Operational Semantics with Semicommutations

Hendrik Maarand[1] and Tarmo Uustalu[2,1]

[1] Department of Software Science, Tallinn University of Technology, Estonia
hendrik@cs.ioc.ee
[2] Department of Computer Science, Reykjavik University, Iceland
tarmo@ru.is

1 Introduction

Consider the following program representing the message-passing pattern.

$$x := 41; y := 1 \parallel r1 := y; r2 := x$$

Here the first thread stores to x the result of some computation and then sets y to 1 to indicate that it has finished the computation. Intuitively, starting from an initial state where everything is set to zero, this program should not reach a final state where $r1 = 1$ and $r2 = 0$. If we apply an optimization (either in advance or at runtime) which happens to reorder the two instructions in either of the threads, then the final state in question becomes valid. For example, the optimized program could be the following where the instructions of the second thread have been reordered.

$$x := 41; y := 1 \parallel r2 := x; r1 := y$$

While the programs $r1 := y; r2 := x$ and $r2 := x; r1 := y$ give the same result in a sequential setting, executing them in parallel with $x := 41; y := 1$ might give different results. Namely, the optimized program allows the final state where $r1 = 1$ and $r2 = 0$.

The phenomenon described above where "safe" optimizations applied to individual threads of a concurrent program leads to additional behaviours is often referred to as relaxed memory models (relaxed in the sense that more behaviours are allowed).

In the following we describe an operational semantics which is able to capture some optimizations like the one described above. The main idea is that given a program $p; q$ we can, under certain conditions, execute an instruction from q even when p is not yet fully executed. This is an extension of our earlier work [2] where we defined reordering derivative-like operations on regular expressions.

2 Preliminaries

A semicommutation alphabet is an alphabet Σ together with an irreflexive relation $\theta \subseteq \Sigma \times \Sigma$ which is called the semicommutation relation [1]. We write $\theta(a, b)$ for $(a, b) \in \theta$. We extend θ to words and letters by $\theta(\varepsilon, a) =_{\mathrm{df}} \mathsf{tt}$ and $\theta(ub, a) =_{\mathrm{df}} \theta(u, a) \wedge \theta(b, a)$.

The set RES of *regular expressions with shuffle* over an alphabet Σ is given by the grammar:

$$E ::= a \in \Sigma \mid 0 \mid E + E \mid 1 \mid EE \mid E^* \mid E \parallel E.$$

In the following we assume a set Σ of instructions (ranged over by $a, b, c, \ldots$) and a set $\mathbb{S}$ of machine states (ranged over by σ). The set of booleans is denoted by $\mathbb{B}$. The interpretation of instructions as (partial) state transformers is given by $[\![\_]\!] : \Sigma \to \mathbb{S} \rightharpoonup \mathbb{S}$. We use the notation $(\sigma)a =_{\mathrm{df}} [\![a]\!]\sigma$ and extend it to words as $(\sigma)\varepsilon =_{\mathrm{df}} \sigma$ and $((\sigma)u)v =_{\mathrm{df}} (\sigma')v$ if $(\sigma)u = \sigma'$ and $\bot$ otherwise. We write $(\sigma)u{\downarrow}$ to express that $(\sigma)u$ is defined. We consider expressions $E \in \mathsf{RES}$ to be (concurrent) programs over the alphabet of instructions Σ.

3 Semantics

We consider an execution of a program to be a sequence of instructions applied to the initial state. Our goal is to describe the execution of programs in the "relaxed" manner described in the introduction. The question now is: given a program, how to determine those instructions that can be executed next? Intuitively, we allow to execute an instruction a before executing the preceding instructions u when we know that both ua and au lead to the same state.

Definition 1. *A semicommutation relation θ is* conservative *when, for every $a, b \in \Sigma$, we have $\theta(a,b) \Longrightarrow \forall\sigma.(\sigma)ab = (\sigma)ba$.*

It follows that if θ is conservative, then $\theta(u,a)$ implies $(\sigma)ua = (\sigma)au$. Thus we take $\theta(u,a)$ to be the justification that allows us to reorder a before u in the sequence of instructions ua. We now extend this idea to programs: given a program Ea we would like to find a program E' such that every execution u of E' is also an execution of E and a is reorderable with u.

Definition 2. *The* nullability *(empty word property) of a program and the θ-*reorderable part *of a program are given by the functions $\varepsilon : \mathsf{RES} \to \mathbb{B}$ and $R^\theta : \mathsf{RES} \times \Sigma \to \mathsf{RES}$ defined recursively by*

$$\begin{array}{rcl@{\qquad}rcl}
\varepsilon(b) & =_{\mathrm{df}} & \mathsf{ff} & R^\theta_a b & =_{\mathrm{df}} & \mathsf{if}\ \theta(b,a)\ \mathsf{then}\ b\ \mathsf{else}\ 0 \\
\varepsilon(0) & =_{\mathrm{df}} & \mathsf{ff} & R^\theta_a 0 & =_{\mathrm{df}} & 0 \\
\varepsilon(E+F) & =_{\mathrm{df}} & \varepsilon(E) \vee \varepsilon(F) & R^\theta_a(E+F) & =_{\mathrm{df}} & R^\theta_a E + R^\theta_a F \\
\varepsilon(1) & =_{\mathrm{df}} & \mathsf{tt} & R^\theta_a 1 & =_{\mathrm{df}} & 1 \\
\varepsilon(EF) & =_{\mathrm{df}} & \varepsilon(E) \wedge \varepsilon(F) & R^\theta_a(EF) & =_{\mathrm{df}} & (R^\theta_a E)(R^\theta_a F) \\
\varepsilon(E^*) & =_{\mathrm{df}} & \mathsf{tt} & R^\theta_a(E^*) & =_{\mathrm{df}} & (R^\theta_a E)^* \\
\varepsilon(E \parallel F) & =_{\mathrm{df}} & \varepsilon(E) \wedge \varepsilon(F) & R^\theta_a(E \parallel F) & =_{\mathrm{df}} & R^\theta_a E \parallel R^\theta_a F
\end{array}$$

Note that $R^\theta_a E$ just replaces those letters b in E with 0 for which $\theta(b,a)$ does not hold. Nullability is essentially a special case of θ-reorderability where we require $\theta(b,a)$ for every $a \in \Sigma$ (i.e., all letters in the expression would be replaced with 0).

Definition 3. *The θ-*reordering operational semantics *of a program is given by the relation $\to^\theta \subseteq \mathbb{S} \times \mathsf{RES} \times \Sigma \times \mathbb{S} \times \mathsf{RES}$:*

$$\frac{(\sigma)a\!\downarrow}{\langle \sigma, a\rangle \to^\theta (a, \langle (\sigma)a, 1\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E+F\rangle \to^\theta (a, \langle \sigma', E'\rangle)} \qquad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, E+F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, EF\rangle \to^\theta (a, \langle \sigma', E'F\rangle)} \qquad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, EF\rangle \to^\theta (a, \langle \sigma', (R^\theta_a E)F'\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E^*\rangle \to^\theta (a, \langle \sigma', (R^\theta_a E)^* E' E^*\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle \sigma', E' \parallel F\rangle)} \qquad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle \sigma', E \parallel F'\rangle)}$$

A configuration $\langle \sigma, E\rangle$ is terminal when $\varepsilon(E)$. If θ is the empty relation, then we get the ordinary operational semantics where nothing is reordered, i.e., $R^\theta_a E$ essentially becomes the

condition $\varepsilon(E)$. If a derivation sequence starts with $\langle \sigma, E\rangle$ and ends with a terminal configuration $\langle \sigma', E'\rangle$, then σ' is a final state of the program E executed from the initial state σ.

Consider as an example the program $ab \parallel (c+d)ef+g$ and say that the next instruction we wish to execute is e (which is not the first instruction of the second thread). If, for example, we have $\theta(c,e)$, $(\sigma)e{\downarrow}$ and $\neg\theta(d,e)$, then it is the case that

$$\langle \sigma, ab \parallel (c+d)ef+g\rangle \to^{\theta} (e, \langle (\sigma)e, ab \parallel (c+0)1f\rangle).$$

Note that $R^{\theta}_{e}(c+d) = (c+0)$ since we have $\theta(c,e)$ but not $\theta(d,e)$. It can be shown that $ab \parallel (c+0)1f$ is equivalent to $ab \parallel cf$. The instruction g disappeared from the expression since the rules for $+$ resolve the nondeterminism and the instruction d disappeared from the expression since it was a predecessor of e and reordering e with d was not justified.

We now return to the example program from the introduction. Note that here we use ; to denote multiplication. Assume θ such that $\theta(r1 := y, r2 := x)$ (this θ could be the *concurrent-read-exclusive-write* condition) and let σ be the initial state where all variables are set to 0. This allows the following derivation sequence (we have omitted the labels on $\to^{\theta}$).

$$\begin{array}{ll}
\langle \sigma, & x := 41; y := 1 \parallel r1 := y; r2 := x\rangle \quad \to^{\theta} \\
\langle \sigma[r2 \mapsto 0], & x := 41; y := 1 \parallel r1 := y; 1\rangle \quad \to^{\theta} \\
\langle \sigma[r2 \mapsto 0][x \mapsto 41], & 1; y := 1 \parallel r1 := y; 1\rangle \quad \to^{\theta} \\
\langle \sigma[r2 \mapsto 0][x \mapsto 41][y \mapsto 1], & 1; 1 \parallel r1 := y; 1\rangle \quad \to^{\theta} \\
\langle \sigma[r2 \mapsto 0][x \mapsto 41][y \mapsto 1][r1 \mapsto 1], & 1; 1 \parallel 1; 1\rangle
\end{array}$$

Since $\varepsilon(1; 1 \parallel 1; 1)$, we have that the program $x := 41; y := 1 \parallel r1 := y; r2 := x$ executed from the initial state σ leads to a final state where $r1 = 1$ and $r2 = 0$.

4 Future Work

Our plan is to extend this framework so that it is possible to describe (a large part of) the multicopy-atomic ARMv8 memory model [3] and then check this description against the existing litmus-tests for this memory model.

One possible extension is to include reordering actions in the framework. For example, when $\theta(a,b)$ and we reorder b before a, then the instruction a acts on b from the left and b acts on a from the right. Now $x := 1; y := x$ can be reordered as $y := 1; x := 1$. The instruction $y := x$ becomes $y := 1$ and thus reads the value written by $x := 1$ before it has reached the memory.

Another extension we are considering is using a state-dependent θ, i.e., for every state σ we may have a different semicommutation relation θ_σ. This is useful for describing allowed reorderings for instructions like $x := [y]$ which stores to variable x the value read from the memory location whose address is given by the variable y.

Acknowledgments This research was supported by the ERDF funded Estonian national CoE project EXCITE (2014-2020.4.01.15-0018) and the the Estonian Ministry of Education and Research institutional grant no. IUT33-13.

References

[1] Mireille Clerbout and Michel Latteux. Semi-commutations. *Inf. Comput.*, 73(1):59–74, 1987.

[2] Hendrik Maarand and Tarmo Uustalu. Reordering derivatives of trace closures of regular languages. In Wan Fokkink and Rob van Glabbeek, editors, *Proc. of 30th Int. Conf. on Concurrency Theory, CONCUR 2019*, volume 140 of *Leibniz International Proceedings in Informatics*, pages 40:1–40:16. Dagstuhl Publishing, 2019.

[3] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018.