# ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations

Kiko Fernandez-Reyes, Dave Clarke, Daniel Mccain



# ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations*

Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain

Department of Information Technology
Uppsala University, Uppsala, Sweden

**Abstract.** The ubiquity of multicore computers has forced programming language designers to rethink how languages express parallelism and concurrency. This has resulted in new language constructs and new combinations or revisions of existing constructs. In this line, we extended the programming languages Encore (actor-based), and Clojure (functional) with an asynchronous parallel abstraction called ParT, a data structure that can dually be seen as a collection of asynchronous values (integrating with futures) or a handle to a parallel computation, plus a collection of combinators for manipulating the data structure. The combinators can express parallel pipelines and speculative parallelism. This paper presents a typed calculus capturing the essence of ParT, abstracting away from details of the Encore and Clojure programming languages. The calculus includes tasks, futures, and combinators similar to those of Orc but implemented in a non-blocking fashion. Furthermore, the calculus strongly mimics how ParT is implemented, and it can serve as the basis for adaptation of ParT into different languages and for further extensions.

## 1 Introduction

The ubiquity of multicore computers has forced programming language designers to rethink how languages express parallelism and concurrency. This has resulted in new language constructs that, for instance, increase the degree of asynchrony while exploiting parallelism. A promising direction is programming languages with constructs for tasks and actors, such as Clojure and Scala [8, 16], due to the lightweight overhead of spawning parallel computations. These languages offer coarse-grained parallelism at the task and actor level, where futures act as synchronisation points. However, these languages are lacking in high-level coordination constructs over these asynchronous computations. For instance, it is not easy to express dependence on the first result returned via a bunch of futures and to safely terminate the computations associated with the other futures. The task of terminating speculative parallelism is quite delicate, as the futures may have attached parallel computations that depend on other futures, creating complex dependency patterns that need to be tracked down and terminated.

* Partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations.

To address this need, this paper presents the design and implementation of ParT, a non-blocking abstraction that asynchronously exploits futures and enables the developer to build complex, data parallel coordination workflows using high-level constructs. These high-level constructs are derived from the combinators of the orchestration language Orc [11, 12]. ParT is formally expressed in terms of a calculus that, rather than being at a high level of abstraction, strongly mimics how this asynchronous abstraction is implemented and is general enough to be applied to programming languages with notions of futures.

The contributions of the paper are as follows: the design of an asynchronous parallel data abstraction to coordinate complex workflows, including pipeline and speculative parallelism, *and* a typed, non-blocking calculus modelling this abstraction, which integrates futures, tasks and Orc-like combinators, supports the separation of the realisation of parallelism (via tasks) from its specification, and offers a novel approach to terminating speculative parallelism.

## 2 Overview

To set the scene for this paper, we begin with a brief overview to asynchronous computations with futures and provide an informal description of the ParT abstraction and its combinators. A SAT solver example is used as an illustration.

In languages with notions of tasks and active objects [2, 8, 16], asynchronous computations are created by spawning tasks or calling methods on active objects. These computations can exploit parallelism by decoupling the execution of the caller and the callee [7]. The result of a spawn or method call is immediately a future, a container that will eventually hold the result of the asynchronous computation. A future that has received a value is said to be *fulfilled*. Operations on futures may be blocking, such as getting the result from a future, or may be asynchronous, such as attaching a callback to a future. This second operation, called *future chaining* and represented by $f \rightsquigarrow \mathit{callback}$, immediately returns a new future, which will contain the result of applying the callback function *callback* to the contents of the original future after it has been fulfilled. A future can also be thought of as a handle to an asynchronous computation that can be extended via future chaining or even terminated. This is an useful perspective that we will further develop in this work. In languages with notions of actors, such as Clojure and Encore [2], asynchrony is the rule and blocking on futures suffers a large performance penalty. But creating complex coordination patterns based on a collection of asynchronous computations without blocking threads (to maintain the throughput of the system) is no easy task.

To address this need, we have designed an abstraction, called ParT, which can be thought of as a handle to an ongoing parallel computation, allowing the parallel computation to be manipulated, extended, and terminated. A ParT is a functional data structure, represented by type $\mathit{Par}\ t$, that can be empty $\{\} :: \mathit{Par}\ t$, contain a single expression $\{-\} :: t \to \mathit{Par}\ t$, or futures attached to computations producing values, using $(-)^\circ :: \mathit{Fut}\ t \to \mathit{Par}\ t$, or computations producing ParTs, embedded using $(-)^\dagger :: \mathit{Fut}\ (\mathit{Par}\ t) \to \mathit{Par}\ t$. Multiple ParTs

```
1  def fut2Par(f: Fut(Maybe a)): Par a
2     (f ⇝ \(m: Maybe a) ->
3              match m with Nothing => {}; Just val => {val})†
4
5  def evaluateFormula(form: Formula, a: Assignment): (Maybe bool, Assignment)
6      ...
7
8  def sat(st: Strategy, fml: Formula, a: Assignment): Par Assignment
9      let variable = st.getVariable(fml, a)
10         a1 = a.extendAssignment(variable, true)
11         a2 = a.extendAssignment(variable, false)
12     in
13        ({evaluateFormula(fml, a1)} || {evaluateFormula(fml, a2)}) >>=
14           \(result: (Maybe bool, Assignment)) ->
15               match result with
16                   (Nothing, ar) => sat(st, fml, ar);
17                   (Just true, ar) => {ar};
18                   (Just false, ar) => {};
19
20 def process(sts: [Strategy], fml: Formula): Par Assignment
21     fut2Par << (each(sts) >>= \(s: Strategy) ->
22                  (async sat(s, fml, new Assignment()))†)
```

Fig. 1: A SAT solver in Encore.

can be combined using the par constructor, $\| :: Par\ t \to Par\ t \to Par\ t$. This constructor does not necessarily create new parallel threads of control, as this would likely have a negative impact on performance, but rather specifies that parallelism is available. The scheduler in the ParT implementation can choose to spawn new tasks as it sees fit — this is modelled in the calculus as a single rule that nondeterministically spawns a task from a par (rule Red-Schedule).

The combinators can express complex coordination patterns and operate on them in a non-blocking manner, and safely terminate speculative parallelism even in the presence of complex workflows. These combinators will be illustrated using an example, then explained in more detail.

*Illustrative example.* Consider a portfolio-based SAT solver (Fig. 1), which creates numerous strategies, of which each finds an assignment of variables to Boolean values for a given proposition, runs them in parallel, and accepts the first solution found. Each strategy tries to find a solution by selecting a variable and creating two instances of the formula, one where the variable is assigned true, the other where it is assigned false (called splitting) — strategies differ in the order they select variables for splitting. These new instances can potentially be solved in parallel.

The example starts in function `process` (line 20) which receives an array of strategies and the formula to solve. Strategies do not interact with each other and can be lifted to a ParT, creating a parallel pipeline (line 21) using the `each`

and bind ($\gg=$) combinators. As soon as one strategy finds an assignment, the remaining computations are terminated via the prune ($\ll$) combinator.

For each strategy, a call to the `sat` function (line 8) is made in parallel using a call to `async`, which in this case returns a value of type *Fut* (*Par* `Assignment`). Function `sat` takes three arguments: a strategy, a formula and an assignment object containing the current mapping from variables to values. This function uses the strategy object to determine which variable to split next, extends the assignment with new valuations (lines 9–11), recursively solves the formula (by again calling `sat`), and returns an assignment object if successful. The evaluation of the formula, `evaluateFormula` returns, firstly, an optional Boolean to indicate whether evaluation has completed, and if it has completed, whether the formula is satisfiable, and secondly, the current (partial) variable assignment. The two calls to `evaluateFormula` are grouped into a new ParT collection (using $||$) and, with the use of the $\gg=$ combinator, a new asynchronous pipeline is created to either further evaluate the formula by calling `sat`, to return the assignment in the case that a formula is satisfiable as a singleton ParT, or $\{\}$ when the assignment does not satisfy the formula (lines 14–18).

Finally, returning back to `process`, the prune combinator ($\ll$) (line 21) is used to select the first result returned by the recursive calls to `sat`, if there is one. This result is converted from an option type to an empty or singleton ParT collection (again asynchronously), which can then be used in a larger parallel operation, if so desired. The prune combinator will begin poisoning and safely terminating the no longer needed parallel computations, which in this case will be an ongoing parallel pipeline of calls to `sat` and `evaluateFormula`.

*ParT Combinators.* The combinators are now described in detail. The combinators manipulate ParT collections and were derived from Orc [11,12], although in our setting, they are typed and redefined to be completely asynchronous, never blocking the thread. Primitive combinators express coordination patterns such as pipeline and speculative parallelism, and more complex patterns can be expressed based on these primitives.

Pipeline parallelism is expressed in ParT with the sequence and bind combinators. The sequence combinator, $\gg :: \mathit{Par}\ t \to (t \to t') \to \mathit{Par}\ t'$, takes a ParT collection and applies the function to each element in the collection, potentially in parallel, returning a new ParT collection. The bind combinator (derived from other combinators) $\gg= :: \mathit{Par}\ t \to (t \to \mathit{Par}\ t') \to \mathit{Par}\ t'$ is similar to the sequence combinator, except that the function returns a ParT collection and the resulting nested ParT collection is flattened. (*Par* is a monad![1]) In the presence of futures inside a ParT collection, these combinators use the future chaining operation to create independent and asynchronous pipelines of work.

Speculative parallelism is realised by the peek combinator, `peek` $:: \mathit{Par}\ t \to \mathit{Fut}\ (\mathit{Maybe}\ t)$, which sets up a speculative computation, asynchronously waits for a single result to be produced, and then safely terminates the speculative work. To terminate speculative work the ParT abstraction poison these specu-

[1] The monad operations on *Par* are essentially the same as for lists but parallelised.

lative computations, which may have long parallel pipelines to which the poison spreads recursively, producing a pandemic infection among futures, tasks and pipelines of computations. Afterwards, poisoned computations that are no longer needed can safely be terminated. Metaphorically, this is analogous to a tracing garbage collector.

The value produced by `peek` is a future to an option type. The option type is used to capture whether the parallel collection was empty or not. The empty collection {} results in `Nothing`, and a non-empty collection results in a `Just` $v$, where $v$ is the first value produced. The conversion to option type is required because ParTs cannot be tested for emptiness without blocking. The `peek` combinator is an internal combinator, i.e., it is not available to the developer and is used by the prune $\ll$ combinator (explained below).

Built on top of `peek` is the prune combinator, $\ll :: (\mathit{Fut}\ (\mathit{Maybe}\ t) \rightarrow \mathit{Par}\ t') \rightarrow \mathit{Par}\ t \rightarrow \mathit{Par}\ t'$, which applies a function in parallel to the future produced by `peek`, and returns a parallel computation.

Powerful combinators can be derived from the ones mentioned above. An example of a derived combinator, which is a primitive in Orc, is the otherwise combinator, $\asymp :: \mathit{Par}\ t \rightarrow \mathit{Par}\ t \rightarrow \mathit{Par}\ t$ (derivation is shown in Section 3.1). Expression $e_1 \asymp e_2$ results in $e_1$ unless it is an empty ParT, in which case it results in $e_2$.

Other ParT combinators are available. For instance, `each` $:: [t] \rightarrow \mathit{Par}\ t$ and `extract` $:: \mathit{Par}\ t \rightarrow [t]$ convert between sequential (arrays) and ParTs. The latter potentially requires a lot of synchronisation, as all the values in the collection need to be realised. Both have been omitted from the formalism, because neither presents any real technical challenge — the key properties of the formalism, namely, deadlock-freedom, type preservation and task safety (Section 3.5), still hold with these extensions in place.

## 3 A Typed ParT Calculus

This section presents the operational semantics and type system of a task-based language containing the ParT abstraction. The formal model is roughly based on the Encore formal semantics [2, 5], with many irrelevant details omitted.

### 3.1 Syntax

The core language (Fig. 2) contains expressions $e$ and values $v$. Values include constants $c$, variables, futures $f$, lambda abstractions, and ParT collections of values. Expressions include values $v$, function application ($e\ e$), task creation, future chaining, and parallel combinators. Tasks are created via the `async` expression, which returns a future. The parallel combinators are those covered in Section 2 ($||$, $\gg$, `peek` and $\ll$), plus some derived combinators, together with the low-level combinator `join` that flattens nested ParT collections. Recall that `peek` is used under-the-hood in the implementation of $\ll$. Status $\pi$ controls how `peek` behaves: when $\pi$ is $\oslash$ and the result in `peek` is an empty ParT collection,

$$
\begin{aligned}
e &::= v \mid e\ e \mid \texttt{async}\ e \mid e \rightsquigarrow e \mid \ \{e\} \mid e \,||\, e \\
&\quad \mid \ e \gg e \mid e \ll e \mid e^{\circ} \mid e^{\dagger} \mid \texttt{join}\ e \mid \texttt{peek}^{\pi}\ e \\
v &::= c \mid f \mid x \mid \lambda x.e \mid \{\} \mid \{v\} \mid f^{\circ} \mid f^{\dagger} \mid v \,||\, v \\
\pi &::= \_ \mid \oslash
\end{aligned}
$$

Fig. 2: Syntax of the language.

the value is discarded and not written to the corresponding future. This status helps to ensure that precisely one speculative computation writes into the future and that a speculative computation fails to produce a value only when all relevant tasks fail to produce a value.

ParT collections are monoids, meaning that the composition operation $e \,||\, e$ is associative and has $\{\}$ as its unit. As such, ParT collections are sequences, though no operations such as getting the first element are available to access them sequentially. As an alternative, adding in commutativity of $||$ would give multiset semantics to the ParT collections — the operational semantics is otherwise unchanged. Two for one!

A number of the constructs are defined by translation into other constructs.

$$
\begin{aligned}
&\texttt{let}\ x = e\ \texttt{in}\ e' \mathrel{\widehat{=}} (\lambda x.e')\ e \\
&e_1 \prec e_2 \mathrel{\widehat{=}} \texttt{let}\ x = e_1\ \texttt{in} \\
&\qquad\qquad (\lambda y.(y \rightsquigarrow (\lambda z.\texttt{match}\ z\ \texttt{with}\ \texttt{Nothing} \to e_2;\ \_ \to x))^{\dagger}) \ll x \\
&e_1 \gg\!= e_2 \mathrel{\widehat{=}} \texttt{join}\ (e_1 \gg e_2) \\
&\texttt{maybe2par} \mathrel{\widehat{=}} \lambda x.\texttt{match}\ x\ \texttt{with}\ \texttt{Nothing} \to \{\};\ \texttt{Just}\ y \to \{y\}
\end{aligned}
$$

The encoding of `let` is standard. In $e_1 \prec e_2$, pruning $\ll$ is used to test the emptyness of $e_1$. If it is not empty, the result of $e_1$ is returned, otherwise the result is $e_2$. The definition of $\gg\!=$ is a standard definition of monadic bind in terms of map ($\gg$) and join. We assume for convenience a Maybe type and pattern matching on it.

### 3.2 Configurations

Running programs are represented by configurations (Fig. 3). Configurations can refer to the global system or a partial view of the system. A global configuration $\{\mathit{config}\}$ captures the complete global state, e.g., $\{(\texttt{fut}_f)\ (\texttt{task}_f\ e)\}$ shows a global system containing a single task running expression $e$. Local configurations, written as *config*, show a partial view of the state of the program. These are multisets of tasks, futures, poison and future chains. The empty configuration is represented by $\epsilon$. Future configurations, $(\texttt{fut}_f)$ and $(\texttt{fut}_f\ v)$, represent unfulfilled and fulfilled futures, respectively. Poison is the configuration $(\texttt{poison}\ f)$ that will eventually terminate tasks and chains writing to future $f$ and their dependencies. A running task $(\texttt{task}_f^{\alpha}\ e)$ has a body $e$ and will write its result to

$$
\begin{aligned}
\mathit{gconfig} &::= \{\mathit{config}\} \\
\mathit{config} &::= \epsilon \mid (\mathtt{fut}_f) \mid (\mathtt{fut}_f\ v) \mid (\mathtt{poison}\ f) \mid (\mathtt{task}_f^{\alpha}\ e) \mid (\mathtt{chain}_f^{\alpha}\ g\ e) \mid \mathit{config}\ \mathit{config} \\
\alpha &::= \text{␣} \mid \Mightarrow
\end{aligned}
$$

Fig. 3: Runtime configurations.

future $f$. The chain configuration $(\mathtt{chain}_f^{\alpha}\ g\ e)$ depends on future $g$ that, when fulfilled, will then run expression $e$ on the value stored in $g$, and write its value into future $f$. Concatenation of configurations, $\mathit{config}\ \mathit{config}'$, is associative and commutative with the empty configuration $\epsilon$ as its unit (Fig. 12).

Tasks and chains have a flag $\alpha$ that indicates the poisoned state of the computation. Whitespace '␣' indicates that the computation has not been poisoned, and $\Mightarrow$ indicates that the computation has been poisoned and can be safely terminated, if it is not needed (see Rule RED-TERMINATE of Fig. 10).

The initial configuration to evaluate expression $e$ is $\{(\mathtt{task}_f\ e)\ (\mathtt{fut}_f)\}$, where the value written into future $f$ is the result of the expression.

### 3.3 Reduction Rules

The operational semantics is based on a small-step, reduction-context based rules for evaluation within tasks, and parallel reduction rules for evaluation across configurations. Evaluation is captured by expression-level evaluation context $E$ containing a hole $\bullet$ that marks where the next step of the reduction will occur (Fig. 4). Plugging an expression $e$ into an evaluation context $E$, denoted $E[e]$, represents both the subexpression to be evaluated next and the result of reducing that subexpression in context, in the standard fashion [21].

$$
\begin{aligned}
E ::= \bullet \mid E\ e \mid v\ E \mid E \rightsquigarrow e \mid v \rightsquigarrow E \mid \{E\} \mid E \,||\, e \mid v \,||\, E \mid E \gg e \mid v \gg E \\
\mid\ E \ll e \mid E^{\circ} \mid E^{\dagger} \mid \mathtt{join}\ E \mid \mathtt{peek}^{\pi} E
\end{aligned}
$$

Fig. 4: Expression-level evaluation contexts.

Reduction of configurations is denoted $\mathit{config} \to \mathit{config}'$, which states that $\mathit{config}$ reduces in a single step to $\mathit{config}'$.

*Core Expressions.* The core reduction rules (Fig. 5) for functions, tasks and futures are well-known or derived from earlier work [5]. Together, the rules RED-CHAIN and RED-CHAINV describe how future chaining works, initially attaching a closure to a future (via the chain configuration), then evaluating the closure in a new task after the future has been fulfilled.

$$\frac{}{(\texttt{task}_g^\alpha\ E[(\lambda x.e)\ v]) \to (\texttt{task}_g^\alpha\ E[e[v/x]])}\ (\textsc{Red-}\beta\textsc{-Red})$$

$$\frac{\mathit{fresh}\ f}{(\texttt{task}_g^\alpha\ E[\texttt{async}\ e]) \to (\texttt{fut}_f)\ (\texttt{task}_f^\alpha\ e)\ (\texttt{task}_g^\alpha\ E[f])}\ (\textsc{Red-Async})$$

$$\frac{}{(\texttt{task}_f^\alpha\ v)\ (\texttt{fut}_f) \to (\texttt{fut}_f\ v)}\ (\textsc{Red-FutV})$$

$$\frac{\mathit{fresh}\ h}{(\texttt{task}_g^\alpha\ E[f \rightsquigarrow v]) \to (\texttt{fut}_h)\ (\texttt{chain}_h^\alpha\ f\ v)\ (\texttt{task}_g^\alpha\ E[h])}\ (\textsc{Red-Chain})$$

$$\frac{}{(\texttt{chain}_g^\alpha\ f\ e)\ (\texttt{fut}_f\ v) \to (\texttt{task}_g^\alpha\ (e\ v))\ (\texttt{fut}_f\ v)}\ (\textsc{Red-ChainV})$$

Fig. 5: Core reduction rules.

*Sequencing.* The sequencing combinator $\gg$ creates pipeline parallelism. Its semantics are defined inductively on the structure of ParT collections (Fig. 6). The second argument must be a function (tested in function application, but guaranteed by the type system). In Red-SeqS, sequencing an empty ParT results in another empty ParT. A ParT with a value applies the function immediately (Red-SeqV). A lifted future is asynchronously accessed by chaining the function onto it (Red-SeqF). Rule Red-SeqP recursively applies $\gg v$ to the two sub-collections. A future whose content is a ParT collection chains a recursive call to $\gg v$ onto the future and lifts the result back into a ParT collection (Red-SeqFP).

$$\frac{}{(\texttt{task}_g^\alpha\ E[\{\} \gg v]) \to (\texttt{task}_g^\alpha\ E[\{\}])}\ (\textsc{Red-SeqS})$$

$$\frac{}{(\texttt{task}_g^\alpha\ E[\{v\} \gg v']) \to (\texttt{task}_g^\alpha\ E[\{v'\ v\}])}\ (\textsc{Red-SeqV})$$

$$\frac{}{(\texttt{task}_g^\alpha\ E[f^\circ \gg v]) \to (\texttt{task}_g^\alpha\ E[(f \rightsquigarrow v)^\circ])}\ (\textsc{Red-SeqF})$$

$$\frac{}{(\texttt{task}_g^\alpha\ E[f^\dagger \gg v]) \to (\texttt{task}_g^\alpha\ E[(f \rightsquigarrow (\lambda x.x \gg v))^\dagger])}\ (\textsc{Red-SeqFP})$$

$$\frac{}{(\texttt{task}_g^\alpha\ E[(v_1 \,||\, v_2) \gg v]) \to (\texttt{task}_g^\alpha\ E[(v_1 \gg v) \,||\, (v_2 \gg v)])}\ (\textsc{Red-SeqP})$$

Fig. 6: Reduction rules for the sequence $\gg$ combinator.

*Join.* The `join` combinator flattens nested ParT collections of type *Par* (*Par t*) (Fig. 7). Empty collections flatten to empty collections (Red-JoinS). Rule Red-JoinV extracts the singleton value from a collection. A lifted future that contains a ParT (type *Fut* (*Par t*)) is simply lifted to a ParT collection (Red-JoinF). In Red-JoinFP, a future containing a nested ParT collection (type *Fut* (*Par* (*Par t*))), chains a call to `join` to flatten the inner structure. Rule Red-JoinP applies the `join` combinator recursively to the values in the ParT collection.

*Prune and Peek.* Pruning is the most complicated part of the calculus, though most of the work is done using the `peek` combinator (Fig. 8). Firstly, rule Red-

(Red-JoinS)
$(\texttt{task}_g^\alpha\ E[\texttt{join}\ \{\}]) \to (\texttt{task}_g^\alpha\ E[\{\}])$

(Red-JoinV)
$(\texttt{task}_g^\alpha\ E[\texttt{join}\ \{v\}]) \to (\texttt{task}_g^\alpha\ E[v])$

(Red-JoinF)
$(\texttt{task}_g^\alpha\ E[\texttt{join}\ f^\circ]) \to (\texttt{task}_g^\alpha\ E[f^\dagger])$

(Red-JoinFP)
$(\texttt{task}_g^\alpha\ E[\texttt{join}\ f^\dagger]) \to (\texttt{task}_g^\alpha\ E[(f \rightsquigarrow (\lambda x.\texttt{join}\ x))^\dagger])$

(Red-JoinP)
$(\texttt{task}_g^\alpha\ E[\texttt{join}\ (v_1 \,||\, v_2)]) \to (\texttt{task}_g^\alpha\ E[(\texttt{join}\ v_1) \,||\, (\texttt{join}\ v_2)])$

Fig. 7: Reduction rules for the `join` combinator.

(Red-Prune)
*fresh* $f$
$(\texttt{task}_g^\alpha\ E[v \ll v']) \to (\texttt{fut}_f)\ (\texttt{task}_f^\alpha\ (\texttt{peek}\ v'))\ (\texttt{task}_g^\alpha\ E[v\ f])$

(Red-PeekS$^\oslash$)
$(\texttt{task}_g^\alpha\ E[\texttt{peek}^\oslash\ \{\}]) \to \epsilon$

(Red-PeekS)
$(\texttt{task}_g^\alpha\ E[\texttt{peek}\ \{\}])\ (\texttt{fut}_g) \to (\texttt{fut}_g\ \texttt{Nothing})\ (\texttt{poison}\ g)$

(Red-PeekV)
$(\texttt{task}_g^\alpha\ E[\texttt{peek}^\pi\ (\{v\} \,||\, v')])\ (\texttt{fut}_g) \to (\texttt{fut}_g\ (\texttt{Just}\ v))\ (\texttt{poison}\ g) \bigcup_{h \in deps(v')} (\texttt{poison}\ h)$

(Red-PeekF)
$(\texttt{task}_g^\alpha\ E[\texttt{peek}^\pi\ (f^\circ \,||\, v)]) \to (\texttt{chain}_g^\alpha\ f\ (\lambda x.\texttt{peek}^\pi\ \{x\}))\ (\texttt{task}_g^\alpha\ (\texttt{peek}^\oslash\ v))$

(Red-PeekFP)
*fresh* $h$
$(\texttt{task}_g^\alpha\ E[\texttt{peek}^\pi\ (f^\dagger \,||\, v)]) \to$
$(\texttt{chain}_g^\alpha\ f\ (\lambda x.\texttt{peek}^\pi\ (x \,||\, (h \rightsquigarrow \texttt{maybe2par})^\dagger)))$
$(\texttt{fut}_h)\ (\texttt{task}_h^\alpha\ (\texttt{peek}\ v))\ (\texttt{chain}_g^\alpha\ h\ (\lambda x.\texttt{peek}^\oslash(\texttt{maybe2par}\ x)))$

Fig. 8: Reduction rules for pruning. For singleton collections are handled via equality $v = v \,||\, \{\}$.

Prune spawns a new task that will peek the collection $v'$, and passes this new task's future to the function $v$. The essence of the `peek` rules is to set up a bunch of computations that compete to write into a single future, with the strict requirement that `Nothing` is written only when all competing tasks cannot produce a value—that is, when the ParT being peeked is empty. This is challenging due to the lifted future ParTs (type *Fut* (*Par t*)) within a collection, because such a future may be empty, but this fact cannot easily be seen in a non-blocking way. Another challenge is to avoid introducing sequential dependencies between entities that can potentially run in parallel, to avoid, for instance, a non-terminating computation blocking one that will produce a result.

A task that produces a ParT containing a value (rule Red-PeekV) writes the value, wrapped in an option type, into the future and poisons all computations writing into that future, recursively poisoning direct dependencies. The $\oslash$ status on `peek` prevents certain `peek` invocations from writing a final empty result, as in rule Red-PeekS$^\oslash$. Contrast with Red-PeekS, in which a task resulting in

an empty ParT writes `Nothing` into the future — in this case it is guaranteed that no other `peek` exists writing to the future.

A lifted future $f$ is guaranteed to produce a result, though it may not produce it in a timely fashion. This case is handled (rule RED-PEEKF) by chaining a function onto it that will ultimately write into future $g$ when the value is produced, if it wins the race. Otherwise, the result of peeking into $v$ is written into $g$, unless the value produced is $\{\}$ (which is controlled by $\oslash$).

A lifted future to a ParT is not necessarily guaranteed to produce a result, and neither is any ParT that runs in parallel with it. Thus, extra care needs to be taken to ensure that `Nothing` is written if and only if both are actually empty. This is handled in rule RED-PEEKFP. Firstly, a function is chained onto the lifted future to get access to the eventual ParT collection. This is combined with future $h$ that is used to peek into $v$ via a new task.

In all cases, computations propagate the poison state $\alpha$ to new configurations.

*Scheduling.* Rule RED-SCHEDULE (Fig. 9) models the non-deterministic scheduling of parallelism within a task, converting some of the parallelism latent in a ParT collection into a new task. Apart from this rule, expressions within tasks are evaluated sequentially.

$$\frac{\textit{fresh } f}{(\texttt{task}_g^\alpha\ E[e_1 \,||\, e_2]) \to (\texttt{task}_g^\alpha\ E[e_1 \,||\, f^\dagger])\ (\texttt{fut}_f)\ (\texttt{task}_f^\alpha\ e_2)}\ \text{(RED-SCHEDULE)}$$

Fig. 9: Spawning of tasks inside a ParT.

*Poisoning and Termination.* The rules for poisoning and termination (Fig. 10) are based on a *poisoned carrier configuration* defined as $(PC_f^\alpha\ e) ::= (\texttt{task}_f^\alpha\ e) \mid (\texttt{chain}_f^\alpha\ g\ e)$; these rules rely on the definition of when a future is needed (Definition 2), which in turn is defined in terms of the futures on which a task depends to produce a value (Definition 1).

**Definition 1.** *The dependencies of an expression $e$, $deps(e)$, is the set of the futures upon which the computation of $e$ depends in order to produce a value:*

$$deps(f) = \{f\}$$
$$deps(c) = deps(\{\}) = deps(x) = \varnothing$$
$$deps(\{e\}) = deps(\lambda x.e) = deps(\texttt{async}\ e) = deps(e^\circ) = deps(e^\dagger) =$$
$$\quad deps(\texttt{peek}^\pi\ e) = deps(\texttt{join}\ e) = deps(e)$$
$$deps(e \,||\, e') = deps(e\ e') = deps(e \gg e') = deps(e \ggg e') =$$
$$\quad deps(e \prec e') = deps(e \rightsquigarrow e') = deps(e \ll e') = deps(e) \cup deps(e')$$
$$deps((\texttt{task}_f^\alpha\ e)) = deps(e)$$
$$deps((\texttt{chain}_f^\alpha\ g\ e)) = \{g\} \cup deps(e).$$

**Definition 2.** *A future $f$ is* needed *in configuration config, denoted $config \vdash needed(f)$, whenever some other element of the configuration depends on it:*

$$config \vdash needed(f) \text{ iff } (PC_g^\alpha\ e) \in config \wedge f \in deps((PC_g^\alpha\ e)) \wedge (\mathtt{fut}_f) \in config.$$

$$\frac{}{(\mathtt{poison}\ f)\ (PC_f\ e) \to (\mathtt{poison}\ f)\ (PC_f^{\Uparrow}\ e) \bigcup_{g \in deps((PC_f\ e))} (\mathtt{poison}\ g)}\ (\text{RED-POISON})$$

$$\frac{\neg(config \vdash needed(f))}{\{(PC_f^{\Uparrow}\ e)\ config\} \to \{config\}}\ (\text{RED-TERMINATE})$$

Fig. 10: Poisoning reduction rules.

Configurations go through a two step process before being terminated. In the first step (rule RED-POISON) the poisoning of future $f$ poisons any task or chain writing to $f$, marks it with $\Uparrow$, and the poison is transmitted to the direct dependencies of the expression $e$ in the task or chain. In the second step (RED-TERMINATE), a poisoned configuration is terminated when there is no other configuration relying on its result — that is, a poisoned task or chain is terminated if there is no expression around to keep it alive. This rule is global, referring to the entire configuration. Termination can be implemented using tracing garbage collection, though in the semantics a more global specification of dependency is used.

An example (Fig. 11) illustrates how poisoning and termination work to prevent a task that is still needed from being terminated. Initially, there is a bunch of tasks (squares) and futures (circles) (Fig. 11A), where one of the tasks completes and writes a value to future $f$. This causes all of the other tasks writing to $f$ to be poisoned, via Rule RED-PEEKV (Fig. 11B). After application of rule RED-POISON, the dependent tasks and futures are recursively poisoned (Fig. 11C). Finally, the application of rule RED-TERMINATE terminates tasks that are not needed (Fig. 11D). Task $e_1$ is not terminated, as future $g$ is required by the task computing $e\ g$.

*Configurations.* The concatenation operation on configurations is commutative and associative and has the empty configuration as its unit (Fig. 12). We assume that these equivalences, along with the monoid axioms for $\|$, can be applied at any time during reduction.

The reduction rules for configurations (Fig. 13) have the individual configuration reduction rules at their heart, along with standard rules for parallel evaluation of non-conflicting sub-configurations, as is standard in rewriting logic [14].

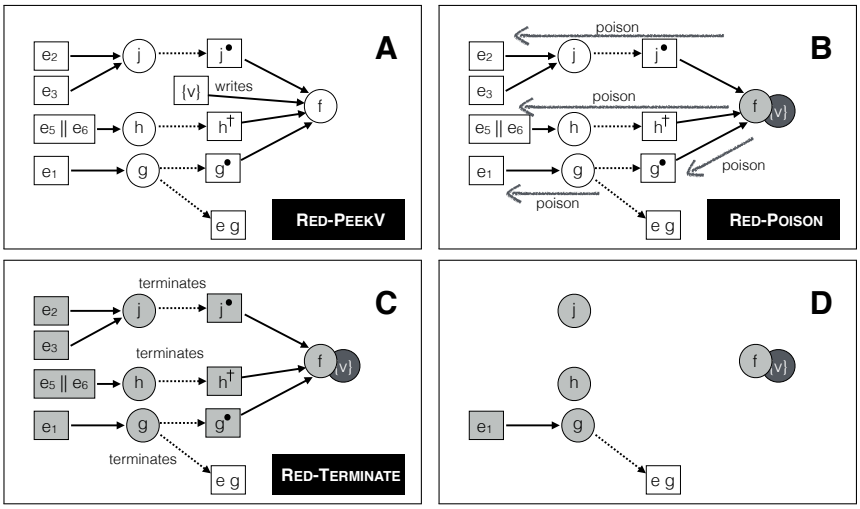


Fig. 11: Safely poisoning and terminating a configuration. The letter in the top right corner indicates the order. Tasks are represented by squares, contain a body and have an arrow to the future they write to. Futures (circles) have dotted arrows to tasks that use them. Grey represents poisoned configurations. Terminated configurations are removed.

$$\epsilon\ \mathit{config} \equiv \mathit{config}\ \epsilon \equiv \mathit{config} \qquad \mathit{config}\ \mathit{config}' \equiv \mathit{config}'\ \mathit{config}$$

$$\mathit{config}\ (\mathit{config}'\ \mathit{config}'') \equiv (\mathit{config}\ \mathit{config}')\ \mathit{config}'' \qquad \frac{\mathit{config} \equiv \mathit{config}'}{\{\mathit{config}\} \equiv \{\mathit{config}'\}}$$

Fig. 12: Configuration equivalence modulo associativity and commutativity.

### 3.4 Type System

The type system (Fig. 14) assigns the following types to terms:

$$\tau ::= K \mid \mathit{Fut}\ \tau \mid \mathit{Par}\ \tau \mid \mathit{Maybe}\ \tau \mid \tau \to \tau$$

where $K$ represents the basic types, $\mathit{Fut}\ \tau$ is the type of a future containing a value of type $\tau$, $\mathit{Par}\ \tau$ is the type of a ParT collection of type $\tau$, $\mathit{Maybe}\ \tau$ represents an option type, and $\tau \to \tau$ represents function types. We also let $\rho$ range over types.

The key judgement in the type system is $\Gamma \vdash_\rho e : \tau$ which asserts that, in typing context $\Gamma$, the expression $e$ is a well-formed term with type $\tau$, where $\rho$ is the expected return type of the task in which this expression appears — $\rho$ is required to type `peek`. The typing context contains the types of both free variables and futures.

Rule TS-Async gives the type for task creation and rule TS-Chain shows how to operate on such values — future chaining has the type of map for the

$$\frac{config \to config'}{config\ config'' \to config'\ config''} \qquad \frac{config_0 \to config_0' \quad config_1 \to config_1'}{config_0\ config_1 \to config_0'\ config_1'}$$

$$\frac{config_0\ config'' \to config_0'\ config'' \quad config_1\ config'' \to config_1'\ config''}{config_0\ config_1\ config'' \to config_0'\ config_1'\ config''}$$

$$\frac{config \to config'}{\{config\} \to \{config'\}}$$

Fig. 13: Configuration reduction rules

(TS-Const)
$$\frac{c \text{ is a constant of type } \tau}{\Gamma \vdash_\rho c : \tau}$$

(TS-Fut)
$$\frac{f : Fut\ \tau \in \Gamma}{\Gamma \vdash_\rho f : Fut\ \tau}$$

(TS-X)
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_\rho x : \tau}$$

(TS-App)
$$\frac{\Gamma \vdash_\rho e_1 : \tau' \to \tau \quad \Gamma \vdash_\rho e_2 : \tau'}{\Gamma \vdash_\rho e_1\ e_2 : \tau}$$

(TS-Fun)
$$\frac{\Gamma,\ x : \tau \vdash_\rho e : \tau'}{\Gamma \vdash_\rho \lambda x.e : \tau \to \tau'}$$

(TS-Async)
$$\frac{\Gamma \vdash_\rho e : \tau}{\Gamma \vdash_\rho \texttt{async}\ e : Fut\ \tau}$$

(TS-Chain)
$$\frac{\Gamma \vdash_\rho e_1 : Fut\ \tau' \quad \Gamma \vdash_\rho e_2 : \tau' \to \tau}{\Gamma \vdash_\rho e_1 \rightsquigarrow e_2 : Fut\ \tau}$$

(TS-EmptyPar)
$$\frac{}{\Gamma \vdash_\rho \{\} : Par\ \tau}$$

(TS-SingletonPar)
$$\frac{\Gamma \vdash_\rho e : \tau}{\Gamma \vdash_\rho \{e\} : Par\ \tau}$$

(TS-LiftF)
$$\frac{\Gamma \vdash_\rho e : Fut\ \tau}{\Gamma \vdash_\rho e^\circ : Par\ \tau}$$

(TS-LiftFP)
$$\frac{\Gamma \vdash_\rho e : Fut\ (Par\ \tau)}{\Gamma \vdash_\rho e^\dagger : Par\ \tau}$$

(TS-Par)
$$\frac{\Gamma \vdash_\rho e_1 : Par\ \tau \quad \Gamma \vdash_\rho e_2 : Par\ \tau}{\Gamma \vdash_\rho e_1 \,||\, e_2 : Par\ \tau}$$

(TS-Sequence)
$$\frac{\Gamma \vdash_\rho e_1 : Par\ \tau' \quad \Gamma \vdash_\rho e_2 : \tau' \to \tau}{\Gamma \vdash_\rho e_1 \gg e_2 : Par\ \tau}$$

(TS-Join)
$$\frac{\Gamma \vdash_\rho e : Par\ (Par\ \tau)}{\Gamma \vdash_\rho \texttt{join}\ e : Par\ \tau}$$

(TS-Otherwise)
$$\frac{\Gamma \vdash_\rho e_1 : Par\ \tau \quad \Gamma \vdash_\rho e_2 : Par\ \tau}{\Gamma \vdash_\rho e_1 \asymp e_2 : Par\ \tau}$$

(TS-Peek)
$$\frac{\Gamma \vdash_{Maybe\ \rho} e : Par\ \rho}{\Gamma \vdash_{Maybe\ \rho} \texttt{peek}^\pi\ e : \tau}$$

(TS-Prune)
$$\frac{\Gamma \vdash_\rho e_1 : Fut\ (Maybe\ \tau) \to Par\ \tau' \quad \Gamma \vdash_\rho e_2 : Par\ \tau}{\Gamma \vdash_\rho e_1 \ll e_2 : Par\ \tau'}$$

(TS-Bind)
$$\frac{\Gamma \vdash_\rho e_1 : Par\ \tau' \quad \Gamma \vdash_\rho e_2 : \tau' \to Par\ \tau}{\Gamma \vdash_\rho e_1 \gg\!= e_2 : Par\ \tau}$$

Fig. 14: Expression Typing.

*Fut* constructor. Rules TS-EmptyPar, TS-SingletonPar, TS-LiftF, TS-LiftFP, and TS-Par give the typings for constructing ParT collections. Rule TS-Sequence implies that sequencing has the type of map for the *Par* constructor. TS-Bind and TS-Join give $\gg\!=$ and `join` the types of the monadic bind and join operators for the *Par* constructor, respectively. Rule TS-Prune captures the communication between the two parameters via the future passed as an argument to the first parameter — the future will contain the first value of the second parameter if there is one, captured by the *Maybe* type. Rule TS-Peek captures the conversion of the singleton or empty argument of `peek` from *Par* $\rho$ to *Maybe* $\rho$, the expected result type of the surrounding task. Because `peek` terminates the task and does not return locally, its return type can be any type.

Well-formed configurations (Fig. 15) are expressed by the judgement $\Gamma \vdash$ *config* `ok`, where $\Gamma$ is the assumptions about the future types in *config*. Rules

$$\text{(T-Fut)}\quad \frac{f \in dom(\Gamma)}{\Gamma \vdash (\texttt{fut}_f)\ \texttt{ok}} \qquad \text{(T-FutV)}\quad \frac{f : Fut\ \tau \in \Gamma \quad \Gamma \vdash_\tau v : \tau}{\Gamma \vdash (\texttt{fut}_f\ v)\ \texttt{ok}} \qquad \text{(T-Poison)}\quad \frac{f : Fut\ \tau \in \Gamma}{\Gamma \vdash (\texttt{poison}\ f)\ \texttt{ok}}$$

$$\text{(T-Task)}\quad \frac{f : Fut\ \tau \in \Gamma \quad \Gamma \vdash_\tau e : \tau}{\Gamma \vdash (\texttt{task}^{\alpha}_{f}\ e)\ \texttt{ok}} \qquad \text{(T-Chain)}\quad \frac{f_1 : Fut\ \tau_1 \in \Gamma \quad f_2 : Fut\ \tau_2 \in \Gamma \quad \Gamma \vdash_{\tau_2} e : \tau_1 \to \tau_2}{\Gamma \vdash (\texttt{chain}^{\alpha}_{f_2}\ f_1\ e)\ \texttt{ok}}$$

$$\text{(T-Config)}\quad \frac{\Gamma \vdash config_1\ \texttt{ok} \quad \Gamma \vdash config_2\ \texttt{ok} \quad futset(config_1) \cap futset(config_2) = \varnothing}{\Gamma \vdash config_1\ config_2\ \texttt{ok}}$$

$$\text{(T-GConfig)}\quad \frac{\Gamma \vdash config\ \texttt{ok} \quad \mathrm{dom}(\Gamma) = futset(config) \quad \textsc{TaskSafe}(config) \quad \textsc{AcyclicDep}(config)}{\Gamma \vdash \{config\}\ \texttt{ok}}$$

Fig. 15: Configuration Typing.

T-Task and T-Chain propagate the eventual expected result type on the turnstile $\vdash$ when typing the enclosed expression. Rule T-Config depends upon the following definition, a function that collects all futures defined in a configuration:

**Definition 3.** *Define* $futset(config)$ *as:*

$$\begin{aligned} &futset((\texttt{fut}_f)) = futset((\texttt{fut}_f\ v)) = \{f\} \\ &futset((config_1\ config_2)) = futset(config_1) \cup futset(config_2) \\ &futset(\_) = \varnothing. \end{aligned}$$

Rule T-GConfig defines the well-formedness of global configurations, judgement $\Gamma \vdash \{config\}\ \texttt{ok}$. This rule depends on a number of definitions that capture properties on futures and tasks and on the dependency between futures. The invariance of these properties is ultimately used to proof type soundness and other safety properties of the system.

**Definition 4.** *Define the following functions for collecting the different kinds of tasks and chains of a configuration:*

$$\begin{aligned} regular_f(config) &= \{(\texttt{task}_f\ e) \in config \mid e \neq \texttt{peek}^{\pi}\ e'\} \\ &\quad \cup \{(\texttt{chain}_f\ g\ e) \in config \mid e \neq \lambda\_.\texttt{peek}^{\pi}\ e'\} \\ peeker_f(config) &= \{(\texttt{task}_f\ (\texttt{peek}\ e)) \in config\} \\ &\quad \cup \{(\texttt{chain}_f\ g\ (\lambda\_.\texttt{peek}\ e)) \in config\} \\ peeker^{\oslash}_f(config) &= \{(\texttt{task}_f\ (\texttt{peek}^{\oslash}\ e)) \in config\} \\ &\quad \cup \{((\texttt{chain}_f\ g\ (\lambda\_.\texttt{peek}^{\oslash}\ e)) \in config\} \end{aligned}$$

Tasks with no `peek` expression are called *regular tasks*, while *peeker tasks* have the `peek` expression — there are both $\oslash$- and non-$\oslash$-peeker tasks. These functions can be used to partition the tasks and chains in a configuration into these three kinds of tasks and chains. These definitions consider `peek` expressions

only at the top level of a task, although the syntax allows them to be anywhere. Based on the reduction rules, one can prove that `peek` only appears at the top level of a task or chain, so no task or chain is excluded by these definitions.

**Definition 5.** *Define predicate* TaskSafe(*config*) *as follows:*

$$
\begin{aligned}
&\textsc{TaskSafe}(\mathit{config}) \textit{ iff for all } f \in \mathit{futset}(\mathit{config}) \\
&\quad |\mathit{regular}_f(\mathit{config}) \cup \mathit{peeker}_f(\mathit{config})| \leq 1 \\
\wedge\ &(\mathtt{fut}_f) \in \mathit{config} \wedge (\mathtt{poison}\ f) \notin \mathit{config} \Rightarrow \\
&\qquad\qquad |\mathit{regular}_f(\mathit{config}) \cup \mathit{peeker}_f(\mathit{config})| = 1 \\
\wedge\ &|\mathit{regular}_f(\mathit{config})| = 1 \Rightarrow \mathit{peeker}_f(\mathit{config}) \cup \mathit{peeker}^{\oslash}_f(\mathit{config}) = \emptyset \\
\wedge\ &(\mathtt{task}^{\alpha}_f\ (\mathtt{peek}\ \{\})) \in \mathit{config} \Rightarrow (\mathtt{fut}_f) \in \mathit{config} \wedge \mathit{peeker}^{\oslash}_f(\mathit{config}) = \emptyset
\end{aligned}
$$

Predicate TaskSafe(*config*) (Definition 5) describes the structure of the configuration *config*. It states that:

- there is at most one regular or non-⊘-peeker task per future;
- if a future has not yet been fulfilled and it is not poisoned, then there exists exactly one regular task or non-⊘-peeker task that fulfils it;
- regular tasks and peeker tasks do not write to the same futures; and
- if a peeker task that is about to fulfil a future with `Nothing`, then the future is unfulfilled and no ⊘-peeker task fulfilling the same future exists.

The following definition establishes dependencies between futures. Predicate $\mathit{config} \vdash f \lhd g$ holds for all future $g$ whose eventual value could influence the result stored in future $f$.

**Definition 6.** *Define the predicate* $\mathit{config} \vdash f \lhd g$ *as the least transitive relation satisfying the following rules:*

$$
\frac{(\mathtt{task}^{\alpha}_f\ e) \in \mathit{config} \quad g \in \mathit{deps}(e)}{\mathit{config} \vdash f \lhd g} \qquad \frac{(\mathtt{chain}^{\alpha}_f\ h\ e) \in \mathit{config} \quad g \in \mathit{deps}(e) \cup \{h\}}{\mathit{config} \vdash f \lhd g}
$$

$$
\frac{(\mathtt{fut}_f\ v) \in \mathit{config} \quad g \in \mathit{deps}(v)}{\mathit{config} \vdash f \lhd g}
$$

**Definition 7.** *Predicate* AcyclicDep(*config*) *holds iff relation* $\lhd$ *is acyclic, where* $\lhd$ *is defined for config in Definition 6.*

Rule T-GConfig for well-formed global configurations requires that precisely the futures that appear in the typing environment $\Gamma$ appear in the configuration, that the configuration is well-formed, and that it satisfies the properties TaskSafe and AcyclicDep. By including these properties as a part of the well-formedness rule for global configurations, type preservation (Lemma 1) makes these invariants. These invariants on the structure of tasks and the dependency relation together ensure that well-typed configurations are deadlock-free, as we explore next.

### 3.5 Formal Properties

The calculus is sound and deadlock-free. These results extend previous work [15] to address the pruning combinator.

**Lemma 1 (Type Preservation).** *If* $\Gamma \vdash \{config\}$ `ok` *and* $\{config\} \to \{config'\}$, *then there exists a* $\Gamma'$ *such that* $\Gamma' \supseteq \Gamma$ *and* $\Gamma' \vdash \{config'\}$ `ok`.

*Proof. By induction on derivation* $\{config\} \to \{config'\}$. *In particular, the invariance of* AcyclicDep *is shown by considering the changes to the dependencies caused by each reduction rule. The only place where new dependencies are introduced is when new futures are created. Adding a future to the dependency relation cannot introduce cycles.* □

The following lemma states that the notion of *needed*, which determines whether or not to garbage collect a poisoned task or chain, is anti-monotonic, meaning that after a future is no longer needed according to the definitions, it does not subsequently become needed.

**Lemma 2 (Safe Task Kill).** *If* $\Gamma \vdash \{config\}$ `ok` *and* $\{config\} \to \{config'\}$, *then* $\neg(config \vdash needed(f))$ *implies* $\neg(config' \vdash needed(f))$.

*Proof.* A future is initially created in a configuration where it is needed. If ever a future disappears from $deps(e)$, if can never reappear. □

This lemma rules out the situation where a task is poisoned and garbage collected, but is subsequently needed. For instance, the application of rule Red-Terminate in Fig. 11C kills tasks $e_2$, $e_3$, $e_5$ and $e_6$ (shown in Fig 11D). If the future into which these tasks were going to write is needed afterwards, there would be a deadlock as a new task could chain on that future but never be fulfilled.

**Definition 8 (Terminal Configuration).** *A global configuration* $\{config\}$ *is* terminal *iff every element of config has one of the following shapes:* $(\texttt{fut}_f)$, $(\texttt{fut}_f\ v)$ *or* $(\texttt{poison}\ f)$.

**Lemma 3 (Deadlock-Freedom/Progress).** *If* $\Gamma \vdash \{config\}$ `ok`, *then config is a terminal configuration, or there exists a* $config'$ *such that* $\{config\} \to \{config'\}$.

*Proof. By induction on a derivation of* $\{config\} \to \{config'\}$, *relying on the invariance of* AcyclicDep *and Lemma 2.* □

Deadlock-freedom guarantees that some reduction rule can be applied to a well-typed, non terminal, global configuration — this is essentially the progress property required to prove type safety. It implies further that there are no local deadlocks, such as a deadlocked configuration like $(\texttt{chain}_f\ g\ e)\ (\texttt{chain}_g\ f\ e')$. Such a configuration fails to satisfy the AcyclicDep invariant, thus cannot exist. If mutable state is added to the calculus, deadlock-freedom is lost.

*Implementations.* There are two prototypes of the ParT abstraction. In the first prototype,[2] ParT has been written as an extension to the Encore compiler (written in Haskell) and runtime (written in C) but, it can be implemented in well-established languages with notions of tasks and futures. This prototype integrates futures produced by tasks and active objects with the ParT abstraction. The other prototype has been written in Clojure,[3] which is not statically typed. Both prototypes follow the semantics to guide the implementation. In practice, this means that the semantic rules are written in such a way that they can be easily mimicked in a library or in a language runtime.

## 4 Related Work

Our combinators have been adapted from those of the Orc [11,12] programming language. In ParT, these combinators are completely asynchronous and are integrated with futures. ParTs are first class citizens and can be nested *Par* (*Par t*), neither of which is possible in Orc, which sits on top of the expression being coordinated and a flat collection of values.

Meseguer et al. [1] used rewriting logic semantics and Maude to provide a distributed implementation of Orc. Their focus on the semantic model allows them to model check Orc programs. In this paper, our semantics is more fine-grained, and guides the implementation in a multicore setting.

ParT uses a monad to encapsulate asynchronous computations, which is not a new idea [3,13,20]. For instance, F# expresses asynchronous workflows using a continuation monad [20] but cannot create more parallelism within the monad, making the model better suited for event-based programming. In contrast, our approach can spawn parallel computations and include them within ParTs.

Other work implements Orc combinators in terms of a monad within the pure functional language Haskell [3,13]. One of these approaches [3] relies on threads and channels and implements the prune $\ll$ combinator using sequential composition, losing potential parallelism. The other approach [13] uses Haskell threads and continuations to model parallel computations and re-designs the prune $\ll$ combinator in terms of a *cut* combinator thats sparks off parallel computations, waits until there is a value available and terminates, in bulk, the remaining computations. In contrast, the ParT abstraction relies on more lightweight tasks instead of threads, has fully asynchronous combinators, which maintain the throughput of the system, and terminates speculative work by recursively poisoning dependencies and terminating computations that are not needed.

An approach to increase parallelism is to create parallel versions of existing collections. For instance, Haskell [10] adds parallel operations to its collections, and the Scala parallel collections [18] adds new methods to their collection, `par` and `seq`, that return a parallel and a sequential version of the collection. However these approaches cannot coordinate complex workflows, which is possible with the ParT abstraction.

[2] Encore ParT prototype: http://52.50.101.143/kompile/encore/
[3] Clojure ParT prototype: https://github.com/kikofernandez/ParT

Recent approaches to creating pipeline parallelism are the Flowpool [19] and FlumeJava [4] abstractions. In the former, functions are attached to Flowpool and, with the `foreach` combinator, the attached functions are applied to items asynchronously added to the Flowpool thereby creating parallel pipelines of computations. The latter, FlumeJava, is a library extending the MapReduce framework; it provides high-level constructs to create efficient data-parallel pipelines of MapReduce jobs, via an optimisation phase. The ParT abstraction can create data-parallel pipelines with the sequence $\gg$ and bind $\gg=$ combinators (at the moment there is no optimisation phase) and further can terminate speculative work.

Existing approaches to safely terminating speculative parallelism [6,9,17] did not integrate well with the ParT abstraction. For instance, the Cilk programming language provides the `abort` keyword to terminate all speculative work generated by a procedure [6]. The termination does not happen immediately, instead, computations are marked as not-runnable; already running computations would get marked as non-runnable but do not stop execution until their work is finished. In other approaches, the developer specifies termination checkpoints at which a task may be terminated [9, 17]. This solves the previous problem and improves responsiveness but, adds an extra overhead (for the checking) and puts the responsibility on the developer, who specifies the location of the checkpoints. In our design, the developer does not need to specify these checkpoints and speculative work is terminated as soon as there are no dependencies. No other approach considers that the results of tasks may be needed elsewhere.

## 5 Conclusion and Future Work

This paper presented the ParT asynchronous, parallel collection abstraction, and a collection of combinators that operate over it. ParT was formalised as a typed calculus of tasks, futures and Orc-like combinators. A primary characteristic of the calculus is that it captures the non-blocking implementation of the combinators, including an algorithm for pruning that tracks down dependencies and is safe with respect to shared futures. The ParT abstraction has prototypes in the Encore (statically typed) and Clojure (dynamically typed) programming languages.

Currently, the calculus does not support side-effects. These are challenging to deal with, due to potential race conditions and terminated computations leaving objects in an inconsistent state. We expect that Encore's capability type system [2] can be used to avoid data races, and a run-time, transactional mechanism can deal with the inconsistent state. At the start of the paper we mentioned that ParT was integrated into an actor-based language, but the formalism included no actors. This work abstracted away the actors, replacing them by tasks and futures—message sends in the Encore programming language return results via futures—which were crucial for tying together the asynchronous computations underlying a ParT. Actors can easily be re-added as soon as the issues of shared mutable state have been addressed. The distribution aspect of actors has

not yet been considered in Encore or in the ParT abstraction. This would be an interesting topic for future work. Beyond these extensions, we also plan to extend the range of combinators supporting the ParT abstraction.

## References


1. Musab AlTurki and José Meseguer. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In Peter Csaba Ölveczky, editor, *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems, RTRTS 2010, Longyearbyen, Norway, April 6-9, 2010.*, volume 36 of *EPTCS*, pages 26–45, 2010.
2. Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.
3. Marco Devesas Campos and Luís Soares Barbosa. Implementation of an orchestration language as a Haskell domain specific language. *Electr. Notes Theor. Comput. Sci.*, 255:45–64, 2009.
4. Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.
5. Dave Clarke and Tobias Wrigstad. Vats: A safe, reactive storage abstraction. In Erika Ábrahám, Marcello M. Bonsangue, and Einar Broch Johnsen, editors, *Theory and Practice of Formal Methods – Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2016.
6. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223. ACM, 1998.
7. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.
8. Rich Hickey. The Clojure programming language. In Johan Brichau, editor, *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, page 1. ACM, 2008.
9. Shams Imam and Vivek Sarkar. The Eureka programming model for speculative task parallelism. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 421–444. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
10. Simon L. Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In G. Ramalingam, editor, *Programming Languages and Systems, 6th*

*Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, page 138. Springer, 2008.
11. David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *Proceedings of the 17th International Conference on Concurrency Theory*, CONCUR'06, pages 477–491, Berlin, Heidelberg, 2006. Springer-Verlag.
12. David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The Orc programming language. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, FMOODS '09/FORTE '09, pages 1–25, Berlin, Heidelberg, 2009. Springer-Verlag.
13. John Launchbury and Trevor Elliott. Concurrent orchestration in Haskell. In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 79–90. ACM, 2010.
14. Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.
15. Daniel McCain. Parallel combinators for the Encore programming language. Master's thesis, Uppsala University, 2016.
16. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.
17. Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
18. Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, volume 6853 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2011.
19. Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers*, volume 7760 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2012.
20. Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2011.
21. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.