A Programming Language Approach to Fault Tolerance for Fork-Join Parallelism

Mustafa Zengin Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)
{zengin,viktor}@mpi-sws.org

***Abstract*—When running big parallel computations on thousands of processors, the probability that an individual processor will fail during the execution cannot be ignored. Computations should be replicated, or else failures should be detected at run-time and failed subcomputations reexecuted. We follow the latter approach and propose a high-level operational semantics that detects computation failures, and allows failed computations to be restarted from the point of failure. We implement this high-level semantics with a lower-level operational semantics that provides a more accurate account of processor failures, and prove in Coq the correspondence between the high- and low-level semantics.**

I. Introduction

As processors get smaller and more distributed, parallel computations run on increasingly larger number of processors. In the not-so-distant future, we may have large simulations running on a million cores for a couple of days, perhaps in the context of an advanced physics or biology experiment, or perhaps used to certify the safety of an engineering design.

The likelihood of a single processing unit failing during such long-running parallel computations is actually quite high, and can no longer be ignored. For example, if we assume that the *mean time between failures* (MTBF) for a single machine is one year, and we use one thousand machines for a single computation, then the MTBF for the whole computation becomes

$$1 \text{ year} \div 1000 \approx 9 \text{ hours}.$$

A simple—albeit expensive—solution is to use *replication*. In theory, we can straightforwardly deal with a single fail-stop failure with 3-way replication [1], and with a single Byzantine failure with 4-way replication [2]. Replication, however, comes at a significant cost, not only in execution time (since fewer execution units are available), but also in the amount of energy required to compute the correct result.

The alternative approach to replication is to use *checkpointing*: that is, to run the computation optimistically with no replication, to detect any failures that occur, and to rerun the parts of the computation affected by those failures [3]. The benefit of checkpointing over the replication approach is that the effective replication rate is determined by the number of actual failures that occurred in an execution and how large a sub-computation was interrupted rather than the maximum number of failures that the system can tolerate. To implement checkpointing, one assumes that some part of the storage space is safe (non-failing) and uses that to store fields needed to recover from failures. This safe storage subsystem may internally be implemented using replication, but this kind of storage replication is much lighter-weight than replicating the entire computation.

As for proving the correctness of these two approaches, that of replication is relatively straightforward, because it uses correctly computed results from one of the replicas in the system. In the checkpointing approach, however, correctness is not so straightforward, because failed processors can be in inconsistent states and partially computed expressions are used in reexecutions.

In this paper, we formalize checkpointing from a programming language perspective and prove its correctness. For simplicity, we will work in the context of a purely functional programming language with fork-join parallelism (see §II). For this language, we develop a high-level formal operational semantics capturing the essence of the checkpointing approach (see §III). In our semantics, the execution of a parallel computation may fail at any point; failures can then be detected and the appropriate parts of a failed computation can be restarted. This high-level semantics is quite simple to understand, and can thus be used as a basis for reasoning about fault-tolerant parallel programs.

To justify the completeness of our semantics with respect to actual implementations, we also develop a lower-level semantics, which models run-time failures and parallel task execution at the processor level (see §IV). We then prove theorems relating the two semantics and showing that our fault-aware semantics are sound: whenever a program evaluates to a value in the fault-aware semantics (perhaps by failing a few times and recovering from the failures), then it can also evaluate to the same value under the standard fault-free semantics (see §V). All lemmas and theorems in this paper are proved using the Coq proof assistant [4] and are available at:

`http://plv.mpi-sws.org/ftpar`

II. Programming Language

For simplicity, we focus on a minimal purely functional language with built-in parallel tuple evaluation, allowing us to express directly interesting large-scale parallel computations by following the fork-join and map-reduce patterns. As we will discuss further in §VI, the lack of side-effects means that parallel tasks are independent, and so failure detection and recovery can be done locally, at the task level.

$$\frac{}{\underline{n_1} \oplus \underline{n_2} \rightsquigarrow \underline{n_1 \oplus n_2}} \qquad \frac{e_1 \rightsquigarrow e_1'}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2}$$

$$\frac{}{\mathbf{let}\ x = v_1\ \mathbf{in}\ e_2 \rightsquigarrow e_2[v_1/x]}$$

$$\frac{}{(\mathbf{fun}\, f(x).\ e)\, v \rightsquigarrow e[v/x][\mathbf{fun}\, f(x).\ e/f]} \qquad \frac{}{\mathbf{fst}\ (v_1, v_2) \rightsquigarrow v_1}$$

$$\frac{}{\mathbf{snd}\ (v_1, v_2) \rightsquigarrow v_2} \qquad \frac{n \neq 0}{\mathbf{if}\ \underline{n}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_1}$$

$$\frac{}{\mathbf{if}\ \underline{0}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_2} \qquad \frac{e_1 \rightsquigarrow e_1'}{(\!|e_1, e_2|\!) \rightsquigarrow (\!|e_1', e_2|\!)}$$

$$\frac{e_2 \rightsquigarrow e_2'}{(\!|e_1, e_2|\!) \rightsquigarrow (\!|e_1, e_2'|\!)} \qquad \frac{}{(\!|v_1, v_2|\!) \rightsquigarrow (v_1, v_2)}$$

Fig. 1. Rules for small-step fault-free evaluation, $e \rightsquigarrow e'$.

In the following, let x range over program variables, $\underline{n}$ over natural numbers, and f over function names. Values, v, and expressions, e, of our language are given by the following grammar:

$$\begin{aligned} v &::= x \mid \underline{n} \mid (v_1, v_2) \mid \mathbf{fun}\, f(x).\, e \\ \oplus &::= + \mid - \mid \times \mid \div \mid = \mid \neq \mid < \mid \leq \mid \ldots \\ e &::= v \mid v_1 \oplus v_2 \mid v_1\, v_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid (\!|e_1, e_2|\!) \mid \\ &\quad\ \mathbf{fst}\, v \mid \mathbf{snd}\, v \mid \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \end{aligned}$$

In our language, values can be variables, natural numbers, value pairs or (recursive) function definitions. Expressions are either values, arithmetic and logical expressions, function applications, let bindings, parallel tuples, first or second projections of pairs, or conditionals. As in C, our if-then-else construct treats $\underline{0}$ as false and non-zero numbers as true. We present the grammar of expressions in *A Normal Form* (ANF) [5] just to make evaluation order explicit. The only place where we differ from standard ANF and have expressions rather than values is in the parallel tuple construct, $(\!|e_1, e_2|\!)$, because we want to model fork-join parallelism by possibly evaluating the two expressions simultaneously. The language can easily be extended with more constructs, types, etc., but such features are orthogonal to the problem at hand.

III. High-Level Semantics

This section contains the standard fault-free semantics for our language both in big-step and small-step style, as well as high-level big-step fault-prone and recovery semantics. Fault prone semantics allows arbitrarily nested fail-stop failures, and recovery semantics re-executes parts of a failed computation. At the end of the section, we shall show correspondences among the high-level semantics.

A. Fault-Free Evaluation

We have two standard fault-free evaluation semantics: (*i*) a big-step fault-free evaluation, $e \Downarrow v$, which is totally standard and omitted for conciseness, and (*ii*) a small-step reduction relation, $e \rightsquigarrow e'$, which is defined as the least fixed point of the rules in Fig. 1. The rules are fairly standard. For example, the first rule says that arithmetic operations of the programming language simply perform the corresponding operation over natural numbers. Perhaps, the only interesting rules are the last three concerning parallel tuples. The two expressions can be evaluated independently; when both have become values, we get the result as a value pair.

$$\frac{}{v \Downarrow^{\mathsf{fp}} v} \qquad \frac{}{\underline{n_1} \oplus \underline{n_2} \Downarrow^{\mathsf{fp}} \underline{n_1 \oplus n_2}} \qquad \frac{e_1 \Downarrow^{\mathsf{fp}} v_1 \quad e_2[v_1/x] \Downarrow^{\mathsf{fp}} v_2}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow^{\mathsf{fp}} v_2}$$

$$\frac{e[v_2/x][\mathbf{fun}\, f(x).\ e/f] \Downarrow^{\mathsf{fp}} r}{(\mathbf{fun}\, f(x).\ e)\, v_2 \Downarrow^{\mathsf{fp}} r} \qquad \frac{}{\mathbf{fst}\ (v_1, v_2) \Downarrow^{\mathsf{fp}} v_1}$$

$$\frac{}{\mathbf{snd}\ (v_1, v_2) \Downarrow^{\mathsf{fp}} v_2} \qquad \frac{n \neq 0 \quad e_1 \Downarrow^{\mathsf{fp}} r_1}{\mathbf{if}\ \underline{n}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \Downarrow^{\mathsf{fp}} r_1}$$

$$\frac{e_2 \Downarrow^{\mathsf{fp}} r_2}{\mathbf{if}\ \underline{0}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \Downarrow^{\mathsf{fp}} r_2} \qquad \frac{e_1 \Downarrow^{\mathsf{fp}} r_1 \quad e_2 \Downarrow^{\mathsf{fp}} r_2}{(\!|e_1, e_2|\!) \Downarrow^{\mathsf{fp}} r_1 \# r_2}$$

$$\frac{}{e \Downarrow^{\mathsf{fp}} \mathbf{BOT}\,(e)} \qquad \frac{e_1 \Downarrow^{\mathsf{fp}} r_1 \quad \mathsf{not\_value}(r_1)}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow^{\mathsf{fp}} \mathbf{LETL}\,(r_1, x, e_2)}$$

$$\frac{e_1 \Downarrow^{\mathsf{fp}} v_1 \quad e_2[v_1/x] \Downarrow^{\mathsf{fp}} r \quad \mathsf{not\_value}(r)}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow^{\mathsf{fp}} \mathbf{LETR}\,(r)}$$

Fig. 2. Rules for big-step fault-prone evaluation, $e \Downarrow^{\mathsf{fp}} r$.

By standard proofs, we can show that big-step evaluation is deterministic and that small-step evaluation is sound and complete with respect to big-step evaluation.

Theorem 1. If $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.

Theorem 2. $e \rightsquigarrow^* v$ if and only if $e \Downarrow v$.

B. Fault-Prone Evaluation

Fault-prone evaluation, $e \Downarrow^{\mathsf{fp}} r$ (see Fig. 2), reduces an expression, e, to a result r, which may be one of the following:

$$\begin{aligned} r ::= v \mid \mathbf{BOT}(e) \mid \mathbf{LETL}(r, x, e) \mid \mathbf{LETR}(r) \mid \\ \mathbf{PTUPLE}(r_1, r_2) \end{aligned}$$

If we get a value, v, the evaluation is successful, thus not requiring any reexecution (we can easily show that $e \Downarrow^{\mathsf{fp}} v \iff e \Downarrow v$). If, however, e reduces to a non-value result r, then we can inspect r to find where the failure occurred and what parts of the computation had already been successfully executed.

The primary cause of failure in the semantics is that any expression can evaluate to bottom, represented as $\mathbf{BOT}(e)$. The failed expression e is recorded so that it may used in the recovery. Evaluation failures are propagated either by passing the results directly (e.g., as in the case of function application), when the premises of these rules are enough to recover the final result of the expression, or by creating a special data structure, such as $\mathbf{PTUPLE}(r_1, r_2)$, $\mathbf{LETL}(r_1, x, e)$ and $\mathbf{LETR}(r)$. We use these structures to store and avoid re-execution of the successful sub-evaluations that are part of a failed computation. For example, in the rule for parallel tuples, there are four possible outcomes, namely, both branches are successful, both

$$\frac{\dfrac{}{1+1 \Downarrow^{\mathsf{fp}} 2} \qquad \dfrac{}{3 \Downarrow^{\mathsf{fp}} \mathbf{BOT}(3)}}{(\!|1+1,3|\!) \Downarrow^{\mathsf{fp}} \mathbf{PTUPLE}(2, \mathbf{BOT}(3))}$$

Fig. 3. Fault-prone evaluation example

$$\frac{}{v \Downarrow^{\mathsf{recover}} v} \qquad \frac{e \Downarrow^{\mathsf{fp}} r}{\mathbf{BOT}\,(e) \Downarrow^{\mathsf{recover}} r} \qquad \frac{r \Downarrow^{\mathsf{recover}} v \quad e[v/x] \Downarrow^{\mathsf{fp}} v_2}{\mathbf{LETL}\,(r,x,e) \Downarrow^{\mathsf{recover}} v_2}$$

$$\frac{r \Downarrow^{\mathsf{recover}} r_1 \qquad \mathsf{not\_value}(r_1)}{\mathbf{LETL}\,(r,x,e) \Downarrow^{\mathsf{recover}} \mathbf{LETL}\,(r_1,x,e)}$$

$$\frac{r \Downarrow^{\mathsf{recover}} v \quad e[v/x] \Downarrow^{\mathsf{fp}} r_2 \quad \mathsf{not\_value}(r_2)}{\mathbf{LETL}\,(r,x,e) \Downarrow^{\mathsf{recover}} \mathbf{LETR}\,(r_2)} \qquad \frac{r \Downarrow^{\mathsf{recover}} v}{\mathbf{LETR}\,(r) \Downarrow^{\mathsf{recover}} v}$$

$$\frac{r \Downarrow^{\mathsf{recover}} r_1 \quad \mathsf{not\_value}(r_1)}{\mathbf{LETR}\,(r) \Downarrow^{\mathsf{recover}} \mathbf{LETR}\,(r_1)}$$

$$\frac{r_1 \Downarrow^{\mathsf{recover}} r_1' \quad r_2 \Downarrow^{\mathsf{recover}} r_2'}{\mathbf{PTUPLE}\,(r_1,r_2) \Downarrow^{\mathsf{recover}} r_1' \# r_2'}$$

Fig. 4. Rules for big-step recovery evaluation, $r \Downarrow^{\mathsf{recover}} r'$.

fail, or one of the branches fails. The operator $\#$ combining the results is defined as follows:

$$r_1 \# r_2 = \begin{cases} (v_1, v_2) & \text{if } r_1{=}v_1 \wedge r_2{=}v_2 \\ \mathbf{PTUPLE}(r_1, r_2) & \text{otherwise} \end{cases}$$

If both branches succeed, we get the value (v_1, v_2). Otherwise, we get a $\mathbf{PTUPLE}(r_1, r_2)$ which records r_1 and r_2 to be used in the recovery process. This will allow recovery, for example, to re-evaluate only r_1 if r_2 were successful. This treatment may be considered as a refinement over a naive semantics that just propagates $\mathbf{BOT}$ to the top-level. However, we stress that the whole point of our approach is to be able to reuse correctly executed sub-computations during recovery. Re-executing the entire computation from the beginning is not only wasteful, but also has high probability for failure.

Figure 3 shows an example of a faulty evaluation of the expression $(\!|1+1,3|\!)$, where the second branch of the parallel pair fails. Note how the failure is recorded in the result.

C. Recovery Evaluation

If the program execution returns a non-value result, we run the recovery process (see Fig. 4). This takes the recorded path as input and attempts to re-evaluate the failed expression. To reflect typical computer behavior, recovery of a failed computation can also fail, just as execution of original computations can fail. Recovery operations are run on the same machines, so we should assume the possibility of repeated failure. That is why in the premise of the $\mathbf{BOT}(e)$ recovery rule, fault-prone evaluation as opposed to fault-free evaluation is used. Fig. 5 illustrates a successful recovery of the failed result produced by Fig. 3. The final result $(2, 3)$ is the expected output from a fault-free evaluation of $(\!|1+1,3|\!)$.

$$\frac{\dfrac{}{2 \Downarrow^{\mathsf{recover}} 2} \qquad \dfrac{\dfrac{}{3 \Downarrow^{\mathsf{fp}} 3}}{\mathbf{BOT}(3) \Downarrow^{\mathsf{recover}} 3}}{\mathbf{PTUPLE}(2, \mathbf{BOT}(3)) \Downarrow^{\mathsf{recover}} (2,\ 3)}$$

Fig. 5. Example recovery

D. Correctness and Progress of Recovery

First, we prove that fault-prone evaluation together with recovery is sound and complete with respect to the fault-free evaluation. Formally, we prove the following theorems, where $(\Downarrow^{\mathsf{recover}})^*$ is the reflexive-transitive closure of $\Downarrow^{\mathsf{recover}}$.

Theorem 3 (Soundness of Recovery)**.** If $e \Downarrow^{\mathsf{fp}} r$ and $r(\Downarrow^{\mathsf{recover}})^* v$ then $e \Downarrow v$.

Theorem 4 (Completeness of Recovery)**.** If $e \Downarrow v$ then $e \Downarrow^{\mathsf{fp}} v$ and $\mathbf{BOT}(e) \Downarrow^{\mathsf{recover}} v$.

Formal proofs of these theorems (as well as all other results mentioned in this paper) can be found in our Coq formalization. Completeness is quite easy as fault-prone evaluations almost syntactically include fault-free evaluations. Soundness, however, is somewhat trickier and relies on the insights that (*i*) recovery evaluation is transitive, (*ii*) $\mathbf{BOT}(e) \Downarrow^{\mathsf{fp}} r \iff e \Downarrow^{\mathsf{fp}} r$, and (*iii*) $e \Downarrow^{\mathsf{fp}} v \iff e \Downarrow v$.

Besides soundness and completeness, we are interested in proving some kind of progress for recovery evaluations. For this purpose we define the preorder $r' \succeq r$ stating that r' is "more advanced" than r.

Definition 1 (Result comparison)**.** Let $r' \succeq r$ be the least fixed point of the following equations.

- $r \succeq \mathbf{BOT}(e)$
- $v \succeq r$
- if $r_1' \succeq r_1$ and $r_2' \succeq r_2$, then $\mathbf{PTUPLE}(r_1', r_2') \succeq \mathbf{PTUPLE}(r_1, r_2)$
- if $r' \succeq r$, then $\mathbf{LETL}(r', x, e) \succeq \mathbf{LETL}(r, x, e)$,
- $\mathbf{LETR}(r') \succeq \mathbf{LETL}(r, x, e)$
- if $r' \succeq r$, then $\mathbf{LETR}(r') \succeq \mathbf{LETR}(r)$

It is easy to show that $\succeq$ is a preorder (i.e., it is reflexive and transitive). Sadly, however, it is not antisymmetric, the reason being that $\mathbf{BOT}(e) \succeq \mathbf{BOT}(e')$ for arbitrary e and e'. The best we can show is the following pseudo-antisymmetry property:

Lemma 5 (Pseudo-antisymmetry)**.** If $r \succeq r'$ and $r' \succeq r$, then $r \approx r'$, where $\approx$ is defined as the least fixed point of the following equations.

- $\mathbf{BOT}(e) \approx \mathbf{BOT}(e')$
- $v \approx v$
- if $r_1' \approx r_1$ and $r_2' \approx r_2$, then $\mathbf{PTUPLE}(r_1', r_2') \approx \mathbf{PTUPLE}(r_1, r_2)$
- if $r' \approx r$, then $\mathbf{LETL}(r', x, e) \approx \mathbf{LETL}(r, x, e)$.
- if $r' \approx r$, then $\mathbf{LETR}(r') \approx \mathbf{LETR}(r)$.

We can show that every recovery step makes 'progress' in that it moves to more advanced states up to our preorder.

Theorem 6 (Progress)**.** If $r \Downarrow^{\mathsf{recover}} r'$ then $r' \succeq r$.

$$\frac{e_1 \rightsquigarrow e_2}{\mathbf{RUN}\ e_1, s, e', s' \rightsquigarrow^1 \mathbf{RUN}\ e_2, s, e', s'} \qquad \frac{}{\mathbf{START}\ e, s \rightsquigarrow^1 \mathbf{RUN}\ e, s, e, s} \qquad \frac{\mathsf{exp\_not\_value}(e)}{\mathbf{RUN}\ C[e], s, e', s' \rightsquigarrow^1 \mathbf{RUN}\ e, \mathbf{Cons}\ C\ s, e', s'}$$

$$\frac{}{\mathbf{RUN}\ v, \mathbf{Cons}\ C\ s, e', s' \rightsquigarrow^1 \mathbf{RUN}\ C[v], s, e', s'} \qquad \frac{}{\mathbf{RUN}\ e, s, e', s' \rightsquigarrow^1 \mathbf{FAILED}\ e', s'}$$

Fig. 6. Rules for single processor evaluation, $state \rightsquigarrow^1 state'$.

$$\frac{pm[pid_1] = (\mathbf{RUN}\ (\!|e_1, e_2|\!), s, e', s', d) \qquad \mathbf{fresh}\ id\ \mathbf{in}\ rm}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (\mathbf{START}\ e_1, \mathbf{Left}\ id, \langle id : e_2\rangle \cdot d)], rm[id := \mathbf{Neither}\ s]}\ \text{FORK}$$

$$\frac{pm[pid_1] = (\mathbf{FAILED}\ e, s, d) \qquad pm[pid_2] = (\mathbf{IDLE}, d')}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (\mathbf{RECOVERED}, d)][pid_2 := (\mathbf{START}\ e, s, d')], rm}\ \text{RECOVER}$$

$$\frac{pm[pid_1] = (pc_1, d_1) \qquad pm[pid_2] = (pc_2, d_2 \cdot t) \qquad pid_1 \neq pid_2}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (pc_1, t \cdot d_1)][pid_2 := (pc_2, d_2)], rm}\ \text{STEAL}$$

$$\frac{pc_1 \rightsquigarrow^1 pc_2 \qquad pm[pid_1] = (pc_1, d)}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (pc_2, d)], rm}\ \text{LOCAL}$$

$$\frac{pm[pid_1] = (\mathbf{IDLE}, \langle id : e\rangle \cdot d)}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (\mathbf{START}\ e, \mathbf{Right}\ id, d)], rm}\ \text{POP\_TASK}$$

$$\frac{pm[pid_1] = (\mathbf{RUN}\ v, \mathbf{Left}\ id, e', s', d) \qquad rm[id] = (\mathbf{Neither}\ s)}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (\mathbf{IDLE}, d)], rm[id := \mathbf{Left}\ v\ s]}\ \text{LEFT\_FIRST}$$

$$\frac{pm[pid_1] = (\mathbf{RUN}\ v, \mathbf{Left}\ id, e', s', d) \qquad rm[id] = (\mathbf{Right}\ v_2\ s)}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (\mathbf{START}\ (v, v_2), s, d)], rm[id := \mathbf{Finished}\ v_3]}\ \text{LEFT\_LAST}$$

$$\frac{pm[pid_1] = (\mathbf{RUN}\ v, \mathbf{Right}\ id, e', s', d) \qquad rm[id] = (\mathbf{Neither}\ s)}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (\mathbf{IDLE}, d)], rm[id := \mathbf{Right}\ v\ s]}\ \text{RIGHT\_FIRST}$$

$$\frac{pm[pid_1] = (\mathbf{RUN}\ v, \mathbf{Right}\ id, e', s', d) \qquad rm[id] = (\mathbf{Left}\ v_1\ s)}{pm, rm \rightsquigarrow^{\mathsf{m}} pm[pid_1 := (\mathbf{START}\ (v_1, v), s, d)], rm[id := \mathbf{Finished}\ v_3]}\ \text{RIGHT\_LAST}$$

Fig. 7. Rules for multiprocessor evaluation, $pm, rm \rightsquigarrow^{\mathsf{m}} pm', rm'$.

IV. Low-Level Semantics

In the low-level semantics, we model the processors executing our program explicitly, together with the usual data structures for distributing parallel tasks to them. In essence, each processor has a queue, where it adds any parallel tasks it creates, and removes them one by one to execute them. At any time (typically when it is idle), a processor can also try to steal a task from the queue of a different processor. This approach, known as work stealing [6], dynamically balances the work among processors, leading to very efficient implementations [7], [8]. In addition to work stealing, our semantics models failures by allowing individual processors to fail, and correctly running ones to recover (rerun) the computation that a failed processor was executing.

A. Configurations

System-wide configurations consist of a processor map, pm, and a result map, rm. The processor map maps processor identifiers to a processor state, $state$, and a deque, d, of tasks to be executed. A processor can be in one of following five different states:

$$state ::= \mathbf{IDLE} \mid \mathbf{START}\ e, s \mid \mathbf{RUN}\ e_1, s_1, e_2, s_2 \mid \mathbf{FAILED}\ e, s \mid \mathbf{RECOVERED}$$

The first state represents the case, when the processor has finished executing any tasks it started and can start another task either by removing one from its deque or by stealing one from another processor's deque. Next is the **START** state, where a processor has selected a task to execute but has not yet started executing it. Here, we store the expression, e, to be evaluated and the corresponding stack, s. The stack is list of contexts ended by a marker identifying the task being executed:

$$s ::= \mathbf{Left}\ id \mid \mathbf{Right}\ id \mid \mathbf{Cons}\ C\ s$$

The **RUN** state represents the case when a task is being executed. Here, the first expression-stack pair (e_1, s_1) is used to perform normal computations, while the second expression-stack pair (e_2, s_2) remains constant throughout execution. We assume that the latter pair is stored in some safe storage that is kept intact in cases of failures. We have separate **START** and **RUN** states to make explicit the step that stores the current expression and stack to a safe storage. Next, is the failed state, which simply drops the first expression-stack component of the running state, and records only the second tuple which is supposed to survive failures. Finally, in order to prevent multiple recoveries of the same failed state, we use **RECOVERED** state to mark processors whose failures have been recovered.

The deque, d, is a (usually optimized) doubly ended queue of tasks, $\langle id : e \rangle$, created by the FORK rule when evaluating a parallel tuple. More specifically, evaluation of a parallel tuple creates a new task for the right branch, pushes it to the deque, and then proceeds directly to execute the left branch. Tasks pushed to the deque are later removed either by the same processor (POP_TASK), when it becomes idle, or at any point by other processors (STEAL). Tasks popped by the same processor are removed from the same end of the deque as they are added, whereas stolen tasks are removed from the other end. Similar to the recorded expression-stack pairs, we assume that deques are stored in a safe storage that is unaffected by failures.

The result map is used to keep track of results of forked parallel tuple computations. It maps fork identifiers to one of the following four possibilities, depending on whether the left and/or the right branches of the fork has finished their computation:

$$res ::= \mathbf{Neither}\ s \mid \mathbf{Left}\ v\ s \mid \mathbf{Right}\ v\ s \mid \mathbf{Finished}\ v$$

If neither of the branches have finished, $rm[id] = \mathbf{Neither}\ s$, where s stores the continuation stack: the computation to be executed once both branches of the parallel tuple finish. If one of the branches has finished, we record its value and the continuation stack. If, however, both branches have finished, we no longer need to store the continuation stack, as a new task performing that work will have been started.

B. The Operational Semantics in Detail

Our operational semantics consists of two relations, $\rightsquigarrow^1$ and $\rightsquigarrow^{\mathsf{m}}$. The former (in Fig. 6) describes execution steps that are local to single processor, whereas the latter (in Fig. 7) defines executions of the whole system.

The single processor evaluation semantics (see Fig. 6) is comprised of five rules. The first takes a small step in the evaluation of current expression in the **RUN** state. The second moves from the **START** state to the **RUN** state by committing e and s to the safe storage. The next two rules push and pop contexts to and from the local stack. The last rule describes failures, taking the processor from **RUN** to **FAILED** state, where it only keeps the fields in the safe storage (i.e., e', s').

Multiprocessor execution (see Fig. 7) consists of nine reduction rules. Whenever the current expression is a parallel tuple, FORK rule applies. This rule assigns first element of parallel tuple as current expression with a **START** label, and pushes second element to deque for further to be executed. It also reserves a key in the result map in order to refer that for recording values coming out of the branches of execution and also getting back to execution with a continuation stack. RECOVER rule applies when there exists a failed processor and an idle processor. Idle processor recovers both lastly executed expression on the failed processor together with its deque. In STEAL rule, topmost task in a deque of one processor is stolen by another processor (i.e. pushed to the deque of latter from bottom). LOCAL rule represents the independent executions of different processors. In other words, if a processor takes a step then it is applied globally with this rule. POP_TASK rule, as the name suggests pops a task from deque and it applies only when a processor is in **IDLE** state. Following four rules record results from a successful partial evaluation of branches which are created by a fork operation earlier. If left branch finishes its execution first, LEFT_FIRST rule applies. As we can see in its premise, we require that we do not have the result of right branch (i.e. we have a record with "Neither" label in the result map indicating that none of the branches has submitted its result yet). After applying the rule, the result map stores the value of left branch keeping the previous continuation stack. In the LEFT_LAST case, the result map already has the value of right branch. Therefore after applying the rule, it stores the value pair consisting of the values coming out of both branches. The continuation stack is also moved from the result map in order for the context to be used afterwards. Finally, RIGHT_FIRST and RIGHT_LAST are symmetric to the previous two.

C. Example Evaluations

An example for fault-free multiprocessor execution in shown in Fig. 8, where we evaluate the expression $(\!|1 + 1, 3|\!)$ returning $(2, 3)$. In step 2 current expression for the first processor goes from **START** to **RUN** state. Step 3 is a fork where we assign id_1 as fork identifier and record the current stack **Right** id_0 in the result map as continuation. Step 4 moves to **RUN** state, but this time for the expression $(1 + 1)$. In step 5 the expression $(1+1)$ is evaluated to 2. In step 6, the second processor steals right branch from the first processor and then evaluates it. We submit the result of right branch first and then the left branch. When both submitted their results, we have $(2, 3)$ as the result of the fork operation. Since left is submitted later by the first processor, it gets the stack **Right** id_0 from the result map as its continuation. After going to **RUN** state once more in step 11, we submit the result of the whole computation in step 12. If we store anything in the result map for the identifier of initial expression (id_0) and if the branches match (**Right**), then we get the result of overall computation. Therefore after applying enough number of steps of $\rightsquigarrow^{\mathsf{m}}$, we get $(2, 3)$ as the result of evaluating $(\!|1 + 1, 3|\!)$.

In our failure and recovery example (see Fig. 9), the first seven steps are exactly the same as in the previous example. We assume that the second processor fails at step 8. After that, since execution of left branch is already completed by the first processor, we record the value 2 as the result of left branch. Then, the first processor becomes idle. Therefore, it can recover the failure of the second processor. It gets the current task on which the failed processor was previously working, that is the right branch of the fork id id_1. The rest of the steps are evaluating the right branch and submitting the result to the result map. At the very end of this failure and recovery execution, we still get the same result. This is an example of the correspondence between the fault free evaluation and fault-prone evaluation with recovery actions in our low-level multiprocessor computation.

V. Proofs of Correspondence

In this section, we define well-formedness of a computation and prove that it is preserved by every multiprocessor step. Our definitions are purposely in an informal style for conciseness: the formal definitions can be found in our Coq development.

#	**Processor 1** ($pm[pid_1]$)	**Processor 2** ($pm[pid_2]$)	**Result Map** (rm)	e **s.t.** $\langle pm, rm\rangle \sim_{id_0} e$
1	START $(\!\lvert 1+1, 3 \rvert\!)$, Right id_0, []	IDLE, []	id_0:Neither s	$(\!\lvert 1+1, 3 \rvert\!)$
2	RUN $(\!\lvert 1+1, 3 \rvert\!)$, Right id_0, [], $(\!\lvert 1+1, 3 \rvert\!)$, Right id_0	IDLE, []	id_0:Neither s	$(\!\lvert 1+1, 3 \rvert\!)$
3	START (1 + 1), Left id_1, [Right id_1 3]	IDLE, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 1+1, 3 \rvert\!)$
4	RUN (1 + 1), Left id_1, [Right id_1 3], (1 + 1), Left id_1	IDLE, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 1+1, 3 \rvert\!)$
5	RUN 2, Left id_1, [Right id_1 3], (1 + 1), Left id_1	IDLE, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
6	RUN 2, Left id_1, [], (1 + 1), Left id_1	IDLE, [Right id_1 3]	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
7	RUN 2, Left id_1, [], (1 + 1), Left id_1	START 3, Right id_1, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
8	RUN 2, Left id_1, [], (1 + 1), Left id_1	RUN 3, Right id_1, [], 3, Right id_1	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
9	RUN 2, Left id_1, [], (1 + 1), Left id_1	IDLE, []	id_0:Neither s, id_1:Right 3 (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
10	START (2, 3), Right id_0, []	IDLE, []	id_0:Neither s, id_1:Finished (2, 3)	$(\!\lvert 2, 3 \rvert\!)$
11	RUN (2, 3), Right id_0, [], (2, 3), (Right id_0)	IDLE, []	id_0:Neither s, id_1:Finished (2, 3)	(2,3)
12	IDLE, []	IDLE, []	id_0:Right (2, 3) s, id_1:Finished (2, 3)	(2,3)

Fig. 8. Example showing a fault-free execution of the low-level semantics with two processors.

#	**Processor 1** ($pm[pid_1]$)	**Processor 2** ($pm[pid_2]$)	**Result Map** (rm)	e **s.t.** $\langle pm, rm\rangle \sim_{id_0} e$
1	START $(\!\lvert 1+1, 3 \rvert\!)$, Right id_0, []	IDLE, []	id_0:Neither s	$(\!\lvert 1+1, 3 \rvert\!)$
2	RUN $(\!\lvert 1+1, 3 \rvert\!)$, Right id_0, [], $(\!\lvert 1+1, 3 \rvert\!)$, Right id_0	IDLE, []	id_0:Neither s	$(\!\lvert 1+1, 3 \rvert\!)$
3	START (1 + 1), Left id_1, [Right id_1 3]	IDLE, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 1+1, 3 \rvert\!)$
4	RUN (1 + 1), Left id_1, [Right id_1 3], (1 + 1), Left id_1	IDLE, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 1+1, 3 \rvert\!)$
5	RUN 2, Left id_1, [Right id_1 3], (1 + 1), Left id_1	IDLE, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
6	RUN 2, Left id_1, [], (1 + 1), Left id_1	IDLE, [Right id_1 3]	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
7	RUN 2, Left id_1, [], (1 + 1), Left id_1	START 3, Right id_1, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
8	RUN 2, Left id_1, [], (1 + 1), Left id_1	RUN 3, Right id_1, [], 3, Right id_1	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
9	RUN 2, Left id_1, [], (1 + 1), Left id_1	FAILED 3, Right id_1, []	id_0:Neither s, id_1:Neither (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
10	IDLE, []	FAILED 3, Right id_1, []	id_0:Neither s, id_1:Left 2 (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
11	START 3, Right id_1, []	RECOVERED, []	id_0:Neither s, id_1:Left 2 (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
12	RUN 3, Right id_1, 3, Right id_1, []	RECOVERED, []	id_0:Neither s, id_1:Left 2 (Right id_0)	$(\!\lvert 2, 3 \rvert\!)$
13	START (2, 3), Right id_0, []	RECOVERED, []	id_0:Neither s, id_1:Finished (2, 3)	$(\!\lvert 2, 3 \rvert\!)$
14	RUN (2, 3), Right id_0, (2, 3), Right id_0, []	RECOVERED, []	id_0:Neither s, id_1:Finished (2, 3)	(2,3)
15	IDLE, []	RECOVERED, []	id_0:Right (2, 3) s, id_1:Finished (2, 3)	(2,3)

Fig. 9. Example showing a low-level execution where processor 1 fails and is recovered by processor 2.

In order to define our well-formedness condition, we need two auxiliary definitions: $apply(s, e)$, that applies the contexts in the stack s to the expression e, and $last(s)$, which bypasses all the contexts and returns the top-most entry in the stack. These two functions are recursively defined as follows:

$$apply(s, e) \stackrel{\text{def}}{=} \begin{cases} apply(s', C[e]) & \text{if } s = \mathbf{Cons}\ C\ s' \\ e & \text{otherwise} \end{cases}$$

$$last(s) \stackrel{\text{def}}{=} \begin{cases} last(s') & \text{if } s = \mathbf{Cons}\ C\ s' \\ s & \text{otherwise} \end{cases}$$

Definition 2 (Well-Formed Configurations)**.** A system-wide configuration (pm, rm) is well formed if all the following conditions hold:

1) Tasks stored in all of the deques together with the running tasks are pairwise unique;
2) Running tasks are not marked as finished in rm; and
3) Whenever $pm[pid] = \mathbf{RUN}\ e, s, e', s', d$, we have (*i*) $apply(s', e') \rightsquigarrow^* apply(s, e)$ and (*ii*) $last(s) = last(s')$. In other words, e and s should be a partially evaluated version of the computation represented by e' and s'.

Lemma 7 (Well-Formedness Preservation)**.** If (pm, rm) is well formed and pm, $rm \rightsquigarrow^{\mathsf{m}\star} pm'$, rm' then (pm', rm') is also well formed.

Our main soundness theorem states whenever a low-level execution returns a value, then there is a high-level fault-free execution returning the same value. Since fault-free high-level big-step executions are deterministic, this means that the low-level executions, if they terminate by returning a value, will always return the 'right' value. The formal statement is as

follows:

Theorem 8 (Soundness)**.**
If $pm = \mathsf{empty}[pid := (\mathbf{START}\, e, \mathbf{Right}\, d, [])]$ and $rm = \mathsf{empty}[id := (\mathbf{Neither}\, (\mathbf{Left}\, id))]$ and $pm, rm \rightsquigarrow^{\mathsf{m}*} pm', rm'$ and $rm'[id] = \mathbf{Right}\, v\, s$, then $e \Downarrow v$.

We prove Theorem 8 by constructing a forward simulation [9]. We define the relation as $\langle pm, rm\rangle \sim_{id} e$ that relates a configuration (pm, rm) and a fork identifier id to the expression e corresponding to the current partially evaluated form of the parallel pair identified by id. For brevity, we omit the formal definition, but show the relation for the example executions in Fig. 8 and 9. A basic property of our simulation relation is that it is deterministic.

Lemma 9. If $\langle pm, rm\rangle \sim_{id} e_1$ and $\langle pm, rm\rangle \sim_{id} e_2$ then $e_1 = e_2$.

Further, we can show that it is indeed a simulation, namely that it is preserved by reduction.

Lemma 10 (Simulation)**.** If $\langle pm, rm\rangle \sim_{id} e$ and $pm,\ rm \rightsquigarrow^{\mathsf{m}} pm',\ rm'$ then there exists an e' such that $e \rightsquigarrow^{*} e'$ and $\langle pm, rm\rangle \sim_{id} e'$.

Having proved these lemmas, we can now prove Theorem 8 by an induction on the length of $\rightsquigarrow^{\mathsf{m}*}$ and appealing to Lemmas 7, 9 and 10.

We also prove a completeness theorem stating that whenever a high-level computation returns a value, it is possible for the low-level computation to return the same value. Given the soundness theorem above (Theorem 8) and the determinism of the high-level fault-free big-step semantics (Theorem 1), this theorem essentially means that low-level computations never get stuck unless the corresponding high-level computations do.

Theorem 11 (Completeness)**.**
If $e \Downarrow v$ and $pm = \mathsf{empty}[pid := (\mathbf{START}\, e, \mathbf{Right}\, d, [])]$ and $rm = \mathsf{empty}[id := (\mathbf{Neither}\, (\mathbf{Left}\, id))]$ then there exists pm' and rm' such that $pm, rm \rightsquigarrow^{\mathsf{m}*} pm', rm'$ and $rm'[id] = \mathbf{Right}\, v\, s$.

To prove Theorem 11, we only need to consider a non-failing execution with a single processor. By applying Theorem 2, which relates fault-free small-step and big-step executions, and our assumption, we know that $e \rightsquigarrow^{*} v$. By induction on the length of the $\rightsquigarrow^{*}$ execution, we can construct a corresponding low-level execution.

VI. Related Work

This paper brings together the checkpointing approach for tolerating failures of distributed computations, and the work stealing approach for scheduling fork-join parallel computations. Both of these topics have been well studied in isolation, but to the best of our knowledge they have not been considered together before.

There are many works on the practice of the checkpointing approach to deal with failures in distributed systems, some of which come with informal proofs of correctness for the proposed implementations (e.g., [10], [11]). A survey can be found in Elnozahy et al. [3]. In general, these works deal with the more complex case where we want to protect an arbitrary computation running on a distributed system against node failures. In such computations, the various nodes of the distributed system typically communicate by exchanging messages, making the node computations highly interdependent. While doing a rollback from a failure these dependencies create the so-called 'domino effect' [12]. Cao et al. [13] uses the notion of dependency graph for checkpointing in order to resolve this problem in propagating the rollback actions. Koo et al. [10] also proposed an algorithm dealing with the dependency issues among processors. In our model of computation, however, the computation is purely functional and divided into independent tasks that can be executed on any processor. One important property of our task representation is that there is no dependency between any two tasks in terms of recovery. Therefore it is sufficient to keep local checkpoints for each task and there is no need to propagate the recovery actions. Recovering a failed computation of a task is enough to get back to consistent state of the system.

When we consider scheduling parallel computations and load balancing, work stealing algorithms schedule fork-join style parallel computations within a near-optimal theoretical bound [6], [14] and have been shown to be very efficient in practice [7], [8]. Because of this, we decided to take work-stealing algorithm as the base for our fault free evaluation, we also modified the steal operation to recover failed processors.

VII. Conclusion

In recent years, parallel computations are run on thousands of processors, all of which are vulnerable to faults. Designing good fault detection and recovery mechanisms is therefore of great importance to people relying on such massively parallel computations. In this paper, we made the first small step in that direction, by approaching the problem from a programming language perspective.

We used a purely functional language that includes parallel pairs in its syntax for representing fork-join style parallel computations. As evaluation schemes of this language, we designed both high- and low-level semantics, which we illustrated using examples. Finally, we proved correspondence properties relating the high- and low-level semantics. The lemmas and theorems we state in this paper were proved using the Coq interactive theorem prover [4], thereby giving us full confidence for their correctness. We also used Ott [15] in order to more conveniently write and typeset the semantics and then generate the corresponding Coq definitions.

The goal of this work is not efficiency but correctness. Therefore, for simplicity, we save every parallel task generated by evaluation in safe storage in order to be able to recover from a possible failure. The granularity at which tasks should be checkpointed can, however, in principle be adjusted allowing us to trade off the cost of frequently saving information against the larger recovery costs. Figuring out a good such trade-off is left for future work.

Acknowledgements

We would like to thank Umut Acar for great discussions that initiated our interest in fault-tolerant parallelism and led to this paper. The research was supported by the EU FET FP7 project ADVENT.

REFERENCES

[1] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd SOSP*. ACM, 2009, pp. 277–290.

[2] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *TOPLAS*, vol. 4, no. 3, pp. 382–401, 1982.

[3] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[4] Coq development team, "The Coq proof assistant," http://coq.inria.fr/.

[5] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993, pp. 237–247.

[6] F. Burton and M. Sleep, "Executing functional programs on a virtual tree of processors," in *Proceedings of the 1981 conference on Functional programming languages and computer architecture*. ACM, 1981, pp. 187–194.

[7] U. A. Acar, A. Charguéraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2013, pp. 219–228.

[8] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005, pp. 21–28.

[9] N. Lynch and F. Vaandrager, "Forward and backward simulations – part I: Untimed systems," *Information and Computation*, vol. 121(2), pp. 214–233, September 1995.

[10] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, no. 1, pp. 23–31, 1987.

[11] M. Kasbekar, C. Narayanan, and C. R. Das, "Selective checkpointing and rollbacks in multi-threaded object-oriented environment," *IEEE Transactions on Reliability*, vol. 48, no. 4, pp. 325–337, 1999.

[12] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, no. 2, pp. 220–232, 1975.

[13] J. Cao and K. Wang, "An abstract model of rollback recovery control in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 26, no. 4, pp. 62–76, 1992.

[14] R. Halstead Jr, "Implementation of multilisp: Lisp on a multiprocessor," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM, 1984, pp. 9–17.

[15] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša, "Ott: Effective tool support for the working semanticist," in *ICFP '07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, Oct. 2007, pp. 1–12.