Dag-Calculus:
A Calculus for Parallel Computation
Technical Appendix

This appendix consists of two main parts: in the first, we prove correct the translations of parallel primitives into the dag calculus, in the second, we prove that the scheduling algorithm implements the rules of the dag calculus. Concretely, Appendix A provides auxiliary definitions and properties for the translations; Appendices B, C and D provide the proofs of translations of fork-join, async-finish and futures, respectively; Appendix E provides the proof of the implementation of dag calculus.

A Auxiliary definitions

Here we recall the syntax and semantics of the dag calculus, which we denote in short dag calculus. These constructs and rules are described in detail in Section 3 of the paper.

Note that while the syntax of the calculus is identical to the one presented in the paper, the formulation of the reduction rules is slightly different: the dag-modification constructs are assumed to always appear in the context of a let-binding or sequential composition, and the reduction reduces them together with the binding/composition. This approach is clearly equivalent to the presentation in the paper, but it significantly reduces the number of otherwise not interesting intermediate steps that would be possible to encounter in the translations, thus shortening the translations and the proofs significantly.

Definition 1. A *computation dag* is a pair (V, E), where the *vertex-map* V is a finite map from threads to a pair consisting of expressions, and status and *edges* E is a edges

$$
\begin{aligned}
e ::= {} & v \mid e \oplus e \mid e \otimes e \mid (e, e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \\
& e\ e \mid \texttt{alloc()} \mid e := e \mid !e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \\
& \texttt{newTd}\ e \mid \texttt{release}\ e \mid \texttt{newEdge}\ (e, e) \mid \texttt{self()} \mid \texttt{yield()} \\
v ::= {} & x \mid \ell \mid \texttt{t} \mid \texttt{n} \mid \texttt{()} \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
K ::= {} & \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e \mid K := e \mid v := K \mid !K \mid \texttt{release}\ K \\
& \mid \texttt{newEdge}\ (K, e) \mid \texttt{newEdge}\ (v, K) \mid \ldots
\end{aligned}
$$

Figure 1: Abstract syntax for DAG-Calculus.

$$\frac{}{\sigma, \texttt{fst}\ (v_1, v_2) \to v_1, \sigma}\ \textsc{Fst} \qquad \frac{}{\sigma, \texttt{snd}\ (v_1, v_2) \to v_2, \sigma}\ \textsc{Snd} \qquad \frac{l \notin \mathrm{dom}(\sigma)}{\sigma, \texttt{alloc} \to l, \sigma[l \mapsto ()]}\ \textsc{Alloc}$$

$$\frac{\sigma(l) = v}{\sigma, !\, l \to v, \sigma}\ \textsc{Deref} \qquad \frac{l \in \mathrm{dom}(\sigma)}{\sigma,\ (l := v) \to (), \sigma[l \mapsto v]}\ \textsc{Assign}$$

$$\frac{}{\sigma, ((\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end})\ v) \to e[f \mapsto \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}][x \mapsto v], \sigma}\ \textsc{Apply}$$

$$\frac{\sigma, e \to e', \sigma'}{\sigma, K[e] \to K[e'], \sigma'}\ \textsc{Context}$$

$$\frac{V(t) = (e_1, \mathsf{X}) \qquad \sigma_1, e_1 \to e_2, \sigma_2}{V, E, \sigma_1 \twoheadrightarrow V[t \mapsto (e_2, \mathsf{X})], E, \sigma_2}\ \textsc{Step} \qquad \frac{V(t) = (v, \mathsf{X}) \qquad E' = E \setminus \{(t, t') \mid t' \in \mathrm{dom}(V)\}}{V, E, \sigma \twoheadrightarrow V[t \mapsto ((), \mathsf{F})], E', \sigma}\ \textsc{Stop}$$

$$\frac{V(t) = (K[\texttt{let}\ x = \texttt{newTd}\ e\ \texttt{in}\ e'], \mathsf{X}) \qquad t'\ \ \mathsf{fresh}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[e'[x \mapsto t']], \mathsf{X})][t' \mapsto (e, \mathsf{N})], E, \sigma}\ \textsc{NewTd} \qquad \frac{V(t) = (e, \mathsf{R}) \qquad \{t' \mid (t', t) \in E\} = \emptyset}{V, E, \sigma \twoheadrightarrow V[t \mapsto (e, \mathsf{X})], E, \sigma}\ \textsc{Start}$$

$$\frac{V(t) = (K[\texttt{release}\ t'; e], \mathsf{X}) \qquad V(t') = (e', \mathsf{N})}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[e], \mathsf{X})][t' \mapsto (e', \mathsf{R})], E, \sigma}\ \textsc{Release}$$

$$\frac{\begin{array}{c}V(t) = (K[\texttt{newEdge}\ t_1\ t_2; e], \mathsf{X}) \qquad t_1, t_2 \in \mathrm{dom}(V) \qquad (\mathrm{status}(V(t_2)) \in \{\mathsf{N}, \mathsf{R}\} \vee t_2 = t) \\ E' = E \uplus (\text{if } \mathrm{status}(V(t_1)) = \mathsf{F} \text{ then } \emptyset \text{ else } \{(t_1, t_2)\}) \qquad E' \text{ cycle-free}\end{array}}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[e], \mathsf{X})], E', \sigma}\ \textsc{NewEdge}$$

$$\frac{V(t) = (K[\texttt{self}], \mathsf{X})}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[t], \mathsf{X})], E, \sigma}\ \textsc{Self} \qquad \frac{V(t) = (K[\texttt{yield}; e], \mathsf{X})}{V, E, \sigma \twoheadrightarrow V[t \mapsto (K[e], \mathsf{R})], E, \sigma}\ \textsc{Yield}$$

Figure 2: Dynamic semantics for dag calculus. N stands for thread status *new*, R for *released*, X for *executing* and F for *finished*. The operation status($V(t)$) denotes the status of thread t, i.e. the second component of $V(t)$.

(edges) between threads, more precisely:

$$\begin{array}{lcl} s & ::= & \mathsf{N} \mid \mathsf{R} \mid \mathsf{X} \mid \mathsf{F} \\ V & : & \mathcal{T} \rightharpoonup_{\mathit{fin}} (e \times s) \\ E & \subseteq & \mathrm{dom}(V) \times \mathrm{dom}(V). \end{array}$$

Definition 2. The operational semantics of dag calculusis a relation $\twoheadrightarrow$ between states made of a computation dag and a store mapping locations to values. It is given in Figure 2. We take a subset of the transitions Start and Stop as *scheduler* transitions, written $\twoheadrightarrow_{\mathrm{s}}$.

Lemma 1. *For any dag* (V, E) *and any state* σ*,* $V, E, \sigma \not\twoheadrightarrow_{\mathrm{s}}^{\infty}$.

Proof. Give weight to a dag by taking for each vertex weight according to its status:

$$w(s) = \begin{cases} 0 & \text{if } s = \mathsf{N} \text{ or } s = \mathsf{F} \\ 1 & \text{if } s = \mathsf{X} \\ 2 & \text{if } s = \mathsf{R} \end{cases}$$

Take the weight of the dag as the sum of the weights of its vertices, then proceed by induction on the weight of the dag. Observe that each scheduler transition decreases the weight of the node it operates on, and hence the whole dag, which ends the proof. □

The proofs of translations to the dag calculus presented in the following appendices involve one key difficulty, related to administrative reduction steps, i.e. to the fact that one reduction step in the source language may correpond to several reduction steps in the target language. Most compiler proofs deal with administrative reduction steps using well-known proofs techniques, typically based on simulation diagrams. However, we have found that these techniques were not directly applicable to the parallel semantics of a language such as that the fork-join language considered here.

When the target program takes a reduction step, this step corresponds either to an administrative step, or to a real step from the source program. Consider the latter case. With a sequential semantics, when the target program takes a real step, the target program then typically corresponds exactly to the translation of the source program. However, with a parallel semantics, this might not be the case. For example, since the two branches of a parallel pair may reduce independently, one branch may take a real step while the other branch is in the middle of performing a sequence of administrative steps. Although the target program takes a real step, it is not, at this point, the translation of any source program. Thus, we cannot easily close a simulation diagram.

To address this challenge, we introduce an *instrumented* programming language, similar to the source programming language, except that each parallel construct gets annotated with information about identities associated with its representation in the target language: number of administrative steps already performed, thread identifiers, locations for the results, etc. We then set up a two-layer simulation diagram: the first layer relates the source program with the instrumented program, while the second layer relates the instrumented program with the target program. We are able to reason about both layers independently, using conventional simulation diagrams, and then conclude by relating the source and the target programs.

B Correctness of translations of fork-join

B.1 Syntax and Operational Semantics

The syntax and semantics in this section are precisely the same as in the paper, and repeated here for convenience.

$$
\begin{aligned}
e &::= v \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \mid\ \texttt{forkjoin}\ (e, e) \\
v &::= x \mid n \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
K &::= \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e \mid \texttt{forkjoin}\ (K, e) \mid \texttt{forkjoin}\ (e, K)
\end{aligned}
$$

$$
\begin{aligned}
K[\texttt{fst}\ (v_1, v_2)] &\to K[v_1] \\
K[\texttt{snd}\ (v_1, v_2)] &\to K[v_2] \\
K[\texttt{let}\ x = v\ \texttt{in}\ e] &\to K[e[x \mapsto v]] \\
K[\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}\ v] &\to K[e[x \mapsto v, f \mapsto \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}]] \\
K[\texttt{forkjoin}\ (v_1, v_2)] &\to K[(v_1, v_2)]
\end{aligned}
$$

B.2 Instrumented Syntax and Operational Semantics

The general pattern of the instrumentation, both for fork-join parallelism and for the other constructs, is to annotate the parallel constructs, which need to be compiled in the translation to dag calculus, with additional information. The crucial part of these annotations is a number that describes how far the evaluation of the compiled code has progressed. Intuitively, 0 always corresponds to an expression, whose translated equivalent has not yet started execution, and each consecutive step of compiled execution takes the number one step higher. If in the evaluation process new information is obtained, such as identities of relevant threads, locations, etc., these are also included in the annotation (cf. the grammar of a_p, the annotations on parallel compositions, below)

Since in dag calculusthe program parts are stored in the vertices of the dag, parts of the program that correspond to separate vertices also need to be annotated. In case that values are passed through the store, like in the compiled version of fork-join, these also need to be present in the annotation, and some additional steps may be possible (cf. the grammar for a_t, the annotations of expressions that correspond to the dag calculusthreads, below). In the particular case of fork-join, 0 means that the expression has not yet had a thread allocated to it and released, $(1, l, t)$ — that the expression is running in thread t and will write its result to location l, and $(2, l)$ — that the expression has finished the evaluation, and the result *has been written* to l. The extra step required to perform the write action is why we increment the number in the annotation here.

$$
\begin{aligned}
e &::= v \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid v\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid (\texttt{forkjoin}\ (e^{a_t}, e^{a_t}))^{a_p} \\
v &::= x \mid n \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
L &::= \bullet \mid \texttt{let}\ x = L\ \texttt{in}\ e \\
K &::= \bullet \mid L^{a_t}[(\texttt{forkjoin}\ (K, e^{a_t}))^{a_p}] \mid L^{a_t}[(\texttt{forkjoin}\ (e^{a_t}, K))^{a_p}] \\
a_t &::= 0 \mid (1, l, t) \mid (2, l) \\
a_p &::= 0 \mid (1, l) \mid (2, l, l) \mid (3, l, l, t) \mid (n, l, l, t, t) \text{ where } 4 \le n \le 7 \\
&\quad \mid (8, l, l, t) \mid (n, l, l) \text{ where } 9 \le n \le 11
\end{aligned}
$$

A *surface* expression is one where all annotations have value 0. Note that these are isomorphic to the expressions of the source language. We define well-formed expressions as follows:

$$
\mathrm{wf}(e) \equiv \begin{cases}
a_1 = 0 \wedge a_2 = 0 \wedge \mathrm{surf}(e_1) \wedge \mathrm{surf}(e_2) \wedge \mathrm{surf}(L) & \text{if } e = L[\texttt{forkjoin}\ (e_1^{a_1}, e_2^{a_2})^{a}] \text{ and } \mathrm{num}(a) \le 7 \\
\mathrm{loc}(a_1) = l_1 \wedge a_2 = 0 \wedge \mathrm{wf}(e_1) \wedge \mathrm{surf}(e_2) \wedge \mathrm{surf}(L) & \text{if } e = L[\texttt{forkjoin}\ (e_1^{a_1}, e_2^{a_2})^{(8, l_1, l_2, t_2)}] \\
\mathrm{loc}(a_1) = l_1 \wedge \mathrm{loc}(a_2) = l_2 \wedge \mathrm{wf}(e_1) \wedge \mathrm{wf}(e_2) \wedge \mathrm{surf}(L) & \text{if } e = L[\texttt{forkjoin}\ (e_1^{a_1}, e_2^{a_2})^{(n, l_1, l_2)}] \text{ and } n \in \{9, 10\} \\
a_1 = (2, l_1) \wedge a_2 = (2, l_2) \wedge \mathrm{surf}(e_1) \wedge \mathrm{surf}(e_2) \wedge \mathrm{surf}(L) & \text{if } e = L[\texttt{forkjoin}\ (e_1^{a_1}, e_2^{a_2})^{(11, l_1, l_2)}] \\
\mathrm{surf}(e) & \text{otherwise}
\end{cases}
$$

We extend well-formedness to annotated expressions e^{a_t} by setting $\mathrm{wf}(e^a) \equiv a \neq 0$, and to parallel contexts, by setting $\mathrm{wf}(K) \equiv \forall e.\ \mathrm{wf}(e) \Rightarrow \mathrm{wf}(K[e])$.

Operational semantics is defined on annotated, well-formed expressions, e^{a_t}. First,

we define some of the transitions between the parallel composition annotations.

$$0 \rightsquigarrow_p (1, l_1) \text{ where } l_1 \text{ fresh} \qquad (1, l_1) \rightsquigarrow_p (2, l_1, l_2) \text{ where } l_2 \text{ fresh}$$
$$(2, l_1, l_2) \rightsquigarrow_p (3, l_1, l_2, t_1) \text{ where } t_1 \text{ fresh} \qquad (3, l_1, l_2, t_1) \rightsquigarrow_p (4, l_1, l_2, t_1, t_2) \text{ where } t_2 \text{ fresh}$$
$$(n, l_1, l_2, t_1, t_2) \rightsquigarrow_p (n+1, l_1, l_2, t_1, t_2) \text{ for } n \in \{4, 5, 6\} \qquad (9, l_1, l_2) \rightsquigarrow_p (10, l_1, l_2)$$

These transitions can be used to make simple administrative reductions on the parallel composition form, as seen in the operational semantics below. Since the transitions through the remaining three steps are not entirely local, they are defined directly in the rules.

$$\texttt{fst}\ (v_1, v_2) \bigcirc\!\!\!\rightarrow v_1 \qquad \texttt{snd}\ (v_1, v_2) \bigcirc\!\!\!\rightarrow v_2 \qquad \texttt{let}\ x = v\ \texttt{in}\ e \bigcirc\!\!\!\rightarrow e[x \mapsto v]$$
$$\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}\ v \bigcirc\!\!\!\rightarrow e[x \mapsto v, f \mapsto \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}]$$
$$\frac{a \rightsquigarrow_p a'}{(\texttt{forkjoin}\ (e_1^{a_1}, e_2^{a_2}))^{a} \bigcirc\!\!\!\rightarrow (\texttt{forkjoin}\ (e_1^{a_1}, e_2^{a_2}))^{a'}}$$
$$(\texttt{forkjoin}\ (e_1^{(0)}, e_2^{(0)}))^{(7,l_1,l_2,t_1,t_2)} \bigcirc\!\!\!\rightarrow (\texttt{forkjoin}\ (e_1^{(1,l_1,t_1)}, e_2^{(0)}))^{(8,l_1,l_2,t_2)}$$
$$(\texttt{forkjoin}\ (e_1^{a_1}, e_2^{(0)}))^{(8,l_1,l_2,t_2)} \bigcirc\!\!\!\rightarrow (\texttt{forkjoin}\ (e_1^{a_1}, e_2^{(1,l_2,t_2)}))^{(9,l_1,l_2)}$$
$$(\texttt{forkjoin}\ (v_1^{(2,l_1)}, v_2^{(2,l_2)}))^{(10,l_1,l_2)} \bigcirc\!\!\!\rightarrow (\texttt{forkjoin}\ (v_1^{(2,l_1)}, v_2^{(2,l_2)}))^{(11,l_1,l_2)}$$
$$(\texttt{forkjoin}\ (v_1^{(2,l_1)}, v_2^{(2,l_2)}))^{(11,l_1,l_2)} \bigcirc\!\!\!\rightarrow (v_1, v_2)$$
$$\frac{\mathrm{wf}(K) \qquad e \bigcirc\!\!\!\rightarrow e'}{K[(L[e])^{(1,l,t)}] \rightarrow K[(L[e'])^{(1,l,t)}]} \qquad \frac{\mathrm{wf}(K)}{K[v^{(1,l,t)}] \rightarrow K[v^{(2,l)}]}$$

B.3 Connecting the instrumented and source languages

We define a map $\lfloor - \rfloor : \mathcal{E}_I \to \mathcal{E}_O$ that removes annotations from the instrumented expressions. We use it to connect the instrumented language to the source language. Note that a map that adds an annotation 0 to parallel compositions is the right inverse of $\lfloor - \rfloor$. For *surface* term, it is also the left inverse.

Lemma 2 (Instr-Step)**.** *For any two expressions $e, e' \in \mathcal{E}_I$ and annotations $a, a' \in \mathcal{A}_t$, if $e^a \to_I e'^{a'}$, then either $\lfloor e \rfloor = \lfloor e' \rfloor$ or $\lfloor e \rfloor \to_O \lfloor e' \rfloor$.*

Proof. By cases on the reduction. If the reduction is one that changes the annotation on a parallel pair or a thread, the erasure is trivially preserved. In the other cases, we can simply pick the corresponding reduction rule to match the erasure of the right-hand side. □

Definition 3. An *administrative* reduction in the instrumented language is a reduction $e^a \to_I e'$ such that $\lfloor e \rfloor = \lfloor e' \rfloor$. We write $e^a \to_a e'^{a'}$ for administrative steps.

Lemma 3 (Admin-Fin)**.** *For well-formed annotated expressions e^a, there are no infinite sequence of administrative reductions, i.e., if* wf e^a *then $e^a \not\to_a^\infty$.*

Proof. Assign a weight to the expression based on the count of its annotations in evaluation positions. Take num(a) to be the natural number in the annotation. Then define

$$w(e) = \begin{cases} 15 - \mathrm{num}(a) - \mathrm{num}(a_1) - \mathrm{num}(a_2) + w(e_1) + w(e_2) & \text{if } e = (\texttt{forkjoin}\ ((e_1)^{a_1}, (e_2)^{a_2})) \\ w(e_1) & \text{if } e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ 0 & \text{otherwise} \end{cases}$$

and extend it to annotated threads as $w(e^a) \equiv 2 - \mathrm{num}(a) + w(e)$. Observe that all administrative reductions decrease the weight of the expression. Thus, by induction on the weight we can conclude that the lemma holds. □

Lemma 4 (Instrument)**.** *For any expression $e \in \mathcal{E}_I$, value $v \in \mathcal{V}_I$ annotation $a \in \mathcal{A}_t$ and location l, if $e^a \to_I^* v^{(2,l)}$ then $\lfloor e \rfloor \to_O^* \lfloor v \rfloor$. Moreover, if $e^a \to_I^\infty$, then $\lfloor e \rfloor \to_O^\infty$.*

Proof. First part, by induction on the reduction sequence and Lemma 2.

Similarly, for the second part we proceed by coinduction: since $e \to_I^\infty$, by Lemmas 3 and 2 there exists e' such that $e \to_I^* e'$, $\lfloor e \rfloor \to_O \lfloor e' \rfloor$ and $e' \to_I^\infty$. Hence, $\lfloor e' \rfloor \to_O^\infty$ and finally $\lfloor e \rfloor \to_O^\infty$. □

B.4 Connecting instrumented language to dag calculus

Translation of surface expressions into dag calculus First, let us define the translation for surface expressions. We define a map $[\![-]\!]^S : \mathcal{E}_I \to \mathcal{E}_d agcalculus$, as follows. Note that the map extends in a simple way to sequential evaluation contexts L that are surface contexts. We write the latter one $[\![-]\!]^L : \mathcal{L}_I \to \mathcal{K}_d agcalculus$.

$$\begin{aligned} [\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]^S &= \texttt{let}\ x = [\![e_1]\!]^S\ \texttt{in}\ [\![e_2]\!]^S \\ [\![\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}]\!]^S &= \texttt{fun}\ f\ x\ \texttt{is}\ [\![e]\!]^S\ \texttt{end} \\ [\![v_1\ v_2]\!]^S &= [\![v_1]\!]^S\ [\![v_2]\!]^S \end{aligned}$$

For the parallel pair, we have

$[\![(\texttt{forkjoin}\ (e_1^{(0)}, e_2^{(0)}))^0]\!]^S$ =

```
  let l1 = alloc
      l2 = alloc
      t1 = newTd (l1 := [[e1]]^S)
      t2 = newTd (l2 := [[e2]]^S)
      t = self ()
  in newEdge (t1, t); newEdge (t2, t);
    release t1; release t2; yield ();
    (!l1, !l2)
```

Translation of well-formed expressions into dag calculus Now we define the translation for well-formed expressions and annotated expressions. For well-formed, annotated expressions we take an annotated expression and a set of its dependencies:

$$[\![e^{(1,l,t)}]\!](T) = [\![e]\!](l, t, T) \qquad [\![v^{(2,l)}]\!](T) = \emptyset, \emptyset, [l \mapsto v]$$

For well-formed expressions we have $[\![-]\!] : \mathcal{E}_I \to \mathcal{L} \times \mathrm{TId} \times \mathcal{P}_{\mathrm{fin}}(\mathrm{TId}) \to \mathcal{S}$, where

$\mathcal{S}$ denotes the dag states. We have:

$$[\![e]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![e]\!]^S, \mathsf{R})], \{(t,t') \mid t' \in T\}, [l \mapsto ()] \quad \text{if } \mathrm{surf}(e)$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^0, (e_2)^0))^{(1,l_1)}]]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[\texttt{par1}([\![e_1]\!]^S, [\![e_2]\!]^S, l_1)], \mathsf{X})], \{(t,t') \mid t' \in T\}, [l \mapsto (), l_1 \mapsto ()]$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^0, (e_2)^0))^{(2,l_1,l_2)}]]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[\texttt{par2}([\![e_1]\!]^S, [\![e_2]\!]^S, l_1, l_2)], \mathsf{X})],$$
$$\quad \{(t,t') \mid t' \in T\}, [l \mapsto (), l_1 \mapsto (), l_2 \mapsto ()]$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^0, (e_2)^0))^{(3,l_1,l_2,t_1)}]]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[\texttt{par3}([\![e_2]\!]^S, l_1, l_2, t_1)], \mathsf{X}), t_1 \mapsto (l_1 := [\![e_1]\!]^S, \mathsf{N})],$$
$$\quad \{(t,t') \mid t' \in T\}, [l \mapsto (), l_1 \mapsto (), l_2 \mapsto ()]$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^0, (e_2)^0))^{(4,l_1,l_2,t_1,t_2)}]]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[\texttt{par4}(l_1, l_2, t_1, t_2)], \mathsf{X}), t_1 \mapsto (l_1 := [\![e_1]\!]^S, \mathsf{N}), t_2 \mapsto (l_2 := [\![e_2]\!]^S, \mathsf{N})],$$
$$\quad \{(t,t') \mid t' \in T\}, [l \mapsto (), l_1 \mapsto (), l_2 \mapsto ()]$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^0, (e_2)^0))^{(5,l_1,l_2,t_1,t_2)}]]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[\texttt{par5}(l_1, l_2, t_1, t_2, t)], \mathsf{X}), t_1 \mapsto (l_1 := [\![e_1]\!]^S, \mathsf{N}), t_2 \mapsto (l_2 := [\![e_2]\!]^S, \mathsf{N})],$$
$$\quad \{(t,t') \mid t' \in T\}, [l \mapsto (), l_1 \mapsto (), l_2 \mapsto ()]$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^0, (e_2)^0))^{(6,l_1,l_2,t_1,t_2)}]]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[\texttt{par6}(l_1, l_2, t_1, t_2, t)], \mathsf{X}), t_1 \mapsto (l_1 := [\![e_1]\!]^S, \mathsf{N}), t_2 \mapsto (l_2 := [\![e_2]\!]^S, \mathsf{N})],$$
$$\quad \{(t,t') \mid t' \in T\} \cup \{(t_1,t)\}, [l \mapsto (), l_1 \mapsto (), l_2 \mapsto ()]$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^0, (e_2)^0))^{(7,l_1,l_2,t_1,t_2)}]]\!](l,t,T) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[\texttt{par7}(l_1, l_2, t_1, t_2)], \mathsf{X}), t_1 \mapsto (l_1 := [\![e_1]\!]^S, \mathsf{N}), t_2 \mapsto (l_2 := [\![e_2]\!]^S, \mathsf{N})],$$
$$\quad \{(t,t') \mid t' \in T\} \cup \{(t_1,t),(t_2,t)\}, [l \mapsto (), l_1 \mapsto (), l_2 \mapsto ()]$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^{a_1}, (e_2)^0))^{(8,l_1,l_2,t_2)}]]\!](l,t,T) =$$
$$\quad [\![(e_1)^{a_1}]\!](l_1,t_1,\{t\}) \uplus ([t \mapsto (l := [\![L]\!]^L[\texttt{par8}(l_1, l_2, t_2)], \mathsf{X}), t_2 \mapsto (l_2 := [\![e_2]\!]^S, \mathsf{N})],$$
$$\quad \{(t,t') \mid t' \in T\} \cup \{(t_2,t)\}, [l \mapsto (), l_2 \mapsto ()])$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^{a_1}, (e_2)^{a_2}))^{(9,l_1,l_2)}]]\!](l,t) =$$
$$\quad [\![(e_1)^{a_1}]\!](l_1,t_1,\{t\}) \uplus [\![(e_2)^{a_2}]\!](l_2,t_2,\{t\}) \uplus$$
$$\quad ([t \mapsto (l := [\![L]\!]^L[\texttt{par9}(l_1, l_2)], \mathsf{X})], \{(t,t') \mid t' \in T\}, [l \mapsto ()])$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^{a_1}, (e_2)^{a_2}))^{(10,l_1,l_2)}]]\!](l,t) =$$
$$\quad [\![(e_1)^{a_1}]\!](l_1,t_1,\{t\}) \uplus [\![(e_2)^{a_2}]\!](l_2,t_2,\{t\}) \uplus$$
$$\quad ([t \mapsto (l := [\![L]\!]^L[(!\, l_1, !\, l_2)], \mathsf{R})], \{(t,t') \mid t' \in T\}, [l \mapsto ()])$$
$$[\![L[(\texttt{forkjoin}\ ((e_1)^{a_1}, (e_2)^{a_2}))^{(11,l_1,l_2)}]]\!](l,t) =$$
$$\quad [t \mapsto (l := [\![L]\!]^L[([\![v_1]\!]^S, !\, l_2)], \mathsf{X})], \{(t,t') \mid t' \in T\}, [l \mapsto (), l_1 \mapsto [\![v_1]\!]^S, l_2 \mapsto [\![v_2]\!]^S]$$

The macrodefinitions are successive partial evaluations of the translation of the parallel composition. These are as follows:

```
par1(e1, e2, l1) =
  let l2 = alloc
      t1 = newTd (l1 := e1)
      t2 = newTd (l2 := e2)
      t = self()
  in newEdge (t1, t); newEdge (t2, t);
    release t1; release t2; yield();
    (!l1, !l2)

par2(e1, e2, l1, l2) =
  let t1 = newTd (l1 := e1)
      t2 = newTd (l2 := e2)
      t = self()
  in newEdge (t1, t); newEdge (t2, t);
    release t1; release t2; yield();
    (!l1, !l2)

par3(e2, l1, l2, t1) =
  let t2 = newTd (l2 := e2)
      t = self()
  in newEdge (t1, t); newEdge (t2, t);
    release t1; release t2; yield();
    (!l1, !l2)

par4(l1, l2, t1, t2) =
  let t = self()
  in newEdge (t1, t); newEdge (t2, t);
    release t1; release t2; yield();
    (!l1, !l2)

par5(l1, l2, t1, t2, t) =
  newEdge (t1, t); newEdge (t2, t);
  release t1; release t2; yield();
  (!l1, !l2)

par6(l1, l2, t1, t2, t) =
  newEdge (t2, t);
  release t1; release t2; yield();
  (!l1, !l2)

par7(l1, l2, t1, t2) =
  release t1; release t2; yield();
  (!l1, !l2)

par8(l1, l2, t2) =
  release t2; yield(); (!l1, !l2)

par9(l1, l2) =
  yield (); (!l1, !l2)
```

Correctness With the translations defined, we can connect the annotated expressions with the dag calculus states (through the definition of *matching* states $\propto$). Then, Lemma 5 allows us to locate any executing thread as a subterm of our computation. This is of

crucial importance, since this is what restricts the reduction steps that the dag calculus can take in that thread: the code must come from the translation of that subcomputation. Using this property, we can prove Lemma 6, which states that every step on the dag calculus side is either a scheduler step, or can be matched on the side of the instrumented expressions. Since we know the precise subexpression where the reduction happens, we can generally restrict ourselves to ensuring that the administrative reductions on the instrumented expression side match, step for step, the execution of the translation of the parallel pair — which is how the translation was designed in the first place. Finally, Lemma 7 gives us the backwards simulation of dag calculuswith the instrumented semantics, and the Correctness Theorem composes it with the instrumentation lemma, Lemma 4. The proof process is much the same for the other translations.

Definition 4. We say that a well-formed annotated instrumented expression e^a such that $\mathrm{wf}(e^a)$ *matches* a dag calculusstate V, E, σ, written $e^a \propto V, E, \sigma$ if there exists a dag state V', E', σ' such that $\sigma' \sqsubseteq \sigma$ and $[\![e^a]\!](\emptyset) = V', E, \sigma'$, and a finite map of vertices V'' such that $\forall t \in \mathrm{dom}(V'')$. $V''(t) = (( ), \mathsf{R})$, and that

$$V' \uplus V'', E', \sigma \twoheadrightarrow_{\mathrm{s}}^{*} V, E, \sigma.$$

Lemma 5. *For any e^a, V, E, σ, t if* $\mathrm{wf}(e^a)$, $e^a \propto V, E, \sigma$, $t \in \mathrm{dom}(V)$ *and* $\mathrm{status}(V(t)) = \mathsf{X}$, *then either*

1. $V(t) = (( ), \mathsf{X})$,
2. *there exist K, e', l such that $e^a = K[e'^{(1,l,t)}]$,*

Lemma 6. *For any e^a, V, V', E, E', σ, σ' if* $\mathrm{wf}(e^a)$, $e^a \propto V, E, \sigma$ *and* $V, E, \sigma \twoheadrightarrow V', E', \sigma'$ *then either $e^a \propto V', E', \sigma'$ or there exists $e'^{a'}$ such that $e^a \to e'^{a'}$ and $e'^{a'} \propto V', E', \sigma'$.*

Lemma 7. *For any e, a, V, V', E, σ, σ', if* $\mathrm{wf}(e^a)$, $e^a \propto V, E, \sigma$ *and* $\forall t \in \mathrm{dom}(V')$. $V'(t) = (( ), \mathsf{F})$, *then the following simulation holds:*

$$V, E, \sigma \twoheadrightarrow^{*} V', \emptyset, \sigma' \quad \Rightarrow \quad \exists v.\ e^a \to^{*} v^{(2,\mathrm{loc}(a))} \wedge \sigma'(l) = [\![v]\!]^S$$
$$V, E, \sigma \twoheadrightarrow^{\infty} \quad \Rightarrow \quad e^a \to^{\infty}$$

Proof. Follows from Lemmas 6 and 1 by the same inductive and coinductive argument as used in the proof of Lemma 4. □

Theorem 1 (Correctness)**.** *Let t be the identifier of the main thread, $e \in \mathcal{E}_O$ be the source program stored in this thread, and l be a designated location in which to store the final result. Let $e_I \in \mathcal{E}_I$ be e annotated with the* 0 *annotations on all parallel compositions. For any integer result n, final state σ such that $\sigma(l) = n$, and final set of vertices V, assuming that all threads t' in V are finished (i.e.* $\mathrm{status}\,(V(t')) = \mathsf{F}$*), we have:*

$$[t \mapsto (l := [\![e_I]\!]^S, \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^{*} V, \emptyset, \sigma \quad \Rightarrow \quad e \to_O^{*} n.$$

Furthermore, divergence in the dag calculus entails divergence in the source language:

$$[t \mapsto (l := [\![e_I]\!]^S, \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^{\infty} \quad \Rightarrow \quad e \to_O^{\infty}.$$

Proof. This theorem follows from the composition of the two simulation diagrams, given by Lemmas 4 and 7. Clearly, $\lfloor e_I \rfloor = e$. Moreover, since $\mathrm{surf}(e_I)$, we also have $[\![e_I^{(1,l,t)}]\!](\emptyset) = [t \mapsto (l{:=}[\![e_I]\!]^S, \mathsf{R})], \emptyset, [l \mapsto ()]$, which gives us $e_I^{(1,l,t)} \propto [t \mapsto (l{:=}[\![e_I]\!]^S, \mathsf{R})], \emptyset, [l \mapsto ()]$.

For the termination case, by Lemma 7 we obtain that there exists a value v such that $e_I^{(1,l,t)} \to_I^* v^{(2,l)}$ and $\sigma'(l) = [\![v]\!]^S$. However, since $\sigma'(l) = n$, we can conclude that $v = n$. By Lemma 4 we can now conclude that $e \to_O^* \lfloor n \rfloor$, which ends the proof, as $\lfloor n \rfloor = n$.

For the nontermination case, from Lemma 7 conclude that $e_I^{(1,l,t)} \to_I^\infty$, which, by Lemma 4 implies that $e \to_O^\infty$. This ends the proof. □

C Correctness of the translation of async-finish

C.1 Syntax and Operational Semantics

$$
\begin{aligned}
e &::= v \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \\
&\quad \mid \texttt{alloc} \mid v := v \mid !\,v \mid \texttt{async}(e) \mid \texttt{finish}(S) \\
v &::= x \mid n \mid l \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
S &\equiv \{e_1, e_2, \ldots, e_n\} \\
K &::= L \mid L[\texttt{finish}(\{K\} \uplus S)] \\
L &::= \bullet \mid \texttt{let}\ x = L\ \texttt{in}\ e
\end{aligned}
$$

The input programs are restricted to use the form $\texttt{finish}(\{e\})$. The operational semantics is given below. It operates on the store $\mathcal{H} = \mathcal{L} \rightharpoonup_{fin} \mathcal{V}$

$$
\frac{}{h, \texttt{fst}\ (v_1, v_2) \to v_1, h} \qquad \frac{}{h, \texttt{snd}\ (v_1, v_2)] \to v_2, h} \qquad \frac{}{h, \texttt{let}\ x = v\ \texttt{in}\ e \to e[x \mapsto v], h}
$$

$$
\frac{}{h, \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}\ v \to e[x \mapsto v, f \mapsto \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}], h}
$$

$$
\frac{l \notin \mathrm{dom}(h)}{h, \texttt{alloc} \to l, h[l \mapsto ()]} \qquad \frac{h(l) = v}{h, !\,l \to v, h} \qquad \frac{l \in \mathrm{dom}(h)}{h, l := v \to (), h[l \mapsto v]}
$$

$$
\frac{}{h, \texttt{finish}(\{L[\texttt{async}(e)]\} \uplus S) \to \texttt{finish}(\{L[()], e\} \uplus S), h}
$$

$$
\frac{\forall e \in S.\ e = ()}{h, \texttt{finish}(S) \to (), h} \qquad \frac{h, e \to e', h'}{h, K[e] \to K[e'], h'}
$$

Note the operational semantics is changed slightly wrt. the paper: the finished async threads are not removed one by one, but rather stay as values associated with the enclosing finish block, which finishes when all the async computations terminate. This is done to facilitate the proof, since the removal of a single finished async would correspond to a scheduler transition (Stop), and it is easier to keep the scheduler transitions separate from the others. It is a simple exercise to show this presentation equivalent to the one presented in the paper: since the removals of finished async threads are nondeterministic,

they can all happen right before the finish block should reduce. The other direction is even simpler.

The following lemma allows us to only consider programs that write their final value to a predetermined location in the state in the following, since these can simulate the other programs.

Lemma 8. *For any program e and location l and result state h, we have the following properties:*

$$[l \mapsto ()], \texttt{let}\ x = e\ \texttt{in}\ l := x \to^* (), h \quad \Rightarrow \quad \emptyset, e \to^* h(l), h \setminus [l \mapsto h(l)],$$

$$[l \mapsto ()], \texttt{let}\ x = e\ \texttt{in}\ l := x \to^\infty \quad \Rightarrow \quad \emptyset, e \to^\infty .$$

Proof. Simple induction. □

C.2 Instrumented Syntax and Operational Semantics

When there is a chance of confusion between the instrumented and original language, we annotate the constructs of the instrumented language with a subscript I, and the constructs of original language with O. The instrumentations are a_a for async and a_f for finish. Additionally, we annotate roots of expressions with thread identifiers.

$$\begin{aligned}
e &::= v \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \mid \texttt{alloc} \mid v := v \mid !\, v \\
&\quad \mid \texttt{async}(e)^{a_a} \mid \texttt{finish}(e)^{a_f} \mid \texttt{finish}(S)^{a_g} \\
v &::= x \mid n \mid l \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
S &\equiv \{e_1^{t_1}, e_2^{t_2}, \ldots, e_n^{t_n}\} \\
L &::= \bullet \mid \texttt{let}\ x = L\ \texttt{in}\ e \\
K &::= \bullet \mid (L[\texttt{finish}(\{K\} \uplus S)^a])^t \\
a_a &::= 0 \mid (1, t) \mid (2, t) \\
a_f &::= 0 \mid 1 \mid (2, t) \mid (3, t) \\
a_g &::= 1 \mid 2
\end{aligned}$$

We define the surface expressions, $\mathrm{surf}(e)$ as ones that only contain annotations of 0 and do not contain subexpressions of the form $\texttt{finish}(S)$. This notion extends in a trivial way to the sequential contexts L. With this, we can proceed to define the well-formed expressions, $\mathrm{wf}(e)$:

$$\mathrm{wf}(e) = \begin{cases} \mathrm{surf}(e') \wedge \mathrm{surf}\ L & \text{if } e = L[\texttt{async}(e')^a] \\ \mathrm{surf}(e') \wedge \mathrm{surf}\ L & \text{if } e = L[\texttt{finish}(e')^a] \\ (\forall e', t.\ (e'^t) \in S \Rightarrow \mathrm{wf}(e')) \wedge \mathrm{surf}\ L & \text{if } e = L[\texttt{finish}(S)^a] \\ \mathrm{surf}\ e & \text{otherwise} \end{cases}$$

This notion simply extends to contexts K.

For operational semantics, let us first define the allowed transitions on annotations, $\rightsquigarrow_a$ for the async annotations, and $\rightsquigarrow_f$ and $\rightsquigarrow_g$ for the finish ones. The essence is that

the associated number can increase by one, and if any new thread identifiers appear, they must be globally fresh. Otherwise, thread ids are preserved.

$$0 \rightsquigarrow_a (1,t) \text{ where } t \text{ is fresh} \qquad (1,t) \rightsquigarrow_a (2,t)$$

$$0 \rightsquigarrow_f 1 \qquad 1 \rightsquigarrow_f (2,t) \text{ where } t \text{ is fresh}$$
$$(2,t) \rightsquigarrow_f (3,t) \qquad 1 \rightsquigarrow_g 2$$

Next, we can define the operational semantics for the instrumented language. For the most part, it is the same as the source language, with key differences being the reduction for annotation steps and the separate preparatory finish construct.

$$\frac{}{h, \mathtt{fst}\ (v_1, v_2) \to v_1, h} \qquad \frac{}{h, \mathtt{snd}\ (v_1, v_2) \to v_2, h} \qquad \frac{}{h, \mathtt{let}\ x = v\ \mathtt{in}\ e \to e[x \mapsto v], h}$$

$$\frac{}{h, \mathtt{fun}\ f\ x\ \mathtt{is}\ e\ \mathtt{end}\ v \to e[x \mapsto v, f \mapsto \mathtt{fun}\ f\ x\ \mathtt{is}\ e\ \mathtt{end}], h}$$

$$\frac{l \notin \mathrm{dom}(h)}{h, \mathtt{alloc} \to l, h[l \mapsto ()]} \qquad \frac{h(l) = v}{h, !\,l \to v, h} \qquad \frac{l \in \mathrm{dom}(h)}{h, l := v \to (), h[l \mapsto v]}$$

$$\frac{t \text{ is fresh}}{h, \mathtt{async}(e)^0 \to \mathtt{async}(e)^{(1,t)}} \qquad \frac{}{h, \mathtt{finish}(\{(L[\mathtt{async}(e)^{(1,t)}])^{t'}\} \uplus S) \to \mathtt{finish}(\{(L[\mathtt{async}(e)^{(2,t)}])^{t'}\} \uplus S), h}$$

$$\frac{}{h, \mathtt{finish}(\{(L[\mathtt{async}(e)^{(2,t)}])^{t'}\} \uplus S) \to \mathtt{finish}(\{(L[()])^{t'}, e^t\} \uplus S), h}$$

$$\frac{a \rightsquigarrow_f a'}{h, \mathtt{finish}(e)^a \to \mathtt{finish}(e)^{a'}, h} \qquad \frac{}{h, \mathtt{finish}(e)^{(3,t)} \to \mathtt{finish}(\{e^t\})^1, h}$$

$$\frac{}{h, \mathtt{finish}(S)^1 \to \mathtt{finish}(S)^2, h} \qquad \frac{\forall e, t.\ e^t \in S \Rightarrow e = ()}{h, \mathtt{finish}(S)^2 \to (), h} \qquad \frac{h, e \to e', h'}{h, K[(L[e])^t] \to K[(L[e'])^t], h'}$$

Lemma 9. *Evaluation preserves well-formedness, i.e., for any* e, e', h, h'*, if* $\mathrm{wf}(e)$ *and* $h, e \to e', h$*, then* $\mathrm{wf}(e')$*.*

C.3 Connecting instrumented language and source language

In order to connect the instrumented language expressions to the source language ones, we introduce an *erasure* function: $\lfloor - \rfloor : \mathcal{E}_I \to \mathcal{E}_O$, that removes annotations from terms and maps $\mathtt{finish}(e)^a$ to $\mathtt{finish}(\{\lfloor e \rfloor\})$. Erasure function extends in the obvious way to heaps and evaluation contexts.

Lemma 10 (Instr-Step)**.** *For any expressions* $e, e' \in \mathcal{E}_I$ *and heaps* $h, h' \in \mathcal{H}_I$*, if* $h, e \to_I e', h'$*, then either* $\lfloor e \rfloor = \lfloor e' \rfloor$ *and* $h = h'$*, or* $\lfloor h \rfloor, \lfloor e \rfloor \to_O \lfloor e' \rfloor, \lfloor h' \rfloor$*.*

Proof. By induction on the reduction judgment. These split into three forms: the administrative reductions of `async` and `finish`, which give the same erasure for both the redex and the reduct, the reductions of sequential constructs, which can be trivially matched, and the two rules that spawn an async thread and terminate the finish block — which also can be easily matched. □

Definition 5. An *administrative* reduction in the instrumented language is a reduction $h, e \to e', h'$ such that $\lfloor e \rfloor = \lfloor e' \rfloor$ and $h = h'$. We write $h, e \to_a e', h'$ for administrative steps.

Lemma 11 (Admin-Fin)**.** *There are no infinite sequences of administrative reductions, i.e., for any $e \in \mathcal{E}_I$ and $h \in \mathcal{H}_I$, if* wf(e) *then* $h, e \not\to_a^\infty$.

Proof. Assign as a weight of the expression based on the number in each of its annotations in evaluation positions. Take num(a) to denote the number associated with the annotation a. Then define

$$w(e) = \begin{cases} 2 - \text{num}(a) & \text{if } e = L[\texttt{async}(e')^a] \\ 3 - \text{num}(a) & \text{if } e = L[\texttt{future}(e')^a] \\ 2 - a + \sum_{e' \in S} w(e) & \text{if } e = L[\texttt{future}(S)^a] \\ 0 & \text{otherwise} \end{cases}.$$

Observe that each administrative reduction decreases the weight. The lemma then holds by simple induction on the weight. □

Lemma 12 (Instrument)**.** *For any expression $e \in \mathcal{E}_I$ and heaps $h, h' \in \mathcal{H}_I$ such that $h, e \to_I^* (), h', \lfloor h \rfloor, \lfloor e \rfloor \to_O^* (), \lfloor h' \rfloor$. Moreover, if $h, e \to_I^\infty$ then $\lfloor h \rfloor, \lfloor e \rfloor \to_O^\infty$.*

Proof. First part, by induction on the reduction sequence and Lemma 10.

Similarly, for the second part we proceed by coinduction: since $h, e \to_I^\infty$, by Lemma 18 and 10 there exist e' and h' such that $h, e \to_I^* e', h', \lfloor h \rfloor, \lfloor e \rfloor \to_O \lfloor e' \rfloor, \lfloor h' \rfloor$ and $h', e' \to_I^\infty$. Hence, $\lfloor h' \rfloor, \lfloor e' \rfloor \to_O^\infty$ and finally $\lfloor h \rfloor, \lfloor e \rfloor \to_O^\infty$. □

C.4 Connecting instrumented language and dag calculus

Translation of surface expressions into dag calculus Note that the translation trivially extends to the evaluation contexts of the instrumented language. We define a translation $[\![- \mid -]\!]^S : \mathcal{V}_{dagcalculus} \times \mathcal{E}_I \to \mathcal{E}_{dagcalculus}$, where the first argument keeps track of the *nearest enclosing finish clock*. Note that this differes from the form presented in the paper only in the order of the arguments. The bulk of the translation is a standard destination-passing translation:

$$[\![t \mid \texttt{let } x = e_1 \texttt{ in } e_2]\!] = \texttt{let } x = [\![t \mid e_1]\!] \texttt{ in } [\![t \mid e_2]\!]$$
$$[\![t \mid \texttt{fun } f\ x \texttt{ is } e \texttt{ end}]\!] = \texttt{fun } f\ (x, t') \texttt{ is } [\![t' \mid e]\!] \texttt{ end}$$
$$[\![t \mid v_1\ v_2]\!] = [\![t \mid v_1]\!]\ ([\![t \mid v_2]\!], t)$$

The translation for parallel primitives is defined as follows:

```
1  [[t | async(e)^0]] =
2    let t' = newTd ([[t | e]])
3    in newEdge (t', t); release t'; ()
4
5  [[t | finish(e)^0]] =
6    let t2 = self ()
7        t1 = newTd ([[t2 | e]])
8    in newEdge (t1, t2); release t1; yield(); ()
```

Note that due to its destination-passing flavour, the translation for values does not depend on the thread identifier. Thus, we can also extend this translation to work on heaps: we denote it $[\![-]\!]^H : \mathcal{H}_I \to \mathcal{H}_dagcalculus$.

Translation of the well-formed expressions into dag calculus The translation for well-formed, thread-annotated expressions is as follows, defined mutually recursively with the translation for expressions. The latter has the form $[\![-]\!] : \mathcal{E}_I \to \mathrm{TId}{\times}\mathrm{TId}^{\perp} \to \mathcal{D}$, where $\mathcal{D}$ denotes computation dags of the form (V, E). The argument that tracks the continuation vertex is $\perp$ outside the topmost finish block. We define a helper function

$$\mathrm{mkEdge}\,(t, t') = \begin{cases} \emptyset & \text{if } t' = \perp \\ \{(t, t')\} & \text{otherwise} \end{cases}$$

The definition for thread-annotated expressions is just shorthand: the thread-annotation is passed as an argument to the main translation.

$$[\![e^{t}]\!](t') = [\![e]\!](t, t')$$

$[\![e]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid e]\!]^{S}, \mathsf{R})], \mathrm{mkEdge}\,(t, t') \quad \text{if } \mathrm{surf}(e)$

$[\![L[\texttt{async}(e)^{(1,t_a)}]]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid L]\!]^{L}[\texttt{async1}(t_a, t')], \mathsf{X}), t_a \mapsto ([\![t' \mid e]\!]^{S}, \mathsf{N})], \mathrm{mkEdge}\,(t, t')$

$[\![L[\texttt{async}(e)^{(2,t_a)}]]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid L]\!]^{L}[\texttt{release}\ t_a], \mathsf{X}), t_a \mapsto ([\![t' \mid e]\!]^{S}, \mathsf{N})], \mathrm{mkEdge}\,(t, t') \cup \mathrm{mkEdge}\,(t_a, t')$

$[\![L[\texttt{finish}(e)^{1}]]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid L]\!]^{L}[\texttt{finish1}([\![t \mid e]\!]^{S}, t)], \mathsf{X})], \mathrm{mkEdge}\,(t, t')$

$[\![L[\texttt{finish}(e)^{(2,t_a)}]]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid L]\!]^{L}[\texttt{finish2}(t_a, t)], \mathsf{X}), t_a \mapsto ([\![t \mid e]\!]^{S}, \mathsf{N})], \mathrm{mkEdge}\,(t, t')$

$[\![L[\texttt{finish}(e)^{(3,t_a)}]]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid L]\!]^{L}[\texttt{finish3}(t_a, t)], \mathsf{X}), t_a \mapsto ([\![t \mid e]\!]^{S}, \mathsf{N})], \mathrm{mkEdge}\,(t, t') \cup \{(t_a, t)\}$

$[\![L[\texttt{finish}(S)^{1}]]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid L]\!]^{L}[\texttt{yield(); ()}], \mathsf{X})], \mathrm{mkEdge}\,(t, t')) \uplus \biguplus_{e^{t_e} \in S} [\![e^{t_e}]\!](t)$

$[\![L[\texttt{finish}(S)^{2}]]\!](t, t') =$

$\quad [t \mapsto ([\![t' \mid L]\!]^{L}[\texttt{()}], \mathsf{R})], \mathrm{mkEdge}\,(t, t')) \uplus \biguplus_{e^{t_e} \in S} [\![e^{t_e}]\!](t)$

The macrodefinitions used above are as follows:

```
1  async1(t, t') = newEdge (t, t'); release t; ()
2
3  finish1(e, t') =
4    let t = newTd (e)
5    in newEdge (t, t'); release t; yield(); ()
6
```

```
 7  finish2(t, t') =
 8    newEdge (t, t'); release t; yield(); ()
 9
10  finish3(t) =
11    release t; yield(); ()
```

Correctness

Definition 6. We say that a configuration of annotated instrumented expression and state e^t, h such that $\mathrm{wf}(e^t)$ and $\mathrm{surf}(h)$ *matches* a dag calculusstate V, E, σ, written $e^t, h \propto V, E, \sigma$ if $\sigma = [\![h]\!]^H$, and there exists a dag (V', E') such that $[\![\,() \mid e^t]\!] = V', E'$ and a finite map of vertices V'' such that $\forall t \in \mathrm{dom}(V'').\ V''(t) = ((), \mathsf{R})$, and that

$$V' \uplus V'', E', \sigma \twoheadrightarrow_{\mathrm{s}}^* V, E, \sigma.$$

Lemma 13. *For any e^{t_0}, h, V, E, σ, t if* $\mathrm{wf}(e)$, *$e^{t_0}, h \propto V, E, \sigma$, $t \in \mathrm{dom}(V)$ and* $\mathrm{status}(V(t)) = \mathsf{X}$, *then either $V(t) = ((), \mathsf{X})$, or there exist K, e' such that $e^{t_0} = K[e'^t]$.*

Lemma 14. *For any e^{t_0}, h, V, V', E, E', σ, σ' if* $\mathrm{wf}(e^{t_0})$, *$e^{t_0}, h \propto V, E, \sigma$ and $V, E, \sigma \twoheadrightarrow V', E', \sigma'$ then either $e^{t_0}, h \propto V', E', \sigma'$ or there exist $e'^{t'_0}$, h' such that $h, e^{t_0} \to e'^{t'_0}, h'$ and $e'^{t'_0}, h' \propto V', E', \sigma'$.*

Lemma 15. *For any e, t, h, V, V', E, σ, σ', if* $\mathrm{wf}(e^t)$, *$e^t, h \propto V, E, \sigma$ and $\forall t' \in \mathrm{dom}(V')$. $V'(t') = ((), \mathsf{F})$, then the following simulation holds:*

$$V, E, \sigma \twoheadrightarrow^* V', \emptyset, \sigma' \quad \Rightarrow \quad \exists v, t', h'.\ h, e^t \to^* v^{t'}, h' \wedge \sigma' = [\![h']\!]^H$$
$$V, E, \sigma \twoheadrightarrow^\infty \quad \Rightarrow \quad h, e^t \to^\infty$$

Proof. Follows from Lemmas 14 and 1 by the same inductive and coinductive argument as used in the proof of Lemma 12. □

Theorem 2 (Correctness)**.** *Let t be the identifier of the main thread, e be the source program stored in this thread, l be a designated location in which to store the final result, and e_I be the result of annotating e with* 0 *on all* `finish` *and* `async` *constructs. For any integer result n, final state σ such that $\sigma(l) = n$, and final set of vertices V, assuming that all threads t' in V are finished (i.e.* $\mathrm{status}\,(V(t')) = \mathsf{F}$*), we have:*

$$[t \mapsto (\texttt{let}\ x = [\![\,() \mid e_I]\!]^S\ \texttt{in}\ l := x, \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^* V, \emptyset, \sigma \ \Rightarrow\ \exists h.\ \emptyset, e \to_O^* n, h$$

Furthermore, divergence in the dag calculus entails divergence in the source language:

$$[t \mapsto (\texttt{let}\ x = [\![\,() \mid e_I]\!]^S\ \texttt{in}\ l := x, \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^\infty \quad \Rightarrow \quad \emptyset, e \to_O^\infty .$$

Proof. This theorem follows from the composition of the two simulation diagrams, given by Lemmas 12 and 15. Clearly, $\lfloor e_I \rfloor = e$. Moreover, since $\mathrm{surf}(e_I)$, we also have $[\![(\texttt{let}\ x = e_I\ \texttt{in}\ l := x)^t]\!](\bot) = [t \mapsto (\texttt{let}\ x = [\![\,() \mid e_I]\!]^S\ \texttt{in}\ l := x, \mathsf{R})], \emptyset$, which gives us $(\texttt{let}\ x = e_I\ \texttt{in}\ l := x)^t, [l \mapsto ()] \propto [t \mapsto (\texttt{let}\ x = [\![\,() \mid e_I]\!]^S\ \texttt{in}\ l := x, \mathsf{R})], \emptyset, [l \mapsto ()]$.

For the termination case, by Lemma 15 we obtain that there exist v, t' and h such that $[l \mapsto ()], (\texttt{let}\ x = e_I\ \texttt{in}\ l := x)^t \to_I^* v^{t'}, h$ and $\sigma' = [\![h]\!]^H$ (and hence $h(l) = n$).

Since the final command of the program is assignment, we can conclude that $v = \texttt{()}$. By Lemma 12 we can now conclude that $[l \mapsto \texttt{()}], \texttt{let } x = e \texttt{ in } l := x \to_O^* \texttt{()}, \lfloor h \rfloor$. Since $h(l) = n$, by Lemma 8 this gives us that $\emptyset, e \to_O^* n, \lfloor h \rfloor \setminus [l \mapsto n]$, which ends the proof.

For the nontermination case, from Lemma 15 conclude that $[l \mapsto \texttt{()}], (\texttt{let } x = e_I \texttt{ in } l := x)^t \to_I^\infty$, which, by Lemma 12 implies that $[l \mapsto \texttt{()}], \texttt{let } x = e \texttt{ in } l := x \to_O^\infty$. Finally, by Lemma 8 we obtain $\emptyset, e \to_O^\infty$ which ends the proof. □

D Correctness of the translation of futures

D.1 Syntax and Operational Semantics

The syntax and semantics in this section are the same as in the paper and repeated here for convenience.

$$
\begin{aligned}
e ::= & x \mid n \mid f \mid (v, v) \mid \texttt{fst } v \mid \texttt{snd } v \mid \texttt{let } x = e \texttt{ in } e \mid v\ v \\
& \mid \texttt{fun } g\ x \texttt{ is } e \texttt{ end} \mid \texttt{future}(e) \mid \texttt{force}(v) \\
v ::= & x \mid n \mid f \mid (v, v) \mid \texttt{fun } g\ x \texttt{ is } e \texttt{ end} \\
K ::= & \bullet \mid \texttt{let } x = K \texttt{ in } e
\end{aligned}
$$

We take the set of configurations $\mathcal{M} \equiv \mathcal{F} \rightharpoonup_{fin} \mathcal{E}$, where $\mathcal{F}$ is the set of future identities and $\mathcal{E}$ — the set of expressions, and write $M \in \mathcal{M}$ for these configurations. We also pick an identifier of the main program, f_0: thus, the initial state is $[f_0 \mapsto e]$ for some program e. The evaluation reaches a terminal state when all the futures in M have values associated with them.

$$
\texttt{fst } (v_1, v_2) \bigcirc\!\!\!\!\to v_1 \qquad \texttt{snd } (v_1, v_2) \bigcirc\!\!\!\!\to v_2 \qquad \texttt{let } x = v \texttt{ in } e \bigcirc\!\!\!\!\to e[x \mapsto v]
$$
$$
\texttt{fun } g\ x \texttt{ is } e \texttt{ end}\ v \bigcirc\!\!\!\!\to e[x \mapsto v, g \mapsto \texttt{fun } g\ x \texttt{ is } e \texttt{ end}]
$$

$$
\frac{M(f) = K[e] \qquad e \bigcirc\!\!\!\!\to e'}{M \to M[t \mapsto K[e']]}
$$
$$
\frac{M(f) = K[\texttt{future}(e)] \qquad f' \text{ fresh}}{M \to M[f \mapsto K[f'], f' \mapsto e]}
$$
$$
\frac{M(f) = K[\texttt{force}(f')] \qquad M(f') = v}{M \to M[f \mapsto K[v]]}
$$

D.2 Instrumented Syntax and Operational Semantics

The instrumented language is presented below. The input terms can only be annotated with 0. When there is a chance of confusion between the instrumented and original

language, we annotate the constructs of the instrumented language with a subscript I, and the constructs of original language with O. The instrumentations are a_s for future creation, a_f for future access, and a_w for future evaluation.

$$
\begin{aligned}
e &::= x \mid n \mid t \mid (v, v) \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \\
&\quad \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \mid \texttt{future}(e)^{a_s} \mid \texttt{force}(v)^{a_f} \\
v &::= x \mid n \mid t \mid (v, v) \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\
a_s &::= 0 \mid (1, l) \mid (2, l, t) \\
a_f &::= n \text{where } 0 \le n \le 4 \\
a_w &::= 0 \mid 1 \\
K &::= \bullet \mid \texttt{let}\ x = K\ \texttt{in}\ e
\end{aligned}
$$

The annotations a_s and a_f can *progress*, written $a \rightsquigarrow_s a'$ (and, respectively $a \rightsquigarrow_f a'$), if the step of a' is one greater than the step of a. All the other parts of a' present in a have to agree, while those absent in a are globally fresh, as follows:

$$
0 \rightsquigarrow_s (1, l) \text{ where } l \text{ is fresh} \qquad (1, l) \rightsquigarrow_s (2, l, t) \text{ where } t \text{ is fresh}
$$
$$
n \rightsquigarrow_f n + 1 \text{ for } 0 \le n \le 3
$$

Sequential evaluation within a future proceeds as follows:

$$
\texttt{fst}\ (v_1, v_2) \circlearrowright v_1 \qquad \texttt{snd}\ (v_1, v_2) \circlearrowright v_2 \qquad \texttt{let}\ x = v\ \texttt{in}\ e \circlearrowright e[x \mapsto v]
$$
$$
\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}\ v \circlearrowright e[x \mapsto v, f \mapsto \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}]
$$
$$
\frac{a \rightsquigarrow_f a'}{\texttt{future}(e)^{a} \circlearrowright \texttt{future}(e)^{a'}} \qquad \frac{a \rightsquigarrow_s a'}{\texttt{force}(e)^{a} \circlearrowright \texttt{force}(e)^{a'}}
$$

The configurations are defined as $\mathcal{M} = \mathcal{F} \rightharpoonup_{\mathit{fin}} (\mathcal{L} \times \mathrm{TId} \times \mathcal{E} \times a_w)$, and well-formed configurations have all the locations and thread id's disjoint. The special starting future, f_0 has a special first thread id, t_0. The evaluation then proceeds as follows.

$$
\frac{M(f) = (l, t, K[e], 0) \qquad K[e] \circlearrowright K[e']}{M \to M[t \mapsto (l, t, K[e'], 0)]}
$$
$$
\frac{M(f) = (l, t, K[\texttt{future}(e)^{(3,l',t')}], 0) \qquad f' \text{ fresh}}{M \to M[f \mapsto (l, t, K[f'], 0), f' \mapsto (l', t', e, 0)]}
$$
$$
\frac{M(f) = (l, t, K[\texttt{force}(f')^{4}], 0) \qquad M(f') = (l', t', v, 1)}{M \to M[f \mapsto (l, t, K[v], 0)]}
$$
$$
\frac{M(f) = (l, t, v, 0)}{M \to M[f \mapsto (l, t, v, 1)]}
$$

An expression is a *surface* expression, written surf(e) if all its annotations are 0. Note that these are isomorphic to the terms of the original language. A term is

well-formed, written $\mathrm{wf}(e)$, if it has at most one non-zero annotation, in an evaluation position, i.e.,

$$\mathrm{wf}(e) \equiv \begin{cases} \mathrm{wf}(e_1) \wedge \mathrm{surf}(e_2) & \text{if } e = \texttt{let } x = e_1 \texttt{ in } e_2 \\ \mathrm{surf}(e') & \text{if } e = \texttt{future}(e')^a \\ \mathrm{surf}(v) & \text{if } e = \texttt{force}(v)^a \\ \mathrm{surf}(e) & \text{otherwise} \end{cases}$$

We write $\mathrm{wf}(M)$ to mean that each of the expressions associated with the futures is well-formed.

Lemma 16. *Evaluation preserves well-formedness, i.e., if* $\mathrm{wf}(e)$ *and* $e \circ\!\!\rightarrow e'$ *then* $\mathrm{wf}(e')$. *Similarly, if* $\mathrm{wf}(M)$ *and* $M \to M'$, *then* $\mathrm{wf}(M')$.

Proof. The first part proceeds by simply checking the reductions. The second follows by simple induction on the structure of the expression associated with the future that reduces. □

D.3 Connecting the instrumented and source languages

We define a map $\lfloor - \rfloor : \mathcal{E}_I \to \mathcal{E}_O$ that removes annotations from the instrumented expressions. This map extends to the configurations, by also removing the additional information in the futures. We use it to connect the instrumented language to the source language. Note that a map that adds an annotation 0 to `future` and `force` is a right inverse of $\lfloor - \rfloor$. For *surface* term, it is also the left inverse.

Lemma 17 (Instr-Step)**.** *For any configurations* $M, M' \in \mathcal{M}_I$, *if* $M \to_I M'$, *then either* $\lfloor M \rfloor = \lfloor M' \rfloor$ *or* $\lfloor M \rfloor \to_O \lfloor M' \rfloor$.

Proof. By cases on the reduction. In cases of spawning a future, forcing a future of finishing a future, trivially true, since $\lfloor K[e] \rfloor = \lfloor K \rfloor [\lfloor e \rfloor]$. For the same reason, suffices to check the same statement for any $e \circ\!\!\rightarrow e'$.

These split into two forms: the administrative reductions of `future` and `force`, which give the same erasure for both the redex and the reduct, and the reductions of sequential constructs, which can be trivially matched. □

Definition 7. An *administrative* reduction in the instrumented language is a reduction $M \to M'$ such that $\lfloor M \rfloor = \lfloor M' \rfloor$. We write $M \to_a M'$ for administrative steps.

Lemma 18 (Admin-Fin)**.** *For well-formed configurations M, there are no infinite sequence of administrative reductions, i.e., for any* $M \in \mathcal{M}_I$, *if* wf M *then* $M \not\to_a^\infty$.

Proof. Assign as a weight to each future based on the count of their annotations in evaluation positions. Take the $\mathrm{cnt}(a)$ to be the natural number in the annotation. Then define

$$w(e) = \begin{cases} 3 - \mathrm{cnt}(a) & \text{if } e = \texttt{future}(e')^a \\ 4 - a & \text{if } e = \texttt{force}(e')^a \\ w(e_1) & \text{if } e = \texttt{let } x = e_1 \texttt{ in } e_2 \\ 0 & \text{otherwise} \end{cases},$$

$w(l, t_1, t_2, e, n) = w(e) + 1 - n$, and $w(M) = \sum_{f \in \mathrm{dom}(M)} w(M(f))$. From there, the proof is analogous to the proof of Lemma 1: by simple induction, each administrative reduction decreases the weight, and since the weight of a given configuration is finite, there is no infinite chain of reductions administrative. □

Lemma 19 (Instrument)**.** *For any configurations M, M' such that M' is final and $M \to_I^* M'$, $\lfloor M \rfloor \to_O^* \lfloor M' \rfloor$ and $\lfloor M' \rfloor$ is final. Moreover, if $M \to_I^\infty$, then $\lfloor M \rfloor \to_O^\infty$.*

Proof. First part, by induction on the reduction sequence and Lemma 17. Since final configurations of the instrumented language map to final configurations of the source language, this ends the proof.

Similarly, for the second part we proceed by coinduction: since $M \to_I^\infty$, by Lemmas 18 and 17 there exists an M' such that $M \to_I^* M'$, $\lfloor M \rfloor \to_O \lfloor M' \rfloor$ and $M' \to_I^\infty$. By coinduction, we obtain that $\lfloor M' \rfloor \to_O^\infty$, and so $\lfloor M \rfloor \to_O^\infty$. □

D.4 Connecting the instrumented language and dag calculus

Translation of surface expressions into dag calculus Note that the translation trivially extends to the evaluation contexts of the instrumented language. We define a translation $[\![-]\!]^S : \mathcal{E}_I \to \mathcal{M}_I \to \mathcal{E}_{dagcalculus}$.

$$[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!]^S(M) = \texttt{let } x = [\![e_1]\!]^S(M) \texttt{ in } [\![e_2]\!]^S(M)$$
$$[\![\texttt{fun } f\ x \texttt{ is } e \texttt{ end}]\!]^S(M) = \texttt{fun } f\ x \texttt{ is } [\![e]\!]^S(M) \texttt{ end}$$
$$[\![v_1\ v_2]\!]^S(M) = [\![v_1]\!]^S(M)\ [\![v_2]\!]^S(M)$$

The translations for the pairs, variables and numbers are analgous. For future id's we use the state M to map them to the thread-location pairs:

$$[\![f]\!]^S(M) = (t, l) \quad \text{where M(f) = (l, t, \_, \_).}$$

For the other constructs, we have:

```
1  [[future(e)^0]]^S =
2    let l = alloc
3        t = newTd (l := [[e]]^S)
4    in release t; (t, l)
5
6  [[force(v)^0]]^S =
7    let (ft, fl) = [[v]]^S
8    in newEdge (ft, self()); yield(); !fl
```

As mentioned, since all expressions in the well-formed evaluation contexts are surface expressions, we can extend this translation to contexts, giving us $[\![-]\!]^K : \mathcal{K}_I \to \mathcal{K}_{dagcalculus}$.

Translation of well-formed futures into dag calculus Here, we define a translation $[\![-]\!] : \mathcal{E}_I \to \mathcal{L} \times \mathrm{TId} \times \mathcal{M}_I \to \mathcal{S}$, where $\mathcal{S}$ are the states of dag calculus. In case the expression is a surface expression, we simply use the surface translation and add the dag structure:

$$[\![e]\!](l, t, M) = [t \mapsto (l := [\![e]\!]^S, \mathsf{R})], \emptyset, [l \mapsto ()].$$

This leaves us with non-surface cases: the creation and reading of futures. The translation of these constructs takes into account the partial evaluation of the code that their annotations denote. For the `future` construct, we have as follows:

$$[\![K[\texttt{future}(e)^{(1,l)}]]\!](l_f, t_f, M) =$$
$$[t_f \mapsto (l_f := [\![K]\!]^K[\texttt{ft1}([\![e]\!]^S, l)], \mathsf{R})], \emptyset, [l_f \mapsto (), l \mapsto ()]$$
$$[\![K[\texttt{future}(e)^{(2,l,t)}]]\!](l_f, t_f, M) =$$
$$[t_f \mapsto (l_f := [\![K]\!]^K[\texttt{ft2}(l, t)], \mathsf{R}); t \mapsto ([\![e]\!]^S, \mathsf{N})], \emptyset, [l_f \mapsto (), l \mapsto ()]$$

In the above, the helper macros are defined as partial evaluation states of the surface translation of `future`:

```
ft1(e, l) =
  let t = newTd (l := e)
  in release t; (t, l)

ft2(l, t) =
  release t; (t, l)
```

We tackle the `force` operation in a similar manner, taking throughout the definition $(t, l) = [\![v]\!]^S(M)$.

$$[\![K[\texttt{force}(v)^1]]\!](l_f, t_f, M) =$$
$$[t_f \mapsto (l_f := [\![K]\!]^K[\texttt{fr1}(t, l)], \mathsf{X})], \emptyset, [l_f \mapsto ()]$$
$$[\![K[\texttt{force}(v)^2]]\!](l_f, t_f, M) =$$
$$[t_f \mapsto (l_f := [\![K]\!]^K[\texttt{fr2}(t, l, t_f)], \mathsf{X})], \emptyset, [l_f \mapsto ()]$$
$$[\![K[\texttt{force}(v)^3]]\!](l_f, t_f, M) =$$
$$[t_f \mapsto (l_f := [\![K]\!]^K[\texttt{fr3}(l)], \mathsf{X})], \{(t, t_f)\}, [l_f \mapsto ()]$$
$$[\![K[\texttt{force}(v)^4]]\!](l_f, t_f, M) =$$
$$[t_f \mapsto (l_f := [\![K]\!]^K[!\, l], \mathsf{R})], \{(t, t_f)\}, [l_f \mapsto ()]$$

Similar to before, we use helper macros that define partial evaluation of the surface translation of `force`:

```
fr1(ft, fl) =
  newEdge (ft, self()); yield(); !fl

fr2(ft, fl, t) =
  newEdge (ft, t); yield(); !fl

fr3(fl) =
  yield(); !fl
```

Finally we define the translation for a future — and then as a translation of the complete state we take a disjoint union of the translation of the individual futures. We have:

$$[\![(l, t, e, 0)]\!](M) = [\![e]\!](l, t, M)$$
$$[\![(l, t, v, 1)]\!](M) = \emptyset, \emptyset, [l \mapsto [\![v]\!]^S(M)]$$

Definition 8. We say that an instrumented configuration M *matches* a dag Calculus state V, E, σ, written $M \propto V, E, \sigma$ if there exists a dag state V', E' such that $[\![M]\!] = V', E', \sigma$ and a finite map of threads V'' such that $\forall t \in \mathrm{dom}(V'').\ V''(t) = (\mathsf{()}, \mathsf{R})$ and that

$$V' \uplus V'', E', \sigma \twoheadrightarrow_{\mathrm{s}}^{*} V, E, \sigma.$$

Lemma 20. *For any M, V, E, σ, t if* $\mathrm{wf}(M)$, $M \propto V, E, \sigma$, $t \in \mathrm{dom}(V)$ *and* $\mathrm{status}(V(t)) = \mathsf{X}$, *then either $V(t) = (\mathsf{()}, \mathsf{X})$ or there exist f, l and e such that $M(f) = (l, t, e, 0)$.*

Lemma 21. *For any M, V, V', E, E', σ, σ' if* $\mathrm{wf}(M)$, $M \propto V, E, \sigma$ *and $V, E, \sigma \twoheadrightarrow V', E', \sigma'$ then either $M \propto V', E', \sigma'$ or there exists an M' such that $M \to M'$ and $M' \propto V', E', \sigma'$.*

Proof. Consider the reduction $V, E, \sigma \twoheadrightarrow V', E', \sigma'$. It is either a Start reduction, in which case it is an administrative reduction, and $M \propto V', E', \sigma'$, or it happens in some thread $t \in \mathrm{dom}(V)$ such that $\mathrm{status}(V(t)) = \mathsf{X}$. Thus, by Lemma 20 we either have $V(t) = (\mathsf{()}, \mathsf{X})$ or there exists a future f such that $M(f) = (l, t, e, 0)$. In the first case, the only applicable reduction rule is Stop, which is an administrative reduction. This leaves us with the second case.

In this case we have $M(f) = (l, t, e, 0)$ and a reduction in thread t. We can decompose e uniquely into K and e' such that $e = K[e']$ and e' is not a let-binding. If e' is neither a value nor a parallel construct, then it's at $[\![e']\!]^S(M)$ that the reduction occurs on the dag calculus side. Since the translation is entirely structural, we can match it by using the corresponding rule. The case if e' is a value is similar: the redex is the innermost let-binding in K, and can be matched by the let-rule. Finally, for parallel primitives, the location of the redex is based on the annotation: however, the translations of the sequences of annotated primitives are specifically designed so that they correspond to one step of reduction on the dag calculusside. Thus, we can locate the redex, obtain any new threads or locations created on the dag calculus side, and use them in the matching reduction. □

Lemma 22. *For any M, V, V', E, σ, σ' if* $\mathrm{wf}(M)$, $M \propto V, E, \sigma$ *and* $\forall t \in \mathrm{dom}(V').\ V'(t) = (\mathsf{()}, \mathsf{F})$, *then the following simulation holds:*

$$V, E, \sigma \twoheadrightarrow^{*} V', \emptyset, \sigma' \quad \Rightarrow \quad \exists M'.\ M \to_I^{*} M' \wedge M' \propto V', \emptyset, \sigma'$$
$$V, E, \sigma \twoheadrightarrow^{\infty} \quad \Rightarrow \quad M \to_I^{\infty}$$

Proof. Follows from Lemmas D.4 and 1 by the same inductive and coinductive argument as used in the proof of Lemma 19. □

Theorem 3 (Correctness)**.** *Let t be the identifier of the main thread, e be the source program stored in this thread, and l be a designated location in which e stores its final result. Let e_I to be the result of annotating all instances of* `future` *and* `force` *in e with 0. For any number n, final state σ such that $\sigma(l) = n$, and final set of vertices V, assuming that all threads t' in V are finished (i.e.* $\mathrm{status}(V(t')) = \mathsf{F}$*), if*

$$[t \mapsto (l := [\![e_I]\!]^S(\emptyset), \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^{*} V, \emptyset, \sigma$$

then there is a final configuration M such that $[f_0 \mapsto e] \to^* M$ and $M(f_0) = n$. Furthermore, divergence is preserved:

$$[t \mapsto (l := [\![e_I]\!]^S(\emptyset), \mathsf{R})], \emptyset, [l \mapsto ()] \twoheadrightarrow^\infty \quad \Rightarrow \quad [f_0 \mapsto e] \to^\infty .$$

Proof. This theorem follows from the composition of the two simulation diagrams, given by Lemmas 19 and 22. Clearly, $\lfloor e_I \rfloor = e$. Moreover, since $\mathrm{surf}(e_I)$, we also have $[\![[f_0 \mapsto (l, t_0, t, e_I, 0)]]\!](\emptyset) = [t \mapsto (l := [\![e_I]\!]^S(\emptyset), \mathsf{R})], \emptyset, [l \mapsto ()]$, which gives us $[f_0 \mapsto (l, t_0, t, e_I, 0)] \propto [t \mapsto (l := [\![e_I]\!]^S(\emptyset), \mathsf{R})], \emptyset, [l \mapsto ()]$.

For the termination case, by Lemma 22 we obtain that there exists a configuration M such that $[f_0 \mapsto (l, t_0, t, e_I, 0)] \to_I^* M$ and $M \propto V', \emptyset, \sigma'$. Inspecting the translation, we find that this requires that the annotations of all the futures in M are equal to 1, and so M is a final configuration. Since this means that $M(f_0) = (l, t_0, t', v, 1)$ for some t' and v, we learn that $\sigma'(l) = [\![v]\!]^S(M)$, and so, sine $\sigma'(l) = n$, that $v = n$. By Lemma 19 we can now conclude that $[f_0 \mapsto e] \to_O^* \lfloor M \rfloor$, which ends the proof, since $\lfloor M \rfloor(f_0) = n$.

For the nontermination case, from Lemma 22 conclude that $[f_0 \mapsto (l, t_0, t, e_I, 0)] \to_I^\infty$, which, by Lemma 4 implies that $[f_0 \mapsto e] \to_O^\infty$. This ends the proof. □

```
processor_local queue<vertex*> workQueue // bag of vertices
processor_local vertex* current // running vertex
processor_local cont* proc_cont // continuation of the scheduler

void schedulerLoop() // executed by each processor
  while true // termination details omited
    if workQueue.empty()
      // implementation-dependent load balancing
      acquireWork() // blocking call
    current = workQueue.pop()
    current->releaseHandle = increment(current->in)
    if(current->cont == null) // initialize the continuation
      current->cont = new_cont(&enter)
    swap_cont(proc_cont, current->cont) // execute the vertex
    if(current->cont == null) // vertex has finished
      parallelNotify(current->out)
    else // vertex has yielded
      decrement(current->releaseHandle)

void enter() // execute the current vertex, assuming it has never yielded
  current->run()
  current->cont = null // mark vertex finished
  jump_cont(proc_cont)

// ``newTd e'' is short for ``createThread(fun () => e)''
vertex* createThread(runMethod)
  vertex* v = new vertex
  v->run = runMethod
  v->in = new_incounter(v)
  v->out = new_outset()
  v->releaseHandle = increment(v->in)
  return v

void release(vertex* v)
  decrement(v->releaseHandle)

void newEdge(vertex* v1, vertex* v2)
  incounterHandle* h = increment(v2->in)
  bool success = add(v1->out, h)
  if not success // vertex v1 has already completed
    decrement(h) // roll back on edge creation

void yield()
  swap_cont(current->cont, proc_cont)

vertex* self()
  return current
```

Figure 3: Realization of the scheduler loop and primitive operations. Details of load balancing and termination detection are omited.

E Correctness of the implementation

Recall the code of the primitive dag operations from the paper, presented in Figure 3.

We continue with the definitions and theorem statement from Section 6 of the paper.

First, we provide the formal statements of the error rules for the semantics.

$$\frac{V(t) = (K[\texttt{release}\ t'], \mathsf{X}) \qquad \text{status}(V(t')) \neq \mathsf{N}}{V, E, \sigma \twoheadrightarrow \bot}\ \textsc{Release-Err}$$

$$\frac{\begin{array}{c} V(t) = (K[\texttt{newEdge}\ (t_1, t_2)], \mathsf{X}) \\ t_1 \notin \text{dom}(V) \vee t_2 \notin \text{dom}(V) \vee \text{status}(V(t_2)) = \mathsf{F} \vee \\ (\text{status}(V(t_2)) = \mathsf{X} \wedge t \neq t_2) \vee (\{(t_1, t_2)\} \cup E) \text{ is cyclic} \end{array}}{V, E, \sigma \twoheadrightarrow \bot}\ \textsc{NewEdge-Err}$$

To prove the theorem, we proceed in two steps. In the first step, we present a slightly modified set of rules for the dag calculus, proved equivalent to the original one. This allows for a tighter fit with the implementation, in particular accounting for the fact that outgoing edges may be removed incrementally and not atomically. The second key step is the statement of a global invariant, that binds the dag calculus configurations with corresponding states of the machine. We prove that this invariant is preserved by each evaluation step of the machine, some of those steps corresponding to transitions from the (alternative presentation of the) dag calculus.

In the following paragraphs, we first discuss the necessary refinement of the dag calculus, and then describe the invariants used in the second part of the proof and the points of the program at which dag reductions happen. We elide some of the technical details involved in relating the computations, particularly when it comes to the compilation relation, and refer the interested reader to the appendix.

Refined semantics To refine the semantics, we introduce a reduction relation $\twoheadrightarrow'$. Most of the rules carry over directly from the original semantics; we only refine the Stop rule, since the implementation removes edges, one-by-one, potentially in parallel—whereas in the original semantics all the edges are removed in one step. Thus, to obtain a less atomic semantics, we replace this rule with two rules: Stop' and RemEdge'. The first of these changes the thread's status to F (finished), but does not remove any outgoing edges, while the second one removes a single edge from a vertex with status F.

$$\frac{V(t) = (v, \mathsf{X})}{V, E, \sigma \twoheadrightarrow' V[t \mapsto ((\,), \mathsf{F})], E, \sigma}\ \textsc{Stop} \qquad \frac{V(t) = ((\,), \mathsf{F})}{V, (E \uplus (t, t')), \sigma \twoheadrightarrow' V, E, \sigma}\ \textsc{RemEdge}$$

With the refined semantics set up, we can prove that it is equivalent to the original one. This means we gain a more fine-grained and less atomic way of talking about programs that is still equivalent to the original formulation.

Lemma 23. *For any dag calculus state V, E, σ, and a terminal state $V', \emptyset, \sigma'$ such that* $\text{status}(V'(t)) = \mathsf{F}$ *for any $t \in \text{dom}(V')$, we have*

$$V, E, \sigma \twoheadrightarrow^* V', \emptyset, \sigma' \iff V, E, \sigma \twoheadrightarrow'^* V', \emptyset, \sigma'.$$

Proof. The left-to-right direction of this lemma holds trivially, since we can encode Stop in the refined calculus by using Stop' and RemEdge'. For the right-to-left direction, we have a derivation with edge removals scattered after it's marked F. The informal argument for why we can replace these with a bulk removal, is that it is always safe to remove an edge earlier, rather than later, as long as the removal happens after the vertex was marked as finished—which exactly matches the semantics of Stop. □

E.1 The invariant

In order to prove the statement of the correctness theorem, we must now relate the states of the dag calculus (with refined semantics), to the machine states. This is done through an invariant of the following shape, where each of the conjuncts represents a specific sub-invariant defined further on, and where H_I and H_O are two finite maps that bind vertex identifiers to sets of incounter handles. These two maps determine the handles stored in the incounters and outsets of each of the vertices.

$$\begin{aligned}\mathcal{I}((V, E, \sigma), (M, S)) \;\equiv\; & \mathrm{wf}(M) \wedge\; \exists H_I, H_O. \\ & \mathcal{I}_V(M, V, S, H_I, H_O) \wedge \mathcal{I}_Q(V, E, S) \wedge \mathcal{I}_S(M, \sigma) \wedge \\ & \mathcal{I}_E(M, V, E, S, H_I, H_O) \wedge \mathcal{I}_R(M, V, S)\end{aligned}$$

Before we turn to discuss specific parts of the invariant, we briefly describe how formalization of the machine state, and of the continuation, incounter and outset data structures.

Description of the machine state This consists of two parts, the memory, M, which maps addresses to values (C-like records, integers, pointers, etc.), and the per-thread state S. The invariant ensures that the memory is well-formed, i.e., we can split it into five disjoint sections, denoted M_V, M_I, M_O, M_K and M_S. These sections represent the parts of storage used to represent, respectively, vertices, incounters, outsets, continuations and the mutable state of the dag calculus. The state of a processor p is modeled as the tuple $(L_p, Q_p, C_p, K_p, \Gamma_p)$, where L_p denotes the line of program currently evaluated by the processor, Q_p denotes its work-queue, C_p denotes the `current` vertex, K_p denotes the stored continuation `proc_cont`, and Γ_p denotes the environment that maps variables to values. The evaluation proceeds by progressing the L_p counter according to the control-flow, updating the environment Γ_p and other variables as necessary. By convention, we thread the call-stack in the environment, by associating the return address and environment with a `ret` variable in the environment of the callee.

Specification of auxiliary data structures We need to assume specifications of the data structures that the scheduler uses: work queues, continuations, incounters, and outsets. Work queues are simply modeled as sets of vertices. For continuations, we assume a representation predicate $\mathrm{Cont}(K, L, \Gamma)$, which relates a representation of the continuation in memory, K to the pair (L, Γ) of a line number in the program and an environment. The incounter is represented by the predicate $\mathrm{InCounter}(I, H, v)$, which states that the incountered represented in the memory by I has a set of handles H attached to itself and is itself attached to node v (which it will wake when the count is decremented to 0). Finally, the outset is represented by the predicate $\mathrm{OutSet}(O, H)$, which matches the memory representation O with the set of the added elements, H. The functions that operate on these objects use the predicates to specify their effect; we provide these specifications in the technical appendix.

Definition of the sub-invariant $\mathcal{I}_V$ The predicate $\mathcal{I}_V$ ensures that the dag vertices are properly represented as records with appropriate values, according to the status of the vertex. We call r, i, o, h, k the contents of the fields of a vertex record.

$$
\begin{aligned}
&\mathcal{I}_V(M, V, S, H_I, H_O) \equiv \\
&\quad \forall t, e, s.\ V(t) = (e, s).\ \exists r, i, o, h, k.\ M_V(t) = \{r, i, o, h, k\} \wedge \\
&\qquad \text{InHandles}(M_I(i), H_I(t), t) \wedge \text{OutHandles}(M_O(o), H_O(t)) \wedge \\
&\qquad s = \mathsf{N} \Rightarrow (\text{Compile}(e, r) \wedge k = \texttt{null} \wedge h \in H_I(t)) \wedge \\
&\qquad s = \mathsf{R} \Rightarrow \ (\text{if } k = \texttt{null} \text{ then Compile}(e, r) \\
&\qquad\qquad \text{else EvalCont}(e, M_K(k)) \wedge \\
&\qquad\quad (h \in H_I(t) \iff \exists p.\ L_p \in \{11 \ldots 15\} \cup \{18\} \wedge C_p = t)) \wedge \\
&\qquad s = \mathsf{X} \Rightarrow (\exists p. C_p = t \wedge \text{EvalEnter}(e, L_p, \Gamma_p) \wedge \\
&\qquad\quad h \in H_I(t) \wedge (L_p = 21 \Rightarrow k = \texttt{null})) \wedge \\
&\qquad s = \mathsf{F} \Rightarrow k = \texttt{null} \wedge (\exists p.\ C_p = t \wedge L_p \in \{15, 16\} \vee H_O(t) = \emptyset)
\end{aligned}
$$

Above, we ensure the correct representation of the incounters and outsets using the InHandles and OutHandles predicates. The invariant then describes the state of the memory based on what is the status of a given vertex. If the status is N (new), the `run` field, represented by r, should point to a line number where the compiled version of e is stored. The same should be true of a R (released) vertex, if its continuation is not yet set up. If the continuation is set up, on the other hand, it is the continuation that represents the computation e. This is achieved through the EvalCont predicate, which states that the continuation stored at k corresponds to some program state, and that state is in process of evaluating the `enter` function with computation e. Formally, it is defined as shown below, where $\text{Eval}(e, L, \Gamma)$ is a predicate that relates a partially executed dag computation e with a program state (L, Γ). Note that this predicate is related to Compile. (We keep both predicates abstract and reason about them axiomatically.)

$$
\begin{aligned}
&\text{EvalCont}(e, K) \equiv \exists L, \Gamma.\ \text{EvalEnter}(e, L, \Gamma) \wedge \text{Cont}(K, L, \Gamma) \\
&\text{EvalEnter}(e, L_p, \Gamma_p) \equiv \\
&\qquad (L_p = 21 \wedge \exists L.\ \text{Compile}(e, L)) \vee \\
&\qquad (L_p \in \{22, 23\} \wedge e = v) \vee \\
&\qquad (\text{Eval}(e, L_p, \Gamma_p) \wedge \Gamma_p(\texttt{ret}) = (22, [\texttt{v} \mapsto C_p]))
\end{aligned}
$$

We also use EvalEnter if the status of the vertex is X (executing); the only difference is that we relate the program state of a processor p, rather than a stored continuation—which matches the intuition about vertices under execution. Finally, if the status of the vertex is F (finished), it can only have outgoing edges if the outset is about to be processed.

Definition of the sub-invariant $\mathcal{I}_Q$ We now turn to the next invariant, $\mathcal{I}_Q$, which together with $\mathcal{I}_V$ contains most of the important properties. $\mathcal{I}_Q$ describes the state of the work queues, namely it states that the work queues and the vertices just popped

from them always contain exactly those released nodes that have no incoming edges, discounting the vertices that have just yielded and still need to have the artificial in-edge removed.

$$
\begin{aligned}
&\mathcal{I}_Q(V, E, S) \equiv \\
&\quad \biguplus_{p \in \{1 \ldots P\}} (Q_p \uplus \text{if } L_p \in \{11 \ldots 14\} \text{ then } \{C_p\} \text{ else } \emptyset) = \\
&\qquad \{t \in \text{dom}(V) \mid \text{status}(V(t)) = \mathsf{R} \wedge \neg \text{JustYielded}(t, S) \wedge \\
&\qquad\quad \forall t'.\, (t', t) \notin E\} \\
&\text{JustYielded}(t, S) \equiv \exists p \in \{1 \ldots P\}.\, C_p = t \wedge L_p \in \{15, 18\}
\end{aligned}
$$

The next part of the invariant, $\mathcal{I}_S$, states that the values found in the part of memory that models the store, M_S, match the values stored in σ. We use the same relation of compilation for values as in the theorem statement, CompileVal, which ensures that part of the conclusion holds. As with the other compilation relations, we leave this abstract.

$$\mathcal{I}_S(\sigma, M) \equiv \text{dom}(\sigma) = \text{dom}(M) \wedge \forall l \in \text{dom}(\sigma).\, \text{CompileVal}(\sigma(l), M_S(l))$$

Definition of the sub-invariant $\mathcal{I}_E$ The second-to-last part of the invariant, $\mathcal{I}_E$, enforces the properties of edges: each edge, as well as each artificial in-edge is uniquely represented by an incounter handle, and the appropriate handles match the representation of incounters and outsets, H_I and H_O. Since this mostly focuses on necessary separation properties, it is less important to the understanding of the relationship. From the technical standpoint, we define this invariant by quantifying over an extension of E that also contains the matching handle for each of the edges and artificial in-edges, and stating the properties with respect to that relation.

$$
\begin{aligned}
&\mathcal{I}_E(M, V, E, S, H_I, H_O) \equiv \exists \bar{E} \subseteq \text{TId}^2 \times \text{Handle}. \\
&\quad (\forall (t_1, t_2, h) \in \bar{E}, (t_1', t_2', h') \in \bar{E}. \\
&\qquad \text{if } h = h' \text{ then } (t_1 = t_1' \wedge t_2 = t_2') \text{ else } (t_1 \neq t_1' \vee t_2 \neq t_2')) \wedge \\
&\quad (\forall t, t'.\, t \neq t' \Rightarrow (t, t') \in E \iff \exists h.\, (t, t', h) \in \bar{E}) \wedge \\
&\quad (\forall (t_1, t_2, h) \in \bar{E}.\, t_1 \neq t_2 \Rightarrow h \in H_I(t_2) \wedge h \in H_O(t_1)) \wedge \\
&\quad (\forall (t, t, h) \in \bar{E}.\, h \in H_I(t) \wedge M_V(t).h = h) \wedge \\
&\quad (\forall h, t.\, h \in H_O(t) \Rightarrow \exists t' \neq t.\, (t, t', h) \in \bar{E}) \wedge \\
&\quad (\forall h, t.\, h \in H_I(t) \Rightarrow (\exists t'.\, (t', t, h) \in \bar{E} \vee \\
&\qquad \exists p.\, L_p \in \{40 \ldots 42\} \wedge h = \Gamma_p(\mathtt{h}) \wedge \Gamma_p(\mathtt{v2}) = t))
\end{aligned}
$$

Definition of the sub-invariant $\mathcal{I}_R$ The final component of the invariant, $\mathcal{I}_R$, describes additional properties of the per-processor state, i.e., the stored continuation K_p, and the information that lets us distinguish between threads that return to the scheduler because they yielded from the ones whose termination finished. The invariant on K_p states that any processor that evaluates the `enter` function has a stored continuation that represents the `schedulerLoop` function suspended at line 15, while the final lines

state that the status of the thread can be checked by checking if its continuation is set to `null`.

$$\begin{aligned}\mathcal{I}_R(M, V, S) \equiv\ & \forall p \in \{1 \dots P\}. \\ & (\texttt{enter} \in \text{CallStack}(p) \Rightarrow\ \text{Cont}(M_K(K_p), 15, \emptyset)) \wedge \\ & L_p = 15 \Rightarrow (\text{if } M_V(C_p).k = \texttt{null} \text{ then } \text{status}(V(C_p)) = \mathsf{F} \\ & \quad \text{else } \text{status}(V(C_p)) = \mathsf{R})\end{aligned}$$

E.2 Structure of the proof

The proof proceeds by three lemmas that can be combined using induction over the evaluation of the machine state. The first, Lemma 24 relates the initial states of the dag calculus and the machine through an invariant. The second, Lemma 25 provides properties of the final state. Finally, Lemma 26 ensures that the invariant is preserved through the evaluation of the machine.

Lemma 24. *Assume the initial state of the theorem:*

- e_0 *to be a dag calculus expression,*
- t_0 *to be a thread identifier,*
- l_0 *a location for the final result,*
- r_0 *the code pointer to the compiled code for* e_0*, i.e. in the sense that* $\text{Compile}(e_0, r_0)$ *holds (we describe* Compile *later),*
- $V_0 = [t \mapsto (e_0, \mathsf{R})]$*, which describes the initial vertex with body* e_0 *and released status,*
- $E_0 = \emptyset$*, which describes the initial set of edges,*
- $\sigma_0 = [l_0 \mapsto ()]$*, which describes the initial heap,*
- $M_0 = [l_0 \mapsto (), t_0 \mapsto \text{InitVertex}(r_0)]$*, which describes the initial memory state, with a memory cell at location* l_0*, and a representation of the initial vertex with run method* r_0*, fresh incounter and outset, and null continuation and releaseHandle (as described by the auxiliary* InitVertex *operator),*
- S_0*, which describes the initial state of the processors, by asserting that they are entering* `schedulerLoop` *(i.e.,* $\forall p \in \{1 \dots P\}.\ L_p = 6$*, where* L_p *denotes the line of processor* p*), and asserting that all work queues are empty except one that contains exactly* t_0 *(i.e.,* $\exists p \in \{1 \dots P\}.\ Q_p = \{t_0\} \wedge (\forall p' \in \{1 \dots P\}.\ p \neq p' \Rightarrow Q_{p'} = \emptyset)$*, where* Q_p *denotes the work queue of processor* p*).*

Then, $\mathcal{I}((V_0, E_0, \sigma_0), (M_0, S_0))$ *holds.*

Proof. By definition of the invariant: the configurations match each other directly. □

Lemma 25. *For any* M*,* S*,* V*,* E*,* σ *such that* $\mathcal{I}((V, E, \sigma), (M, S))$*, if* S *is a terminal state (i.e.,* $L_p = 9 \wedge Q_p = \emptyset$ *for any processor* p*), then* $I_S(\sigma, M_S)$ *and* (V, E, σ) *is a terminal configuration (i.e.,* $\text{status}(V(t)) = \mathsf{F}$ *for any* $t \in \text{dom}(V)$ *and* $E = \emptyset$*).*

Proof. By definition of the invariant. □

Lemma 26. *For any two machine states* (M, S) *and* (M', S') *such that* $(M, S) \rightarrow (M', S')$ *and for any configuration* (V, E, σ) *such that* $\mathcal{I}((V, E, \sigma), (M, S))$*, if* $\neg\ ((V, E, \sigma) \twoheadrightarrow^* \bot)$ *then there exists a dag configuration* (V', E', σ') *such that* $(V, E, \sigma) \twoheadrightarrow^* (V', E', \sigma')$ *and* $\mathcal{I}((V', E', \sigma'), (M', S'))$*.*

Proof. We need to check that every possible reduction in our program preserves the invariant. Reductions within the body of one of the run methods, including ones where Step should apply, are governed by the Eval predicate. However, this still leaves us with checking all the lines of our code. Among these, most evaluation steps do not require a change of the dag calculus state and the concomitant reduction in the dag calculus; the interesting steps are the ones in which we do need to perform a reduction. We consider these below, and explain in which line of code we perform the matching reduction in the dag calculus.

The high-level justifications of why each of the lines of our code preserves the invariants follow below:

- $L_p = 7$: ensures we only pop from non-empty queues
- $L_p = 9$: by specification of `acquireWork`, ensures queue non-empty
- $L_p = 10$: by specification of `workQueue.pop()`; preserves $\mathcal{I}_Q$
- $L_p = 11$: by specification of `increment`; note the handle obtained is fresh, so invariant $\mathcal{I}_E$ is preserved
- $L_p = 12$: ensures we only restore non-null continuations; invariant $\mathcal{I}_V$ is preserved
- $L_p = 13$: by specification of `new_cont` and `enter`; invariant $\mathcal{I}_V$ is preserved
- $L_p = 14$: by specification of `swap_cont`; apply the Start rule, changing status from R to X; preserves the $\mathcal{I}_V$ invariant
- $L_p = 15$: status has changed from X; checks whether returning vertex has status R or F; exploits $\mathcal{I}_R$ to deduce information
- $L_p = 16$: by specification of `parallelNotify`; ensures that the outset of the current vertex is empty; apply RemEdge' rule for each outgoing edge of the current vertex, noting that these might execute concurrently to other steps.
- $L_p = 18$: by specification of `decrement`; reestablishes the normal state of a released node (i.e., without an artificial in-edge); note status is already R.
- $L_p = 21$: by the relationship of Compile and Eval predicates: $\text{Compile}(e, r) \Rightarrow \text{Eval}(e, r, \emptyset)$
- $L_p = 22$: establishes that the continuation is null in preparation for switching status to F
- $L_p = 23$: by specification of `jump_cont`; the R-F distinction enforced by the invariant established in the previous line; apply the Stop' rule, changing status

from X to F.

- L_p = 27: creates a vertex in M_V; since the vertex is not yet in V, it does not violate the invariants
- L_p = 28: set up the `run` field; from this point the Compile(e, r) holds
- L_p = 29: set up the incounter; from this point on the InHandles predicate holds for the vertex
- L_p = 30: set up the outset; from this point on the OutHandles predicate holds
- L_p = 31: set up the release handle by specification of `increment`; note the disjointness preserved since the handle is fresh
- L_p = 32: by a property of Eval; apply the NewTd rule; invariant $\mathcal{I}_V$ holds for this vertex with status N
- L_p = 35: performs a Release reduction; the released vertex satisfies the invariant, since its `cont` field still contains null; preserves invariant $\mathcal{I}_V$, status changes from N to R
- L_p = 38: by specification of `increment`; the handle `h` is fresh; at this point enter the special branch of $\mathcal{I}_E$
- L_p = 39: by specification of `add`; performs a NewEdge transition; if addition successful, the edge was added, if not — the vertex has state F, so transition is allowed
- L_p = 40: ensures we do not decrement if addition was successful
- L_p = 41: by specification of `decrement`; leaves the special branch of $\mathcal{I}_E$
- L_p = 44: by specification of `swap_cont`; the R-F distinction enforced by `swap_cont` itself; applies the Yield rule, status changes from X to R; preserves $\mathcal{I}_V$
- L_p = 47: by a property of Eval and $\mathcal{I}_V$; performs a Self reduction

□

Theorem 4 (Correctness)**.** *As stated in the paper.*

Proof. First, we deduce from Lemma 24 that the invariant holds. Then, proceed by induction on the execution of the machine. If the derivation is empty, then the configuration of the machine is terminal, and the conclusion follows easily from Lemma 25.

In the inductive case, we can use Lemma 26 on the first step of the derivation. Thus, we establish that the invariant is preserved. Since the correctness of the program is also preserved by the evaluation, we can use the induction hypothesis to end the proof. □