Chapter 4: Theoretical basis of LOTOS

- **Translation of LOTOS behaviour expressions into a mathematical model**
    - The Labelled Transition System (LTS) model
    - Operational semantic rules
    - Concept of an equivalence (bisimulation) over LOTOS processes

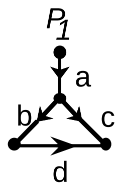
- **Algebraic data type**
    - Equational theory, congruence of terms
    - Operational semantics of (full) LOTOS

Translation into a LTS model

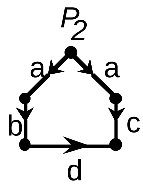
The LOTOS **operational** semantics is defined by axioms and inference rules for **all** LOTOS operators.

LOTOS **behaviour expression** i.e. **instantiation** of a LOTOS process —— **translation by using the semantics** ——> **L**abelled **T**ransition **S**ystem

P1 := a; (b; d; stop [] c; stop) —— **translation by using the semantics** ——>



P2 := a; b; d; stop [] a; c; stop —— **translation by using the semantics** ——>



Labelled Transition System

A **Labelled Transition System** Sys is a 4-tuple $<S, A, T, s_0>$ where

(i) S is a non-empty set of **states**,

(ii) A is a set of **actions**,

(iii) T is a set of **transition relations** $T_a \subseteq S \times S$, one for each $a \in A$;

T_a is a set of **transitions** of the form: cur $\xrightarrow{a}$ next, where cur, next $\in S$

(iv) $s_0 \in S$ is the **initial state** of Sys.

A **state** is unambiguously identified by a **behaviour expression**

An **action** is of the form $\mathbf{g v_1 \ldots v_n}$ where g is a gate name and the v_i are values of some sort

We define: name $(g v_1 \ldots v_n) = g$

There is a distinguished (internal) action: i, which has no associated value.

There is a distinguished (terminating) gate name: δ

But we will consider first Basic LOTOS (without data types)

Operational semantic rules for Basic LOTOS

$a;P \xrightarrow{a} P$

$exit \xrightarrow{\delta} stop$

$$\frac{P \xrightarrow{a} P'}{P\ []\ Q \xrightarrow{a} P'}$$

$$\frac{P \xrightarrow{a} P'}{P\ |[\Gamma]|\ Q \xrightarrow{a} P'\ |[\Gamma]|\ Q}\ (a \notin \Gamma \cup \{\delta\})$$

$$\frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P\ |[\Gamma]|\ Q \xrightarrow{a} P'\ |[\Gamma]|\ Q'}\ (a \in \Gamma \cup \{\delta\})$$

$$\frac{P \xrightarrow{a} P'}{\text{hide } \Gamma \text{ in } P \xrightarrow{a} \text{hide } \Gamma \text{ in } P'}\ (a \notin \Gamma)$$

$$\frac{P \xrightarrow{a} P'}{\text{hide } \Gamma \text{ in } P \xrightarrow{i} \text{hide } \Gamma \text{ in } P'}\ (a \in \Gamma)$$

$$\frac{P \xrightarrow{a} P'}{P >> Q \xrightarrow{a} P' >> Q}\ (a \neq \delta)$$

$$\frac{P \xrightarrow{\delta} P'}{P >> Q \xrightarrow{i} Q}$$

$$\frac{P \xrightarrow{a} P'}{P\ [>\ Q \xrightarrow{a} P'\ [>\ Q}\ (a \neq \delta)$$

$$\frac{P \xrightarrow{\delta} P'}{P\ [>\ Q \xrightarrow{\delta} P'}$$

$$\frac{Q \xrightarrow{a} Q'}{P\ [>\ Q \xrightarrow{a} Q'}$$

$$\frac{P[g_1/h_1,\ldots g_n/h_n] \xrightarrow{a} P',\ Q[h_1,\ldots h_n] := P}{Q\ [g_1,\ \ldots g_n] \xrightarrow{a} P'}$$

Derivation of the LTS associated with a (closed) LOTOS behaviour expression

The given system of axioms and inference rules (denoted D) is used to build a LTS $<S, A, T, s_0>$ associated with any **closed** behaviour expression B as follows:

No free variables

$s_0 = B$

S = Der (B) where Der (B) is the set of derivatives of B,

i.e. the **smallest** set satisfying

(a) $B \in$ Der (B)

(b) if $B' \in$ Der (B) and $D \vdash B' \xrightarrow{a} B''$ for some a, then $B'' \in$ Der (B).

Intuitively, S is the set of states reachable from the initial state B.

$A = G \cup \{i, \delta\}$ where G is the set of gates of B,

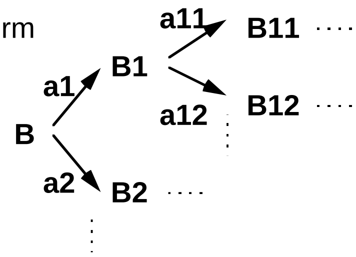
A is the alphabet of the transition system, i.e. all the possible actions.

$T = \{ \xrightarrow{a} \mid a \in A\}$ where $\xrightarrow{a} = \{ < B_1, B_2 > \mid D \vdash B_1 \xrightarrow{a} B_2 \}$

T is the set of transitions derived from D.

Derivation tree

A derivation tree (of B) is of the form



where the outgoing arcs from each non-leaf node are all the actions of the expression at that node.

The tree can be infinite in depth and in width (e.g. in presence of recursion).

It can be seen as the unfolding of the LTS associated with B.

Equivalence over processes

We seek an appropriate equivalence relation over processes, which gives no special status to the silent i-action.

There are of course other weaker equivalences which reflect the idea that the i-action should indeed be silent, i.e. unobservable.

Perhaps the most obvious equivalence of processes is one which requires merely that they should possess the same traces (or sequences of transitions). More exactly, we might declare P and Q to be equivalent just when, for all trace $\sigma = a1.a2\ldots an \in A^*$,

$P \xrightarrow{\sigma}$ iff $Q \xrightarrow{\sigma}$.

But consideration of deadlock leads to the rejection of this proposal. For P and Q would be equivalent if they have the following derivation trees:

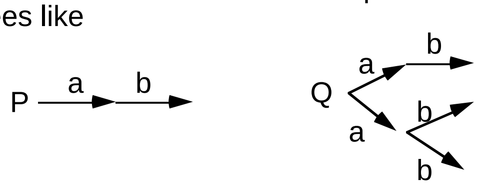
But in this case, after performing a, P will always be able to perform b while Q may not.

Thus, in an "environment" which *demands* b after a, P will not deadlock while Q may.

So, apparently this equivalence is too large.

Equivalence over processes (2)

On the other hand we may be too restrictive; we may take P and Q to be equivalent just when their derivation trees are isomorphic. This would deny the equivalence of P and Q with trees like



even though, at each stage, the same actions are possible.

We therefore seek an intermediate notion, with the following property:

P and Q are equivalent iff
for all a ∈ A, **each** a-successor of P is equivalent to **some** a-successor of Q,
and conversely

where an a-successor of P is any P' such that $P \xrightarrow{a} P'$

Such an equivalence (denoted ~) exists, is a congruence, and has useful algebraic properties.

Towards a definition of ~

This **property** can be expressed more formally as follows:

P ~ Q iff, for all $a \in A$, (*)

(i) whenever $P \xrightarrow{a} P'$ then $\exists\, Q' \bullet Q \xrightarrow{a} Q'$ and P' ~ Q';

(ii) whenever $Q \xrightarrow{a} Q'$ then $\exists\, P' \bullet P \xrightarrow{a} P'$ and P' ~ Q'

However, it is **not a definition**, since there are many relations ~ which satisfy it (including the empty relation).

What we really want is the largest (or weakest, or more generous) relation ~ which satisfies the above property (*).

But is there a largest such relation ?

To see that there is, we adopt an approach which may seem indirect, but which gives us more than a positive answer to the question; it gives us a natural and powerful proof technique.

Strong bisimulation and strong equivalence

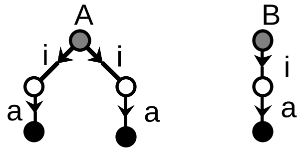
Definition

Let F be a function over binary relations $R \subseteq S \times S$ defined as follows:

$<P, Q> \in F(R)$ iff, for all $a \in A$,

(i) whenever $P \xrightarrow{a} P'$ then $\exists Q' \bullet Q \xrightarrow{a} Q'$ and $<P', Q'> \in R$;

(ii) whenever $Q \xrightarrow{a} Q'$ then $\exists P' \bullet P \xrightarrow{a} P'$ and $<P', Q'> \in R$



An example of a strong bisimulation:

R is composed of all the pairs of states of the same colour

Note that F is monotone, i.e. $R1 \subseteq R2$ implies $F(R1) \subseteq F(R2)$.

Definition

$R \subseteq S \times S$ is a strong bisimulation iff $R \subseteq F(R)$

The empty relation, the identity relation, and the union of two strong bisimulations are strong bisimulations

Definition

P and Q are strongly equivalent (or strongly bisimilar), written $P \sim Q$,

if **there exists** a strong bisimulation R such that $<P, Q> \in R$.

This may be equivalently expressed as follows: $\sim = \bigcup \{R \mid R \text{ is a strong bisimulation}\}$

Properties of ~

1. ~ is the largest strong bisimulation

   Because it is the union of all strong bisimulations, which is still a strong bisimulation

2 ~ is an equivalence

   Because it is reflexive, symmetric and transitive (The identity relation, the converse of a strong bisimulation and the composition of two strong bisimulations are strong bisimulations)

3. ~ is a fixed point of F, i.e. ~ = F (~)

   We know that $\sim\, \subseteq F(\sim)$ because ~ is a strong bisimulation. We show that $F(\sim) \subseteq\, \sim$. From $\sim\, \subseteq F(\sim)$ and the monotonicity of F, we derive $F(\sim) \subseteq F(F(\sim))$. So F (~) is a strong bisimulation, and then $F(\sim) \subseteq\, \sim$ because ~ is the largest one.

4. ~ is the largest fixed point of F

   Let R be a fixed point. Then R is a strong bisimulation as any fixed point. Then $R \subseteq\, \sim$ because ~ is the largest strong bisimulation. So ~ being a fixed point is the largest one.

So ~ can be defined as **the largest relation ~** that satisfies the following property:

P ~ Q iff, for all $a \in A$, (i) whenever $P \xrightarrow{a} P'$ then $\exists\, Q' \bullet Q \xrightarrow{a} Q'$ and $P' \sim Q'$;

(ii) whenever $Q \xrightarrow{a} Q'$ then $\exists\, P' \bullet P \xrightarrow{a} P'$ and $P' \sim Q'$

Simpler definition of a strong bisimulation

A relation $R \subseteq S \times S$ is a strong bisimulation iff:

If $<P, Q> \in R$ **then**, for all $a \in A$,

(i) whenever $P \xrightarrow{a} P'$ then $\exists Q' \bullet Q \xrightarrow{a} Q'$ and $<P', Q'> \in R$;

(ii) whenever $Q \xrightarrow{a} Q'$ then $\exists P' \bullet P \xrightarrow{a} P'$ and $<P', Q'> \in R$

Here F is not defined explicitly, and the inclusion $R \subseteq F(R)$ is implicit in the if-then-else construct:

$<P, Q> \in R$ implies $<P, Q> \in F(R)$,

and $<P, Q> \in F(R)$ iff, for all $a \in A$, (i) and (ii) hold.

This definition is the standard definition of the strong bisimulation.

Proof technique for ~

Problem

Given two processes P and Q. Prove that P ~ Q.

Method (= exhibit an appropriate strong bisimulation containing the pair <P, Q>)

Find a relation R such that $<P, Q> \in R$. This is like finding an invariant.

Prove that R is a strong bisimulation.

Example

Prove that P [] P ~ P. **Note that P is not defined explicitly !** This is an **open** beh. expr.

Let $R = Id \cup \{<P\ []\ P, P> \mid P \in S\}$.

First case: let $P\ []\ P \xrightarrow{a} P'$. It is enough to find P'' such that $P \xrightarrow{a} P''$ and $<P', P''> \in R$.

But $P\ []\ P \xrightarrow{a} P'$ must be inferred from the choice rule, so $P \xrightarrow{a} P'$ (premise of the rule).

Therefore, it suffices to take P'' = P' because $Id \subseteq R$.

The other cases are similar.

Proof of the strong equivalence of two closed behaviour expressions

When the two behaviour expressions are **closed** and the associated LTS are **finite-state**, there are algorithms to prove the strong equivalence of the LTS in polynomial time (with respect to the size of the LTS, not the size of the LOTOS expression).

Example of a LOTOS expression that generates an infinite LTS:

B := a; stop ||| B

However, it is strongly equivalent to the LTS of a; B1 where B1 := a; B1 which is finite.

This is because (stop ||| (stop ||| P)) ~ (stop ||| P).

Therefore by using some strong equivalence laws, it is possible to extend the class of behaviour expressions that have associated finite LTS.

We will give some of them. Note that no sound and complete set of laws for ~ can exist because Basic LOTOS is Turing powerful.

Equational properties of ~

Monoid laws:

P [] Q ~ Q [] P   P [] (Q [] R) ~ (P [] Q) [] R

P [] P ~ P   P [] stop ~ P

Note that the distributive law: a; (P [] Q) ~ a; P [] a; Q is **not** satisfied.

Static laws:

P |[Γ]| Q ~ Q |[Γ]| P

P |[Γ]| (Q |[Γ]| R) ~ (P |[Γ]| Q) |[Γ]| R   Note that the gate sets Γ **must be equal**

stop >> P ~ stop   exit >> P ~ i; P

P >> (Q >> R) ~ (P >> Q) >> R

Note that P >> stop ~ P and P ||| stop ~ P are **not** satisfied. Why ?

P [> (Q [> R) ~ (P [> Q) [> R   P [> stop ~ P

(P [> Q) [] Q ~ P [> Q   stop [> P ~ P

exit [> P ~ exit [] P

They can all be proved by exhibiting an appropriate strong bisimulation

Expansion laws (or expansion theorems)

The purpose of these laws is to expand behaviour expressions by pushing the parallel composition, the disabling and the hiding operators deeper in the process structure.

Let P = Σ {aj; Pj | j $\in$ J} and Q = Σ {bk; Qk | k $\in$ K }

where Σ {Bj | j $\in$ Nat} denotes B0 [] B1 [] B2 [] B3 [] …

These expressions of P and Q can be seen as derivation trees.

Expansion laws:

P |[Γ]| Q ~ Σ {aj; (Pj |[Γ]| Q) | j $\in$ J, aj $\notin$ $\Gamma \cup \{\delta\}$}

[] Σ {bk; (P |[Γ]| Qk) | k $\in$ K, bk $\notin$ $\Gamma \cup \{\delta\}$}

[] Σ {c; (Pj |[Γ]| Qk) | j $\in$ J, k $\in$ K, c = aj = bk $\in$ $\Gamma \cup \{\delta\}$}

P [> Q ~ Q [] Σ {aj; (Pj [> Q) | j $\in$ J, aj $\neq$ δ} [] Σ {aj; Pj | j $\in$ J, aj = δ }

hide Γ in P ~ Σ {aj; hide Γ in Pj | j $\in$ J, aj $\notin$ Γ} [] Σ {i; hide Γ in Pj | j $\in$ J, aj $\in$ Γ}

Congruence

Definition

A LOTOS context C [•] is a LOTOS behaviour expression with a formal process parameter [•] called a hole.

For example, C [•] := hide a in (P ||| •) is a LOTOS context.

If C [•] is a context and P is a behaviour expression, then C [P] is the behaviour expression that is the result of replacing the • occurrence by P.

In the example above, C [Q] := hide a in (P ||| Q)

Definition

An equivalence relation R is a congruence in LOTOS iff, for all P, Q and LOTOS context C [•],

$<P, Q> \in R$ implies $<C[P], C[Q]> \in R$

Theorem: ~ is a congruence in LOTOS

This allows the substitution of a process by a strongly equivalent one in any LOTOS context.

Note that the definition of a congruence is language dependent.

Conclusion on ~

Strong equivalence (congruence) ~ provides a tractable notion of equality of processes.

It allows many nontrivial equalities to be derived.

However, it is deficient in a vital respect: it treats the internal action i on the same basis as all other actions, and properties which we would expect to hold if i is unobservable, such as a; i; P = a; P, do not hold if '=' is taken to mean strong equivalence.

This defect can be removed by defining a weaker equivalence based on the concept of a weak bisimulation. Refer to chapter on equivalence relations.

However, as ~ is the strongest meaningful equivalence, all the equivalence laws that we have presented will remain valid when weaker equivalences are used in the sequel.

Algebraic data types

- Notion of algebraic data type
- ACT ONE semantics
  (equational theory, congruence of terms, quotient algebra, initial algebra)
- Operational semantics of (full) LOTOS
- (Free) constructor, semi-constructor, function
- Completeness and consistency

Algebraic data types

Data type: (It is not a set of values)

Characterized by one or more sets of values **AND** by the allowed operations on the values

Abstract data type:

Data are treated as abstract objects and the semantics of functions operating on data are described by properties

Algebraic data type:

When properties are given in the form of **axioms (logical formulas)**

Equational algebraic data type:

When the axioms are restricted to **equations**

Positive conditional algebraic data type:

When the axioms are restricted to **implications from conjonctions of equations to one equation** (Horn Clause with equality)

E.g. : $X = Z \;\&\; Z = Y \;\Rightarrow\; X = Y$

ACT ONE is a positive conditional algebraic data type language.

Specification of an ADT in ACT ONE

An ADT in ACT ONE is specified by **sorts**, **operations** and **(conditional) equations**

I. Specification of sorts

sorts *Nat*

II. Specification of operations

```
opns   0 :              -> Nat
       succ : Nat       -> Nat
       _+_ : Nat, Nat   -> Nat
```

sorts + operations (over these sorts) = a **signature**

III. Specification of equations

```
eqns   forall x, y : Nat
       ofsort Nat
       x + 0          = x ;
       x + succ(y)    = succ(x+y) ;
```

Combining operations yields **terms**
= <u>representations</u> of values contained in the sorts.

E.g.: *0, succ(0), 0+0, 0+succ(0), succ(0+0)…*

Equational theory

Let E be a set of equations over a set of terms.

The **equational theory, Th (E),** is the **set of equations** that can be obtained by taking

- all instances of equations in E as **axioms**, and
- reflexivity, symmetry, transitivity and context applications as **inference rules.**

For example, the following equations belong to the equational theory associated with the two equations given for _+_:

0 + 0 = 0	instance of first equation
0 + succ(0) = succ(0+0)	instance of second equation
0 = 0	reflexivity
succ(0+0) = 0 + succ(0)	symmetric of *0 + succ(0) = succ(0+0)*
succ(0+0) = succ(0)	by application of context *succ(.)* to *0 + 0 = 0*
0 + succ(0) = succ(0)	transitivity of *0 + succ(0) = succ(0+0)* and *succ(0+0) = succ(0)*

In LOTOS, one uses the concept of **derivation system** instead of an equational theory. The derivation system associated with an ACT ONE specification is composed of the set of axioms and the set of inference rules enumerated in the definition of the equational theory.

Congruence and congruence class

Let <S,OP,E> be an ACT ONE specification (S = Sorts, OP = OPerations, E = Equations) and DS the derivation system generated from it.

Two ground terms s and t are called **E-congruent** iff DS |— s = t or simply E |— s = t

That is if $s=t \in Th(E)$

Other notation: $s \stackrel{E}{=} t$

The **E-congruence class** [t] of a ground term t is the set of all terms E-congruent to t.

[t] = {t' | E |— t' = t}

Ground terms denote values. Congruent ground terms are different denotations for the same value. E.g. '2', '1'+'1', '0' + '2', …

Each value will be represented by the **set** of all its denotations. This leads to the concept of a **quotient** term algebra.

Quotient term algebra or initial algebra

The **quotient term algebra** (or initial algebra) Q(E) of a set of equations E is a model in which the universe consists of one element for each E-congruence class of ground terms.

It is **initial** in the sense that the E-congruence classes are the smallest ones:

two terms are in the same class if this can be proved, otherwise they are considered distinct (no additional properties are considered)

Positive conditional algebraic data types constitute the **largest class** of algebraic data types for which **an initial algebra always exists**.

(e.g. a (non-positive conditional) axiom like $a=b \vee b=c$ has no initial algebra)

The semantical interpretation of an ACT ONE specification <S,OP,E> is the many-sorted algebra <Dq, Oq>, called the **quotient term algebra**, where

- Dq is the set {Q(s) | s ∈ S} where Q(s) = { [t] | t is a ground term of sort s} for each s ∈ S
- Oq is the set of operations {Q(op) | op ∈ OP}, where the Q(op) are defined by

Q(op) ([t1], … [tn]) = [op(t1, … tn)]

The arguments and result of Q(op) are "classes of terms".

[0] **[succ(0)]** **[succ(succ(0))]** **…**

Q(+)

Derivations

A **substitution** σ is a special kind of replacement operation, uniquely defined by a mapping from variables to terms.

Example: Let a substitution σ be defined by {x→succ(0), y→0} and the term s = succ(x+y). Then sσ = succ(succ(0)+0)

A **context** is a term with a hole. For example: succ(•).

$s \underset{E}{\leftrightarrow} t$ iff s = u(lσ) and t = u(rσ) for some equation l=r, context u(•) and substitution σ.

One term can be obtained from the other by one replacement of equal terms

For example: succ(0+0) $\underset{E}{\leftrightarrow}$ succ(0) with equation x+0=x, σ={x→0} and context succ(•)

$s \overset{*}{\underset{E}{\leftrightarrow}} t$ is the reflexive-transitive closure of $s \underset{E}{\leftrightarrow} t$

There is a **derivation** between s and t

The following result holds: $s \overset{*}{\underset{E}{\leftrightarrow}} t$ iff E |— s = t

Unification, matching and narrowing

Let E be a set of equations.

A substitution σ is an **E-unifier** of s = t if $s\sigma =_E t\,\sigma$

For example $\sigma = \{x \to 0+0,\ y \to succ(0)\}$ is a unifier of succ(x) = y (in the Nat theory)

s and t are **E-unifiable** if there exists an E-unifier of s = t

t **E-matches** s if there exists a substitution σ such that $s\sigma =_E t$

The **unification problem** is to determine the set of all E-unifiers σ of s = t.

A **narrower** is an algorithm that finds E-unifiers

A **complete narrower** is an algorithm that solves the unification problem (i.e. that finds all the E-unifiers)

Operational semantics of full LOTOS

First phase: the flattening mapping

This phase produces a canonical LOTOS specification (CLS) where all identifiers are made **unique** (by a suitable relabelling) and **defined at one global level**.

A **canonical LOTOS specification** CLS is a 2-tuple <CAS, CBS> composed of:

- CBS = <PDEFS, pdef0> : a canonical behaviour specification, i.e. a set of process definitions PDEFS with an initial definition pdef0 $\in$ PDEFS (the **behaviour** of the spec)
- CAS = <S,OP,E> : a canonical algebraic specification such that the signature <S, OP> contains all sorts and operations occurring in CBS

This flattening mapping is **partial** since only **static semantically** correct specifications have a well-defined CLS.

Second phase: buiding of the derivation system DS of CAS

The semantic interpretation of CAS is the many-sorted Quotient term algebra Q(CAS)

Third phase: mapping of CLS onto a LTS

Based on a set of operational semantic rules (see next slide)

Operational semantic rules for full LOTOS

An **action** is of the form $\mathbf{gv_1 \ldots v_n}$ where g is a gate name and the v_i are values of some sort

We define: name $(gv_1 \ldots v_n) = g$

Two examples of axioms:

$\text{Exit}\ (E_1, \ldots E_n) \xrightarrow{\delta v_1 \ldots v_n} \text{stop}$ **provided that**

- $v_i = [E_i]$ if E_i is a ground term
- $v_i \in Q(s_i)$ if E_i = **any si**

$\text{exit (true or false)} \xrightarrow{\delta\ \text{true}} \text{stop}$

$g d_1 \ldots d_n\ [SP];\ P \xrightarrow{g v_1 \ldots v_n} [ty_1 / y_1, \ldots ty_m / y_m]\ P$ **provided that**

- $v_i = [E_i]$ if $d_i = !E_i$
- $v_i \in Q(s_i)$ if $d_i = ?x_i{:}s_i$
- $\{y_1, \ldots y_m\} = \{x_i \mid d_i = ?x_i{:}s_i\}$
- The ty_j are term instances with $[ty_j] = v_i$ if $y_j = x_i$ and
- $\mathbf{DS \vdash [ty_1 / y_1, \ldots ty_m / y_m]\ SP}$

$g?x{:}nat!true\ [x \le 1];\ P \xrightarrow{g\ 1\ \text{true}} [1/x]\ P$

This implies to find all the solutions of SP (Cf. unification problem)

Constructors and functions

There are algebraic data type languages in which the operations are clearly partitioned into two classes: the constructors and the functions.

Even if it is not the case in ACT ONE (where there are just operations), it is useful to make this distinction because most LOTOS tools are based on this distinction or require the user to provide this extra piece of information.

Constructors are used to 'build data'.

For example: 0 and succ to define the natural numbers

Functions are all the operations that are not constructors

For example: _+_

If there exists an equation that involves constructors **only**, these constructors are called **semi-constructors**.

This is because they are used to build data like constructors, but they also look like functions due to the presence of these equations.

Other constructors are called **free constructors.**

Examples of semi-constructors

Integers

```
sort int
opns   0: -> int                (* free constructor *)
       succ, pred: int -> int   (* semi-constructors*)
eqns forall x:int
       ofsort int
       succ(pred(x)) = x
       pred(succ(x)) = x
```

Sets

```
sort elem, set
opns   a,b,c : -> elem            (* free constructors *)
       Ø: -> set                  (* free constructor *)
       insert: elem,set -> set    (* semi-constructor *)
eqns forall e1, e2: elem, s:set
       ofsort set
       insert (e1, insert (e1, s)) = insert (e1, s)
       insert (e1, insert (e2, s)) = insert (e2, insert (e1, s))
```

Many tools don't like semi-constructors

Removal of semi-constructors

Example: succ and pred are semi-constructors in the integer theory

ofsort int
succ (pred (x)) = x
pred (succ (x)) = x

These equations should be rewritten as follows:

succ (0) = succ' (0)
succ (succ' (x)) = succ' (succ' (x))
succ (pred' (x)) = x
pred (0) = pred' (0)
pred (pred' (x)) = pred' (pred' (x))
pred (succ' (x)) = x

succ' and pred' are **constructors**
succ and pred are **functions**

! **pred' (succ' (x)) ≠ x**

But terms like pred' (succ' (x)) should never appear

All terms composed of succ and pred can be rewritten into terms composed of **either** succ' only, **or** pred' only, **but not both**

Completeness

The specification E is **sufficiently complete** (or "has no junk") **with respect to the set of constructors**, if every ground term t is provably equal to a constructor term (i.e. a term that is built from constructors only).

Informally, this means that **all functions are total**, or totally defined.

Partial functions lead to incompleteness and introduce "junk" terms.

Example of an incomplete specification:

```
sort nat
opns   0: -> nat                    (* free constructor *)
       succ: nat -> nat             (* free constructor *)
       pred: nat -> nat             (* function *)
eqns forall x:nat
    ofsort nat
       pred (succ (x)) = x;
```

'pred (0)' cannot be proved equal to a constructor term. It is a "junk" term.

The reason is that the pred function is partial because no equation is given for 'pred (0)'.

Consistency

The specification E is **consistent** (or "has no confusion") **with respect to the set of constructors**, if for arbitrary ground **constructor** terms s and t,

$E \models s = t \quad \text{iff} \quad E_C \models s = t$

where E_C is the subset of equations involving constructors only (e.g. pred(succ(int))=int)

Informally, a specification is consistent if constructor terms that cannot be equated by means of equations in E_C denote distinct values (i.e. no confusion).

If all constructors are free, then E_C = Ø, and s = t cannot hold for constructor terms s and t.

Example of an inconsistent specification:

sort nat
opns 0: -> nat (* free constructor *)
error: -> nat (* semi-constructor *)
succ: nat -> nat (* semi-constructor *)
pred: nat -> nat (* function *)
_*_: nat, nat -> nat (* function *)
eqns forall x, y:nat
ofsort nat
succ (error) = error;
pred (succ (x)) = x; pred (0) = error; pred (error) = error
x * 0 = 0; x * error = error; x * succ (y) = …; x*y = y*x;

This equation constitutes E_C

One can prove 0 = error * 0 = error but 0 = error cannot be proved from E_C
In other words, the function _*_ turns two distinct values into equivalent ones