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
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

\[
P_1 := a; (b; d; \text{stop}) [] c; \text{stop}
\]

\[
P_2 := a; b; d; \text{stop} [] a; c; \text{stop}
\]
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: $\text{cur} \rightarrow \text{next}$, where $\text{cur}, \text{next} \in S$
(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 $gv_1...v_n$ where $g$ is a gate name and the $v_i$ are values of some sort

We define: $\text{name}(gv_1...v_n) = g$

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

There is a distinguished (terminating) gate name: $\delta$

But we will consider first Basic LOTOS (without data types)
Operational semantic rules for Basic LOTOS

\[
\begin{align*}
\frac{a;P \xrightarrow{a} P}{P \xrightarrow{a} P'} & \quad \text{exit } \delta \rightarrow \text{stop} \\
\frac{P \xrightarrow{a} P'}{P \parallel \Gamma || Q \xrightarrow{a} P' \parallel \Gamma || Q} & \quad (a \in \Gamma \cup \{\delta\}) \\
\frac{P \xrightarrow{a} P'}{\text{hide } \Gamma \text{ in } P \xrightarrow{a} \text{ hide } \Gamma \text{ in } P'} & \quad (a \notin \Gamma) \\
\frac{P \parallel Q \xrightarrow{a} P' \parallel \Gamma || Q}{P \parallel Q \xrightarrow{a} Q'} & \quad (a \in \Gamma \cup \{\delta\}) \\
\frac{P \parallel Q \xrightarrow{a} Q'}{\text{hide } \Gamma \text{ in } P \parallel Q \xrightarrow{a} \text{ hide } \Gamma \text{ in } P'} & \quad (a \in \Gamma) \\
\frac{P \gg Q \xrightarrow{a} P' \gg Q}{P \gg Q \xrightarrow{a} Q'} & \quad (a \notin \delta) \\
\frac{P \gg Q \xrightarrow{a} P' \gg Q}{P \gg Q \xrightarrow{a} Q'} & \quad (a \notin \delta) \\
\frac{P \parallel Q \xrightarrow{a} P' \parallel Q}{P \parallel Q \xrightarrow{a} Q'} & \quad (a \notin \delta) \\
\frac{P[\frac{g_1}{h_1}, \ldots, \frac{g_n}{h_n}] \xrightarrow{a} P', Q[\frac{h_1}{h_1}, \ldots, \frac{h_n}{h_n}] := P}{Q[\frac{g_1}{h_1}, \ldots, \frac{g_n}{h_n}] \xrightarrow{a} P'} & \quad (a \notin \delta)
\end{align*}
\]
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:

$s_0 = B$

$S = \text{Der}(B)$ where $\text{Der}(B)$ is the set of derivatives of $B$,

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

(a) $B \in \text{Der}(B)$

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

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.

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

$T$ is the set of transitions derived from $D$. No free variables
A derivation tree (of $B$) is of the form

![Derivation tree diagram]

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 $\sigma = a_1.a_2…a_n \in A^*$,

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

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

\[
\text{P and Q are equivalent iff for all } a \in A, \text{ each } a\text{-successor of } P \text{ is equivalent to some } a\text{-successor of } Q, \text{ and conversely}
\]

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:

\[ P \sim Q \text{ iff, for all } a \in A, \quad (*) \]

(i) whenever \( P \xrightarrow{a} P' \) then \( \exists Q' \cdot Q \xrightarrow{a} Q' \) and \( P' \sim Q' \);

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

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

What we really want is the largest (or weakest, or more generous) relation \( \sim \) 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:

\[
<P, Q> \in F(R) \text{ iff, for all } a \in A,
\]

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

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

Note that \( F \) is monotone, i.e. \( R_1 \subseteq R_2 \) implies \( F(R_1) \subseteq F(R_2) \).

**Definition**
\( R \subseteq S \times S \) is a strong bisimulation iff \( R \subseteq F(R) \).

**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 ~ \subseteq F (~) because ~ is a strong bisimulation. We show that F (~) \subseteq ~.
   From ~ \subseteq F (~) and the monotonicity of F, we derive F (~) \subseteq F (F (~)). So F (~) is a strong bisimulation, and then F (~) \subseteq ~ 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 ~ 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:

\[ P \sim Q \text{ iff, for all } a \in A, \begin{cases} (i) & \text{whenever } P \xrightarrow{a} P' \text{ then } \exists Q' \cdot Q \xrightarrow{a} Q' \text{ and } P' \sim Q'; \\ (ii) & \text{whenever } Q \xrightarrow{a} Q' \text{ then } \exists P' \cdot P \xrightarrow{a} P' \text{ and } P' \sim Q' \end{cases} \]
A relation $R \subseteq S \times S$ is a strong bisimulation iff:

If $\langle P, Q \rangle \in R$ then, for all $a \in A$,

(i) whenever $P \xrightarrow{a} P'$ then $\exists Q' \cdot Q \xrightarrow{a} Q'$ and $\langle P', Q' \rangle \in R$;

(ii) whenever $Q \xrightarrow{a} Q'$ then $\exists P' \cdot P \xrightarrow{a} P'$ and $\langle P', Q' \rangle \in R$

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

$\langle P, Q \rangle \in R$ implies $\langle P, Q \rangle \in F(R)$,

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

This definition is the standard definition of the strong bisimulation.
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> ∈ 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 ∪ {<P [] P, P> | P ∈ S }.
First case: let P [] P a → P'. It is enough to find P" such that P → a P" and <P', P"> ∈ R.
But P [] P a → P' must be inferred from the choice rule, so P → a P' (premise of the rule).
Therefore, it suffices to take P" = P' because Id ⊆ 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:

\[ P \parallel Q \sim Q \parallel P \]
\[ P \parallel P \sim P \]

Note that the distributive law: \( a; (P \parallel Q) \sim (P a; Q) \parallel a; Q \) is not satisfied.

Static laws:

\[ P [[\Gamma]] Q \sim Q [[\Gamma]] P \]
\[ P [[\Gamma]] (Q [[\Gamma]] R) \sim (P [[\Gamma]] Q) [[\Gamma]] R \]

Note that the gate sets \( \Gamma \) must be equal

\[ \text{stop} \triangleright P \sim \text{stop} \]
\[ \text{exit} \triangleright P \sim \text{i} \parallel P \]
\[ \text{P} \triangleright (Q \triangleright R) \sim (P \triangleright Q) \triangleright R \]
\[ (P \triangleright Q) \parallel Q \sim P \triangleright Q \parallel P \]
\[ \text{exit} \triangleright P \sim \text{exit} \parallel P \]

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 \( P = \Sigma \{ a_j; P_j \mid j \in J \} \) and \( Q = \Sigma \{ b_k; Q_k \mid k \in K \} \)

where \( \Sigma \{ B_j \mid j \in \text{Nat} \} \) denotes \( B_0 \Box B_1 \Box B_2 \Box B_3 \Box \ldots \)

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

Expansion laws:

\[
P \parallel_\Gamma Q \sim \Sigma \{ a_j; (P_j \parallel_\Gamma Q) \mid j \in J, a_j \not\in \Gamma \cup \{ \delta \} \}
\]

\[
[\ ] \Sigma \{ b_k; (P \parallel_\Gamma Q_k) \mid k \in K, b_k \not\in \Gamma \cup \{ \delta \} \}
\]

\[
[\ ] \Sigma \{ c; (P_j \parallel_\Gamma Q_k) \mid j \in J, k \in K, c = a_j = b_k \in \Gamma \cup \{ \delta \} \}
\]

\[
P > Q \sim Q \mid \Sigma \{ a_j; (P_j > Q) \mid j \in J, a_j \neq \delta \} \mid \Sigma \{ a_j; P_j \mid j \in J, a_j = \delta \}
\]

hide \( \Gamma \) in \( P \) \sim \Sigma \{ a_j; \text{hide } \Gamma \text{ in } P_j \mid j \in J, a_j \not\in \Gamma \} \mid \Sigma \{ i; \text{hide } \Gamma \text{ in } P_j \mid j \in J, a_j \in \Gamma \}
\]
**Definition**
A LOTOS context $C[\cdot]$ is a LOTOS behaviour expression with a formal process parameter $[\cdot]$ called a hole.

For example, $C[\cdot] := \text{hide } a \text{ in } (P ||| \cdot)$ is a LOTOS context.

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

In the example above, $C[Q] := \text{hide } a \text{ in } (P ||| Q)$

**Definition**
An equivalence relation $R$ is a congruence in LOTOS iff, for all $P$, $Q$ and LOTOS context $C[\cdot]$,

$\langle P, Q \rangle \in R \implies \langle C[P], C[Q] \rangle \in R$

**Theorem:** $\sim$ 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 conjunctions of equations to one equation (Horn Clause with equality)
  E.g.: \( X = Z \land Z = Y \Rightarrow X = Y \)

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
= representations of values contained in the sorts.

E.g.: 0, succ(0), 0+0, 0+succ(0), succ(0+0)…
Let $E$ be a set of equations over a set of terms. The **equational theory**, $\text{Th}(E)$, 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 $+_\mathbb{N}$:

- $0 + 0 = 0$ instance of first equation
- $0 + \text{succ}(0) = \text{succ}(0+0)$ instance of second equation
- $0 = 0$ reflexivity
- $\text{succ}(0+0) = \text{succ}(0)$ symmetric of $0 + \text{succ}(0) = \text{succ}(0+0)$
- $\text{succ}(0+0) = \text{succ}(0)$ by application of context $\text{succ}(.)$ to $0 + 0 = 0$
- $0 + \text{succ}(0) = \text{succ}(0)$ transitivity of $0 + \text{succ}(0) = \text{succ}(0+0)$ and $\text{succ}(0+0) = \text{succ}(0)$

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

Two ground terms \(s\) and \(t\) are called **E-congruent** iff \(DS \vdash s = t\) or simply \(E \vdash s = t\).

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

Other notation: \(s \equiv t\).

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

\([t] = \{t' \mid E \vdash t' = t\}\)

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.
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 $a=b \lor b=c$ has no initial algebra)

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

- $Dq$ is the set $\{Q(s) \mid s \in S\}$ where $Q(s) = \{ [t] \mid t$ is a ground term of sort $s \}$ for each $s \in S$

- $Oq$ is the set of operations $\{Q(op) \mid op \in \text{OP}\}$, where the $Q(op)$ are defined by

  $$Q(op) ([t_1], \ldots [t_n]) = [op(t_1, \ldots t_n)]$$

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

\[
\begin{array}{cccccc}
0 & \text{succ}(0) & \text{succ(succ}(0)) & \ldots \\
\hline
Q(+) & & & & \\
\end{array}
\]
A substitution $\sigma$ is a special kind of replacement operation, uniquely defined by a mapping from variables to terms.

Example: Let a substitution $\sigma$ be defined by $\{x \rightarrow \text{succ}(0), \ y \rightarrow 0\}$ and the term $s = \text{succ}(x+y)$. Then $s\sigma = \text{succ}($succ$(0)+0)$

A context is a term with a hole. For example: succ($\ast$).

$s \leftrightarrow t$ iff $s = u(l\sigma)$ and $t = u(r\sigma)$ for some equation $l=r$, context $u(\ast)$ and substitution $\sigma$.

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

For example: succ($0+0$) $\leftrightarrow$ succ($0$) with equation $x+0=x$, $\sigma = \{x \rightarrow 0\}$ and context succ($\ast$)

$s \leftrightarrow t$ is the reflexive-transitive closure of $s \leftrightarrow t$

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

The following result holds: $s \leftrightarrow t$ iff $E \mid \vdash s = t$
Let $E$ be a set of equations.

A substitution $\sigma$ is an $E$-unifier of $s = t$ if $s\sigma = t\sigma$.

For example $\sigma = \{x \rightarrow 0 + 0, y \rightarrow \text{succ}(0)\}$ is a unifier of $\text{succ}(x) = y$ (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 $\sigma$ such that $s\sigma = t$.

The unification problem is to determine the set of all $E$-unifiers $\sigma$ 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: building 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 $gv_1...v_n$ where $g$ is a gate name and the $v_i$ are values of some sort

We define: $\text{name (}gv_1...v_n\text{)} = g$

Two examples of axioms:

$\text{Exit (}E_1,...E_n\text{)} \overset{\delta v_1...v_n}{\rightarrow} \text{stop}$ provided that

- $v_i = [E_i]$ if $E_i$ is a ground term
- $v_i \in Q(s_i)$ if $E_i = \text{any } s_i$

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

$\text{gd}_1...d_n [SP]; P \overset{\delta v_1...v_n}{\rightarrow} [ty_1/y_1,...ty_m/y_m] P$ provided that

- $v_i = [E_i]$ if $d_i = !E_i$
- $v_i \in Q(s_i)$ if $d_i = ?x_i:s_i$
- $\{y_1,...,y_m\} = \{x_i \mid d_i = ?x_i:s_i\}$

The $ty_j$ are term instances with $[ty_j] = v_i$ if $y_j = x_i$ and

$\text{DS} \vdash [ty_1/y_1,...ty_m/y_m] SP$

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

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

pred' (succ' (x)) ≠ x

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. $\text{pred}(\text{succ}(<\text{int}>) = \text{int}$).

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 $E_C = \emptyset$, and $s = t$ cannot hold for constructor terms $s$ and $t$.

Example of an inconsistent specification:

$$\begin{align*}
\text{sort} & \quad \text{nat} \\
\text{opns} & \quad 0: \to \text{nat} \quad \text{(\* free constructor \*)} \\
& \quad \text{error:} \to \text{nat} \quad \text{(\* semi-constructor \*)} \\
& \quad \text{succ: nat} \to \text{nat} \quad \text{(\* semi-constructor \*)} \\
& \quad \text{pred: nat} \to \text{nat} \quad \text{(\* function \*)} \\
& \quad \_ \_ : \text{nat, nat} \to \text{nat} \quad \text{(\* function \*)} \\
\text{eqns for all} & \quad x, y: \text{nat} \\
\text{ofsort} & \quad \text{nat} \\
\text{eqns} & \quad \forall x, y: \text{nat}
\end{align*}$$

This equation constitutes $E_C$

One can prove $0 = \text{error} \times 0 = \text{error}$ but $0 = \text{error}$ cannot be proved from $E_C$

In other words, the function \_\_ \_ turns two distinct values into equivalent ones.