Abstract Data Types (ADTs)

Why ADTs?
Main Concepts of ADTs
ADTs as Objects

Why Abstract Data Types (ADTs)?

<table>
<thead>
<tr>
<th>Why Module Interconnection Languages (MILs)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Programming-in-the-small vs. programming-in-the-large&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming-in-the-small</th>
<th>Programming-in-the-small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>building &quot;programs&quot;</td>
</tr>
<tr>
<td>Problem</td>
<td>building &quot;software&quot;</td>
</tr>
<tr>
<td>Emphasis</td>
<td>Usually clear, small</td>
</tr>
<tr>
<td></td>
<td>Usually unclear, large</td>
</tr>
<tr>
<td>Technique</td>
<td>Detailed design &amp; impl.</td>
</tr>
<tr>
<td></td>
<td>sw. architecture</td>
</tr>
<tr>
<td>Notation</td>
<td>Structured programming</td>
</tr>
<tr>
<td></td>
<td>&quot;divide &amp; conquer&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;separation of concerns&quot;</td>
</tr>
<tr>
<td>Version</td>
<td>single person/small number</td>
</tr>
<tr>
<td></td>
<td>multi-person</td>
</tr>
<tr>
<td></td>
<td>interface-oriented</td>
</tr>
<tr>
<td></td>
<td>usually single</td>
</tr>
<tr>
<td></td>
<td>multi-version</td>
</tr>
</tbody>
</table>

✱ hiding the representations
- localized change: change in detailed design & impl. does not affect other (client) objects

✱ specifying the representations
- conceptual consolation

✱ conquering complexity
- decomposing problems into collections of interacting components (-> encapsulation)
Main Concepts of ADTs

- **model system as a collection of ADTs**
  - algebraic specification
  - model-theoretic (axiomatic) specification $\rightarrow \mathbb{Z}$

- **Larch is for Algebraic Interface Specification**

- **Each ADT includes:**
  - data objects
  - operations on data objects
    - (specified as functions with domain and range)
  - essential properties of operations algebraic equations in FOL with equality
    - (specified in algebraic equations in FOL with equality)

- **Two-tiered approach to software development**
  - LCL - Larch Common Language
    - PL-independent notation for writing interface spec.
  - LSL - Larch Shared Language
    - PL-dependent notation for writing interface spec.
    - LSLs exist for SmallTalk, Modula-3, C, C++, CLU, ...

- **large library of predefined specs for common data types**
- **Larch theorem prover for correctness of properties**

Example

```
Stack (E, C): trait
/* A trait describes an ADT, i.e., data objects, ops, properties of ops & links to other ADTs */
/* E is an element (e.g., int, str, bool) of C, a container */
introduces
  new: -> C
  push: C, E -> C
  top: C -> E exemption top (new)
  pop: C -> C exemption pop (new)
  isEmpty: C -> Bool
asserts
  C generated by new, push
  forall stk: C, e: E
    top (push (stk, e)) == e
    pop (push (stk, e)) == stk
    isEmpty (new)
    ~ isEmpty (push (stk, e))
implies
  LinearContainer (push for insert, top for first, pop for rest)
```
Main Concepts of ADTs

Questions

Stack \((E, C)\): trait

- introduces
  - `new` \( \rightarrow C \)
  - `push` \( C, E \rightarrow C \)
  - `top` \( C \rightarrow E \) exempting top (new)
  - `pop` \( C \rightarrow C \) exempting pop (new)
  - `isEmpty` \( C \rightarrow Bool \)

- asserts
  - \( C \) generated by `new`, `push`
  - `top (push (stk, e)) == e`
  - `pop (push (stk, e)) == stk`
  - `isEmpty (new) ~ isEmpty (push (stk, e))`

implies

- `LinearContainer (push for insert, top for first, pop for rest)`

- `pop (top (new)) ==`
- `isEmpty (pop (push (new, 5))) ==`
- `first (rest (insert (insert (new, 10), 11))) ==`

  /* Assume `new` for new */

ADTs as Objects

- Does Fortran/COBOL/C/Pascal offer subroutines for operations?
  Does Fortran/COBOL/C/Pascal offer information hiding?

- Objects (also called "managers") are responsible for
  - preserving the integrity of their resources &
  - hiding the representations from other objects
  - interactions take place mostly thru function and procedure invocations

- encourages reuse (← hiding & encapsulation)
- interaction via procedure call

  - a (client) object should know the identity of another (server) object (cf. pipe&filter)
  - change of an object identity necessitates modification of all other objects
    that explicitly invoke it

Manager

Obj

Procedure Call

Obj

Obj

Obj

Obj