Egon Börger: Turbo ASMs for Recursive Equations

Starting point: Moschovakis'

- claim that identifying algorithms with abstract machines 'does not square with our intuitions about algorithms and the way we interpret and apply results about them'
- suggestion that 'algorithms are *recursive definitions* while machines model *implementations*, a special kind of algorithms'

Moschovakis' concrete challenge: 'If algorithms are machines, then which machine is the mergesort?'

Answer by Blass and Gurevich (Bulletin EATCS 2002)

Describing 'the mergesort algorithm, on its natural level of abstraction, in terms of distributed abstract state machines', instantiating the more general scheme by Gurevich and Spielmann (JUCS 1997) with the intention to support the ASM thesis.

It is questionable whether distributed computation really is 'the natural level of abstraction' of recursive algorithms like Mergesort.

It is a conceptual overkill to explain on the basis of the complex notion of distributed ASM runs the simple mathematical term evaluation mechanism used in standard recursion schemes:

> Converting a recursive ASM to a distributed ASM amounts to making explicit the creation of vassals, the waiting for the vassals' return values, and the use of these return values to continue the computation.

The functional programming answer

Interpreters of functional languages are well-known abstract machines for mergesort-like algorithms.

However one may argue that defining such interpreters has to deal with features which are typical for *implementing* functional languages, like passing subcomputation values, spawning or deleting subtasks, etc.—routine matters for implementors of functional languages and involving much more than a mathematical user of systems of recursive equations would like to see.

An abstract answer in terms of turbo ASMs

We explicitly extract the high-level *mathematical* machinery, a simple mono-agent turbo ASM, which is tacitly used for the standard functional calculations and without which recursive equations would not constitute the description of an algorithm.

We take care to provide our model for the conceptual ingredients of functional programming at a more abstract level than that of the ASM engines AsmGofer, AsmL, XASM.

Definition of Turbo ASMs

Let $R(x_1, \ldots, x_n) = body$ be the declaration of a turbo ASM rule R, let $\mathfrak{A}$ be a state. If $[\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$ is defined, then also $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$ is defined and its value is $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$.

By this definition every call $R(a)$ of a turbo submachine provides its global result in *one* step (if it terminates at all), namely by yielding the cumulative update set of its entire computation (where in the case of overwriting due to sequential execution the last write wins).

Similar definition for **seq** and **iterate**. See E. Börger and J. Schmid: *Composition and Submachine Concepts for Sequential ASMs*. Proc. CSL'2000, Springer LNCS 1862

Projecting return values out of the final state

To exploit the atomic view of turbo ASMs for returning values by machines which are supposed to compute functions of their input, it suffices to project that value out of the total computational effect $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$.

Let **result** represent the interface for communicating results from a rule execution to a location l which can be determined by the caller.

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

This models the *functional abstraction* from everything in R's computation except the resulting input-output (argument-value) relation.

Introducing placeholders for subcomputation results

Let R_i, S be arbitrary turbo ASMs with formal parameter sequences x_i of R_i and parameters y_i of S. For corresponding actual parameter sequences a_i define:

let $\{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\}$ **in** $S \equiv$

  let $l_1, \ldots, l_n = new(FUN_0)$ **in**

    forall $1 \leq i \leq n$ **do** $l_i \leftarrow R(a_i)$

    seq

    let $y_1 = l_1, \ldots, y_n = l_n$ **in** S

new is supposed to provide each time a completely fresh location (neither used before nor used for any other simultaneous call of new).

Alternative: turn **result** into a monadic function which takes the list of parameters as arguments, so that the results of invokations with different arguments are stored in different locations.

Remarks on abstract placeholders

Passing (by value) the result returned by a turbo ASM captures the implicit storage of intermediate values by subterms during the evaluation of functional equations, abstracting from their ordering and from their deletion (as realized e.g. by a stack discipline).

The definition does not invite to mix terms (which are to be evaluated to objects) and machines (which have to be executed to obtain the desired state change).

> "The idea of XASM is to generalize the original idea of Gurevich, resulting in a more practical specification and implementation tool" by introducing a new concept of "XASM call" that "leads to a design where every construct (including expressions and rules of Gurevich's ASMs) is denoted by both a value and an update set" (Kutter 2002).

A recursive turbo ASM for Quicksort

Evaluating the well-known recursive equations: FIRST partition the $tail$ of the list into the two sublists $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ of elements $< head(L)$ respectively $\geq head(L)$ and quicksort these two sublists separately (independently of each other), THEN $concatenate$ the results placing $head(L)$ between them.

$\textsc{Quicksort}(L) =$

if $| L | \leq 1$ **then result** $:= L$ **else**

let

$x = \textsc{Quicksort}(tail(L)_{<head(L)})$

$y = \textsc{Quicksort}(tail(L)_{\geq head(L)})$

in result $:= concatenate(x, head(L), y)$

No mention of agents, creation of vassals, returning values!

A recursive turbo ASM for Mergesort

FIRST split the given list into a $LeftHalf(L)$ and a $RightHalf(L)$ (if there is something to split) and mergesort these two sublists separately (independently of each other), THEN $Merge$ the two results by an auxiliary elementwise $Merge$ operation.

$\textsc{Mergesort}(L) =$

if $|\ L\ | \leq 1$ **then result** $:= L$ **else**

let

$x = \textsc{Mergesort}(LeftHalf(L))$

$y = \textsc{Mergesort}(RightHalf(L))$

in result $:= Merge(x, y)$

A recursive turbo ASM for Merge

If both lists are non-trivial, by a case distinction the smaller one of the two list heads is determined and placed as the first element of the $result$ list, concatenating it with the result of a separate and independent $Merge$ operation for the two lists remaining after having removed the chosen smaller head element.

$\textsc{Merge}(L, L') =$

if $L = \emptyset$ **or** $L' = \emptyset$ **then** **result** $:= \iota l(l \in \{L, L'\}$ **and** $l \neq \emptyset)$

elseif $head(L) \leq head(L')$ **then**

let $x = \textsc{Merge}(tail(L), L')$ **in** **result** $:=$ $concatenate(head(L), x)$

elseif $head(L') \leq head(L)$ **then**

let $x = \textsc{Merge}(L, tail(L'))$ **in** **result** $:=$ $concatenate(head(L'), x)$

References

Y. N. Moschovakis, What is an algorithm? In: Mathematics Unlimited—2001 and beyond (B. Engquist and W. Schmid, Ed.), Springer 2001.

A. Blass and Y. Gurevich, Algorithms vs. Machines. In: Bulletin EATCS 2002.

E. Börger and J. Schmid, Composition and Submachine Concepts for Sequential ASMs. In: Computer Science Logic (Proceedings of CSL 2000). Springer LNCS 1862

P. Kutter, Montages — Engineering of Computer Languages. Dissertation ETH Zürich 2002.

