phantom{=}\ \text{by } \langle\Box\text{-dist}\rangle, \langle\sqcap\text{-idem}\rangle, \text{etc.}
\end{aligned}
$$

One can inductively extend this to show (for any $N$ and $Q_i$)

$$\bigsqcap_{i=1}^{N} Q_i = (\Box_{i=1}^{N} Q_i) \sqcap (\bigsqcap_{i=1}^{N} Q_i)$$

## Manipluating acceptances II

When $Q_i = x?A_i \to P_i$, the extra process $Q_1 \,\Box\, \ldots \,\Box\, Q_N$ can be transformed using $\langle\Box$-step$\rangle$ to the form

$$?x : A^\dagger \to R$$

where $A^\dagger = \bigcup_{i=1}^{N} A_i$ and such that, for any $a \in A_j$, we have

$$P_j[a/x] \sqcap R[a/x] = R[a/x]$$

*This gives us the union and a very useful form of $R$: see next slide.*

## Making the successors uniform

$$
\begin{array}{ll}
 & (?x : A^{\dagger} \to R) \\
 & \sqcap\ (?x : A_i \to P_i) \\[1em]
= & (?x : A^{\dagger} \setminus A_i \to R \ \Box ?x : A_i \to R) \\
 & \sqcap\ (?x : A_i \to P_i) \\
 & \text{by } \langle \Box\text{-step} \rangle \text{ and } \langle \nless \cdot \ngtr\text{-idem} \rangle \\[1em]
= & (?x : A^{\dagger} \setminus A_i \to R \ \sqcap ?x : A_i \to P_i) \\
 & \Box\ (?x : A_i \to R \ \sqcap ?x : A_i \to P_i) \\
 & \text{by } \langle \sqcap\text{-}\Box\text{-dist} \rangle \\[1em]
= & (?x : A^{\dagger} \setminus A_i \to R \ \sqcap ?x : A_i \to P_i) \\
 & \Box\ (?x : A_i \to (R \sqcap P_i)) \\
 & \text{by } \langle \text{input-dist} \rangle \\[1em]
= & (?x : A^{\dagger} \setminus A_i \to R \ \sqcap ?x : A_i \to P_i) \\
 & \Box\ (?x : A_i \to R) \\[1em]
= & (?x : A^{\dagger} \setminus A_i \to R \ \Box ?x : A_i \to R) \\
 & \sqcap\ (?x : A_i \to P_i \ \Box ?x : A_i \to R) \\
 & \text{by } \langle \Box\text{-dist} \rangle \\[1em]
= & (?x : A^{\dagger} \to R) \sqcap (?x : A_i \to P_i \sqcap R) \\
 & \text{by } \langle \Box\text{-step} \rangle\text{, etc.} \\[1em]
= & (?x : A^{\dagger} \to R) \sqcap (?x : A_i \to R)
\end{array}
$$

# Manipulating acceptances III

The transformation on the last slide shows that all the successor processes $P_i$ can be changed to $R$.

This is both necessary for the aims of normalisation and helpful in the following: removing any non-minimal $A_j$ ($\supset A_i$, say) for $j > 1$.

The following identity *(convexity)* is proved in an exercise:

$$(P \mathbin{\Box} Q \mathbin{\Box} S) \sqcap P = (P \mathbin{\Box} Q \mathbin{\Box} S) \sqcap (P \mathbin{\Box} Q) \sqcap P$$

For any $A_i \subset A_j$, set

$$\begin{array}{rcl} P & = & ?x : A_i \to R \\ Q & = & ?x : A_j \setminus A_i \to R \\ S & = & ?x : A^{\dagger} \setminus A_j \to R \end{array}$$

and then apply convexity from right to left.

The completes the normalisation of the first step behaviour: we have built a *one-step normal form*.

The clear relationship with the structure of $\mathcal{N}$ makes it easy to see that it really does do what we claim.

## The normal form

The definition of the full normal form is the same except that the successor processes $R[a/x]$ must also be normal form.

To transform to normal form you first transform to one-step normal form, and then normalise the successors.

Are we finished?

- *YES* for recursion-free processes (that may include $\mathbf{div}$).
- *NO* for processes that may go on for ever, since the normal forms would be infinite and the procedure would regress.

We should realise that there is no hope of giving an effective procedure that will work finitely for all processes including recursion, but we do need to know how recursive processes fit into the semantics.

There are several ways of dealing with this problem....all of them at least potentially infinitary.

## Infinite normal forms

One approach is to accept infinite normal form trees, treating them as infinite mutual recursions indexed by

$$T = traces_{\perp}(P) \setminus \{s\hat{\ }t \mid s \in divergences(P) \wedge t \neq \langle\rangle\}$$

If $s \in T$ is such a trace, we can compute $R_s$ as follows.

$$\begin{array}{rcl} R_{\langle\rangle} & = & P \\ R_{s\hat{\ }\langle a\rangle} & = & Q[a/x], \end{array}$$

where applying the one-step normal form transformation to $R_s$ yields something of the form $\bigsqcap_{i=1}^{N} ?x : A_i \rightarrow Q$

Doing this you also get a normal-form expression for each $R_s$ in terms of the $R_{s\hat{\ }\langle a\rangle}$. This yields a constructive tail recursion which defines the vector $\langle R_s \mid s \in T\rangle$ uniquely: by UFP

$$\langle R_s \mid s \in T\rangle = \langle P/s \mid s \in T\rangle$$

Any process equivalent to $P$ will produce an identical recursion: thus we can claim to have a complete algebraic characterisation of our semantics, all be it infinitary.

## Alternative ways of handling infinity

See the book for the following:

1. Since $P =_{FD} Q$ iff, for all $n$, $P \downarrow n =_{FD} Q \downarrow n$, one can decide the former by deciding the latter, each individual case of which will terminate under our assumptions.
2. In many cases the infinite normal forms can be made finite, replacing $T$ by $T/\cong$ where $s \cong t \Leftrightarrow (P/s =_{FD} P/t)$. *This gives the normal form used by FDR.*
3. Recursion can be eliminated from our arguments via the concept of *syntactic approximation*.

## Derived laws

It is noteworthy that many of the laws of CSP are never used in the transformation to normal form. For non-choice operators we really only use zero, step and distributive laws.

But we know that the rest of the laws are true in the equivalence generated by the laws we have used.

Therefore all these other laws ought to be provable from the ones we have used. This is true, but the proofs are usually far from elementary and use inductions plus the infinitary rules.

For example, you might prove inductively that

$$((P \mid\mid\mid Q) \mid\mid\mid R) \downarrow n = (P \mid\mid\mid (Q \mid\mid\mid R)) \downarrow n$$

for all $n$.