Abstract Data Types
From TRSs to Abstract Data Types (ADTs)

ADTs are a very powerful specification technique which exist in many forms (languages).

These languages are often given operational semantics in a way similar to TRSs (in fact, they are pretty much equivalent)

Most ADTs have the following parts ---

• A type which is made up from sorts
• Sorts which are made up of equivalent sets
• Equivalent sets which are made up of expressions

For example, the integer type could be made up of

• sorts integer and boolean
• 1 equivalence set of the integer sort could be \{3, 1+2, 2+1, 1+1+1\}
• 1 equivalence set of the boolean sort could be \{3=3, 1=1, \text{not}(false)\}
Problem 4: A simple ADT specification

TYPE integer SORTS integer, boolean
OPNS
0:-> integer
succ: integer -> integer
eq: integer, integer -> boolean
+: integer, integer -> integer
EQNS forall x,y: integer
0 eq 0 = true; succ(x) eq succ(y) = x eq y;
0 eq succ(x) = false; succ(x) eq 0 = false;
0 + x = x; succ(x) + y = x + (succ(y));
ENDTYPE
**Problem 4: A simple ADT specification**

<table>
<thead>
<tr>
<th>TYPE integer SORTS integer, boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPNS</td>
</tr>
<tr>
<td>0:-&gt; integer</td>
</tr>
<tr>
<td>succ: integer -&gt; integer</td>
</tr>
<tr>
<td>eq: integer, integer -&gt; boolean</td>
</tr>
<tr>
<td>+: integer, integer -&gt; integer</td>
</tr>
</tbody>
</table>

EQNS for all \( x, y: \text{integer} \):

\[
0 \text{ eq } 0 = \text{true}; \quad \text{succ}(x) \text{ eq succ}(y) = x \text{ eq } y;
\]

\[
0 \text{ eq succ}(x) = \text{false}; \quad \text{succ}(x) \text{ eq } 0 = \text{false};
\]

\[
0 + x = x; \quad \text{succ}(x) + y = x + (\text{succ}(y));
\]

ENDTYPE

**Question:** how do we show, for example ---

- \(1+2 = 3\),
- \(3+2 = 4+1\),
- \(2+2 \neq 3+2\)
### Problem 4: A simple ADT specification

<table>
<thead>
<tr>
<th>TYPE integer SORTS integer, boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPNS</td>
</tr>
<tr>
<td>0:-&gt; integer</td>
</tr>
<tr>
<td>succ: integer -&gt; integer</td>
</tr>
<tr>
<td>eq: integer, integer -&gt; boolean</td>
</tr>
<tr>
<td>+: integer, integer -&gt; integer</td>
</tr>
<tr>
<td>EQNS</td>
</tr>
<tr>
<td>forall x,y: integer 0 eq 0 = true; succ(x) eq succ(y) = x eq y; 0 eq succ(x) = false; succ(x) eq 0 = false; 0 + x = x; succ(x) + y = x + (succ(y));</td>
</tr>
<tr>
<td>ENDTYPE</td>
</tr>
</tbody>
</table>

**Note:** this model is complete and consistent with respect to the modelling of the addition of integers (like the TRS pq-)

**Question:** extend this model to include multiplication
Problem 4: An equivalent ADT specification

Consider changing the original specification to make explicit the fact that $x+y = y+x$, for all integer values of $x$ and $y:

\[
\begin{array}{ll}
\text{TYPE} & \text{integer} \\
\text{SORTS} & \text{integer, boolean} \\
\text{OPNS} & \text{0:-> integer} \\
 & \text{succ: integer -> integer} \\
 & \text{eq: integer, integer -> boolean} \\
 & \text{+: integer, integer -> integer} \\
\text{EQNS} & \forall x, y: \text{integer} \\
 & 0 \text{ eq } 0 = \text{true}; \text{succ}(x) \text{ eq } \text{succ}(y) = x \text{ eq } y; \\
 & 0 \text{ eq } \text{succ}(x) = \text{false}; \text{succ}(x) \text{ eq } 0 = \text{false}; \\
 & 0 + x = x; \text{succ}(x) + y = x + (\text{succ}(y)); \\
 & x+y = y+x; \\
\end{array}
\]

\text{ENDTYPE}

Note: this does not change the meaning of the specification but it may affect the implementation of the evaluation of expressions.
Problem 4: Evaluation termination

If expressions are evaluated as left to right re-writes (as they often are) then evaluation may not terminate:

\[ 3 + 4 = 4 + 3 \] may be re-written as

\[ 4 + 3 = 3 + 4 \] which may be re-written as

\[ 3 + 4 = 4 + 3 \] …

Consequently, there are 4 important properties of ADT specifications:

- **completeness**
- **consistency**
- **confluence**
- **terminating**

With respect to the interpretation Convergent (for both)
Problem 4: Incompleteness, inconsistency and termination

Not having enough equations can make a specification incomplete. For example, the integer ADT specification would be incomplete without the equation:

\[ 0 \text{ eq } 0 = \text{true} \]

Having too many equations can make a specification inconsistent. For example, the integer ADT specification is inconsistent if we add the equation:

\[ x + \text{succ}(0) = x \]

but adding the equation:

\[ x + \text{succ}(0) = \text{succ}(x) \]

would not introduce inconsistency (just redundancy)

Changing the equations may affect termination:

\[ 0 + x = x \text{ to } x + 0 = x \]

would introduce non-termination to the original ADT specification
Problem 4b --- A Set ADT specification

```
TYPE Set SORTS Int, Bool

OPNS
empty:-> Set
str: Set, int -> Set
add: Set, int -> Set
contains: Set, int -> Bool

EQNS forall s:Set, x, y :int
contains(empty, x) = false;
x eq y => contains(str(s,x), y) = true;
not (x eq y) => contains(str(s,x), y) = contains(s,y);
contains(s,x) => add(s,x) = s;
not(contains(s,x)) => add(s,x) = str(s,x)

ENDTYPE
```

Notes:

• use of str and add
• preconditions
• completeness?
• consistency?

Question:

add operations for --

• remove
• union
• equality
Set (model) verification

**Invariant Property:** verify that a set never contains any repeated elements

We would like to verify the following properties:

- \( e \notin (S - e) = \text{true} \)
- \( e \in S_1 \cup S_2 \Rightarrow e \in S_1 \lor e \in S_2 \)

**Question:** Can you sketch the proof (for your set specification)?