A Separation Logic for Parallel Time Complexity with Work and Span Credits

ALEXANDRE MOINE, New York University, USA
SAM WESTRICK, New York University, USA
JOSEPH TASSAROTTI*, New York University, USA

We present Parcas, a concurrent separation logic for verifying the parallel time complexity of fork-join programs. In order to abstract from the specifics of the machine, time complexity for parallel programs is given in terms of two metrics: the work, measuring the total number of operations, and the span, measuring the longest chain of sequential dependencies. Together, these two metrics determine the running time on any number of processors. For proving bounds on the work and span, Parcas is equipped with work credits and span credits, logical devices that represent permissions to incur costs.

Work credits are a straightforward adaptation of time credits, a standard tool for bounding time complexity of sequential programs, and can be split additively between parallel tasks. Span credits, however, require a fundamentally different treatment. Indeed, the span of the parallel composition of two tasks is the maximum of the span of the two tasks. To account for this, we propose a rule for duplicating span credits at fork points, with each copy tagged by a logical task identifier that restricts which task may spend them. A transfer rule allows unused span credits to be forwarded across sequential compositions to subsequent tasks. The logic is expressive enough to give modular, higher-order specifications for common parallel primitives such as a parallel for loop and a tabulate function. We demonstrate Parcas on several case studies, including parallel prefix sums, parallel merge sort, and a variant of Treiber's lock-free stack that mixes concurrency with parallelism. All the presented results are mechanized in the Rocq prover using the Iris separation logic framework.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; • **Theory of computation** → **Separation logic**; **Program verification**.

Additional Key Words and Phrases: parallelism, separation logic, time complexity

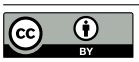
ACM Reference Format:
Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. A Separation Logic for Parallel Time Complexity with Work and Span Credits. *Proc. ACM Program. Lang.* 10, ICFP, Article 281 (August 2026), 30 pages. https://doi.org/10.1145/3828679

1 Introduction

A variety of techniques exist for formally establishing bounds on the resource consumption of programs. One approach is to introduce substructural *credits* or *potentials* into a type system [36] or program logic [3], which must be spent at each point in which a program consumes a corresponding resource. The credits thereby represent a "budget" that upper bounds the resource consumption of a program. This approach has been used to reason about a range of different kinds of resources,

*Also affiliated with Amazon Web Services. This paper does not reflect the views of Amazon Web Services.

Authors' Contact Information: Alexandre Moine, New York University, New York, USA, alexandre.moine@nyu.edu; Sam Westrick, New York University, New York, USA, shw8119@nyu.edu; Joseph Tassarotti, New York University, New York, USA, jt4767@nyu.edu.



ACM 2475-1421/2026/8-ART281
https://doi.org/10.1145/3828679

including running time [32, 18, 50], heap memory under both manual management [36] and garbage collection [53, 43, 46], and stack space [14, 15].

This paper introduces Parcas, a concurrent separation logic with credits for bounding running time cost in fork-join parallel programs. We consider programs written in a parallel functional language, similar in style to NESL [7] and parallel ML-family languages (such as Manticore [23], MaPLe [1], multicore OCaml [57], and Futhark [30]). For such programs, running time cost can be accounted for in terms of two metrics: the *work*, which measures the total number of operations that must be performed, and the *span* (or depth), which measures the longest chain of dependent operations. This latter metric is important because it limits how much speed-up can be obtained from increasing the number of parallel processors used to execute the program. More precisely, when a parallel program with work w and span s is executed on p processors using an appropriate scheduling algorithm, the running time will be $O(\max(w/p, s))$ by Brent's theorem [13, 10], and this bound is optimal up to constant factors. Thus, analyzing both work and span is important to get a complete picture of a parallel program's execution costs.

For reasoning about work in parallel programs, Parcas uses a relatively straightforward adaptation of prior logics' techniques for representing and tracking credits. Since the total work is the sum of the work incurred by each parallel thread, one can additively split credits for work between threads, which is easily represented in separation logic. And, if a thread consumes fewer work credits than its budget allows, we can combine back together the excess credits at the join point.

However, reasoning about span is fundamentally different, because the total span of a parent thread in the fork-join model is the *maximum* of the span of its children. Thus, additively splitting credits across children is not sufficient, and it would be unsound to naively combine together span credits at the join point. While there has been prior work developing a substructural type system using potentials to bound work and span in a language similar to the one we consider [34], that work restricts to first-order pure programs. Moreover, the approach used in that work exploits certain constraints of the type system, which are challenging to translate to the setting of a program logic without imposing severe restrictions on the expressivity of the logic. Other works have developed type systems with sized types for analyzing a form of parallel complexity in the Pi-calculus [4], session typed languages [21], or interaction nets [24], but these execution models are quite different from the data-parallel functional language with shared mutable state that we consider here.

Parcas addresses the challenge of reasoning about span by tagging span credits with a logical *task identifier*, which restricts which threads may spend the credit. When a parent thread forks, it creates duplicate copies of its span credits, one set for each child, with updated identifiers that restrict each child's set of credits to be spent only by that child. Additionally, the logic provides a proof rule to promote the identifier associated with a credit, allowing unused credits to be transferred and then spent by other threads, so long as there is an appropriate ordering between them in the computation graph. Apart from credits, Parcas is a standard separation logic.

We show that Parcas is sound: if a program is verified with w initial work credits and s initial span credits, then it is always safe (no execution ever gets stuck), and the work and span of every execution are bounded by w and s, respectively. Here, the work and span of individual executions are given in terms of a graph structure called a *computation graph* (also known as a *cost graph* [27]), and Parcas provides worst-case bounds on the work and span of such graphs. We note that this approach is sufficient for the applicability of scheduling theorems (such as Brent's theorem discussed above), because every execution under a particular scheduler yields a computation graph within our model, and Parcas gives bounds over all such graphs.

By combining traditional work credits with these new span credits, Parcas is expressive enough to establish asymptotically tight bounds for a number of challenging parallel algorithms. Moreover,

because the logic is higher-order, we are able to give modular, parametric specifications for higher-order parallel operations such as a parallel for loop and a tabulate function. We demonstrate Parcas by verifying cost bounds for a number of examples, including parallel prefix sum, merge sort, and a variant of Treiber's stack, which mixes concurrency with parallelism.

All of the results in this paper have been mechanized in the Rocq proof assistant on top of the Iris separation logic framework [48].

2 Key Ideas

In this section, we present the key ideas of our approach. First, we describe the parallel execution model we consider (§ 2.1). Then, we present a small example illustrating the notions of work and span (§ 2.2). Next, we present work and span credits as well as the weakest precondition of Parcas (§ 2.3). We show the key reasoning rules of work and span credits (§ 2.4) and conclude with a proof sketch of the verification of our small example (§ 2.5).

2.1 Computation Graphs, Work, and Span

We consider programs in an ML-like language with a parallel primitive $e_1 \mid\mid e_2$. The full formal semantics of the language is given later in Section 3, but for now, the high-level idea is that the expression $e_1 \mid\mid e_2$ evaluates the two expressions e_1 and e_2 in parallel, waits for them to terminate their executions on values v_1 and v_2, respectively, and returns an array of size two storing v_1 and v_2. Parallel primitives can be arbitrarily nested.

In order to keep track of the parallel structure of the program, the semantics uses a *computation graph* [9]. A vertex of the computation graph is called a *task* and represents a sequential unit of computation. Every task is annotated with a unique identifier $t \in \mathsf{TaskIds}$. Edges between tasks represent a dependency. Two operations add edges to the computation graph. First, when a task t starts executing a parallel expression $e_1 \mid\mid e_2$, it *forks*. To reflect this operation, we generate two fresh tasks t_1 and t_2, and add edges (t, t_1) and (t, t_2) to the computation graph. Second, when t_1 and t_2 finish their execution, they *join*. To reflect this operation, we generate a fresh task t' for the continuation, and add edges (t_1, t') and (t_2, t') to the computation graph. The computation graph is always acyclic.

To represent operations that have some cost to execute, our programming language is equipped with a ghost operation "tick" that incurs a cost of 1 for the task executing it. We reflect this cost by adding a *weight* to each vertex in the computation graph, reflecting the number of tick operations the task executed. We can then formally define the *work* as the total weight, that is, the sum of the weight of every vertex. The *span* is defined as the heaviest path (or *critical path*), where the weight of a path is the sum of the weights of its vertices. By instrumenting a program appropriately with tick, we can track different notions of cost depending on the application under consideration. For example, for a sorting algorithm, we might add a tick only after each comparison, so that the cost incurred by the program corresponds to the number of comparisons.

Equipped with these weights, our computation graphs form a cost model (the *binary fork-join model*), commonly used for analyzing parallel programs. Because our language has mutable state that can be shared by threads, different interleavings of thread steps can give rise to different computation graphs. When analyzing the cost of a program, we will upper bound the worst-case cost across all of these possible graphs. A worst-case analysis on the work and span allows us to give an upper bound on the asymptotic running time on other parallel cost models, such as PRAM, when the program is executed using an appropriate scheduler [8]. It is important to consider the worst case across all computation graphs for a program, because some graphs may have an unrealistically small cost due to a particular ordering of load and store operations.

```
let f (x: int) : int =
  (if x <= 3
   then (tick; ...) else (tick; ...)); x
          2k ticks total      k ticks total
let example =
  let a = f 1
  let left() = f 2
  let right() = (tick; (f 3 || f 4))
  let (b, (c, d)) = (left() || right())
  let e = f 5
  in a + b + c + d + e
```

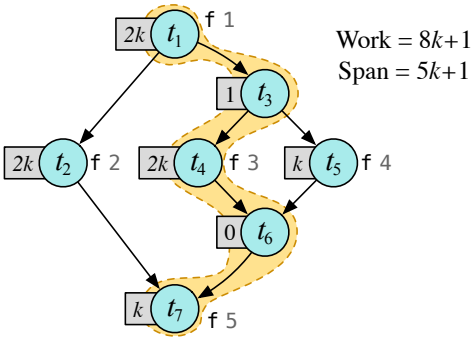


Fig. 1. Small example program and its computation graph. The function f performs either k or $2k$ sequential ticks, depending on its argument; these ticks occur sequentially (no use of || inside f). Vertices in the computation graph ($t_1, t_2, \ldots$) are annotated with their weights (number of ticks executed). The work (total number of ticks executed) is $8k + 1$. The span (weight of the critical path, highlighted in the graph) is $5k + 1$.

2.2 A Small Example

In order to illustrate the notions of computation graph, work, and span, Figure 1 presents a small example. The example program makes use of a function f from **int** to **int**, which incurs sequential cost. Executing f performs either $2k$ ticks if its argument is less than or equal to 3, or k ticks otherwise, with k being an arbitrary constant. The function f is purely sequential and illustrates the fact that the runtime cost of a function can depend on the value of its arguments.

The example program first calls f 1. It then defines two functions: the left function, which calls f 2, and the right function, which itself executes a tick, and calls in parallel f 3 and f 4. The program executes in parallel left() and right(), and finally calls f 5 before returning the sum of all results. The computation graph, drawn on the right of Figure 1, illustrates the parallel structure of the program. Vertices are annotated with their identifiers, and on the right with their weights (number of ticks executed). By counting the calls to f and the ticks they produced, we can see that the total work is $8k + 1$. For the span, we need to identify the heaviest path. The heaviest path of this program is the one highlighted in the figure, which has a weight of $5k + 1$.

2.3 Work and Span Credits and the Weakest Precondition

Credits. Our logic employs two kinds of credits: *work credits* to account for work and *span credits* to account for span. First, n *work credits*, with $n \in \mathbb{N}$ and written $\mathcal{W}(n)$, allow for executing a total of n tick operations. Work credits can be distributed freely between tasks. Second, n *span credits*, with $n \in \mathbb{N}$ and written $\mathcal{S}(t, n)$ for some task t, allow for executing multiple tick operations, perhaps in parallel, which in all contribute no more than weight n to the heaviest path from t. Both work credits and span credits can be split and combined, as shown by the following rules.

WorkSplit
$$\mathcal{W}(n + m) \equiv \mathcal{W}(n) * \mathcal{W}(m)$$

SpanSplit
$$\mathcal{S}(t, n + m) \equiv \mathcal{S}(t, n) * \mathcal{S}(t, m)$$

Work credits can always be freely split and combined and are not associated with any particular task. This aligns with the intuition that work credits only serve to count the total number of tick operations, regardless of which task executes them. For span credits, $\mathcal{S}(t, n)$ allows only the task t to execute n tick operations. This restriction is necessary to ensure that span credits can be associated with paths in the computation. As we will see, span credits can be soundly transferred across subexpression evaluation, which in graph terms corresponds to transferring credits from a task t to some other task t' that is a member of all paths originating from t. For example, span credits can be transferred from a parent task about to fork two children to the task joining them

$$\textsc{Tick}\quad \frac{\mathcal{W}(1) \qquad \mathcal{S}(t,1)}{\mathsf{wp}\ t\ \mathsf{tick}\ \{\lambda\_\, v.\ulcorner v = ()\urcorner\}} \qquad \textsc{Transfer}\quad \frac{\mathcal{S}(t,n) \qquad \mathsf{wp}\ t\ e\ \{\lambda\, t'\, v.\, \mathcal{S}(t',n) \mathrel{-\!\!*} \psi\, t'\, v\}}{\mathsf{wp}\ t\ e\ \{\psi\}}$$

$$\textsc{Par}\quad \frac{\mathcal{S}(t,n) \qquad \forall t_1\, t_2.\, \mathcal{S}(t_1,n) * \mathcal{S}(t_2,n) \mathrel{-\!\!*} \mathsf{wp}\ t_1\ e_1\ \{\psi_1\} * \mathsf{wp}\ t_2\ e_2\ \{\psi_2\}}{\mathsf{wp}\ t\ (e_1 \,||\, e_2)\ \{\lambda\_\, v.\, \exists \ell\, t_1\, t_2\, v_1\, v_2.\ulcorner v = \ell\urcorner * \ell \mapsto [v_1; v_2] * \psi_1\, t_1\, v_1 * \psi_2\, t_2\, v_2\}}$$

Fig. 2. Reasoning rules for credits, and for the tick and the parallel primitives

(recall that a fork operation creates two new tasks, one for each child, and a join operation creates a new task for the continuation). This notion of "transfer" is encoded within the logic as a key reasoning rule, which we discuss in more detail in Section 2.4.

The weakest precondition. In order to relate work and span credits to the execution of a program, we introduce Parcas. Parcas is built on Iris [40], and we follow its syntax. We denote an Iris assertion by φ, a separating conjunction by $\varphi * \varphi'$, and a separating implication (magic wand) by $\varphi \mathrel{-\!\!*} \varphi'$. A meta-logical proposition U is called *pure* and written $\ulcorner U \urcorner$. Assertion equivalence is denoted $\varphi \equiv \varphi'$. Central to our logic is the weakest precondition (WP) modality:

$$\mathsf{wp}\ t\ e\ \{\lambda\, t'\, v.\, \varphi\}$$

Compared to the standard Iris WP, this WP accounts for task identifiers, in a similar manner as Moine et al. [47]. Here, t represents the identifier of the task executing e, which we refer to as the *current task*. The postcondition has the form $\lambda\, t'\, v.\ \varphi$, binding two variables in φ: the *end task* t' and the result value v. The end task identifier characterizes the task when the computation is completed. The current and end task identifiers need not coincide: in the `example` program from Section 2.2, at the time of executing the program, the current identifier is t_1 and the end identifier is t_7. When the details of the bound variables in the postcondition can be omitted, we abbreviate it using the metavariable ψ.

The formal statement of the soundness theorem of Parcas will be presented later in Section 4.3, but intuitively, it says that if the user verified a program e with w initial work credits and s initial span credits, that is, if $\mathcal{W}(w) * \mathcal{S}(t_0, s) \vdash \mathsf{wp}\ t_0\ e\ \{\lambda\_\_.\ \top\}$ holds for some initial task t_0, then (1) the work of e is bounded by w, (2) the span of e is bounded by s, and (3) e is safe (that is, never gets stuck, a standard property guaranteed by separation logic). Note that when applying the soundness theorem, we fix the initial work and span credits that will be available to verify e up front— there is no way to create additional credits during the proof.

2.4 Key Reasoning Rules for Credits

Figure 2 presents reasoning rules associated with credits—the other reasoning rules of Parcas are mostly standard and presented in Section 4.1. We present reasoning rules as inference rules where premises are implicitly joined by $*$, and the horizontal bar stands for entailment.

Tick and Par. Tick shows that a tick operation consumes both a work credit and a span credit, with a matching identifier. This follows from the fact that the tick operation incurs a cost of 1 for the task executing it.

Par constitutes the crux of our approach. At first sight, Par has the ingredients of the standard separation logic rule for a parallel composition: to verify that $e_1 \,||\, e_2$ is correct, the user has to verify e_1 and e_2 separately. Par differs from the standard rule in two ways. First, the rule is indexed by the current task t, and universally quantifies over the identifiers t_1 and t_2 of the sub-tasks. Second, the precondition of Par consumes span credits $\mathcal{S}(t,n)$ and "duplicates" them for the two

generated tasks t_1 and t_2. Indeed, the precondition requires the user to verify the WP of e_1 at some identifier t_1 and of e_2 at some identifier t_2, while having access to span credits $\mathcal{S}(t_1, n)$ and $\mathcal{S}(t_2, n)$. The postcondition of Par reflects that the parallel primitive returns an array containing the result of the two evaluated expressions, and that their respective postconditions hold.

Why is this "duplication" of span credits sound? The key is that span credits cannot be transferred between unrelated tasks. When the parent task t (with a budget $\mathcal{S}(t, n)$) forks into two child tasks t_1 and t_2, the two children are given two separate budgets $\mathcal{S}(t_1, n)$ and $\mathcal{S}(t_2, n)$, and these two budgets cannot intermix. Par therefore is able to independently bound the span of both e_1 and e_2 by n, which in turn bounds the span of $(e_1 \mid\mid e_2)$ by n as well.

The reader might wonder why Par does not mention the *work* credits needed for the two expressions e_1 and e_2. The reason is that, because we are working in a weakest precondition calculus, any work credits needed for executing the two expressions can be included as part of the proof of the second premise. For example, suppose that we have previously proved a specification for e_i (for $i \in \{1, 2\}$) of the form $\forall t_i.\ \mathcal{W}(K) \mathrel{-\!\!*} \mathcal{S}(t_i, C) \mathrel{-\!\!*} \mathsf{wp}\ t_i\ e_i\ \{\psi_i\}$, which shows that e_i requires K work credits. Then by combining this specification with $\mathcal{W}(2 \times K)$, and by using WorkSplit, we can derive $\forall t_1\ t_2.\ \mathcal{S}(t_1, n) * \mathcal{S}(t_2, n) \mathrel{-\!\!*} \mathsf{wp}\ t_1\ e_1\ \{\psi_1\} * \mathsf{wp}\ t_2\ e_2\ \{\psi_2\}$, matching the second premise of Par. In contrast, the span credits need to appear explicitly in Par because they must be transferred to the child threads.

The need for a transfer rule. While Par allows a parent task to give span credits to its children, there is no way for a child task to give span credits back to its parent in a form they can use. In particular, although t_1 and t_2 can put unused credits $\mathcal{S}(t_1, n_1)$ and $\mathcal{S}(t_2, n_2)$ into their postconditions ψ_1 and ψ_2, which are passed back to the parent at the join point, the parent has no way to spend these credits or any leftover credits $\mathcal{S}(t, n')$ it may have had from before the fork. This is because the continuation of the parent task, after the join operation, gets a fresh identifier t', so these credits with different identifiers cannot be used. A restriction on only spending span credits with the appropriate tag is necessary to ensure soundness of the logic—since credits were duplicated as part of Par, we must be careful about which task can use them—but without some other mechanism, this restriction is too severe. To see why, consider the following Sequence rule for a sequential composition, which is the same as the standard rule found in similar program logics, except for the addition of the task identifiers.[1]

$$\textsc{Sequence}\ \frac{\mathsf{wp}\ t\ e_1\ \{\lambda\, t'\, v.\, \mathsf{wp}\ t'\ ([v/x]e_2)\ \{\psi\}\}}{\mathsf{wp}\ t\ (\mathsf{let}\, x\ =\ e_1\, \mathsf{in}\, e_2)\ \{\psi\}}$$

This rule allows for verifying a sequential composition $\mathsf{let}\, x\ =\ e_1\, \mathsf{in}\, e_2$ by first verifying e_1 and then e_2, in which we substitute x with v, the result of e_1. However, notice the identifiers: e_1 is being verified at current task t, while e_2 is being verified at t', which is the end task of e_1. These identifiers might potentially be different if e_1 contained a parallel composition. As a result, with the rules we have discussed so far, if there are some leftover span credits $\mathcal{S}(t, n)$ after verifying e_1, they cannot be used to verify e_2.

To solve this problem, we provide Transfer, which allows for transferring span credits from the current task to the end task. Reading from bottom to top, the Transfer rule says: if the current task is t executing e with postcondition ψ, and the user has $\mathcal{S}(t, n)$ span credits, then it suffices to verify e but with a postcondition enriched with $\mathcal{S}(t', n)$ span credits, where t' is the end task.

For some intuition as to why this rule is sound, note that in the computation graph (1) the task t *dominates* the task t', meaning that every path that passes through t' must pass through t, and (2) the task t' is an active task, meaning that at this point in the proof, it has no successor.

[1]We derive Sequence from Bind and LetVal presented in Section 4.1.

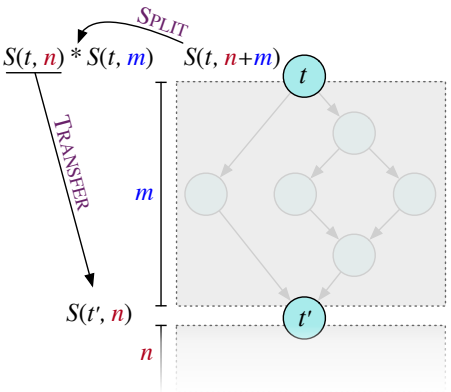


Fig. 3. Illustration of using the TRANSFER rule to bound the span of a region of the computation graph. All paths from t must pass through t', so the weight of the heaviest path from t is the sum of the heaviest path in the grey region and the heaviest path from t'.

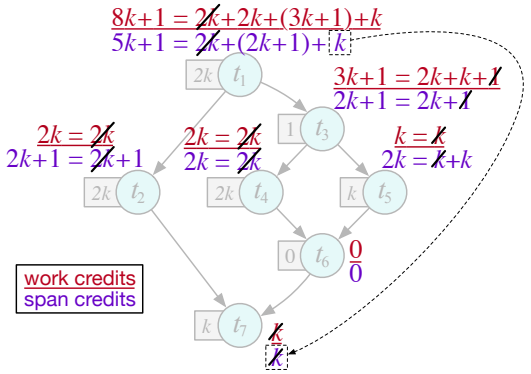


Fig. 4. Accounting for all credits in the verification of Figure 1. The figure can be read from top to bottom: we start with $8k+1$ work credits and $5k+1$ span credits at t_1. Work credits are shown in red (underlined), and span credits in purple. Crossed-out credits are consumed at the associated task. The dashed arrow indicates a TRANSFER.

The first property ensures that the heaviest path starting from t' is a suffix of the heaviest path starting from t, and hence that the weight of the latter bounds the weight of the former. The second property ensures that we don't have to worry about possible paths starting from t' yet, there are none. Figure 3 illustrates how the respective credits in TRANSFER cover different portions of the computation graph. Combining SEQUENCE with TRANSFER, we can easily prove the following rule that exactly solves the problem with the initial version of SEQUENCE.

$$\frac{\mathcal{S}(t, n) \qquad \mathsf{wp}\ t\ e_1\ \{\lambda\, t'\, v.\, \mathcal{S}(t', n) \mathrel{-\!\!*} \mathsf{wp}\ t'\ ([v/x]e_2)\ \{\psi\}\}}{\mathsf{wp}\ t\ (\mathsf{let}\, x\ =\ e_1\, \mathsf{in}\, e_2)\ \{\psi\}}$$

Here, the $\mathcal{S}(t, n)$ credits initially held by t are not needed for the execution of e_1, so they are transferred to use in verifying t'. (The $\mathrel{-\!\!*}$ in the postcondition of the premise means that when proving the weakest precondition about $[v/x]e_2$, we may assume access to $\mathcal{S}(t', n)$ credits.)

The usage pattern for organizing span credits before executing a parallel primitive (or any library function that may call the parallel primitive) is to split the available span credits in two assertions: one will be given to PAR so that children have span credits to spend, and the other will be transferred to the continuation with TRANSFER.

2.5 Verification of our Small Example

With the rules presented in Section 2.4 (as well as the other rules of Parcas presented in Section 4.1), we can verify the work and span bounds of the program from Section 2.2. First, we specify the auxiliary sequential function f as follows:

$$\frac{(\ulcorner \mathsf{n} \le 3 \urcorner * \mathcal{W}(2k) * \mathcal{S}(t, 2k)) \quad \vee \quad (\ulcorner \mathsf{n} > 3 \urcorner * \mathcal{W}(k) * \mathcal{S}(t, k))}{\mathsf{wp}\ t\ (\mathsf{f}\ \mathsf{n})\ \{\lambda\, t'\, v.\, \ulcorner t' = t \wedge v = \mathsf{n} \urcorner\}}$$

In the above specification, note that the task t is (implicitly) universally quantified, meaning that this specification can be used by any task. Moreover, this specification enforces that f is sequential: the end task identifier t' must be equal to the current task identifier t. Second, we verify the main

Values $\mathcal{V}$	$v ::= () \mid b \in \{\mathsf{true}, \mathsf{false}\} \mid i \in \mathbb{Z} \mid \ell \in \mathcal{L} \mid \hat{\mu} f\, x.\, e$			
Primitives	$\bowtie ::= + \mid - \mid \times \mid \div \mid \mathsf{mod} \mid == \mid < \mid \leq \mid > \mid \geq \mid \vee \mid \wedge$			
Expressions	$e ::= v, w$	*value*	tick	*cost operation*
	x	*variable*	alloc e e	*array allocation*
	$\mathsf{let}\, x = e\, \mathsf{in}\, e$	*sequencing*	$e[e]$	*array load*
	$\mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e$	*conditional*	$e[e] \leftarrow e$	*array store*
	$\mu f\, x.\, e$	*abstraction*	length e	*array length*
	$e\, e$	*call*	$e \mid\mid e$	*parallelism*
	$e \bowtie e$	*primitive operation*	CAS $e\, e\, e$	*compare-and-swap*
Contexts	$K ::= \mathsf{let}\, x = \square\, \mathsf{in}\, e$	$\mid \mathsf{if}\, \square\, \mathsf{then}\, e\, \mathsf{else}\, e$	$\mid \mathsf{alloc}\, e\, \square$	$\mid \mathsf{alloc}\, \square\, v$ $\mid e[\square]$
	$\mid \square[v]$	$\mid e[e] \leftarrow \square$	$\mid e[\square] \leftarrow v$	$\mid \square[v] \leftarrow v$ $\mid \mathsf{length}\, \square$
	$\mid e\, \square$	$\mid \square\, v$	$\mid e \bowtie \square$	$\mid \square \bowtie v$
	$\mid \mathsf{CAS}\, e\, e\, e\, \square$	$\mid \mathsf{CAS}\, e\, e\, \square\, v$	$\mid \mathsf{CAS}\, e\, \square\, v\, v$	$\mid \mathsf{CAS}\, \square\, v\, v\, v$

Fig. 5. Syntax of ParLang

program as follows:

$$\frac{\mathcal{W}(8 \times k + 1) \;*\; \mathcal{S}(t_1, 5 \times k + 1)}{\mathsf{wp}\; t_1\; (\texttt{example}\; ())\; \{\lambda \_\, v.\, \ulcorner v = 15 \urcorner\}}$$

The proof goes as follows, with Figure 4 illustrating the flow of credits throughout the argument. First, we split work and span credits and consume $2k$ of both, to account for the cost of `f 1`. Next, we face a par primitive, to execute `left()` and `right()` in parallel. We observe that the continuation of the program is a call to `f 5`, which has a cost of k. Hence, we split the remaining $3k + 1$ span credits into k for the continuation (which we transfer with Transfer) and $2k + 1$ for the upcoming primitive parallel operation. While we're at it, we also split the remaining $6k + 1$ work credits into k for the continuation (which we transfer with a standard Frame rule) and $5k + 1$ for the parallel operation. We then apply Par, giving $2k + 1$ span credits to the two sub-tasks. We also split the $5k + 1$ work credits into $2k$ for `left` and $3k + 1$ for `right`. We first verify `left()`, which calls `f 2` and consumes $2k$ work credits and $2k$ span credits (which we have, since we have $2k + 1$ span credits available). We then verify `right()`, which executes a tick (consuming 1 work credit and 1 span credit) and calls in parallel `f 3` and `f 4`. We again use Par and conclude this subgoal. We finally go back to verify the continuation call to `f 5` and conclude.

Now that we have seen the key ideas behind Parcas we turn to a more formal presentation of the language under consideration (Section 3) and the proof rules (Section 4).

3 Syntax and Semantics

We call the formal language we study ParLang. ParLang is a call-by-value lambda calculus with mutable state and parallelism. We first present its syntax (§3.1) and then its semantics (§3.2). ParLang is similar to HeapLang, the language that ships with Iris, except that it implements structured parallelism instead of fork-based concurrency, and provides a tick primitive for cost accounting.

3.1 Syntax

Figure 5 presents the syntax of ParLang. A value $v \in \mathcal{V}$ is either the unit value $()$, a Boolean $b \in \{\mathsf{true}, \mathsf{false}\}$, an idealized integer $i \in \mathbb{Z}$, a location ℓ from an infinite set of locations $\mathcal{L}$, or a recursive function $\hat{\mu} f\, x.\, e$. A computation in ParLang is described by an expression e, whose syntax is mostly standard. Recursive functions are written $\mu f\, x.\, e$ and will evaluate to their value counterpart. We write $\lambda x.\, e \triangleq \mu \_\, x.\, e$ for non-recursive functions, and define functions with multiple arguments as a chain of function constructors. The tick operation is used for cost accounting. Mutable state is

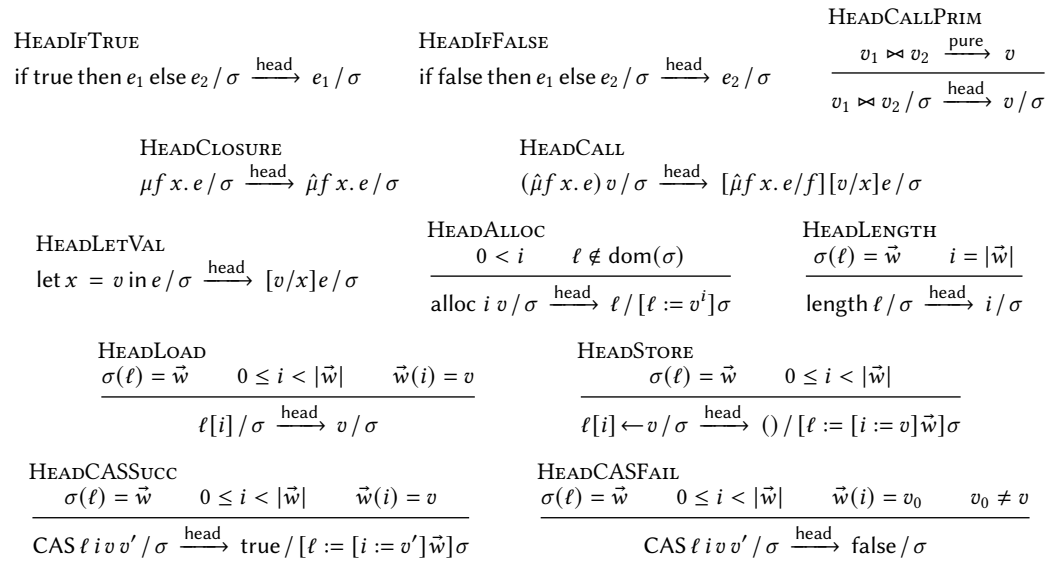


Fig. 6. Head reduction relation

available through arrays. The expression alloc e_1 e_2 allocates an array of size e_1 initialized with whatever value e_2 evaluates to in each cell. For the sake of simplicity at the logical level, arrays are initialized, but the cost of initialization is not accounted for. This is unrealistic for large arrays, where initialization typically costs $O(N)$ work and $O(\log N)$ span for an array of size N. If a verified program relies on the fact that the array is initialized, then cost should be adequately incurred using tick operations placed before the allocation point. Parallelism is available with a primitive $e_1 \,||\, e_2$, which evaluates the two expressions in parallel and returns their results as an array of size 2. ParLang also has a primitive compare-and-swap instruction CAS e_1 e_2 e_3 e_4, which targets an array entry and has 4 parameters: the array location, the offset into the array, the old value, and the new value.

An evaluation context K describes an expression with a hole $\square$ and dictates the right-to-left evaluation order of ParLang.

3.2 Semantics

Head reduction relation. Figure 6 presents the head reduction relation $e / \sigma \xrightarrow{\text{head}} e' / \sigma'$, describing a single step of expression e with store σ into expression e' and store σ'. A store is a map from locations to arrays, which are modeled as lists of values. We write $\emptyset$ for the empty store and $\sigma(\ell)$ for the array at location ℓ in σ. To insert or update a location ℓ with array $\vec{v}$ in store σ, we write $[\ell := \vec{v}]\sigma$, and similarly write $[i := w]\vec{v}$ to update offset i with value w in array $\vec{v}$. The length of an array $\vec{v}$ is written as $|\vec{v}|$, and v^i represents an array of size i initialized with value v in each cell.

The reduction rules are standard. HeadIfTrue and HeadIfFalse handle conditionals. HeadCallPrim evaluates primitive operations. HeadClosure converts a function expression into a function value. HeadCall performs function application. HeadLetVal substitutes the bound value into the body. HeadAlloc allocates an array initialized with the given value and returns its location, which is selected nondeterministically. HeadLoad and HeadStore perform array loads and stores, respectively. HeadLength returns the length of an array. HeadCASSucc and HeadCASFail perform an atomic compare-and-swap.

$$\textsc{SchedHead}\ \frac{e \,/\, \sigma \xrightarrow{\text{head}} e' \,/\, \sigma'}{e \,/\, t \,/\, \sigma \,/\, c \,/\, g \xrightarrow{\text{sched}} e' \,/\, t \,/\, \sigma' \,/\, c \,/\, g} \qquad \textsc{SchedTick}\ \frac{c' = [t := c(t) + 1]c}{\text{tick} \,/\, t \,/\, \sigma \,/\, c \,/\, g \xrightarrow{\text{sched}} () \,/\, t \,/\, \sigma \,/\, c' \,/\, g}$$

$$\textsc{SchedFork}\ \frac{t_1, t_2 \notin \text{vertices}(g) \qquad c' = [t_2 := 0][t_1 := 0]c \qquad g' = g \cup \{(t_0, t_1), (t_0, t_2)\}}{e_1 \,||\, e_2 \,/\, t_0 \,/\, \sigma \,/\, c \,/\, g \xrightarrow{\text{sched}} e_1 \,||\, e_2 \,/\, t_1 \otimes t_2 \,/\, \sigma \,/\, c' \,/\, g'}$$

$$\textsc{SchedJoin}\ \frac{\ell \notin \text{dom}(\sigma) \qquad t_3 \notin \text{dom}(c) \qquad t_3 \notin \text{vertices}(g) \qquad g' = g \cup \{(t_1, t_3), (t_2, t_3)\}}{v_1 \,||\, v_2 \,/\, t_1 \otimes t_2 \,/\, \sigma \,/\, c \,/\, g \xrightarrow{\text{sched}} \ell \,/\, t_3 \,/\, [\ell := [v_1; v_2]]\sigma \,/\, [t_3 := 0]c \,/\, g'}$$

$$\textsc{StepSched}\ \frac{e \,/\, T \,/\, \sigma \,/\, c \,/\, g \xrightarrow{\text{sched}} e' \,/\, T' \,/\, \sigma' \,/\, c' \,/\, g'}{e \,/\, T \,/\, (\sigma, c, g) \longrightarrow e' \,/\, T' \,/\, (\sigma', c', g')} \qquad \textsc{StepBind}\ \frac{e \,/\, T \,/\, S \longrightarrow e' \,/\, T' \,/\, S'}{K\langle\!\langle e \rangle\!\rangle \,/\, T \,/\, S \longrightarrow K\langle\!\langle e' \rangle\!\rangle \,/\, T' \,/\, S'}$$

$$\textsc{StepParL}\ \frac{e_1 \,/\, T_1 \,/\, S \longrightarrow e_1' \,/\, T_1' \,/\, S'}{e_1 \,||\, e_2 \,/\, T_1 \otimes T_2 \,/\, S \longrightarrow e_1' \,||\, e_2 \,/\, T_1' \otimes T_2 \,/\, S'} \qquad \textsc{StepParR}\ \frac{e_2 \,/\, T_2 \,/\, S \longrightarrow e_2' \,/\, T_2' \,/\, S'}{e_1 \,||\, e_2 \,/\, T_1 \otimes T_2 \,/\, S \longrightarrow e_1 \,||\, e_2' \,/\, T_1 \otimes T_2' \,/\, S'}$$

Fig. 7. Reduction under a context and parallelism

Scheduler reduction relation. During execution, each task has an associated *task identifier*. To keep track of the identifier of each task and whether it started executing or not, we follow Westrick et al. [60] and use a *task tree*. A task tree $T \triangleq t \mid T_1 \otimes T_2$ is either a leaf labeled with a task identifier t, or a parallel composition of two sub-trees T_1 and T_2. Leaves represent individual tasks, while internal nodes represent `par` expressions whose children are currently executing in parallel. Vertices of task trees and their dependencies are recorded in a *computation graph* g, modeled as a set of edges. Weights of vertices (their number of executed tick operations) are recorded by a *tick map* c. The upper part of Figure 7 presents the scheduling relation $e \,/\, T \,/\, \sigma \,/\, c \,/\, g \xrightarrow{\text{sched}} e' \,/\, T' \,/\, \sigma' \,/\, c' \,/\, g'$ reducing expression e with a task tree T, a store σ, a tick map c, and a computation graph g. SchedHead performs a head step. SchedTick increments the tick count for the current task and returns the unit value. SchedFork converts a leaf executing a parallel primitive into a node with two leaves. In the computation graph, these two leaves form two fresh vertices with edges from the original task and an initial cost of 0. SchedJoin joins a node with two leaves where both sides reached a value into a new leaf, returning a location pointing to an array storing the two values. In the computation graph, this leaf forms a fresh vertex with edges from the two previous tasks.

Note that these joins are the only intended primitive for a thread to wait or synchronize with another thread without generating additional cost. Because the language has shared mutable state, programs can also implement a form of busy-waiting between threads by spinning on a shared reference. However, it is important to include a tick in any such loop, or else the resulting computation graph would have an unrealistic form of cost-free synchronization between threads.

Main reduction relation. The lower part of Figure 7 presents the main reduction relation, written $e \,/\, T \,/\, S \longrightarrow e' \,/\, T' \,/\, S'$, where $S = (\sigma, c, g)$ abbreviates the shared state. StepSched lifts a scheduling step to the main reduction relation. StepBind performs a step under an evaluation context. StepParL and StepParR implement parallelism: these two rules allow the main reduction relation to nondeterministically step either the left or right side of an active parallel composition. We write the reflexive-transitive closure of the reduction relation as $e \,/\, T \,/\, S \longrightarrow^* e' \,/\, T' \,/\, S'$.

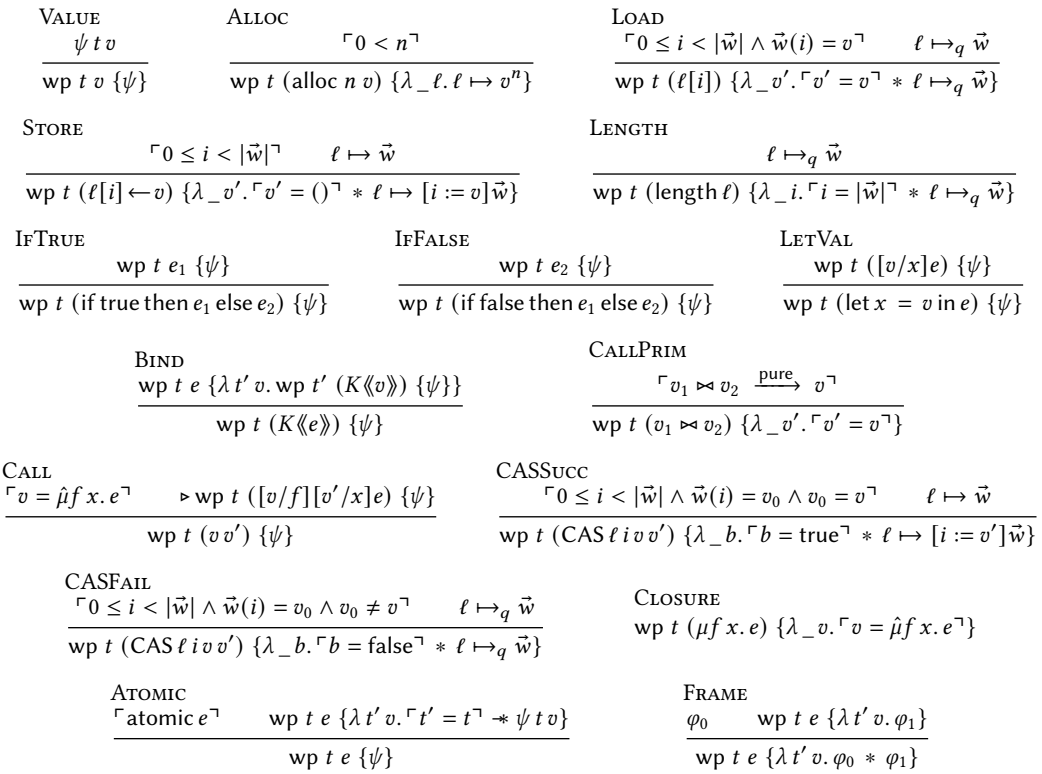


Fig. 8. Reasoning rules for base constructs, except for the tick and the parallel primitives

4 The Program Logic and its Soundness Theorem

In this section, we present the non-credit reasoning rules of Parcas that were not previously shown (§4.1), and explain how we derive Transfer from more primitive rules (§4.2). We next present the formal statement of the soundness theorem of Parcas (§4.3). We then turn to the proof of this theorem and its technical details: we define the weakest precondition (§5.2), and set up the state interpretation predicate relating physical state to logical state (§5.3), focusing in particular on our handling of credits (§5.4).

4.1 Reasoning Rules of Parcas

Figure 8 presents reasoning rules for basic constructs, except for the tick and the parallel primitive that were explained previously (see §2.4). Apart from the task identifier, they are mostly standard. Parcas makes use of fractional [12, 11] points-to assertions of the form $\ell \mapsto_q \vec{w}$, where q denotes a positive fraction less than or equal to 1. When $q = 1$ we write $\ell \mapsto \vec{w}$. Value ensures that when the execution reaches a value, the postcondition is satisfied. Alloc, Load, Store, and Length are the usual array operations, with Load and Length requiring only fractional ownership. IfTrue and IfFalse handle conditionals. LetVal substitutes a value in a let binding. Bind is the sequencing rule for evaluation contexts. CallPrim handles primitive operations, and Call handles function application, with a later modality $\triangleright$ [40]. Closure creates function values. CASSucc and CASFail describe a successful and a failed CAS, respectively.

$$\textsc{GenerateTransferable}\ \frac{\mathsf{wp}\ t\ e\ \{\lambda\, t'\, v.\, \mathsf{transferable}\ t\ t' \mathrel{-\!\!*} \psi\ t'\ v\}}{\mathsf{wp}\ t\ e\ \{\psi\}} \qquad \textsc{PrimitiveTransfer}\ \frac{\mathsf{transferable}\ t\ t' \qquad \mathcal{S}(t,n) \qquad (\mathcal{S}(t',n) \mathrel{-\!\!*} \mathsf{wp}\ t'\ e\ \{\psi\})}{\mathsf{wp}\ t'\ e\ \{\psi\}} \qquad \textsc{End}\ \frac{\mathsf{wp}\ t\ e\ \{\lambda\, t'\, v.\, \mathsf{wp}\ t'\ v\ \{\psi\}\}}{\mathsf{wp}\ t\ e\ \{\psi\}}$$

$$\textsc{TF-Refl}\ \mathsf{transferable}\ t\ t \qquad \textsc{TF-Trans}\ \frac{\mathsf{transferable}\ t\ t'' \qquad \mathsf{transferable}\ t''\ t'}{\mathsf{transferable}\ t\ t'} \qquad \textsc{TF-Diamond}\ \frac{\mathsf{transferable}\ t_l\ t_l' \qquad \mathsf{transferable}\ t_r\ t_r' \qquad \mathsf{is\_fork}\ t\ t_l\ t_r \qquad \mathsf{is\_join}\ t_l'\ t_r'\ t'}{\mathsf{transferable}\ t\ t'}$$

Fig. 9. Reasoning principles for transferable

For conciseness, the above rules do not mention that sequential operations preserve the identifier of the current task. Atomic addresses this: for any atomic operation (*i.e.*, one that reduces to a value in exactly one step), the end task is the same as the current task.

The WP of Parcas also satisfies other standard rules of separation logic, such as the Frame rule.

4.2 Inside the Transfer Rule

The Transfer rule is derived from more primitive and slightly more flexible reasoning rules, presented in Figure 9. The rules make use of the persistent (hence duplicable) assertion $\mathsf{transferable}\ t\ t'$, which asserts that span credits can be transferred from t to t'. The transferable assertion is reflexive (TF-Refl) and transitive (TF-Trans).

GenerateTransferable allows the user to generate permission to transfer span credits from the current task t to the end task t', by producing an assertion $\mathsf{transferable}\ t\ t'$ in the postcondition. PrimitiveTransfer actively transfers credits from one task to another: it requires a witness $\mathsf{transferable}\ t\ t'$, as well as span credits $\mathcal{S}(t,n)$ while facing a weakest precondition for t', consumes these two previous assertions and gives back $\mathcal{S}(t',n)$ span credits for pursuing the proof. Together, GenerateTransferable and PrimitiveTransfer are more flexible than Transfer. They allow for postponing the transfer of span credits to a later point in the proof, rather than manually applying Transfer before verifying expressions involving a parallel primitive. End is a technical rule that, read from bottom to top, creates at will a trivial weakest precondition on a value in the postcondition. End is useful since PrimitiveTransfer requires the goal to be a weakest precondition.

Equipped with these rules, we derive Transfer as follows: first we apply End to generate a trivial WP in the postcondition, then we apply GenerateTransferable to generate an assertion $\mathsf{transferable}\ t\ t'$, next we apply Frame to move span credits in the postcondition, and then move the reasoning to the postcondition, where we apply PrimitiveTransfer and Value to conclude.

The lower part of Figure 9 presents the definition of the assertion $\mathsf{transferable}\ t\ t'$, which is defined inductively as a least fixed point of the rules shown in that figure. TF-Refl and TF-Trans state that the relation is reflexive and transitive, respectively. The most complex rule is TF-Diamond, which in turn uses more primitive assertions $\mathsf{is\_fork}\ t\ t_l\ t_r$ and $\mathsf{is\_join}\ t_l'\ t_r'\ t'$, which describe the corresponding properties of the structure of the computation graph. In particular, this rule captures the case of a "diamond" in the computation graph: the task t forks into two tasks t_l and t_r, which execute, possibly forking and joining, ultimately reaching tasks t_l' and t_r' that join into t'. Although "end users" of the logic will typically use the simple transfer rule we saw in Section 2, deriving this rule out of these more primitive concepts simplifies the soundness proof that we carry out later.

4.3 Soundness Theorem

To state the soundness theorem of Parcas, we first explain how to express bounds on the work and span of programs, and then define what it means for an expression to be safe (*i.e.*, to not crash).

$$\textsc{WorkBound}\ \frac{\mathsf{sumall}(c) \le n}{\mathsf{WorkBound}\, c\, n} \qquad \textsc{PathNil}\ \frac{c(t) = n}{g, c \vdash t \rightsquigarrow_n t} \qquad \textsc{PathCons}\ \frac{(t_1, t_2) \in g \qquad c(t_1) = n \qquad g, c \vdash t_2 \rightsquigarrow_{n'} t_3}{g, c \vdash t_1 \rightsquigarrow_{n+n'} t_3}$$

$$\textsc{SpanBound}\ \frac{\forall t_1\, t_2\, n'.\quad g, c \vdash t_1 \rightsquigarrow_{n'} t_2 \implies n' \le n}{\mathsf{SpanBound}\, g\, c\, n}$$

Fig. 10. Definitions of the WorkBound and SpanBound predicates

$$\textsc{RedSched}\ \frac{e \,/\, T \,/\, S \xrightarrow{\text{sched}} e' \,/\, T' \,/\, S'}{\mathsf{AllRed}\, e\, T\, S} \qquad \textsc{RedCtx}\ \frac{\mathsf{AllRed}\, e\, T\, S}{\mathsf{AllRed}\, (K\langle\!\langle e \rangle\!\rangle)\, T\, S}$$

$$\textsc{RedPar}\ \frac{(e_1 \notin \mathcal{V} \implies \mathsf{AllRed}\, e_1\, T_1\, S) \qquad (e_2 \notin \mathcal{V} \implies \mathsf{AllRed}\, e_2\, T_2\, S) \qquad (e_1 \notin \mathcal{V} \ \vee\ e_2 \notin \mathcal{V})}{\mathsf{AllRed}\, (e_1 \,||\, e_2)\, (T_1 \otimes T_2)\, S}$$

$$\textsc{Safe}\ \frac{(e \in \mathcal{V} \wedge T \in \mathcal{L}) \quad \vee \quad \mathsf{AllRed}\, e\, T\, S}{\mathsf{Safe}\, e\, T\, S}$$

Fig. 11. Reducibility and safety of a configuration

Formal bounds on work and span. Figure 10 presents the WorkBound and SpanBound predicates. The property WorkBound $c\ n$ holds when the work—that is, the sum of all ticks recorded in the tick map c—is at most n (WorkBound).

Bounding the span requires bounding the cost along any execution path in the task graph. To this end, we first define an auxiliary predicate $g, c \vdash t_1 \rightsquigarrow_n t_2$, which asserts that there exists a path from t_1 to t_2 in g whose total tick count (summing $c(t)$ for each vertex t on the path) is n. PathNil handles the base case where the path consists of a single vertex ($t_1 = t_2$), and PathCons extends a path by prepending an edge. The property SpanBound $g\ c\ n$ holds when every path in the graph has a total tick count of at most n (SpanBound).

Safety of an expression. Figure 11 first presents the notion of reducibility for ParLang. Intuitively, a configuration satisfies all-task reducibility, written AllRed $e\ T\ S$ if *every* task of expression e described by the task tree T can take a step with state S. RedSched asserts that any expression that can take a scheduling step is all-task reducible, since there is a single task and it can take a step. RedCtx accounts for evaluation contexts. RedPar asserts that, when facing a parallel primitive $e_1 \,||\, e_2$ with two subtrees T_1 and T_2, then if e_1 is not a value, then it should be all-task reducible with T_1, if e_2 is not a value, then it should be all-task reducible with T_2, and at least one of e_1 and e_2 has to not be a value (otherwise, a join is possible). Figure 11 then presents the notion of safety. Safe asserts that an expression e is safe with a task tree T and a state S, written Safe $e\ T\ S$, if either the expression is a value and the task tree a single leaf, or the configuration is all-task reducible.

Soundness statement. We can then state the soundness of Parcas, which asserts that, if the user verified the program e with w initial work credits and s initial span credits, then (1) the program is always safe, (2) the work is bounded by w, and (3) the span is bounded by s.

Theorem 4.1 (Soundness of Parcas). *If* $\mathcal{W}(w) * \mathcal{S}(t_0, s) \vdash wp\ t_0\ e\ \{\lambda \_\_.\ \top\}$ *holds, and if* $e \,/\, t_0 \,/\, (\emptyset, \{[t_0 := 0]\}, \emptyset) \longrightarrow^* e' \,/\, T \,/\, (\sigma, c, g)$ *then:*

(1) *Safe* $e'\ T\ (\sigma, c, g)$ *holds,*
(2) *WorkBound* $c\ w$ *holds, and*
(3) *SpanBound* $g\ c\ s$ *holds too.*

As mentioned earlier, the relation $\longrightarrow^*$ is non-deterministic, and so there are multiple possible tick maps c and computation graphs g that can arise from execution. However, the bounds in the

theorem above apply to *any* such c and g obtained from executing e, and so they upper bound the work and span across all possible interleavings of steps.

The proof of this theorem follows the standard Iris recipe, as we explain in the next section.

5 Proof of the Soundness Theorem

This section explains the soundness theorem. We start with a high-level overview of the structure (§5.1). Then we define the weakest precondition (§5.2) and present the properties of the key *state interpretation predicate* (§5.3), which gives meaning to work and span credits and relates them to the concrete work and span values (§5.4).

5.1 Proof Overview

The proof of Theorem 4.1 follows the standard Iris progress-and-preservation recipe [40, §6.4], which is made possible through the definition of the weakest precondition (§5.2). Indeed, this definition ensures that the state interpretation predicate (§5.3) is maintained as an invariant between steps of the execution. In particular, this predicate relates the physical state (the store, tick map, and computation graph) to ghost resources (points-to assertions, work and span credits, ...). The WP guarantees that every configuration reachable by execution will satisfy the state interpretation. Using the properties of the state interpretation, we can then prove the three properties of Theorem 4.1: (1) safety, (2) the work bound, and (3) the span bound, presented in increasing order of proof difficulty.

Safety. To verify safety, we extend the Iris WP to account for the task tree (§5.2).

Work bound. We bound the work following the same technique as for standard *time credits* [3, 50]. Each application of Tick consumes one work credit supplied by the user, which the state interpretation stores away as a witness that the user indeed paid for that unit of work. Since the user starts with w credits and credits are never created out of thin air, the state interpretation may store at most w credits, and thus the work is bounded by w.

Span bound. Span credits are annotated with a task identifier, and are tracked using one counter per task, representing the amount of span credits in circulation for this task. In order to bound the total span, the state interpretation stores span credits in two places. First, similarly to work credits, the state interpretation stores one span credit for the task t upon each application of Tick for t. This credit forms a witness that the user paid. Second, the state interpretation stores span credits expressing the relationship between the span of a task and the span of its children. More precisely, upon a Par that forks tasks t_1 and t_2 from t, the user gives up n span credits of t, and obtains n span credits of t_1 and n span credits of t_2. The initial n credits of t are stored inside the state interpretation, as witness of the number of span credits available to t_1 and t_2. Using these two invariants over credits, we can prove that the number of span credits available to each task bounds the heaviest path starting from this task (§5.4).

The delicate point is how span credits flow when a task t transfers them to another task t', via PrimitiveTransfer. Indeed, the second invariant described above seems to fix forever the amount of span credits available for each task. Our idea is to generate fresh credits in a *cascade*: starting from t, we walk along every task on the paths leading to t', and at each intermediate task we generate and deposit into the state interpretation enough credits to justify the increase of its successors' credits, thereby preserving the invariant. This argument is sound in particular because the destination t' is the "continuation" of t: every path reaching t' must go through t. This property is captured by the assertion transferable $t\ t'$ in the precondition of PrimitiveTransfer.

$$
\begin{aligned}
&\mathsf{wp}\ t\ e\ \{\psi\} \triangleq \mathsf{wpg}\ t\ e\ \{\psi\} \\
&\mathsf{wpg}\ T\ e\ \{\psi\} \triangleq \forall S.\ \mathsf{interp}\ S\ T\ e \Rrightarrow \\
&\quad \big(\ulcorner e \in \mathcal{V} \urcorner * \Rrightarrow \exists t.\ \ulcorner T = t \urcorner * \mathsf{interp}\ S\ T\ e * \psi\ t\ e\big) \\
&\quad \vee \big(\ulcorner e \notin \mathcal{V} \urcorner * \ulcorner \mathsf{AllRed}\ e\ T\ S \urcorner * \forall S'\ T'\ e'.\ \ulcorner e \,/\, T \,/\, S \longrightarrow e' \,/\, T' \,/\, S' \urcorner \\
&\qquad \Rrightarrow \triangleright \Rrightarrow \mathsf{interp}\ S'\ T'\ e' * \mathsf{wpg}\ T'\ e'\ \{\psi\}\big)
\end{aligned}
$$

Fig. 12. Definition of the weakest precondition modalities

5.2 Definition of the Weakest Precondition

Our definition of the WP and of assertions makes use of ghost state, relating the physical state of an execution to the logical state of the proof. The ghost state is updated with a *ghost update*, written $\Rrightarrow$. The assertion $\Rrightarrow \varphi$ means that φ holds after updating the ghost state. As usual, we write $\varphi \Rrightarrow \varphi'$ to mean $\square(\varphi \mathrel{-\!\!*} \Rrightarrow \varphi')$, where $\square$ is the persistence modality. Formally, ghost updates and our WP are parameterized by *masks*, a standard syntactical device that controls which invariants the user is allowed to open [40]. For brevity, we omit masks and refer the reader to our formalization [48].

The structure of the definition of our WP roughly follows the one by Moine et al. [47]. The first line of Figure 12 presents the definition of our WP. First, we define $\mathsf{wp}\ t\ e\ \{\psi\}$ in terms of a more general assertion, written $\mathsf{wpg}\ T\ e\ \{\psi\}$, parameterized not by a single task identifier t but rather a whole task tree T. The remainder of Figure 12 defines this general WP. This definition makes use of a state interpretation predicate $\mathsf{interp}\ S\ T\ e$, which relates the physical state S, the task tree T and the expression e to the logical state; we define it formally in Section 5.3. The definition quantifies over the state S—recall that the state encompasses the store, the tick map, and the computation graph (§3.2)—and assumes ownership of the state interpretation $\mathsf{interp}\ S\ T\ e$. Then, after a ghost update, two cases arise. If e is a value, the task tree T must be a single task t, the state interpretation must hold unchanged, and the postcondition $\psi\ t\ e$ must hold too. If e is not a value, it must be all-task reducible with T and S, and for every possible step to expression e', task tree T', and state S', then after several ghost updates, both the state interpretation $\mathsf{interp}\ S'\ T'\ e'$ and the recursive call to the WP for e' with T' must hold. The recursive call is guarded by a *later* modality to ensure this recursive definition has a fixed point.

Because the general WP is indexed by an arbitrary task tree T, it can represent computations happening in parallel. For example, the assertion $\mathsf{wpg}\ (T_1 \otimes T_2)\ (e_1 \,||\, e_2)\ \{\psi\}$ verifies that both sub-expressions e_1 and e_2 with their respective subtrees T_1 and T_2 can make progress in parallel. This generality is used when deriving versions of Par or Bind.

In order to prove Theorem 4.1, we follow a *progress and preservation* approach. We prove three lemmas. First, that the state interpretation holds for the initial state, and the initial amount of work and span credits can be given to the user. Second, if the WP holds for some expression and the state interpretation holds for some state, then one step of the expression with this state preserves the state interpretation and the WP. Third, if the WP holds for some expression and the state interpretation holds for some state, then the work and span bounds hold, and the expression is safe.

5.3 State Interpretation

The state interpretation predicate $\mathsf{interp}\ S\ T\ e$ relates the physical state $S = (\sigma, c, g)$ of an execution of the program e with task tree T to the logical state of the proof. This predicate holds at every execution step, and its definition appears in Figure 13. The state interpretation is composed of two parts: one "resourceful" assertion interp_res, defining the ghost state, and one "pure" part pureinv, asserting well-formedness properties of the different components.

$$\text{interp}\,(\sigma, c, g)\,T\,e \triangleq \text{interp\_res}\,\sigma\,c\,g * \ulcorner\text{pureinv}\,g\,(\text{dom}(c))\,T\,e\urcorner$$
$$\text{interp\_res}\,\sigma\,c\,g \triangleq \text{heap}\,\sigma * \text{interp\_work}\,(\text{sumall}(c)) * \text{interp\_span}\,g\,c * \text{forks\_and\_joins}\,g$$

Fig. 13. Definition of the state interpretation predicate

$$\text{pureinv}\,g\,d\,T\,e \triangleq \text{has\_no\_loop}\,g \wedge \text{vertices}(g) \subseteq d \wedge \text{leaves}(T) \subseteq d \wedge \text{leaves}(T) \cap \text{sources}(g) = \emptyset \wedge \text{CompTree}\,T\,e$$

CT-Leaf
$$\text{CompTree}\,t\,e$$

CT-Bind
$$\frac{\text{CompTree}\,T\,e}{\text{CompTree}\,T\,(K\langle\!\langle e\rangle\!\rangle)}$$

CT-Par
$$\frac{\text{CompTree}\,T_1\,e_1 \qquad \text{CompTree}\,T_2\,e_2 \qquad \text{leaves}(T_1) \cap \text{leaves}(T_2) = \emptyset}{\text{CompTree}\,(T_1 \otimes T_2)\,(e_1 \,||\, e_2)}$$

Fig. 14. Definition of pure invariants

The assertion $\text{interp\_res}\,\sigma\,c\,g$ asserts ownership of heap σ, which gives meaning to points-to assertions; this assertion is standard [37]. The state interpretation next asserts ownership of $\text{interp\_work}\,(\text{sumall}(c))$, which gives meaning to work credits—recall that $\text{sumall}(c)$ sums all ticks recorded in c—and of $\text{interp\_span}\,g\,c$, which gives meaning to span credits. Both assertions are described in Section 5.4. The state interpretation then asserts ownership of $\text{forks\_and\_joins}\,g$. This technical assertion gives meaning to the assertions $\text{is\_fork}\,t\,t_1\,t_2$ and $\text{is\_join}\,t_1\,t_2\,t$ that we saw in Section 4.2.

Figure 14 presents the pure invariant $\text{pureinv}\,g\,d\,T\,e$, with the argument d naming the domain of the tick map c. This invariant requires that the computation graph g has no loop, that both its vertices and the leaves of the task tree T are in d, and that these leaves are not *sources* in g—that is, they have no successors. Moreover, the invariant requires that T is compatible with the expression e, a fact denoted by $\text{CompTree}\,T\,e$. This property is inductively defined in the lower part of Figure 14, and intuitively asserts that there are at least as many parallel expressions as nodes in the task tree, and that the leaves of the tree are disjoint. Indeed, CT-Leaf asserts that a leaf is compatible with any expression. CT-Bind asserts that if a task tree is compatible with an expression, then it is also compatible with this expression within any evaluation context. CT-Par asserts that a node with two subtrees T_1 and T_2 is compatible with the parallel composition of two expressions e_1 and e_2 if T_1 is compatible with e_1, T_2 is compatible with e_2, and the leaves of T_1 and T_2 are disjoint.

5.4 Interpretation of Credits

In this section, we present the interpretation of work and span credits. While in this description we axiomatize the ghost state for simplicity, the seasoned Iris reader will recognize standard ghost state: work credits are implemented with the camera $\mathit{Auth}(\mathbb{N})$ and span credits with the camera $\mathit{Auth}(\mathit{Map}(\text{TaskIds}, \mathbb{N}))$.

Work credits. The upper part of Figure 15 presents the internal reasoning rules for work credits. WorkAlloc allocates the initial amount of work credits w_0, producing both $\text{auth\_work}\,w_0$ and $\mathcal{W}(w_0)$ after a ghost update. The assertion $\text{auth\_work}\,w_0$ keeps track of the total amount of work credits in circulation: WorkValid asserts that if $\text{auth\_work}\,w_0$ and $\mathcal{W}(w)$ hold, then w is at most w_0.

The lower part of Figure 15 defines $\text{interp\_work}\,w$, where w represents the work already done. It asserts ownership of $\mathcal{W}(w)$ and of $\text{auth\_work}\,w_0$, for some existentially quantified initial amount of work credits w_0. The additional assertion $\text{init\_work}\,w_0$ is a proof artifact, a persistent assertion that is used internally to remember that the initial number of credits never changes.

$$\Rrightarrow \mathsf{auth\_work}\, w_0 * \mathcal{W}(w_0) \qquad (\textsc{WorkAlloc})$$
$$\mathsf{auth\_work}\, w_0 * \mathcal{W}(w) \twoheadrightarrow \ulcorner w \le w_0 \urcorner \qquad (\textsc{WorkValid})$$

$$\mathsf{interp\_work}\, w \triangleq \mathcal{W}(w) * \exists w_0.\, \mathsf{auth\_work}\, w_0 * \mathsf{init\_work}\, w_0$$

Fig. 15. Ghost state for work credits

$$\Rrightarrow \mathsf{auth\_span}\, \{[t_0 := s_0]\} * \mathcal{S}(t_0, s_0) \qquad (\textsc{SpanAlloc})$$
$$\mathsf{auth\_span}\, c_0 * \mathcal{S}(t, s) \twoheadrightarrow \ulcorner s \le c_0(t) \urcorner \qquad (\textsc{SpanValid})$$
$$\mathsf{auth\_span}\, c_0 \Rrightarrow \mathsf{auth\_span}\, (\mathsf{incr}\, c_0\, t\, n) * \mathcal{S}(t, n) \qquad (\textsc{SpanUpdate})$$

$$\mathsf{interp\_span}\, g\, c \triangleq \left(\textstyle\ast_{(t,n)\in c} \mathcal{S}(t, n)\right) * \exists c_0.\, \ulcorner \mathsf{dom}(c) = \mathsf{dom}(c_0) \urcorner * \mathsf{auth\_span}\, c_0 * \mathsf{init\_span}\, g\, c_0 * \mathsf{cascade}\, g\, c_0$$
$$\mathsf{cascade}\, g\, c \triangleq \textstyle\ast_{t\in \mathsf{sources}(g)} \exists n.\, \mathcal{S}(t, n) * \ulcorner \forall t'.\, (t, t') \in g \implies c(t') \le n \urcorner$$

Fig. 16. Ghost state for span credits

Intuitively, each time Tick is applied in the proof, the one work credit given by the user of the proof rule is stored in the assertion $\mathsf{interp\_work}\, w$, as a witness that the user indeed paid. Hence, at any point during the execution, one can use WorkValid to conclude that the work done so far is at most the initial amount of work credits w_0.

Span credits. The upper part of Figure 16 presents the reasoning rules for span credits. SpanAlloc allocates the initial amount of span credits s_0 for the initial task t_0, producing $\mathsf{auth\_span}\, \{[t_0 := s_0]\}$ and $\mathcal{S}(t_0, s_0)$ after a ghost update. The assertion $\mathsf{auth\_span}\, c_0$, where c_0 is a map from task identifiers to natural numbers, keeps track of the total amount of span credits in circulation for each task. SpanValid asserts that if $\mathsf{auth\_span}\, c_0$ and $\mathcal{S}(t, s)$ hold, then s is at most $c_0(t)$. As we explain in the next paragraph, the total amount of span credits available for a task may vary during the proof because of PrimitiveTransfer, except for the initial task. To allow for that pattern, SpanUpdate allows for updating the total amount of span credits for a task t and generating said credits. The function $\mathsf{incr}\, c\, t\, n$ updates the map c by setting the value of t to n if t was not in c or to $c(t) + n$ otherwise.

The lower part of Figure 16 defines $\mathsf{interp\_span}\, g\, c$, where g is the computation graph and c the tick map. It asserts ownership of $\mathcal{S}(t, n)$ for each identifier t in the tick map, where n is the number of ticks recorded for t in c. Second, it asserts the existence of a map c_0 from task identifiers to natural numbers, with the same domain as c, representing the amount of span credits in circulation for each task, as witnessed by $\mathsf{auth\_span}\, c_0$. The assertion $\mathsf{init\_span}\, g\, c_0$ is a proof artifact used internally to remember that the amount of span credits for the initial task (the only task without incoming edges in the computation graph) does not change during the proof. The assertion $\mathsf{cascade}\, g\, c_0$ captures the core of the interpretation of span credits: for every source t of g (that is, a vertex with at least one outgoing edge), there exists some amount of span credit n of t such that, for any successor t' of t in g, n bounds the amount of span credits for t'.

Intuitively, each time Tick is applied in the proof, the one span credit given by the user is stored in the assertion $\mathsf{interp\_span}\, g\, c$, as a witness that the user indeed paid. Moreover, each time Par or Transfer is applied, the span credits of the parent task given by the user are stored in $\mathsf{cascade}\, g\, c_0$, as a witness that the span of the parent bounds the span of all of its children. Hence, at any point during the execution, one can use SpanValid to conclude that the initial amount of span credits of the initial task bounds the weight of the heaviest path starting from it.

The reader might wonder what happens during the join of two tasks t_1 and t_2 into a continuation t'. Indeed, one has to update the assertion $\mathsf{cascade}\, g\, c_0$ into $\mathsf{cascade}\, (\{(t_1, t'), (t_2, t')\} \cup g)\, ([t' := c']c_0)$

```
parfor ≜ μ̂f. λa b h.
  if (b − a) ≤ 0 then ()
  else if (b − a) == 1 then h a
  else let mid = a + (b − a)/2 in
    tick; (f a mid h) || (f mid b h)
```

$$\textsc{Parfor}\quad \frac{\mathcal{W}(b-a-1) \qquad \mathcal{S}(t, \lceil \log_2(b-a) \rceil + C) \qquad \ast_{i\in[a;b)}\, \forall t'.\ \mathcal{S}(t', C) \mathrel{-\!\!*} \mathsf{wp}\ t'\ (h\ i)\ \{\lambda \_\_.\ Q\, i\}}{\mathsf{wp}\ t\ (\mathsf{parfor}\, a\, b\, h)\ \{\lambda \_\_.\ \ast_{i\in[a;b)}\, Q\, i\}}$$

Fig. 17. The parallel for loop and its specification

for some initial amount of span credits c' for t'. Because of the cascade assertion, this update requires providing at least c' span credits of t_1 and t_2, since the new computation graph is updated with two new edges from t_1 and t_2 to t' (see SchedJoin in Figure 7). Yet, the user is not required to give any span credit at the completion of a parallel composition, and thus no credits are available. Hence, we choose $c' = 0$, that is, to initially give 0 span credit to t'. How are the span credits of t' created then? The answer lies in the proof of the PrimitiveTransfer rule, which we cover next.

Validity of the transfer of span credits. Recall that PrimitiveTransfer allows for transferring span credits from a task t to another task t', provided that the current goal is a WP with current task t', and that the user has a witness transferable $t\, t'$. From this witness, we deduce that two facts hold: (1) there is a path from t to t' in the underlying computation graph g, and (2) every predecessor of t' in g can be reached by t. Moreover, since PrimitiveTransfer must be applied facing a WP with current task t', we know that t' has no successor in g. From these three facts, we use n span credits of t to generate n more span credits for all the tasks t can reach (including t'), in cascade. For every task that has successors, we store in the cascade assertion additional span credits, in order to justify the validity of the increase of the span credits of their successors. Since t' has no successor, we don't need to store away the generated n credits in cascade, and we can give them back to the user.

6 Case Studies

In this section, we illustrate the expressiveness of Parcas by verifying multiple case studies. We first present our implementation and specification of two fundamental building blocks of parallel computations: the parallel for loop (§6.1) and the scan operation (§6.2). We then devote our attention to parallel sorting, and verify a parallel merge function (§6.3), on which we build a parallel merge sort (§6.4). We finally verify Treiber's stack, a linearizable lock-free stack, for which we give work and span bounds (§6.5). While we do not prove any tightness results for these bounds, the costs of all examples are asymptotically as tight as one can expect from the literature.

Since tick operations model costs, their placement is critical. In our case studies, we place a tick operation in each branch that performs a recursive call.

6.1 The Parallel For Loop

The parallel for loop parfor is a fundamental building block of parallel computations. It allows for executing a given function in parallel on every index of a given range.

Code. The upper part of Figure 17 presents the code of parfor. The parfor function takes three arguments: a lower bound a, an upper bound b, and a function h to execute in parallel at every index in $[a, b)$. The function is defined recursively, with f as the recursive name. It first checks whether $(b - a)$ is negative or zero, in which case there is nothing to do, and then whether $(b - a)$ is equal to 1, in which case the function h is called on a. If $(b - a) \geq 2$, then the function computes the midpoint *mid*, performs a tick operation, and recursively calls itself on the ranges $[a, mid)$

$$\begin{array}{l}\mathsf{tabulate} \triangleq \lambda h\, n.\\ \quad \mathsf{let}\, x = \mathsf{alloc}\, n\, () \,\mathsf{in}\\ \quad \mathsf{let}\, k = \lambda i.\, x[i] \leftarrow (h\, i) \,\mathsf{in}\\ \quad \mathsf{parfor}\, 0\, n\, k; x\end{array} \qquad \begin{array}{l}\mathsf{scan} \triangleq \hat{\mu} f.\, \lambda s.\\ \quad \mathsf{let}\, n = \mathsf{length}\, s \,\mathsf{in}\\ \quad \mathsf{if}\, n == 1 \,\mathsf{then}\, \mathsf{let}\, r = \mathsf{alloc}\, 2\, 0 \,\mathsf{in}\, r[1] \leftarrow s[0]; r \,\mathsf{else}\\ \quad \mathsf{let}\, c = \mathsf{tabulate}\, (\lambda i.\, s[2\times i] + s[2\times i+1])\, (n/2) \,\mathsf{in}\\ \quad \mathsf{let}\, p = \mathsf{tick}; f\, c \,\mathsf{in}\\ \quad \mathsf{let}\, g = \lambda i.\, \mathsf{let}\, j = i/2 \,\mathsf{in}\\ \quad\quad \mathsf{if}\, (i \bmod 2) == 0 \,\mathsf{then}\, p[j] \,\mathsf{else}\, p[j] + s[i-1] \,\mathsf{in}\\ \quad \mathsf{tabulate}\, g\, (n+1)\end{array}$$

$$\textsc{Tabulate}\quad \frac{n \neq 0 \qquad \mathcal{W}(n-1) \qquad \mathcal{S}(t, \lceil \log_2 n \rceil + C) \qquad \mathbin{*}_{i\in[0;n)} \forall t'.\, \mathcal{S}(t', C) \mathrel{-\!\!*} \mathsf{wp}\; t'\; (h\; i)\; \{\lambda \_ v.\, Q\, i\, v\}}{\mathsf{wp}\; t\; (\mathsf{tabulate}\, h\, n)\; \{\lambda \_ v.\, \exists \ell\; \vec{w}.\, \ulcorner v = \ell \;\wedge\; |\vec{w}| = n \urcorner * \ell \mapsto \vec{w} * \mathbin{*}_{i\in[0;n)} Q\, i\, \vec{w}(i)\}}$$

$$\textsc{Scan}\quad \frac{\ulcorner |\vec{v}| = 2^k \urcorner \qquad \mathcal{W}(3\times 2^k) \qquad \mathcal{S}(t, (k+1)^2) \qquad s \mapsto \vec{v}}{\mathsf{wp}\; t\; (\mathsf{scan}\; s)\; \{\lambda \_ v.\, \exists \ell\; \vec{w}.\, \ulcorner v = \ell \;\wedge\; \mathsf{scanned}\, \vec{v}\, \vec{w} \urcorner * s \mapsto \vec{v} * \ell \mapsto \vec{w}\}}$$

$$\mathsf{scanned}\, \vec{v}\, \vec{w} \triangleq |\vec{w}| = |\vec{v}| + 1 \;\wedge\; \forall i.\, 0 \le i \le |\vec{v}| \implies \vec{w}(i) = \sum_{j<i} \vec{v}(j)$$

Fig. 18. The tabulate primitive, the scan function, and their specifications

and $[mid, b)$. We place a tick operation before the parallel primitive: the cost incurred by this tick operation accounts for the generation of a binary tree of tasks.

Specification. The lower part of Figure 17 presents the specification of a call to parfor $a\, b\, h$ on task t. The work and span costs of parfor reflect the fact that the execution graph of this function contains a parallel binary tree with $(b-a)$ leaves calling h, with a tick operation at every internal node. As a result, the first part of the precondition consumes $(b-a-1)$ work credits, which upper bounds the number of internal nodes in this tree. Next, the second premise of the precondition consumes $\lceil \log_2(b-a) \rceil + C$ span credits of t, where C is a constant selected when applying the rule. The logarithmic factor accounts for the height of the binary execution tree generated by parfor. Finally, the third premise of the precondition requires the user to prove a weakest precondition about every call to h for an argument i in $[a; b)$, at a universally quantified identifier t', assuming C span credits are available. These C credits come from what is put in as part of the second premise, and represent the remaining span credits that reach each leaf before its call to h. The postcondition returns the iterated conjunction of the postconditions of the calls to h.

6.2 Scan (Parallel Prefix Sums)

The scan operation, or parallel prefix sum, is another standard building block of parallel algorithms [6]. Given an array $\vec{v}$ and an associative sum operation, the scan operation returns another array $\vec{w}$ such that $\vec{w}[0] = 0$ and $\vec{w}[i]$ stores the sum of $\vec{v}[0], \ldots, \vec{v}[i-1]$. Formally, the last line of Figure 18 defines an assertion $\mathsf{scanned}\, \vec{v}\, \vec{w}$ stating that $\vec{w}$ stores the result of the scan of array $\vec{v}$.

While the scan operation might at first seem to be inherently sequential, it can be efficiently implemented in parallel. In this section, for simplicity, we focus on arrays of integers that have a size that is a power of 2.

Code. The upper part of Figure 18 first presents the code of the tabulate function, a utility function that takes as argument a function f and an integer n, and which constructs a fresh array of size n whose i-th entry is the result of $f\; i$, computed in parallel using parfor. The upper part

of Figure 18 then presents the scan function itself, which implements a contraction algorithm, similar to the standard three-phase algorithm [19]. This function takes as argument an array s and is defined recursively, with recursive name f. The function first tests if the argument contains a single element, in which case it returns a new array of size 2, where the first index stores 0 and the second index stores the single element of s. Otherwise, the function has three phases. First, it allocates the contraction c, which is an array of half the size of the argument, which stores the sum of each pair of consecutive elements. Second, it performs a tick operation and calls itself recursively to compute the prefix sum of this intermediate array c and names the result p. Third, it expands the result back to full size using a second tabulate: even-indexed entries are taken directly from p, while odd-indexed entries are each obtained by adding an input element to the preceding even-index prefix sum. As we will see, our implementation of scan has $O(n)$ work and $O(\log^2(n))$ span, where n is the length of the input array. Our implementation of scan has a tick operation before the recursive call, accounting for the (logarithmic) depth of the recursion. Note that other costs will be incurred by calls to tabulate.

Specification. The lower part of Figure 18 first presents the specification of a call to tabulate $h\ n$ on task t. Its specification is similar to the one of parfor. It consumes $n-1$ work credits and $\lceil \log_2 n \rceil + C$ span credits of t, and requires the user to verify every call to h for an argument i in $[0; n)$, at a universally quantified identifier t', assuming C span credits are available. The postcondition asserts that the function returns an array $\vec{w}$ of length n, such that $Q\ i\ \vec{w}(i)$ holds for every $i \in [0; n)$.

The lower part of Figure 18 then presents the specification of a call to scan s on task t. The precondition requires that s points to an array $\vec{v}$ of size 2^k. It also consumes 3×2^k work credits and $(k+1)^2$ span credits of t, yielding linear work and polylog span. The postcondition asserts that the function returns a fresh array ℓ with the prefix sums of the argument s, which is returned untouched. Instead of working directly with closed-form expressions for the work and span, we establish the proof using open forms in terms of recurrence relations. More precisely, we first define (at the logical level) recursive functions $\mathrm{scan}_W^{eq}(n)$ and $\mathrm{scan}_S^{eq}(n)$, which will represent the total work and span credits, respectively, that will be needed for an input of length n (since n is always a power of two, we ignore rounding in division):

$$\mathrm{scan}_W^{eq}(n) \triangleq \text{if } n \le 1 \text{ then } 0 \text{ else } (n/2 - 1) + 1 + \mathrm{scan}_W^{eq}(n/2) + n$$
$$\mathrm{scan}_S^{eq}(n) \triangleq \text{if } n \le 1 \text{ then } 0 \text{ else } \lceil \log_2(n/2) \rceil + 1 + \mathrm{scan}_S^{eq}(n/2) + \lceil \log_2(n+1) \rceil$$

For both work and span, the non-base case is a sum of four terms. The first term accounts for the cost of the first tabulate, which allocates an array of size $n/2$ and for which the executed closure has no cost (the constant C in Tabulate is instantiated to 0). The second term accounts for the tick operation before the recursive call. The third term accounts for the recursive call. The fourth term accounts for the cost of the second tabulate, which allocates an array of size $n+1$ with again a free closure. Working in terms of these recurrence relations, we split up the parts of the non-base case to pay for each of these respective components. Outside of the logic, we then separately establish closed upper bounds on the recurrence relation showing that $\mathrm{scan}_W^{eq}(n) \le 3 \times n$ and $\mathrm{scan}_S^{eq}(n) \le (\lceil \log_2(n) \rceil + 1)^2$. Since Parcas is affine, we can use WorkSplit and SpanSplit to weaken our specification to the form shown in Scan with these upper bounds.

The proof makes crucial use of Transfer. Indeed, before executing the first tabulate operation, we split span credits in two, one assertion with $\log_2(n/2)$ for the upcoming tabulate, and $(\mathrm{scan}_S^{eq}(n/2) + \log_2(n+1))$ for the remaining computation. However, because tabulate will change the task identifier, we have to use Transfer in order to transfer the second assertion to the continuation.

$$
\begin{array}{l}
\mathsf{merge} \triangleq \hat{\mu} f.\, \lambda g\ s_1\ s_2\ d. \\
\quad \mathsf{if}\ \mathsf{s\_length}\ d \le g\ \mathsf{then}\ \mathsf{merge\_seq}\ s_1\ s_2\ d \\
\quad \mathsf{else\ let}\ n_1 = \mathsf{s\_length}\ s_1\ \mathsf{in\ let}\ n_2 = \mathsf{s\_length}\ s_2\ \mathsf{in} \\
\qquad \mathsf{if}\ n_1 == 0\ \mathsf{then}\ \mathsf{copy\_seq}\ s_2\ d \\
\qquad \mathsf{else\ let}\ mid_1 = n_1/2\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ p = \mathsf{s\_load}\ s_1\ mid_1\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ mid_2 = \mathsf{binsearch}\ s_2\ p\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ l_1 = \mathsf{s\_split}\ s_1\ 0\ mid_1\ \mathsf{in\ let}\ r_1 = \mathsf{s\_split}\ s_1\ (mid_1+1)\ n_1\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ l_2 = \mathsf{s\_split}\ s_2\ 0\ mid_2\ \mathsf{in\ let}\ r_2 = \mathsf{s\_split}\ s_2\ mid_2\ n_2\ \mathsf{in} \\
\qquad\quad \mathsf{s\_store}\ d\ (mid_1 + mid_2)\ p; \\
\qquad\quad \mathsf{let}\ d_l = \mathsf{s\_split}\ d\ 0\ (mid_1 + mid_2)\ \mathsf{in\ let}\ d_r = \mathsf{s\_split}\ d\ (mid_1 + mid_2 + 1)\ (\mathsf{s\_length}\ d)\ \mathsf{in} \\
\qquad\quad \mathsf{tick};\ (f\ g\ l_1\ l_2\ d_l)\ ||\ (f\ g\ r_1\ r_2\ d_r)
\end{array}
$$

$$
\mathsf{merge}_W(n,m) \triangleq 11 \times (n+m) \qquad \mathsf{merge}_S(g,n,m) \triangleq g + 4 \times (\lceil \log_2 n \rceil + 1) \times (\lceil \log_2 m \rceil + 1)
$$

MERGE

$$
\frac{\mathcal{W}(\mathsf{merge}_W(|\vec{v}_1|,|\vec{v}_2|)) \qquad \mathcal{S}(t, \mathsf{merge}_S(g,|\vec{v}_1|,|\vec{v}_2|)) \qquad \ulcorner \mathsf{sorted}\ \vec{v}_1 \ \wedge\ \mathsf{sorted}\ \vec{v}_2 \ \wedge\ |\vec{w}| = |\vec{v}_1| + |\vec{v}_2| \urcorner \qquad \mathsf{slice}\ s_1\ \vec{v}_1 \qquad \mathsf{slice}\ s_2\ \vec{v}_2 \qquad \mathsf{slice}\ d\ \vec{w}}{\mathsf{wp}\ t\ (\mathsf{merge}\ g\ s_1\ s_2\ d)\ \{\lambda\_ v.\, \mathsf{slice}\ s_1\ \vec{v}_1 * \mathsf{slice}\ s_2\ \vec{v}_2 * \mathsf{slice}\ d\ (\mathsf{puremerge}(\vec{v}_1,\vec{v}_2))\}}
$$

Fig. 19. The parallel merge operation and its specification

6.3 Parallel Merge

The parallel merge sort that we will present in Section 6.4 relies on a fundamental component: the ability to merge two sorted arrays in parallel. In this section, we first show how to verify such a parallel merge. The code we verify is a direct translation into ParLang of the implementation of merge from the MaPLe standard library [49].

Slices. In order to preserve a relatively concise definition for merge, the function doesn't work directly on arrays but on *slices*. A slice is a tuple of an array location, a lower index and an upper index, describing a portion of the underlying array. Above this wrapper, we define functions s_length, s_load, s_store as expected. We also define the function s_split, which extracts a sub-slice between two indices. Note that s_split does not allocate a new array—the returned slice shares the same underlying array as its argument. All of these operations are constant time and have no tick operations. At the specification level, slices are represented with the slice $\ell\ \vec{v}$ predicate, which behaves close to the standard points-to predicate. This predicate is implemented with standard ghost state, which we omit for brevity.

Code. The upper part of Figure 19 presents the code of merge. The function takes four arguments: a granularity threshold g, two sorted source slices s_1 and s_2, and a target slice d. The function is defined recursively, with f as the recursive name, and merges s_1 and s_2 into d, overwriting its content. If $\mathsf{s\_length}\ d \le g$, the function falls back to a sequential merge (code omitted). If s_1 is empty, the function copies s_2 into d. Otherwise, it selects the median element of s_1 as a pivot p, and uses binary search to find the index mid_2 in s_2 such that all elements before mid_2 are smaller than p and all elements after are greater. It places p at its final position $mid_1 + mid_2$ in d, and splits all three slices around this pivot position. After a tick operation, the function recursively merges the left and right halves in parallel. By placing a tick operation before the parallel primitive, we account for the creation of a binary tree of tasks.

Specification. Before diving into the work and span bounds of merge, we first present bounds for intermediate sequential functions—for which work and span are equal—assuming n is the length of the first array s_1 and m is the length of the second array s_2.

Function	merge_seq s_1 s_2 d	copy_seq s_2 d	binsearch s_2 p
Work and Span	$n + m$	m	$\lceil \log_2(m) \rceil + 1$

The center part of Figure 19 presents the work and span bounds we derive for merge. The function $\text{merge}_W(n, m)$ gives the work of merge when the first argument has length n and second argument has length m, and states that the work is in $O(n + m)$. The function $\text{merge}_S(g, n, m)$ presents the span, which is in $O(g + \log_2(n) \times \log_2(m))$. Indeed, merge calls a binary search on the second array, which has logarithmic span, at every recursive call, and the depth of the recursion is logarithmic. As for scan (§6.2), we obtained these bounds by first conducting the proof using open recurrence equations

$$\begin{aligned}
\text{merge}^{eq}_W(g, n, m) \triangleq\ & \text{if } n + m \le g \text{ then } n + m \text{ else if } n = 0 \text{ then } m - 1 \\
& \text{else } 1 + (1 + \lceil \log_2 m \rceil) + \\
& \quad \max_{0 \le p \le m} \left(\text{merge}^{eq}_W(g, \lfloor n/2 \rfloor, p) + \text{merge}^{eq}_W(g, n{-}1{-}\lfloor n/2 \rfloor, m{-}p) \right) \\
\text{merge}^{eq}_S(g, n, m) \triangleq\ & \text{if } n + m \le g \text{ then } n + m \text{ else if } n = 0 \text{ then } \lceil \log_2 m \rceil \\
& \text{else } (1 + \lceil \log_2 m \rceil) + 1 + \\
& \quad \max_{0 \le p \le m} \left(\max\ (\text{merge}^{eq}_S(g, \lfloor n/2 \rfloor, p))\ (\text{merge}^{eq}_S(g, n{-}1{-}\lfloor n/2 \rfloor, m{-}p)) \right)
\end{aligned}$$

We then show in Rocq that $\text{merge}^{eq}_W(g, n, m) \le \text{merge}_W(n, m)$ and $\text{merge}^{eq}_S(g, n, m) \le \text{merge}_S(g, n, m)$. Formally proving a closed form for the work was a difficult task; we posit that we are the first to give a machine-checked proof of this.

The lower part of Figure 19 presents the formal specification of a call to merge $g\ s_1\ s_2\ d$ on task t. The precondition requires the two source slices s_1 and s_2 to represent arrays $\vec{v}_1$ and $\vec{v}_2$, which must be sorted, and the target slice d to represent some irrelevant array $\vec{w}$ whose length equals $|\vec{v}_1| + |\vec{v}_2|$. It also consumes work credits $\text{merge}_W(|\vec{v}_1|, |\vec{v}_2|)$ and span credits $\text{merge}_S(g, |\vec{v}_1|, |\vec{v}_2|)$. The postcondition returns ownership of the source slices unchanged, and asserts that the destination slice d now contains $\text{puremerge}(\vec{v}_1, \vec{v}_2)$, the sorted merge of $\vec{v}_1$ and $\vec{v}_2$.

6.4 Parallel Merge Sort

The parallel merge sort we verify is a direct translation into ParLang of the implementation of merge sort from the MaPLe standard library [49].

Code. The code appears in the upper part of Figure 20, and uses a ping-pong/double buffering approach to remove the need for copies. The function is parameterized by a granularity threshold g, a boolean b, a source array s, and a destination array d of the same size. When $b = \text{true}$, the function writes the sorted contents of s into d, leaving s with garbage content. When $b = \text{false}$, the function sorts s in-place, using d as temporary memory. The function sort is defined recursively, with f as the recursive name. If s is empty, there is nothing to do. If s has a single element, it is already sorted; the function copies this single element to d if $b = \text{true}$, and does nothing otherwise. In the general case, the function computes the midpoint mid, splits both s and d into left and right halves around mid, and after a tick operation, recursively sorts the two halves in parallel. The boolean argument is flipped for the recursive calls, alternating the buffer. Finally, the function merges the two sorted halves: if $b = \text{true}$, the halves of s are merged into d; and otherwise, if $b = \text{false}$, the

$$
\begin{array}{l}
\text{sort } g \triangleq \hat{\mu} f.\, \lambda b\ s\ d. \\
\quad \text{let } n = \text{s\_length } s \text{ in} \\
\quad \text{if } n == 0 \text{ then } () \\
\quad \text{else if } n == 1 \text{ then if } b \text{ then } (\text{s\_store } d\ 0\ (\text{s\_load } s\ 0)) \text{ else } () \\
\quad \text{else let } mid = n/2 \text{ in} \\
\quad\quad \text{let } s_l = (\text{s\_split } s\ 0\ mid) \text{ in let } s_r = (\text{s\_split } s\ mid\ n) \text{ in} \\
\quad\quad \text{let } d_l = (\text{s\_split } d\ 0\ mid) \text{ in let } d_r = (\text{s\_split } d\ mid\ n) \text{ in} \\
\quad\quad \text{tick}; (f\ \neg b\ s_l\ d_l) \mid\mid (f\ \neg b\ s_r\ d_r); \\
\quad\quad \text{if } b \text{ then } (\text{merge } g\ s_l\ s_r\ d) \text{ else } (\text{merge } g\ d_l\ d_r\ s)
\end{array}
$$

$$
\text{sort}_W(n) \triangleq 12 \times n \times \lceil \log_2 n \rceil \qquad \text{sort}_S(g, n) \triangleq g + (g+5) \times (\lceil \log_2 n \rceil + 1)^3
$$

$$
\frac{\text{SORT} \quad \ulcorner |\vec{v}| = |\vec{w}| \urcorner \quad \mathcal{W}(\text{sort}_W(|\vec{v}|)) \quad \mathcal{S}(t, \text{sort}_S(g, |\vec{v}|)) \quad \text{slice } s\ \vec{v} \quad \text{slice } d\ \vec{w}}{\text{wp } t\ (\text{sort } g\ b\ s\ d)\ \left\{ \lambda\_ v.\ \begin{array}{l} \exists \vec{v}'\ \vec{w}'.\ \ulcorner \text{sorted } \vec{v}' \ \wedge\ \text{permutation } \vec{v}'\ \vec{v} \ \wedge\ |\vec{w}'| = |\vec{w}| \urcorner\ * \\ \text{slice } s\ (\text{if } b \text{ then } \vec{w}' \text{ else } \vec{v}')\ *\ \text{slice } d\ (\text{if } b \text{ then } \vec{v}' \text{ else } \vec{w}') \end{array} \right\}}
$$

Fig. 20. The parallel merge sort function and its specification

halves of d are merged into s. Similarly to parfor and merge, by placing a tick operation before the parallel primitive, we account for the creation of a binary tree of tasks.

Specification. The work and span bounds of sort are given in the center part of Figure 20. We show that sort has work in $O(n \log n)$ and that its span is in $O(\log^3 n)$, where n is the length of the input array. These bounds are again obtained by closing open recurrence equations—we refer the curious reader to our formalization [48].

The specification of a call to sort $g\ b\ s\ d$ on task t appears in the lower part of Figure 20. The precondition requires that s is a slice of content $\vec{v}$ and d a slice of content $\vec{w}$, of the same length. The precondition also consumes the expected work and span credits. The postcondition asserts that there exist two new arrays. First $\vec{v}'$, the sorted content of $\vec{v}$—this is witnessed by the property sorted $\vec{v}'$ that says that $\vec{v}'$ itself is sorted, and permutation $\vec{v}'\ \vec{v}$, that says that $\vec{v}'$ contains a permutation of $\vec{v}$. Second $\vec{w}'$ is the content of the buffer that was used, that is, an array with unspecified content but of the same length as the arguments. The content of the two slices s and d depends on the value of the boolean b. If $b = \text{true}$, the sorted array $\vec{v}'$ is stored in d and s is left with unspecified content $\vec{w}'$; if $b = \text{false}$, the sorted array $\vec{v}'$ is stored in s and d is left with unspecified content $\vec{w}'$.

6.5 Treiber's Stack

Treiber's stack [59] is a concurrent, linearizable [31], and lock-free stack. With Parcas, we show that indeed Treiber's stack is linearizable (in a style similar to those in prior Iris-based logics), and that it satisfies a performance property: assuming a fixed number of participants, we provide work bounds for push and pop, meaning that multiple push and pop operations can happen in parallel, and none of them can be indefinitely delayed by the others. Moreover, the span bounds we provide for push and pop intuitively illustrate the worst case cost of contention.

Code. The upper part of Figure 21 presents our implementation of Treiber's stack. The stack is represented as a reference on a linked list of nodes. Each node is an array of size 2, where the first index stores the value of the node and the second index stores a pointer to the next node. The nil case is represented by the unit value (). The function create makes a new stack by allocating a reference to an empty list. The function call push $s\ x$ pushes a value x on the stack s. This function

```
create ≜ λ_. alloc 1 ()

push ≜ μ̂f. λs v.
  let h' = alloc 2 v in
  let h = s[0] in
  h'[1] ← h;
  if CAS s 0 h h' then () else (tick; f s v)

pop ≜ μ̂f. λs.
  let h = s[0] in
  if h == () then () else
    let h' = h[1] in
    if CAS s 0 h h' then h[0] else (tick; f s)
```

CREATE

$$\frac{0 \le n \qquad 0 \le m}{\mathsf{wp}\ t\ (\mathsf{create}\ ())\ \{\lambda\_ v.\ \exists s.\ \ulcorner v = s \urcorner * \mathsf{stack}\ s\ n\ m\ [] * \text{✻}_{[0;n)}\ (\forall t'.\ \mathcal{S}(t', m) \mathrel{-\!\!*} \mathsf{participant}\ s\ t')\}}$$

PARTICIPANTTRANSFER

$$\frac{\mathsf{participant}\ s\ t \qquad \mathsf{wp}\ t\ e\ \{\lambda\, t'\, v.\ \mathsf{participant}\ s\ t' \mathrel{-\!\!*} \psi\ t'\ v\}}{\mathsf{wp}\ t\ e\ \{\psi\}}$$

STACKPUSH

$$\{\mathsf{participant}\ s\ t\}\quad \langle \forall m\, xs.\, \mathcal{W}(n-1) * \mathsf{stack}\ s\ n\ (m+1)\ xs\rangle\quad t : (\mathsf{push}\ s\ x)\quad \langle \mathsf{stack}\ s\ n\ m\ (x :: xs)\rangle\quad \{\lambda t'\ \_.\ \ulcorner t' = t \urcorner * \mathsf{participant}\ s\ t\}$$

STACKPOP

$$\{\mathsf{participant}\ s\ t\}\quad \langle \forall m\, x\, xs.\, \mathcal{W}(n-1) * \mathsf{stack}\ s\ n\ (m+1)\ (x :: xs)\rangle\quad t : (\mathsf{pop}\ s)\quad \langle \mathsf{stack}\ s\ n\ m\ xs\rangle\quad \{\lambda t'\ v.\ \ulcorner t' = t \wedge v = x \urcorner * \mathsf{participant}\ s\ t\}$$

Fig. 21. Code and specifications for Treiber's stack

first allocates a new node, writes x and the current head of the stack in this node, and then performs an atomic CAS operation to try to update the reference s to this new cell. If the CAS succeeds, the function returns, and otherwise it calls itself recursively to make another attempt. The function call pop s pops the head of the stack s. The function first loads the current content of s into h. If $h = ()$, the stack is empty and the call returns. Otherwise, we know that h is a proper list node, the function then tries to update s to the next node using an atomic CAS operation. If the CAS succeeds, the function returns the value of the head of the stack stored in h, and otherwise it calls itself recursively to make another attempt. For both push and pop, the tick operation accounts for the cost of a failed CAS operation, that is, for the cost of contention.

Atomic triples. Our specifications of push and pop specify that these functions are linearizable. To express this property, we use an *atomic triple* [20, 41], which indeed guarantees linearizability [5]. In our work, an atomic triple takes the form

$$\{\varphi_1\}\langle \forall x.\, \varphi_2\rangle\ t : e\ \langle \varphi_3\rangle\{\lambda t'\ v.\ \varphi_4\}$$

Such a triple specifies the execution of e on task t. The precondition φ_1 is called the *private precondition* and must hold at the beginning of the execution of e. The precondition φ_2 is called the *public precondition* and is guaranteed to be updated atomically, at some point during the execution of e, to the public postcondition φ_3. Finally, the private postcondition φ_4 holds at the end of the execution of e. Note the universal quantifier in the public precondition, which scopes over the public precondition, and both the public and private postconditions. Our encoding of atomic triples follows the standard Iris recipe [38].

Specification. The lower part of Figure 21 presents the specification of the three functions create, push, and pop. These specifications involve two new assertions. First, the assertion stack $s\ n\ m\ xs$ states that s is a stack that may be used concurrently by at most n participants to perform at

most m operations, and that the content of the stack is the list of values xs. Second, the assertion participant $s\ t$ states that t is a participant for stack s, that is, a task that may perform operations on s. These two assertions are not duplicable. Moreover, the assertion participant $s\ t$ behaves as a storage of span credits. As such, we offer ParticipantTransfer that mimics Transfer (and is derived from it), and allows for transferring a participant assertion to a subsequent task.

The figure presents the specification of create (). The postcondition asserts that the returned value is a location s that represents a stack with at most n participants and m operations (n and m being chosen by the user), and that the content of the stack is empty. Moreover, the postcondition asserts an n-fold iterated conjunction of separation implications, each producing a participant assertion for a given task t' in exchange for m span credits of t'. Indeed, the intention is to bound the span of every task t' that performs an operation on the stack by m. Yet, span credits for t' are not yet available at the moment of the call to create, since t' likely does not even exist yet.

The figure then presents the specification of push $s\ x$ on task t. The private precondition requires t to be a declared participant of s as it consumes the assertion participant $s\ t$. The public precondition universally quantifies over the remaining number of operations m and the content of the stack xs, and consumes the assertion stack $s\ n\ (m+1)\ xs$ as well as $(n-1)$ work credits. The $m+1$ ensures that at least one operation is still allowed. The $(n-1)$ work credits are intuitively used to "compensate" for the success of the CAS operation. Indeed, there are at most $n-1$ other participants (executing either push or pop) concurrently. These participants may have already read the content of s and may themselves be planning to perform a CAS operation to update s. However, because the CAS operation of t succeeded, the other participants are bound to fail their CAS, and as such will have to pay for a tick operation. The public postcondition asserts that x has been inserted in the stack, and that one less operation is allowed. The private postcondition gives back the participant $s\ t$ assertion, allowing for a further call to push or pop by t.

The specification of pop s is similar. Indeed, the private precondition consumes a participant $s\ t$ assertion. The public precondition universally quantifies over the remaining number of operations m, the top of the stack x and the content xs; it then consumes stack $s\ n\ (m+1)\ (x :: xs)$ as well as $(n-1)$ work credits. The work credits are used in the same way as for push. The public postcondition asserts that x has been popped from the stack, and that one less operation is allowed. The private postcondition asserts that the returned value is x and gives back the participant $s\ t$ assertion.

Proof. We present the proof technique we use for the credits consumption, and refer the curious reader to our mechanization for more details [48].

Span credits are handled directly with the participant assertion. Create globally fixes m_0, the maximum number of push/pop operations, and we track the number d of operations already made (both are constrained using ghost state). The assertion participant $s\ t$ stores $(m_0 - d)$ span credits for t, which suffice to pay for the tick of each failed CAS.

Work credits are more subtle. Following the literature on lock-freedom proofs, a task performing a push/pop operation pays for the work it may cause parallel tasks to incur. The protocol is enforced by the stack invariant using two ghost tokens. A "passive" token, stored in the participant assertion, denotes the capability to perform an operation; an "active" token, tagged with a location ℓ, denotes a task currently performing an operation assuming the head of the list is ℓ. The invariant stores one work credit per active token whose assumed head is outdated. A task reading the head exchanges its passive token for an active one tagged with the location read. Upon a CAS, it accesses the public precondition to obtain $n-1$ work credits, where n is the number of participants. If the CAS succeeds, it stores one credit for each of the (at most $n-1$) other tasks whose active tokens are now outdated, exchanges its active token for a passive one, and returns. If the CAS fails, it returns the

$n - 1$ credits to the user, uses its passive token to reclaim the one credit stored by the contending task that succeeded, pays for the tick, and retries.

7 Related Work

Program logics for resource usage. Program logics for reasoning about resource usage date back to Nielson [51, 52], who proposes a logic for the time complexity of pure programs with while loops. This logic has no notion of credits. The idea of credits has its roots in type systems for time complexity [35, 55], and was introduced in separation logic by Atkey [3]. Haslbeck and Nipkow [29] compare a specialized Hoare logic and a separation logic with credits to verify time bounds.

Time credits in separation logic have been used extensively for verifying the time complexity of sequential programs. For instance, Charguéraud and Pottier [17, 18] use time credits to verify the time complexity of a union-find implementation, using the CFML framework [16]. In the same framework, Moine et al. [44] verify the time complexity of a transient data structure. Mével et al. [50] bring time credits to the Iris world. They also introduce the dual notion of *time receipts*, for proving a lower bound on the time complexity of a program. We posit that these receipts could also be adapted for work and span. Our specifications with work and span credits are extremely precise (§6), with explicated constants instead of big-O notations. Guéneau et al. [26] present a way of encoding big-O notations in separation logic with time credits, and we posit that their approach could be adapted to our setting. More recently, Pottier et al. [56] apply time credits in Iris to programs using thunks to achieve good amortized time complexity. Standard time credits are well suited for amortization: one can surcharge the actual cost of an operation, and store the difference inside another assertion, prepaying a part of the totality of a future operation. As our case study on Treiber's stack shows (§6.5), work credits are also well suited for amortization. Span credits, however, are annotated with task identifiers, and are thus less amenable to amortization. This is not a problem in practice, since span analysis is very rarely amortized.

All the tools presented so far are for reasoning about *worst case* time complexity. Haselwarter et al. [28] present Tachis, a separation logic for reasoning about the *expected* cost (and in particular expected time) of probabilistic programs. When reasoning on a sampling instruction, Tachis allows for distributing credits between the different outcomes, as long as the expected value is preserved.

Credits have also been used for reasoning about other resources. For instance, Madiot and Pottier [43] and Moine et al. [46] use space credits to verify the heap space complexity of programs under garbage collection. Subsequently, Moine et al. [45] scaled their approach to a concurrent setting. Aguirre et al. [2] and Li et al. [42] propose *error credits*, a logical tool for bounding the error probability of a probabilistic program. Credits are also useful as an internal logical tool, as in *later credits*, which simplify reasoning about the so-called later modality [58].

Type systems for work and span. Many type systems have been developed for sequential time complexity, some of which we alluded to in the introduction. We focus here on the closest related work, concerning work and span. Hoffmann and Shao [34] present a type system with an inference algorithm to bound the work and span in a pure, parallel, and first-order programming language. In such a restricted setting, they can automatically infer polynomial bounds (their tool does not support logarithmic factors). In their type system, types are equipped with a potential, which can be thought as a bag of credits. Similarly to us, they note the unsoundness of allowing unrestricted duplication of potentials at the par rule site. Contrary to us, they chose a different approach, without duplication. In their system, at the application of the par rule, they type check each sub-expression twice, with potentially different context-splitting of potentials, but with the same result type (and hence same potential) for each typing. The first typing is made with typing rules in which operations incur cost, whereas the second is made using a "cost-free" semantics, in which potentials

are never consumed. The soundness argument comes from the fact that at runtime, one of the two sub-expressions will generate the heaviest path, and hence, one of the pairs of typing derivations (one with cost, and the other cost-free) can be used to validate the original bound. Such an approach is not desirable in a program logic, where we want to verify a program only once.

Baillot and Ghyselen [4] propose a type system with sized types for the work and span of programs written in the Pi-calculus, a formal model of parallelism without shared memory but in which tasks (or processes) can exchange data using channels. Their approach is more direct than ours: in their system, the span of the parallel composition of two processes is computed as the maximum of the span of each process; whereas in Parcas, the duplication of span credits implicitly encodes the maximum operation.

Niu et al. [54] and Grodin et al. [25] present Calf and Decalf, cost-aware logical frameworks based on type theory, which allow for proving refinements between program behaviors as well as between their costs. Calf supports parallelism and offers a way to reason about the work and span. However, because they work outside of separation logic, they focus on purely functional code and verify pure variants of some of the parallel algorithms we consider in this paper. Niu et al. [54] verify the work and span of a parallel merge sort implementation, similar to ours (§6.4). In unpublished work, Zhou [61] verifies several implementations of the scan primitive, and in particular a contraction-based implementation similar to ours (§6.2).

Das et al. [22] propose a session-typed system for bounding the work (in their setting, the total number of messages exchanged). They scale their approach to the span in a later paper [21].

Performance properties of concurrent programs. In Section 6.5, we give an API to Treiber's stack in Parcas. Our specifications for push and pop show some performance property. While this is not a formal proof of lock-freedom, it is close to it, and we draw inspiration from prior work. In particular, Hoffmann et al. [33] propose a separation logic for proving lock-freeness using a *quantitative compensation scheme*, which "ensures that a thread is compensated for loop iterations that are caused by progress [...] in another thread". This is related to the pattern followed by our specifications for push and pop, which requires $n - 1$ work credits (n being the number of participants), in order to compensate them for a potential incurred recursion. Jia et al. [39] propose another approach, more amenable to automation, using ghost variables to witness that if a task failed to make progress, then another task has made progress.

8 Conclusion and Future Work

We present Parcas, a separation logic for proving bounds on the work and span of parallel programs. We use work credits, akin to standard time credits, to account for work, and span credits, a new kind of credits that are duplicated upon a par operation, to account for span. Crucially, span credits are tagged with a logical task identifier, restricting their use. We support a key Transfer rule for transferring span credits between tasks when possible. We prove the soundness of Parcas and illustrate it with several case studies, including a higher-order parallel for loop, as well as a parallel merge sort algorithm and Treiber's lock-free stack. All results are mechanized in the Rocq prover using the Iris separation logic framework [48]. As future work, we are interested in three directions. First, we would like to support less-structured parallelism, for example programs written using *futures*, and eventually full-blown concurrency. We posit that the idea of span credits tagged with logical task identifiers will scale to these settings, making use of the "transferable" approach (§4.2). Second, we would like to adapt the expected time credit idea of Haselwarter et al. [28] to allow for proving bounds on the expected value of work and span in parallel randomized algorithms. Third, we are also interested in adapting the ideas of Aguirre et al. [2] for proving bounds with high probability on the span.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 2319168. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] Umut A. Acar, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. 2020. MPL: A High-Performance Compiler for Parallel ML. https://github.com/MPLLang/mpl.

[2] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, ICFP (2024), 284–316. doi:10.1145/3674635

[3] Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2:17 (2011), 1–33. https://lmcs.episciences.org/685/pdf

[4] Patrick Baillot and Alexis Ghyselen. 2022. Types for Complexity of Parallel Computation in Pi-calculus. *ACM Trans. Program. Lang. Syst.* 44, 3, Article 15 (July 2022), 50 pages. doi:10.1145/3495529

[5] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473586

[6] Guy. E. Blelloch. 1990. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*, John H. Reif (Ed.). https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

[7] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a Portable Nested Data-Parallel Language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993*, Marina C. Chen and Robert Halstead (Eds.). ACM, 102–111. doi:10.1145/155332.155343

[8] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 89–102. doi:10.1145/3350755.3400227

[9] Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. *SIGPLAN Not.* 31, 6 (June 1996), 213–225. doi:10.1145/232629.232650

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. doi:10.1145/324133.324234

[11] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*. 259–270. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/permissions_paper.pdf

[12] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[13] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (1974), 201–206. doi:10.1145/321812.321815

[14] Brian Campbell. 2009. Amortised Memory Analysis Using the Depth of Data Structures. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 5502)*. Springer, 190–204. https://homepages.inf.ed.ac.uk/bcampbe2/depth-analysis/depth.pdf

[15] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation (PLDI)*. 270–281. http://flint.cs.yale.edu/flint/publications/veristack.pdf

[16] Arthur Charguéraud. 2024. The CFML tool and library. http://www.charguer.org/softs/cfml/.

[17] Arthur Charguéraud and François Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science, Vol. 9236)*. Springer, 137–153. http://cambium.inria.fr/~fpottier/publis/charguer-pottier-uf.pdf

[18] Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365. http://cambium.inria.fr/~fpottier/publis/charguer-pottier-uf-sltc.pdf

[19] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. 1990. Scan primitives for vector computers. In *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*, Joanne L. Martin, Daniel V. Pryor, and Gary R. Montry (Eds.). IEEE Computer Society, 666–675. doi:10.1109/SUPERC.1990.130084

[20] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science,*

Vol. 8586), Richard E. Jones (Ed.). Springer, 207–231. https://vtss.doc.ic.ac.uk/publications/daRochaPinto2014TaDA.pdf

[21] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.* 2, ICFP (2018), 91:1–91:30. doi:10.1145/3236786

[22] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 305–314. doi:10.1145/3209108.3209146

[23] Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. 2008. Implicitly-threaded parallelism in Manticore. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 119–130. doi:10.1145/1411204.1411224

[24] Stéphane Gimenez and Georg Moser. 2016. The complexity of interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 243–255. doi:10.1145/2837614.2837646

[25] Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proc. ACM Program. Lang.* 8, POPL (2024), 273–301. doi:10.1145/3632852

[26] Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 533–560. http://cambium.inria.fr/~fpottier/publis/gueneau-chargeraud-pottier-esop2018.pdf

[27] Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press. https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

[28] Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. 2024. Tachis: Higher-Order Separation Logic with Credits for Expected Costs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1189–1218. doi:10.1145/3689753

[29] Maximilian P. L. Haslbeck and Tobias Nipkow. 2018. Hoare Logics for Time Bounds: A Study in Meta Theory. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 10805)*. Springer, 155–171. https://www21.in.tum.de/~nipkow/pubs/tacas18.pdf

[30] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. doi:10.1145/3062341.3062354

[31] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[32] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 781–786. http://dx.doi.org/10.1007/978-3-642-31424-7_64

[33] Jan Hoffmann, Michael Marmar, and Zhong Shao. 2013. Quantitative Reasoning for Proving Lock-Freedom. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 124–133. doi:10.1109/LICS.2013.18

[34] Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 132–157. doi:10.1007/978-3-662-46669-8_6

[35] Martin Hofmann. 1999. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *Logic in Computer Science (LICS)*. 464–473. https://doi.org/10.1109/LICS.1999.782641

[36] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Principles of Programming Languages (POPL)*. 185–197. http://www2.tcs.ifi.lmu.de/~jost/research/POPL_2003_Jost_Hofmann.pdf

[37] Iris. 2026. `iris.base_logic.lib.gen_heap`. https://plv.mpi-sws.org/coqdoc/iris/iris.base_logic.lib.gen_heap.html.

[38] Iris Developpement Team. 2026. The atomic triple library of the Iris framework. https://plv.mpi-sws.org/coqdoc/iris/iris.program_logic.atomic.html.

[39] Xiao Jia, Wei Li, and Viktor Vafeiadis. 2015. Proving Lock-Freedom Easily and Automatically. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, Xavier Leroy and Alwen Tiu (Eds.). ACM, 119–127. doi:10.1145/2676724.2693179

[40] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

[41] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages*

(POPL). 637–650. http://plv.mpi-sws.org/iris/paper.pdf

[42] Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2025. Modular Reasoning about Error Bounds for Concurrent Probabilistic Programs. *Proc. ACM Program. Lang.* 9, ICFP (2025), 276–305. doi:10.1145/3747514

[43] Jean-Marie Madiot and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 718–747. http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf

[44] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2022. Specification and verification of a transient stack. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 82–99. doi:10.1145/3497775.3503677

[45] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2025. Will It Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection. *ACM Trans. Program. Lang. Syst.* 47, 1 (2025), 3:1–3:71. doi:10.1145/3716312

[46] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 718–747. https://doi.org/10.1145/3571218

[47] Alexandre Moine, Sam Westrick, and Stephanie Balzer. 2024. DisLog: A Separation Logic for Disentanglement. *Proc. ACM Program. Lang.* 8, POPL, Article 11 (Jan. 2024), 30 pages. doi:10.1145/3632853

[48] Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. A Separation Logic for Parallel Time Complexity with Work and Span Credits (Artifact). Zenodo. doi:10.5281/zenodo.20432258 Also available at https://github.com/nobrakal/parcas/.

[49] MPL Contributors. 2026. mpllib: A grab-bag library for the MPL compiler. https://github.com/MPLLang/mpllib.

[50] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 1–27. http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-time-in-iris-2019.pdf

[51] Hanne Riis Nielson. 1984. *Hoare logic's for run-time analysis of programs.* Ph. D. Dissertation. The University of Edinburgh.

[52] Hanne Riis Nielson. 1987. A Hoare-Like Proof System for Analysing the Computation Time of Programs. *Sci. Comput. Program.* 9, 2 (1987), 107–136. doi:10.1016/0167-6423(87)90029-3

[53] Yue Niu and Jan Hoffmann. 2018. Automatic Space Bound Analysis for Functional Programs with Garbage Collection. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR) (EPiC Series in Computing, Vol. 57)*. 543–563. https://easychair.org/publications/paper/dcnD

[54] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL, Article 9 (Jan. 2022), 31 pages. doi:10.1145/3498670

[55] Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*. http://cambium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity.pdf

[56] François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL (2024), 1482–1508. doi:10.1145/3632892

[57] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30. doi:10.1145/3408995

[58] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

[59] R. Kent Treiber. 1986. Systems programming: Coping with parallelism. https://dominoweb.draco.res.ibm.com/reports/rj5118.pdf

[60] Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement detection with near-zero cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (Aug. 2022), 32 pages. doi:10.1145/3547646

[61] Andrew Zhou. 2025. Formally Verified Cost of the Parallel Prefix Sum Algorithm. (2025). https://www.cs.cmu.edu/~runmingl/student/zhou-scan.pdf

Received 2026-03-01; accepted 2026-05-13