Turbo ASMs
(Composition and Submachine Concepts)

Reference:
Egon Börger and Joachim Schmid:
Composition and Submachine Concepts for Sequential ASMs
Proc. CSL'2000, P. Clote and H. Schwichtenberg (Eds.)
Springer LNCS 1862, August 2000, pages 41–60

See also Chapter 4.1 of:
E. Börger and R. Stärk: Abstract State Machines.
A Method for High-Level System Design and Analysis.
Springer-Verlag 2003

The challenge

Starting point: standard ASMs come with **parallel** execution of **atomic actions** in a **global state** providing clear notions of

- state
- next-step-function

Goal: incorporate **non atomic structuring** concepts—SEQ, iteration, calling parameterized submachines, returning values, local state, error handling—**as standard refinement**s to naturally support

- incremental and modular design of machines
- implementations leading to executable machines

Semantics of Standard ASMs

$$next_R : State(\Sigma) \times Env \to State(\Sigma)$$
$$next_R(\mathfrak{A}, \zeta) = fire_{\mathfrak{A}}([\![R]\!]^{\mathfrak{A}}_{\zeta})$$

$[\![R]\!]^{\mathfrak{A}}_{\zeta}$ is the set of updates defined by rule R in state $\mathfrak{A}$.

Updates are pairs (loc, val) of locations loc and values val, to which the location is intended to be updated.

Locations $f\langle a_1, \ldots, a_n\rangle$ consist of an n-ary function name f with a sequence of length n of elements in the domain of $\mathfrak{A}$.

An update set u is called *inconsistent* if u contains at least two pairs (loc, v_1) and (loc, v_2) with $v_1 \neq v_2$.

Update Sets of Standard ASMs

$$
\begin{aligned}
[\![\mathbf{skip}]\!]^{\mathfrak{A}}_{\zeta} &= \emptyset \\
[\![f(t_1,\ldots,t_n) := s]\!]^{\mathfrak{A}}_{\zeta} &= \{(f\langle [\![t_1]\!]^{\mathfrak{A}}_{\zeta},\ldots,[\![t_n]\!]^{\mathfrak{A}}_{\zeta}\rangle, [\![s]\!]^{\mathfrak{A}}_{\zeta})\} \\
\\
[\![\{R_1,\ldots,R_n\}]\!]^{\mathfrak{A}}_{\zeta} &= [\![R_1]\!]^{\mathfrak{A}}_{\zeta} \cup \cdots \cup [\![R_n]\!]^{\mathfrak{A}}_{\zeta} \\
\\
[\![\mathbf{if}\ t\ \mathbf{then}\ R\ \mathbf{else}\ S]\!]^{\mathfrak{A}}_{\zeta} &= \begin{cases} [\![R]\!]^{\mathfrak{A}}_{\zeta}, & \text{if } [\![t]\!]^{\mathfrak{A}}_{\zeta} = \text{true}^{\mathfrak{A}} \\ [\![S]\!]^{\mathfrak{A}}_{\zeta}, & \textbf{otherwise} \end{cases}
\end{aligned}
$$

Update Sets of Standard ASMs (Ct'd)

$$[\![\textbf{let } x = t \textbf{ in } R]\!]^{\mathfrak{A}}_{\zeta} = [\![R]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} \text{ where } v = [\![t]\!]^{\mathfrak{A}}_{\zeta}$$

$$[\![\textbf{forall } x \textbf{ with } p \textbf{ do } R]\!]^{\mathfrak{A}}_{\zeta} = \bigcup_{v \in V} [\![R]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}}$$

$$\text{where } V = \{v \mid [\![p]\!]^{\mathfrak{A}}_{\zeta \frac{x}{v}} = \text{true}^{\mathfrak{A}}\}$$

Sequencing: Programming Solution

if $mode = \text{First}$ **then** $R,\ mode := \text{Second}$
if $mode = \text{Second}$ **then** $S,\ mode := \ldots$

Problems: - initialization of *mode*
- interleaving of parallel refinements

say of R by R_1 **seq** R_2 **seq** R_3 and of S by S_1 **seq** S_2

Encapsulation of Bounded Execution Time by **seq**

Define *pop_back* (Standard Template Library for C++) from

- *move_to_last* (setting pointer to last element)
- *delete* (removing the current element)

if $mode_{pop\_back}$ = Move **then**
  move_last, $mode_{pop\_back}$:= Delete
else
  delete, $mode_{pop\_back}$:= Move

Problem: proper usage presupposes the knowledge that the result of the action is available only after two steps.

Abstract specification: *pop_back* ≡ *move_last* **seq** *delete*

Abstracting Sequencing into a Rule Constructor

Idea. Turn white box view "first P then Q" into black box view P **seq** Q executable in parallel with other *atomic* actions

Semantics: Let P and Q be rules. Define

$$[\![P \textbf{ seq } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}} \oplus [\![Q]\!]^{\mathfrak{A}'}$$

where $\mathfrak{A}' = next_P(\mathfrak{A}) = \mathit{fire}_{\mathfrak{A}}([\![P]\!]^{\mathfrak{A}})$

- if consistent u: add new updates, possibly overwriting

$$u \oplus v = \{(loc, val) \mid (loc, val) \in u \wedge loc \notin Loc(v)\} \cup v$$

- if inconsistent u: keep inconsistent updates

$$u \oplus v = u$$

Lemma 1 *(Compositionality of* **seq***).*

$$next(R_1 \textbf{ seq } R_2) = next(R_2) \circ next(R_1).$$

Lemma 2 *The ASM constructor* **seq** *has left and right neutral element and is associative.*

$$[\![\textbf{skip seq } R]\!]^{\mathfrak{A}} = [\![R \textbf{ seq skip}]\!]^{\mathfrak{A}} = [\![R]\!]^{\mathfrak{A}}$$

$$[\![R_1 \textbf{ seq } (R_2 \textbf{ seq } R_3)]\!]^{\mathfrak{A}} = [\![(R_1 \textbf{ seq } R_2) \textbf{ seq } R_3]\!]^{\mathfrak{A}}$$

Encapsulating Bounded Iteration

Goal: encapsulate a known finite number n of steps into one atomic action.

$$R^n = \begin{cases} \textbf{skip}, & \text{if } n = 0 \\ R^{n-1} \textbf{ seq } R, & \text{if } n > 0 \end{cases}$$

$$\mathfrak{A}_n = \mathit{fire}_{\mathfrak{A}}([\![R^n]\!]^{\mathfrak{A}})$$

Lemma 3 *(Persistence of inconsistency). If* $[\![R_1]\!]^{\mathfrak{A}}$ *is not consistent, then* $[\![R_1 \textbf{ seq } R_2]\!]^{\mathfrak{A}} = [\![R_1]\!]^{\mathfrak{A}}$.

Lemma 4 *(Fixpoint Condition)* $\forall m \geq n > 0$*:*
if $[\![R]\!]^{\mathfrak{A}_{n-1}}$ *is not consistent or if it is empty, then*

$$[\![R^m]\!]^{\mathfrak{A}} = [\![R^n]\!]^{\mathfrak{A}}.$$

Encapsulating Unbounded Iteration

Goal: encapsulate a finite but unbounded number of steps into one atomic action.

Let R be a rule. Define

$$[\![\textbf{iterate}(R)]\!]^{\mathfrak{A}} =$$

$$\begin{cases} \lim\limits_{n\to\infty} [\![R^n]\!]^{\mathfrak{A}}, & \text{if } [\![R]\!]^{\mathfrak{A}_{n-1}} = \emptyset \text{ or } \mathit{inconsistent}([\![R]\!]^{\mathfrak{A}_{n-1}}) \\ & \text{for some } n \\ \mathit{undef}, & \textbf{otherwise} \end{cases}$$

Example: Hiding Iteration Steps by **iterate**()

Java: Initialization of a class and of all its superclasses using an abstract atomic action *initializeClass*(c):

$$
\begin{aligned}
&\mathit{initialize} \equiv \\
&\quad c := \mathit{class}\ \mathbf{seq}\ \mathbf{iterate}(\mathbf{if}\ \neg \mathit{initialized}(c) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{createInitFrame}(c) \\
&\qquad\qquad\qquad\qquad\quad \mathbf{if}\ \neg \mathit{initialized}(\mathit{superClass}(c)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ c := \mathit{superClass}(c))
\end{aligned}
$$

- well defined by finiteness of the class inheritance hierarchy
- abstracts from sequential implementation along class hierarchy

Java Class Initialization Macro

Let $currframe = (method, program, pos, localVars)$ describe
the current computation state.

$$
\begin{array}{l}
createInitFrame(c) \equiv \\
\quad classState(c) := \text{InProgress} \\
\quad frames \quad := frames \cdot (method, program, pos, localVars) \\
\quad method \quad := c/\texttt{<clinit>} \\
\quad program \quad := body(c/\texttt{<clinit>}) \\
\quad pos \quad := 0 \\
\quad localVars \quad := \emptyset
\end{array}
$$

WHILE as guarded iteration

while $(cond)\ R = \mathbf{iterate}(\mathbf{if}\ cond\ \mathbf{then}\ R)$

(success)	**while** $Cond$ **skip**	
(success)	**while** $(false)\ R$	
(failure)	**while** $(true)$	$a := 1$
		$a := 2$
(divergence)	**while** $(true)\ a := a$	

Böhm-Jacopini-ASMs: Functional/Structural Programming

- defined from basic ASMs using seq and iterate
- the only monitored functions: 0-ary in_M
  one per machine M for inputting the sequence of args
- the only static functions: the *initial* functions
  projection, const, $+1, \neq 0$
- each M with only one output function: $out_M : N$
- no shared functions

Computability of Functions

Definition. $f : N^n \to N$ is **M-computable**:
M started with input x terminates with output $f(x)$.

Definition. M is **started with input** x:
in the initial state of M

- $f = undef$ for every dynamic $f \neq in_M$
- $in_M = x$

Definition. M **terminates** in state S **with output** y: in this state out_M gets updated for the first time, namely to y.

Assume: input functions do not change during a computation,
output functions out_M are controlled but not read by M.

Structured Programming Theorem

Proposition. Every partial recursive function can be computed by a Böhm-Jacopini-ASM.

Proof by induction. Each initial function f is computed by the following machine F consisting of only one function update (which reflects the defining equation of f):

$$F \equiv out_F := f(in_F)$$

macros for using F for given arguments in, possibly including to assign its output to out:

$$F(in) \equiv in_F := in \textbf{ seq } F$$
$$out := F(in) \equiv F(in) \textbf{ seq } out := out_F$$

Let $g, h_1, \ldots, h_m$ be computed by $G, H_1, \ldots, H_m$. Then

$$f(x) := g(h_1(x), \ldots, h_m(x))$$

is computed by the following machine F:

$$\{H_1(in_F), \ldots, H_m(in_F)\} \textbf{ seq } out_F := G(out_{H_1}, \ldots, out_{H_m})$$

Computing Primitive Recursion

Let $f(x,0) = g(x), \quad f(x,y+1) = h(x,y,f(x,y))$ and g, h be computed by G, H. Then f is computed by the following machine F:

$$
\begin{array}{l}
\textbf{let } (x,y) = in_F \textbf{ in} \\
\{ival := G(x),\ rec := 0\}\ \textbf{seq} \\
(\textbf{while } (rec < y)\, \{ival := H(x, rec, ival),\ rec := rec + 1\})\ \ \textbf{seq} \\
out_F := ival
\end{array}
$$

Computing μ-Operator

Let $f(x) = \mu y(g(x, y) = 0)$ and Böhm-Jacopini-ASM G
compute g. Then f is computed by the following machine F:

$$\{G(in_F, 0),\ rec := 0\}\ \mathbf{seq}$$
$$(\mathbf{while}\ (out_G \neq 0)\ \{G(in_F, rec + 1),\ rec := rec + 1\})\ \ \mathbf{seq}$$
$$out_F := rec$$

Gödel versus Turing programming (Martin Davis)

> " ... *imperative* ... the successive lines of programs written in these languages can be thought of as *commands* to be executed by the computer ... In the so-called *functional* programming languages (like LISP) the lines of a program are definitions of operations. Rather than telling the computer what to do, they *define* what it is that the computer is to provide."

The equations which appear in the Gödel-Herbrand type equational definition of partial recursive functions "define what it is that the computer is to provide" only within the environment for evaluation of subterms. The corresponding Böhm-Jacopini-ASMs make this context explicit.

Parameterized Recursive Rules: Goal

Goal: a *practitioner's* definition which allows rules as parameters, like in the following example where the given dynamic function *stdout* is updated to "hello world":

rule $R(output) =$
  $output($`"hello world"`$)$

rule $output\_to\_stdout(msg)$
  $stdout := msg$

$R(output\_to\_stdout)$

Parameterized Recursive Rules: Definition

Syntax. If *body* is a rule, then also $R(x_1, \ldots, x_n)$ is with defining equation **rule** $R(x_1, \ldots, x_n) = body$.

Semantics. Idea: unfold nested calls of R into incarnations $R_0, R_1, \ldots$ where each R_i may trigger one more execution of the rule body.

If $[\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$ is defined, then also the meaning of $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$ is defined and

$$[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$$

NB. $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$ is undefined if $R_0, R_1, \ldots$ is infinite.

Examples of Recursive Rules

rule $R(x) = R(x)$: $[\![R(a)]\!]^{\mathfrak{A}}$ is undefined (Kleene IM 1952).

Recursive rule for Java class initialization:

rule *initialize*(c) =
  if *initialized*(*superClass*(c)) **then**
    createInitFrame(c)
  else
    createInitFrame(c) **seq** *initialize*(*superClass*(c))

rule $R(x)$ =**if** $x < 10$ **then** $R(x+1)$ **else** *result* $:= x$:

$$[\![R(a)]\!]^{\mathfrak{A}} = \begin{cases} (result, 10) & \text{for } a \leq 10 \\ (result, a) & \text{for } a \geq 10 \end{cases}$$

Recursive Definition of WHILE

rule *while*$(cond, R) =$
  if *cond* **then**
    R **seq** *while*$(cond, R)$

NB. This WHILE leads to termination only if *cond* eventually becomes false and R does not diverge.

(success) **while** (*false*) R

(divergence) **while** (*true*) R

(divergence) **while** (*cond*)R (in case R diverges)

Rules with Local State: Syntax

Extend declarations of named rules by local dynamic functions f_i with initialization rule $Init_i$:

$$
\begin{array}{l}
\textbf{rule } name(x_1, \ldots, x_n) = \\
\quad \textbf{local } f_1[Init_1] \\
\quad \vdots \\
\quad \textbf{local } f_k[Init_k] \\
\quad body
\end{array}
$$

where $body, Init_i$ are rules.

We write $\textbf{local } f := t$ for $\textbf{local } f[f := t]$.

Rules with Local State: Semantics

The lifetime of each f_i is that of the rule incarnation.
Therefore we remove from the update set u, which is available after the execution of the body of the call, the set $\mathit{Updates}(f_1, \ldots, f_k)$ of updates concerning the local functions.

$$
\begin{aligned}
&[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = \\
&\quad [\![(\{\mathit{Init}_1, \ldots, \mathit{Init}_k\} \ \mathbf{seq} \ \mathit{body})[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}} \\
&\qquad \setminus \mathit{Updates}(f_1, \ldots, f_k)
\end{aligned}
$$

We assume that each rule incarnation has its own set of local dynamic functions (no name clash).

Local State: Example of Primitive Recursion

rule $F(x, y) = body$

where *body* is defined as follows:

local $ival := g(x)$

local $rec \ := 0$

$(\textbf{while}\ (rec < y)\,\{ival := h(x, rec, ival),\ rec := rec + 1\})\ \textbf{seq}$

$out := ival$

ASMs with Return Value

Goal. Information hiding for returning computed values to destinated locations.

Syntax. Let **rule** $R(x_1, \ldots, x_n) = body$ be a rule declaration and *result* occur in *body*, to be used as placeholder for a location. Let t denote a location. Then

$$t \leftarrow R(a_1, \ldots, a_n)$$

is a rule.

Semantics. Let **rule** $R_t(x_1, \ldots, x_n) = body[t/result]$. Define:

$$[\![t \leftarrow R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![R_t(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$$

t is the interface, chosen by the programmer, for communicating results from the callee to the rule caller.

ASMs with Return Value: Example 1

Calling the primitive recursive machine **rule** $F(x, y) = body$,

where *body* uses calls to submachines G and H:

$$\begin{array}{l}
\textbf{local } ival \leftarrow G(x) \\
\textbf{local } rec \ := 0 \\
\textbf{while } (rec < y)\, \{ival \leftarrow H(x, rec, ival),\ rec := rec + 1\}\ \textbf{seq} \\
result := ival
\end{array}$$

becomes

$$f(x, y) \leftarrow F(x, y)$$

ASMs with Return Value: Example 2

Recursive machine computing the factorial function, using multiplication as static function:

rule $Fac(n) =$
  local $x := 1$
  if $n = 1$ **then**
    $result := 1$
  else
    $(x \leftarrow Fac(n-1))$ **seq** $result := n * x$

Using Return Values for Recursion

Goal. Integrate usual recursion schemes into turbo ASMs

Abbreviation. Let R_i, S be turbo ASMs with param sequences x_i of R_i and params y_i of S

$$
\begin{array}{l}
\textbf{let } y_1 = R_1(a_1), \ldots, y_n = R_n(a_n) \textbf{ in } S = \\
\quad \textbf{let } l_1, \ldots, l_n = new(FUN_0) \textbf{ in} \\
\quad\quad \textbf{forall } 1 \leq i \leq n \textbf{ do } l_i \leftarrow R_i(a_i) \textbf{ seq} \\
\quad\quad\quad \textbf{let } y_1 = l_1, \ldots, y_n = l_n \textbf{ in } S
\end{array}
$$

Recursive Quicksort machine

$$
\begin{array}{l}
Quicksort(L) = \\
\textbf{if}\mid L \mid \leq 1 \textbf{ then } result := L \textbf{ else} \\
\quad \textbf{let} \\
\qquad x = Quicksort(tail(L)_{<head(L)}) \\
\qquad y = Quicksort(tail(L)_{\geq head(L)}) \textbf{ in} \\
\quad result := concatenate(x, head(L), y)
\end{array}
$$

Recursive Mergesort machine

$Mergesort(L) =$
if$\mid L \mid \leq 1$ **then** $result := L$ **else**
  let
    $x = Mergesort(Lefthalf(L))$
    $y = Mergesort(Righthalf(L))$ **in**
  $result := Merge(x, y)$

Recursive Merge machine

$Merge(L, L') =$
if $L = \emptyset$ **then** $result := L'$
if $L' = \emptyset$ **then** $result := L$
  elseif $head(L) \leq head(L')$ **then**
    let $x = Merge(tail(L), L')$ **in** $result := concatenate(head(L), x)$
  else
    let $x = Merge(L, tail(L'))$ **in** $result := concatenate(head(L'), x)$

ASMs with Error Handling

Goal. Separate error handling from normal execution. Producing an inconsistent update set is an abstract form of throwing an exception.

- If R fails with an inconsistency over $f(t_1, \ldots, t_n)$, then remove the effects of executing R and execute S.
- If R fails but without inconsistency over $f(t_1, \ldots, t_n)$ or if R succeeds without inconsistency, then proceed with the result of R.

ASMs with Error Handling: Definition

Semantics: Let R and S be rules, $f(t_1, \ldots, t_n)$ a term.

$$[\![\textbf{try } R \textbf{ catch } f(t_1, \ldots, t_n)\ S]\!]^{\mathfrak{A}} =$$
$$\begin{cases} v, & \exists\, v_1 \neq v_2 : (loc, v_1) \in u \wedge (loc, v_2) \in u \\ u, & \textbf{otherwise} \end{cases}$$

where $u = [\![R]\!]^{\mathfrak{A}}$ and $v = [\![S]\!]^{\mathfrak{A}}$ are the update sets of R and S respectively, and loc is the location $f\langle [\![t_1]\!]^{\mathfrak{A}}, \ldots, [\![t_n]\!]^{\mathfrak{A}} \rangle$.