

Chocola: Composable Concurrency Language

JANWILLEM SWALENS, JOERI DE KOSTER, and WOLFGANG DE MEUTER,
Vrije Universiteit Brussel, Belgium

Programmers often combine different concurrency models in a single program, in each part of the program using the model that fits best. Many programming languages, such as Clojure, Scala, and Java, cater to this need by supporting different concurrency models. However, existing programming languages often combine concurrency models in an ad hoc way, and the semantics of the combinations are not always well defined.

This article studies the combination of three concurrency models: futures, transactions, and actors. We show that a naive combination of these models invalidates the guarantees they normally provide, thereby breaking the assumptions of programmers. Hence, we present **Chocola**: a unified language of futures, transactions, and actors that maintains the guarantees of all three models wherever possible, even when they are combined.

We describe and formalize the semantics of this language and prove the guarantees it provides. We also provide an implementation as an extension of Clojure and demonstrated that it can improve the performance of three benchmark applications for relatively little effort from the developer.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages; Concurrent programming languages; Multiparadigm languages; Concurrent programming structures;**

Additional Key Words and Phrases: Futures, software transactional memory, actor model

ACM Reference Format:

Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2021. Chocola: Composable Concurrency Language. *ACM Trans. Program. Lang. Syst.* 42, 4, Article 17 (January 2021), 56 pages.
<https://doi.org/10.1145/3427201>

1 INTRODUCTION

Since the introduction of multicore processors, concurrency and parallelism have become essential aspects of software development. However, programming with concurrency is notoriously difficult [23, 39, 46]. Over the years, a plethora of concurrency models have been invented, including locks and semaphores [22], futures [5], Communicating Sequential Processes [38], actors [1], and transactions [32, 54]. A **concurrency model** provides constructs to introduce parallelism in a program. At the same time, it imposes restrictions on the program to provide **guarantees** to the programmer that prevent common errors. For example, the actor model introduces actors that carry out computations in parallel, but it requires that data is shared only using messages, thereby preventing low-level data races.

Authors' address: J. Swalens, J. De Koster, and W. De Meuter, Vrije Universiteit Brussel, Pleinlaan 2, 1050, Brussels, Belgium; emails: {janwillem.swalens, joeri.de.koster, wolfgang.de.meuter}@vub.be.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

0164-0925/2021/01-ART17 \$15.00

<https://doi.org/10.1145/3427201>

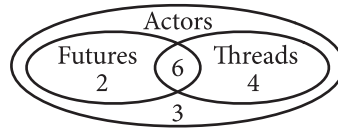


Fig. 1. Tasharofi et al. [60] found that, out of 15 Scala projects that use actors, 12 combine them with futures and/or threads.

Today, several of these concurrency models have found their way into modern mainstream programming languages and frameworks. These often support many different models. For instance, Clojure provides six different concurrency models: futures, promises, atomic variables, transactional memory, channels, and agents; Java supports threads, locks, atomic variables, futures, promises, Fork/Join, and parallel collections; and Haskell supports threads, locks, atomic variables, transactions, and channels.

Tasharofi et al. [60] have shown that developers often *combine multiple models in a single program*: In a sample of 15 GitHub projects in Scala that use the actor model, 12 combined it with another model (illustrated in Figure 1). Eight out of 15 programs used actors and futures and 10 out of 15 programs used actors and threads, including 6 that used all three models. Only 3 out of the 15 programs used only actors.

While programming languages allow their concurrency models to be mixed freely by the developer, and developers do this in practice, unfortunately, doing so correctly is far from trivial. Programming languages often integrate concurrency models in an ad hoc way and the semantics of their combination is not always well defined. We observe that when the language constructs of different concurrency models are combined, their original *guarantees are often invalidated*. In a case study of Clojure [58], we found several such cases. For instance, when a message is sent to a channel in a transaction, and the transaction rolls back, the message is not retracted.

We argue that the combination of multiple concurrency models must be carefully considered. Most concurrency models fall into one of three categories: deterministic, shared-memory, and message-passing models [61]. In this article, we present **Chocola** (for **composable concurrency language**), a programming language that integrates three concurrency models, one from each category: futures, transactions, and actors.¹

The goal of Chocola is to combine these three models and maintain the guarantees of each model wherever possible. We have two requirements. First, when used separately, the semantics of each model should remain unchanged, ensuring backwards compatibility. Second, when models are combined, we seek a semantics for their combination that maintains each model's guarantees. When this is impossible, we replace the original guarantee with a (slightly) less restrictive one.

In this article, we first describe the three concurrency models we studied separately (Section 2). Next, we motivate why it is desirable to combine concurrency models (Section 3) and define the requirements for such a combination (Section 4). Then, we examine each pairwise combination: We describe the problems that appear and create a semantics that satisfies the requirements, thereby defining transactional futures (Section 5), transactional actors (Section 6), and futures for intra-actor parallelism (Section 7). We consolidate these three partial solutions into Chocola and demonstrate it with an example (Section 8). Afterwards, we formalize Chocola's semantics and prove its guarantees (Section 9). We present an implementation as an extension of Clojure (Section 10) and evaluate its performance and expressivity using three benchmark applications (Section 11).

¹Chocola is available online at <http://soft.vub.ac.be/~jswalens/chocola> and <https://github.com/jswalens/chocolib>.

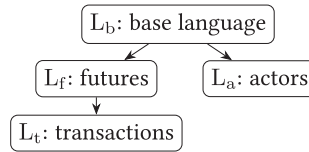


Fig. 2. The relations between the languages presented in the following sections.

2 BACKGROUND: FUTURES, TRANSACTIONS, AND ACTORS

In this section, we describe the three models we studied: futures (Section 2.1), transactions (Section 2.2), and actors (Section 2.3). We illustrate each model building up towards an example that combines all three: a holiday reservation system that books flights, hotels, and cars.

We formalize the semantics of each model in separate languages: L_f for futures, L_t for transactions, and L_a for actors. These are built on top of a base language L_b , as illustrated in Figure 2. We list the guarantees of each model, illustrated throughout the text with boxes like `Det`, and formalize them. Note that we will not prove the guarantees of the separate models in this section, instead we will prove the guarantees of our unified language Chocola in Section 9 and the appendices.

In practice, Chocola extends Clojure, a Lisp-like language that runs on top of Java and supports futures and transactions.² In the formalization, L_b is a (very small) subset of Clojure, without its support for futures or transactions. The languages L_f and L_t model Clojure’s support for futures and transactions, respectively. We also extended Clojure with support for actors, formalized in L_a .

2.1 Futures

A parallel **task** (or thread) is a fragment of the program that can be executed in parallel with the rest of the program. A run-time system schedules tasks over the available processing units (cores or processors). A parallel task is created using the expression `(fork e)`, which begins the evaluation of the expression e in a new task, and immediately returns a future.³ A **future** is a placeholder variable that represents the result of a concurrent computation [5, 31]. Initially, the future is **unresolved**. Once the parallel evaluation of e yields a value v , we say the future is **resolved** to v . Other tasks can retrieve this result by calling `(join f)`. If the future is resolved, this returns its value immediately; if the future is still unresolved, this call blocks until it is resolved and then returns its value.

The following example defines a function `parallel-filter` to filter a list xs using a function f :

```

1 (defn parallel-filter [f xs]
2   (let [parts (partition 8 xs)                ; Divide xs into 8 parts
3         futures (map (fn [p] (fork (filter f p))) parts) ; Fork futures that use the built-in sequential filter
4         results (map (fn [fut] (join fut)) futures)] ; Join futures
5     (apply concat results))                  ; Concatenate results of each part
  
```

This function first uses Clojure’s `partition` function to partition the list into 8 parts. On line 3, we fork a future for each part, which uses Clojure’s built-in `sequential filter` function to filter its part. The syntax `(fn [x] e)` defines an anonymous (lambda) function. On line 4 these futures are joined, and line 5 concatenates their results. This example can therefore exploit up to 8 cores.

²<https://clojure.org>.

³Clojure implements futures as described here except for some syntactical differences, and Scala and Haskell offer similar constructs: `(fork e)` is `(future e)` in Clojure, `Future { e }` in Scala, and `forkIO e` in Haskell. We aim to approximate these languages closely to demonstrate how the problems we describe also apply to them.

Syntax

$c \in \text{Constant} ::= \text{nil} \mid \text{true} \mid \text{false} \mid \emptyset \mid 1 \mid \dots$
 $\quad \mid \text{""} \mid \text{"a"} \mid \dots$
 $x \in \text{Variable}$
 $f \in \text{Future}$
 $v \in \text{Value} ::= c \mid x \mid \text{fn } [\bar{x}] e \mid f$
 $e \in \text{Expression} ::= v \mid e \bar{e} \mid \text{if } e e e \mid \text{let } [x e] e$
 $\quad \mid \text{do } \bar{v}; e \mid \text{fork } e \mid \text{join } e$

Reduction rules

$\text{congruence}|_f \quad T \cup \langle f, \mathcal{E}[e] \rangle \quad \rightarrow_f \quad T \cup \langle f, \mathcal{E}[e'] \rangle \quad \text{if } e \rightarrow_b e'$
 $\text{fork}|_f \quad T \cup \langle f, \mathcal{E}[\text{fork } e] \rangle \quad \rightarrow_f \quad T \cup \langle f, \mathcal{E}[f_*] \rangle \cup \langle f_*, e \rangle \quad \text{with } f_* \text{ fresh}$
 $\text{join}|_f \quad T \cup \langle f, \mathcal{E}[\text{join } f_*] \rangle \cup \langle f_*, v \rangle \quad \rightarrow_f \quad T \cup \langle f, \mathcal{E}[v] \rangle \cup \langle f_*, v \rangle$

Evaluation contexts

$\mathcal{P} ::= T \cup \langle f, \mathcal{E} \rangle$
 $\mathcal{E} ::= \square \mid (\bar{v} \mathcal{E} \bar{e}) \mid \text{if } \mathcal{E} e e \mid \text{let } [x \mathcal{E}] e$
 $\quad \mid \text{do } \bar{v}; \mathcal{E}; \bar{e} \mid \text{join } \mathcal{E}$

State

$\text{Program state } \quad p \quad ::= T$
 $\text{Tasks } \quad T \subset \text{Task}$
 $\text{Task } \quad \text{task} \in \text{Task} ::= \langle f, e \rangle$
 $\text{Initial state } \quad p_0 \quad = \{ \langle f_0, e \rangle \}$

Fig. 3. Operational semantics of L_f , a language with futures.

2.1.1 Guarantee: Determinacy. Futures guarantee **determinacy** $\boxed{\text{Det}}$: For a given input, a program always produces the same output.⁴ This means that a program’s result does not depend on the order in which its tasks are interleaved. Futures are commonly used to parallelize homogeneous operations over lists, e.g., searching and sorting [31], as in the example above. In these cases, determinacy is often desired, as the end result should not depend on how tasks are scheduled.

2.1.2 Formalization of Operational Semantics. Figure 3 defines L_f : an extension of the base language that supports parallel tasks and futures. This formalism is based on the work of Flanagan and Felleisen [25] and Welc et al. [65]. We describe the operational semantics piece by piece.

Syntax. The syntax consists of the base language—supporting function application ($e \bar{e}$), conditionals (if), local variables (let), and blocks (do)—futures, and the expressions fork and join.⁵

State. A program’s state p is a set of tasks. A task is a tuple containing the future that represents its final value and the expression it is currently evaluating. The future f associated with each task is unique and can therefore be used an identifier for the task. To kickstart evaluation, a program e is converted into the initial state $\{ \langle f_0, e \rangle \}$, i.e., it contains one “root” task that executes e .

Evaluation contexts. Evaluation contexts define the evaluation order within expressions. In the program evaluation context \mathcal{P} , an arbitrary task in which a reduction is possible can be chosen; this models that a parallel execution of the program can interleave different tasks in any order. \mathcal{E} is an expression with a “hole \square .” We write $\mathcal{E}[e]$ for the expression obtained by replacing the hole \square with e in \mathcal{E} . In every reduction rule, we highlight the reduced expression in blue.

Reduction rules. We define the operational semantics of L_f using transitions $p \rightarrow_f p'$.

The rule $\text{congruence}|_f$ defines that the base language can be used in each task. Transitions in the base language are written $e \rightarrow_b e'$. They define a standard functional calculus and are fully listed in Appendix B.2.

⁴This is sometimes called observable determinism. Some literature, such as Denning and Dennis [21], distinguishes between *determinism*, which requires that the tasks are interleaved in the same way for each execution, and *determinacy*, which requires only the same output for a given input. We are concerned with determinacy, not determinism.

⁵While our syntax uses S-expressions, we will not repeat the parentheses in every rule.

The rule $\text{fork}|_f$ specifies that the expression $\text{fork } e$ creates a new task in which e will be evaluated and reduces to a freshly created future f . After the expression e has been fully reduced to a value v , $\text{join } f$ will reduce to v . A task can be joined multiple times; each join reduces to the same value. A join can be resolved by the rule $\text{join}|_f$ only if the corresponding task has been fully reduced to a value; this detail encodes the blocking nature of our futures.

2.1.3 Formalization of Determinacy. Intuitively, determinacy means that different evaluations of a program might take different steps during the reduction, but eventually they will all lead to equivalent end results. We first define equivalence.

Definition 2.1 (Equivalence \cong). Our operational semantics generates identifiers in several places: variables (x) and futures (f) in this section, but also transactions (n), transactional variables (r), and actors (a) in the following sections. We say that the two states p_1 and p_2 are **equivalent**, written $p_1 \cong p_2$, if there exists a renaming of these identifiers such that $p_1 = p_2$.

THEOREM 2.2 (DETERMINACY). *If $p_0 \rightarrow^* p_1$ and $p_0 \rightarrow^* p_2$ with p_1 and p_2 final states, then $p_1 \cong p_2$.*

In the special case where this theorem is applied to the first state of a program, i.e., $p_0 = \{\langle f_0, e \rangle\}$ for any program e , it proves determinacy of the whole program.

2.2 Transactions

Software Transactional Memory (STM) is a concurrency model that allows parallel tasks to access shared memory locations [32, 35, 54]. Each shared memory location is encapsulated in a **transactional variable**, which is created using $(\text{ref } v)$ with initial value v . Access to shared memory can occur only in a **transaction**: a block of code ($\text{atomic } e$) in which transactional variables can be read using $(\text{deref } r)$ and modified using $(\text{ref-set } r v)$. A transaction reads a consistent snapshot of the shared memory, and furthermore, the intermediate states of a transaction cannot be observed outside the transaction. STM is implemented in Clojure and in Haskell’s GHC compiler [34]. Chocola re-uses Clojure’s implementation, explained by Halloway [30].

The code below implements a flight reservation system using transactions. Line 1 defines a flight as a mapping of certain symbols (prefixed with a colon) to the corresponding value, encapsulated in a transactional variable. Line 7 contains a transaction that reserves three seats on this flight, by reading the flight and updating its number of available seats.⁶ Encapsulating this in a transaction prevents race conditions: All refs modified in a transaction are updated atomically when the transaction commits.

```

1 (def BA213 (ref {:from "LHR" :to "BOS" :price 499 :seats 243})) ; Define flight as ref containing map
2
3 (defn reserve-seats [flight n]
4   (let [new-seats (- (get (deref flight) :seats) n)           ; Read number of available seats and reserve n
5         new-flight (assoc (deref flight) :seats new-seats)] ; "assoc"iate key :seats with new value
6     (ref-set flight new-flight)) ; Set new value of flight
7 (atomic (reserve-seats BA213 3)) ; Reserve 3 seats on flight, in a transaction

```

In contrast to locks, which are said to be pessimistic, STM is “optimistic” [33]: The code in a transaction is immediately executed. When different transactions access the same transactional variable(s), a **conflict** occurs and the transaction will **abort**. An aborted transaction is retried,

⁶Clojure’s `get` retrieves a key from a mapping. `assoc` updates a key, but because Clojure uses immutable data structures, it does not update the flight map in place, but returns a new map `new-flight`. The flight must be updated explicitly to this value using `ref-set`.

which means that its changes are discarded (rolled back) and its contents are reexecuted. A transaction thus executes one or several **attempts**. When no conflicts occur, a transaction **commits** successfully and its changes become visible to other transactions.

2.2.1 Guarantees: Isolation and Progress. Transactional systems provide two useful guarantees.

First, transactions run in isolation. Different isolation “levels” have been defined, such as serializability [35], opacity [28], and snapshot isolation [9]. Like Clojure, we provide snapshot isolation. **Snapshot isolation** [SI] requires that (1) a transaction sees a consistent view of the memory (this is its snapshot), and (2) a transaction can commit only if none of its updates conflict with any concurrent updates made since the snapshot. In the example, this entails that (1) the values of the flight variable on lines 4 and 5 must be equal, even if another thread modified them concurrently, and (2) two transactions cannot reserve seats on the same flight on line 6. Thanks to isolation, developers can reason about the program at the level of transactions: While transactions execute in parallel, it does not matter in which order their constituent instructions are interleaved, only in which order the transactions as a whole are committed.

Second, transactional systems provide a progress guarantee. While traditional locking systems are prone to issues such as deadlocks, livelocks, and starvation, transactional systems aim to free the programmer from worrying about these issues. Similar to isolation levels, different STMs implement a range of different progress guarantees. Many systems guarantee **deadlock freedom** [DLF] [33]: When two transactions conflict, progress is guaranteed by a mechanism that decides which transaction(s) to delay so another can make progress.

2.2.2 Implementation. Like Clojure, we implement STM using multiversion concurrency control (MVCC) [10, 30, 36]. Each transactional variable contains a (limited) history of its values. During a transaction, reading a transactional variable returns the value it had when the transaction started. Writing to a transactional variable stores its new value locally in the transaction. At the end of the transaction, it attempts to commit, and if successful the new values atomically become visible for new transactions. The commit is unsuccessful when two transactions wrote to the same variable: We say a conflict occurred on that variable. Then, the most recent transaction is aborted, i.e., it discards its changes and retries. We refer to the related literature for more details [30, 36].

2.2.3 Formalization of Operational Semantics. Figure 4 defines L_t , a language with transactions. L_t extends L_f : As STM does not provide any constructs to *create* parallel tasks, it must rely on futures to create the tasks in which transactions run.

The formal semantics describes an algorithm that is simpler than MVCC. It guarantees snapshot isolation in a trivial way, by making a complete copy of the transactional memory when a transaction is started. The actual implementation uses MVCC and provides the same guarantee, but avoids creating copies, by versioning of transactional variables and taking locks.

Syntax. The syntax adds the elements described above. Additionally, we add the construct $\text{atomic}\star e$: this runtime expression cannot be used by the programmer in the source program, but we will use it in the reduction rules to represent a transaction that is being executed.

State. The program state, previously a set of tasks T , is extended with a map of transactions τ and the transactional heap σ . Each transaction in the map τ is identified using a transaction number n , and consists of:

- \circ : its *status*, initially \triangleright (running) and resolving to either \checkmark (committed) or \times (aborted).
- $\hat{\sigma}$: its *snapshot* of the heap, copied when the transaction started and never modified. This provides a consistent view of the transactional memory during the transaction.

Syntax

$r \in \text{TVar}$
 $v \in \text{Value} ::= \dots \mid r$
 $e \in \text{Expression} ::= \dots \mid \text{atomic } e \mid \text{atomic}\star e$
 $\mid \text{ref } e \mid \text{deref } e \mid \text{ref-set } e e$

Evaluation contexts

$\mathcal{P} ::= \text{T} \cup \langle f, \mathcal{E}, n^? \rangle, \tau, \sigma$
 $\text{with } n^? ::= n \mid \bullet$
 $\mathcal{E} ::= \dots \mid \text{atomic}\star \mathcal{E} \mid \text{ref } \mathcal{E} \mid \text{deref } \mathcal{E}$
 $\mid \text{ref-set } \mathcal{E} \mid \text{ref-set } r \mathcal{E}$

State

Program state	$\mathfrak{p} ::= \text{T}, \tau, \sigma$	Task	$\text{task} \in \text{Task} ::= \langle f, e, n^? \rangle$
Tasks	$\text{T} \subset \text{Task}$	Tx number	$n \in \text{TxNr} = \mathbb{N}^+$
Transactions	$\tau : \text{TxNr} \rightarrow \text{Tx}$	Transaction	$\text{tx} \in \text{Tx} ::= \langle \circ, \tilde{\sigma}, \tilde{e}, \delta \rangle$
Heap, snapshot, store	$\sigma, \tilde{\sigma}, \delta : \text{TVar} \rightarrow \text{Value}$	Tx status	$\circ ::= \triangleright \mid \surd \mid \times$
Initial state	$\mathfrak{p}_0 = \{ \langle f_0, e, \bullet \rangle \}, \emptyset, \emptyset$		

Reduction rules

$\text{congruence}_{\downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[e], n^? \rangle, \tau, \sigma \rightarrow_{\text{t}} \text{T}' \cup \langle f, \mathcal{E}[e'], n^? \rangle, \tau, \sigma$
 $\text{if } \text{T} \cup \langle f, \mathcal{E}[e] \rangle \rightarrow_{\text{f}} \text{T}' \cup \langle f, \mathcal{E}[e'] \rangle$

$\text{atomic}_{\downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[\text{atomic } e], \bullet \rangle, \tau, \sigma$
 $\rightarrow_{\text{t}} \text{T} \cup \langle f, \mathcal{E}[\text{atomic}\star e], n \rangle, \tau[n \mapsto \langle \triangleright, \sigma, e, \emptyset \rangle], \sigma \quad \text{with } n \text{ fresh}$

$\text{atomic}_{\times \downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[\text{atomic } e], n \rangle, \tau, \sigma \rightarrow_{\text{t}} \text{T} \cup \langle f, \mathcal{E}[e], n \rangle, \tau, \sigma$

$\text{ref}_{\downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[\text{ref } v], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma$
 $\rightarrow_{\text{t}} \text{T} \cup \langle f, \mathcal{E}[r], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta[r \mapsto v] \rangle], \sigma \quad \text{with } r \text{ fresh}$

$\text{deref}_{\downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[\text{deref } r], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma$
 $\rightarrow_{\text{t}} \text{T} \cup \langle f, \mathcal{E}[(\tilde{\sigma} :: \delta)(r)], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma$

$\text{ref-set}_{\downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[\text{ref-set } r v], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma$
 $\rightarrow_{\text{t}} \text{T} \cup \langle f, \mathcal{E}[v], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta[r \mapsto v] \rangle], \sigma$

$\text{commit}_{\surd \downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[\text{atomic}\star v], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma$
 $\rightarrow_{\text{t}} \text{T} \cup \langle f, \mathcal{E}[v], \bullet \rangle, \tau[n \mapsto \langle \surd, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma :: \delta \quad \text{if } \forall r \in \text{dom}(\delta) : \sigma(r) = \tilde{\sigma}(r)$

$\text{commit}_{\times \downarrow \text{t}} \quad \text{T} \cup \langle f, \mathcal{E}[\text{atomic}\star v], n \rangle, \tau[n \mapsto \langle \triangleright, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma$
 $\rightarrow_{\text{t}} \text{T} \cup \langle f, \mathcal{E}[\text{atomic } \tilde{e}], \bullet \rangle, \tau[n \mapsto \langle \times, \tilde{\sigma}, \tilde{e}, \delta \rangle], \sigma \quad \text{if } \exists r \in \text{dom}(\delta) : \sigma(r) \neq \tilde{\sigma}(r)$

Fig. 4. Operational semantics of L_{t} , a language with transactions.

- \tilde{e} : its *original expression*, which is copied when the transaction starts. If the transaction aborts, this is restored.
- δ : its *local store*, consisting of the new values of modified transactional variables. The domain of δ is referred to as the transaction's write set.

Tasks are extended with an optional number n that identifies the active transaction. If no transaction is active, this is \bullet . (Throughout our formalization, we use the question mark as meta-syntax to indicate an optional element, i.e., $n^? ::= n \mid \bullet$.) In practice this is implemented as a thread-local variable. Throughout its lifetime, a task can execute different transactions, but each transaction belongs to only one task.

Next, we define the reduction relation \rightarrow_{t} for L_{t} .

Start of transaction. The rule $\text{atomic}_{\downarrow \text{t}}$ starts a new transaction. A (globally unique) new transaction number is stored in the current task, and the new transaction is added to τ . $\text{atomic } e$ is replaced with $\text{atomic}\star e$, so in the following transitions e is reduced. This will always eventually reduce to a single value, $\text{atomic}\star v$, at which point the transaction is *ready to commit* and one of

the commit rules below can be applied. The rule $\text{atomic}_{\times|t}$ corresponds to the “closed nesting” of transactions [51], discussed in Section 4.2.

Transactional operations. ref , deref , and ref-set , respectively, create, read from, and write to transactional variables, using the snapshot and local store. Transactional variables created using ref remain local to the transaction until it is committed. In deref , a consistent view of the transactional memory is ensured, as the snapshot is a copy of the heap made when the transaction started. (The operator $::$ concatenates two maps and is right-preferential; it is defined in Appendix B.5.)

Commit. A transaction commits successfully if none of the transactional variables in its write set have been modified by another transaction since the start of this transaction.

If this validation succeeds, the transaction can commit ($\text{commit}_{\surd|t}$): Its changes are written to the transactional memory by appending its local store to the heap. This occurs in a single step, ensuring that the changes atomically become visible to the other tasks.

If the validation fails, the transaction rolls back and retries ($\text{commit}_{\times|t}$). The transaction is marked as failed and the task is restored to its state before the transaction started, using \tilde{c} . In the next transition, the rule $\text{atomic}_{|t}$ is applicable and restarts the transaction. Note that different attempts of the same transaction have different numbers, so in fact n identifies the attempt and not the transaction.⁷

2.2.4 Formalization of Snapshot Isolation. A program’s reduction can be distilled into a transactional history. Like Berenson et al. [9], we define snapshot isolation as the absence of five anomalies: patterns that may not appear in the transactional history.

Definition 2.3 (Transactional History). A program’s **transactional history** is the sequence of the transactional operations in the program’s execution. The following transactional operations can occur, each corresponding to a specific reduction rule:

- $x \rightarrow_1 v$: transaction 1 reads transactional variable x , retrieving the value v (rule $\text{deref}_{|t}$).
- $x \leftarrow_1 v$: transaction 1 writes the value v to transactional variable x (rule $\text{ref-set}_{|t}$).
- \surd_1 : transaction 1 commits successfully (rule $\text{commit}_{\surd|t}$).
- \times_1 : transaction 1 aborts (rule $\text{commit}_{\times|t}$).

A program reduction can be converted into a transactional history by listing the transactional operations that correspond to these reduction rules. Note that aborted transactions are part of the program’s history, too: Even in a transaction that will eventually abort, we expect to get a consistent view of the memory (not doing so may for instance trigger unexpected errors [28]). We also note that a transaction attempt cannot both commit successfully and abort (i.e., only one of \surd_n or \times_n appears for any n), and that this is the last operation for the attempt (i.e., all \rightarrow_n and \leftarrow_n must precede \surd_n or \times_n).

THEOREM 2.4 (SNAPSHOT ISOLATION). L_t provides snapshot isolation, i.e., the transactional history of any program reduction does not contain dirty reads, dirty writes, non-repeatable reads, lost updates, or read skew.

Each anomaly is defined using a transactional history pattern in Appendix D.

2.2.5 Formal Discussion of Deadlock Freedom. To discuss the issue of deadlocks between transactions, we need to differentiate between the algorithm implemented by the formal semantics and the one used in the practical implementation. In the operational semantics, transactional variables

⁷Although not necessary here, we keep failed transaction attempts in the program state; these will prove useful later.

are not locked, instead the commit rules simply check and perform all writes to the transactional memory *in a single transition*. Deadlocks are therefore impossible. In the practical implementation, we implement STM using the Multiversion Concurrency Control algorithm, which guarantees deadlock freedom [36]. Deadlock freedom of transactions ensures that, when two transactions conflict, one is delayed so another can make progress [33]. This guarantee is a property of the implementation algorithm; it is not visible in the semantics.

2.3 Actors

In the actor model, actors run concurrently and communicate using asynchronous messages [1, 37]. In Chocola, we use a “classic actors” model [20], in which an **actor** consists of three elements. First, each actor has a unique and immutable **address**, used to send it messages. Second, its **inbox** is a queue of messages. In our model, a message is simply a tuple of values. Third, a **behavior** specifies how an actor responds to a message. In Chocola, a behavior is defined as follows:⁸

```

1 (def travel-agent-behavior
2   (behavior [flights hotels] ; Internal memory of actor contains flights and hotels
3     [:book orig dest n] ; When a message that matches this pattern comes in, execute the code below:
4     (let [flight (search-flight flights orig dest) ; Search for a flight
5           hotel (search-hotel hotels dest) ; Search for a hotel
6               flights' (reserve-seats flights flight n) ; Reserve n seats on the flight
7               hotels' (reserve-room hotels hotel n)] ; Reserve a room with n beds in the hotel
8       (become travel-agent-behavior flights' hotels')))) ; Update current actor's behavior

```

This behavior specifies an actor that represents a travel agent. Other actors can send a message to this actor to book a holiday, consisting of a flight and a hotel room. The behavior of an actor defines how it responds to an incoming message. It first contains a list of parameters that define the **internal memory** of the actor: here, maps containing the `flights` and `hotels`. Second, it contains a list of patterns for incoming messages and the corresponding response. Here, there is only one pattern, matching a message that starts with the symbol `:book` followed by three values, which are bound to the variables `orig`, `dest`, and `n`. Actors are spawned using `spawn`:

```

9 (def flights {"BA212" {:from "BOS" :to "LHR" :price 499 :seats 243}
10              "BA213" {:from "LHR" :to "BOS" :price 499 :seats 243}})
11 (def hotels {"Hilton" {:in "BOS" :price 100 :rooms 300}})
12 (def agent (spawn travel-agent-behavior flights hotels))

```

This creates a new actor with `travel-agent-behavior` as initial behavior and the maps of `flights` and `hotels` as internal memory. `spawn` returns the address of the new actor.

(`send agent :book "LHR" "BOS" 3`) puts a message in this actor’s inbox, containing the values `:book`, `"LHR"`, `"BOS"`, and `3`. When the receiving actor processes the message, the pattern on line 3 is matched, and hence the code on lines 4 to 8 is executed with `flights` and `hotels` bound to the values given when the actor was spawned and `orig`, `dest`, and `n` bound to the message’s values.

An actor can change its behavior and internal memory using `become`. On line 8 in the example, `become` updates the agent actor, keeping its behavior identical but updating its internal memory to new maps of `flights` and `hotels` in which the requested reservations were made.

⁸Because Clojure’s data structures are immutable, `reserve-seats` and `reserve-room` do not modify `flights` and `hotels`, but instead return updated data structures.

Syntax

$a \in \text{Address}$
 $b \in \text{BehaviorDef} ::= \text{behavior } [\bar{x}_{\text{beh}}] [\bar{x}_{\text{msg}}] e$
 $v \in \text{Value} ::= \dots \mid a \mid b$
 $e \in \text{Expression} ::= \dots \mid \text{spawn } e \bar{e} \mid \text{become } e \bar{e} \mid \text{send } e \bar{e}$

Evaluation contexts

$\mathcal{P} ::= A \cup \langle a, \mathcal{E}, \text{beh} \rangle, \mu$
 $\mathcal{E} ::= \dots \mid \text{spawn } \mathcal{E} \bar{e} \mid \text{spawn } b \bar{v} \mathcal{E} \bar{e}$
 $\quad \mid \text{become } \mathcal{E} \bar{e} \mid \text{become } b \bar{v} \mathcal{E} \bar{e}$
 $\quad \mid \text{send } \mathcal{E} \bar{e} \mid \text{send } a \bar{v} \mathcal{E} \bar{e}$

State

Program state	$p ::= A, \mu$	Inboxes	$\mu : \text{Address} \rightarrow \overline{\text{Message}}$
Actors	$A \subset \text{Actor}$	Actor	$\text{act} \in \text{Actor} ::= \langle a, e^?, \text{beh} \rangle$
Initial state	$p_0 = \{ \langle a_0, e, \text{beh}_0 \rangle \}, \emptyset$	Behavior	$\text{beh} \in \text{Behavior} ::= \langle b, \bar{v} \rangle$
	$\text{beh}_0 = \langle \text{behavior } [] [] \text{nil}, [] \rangle$	Message	$\text{msg} \in \text{Message} ::= \langle \bar{v} \rangle$

Reduction rules

$\text{congruence}|_a \quad A \cup \langle a, \mathcal{E}[e], \text{beh} \rangle, \mu \rightarrow_a A \cup \langle a, \mathcal{E}[e'], \text{beh} \rangle, \mu \quad \text{if } e \rightarrow_b e'$
 $\text{spawn}|_a \quad A \cup \langle a, \mathcal{E}[\text{spawn } b_* \bar{v}], \text{beh} \rangle, \mu \rightarrow_a A \cup \langle a, \mathcal{E}[a_*], \text{beh} \rangle \cup \langle a_*, \bullet, \langle b_*, \bar{v} \rangle \rangle, \mu[a_* \mapsto []]$
 $\quad \text{with } a_* \text{ fresh}$
 $\text{send}|_a \quad A \cup \langle a, \mathcal{E}[\text{send } a_{\text{to}} \bar{v}], \text{beh} \rangle, \mu[a_{\text{to}} \mapsto \overline{\text{msg}}] \rightarrow_a A \cup \langle a, \mathcal{E}[\text{nil}], \text{beh} \rangle, \mu[a_{\text{to}} \mapsto \overline{\text{msg}} \cdot \langle \bar{v} \rangle]$
 $\text{receive}|_a \quad A \cup \langle a, \bullet, \text{beh} \rangle, \mu[a \mapsto \langle \bar{v}_{\text{msg}} \rangle \cdot \overline{\text{msg}}] \rightarrow_a A \cup \langle a, \text{bind}(\text{beh}, \bar{v}_{\text{msg}}), \text{beh} \rangle, \mu[a \mapsto \overline{\text{msg}}]$
 $\quad \text{with } \text{bind}(\langle \text{behavior } [\bar{x}_{\text{beh}}] [\bar{x}_{\text{msg}}] e, \bar{v}_{\text{beh}}, \bar{v}_{\text{msg}} \rangle) = \text{let } [x_{\text{beh}} v_{\text{beh}}] (\text{let } [x_{\text{msg}} v_{\text{msg}}] e)$
 $\text{turn-end}|_a \quad A \cup \langle a, v, \text{beh} \rangle, \mu \rightarrow_a A \cup \langle a, \bullet, \text{beh} \rangle, \mu$
 $\text{become}|_a \quad A \cup \langle a, \mathcal{E}[\text{become } b_* \bar{v}], \text{beh} \rangle, \mu \rightarrow_a A \cup \langle a, \mathcal{E}[\text{nil}], \langle b_*, \bar{v} \rangle \rangle, \mu$

Fig. 5. Operational semantics of L_a , a language with actors.

An actor alternates between two states: ready to accept a message or busy processing a message. A **turn** is the processing of a single message by an actor, that is, the process of an actor taking a message from its inbox and processing that message to completion [20].

2.3.1 Guarantee: Isolated Turn Principle. The actor model guarantees the **isolated turn principle**, which states that once a turn has started, it will always run to completion, in isolation [18, 20]. Hence, developers do not need to care how individual instructions within a turn are interleaved with those from other actors; instead, they can reason about their program at the level of turns.

De Koster [18] defines the isolated turn principle as a combination of three guarantees:

- Continuous message processing** Cont Deadlocks cannot occur while processing a single message, i.e., turns are free from deadlocks.
- Consecutive message processing** Cons An actor processes messages only from its own inbox and processes them one by one. Hence, within one actor different turns cannot be interleaved.
- Isolation** Iso Memory is isolated: An actor can access only its own memory. Actor systems usually achieve this by disallowing shared mutable state between actors. Hence, turns are free from low-level data races.

2.3.2 Formalization of Operational Semantics. Figure 5 defines the actor language L_a and corresponding reduction relation \rightarrow_a . This language extends the base language and is based on the work of Agha et al. [2].

Syntax. L_a introduces two new values. First, an address is a unique reference to an actor. Second, a behavior definition specifies how an actor processes its messages. It first lists a number of

variables \bar{x}_{beh} that are internal to the actor. In this formal semantics, we elide the pattern matching functionality of our language; instead, we suppose that each behavior definition contains only one pattern that consists of a list of parameters \bar{x}_{msg} and a corresponding expression. The expression is thus parameterized by \bar{x}_{beh} (internal memory of actor, set using `spawn` and `become`) and \bar{x}_{msg} (message parameters, bound when a message is received).

State. The state of a program is represented as the actors A and their inboxes μ . There is one inbox per actor. An inbox is a list of messages, where each message simply contains its arguments. An actor consists of three elements: its unique address, the expression it is currently reducing (or \bullet between turns, when the actor is idle), and its current behavior.

Note the distinction between a behavior and a behavior definition. A behavior definition b is a syntactical element that specifies the code that an actor executes. In contrast, a behavior beh is a runtime element that consists of such a behavior definition and the values for \bar{x}_{beh} . We call these values the **internal memory** of the actor.

Reduction rules. \rightarrow_a defines the reduction relation for L_a . Using the rule `congruence|a`, the base language can be used in any actor. The rule `spawn|a` adds a new idle actor to A , with a unique address and with its behavior initialized as given. The rule `send|a` simply appends a message to the end of the receiver’s inbox.

The rule `receive|a` specifies that when an actor is idle and there is a message in its inbox, it can start a turn. The expression encapsulated in its current behavior definition (e) will be evaluated, with the first list of parameters \bar{x}_{beh} bound to the actor’s internal memory and the second list of parameters \bar{x}_{msg} bound to the message’s arguments (as specified in the auxiliary function `bind`).

After the rule `receive|a` has evaluated, the expression in the actor will be further reduced. Eventually, this will result in a single value, at which point the rule `turn-end|a` triggers. This rule resets the actor to its idle state. If there are more messages in the actor’s inbox, another turn can start. In this semantics, messages are always processed in the order they were received.

Last, the rule `become|a` updates the behavior and internal memory of the current actor. As in the actor model of Agha et al. [2], the state of an actor can be changed only using `become`. Note that these changes will take effect only in the next turn.

We note that in these rules, actors are never removed. This entails the receiver of `send` will always exist. Once no more references exist to an actor and it has processed all its messages, it will stay idle forever. It can be garbage collected safely, but this is not implemented in our reduction rules.

2.3.3 Formalization of Guarantees. We formalize the isolated turn principle as a combination of its three constituent guarantees.

Definition 2.5 (Deadlock). A set of actors is **deadlocked** if each actor in the set is waiting for an event that only another actor in the set can cause [59].

LEMMA 2.6 (CONTINUOUS MESSAGE PROCESSING). *While an actor processes a turn, it cannot be deadlocked.*

LEMMA 2.7 (CONSECUTIVE MESSAGE PROCESSING). *For each actor, at each step in the reduction at most one turn is active.*

LEMMA 2.8 (ISOLATION). *Memory is isolated: Each variable in memory can be read and written only by one actor.*

THEOREM 2.9 (ISOLATED TURN PRINCIPLE). L_a *provides the isolated turn principle, i.e., it guarantees continuous message processing, consecutive message processing, and isolation.*

3 MOTIVATIONS FOR COMBINING CONCURRENCY MODELS

We motivate why it is desirable to combine concurrency models based on three observations.

Observation 1: Existing applications combine multiple concurrency models. Tasharofi et al. [60] studied 15 large, mature, and actively maintained Scala projects that use the actor model. We summarize three observations from this study (illustrated in Figure 1). First, 8 of the 15 applications (53%) combine actors and futures. Here, a future is used to represent the “return value” of an asynchronous message sent to an actor. This pattern is a common combination of actors and futures, supported by Scala [52, Chapter 4] but dating as far back as ABCL [66]. Second, 10 of the 15 applications (67%) combine actors with Scala’s `Runnable`, a form of threads. Contact with the developers of these programs established several reasons for doing so: finding locks more suitable than asynchronous messages for their use case, the need to reduce overhead, developers’ inexperience with actors, legacy code, or personal preference. Third, we note that 6 out of the 15 applications (40%) use actors, futures, and threads, thus combining three concurrency models.

These results are corroborated by a survey on the use of concurrency among Microsoft employees [27]. In this survey, around 45% of respondents indicated that they combine shared-memory and message-passing concurrency in their product. These studies thus confirm that developers choose to use several concurrency models throughout a single program.

Observation 2: Programming languages support multiple concurrency models and allow them to be combined. Many programming languages and frameworks already support more than one model. A selection is shown in the table in Appendix A. Clojure is the best example: It has constructs for six concurrency models, and, as it is built on top of the JVM, provides access to four more models. Scala similarly supports eight models: four through its own constructs and four through the JVM. Other examples are Java (seven models), Haskell (five), and C++ (five). The designers of these languages evidently consider it necessary to support a smorgasbord of concurrency models.

Moreover, we see that in cases where a language does not have built-in support for a model, often libraries have been developed. In that case, programmers decide they need to create libraries to extend the language they use with support for additional models.

In almost all of these examples, the languages impose no restrictions on combinations of concurrency models and developers can freely mix multiple models in a single program. (Swalens et al. [58] perform a case study of Clojure.) However, as we will see in the next section, these naive combinations can break the guarantees of the separate models, potentially introducing bugs even in code considered “safe.”

Observation 3: Complex applications consist of different parts that suit different concurrency models. Even if many developers combine multiple models and programming languages support this, one might still wonder whether this is a good idea. We argue it is. Originally, concurrency models were devised to each address a specific concurrency issue that occurs in a specific scenario. However, a typical application consists of many different parts, which may each benefit from concurrency. As each model is aimed at specific types of problems, different parts may need different models, and thus it is desirable to combine several models.

4 GOAL: A COMBINATION OF CONCURRENCY MODELS THAT MAINTAINS THEIR GUARANTEES

In this article, we present *Chocola*, a language that combines futures, actors, and transactions into a unified model. This section outlines the requirement for this language and presents a running example. The subsequent Sections 5 to 7 then delve into each pairwise combination in more detail; these combinations are afterwards consolidated in Section 8.

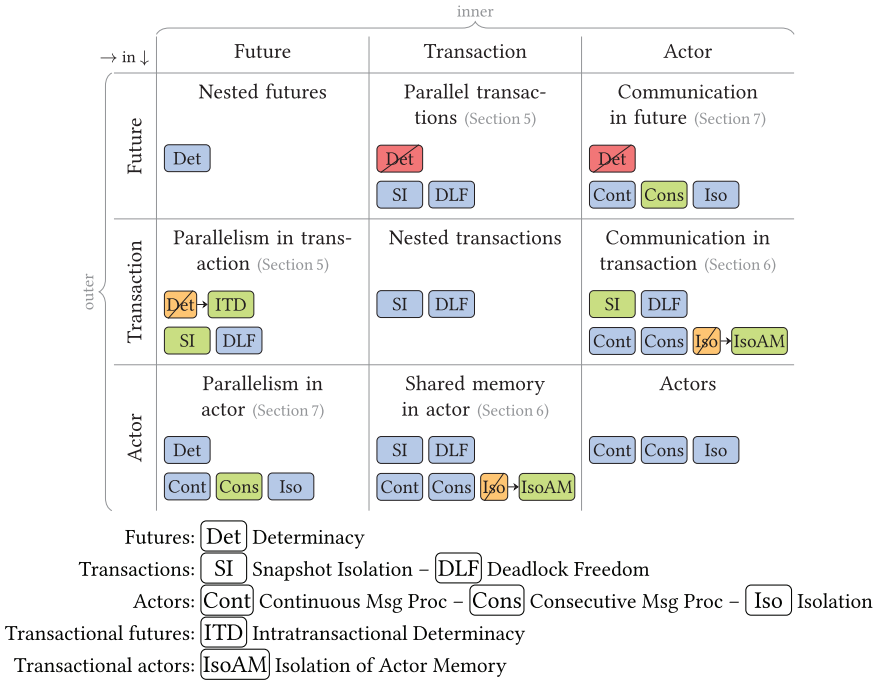


Fig. 6. Pairwise combinations of the three concurrency models of Chocola and their guarantees. The colors are explained in Section 4.

Goal. Our goal is to find a suitable semantics for the unified model of futures, actors, and transactions, even when concurrency models are combined. We define two requirements:

- (1) First, **the semantics of the separate models should remain unchanged**, so programs that do not use combinations work unchanged, ensuring *backwards compatibility*.
- (2) Second, **the guarantees of all models should be maintained even when they are combined, whenever possible**. Hence, developers can use concurrency models under the same assumptions in all contexts. In some cases, we will see that it is impossible to combine the guarantees of all models, because they inherently conflict. For instance, when a non-deterministic model is used in a deterministic one, it is impossible to maintain determinism. In these cases, we will need to relinquish one of the original guarantees and define a modified, less restrictive guarantee that can be provided.

Approach. We first study the pairwise combinations of these three models (in Sections 5 to 7) and afterwards combine them into a unified model (Section 8). To study the problems, for each pairwise combination, we consider a *naive combination* of the two models, in which we simply embed the models as described in Section 2 in each other. We check whether the naive combination breaks any of the guarantees of its constituent models. If so, we define a modified semantics that satisfies the requirements outlined above.

We consider the *dynamic extent* of each construct: If one model is used in another at execution time, then we say they are nested. This does *not* necessarily require their constructs to be nested *lexically*. For instance, if a library function that uses futures is called in a transaction, then the construct `fork` will not appear in the atomic block in the code (lexically), but at execution time

a future will be created while a transaction is running (dynamically). In the rest of the article, whenever we say that two constructs are nested, we refer to dynamic nesting.

Result. Figure 6 tabulates the 9 (3×3) pairwise combinations: Each cell describes how one model is nested in another. The table summarizes the results presented in the following sections. The colors indicate which guarantees are valid in a naive combination and in Chocola:

- Guarantees in blue are upheld even in a naive combination. No changes to the semantics are needed to satisfy our requirements.
- Guarantees in green are broken in a naive combination, so in Chocola, we modify the semantics to reestablish the guarantee.
- Guarantees in red are inevitably broken, in a naive combination as well as in Chocola. This occurs in two cases and is the result of nesting a non-deterministic in a deterministic model.
- Det → ITD indicates that a guarantee (here, determinacy) is broken in a naive combination and cannot be maintained by Chocola. Instead, we defined a (slightly) less restrictive guarantee that can be upheld (here, intratransactional determinacy).

Note that there is a sort of “anti-symmetry” in the table. The diagonal contains models nested in themselves. These “trivial” combinations all maintain the guarantees and are briefly discussed in Section 4.2. All other cells have an opposite across the diagonal, e.g., the top-right cell represents actors in futures, while the bottom-left cell represents futures in actors. In these cells, when *different* models are combined naively, the guarantees are broken. These pairwise combinations are discussed in Sections 5, 6, and 7. Afterwards, in Section 8, all three models are unified in Chocola.

4.1 Example: Holiday Reservation System Using Three Concurrency Models

We now present a running example that combines all three models: a holiday reservation system that books flights, hotels, and cars. It is based on the Vacation benchmark from the STAMP benchmark suite [48], which we also use for our evaluation in Section 11. The code is shown in Figure 7.

This program contains several shared data structures, encapsulated in transactional variables (lines 1–4): flights, hotels, cars, and customers. Each customer record contains the trip’s origin and destination, and the number of travellers.

The program also contains several types of actors. First, a set of *travel agent actors*, using *travel-agent-behavior*, receive messages from each customer to process their reservations (lines 35–37). A reservation consists of a transaction in which two flights, a hotel room, and a car are reserved, and a confirmation code is generated (lines 28–33). For each of the four items, a message is sent to a set of *reservation actors*, e.g., *airlines* with the behavior *airline-behavior*. (The hotels and car-rentals actors look similar.)

When an airline actor receives a message to reserve a flight (lines 18–21), it first filters the flights using the customer’s criteria and then reserves that flight. This is protected using a transaction to ensure that an item cannot be reserved multiple times. Searching for a flight (lines 6–9) is parallelized using futures (using `parallel-filter` from Section 2.1).

Thus, this application combines the three models:

- It uses *actors* to concurrently process requests from different customers and for different items. A message-passing model naturally matches this use case: Customers that want to initiate a reservation send a message to an actor in an event-driven way.
- Two sections of the code access shared memory and must therefore be protected using *transactions*. First, when processing a customer, we must ensure either all items are reserved or

```

1 (def flights {"BA212" (ref {:from "BOS" :to "LHR" :price 499 :seats 243}) ...})
2 (def hotels {"Hilton" (ref {:in "BOS" :price 100 :rooms 300}) ...})
3 (def cars ...)
4 (def customers [(ref {:id 0 :orig "LHR" :dest "BOS" :n 3}) ...])
5
6 (defn search-flight [flights orig dest] ; Search flight from orig to dest in flights.
7   (first (parallel-filter ; parallel-filter as defined in Section 2.1
8     (fn [flight] (and (= (get (deref flight) :from) orig) (= (get (deref flight) :to) dest)))
9     (vals flights)))) ; vals returns all values of a map
10
11 (defn reserve-seats [flight n] ; Reserve n seats on flight.
12   (let [new-seats (- (get (deref flight) :seats) n)
13         new-flight (assoc (deref flight) :seats new-seats)]
14     (ref-set flight new-flight)))
15
16 (def airline-behavior
17   (behavior []
18     [:flight orig dest n]
19     (atomic
20       (let [flight (search-flight flights orig dest)]
21         (reserve-seats cheapest n))))))
22
23 (def airlines [(spawn airline-behavior) (spawn airline-behavior) ...])
24
25 (def travel-agent-behavior
26   (behavior [id]
27     [:process c] ; Find and reserve flights, hotel, car (concurrently), and generate confirmation code.
28     (atomic
29       (send (rand-nth airlines) :flight (get (deref c) orig) ...)
30       (send (rand-nth airlines) :flight (get (deref c) dest) ...)
31       (send (rand-nth hotels) :hotel (get (deref c) dest) ...)
32       (send (rand-nth car-rentals) :car (get (deref c) dest) ...)
33       (ref-set c (assoc (deref c) :confirmation-code (generate-confirmation))))))
34
35 (def travel-agents [(spawn travel-agent-behavior 0) ...])
36 (for [c customers]
37   (send (rand-nth travel-agents) :process c)) ; rand-nth selects a random element from a list

```

Fig. 7. Code snippet of a holiday reservation program that combines futures, transactions, and actors.

none (lines 28–33). Second, we must prevent items from being reserved multiple times (lines 19–21). Transactions ensure safe access to shared memory.

- *Futures* are used to exploit parallelism in deterministic operations, here filtering a list. Futures guarantee that the output of the filter operation is determinate and thus does not depend on the order in which threads are scheduled.

With a naive combination of the three concurrency models, this program will not work as expected, as the guarantees of the separate models no longer hold when they are combined. In Sections 5, 6, and 7, we examine these problems using this example, and we define the semantics of Chocola to maintain the desired guarantees.

4.2 Trivial Combinations

We briefly discuss the combinations on the diagonal of Figure 6, in which each model is nested in itself. These have been studied in existing literature and maintain the model’s guarantees.

Nested futures. Forking one parallel task in another is common in programs that use futures. Nesting futures does not break the determinacy of the program: No matter where futures are introduced, the program remains equivalent to the same program without futures [25].

Nested actors. Dynamically “nesting” actors simply corresponds to spawning one in another. This is a standard part of the actor model and maintains the guarantees of actors.

Nested transactions. When a transaction is started while another transaction is already active, this is a *nested transaction*. The nesting of transactions is a well-studied problem [33]. Moss and Hosking [51] distinguish two types of nesting. Open nesting enables better performance, but is complex to use and breaks the isolation of the outer transaction. Closed nesting is simpler, and in practice it is the norm: Clojure, Haskell, and ScalaSTM all use it. Choccola therefore does so, too.

5 TRANSACTIONAL FUTURES: PARALLELISM IN TRANSACTIONS

In this section, we study the combination of transactions and futures. We first discuss creating transactions in futures (Section 5.1), which is common in languages with transactions. Next, we focus on the opposite combination, the creation of futures in a transaction, and we show that a naive combination breaks snapshot isolation (Section 5.2). Hence, we introduce *transactional futures*: futures created in a transaction with access to the encompassing transaction’s context (Section 5.3). Transactional futures maintain the snapshot isolation and deadlock freedom of transactions and guarantee intratransactional determinacy. We first described transactional futures in previous work [56].

5.1 Transactions in Futures for Parallel Transactions


We first focus on the use of transactions in parallel tasks, i.e., the construct `atomic` within the dynamic extent of a `fork`. As the transactional model does not provide any construct to create parallelism, this combination is standard in languages with transactions: Any use of transactions requires another model to create the tasks in which they run. The semantics of this combination was already specified accordingly in Section 2.2 where we defined the language L_t with transactions as an extension of the language L_f with futures.


The question remains whether this combination preserves the guarantees of both models. As this combination is standard in transactional systems, it guarantees snapshot isolation **SI** and deadlock freedom **DLF**. However, the **determinacy of futures is broken** **Det**: A program with transactions is equivalent to a serialization of the transactions, but usually there are many possible serializations.

Breaking determinacy is unavoidable: As soon as a non-deterministic model such as transactional memory is introduced, it is no longer possible to guarantee determinacy of the whole program. Amongst others, Bocchino et al. [12], Lee [42], Van Roy and Haridi [61] recommend that developers use determinism wherever possible and carefully introduce non-determinism only where it is inescapable. Following this line of thought, we argue that the loss of determinacy does not pose a problem: When developers decide to use a non-deterministic model, they do so because they need its non-determinism, and therefore they are aware that determinacy is not guaranteed. Moreover, transactions still limit the number of possible outputs developers need to consider to the number of possible serializations of the transactions.

5.2 Futures in Transactions for Intra-transaction Parallelism: Motivation and Problems

To demonstrate the use of futures in a transaction, we revisit our running example of Figure 7. On lines 19–21, it contains a transaction that searches for a flight and reserves seats on that flight. It uses the function `search-flight`, defined on lines 6–9, which searches for a suitable flight by filtering the list of all flights based on their trajectory, returning the first that matches. In this function, we want to improve the performance of filtering flights by using `parallel-filter`, as defined in Section 2.1. This function divides the list of flights into several partitions that are filtered

in parallel tasks. However, with a naive combination of futures and transactions this program does not work as expected! In a language like Clojure, a transaction is thread-local, i.e., it is bound to the task it was created in. The tasks created in `parallel-filter` do not have a transaction running when they execute the function on line 8 of Figure 7, hence, they can see inconsistent values for the flight. As such, the **determinacy of the futures is no longer guaranteed** : Depending on the order in which their instructions are interleaved, they see different values.

The problem is that a naive combination allows futures to be created in a transaction, but they are not part of that transaction’s context. When an atomic block appears in a new task, a separate transaction is created with its own, possibly inconsistent, snapshot of the shared memory. This transaction will commit independently. As such, the **isolation of the transaction is broken** . This problem occurs amongst others in Clojure, ScalaSTM, Deuce STM for Java, and GCC’s support for STM in C and C++.⁹



In Haskell, however, the type system rejects the scenario above: Transactions are encapsulated in the STM monad while forking a task using `forkIO` is possible only in the IO monad. Hence, the isolation of transactions is guaranteed, but the **potential parallelism is limited**: Every time transactions are introduced to isolate some computation from other tasks, the potential performance benefits of parallelism *inside* this computation are forfeited. This is throwing out the baby with the bathwater.

5.3 Solution: Transactional Futures

Chocola solves these issues by defining transactional futures. A **transactional future** is the future associated with a so-called **transactional task**: a task that is forked while a transaction is running. A transactional task operates within the context of its encapsulating transaction.


Conceptually, each transactional task creates a copy of the transactional memory and accesses only that copy. This ensures that tasks can run concurrently without interfering with each other. To this end, a transactional task contains two data structures: a (read-only) **snapshot** containing a conceptual copy of the state of the transactional memory when it was forked and a **local store** containing modifications made in the task.

Each transaction starts with one *root* task that evaluates the transaction’s body. Its snapshot is a copy of the transactional heap; its local store starts empty. When a task is forked, the new task’s snapshot represents the current state of the transactional memory, hence it is the snapshot of its parent task modified with the current local store of the parent. The local store of a newly forked task starts empty. (These are conceptual copies; we avoid actually copying memory in the implementation, described in Section 10.) While a task executes, `deref` looks up values in the snapshot and `ref-set` stores new values in the local store. In the example of the previous section, this means each task created in the `parallel-filter` function has a copy of the flights in its snapshot and reads these, resulting in a consistent view.

Because a task uses only its own snapshot and local store, tasks do not interfere: The order in which their instructions are interleaved does not affect the end result. Hence, futures are determinate within the transaction, a guarantee we call **intratransactional determinacy**  . This is a less restrictive version of the determinacy of the whole program that futures usually provide, a guarantee that is inevitably broken when transactions are introduced.

⁹<https://nbronson.github.io/scala-stm/>, <https://sites.google.com/site/deucestm/>, and <https://gcc.gnu.org/wiki/TransactionalMemory>.

When a task is joined for the first time, its local store is merged into the task performing the join.¹⁰ This way, changes propagate from child tasks to their parent. Subsequent joins of the same task will not repeat this, as their changes are already merged.


At the end of the transaction, all transactional tasks in the transaction should have been joined (directly or indirectly) into the root task. Hence, all changes from all tasks have been incorporated into the local store of the root task. All these changes are then committed in a single step, thus **maintaining the isolation of the transaction** . If a conflict occurs at commit time, the whole transaction is aborted and retried. If a conflict occurs in one of the tasks while the transaction is still running, *all* tasks are aborted and the whole transaction is retried. In other words, the tasks within a transaction are coordinated to either all succeed or all fail: They form one atomic group.

6 TRANSACTIONAL ACTORS: COMMUNICATION BETWEEN TRANSACTIONS

In this section, we study the combination of transactions and actors. First, we describe the motivation for and problems with the use of transactions in actors (Section 6.1) and the use of actors in transactions (Section 6.2). Next, we introduce *transactional actors* as a safe way to share memory between actors (Section 6.3). Transactional actors maintain the snapshot isolation and deadlock freedom of transactions and provide a variant of the isolated turn principle, which we refer to as the consistent turn principle. We first introduced transactional actors in previous work [57].

6.1 Transactions in Actors to Safely Share Memory: Motivation and Problems

In Section 3, we established that introducing shared memory in an actor system can be useful and occurs in practice. We can also observe this in the holiday reservation example. In the example of Section 2.3, each travel agent had its own separate set of flights and hotels. In Figure 7, a more typical reservation system is shown in which multiple travel agents use the same flights and hotels, thus sharing memory between actors.

We discuss how this is currently achieved in two types of actor systems: pure and impure systems [19]. *Pure actor systems*, such as Erlang, enforce strict isolation between actors: Each actor can access only its own memory. In these systems, developers often represent shared state using two patterns: replication or delegation of the shared state [19]. However, both patterns require the developer to ensure consistency of the shared data and to prevent race conditions and deadlocks. Thus, pure actor systems maintain the isolated turn principle, but **representing shared state in them is complex and error prone**. However, *impure actor systems* do not enforce strict isolation, so developers can use the underlying shared-memory model of the language. This is the approach used by the Scala projects in the study of Tasharofi et al. [60]. In these systems, the isolated turn principle is broken as **actors' memory is no longer isolated** , and it is up to the developer to ensure correct access to the shared memory.


As we will see in the rest of this section, this broken guarantee can be reintroduced by carefully combining actors with transactions, as transactions guarantee isolation. In Figure 7, we applied this idea by encapsulating the shared flights, hotels, and cars in transactional variables (lines 1–3), and using transactions to access them in the travel agents (lines 19–21).

6.2 Actors in Transactions to Distribute and Coordinate Work: Motivation and Problems



Not only are transactions useful to protect access to shared state between actors, conversely, actors are also useful to coordinate work between transactions. In our running example, we improve the

¹⁰When two tasks modify the same transactional variable, a conflict occurs. To solve this, we allow the developer to specify a *conflict resolution function*, which takes the conflicting values and returns the new value for the variable.

performance by creating separate actors that search and reserve flights, hotels, and cars in parallel. The travel agent sends messages to these actors to do this concurrently (lines 29–32).


In a naive combination of transactions and actors, this program will not work correctly. The transaction in `travel-agent-behavior` sends four messages, but if the transaction aborts, e.g., because the `ref-set` on line 33 fails, the messages are not rolled back. Messages can thus be sent multiple times and their effects are visible even if the transaction is aborted, **breaking the isolation of the transaction** .

6.3 Solution: Transactional Actors

In Chocola, actors can share memory using transactions, as in the example of Section 6.1, we then call these **transactional actors**. While this breaks the actor model’s guarantee of isolation of all memory, transactional actors can provide a less restrictive guarantee: the **isolation of actor memory**  , which states that (only) the internal memory of actors is isolated. Hence, each actor has its own memory that it can access safely because it is isolated, while memory that is shared between actors is protected using transactions.

Further, to ensure the isolation of transactions, Chocola must make any effects on actors that occur during a transaction part of the transaction. We consider how the four constructs of the actor model can safely be nested in a transaction:

- **behavior**: Defining a behavior in a transaction is no problem, as this operation has no side effects. A behavior can refer to variables in its lexical scope—it is essentially a closure—but will run at a later time and thus does not have access to the encapsulating transaction.
- **spawn**: Spawning an actor is an effect that must be part of the transaction. As it is costly, Chocola delays it until the transaction commits (successfully). This ensures that the transaction’s isolation is maintained and the creation cost is paid only once.
- **become**: `Become` is a construct that is delayed by construction: Its effect takes place only upon the start of a new turn. As a transaction cannot span multiple turns, the transaction will always be committed before the effect of `become` is made visible, maintaining isolation.
- **send**: As illustrated in Section 6.2, (the effects of) a message sent in a transaction must be rolled back if the transaction aborts. In Chocola, messages sent from within a transaction get a **dependency** on the transaction and are said to be **tentative**, while traditional messages sent outside a transaction have no such dependency and are **definitive**. The receiver of a tentative message must take this into account. When an actor takes a tentative message from its inbox, the turn that processes it also becomes tentative: The message is processed, but the effects it causes should not be persisted yet. Even though this turn is not a transaction, it executes in the same “tentative” manner, as its effects can roll back. When a **tentative turn** ends, the actor waits until the transaction on which it depends has committed. After a successful commit of its dependency, the actor can continue to its next turn, and we say the turn was successful. If its dependency aborts, the tentative turn fails and all of its effects are discarded. The actor then processes the next message in its inbox as if nothing happened.

With these changes to the semantics of actors’ constructs, our example now works as expected. The messages sent on lines 29–32 are tentative. When they are processed by the `airline` actor, the turn on lines 18–21 is tentative. The effects of this turn, i.e., the reservation of the seats on the flights, are persisted only if the original transaction succeeds. If the original transaction aborts, its effects as well as the effects of its dependent transactions are discarded. Hence, the **transaction’s isolation is maintained** .

```


1 (def travel-agent-behavior
2   (behavior []
3     [orig dest n]
4     (do (fork (book-flight orig dest n))
5         (fork (book-hotel dest n))))))

```

Fig. 8. The two forked tasks may continue executing after the turn has finished (“escaping” their turn), interleaved with the next turn. If they send messages or use become, unexpected results can occur.

7 ACTORS AND FUTURES: INTRA-ACTOR PARALLELISM


In this section, we study the combination of futures and actors, which poses fewer problems. Combining actors and futures can be useful: As we saw in Section 3, in the study of Tasharofi et al. [60], 80% of the studied projects combine actors with futures and/or threads. On the one hand, futures can be introduced in an actor to process a turn in parallel: this is **intra-actor parallelism** (the bottom-left cell of Figure 6). On the other hand, actors can be used in a program with futures to introduce communication between parallel tasks (the top-right cell of Figure 6). We examine the effect of the combinations on the guarantees of both models.

Determinacy of Futures. When futures are created in an actor, but they do not contain any operations on actors, determinacy remains guaranteed (bottom-left cell of Figure 6). Conversely, when the actor constructs send and become are used in futures, **determinacy is broken**  (top-right cell of Figure 6), as they may execute in any order. Breaking determinacy for this combination is inevitable: When futures are combined with any non-deterministic model, their determinacy will always be broken. We argue that this is no problem, because this occurs only in those places where the programmer explicitly uses send and become, constructs of the non-deterministic model. As programs using *only* actors can also have a non-deterministic result, the developer should expect non-determinism, whether futures are used or not.

Isolated Turn Principle of Actors. A naive combination of actors and futures breaks the isolated turn principle. We consider the three constituent guarantees separately.

Isolation Even when futures are introduced, actor’s memory remains isolated.

Continuous message processing An actor should be free from deadlocks within a turn. Futures introduce the blocking construct join, which waits for a future to resolve, and thus could potentially introduce deadlocks. Fortunately, a deadlock cannot occur by design. The tasks created in a turn form a “fork tree” [11]: Every turn starts with one “root” task, which can fork tasks, which can themselves fork more tasks, and so on, conceptually forming a tree. No cyclical dependencies are possible in this tree, hence, it is impossible to reach a deadlock between tasks that join each other. We formally explain and prove this in Appendix F.1.

Consecutive message processing An actor must process its messages one by one, without interleaving turns. This requirement is broken by naively introducing futures. Figure 8 illustrates the problem: An actor forks two tasks that are never joined, hence, the actor can proceed to the next turn while the child tasks are still running. We say they “escape” the turn they were forked in. The two turns overlap, so **consecutive message processing is broken** . This leads to two unexpected results: (1) if the child task sends a message, it can arrive *after* messages sent in the next turn, and (2) become in an escaped task can still change the actor’s behavior *after* the next turn started with the old behavior.

Fortunately, Chocolate can reintroduce this broken guarantee using a simple requirement: All tasks must be joined before the turn in which they were forked ends. Hence, all child tasks must have finished before the root task finishes and the turn ends. Consequently, turns

cannot be interleaved and **consecutive message processing is guaranteed** Cons. We believe this requirement is not overly restrictive: it applies only when a task is forked in a turn but its result is never used in that turn. Moreover, a similar requirement exists for transactional futures, where we require that all transactional tasks are joined before the transaction ends, thus both techniques provide a consistent model.

8 CHOCOLA: COMPOSABLE CONCURRENCY LANGUAGE

In this section, we informally describe the ontology of Chocola: the various concepts introduced in the previous sections and their relations (Section 8.1). We demonstrate how the three models are combined in an example (Section 8.2).

8.1 Ontology of Chocola: Its Linguistic Concepts and Their Relations

In Chocola, a program starts as a single actor containing a single task that evaluates the code. The program can then spawn actors, fork futures, and create transactions. We summarize the main concepts and constructs.

Actors. A Chocola program consists of *actors* that run concurrently and have an address, a behavior, and an inbox. Actors are created using `spawn`, which is given an initial behavior. A *behavior* is created using the construct `behavior` and contains the code that defines how an actor behaves when it receives a message. The behavior also contains the *internal memory* of the actor: variables private to the actor. An actor can change its behavior (both the code and its internal memory) using `become`.

An actor's *inbox* is a queue of messages. Each *message* is a list of values that was sent to the actor using `send`. An actor consecutively processes each message in its inbox; the handling of one message is called a *turn*. When a message is sent from within a transaction or a tentative turn, it is *tentative* and has a dependency on a transaction; otherwise it is *definitive*. A turn that is the result of a tentative message is a *tentative turn*. In a transaction and in a tentative turn, `spawn` and `become` are delayed: These are *effects on actors* that are gathered to be executed later, when the dependency has successfully committed.

Transactions. A *transaction* is a section of the code encapsulated in an atomic block that can access shared memory. The shared memory is represented using *transactional variables*. These can be created, read, and written using `ref`, `deref`, and `ref-set` in a transaction. (Outside a transaction, these constructs raise an error.)

Each transaction has an associated *transactional context* containing data related to the current transaction: (1) its *snapshot*: a (conceptual) copy of the transactional memory before the transaction started, (2) its *local store*: the modifications made to the transactional memory in the transaction, and (3) the *effects on actors* that occurred during the transaction and will be executed if and when the transaction commits. Note that it is not the transaction itself but its tasks that each contain a transactional context.

Futures. A *task* is a section of the program that runs concurrently with the rest of the program. It can be created using `fork`, which returns a *future*: a placeholder for the result of the task. This value can be retrieved using `join`, which blocks until the task has finished and then returns its result. When a task is forked in a transaction, we say it is a *transactional task* (which has an associated *transactional future*). Each transactional task has a transactional context, which it adopted from its parent when it was forked and which is merged into its parent when it is joined.

Note. It is important to note that Chocola does **not** introduce *new* syntactical constructs, nor does it change the semantics of its constituent models when used separately. The novelty of

Chocola is that it defines the semantics of the constructs of these concurrency models when they are combined with one another.

8.1.1 When to Use Futures or Actors for Parallelism Inside a Transaction. When parallelizing a program with transactions, one might wonder whether to use transactional futures or transactional actors. At first glance, both seem like similar mechanisms, as both parallelize the internals of a transaction. However, they function quite differently and have different use cases:

- A transactional future runs completely *within* the context of its encapsulating transaction. It has its own copy of the heap on which it acts in isolation, but its changes will always be joined back into its parent later. Hence, the changes to transactional memory from different tasks of the same transaction will be committed at the same time.
- In contrast, transactional actors make it possible to “escape” a transaction. When a message is sent in a transaction, it carries a dependency on that transaction. However, the turn that is executed as a result of the message, while dependent on the transaction, does not run within the original transaction’s context. A second transaction can be started in this turn, and this second transaction will depend on the first, but both run in isolation and have their own copy of the heap, and both commit separately.

These differing semantics are a result of the different use cases of the two models. Futures are used to speed up a deterministic calculation. Thus, when they appear in a transaction, they remain part of the transaction while locally enabling parallelism. However, actors represent separate components in the application that occasionally communicate using messages. When a message is sent in a transaction, the sender signals to another actor that it must act, but the receiver runs separated from the sender. Which technique to use therefore depends on the intention: to locally parallelize a calculation, use futures; to communicate with a separate component, use actors.

8.2 Revisiting the Running Example

We revisit the running example originally introduced in Figure 7 of Section 4.1. As explained throughout the previous sections, using a naive combination of the three concurrency models, this example does not work as expected. Chocola defines a semantics for the combinations of the concurrency models that fixes these problems. We summarize the problems in the original example and Chocola’s solution:

- The tasks in `parallel-filter` (line 8) did not have access to the encapsulating transaction (lines 19–21), breaking the transaction’s isolation and the futures’ determinacy (Section 5.2). Transactional futures provide such safe access, maintaining snapshot isolation SI and intratransactional determinacy Det → ITD.
- Sharing the flights, hotels, and cars (lines 1–4) between actors (lines 18–21) broke the actor model’s guarantee of isolation of memory (Section 6.1). Chocola relaxes this guarantee to the isolation of actor memory Iso → IsoAM.
- The messages to reserve items sent in a transaction (lines 29–32) broke the transaction’s isolation (Section 6.2). Transactional actors re-introduce this guarantee SI by making the messages tentative.
- Tasks created in an actor could “escape” the turn (Section 7). Chocola enforces that they are joined before end of turn, thereby maintaining consecutive message processing Cons.
- In a future, when using operations on transactional memory (line 8) or actors, determinacy is broken (Section 5.1 and 7). This is unavoidable whenever a non-deterministic model is used within a deterministic one, but we argue this can be expected by the programmer Def.

Chocola’s guarantees for the different combinations are summarized in Figure 6.

Progr. state	$p ::= A, T, \mu, \tau, \sigma$	Actor	$act \in \text{Actor} ::= \langle a, f_{\text{root}}^?, \text{beh}, n_{\text{dep}}^? \rangle$
Actors	$A \subset \text{Actor}$	Message	$\text{msg} \in \text{Message} ::= \langle \bar{v}, n_{\text{dep}}^? \rangle$
Tasks	$T \subset \text{Task}$	Task	$\text{task} \in \text{Task} ::= \langle f, a, e, F_c, F_j, \text{eff}, \text{ctx}^? \rangle$
Inboxes	$\mu : \text{Address} \rightarrow \overline{\text{Message}}$	Child, joined fut's	$F_c, F_j \subset \text{Future}$
Tx's	$\tau : \text{TxNr} \rightarrow \text{Tx}$	Effects on actors	$\text{eff} ::= \langle \bar{A}, \overline{\text{beh}}^? \rangle$
Tx heap	$\sigma : \text{TVar} \rightarrow \text{Value}$	Transact'l context	$\text{ctx} ::= \langle n, \bar{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle$
Initial state	$p_0 = \langle \{ \langle a_0, f_0, \langle \text{beh}_0, \bullet \rangle \}, \{ \langle f_0, a_0, e, \emptyset, \emptyset, \langle \emptyset, \bullet \rangle, \bullet \rangle \}, \emptyset, \emptyset, \emptyset \rangle$	Transaction	$\text{tx} \in \text{Tx} ::= \langle \circ, \bar{v} \rangle$

Fig. 9. The representation of the program state and its elements in PureChocola.

9 PureChocola: AN OPERATIONAL SEMANTICS

This section presents PureChocola: a formal operational semantics of Chocola. We define its syntax and program state (Section 9.1) and its reduction rules (Section 9.2). Next, we formalize Chocola's guarantees based on the formal semantics (Section 9.3) with proofs included in the appendices. We also created an executable implementation of parts of PureChocola using PLT Redex [24].¹¹ The differences between PureChocola and the actual implementation of Chocola will be listed at the end of Section 10.

9.1 Syntax and Program State

9.1.1 Syntax. Chocola does not introduce any new syntactical constructs; we merely defined their semantics in certain contexts. Thus, the syntax of Chocola simply consists of the three separate syntaxes combined. It is listed in Figure 19 of Appendix B.

9.1.2 State. The program state of PureChocola consists of all actors and tasks that have been created up to this point and three shared data structures, as shown in Figure 9. We describe each.

Actors. Like before, an actor has a unique address a and a behavior beh . Instead of an expression, it now stores $f_{\text{root}}^?$: the future of the root task of the current turn (or \bullet between turns). Each actor can contain many tasks: one root task and all its descendants. In contrast, each task belongs to exactly one actor. Furthermore, when an actor is processing a tentative message, $n_{\text{dep}}^?$ refers to the transaction on which it depends. In that case, the actor is in a tentative turn; in a definitive turn (and between turns) this is \bullet .

Tasks. As before, a task contains a future f and the expression e that it is currently evaluating. Additionally, tasks are extended to contain information about their effects:

- a : the actor in which the task runs. A task always runs within one actor.
- F_c : the (child) futures of the tasks forked in this task, which need to be joined before it finishes. A task can have no, one, or several children. Conversely, a task has either one parent or it is a root task within its turn, but these back references do not need to be stored.
- F_j : the futures joined in this task. This includes both the futures that were directly joined in this task as well as those joined by other tasks that were joined into this one. Hence, these are the tasks whose effects have been incorporated into the current task. This entails F_j can grow only throughout the reduction of a task.
- eff : the task's delayed **effects on actors**. These are gathered and will be performed at the end of the turn, when it is sure they do not need to be rolled back. There are two kinds of effects: spawned actors \bar{A} and the result of become $\overline{\text{beh}}^?$ (optional, only if a become occurred).

¹¹<https://github.com/jsvalens/chocola-redex>.

- $\text{ctx}^?$: the task's **transactional context**. Outside a transaction this is •. It consists of:
 - n : the number of the transaction that this task is part of.
 - $\bar{\sigma}$: the snapshot of this task. For the root task of a transaction, this is a copy of the heap. For tasks forked during the transaction, this is the snapshot taken at the moment of the fork.
 - δ : the local store of changes made to transactional variables in this task only.
 - eff_{tx} : the effects on actors that occurred in this task during the transaction.

Effects on actors can be stored in two locations: in the task (eff) and in its transactional context (eff_{tx}). When such effects occur inside a transaction, they need to be stored in the transactional context so they can be rolled back if the transaction aborts. When these effects occur outside a transaction but in a tentative turn, they need to be stored in the task and may need to be rolled back when the turn ends. Keeping these effects in two places is a result of the fact that there are two kinds of “tentative” sections that may need to roll back: transactions and tentative turns.

Shared data structures. Three data structures can be accessed from multiple tasks. First, the inboxes of the actors in μ . Tentative messages now contain a dependency ($n_{\text{dep}}^?$) that refers to the transaction in which they were sent; for definitive messages this is •. Second, τ maps transactions to their status. When an actor processes a tentative message, it uses τ to verify the status of its dependency. Third, the transactional heap σ , which works as before.

9.1.3 Evaluation Contexts. The program evaluation context \mathcal{P} can again choose an arbitrary task to evaluate next. The definition of \mathcal{E} is simply the combination of the three definitions of the separate models from Section 2 (listed fully in Figure 19 of Appendix B).

$$\mathcal{P} ::= A, T \cup \langle f, a, \mathcal{E}, F_c, F_j, \text{eff}, \text{ctx}^?, \mu, \tau, \sigma \rangle$$

9.1.4 Helper Functions and Operations. In Appendix B, Figure 20 defines the three helper functions to extract elements out of the program state. Figure 21 defines the operations $|$ and $+=$, which will prove useful when tasks are joined and their effects need to be merged.

9.2 Reduction Rules

We can now describe the reduction relation \rightarrow of PureChocola. We start with the base language (Section 9.2.1) and then define the operations on futures (Section 9.2.2), transactions (Section 9.2.3), and actors (Section 9.2.4). These rules are modifications of those from Section 2.

In some rules, the task being reduced accesses only its own state. We call these *local* reductions and will write them using the shorthand \mathcal{T} :

$$\mathcal{T}\langle f, a, e, F_c, F_j, \text{eff}, \text{ctx}^? \rangle = A, T \cup \langle f, a, e, F_c, F_j, \text{eff}, \text{ctx}^?, \mu, \tau, \sigma \rangle$$

This syntax not only simplifies these rules, but also distinguishes local and non-local reductions, which will be useful when proving intratransactional determinacy (in Appendix C).

9.2.1 Base Language. As before, the base language can be used in any context, whether in or out a transaction and whether in a definitive or a tentative turn. \rightarrow_{b} is defined in Appendix B.2.

$$\frac{\text{congruence|}_c}{\mathcal{T}\langle f, a, \mathcal{E}[e], F_c, F_j, \text{eff}, \text{ctx}^? \rangle \rightarrow \mathcal{T}\langle f, a, \mathcal{E}[e'], F_c, F_j, \text{eff}, \text{ctx}^? \rangle \text{ if } e \rightarrow_{\text{b}} e'}$$

9.2.2 Futures. We define how to fork and join a future in Figure 10.

fork. As before, forking a future creates a new task that evaluates the given expression. If the task was forked in a transaction, it now has a transactional context, containing amongst others a snapshot of the transactional memory at the current point ($\bar{\sigma} :: \delta$) and an empty local store. A new

fork_{|c}

$A, T \cup \langle f, a, \mathcal{E}[\text{fork } e], F_c, F_j, \text{eff}, \text{ctx}^? \rangle, \mu, \tau, \sigma \rightarrow A, T \cup \langle f, a, \mathcal{E}[f_*], F_c \cup f_*, F_j, \text{eff}, \text{ctx}^? \rangle \cup \text{task}_*, \mu, \tau, \sigma$
 with $\text{task}_* = \langle f_*, a, e, \emptyset, F_j, \langle \emptyset, \bullet \rangle, \text{ctx}_*^? \rangle$ and f_* fresh

$$\text{ctx}_*^? = \begin{cases} \bullet & \text{if } \text{ctx}^? = \bullet \\ \langle n, \bar{\sigma} :: \delta, \emptyset, \langle \emptyset, \bullet \rangle \rangle & \text{if } \text{ctx}^? = \langle n, \bar{\sigma}, \delta, \langle \bar{A}, \bar{\text{beh}}^? \rangle \end{cases} \quad \begin{array}{l} \text{(outside transaction)} \\ \text{(in transaction)} \end{array}$$

join₁|c

$A, T \cup \langle f, a, \mathcal{E}[\text{join } f_*], F_c, F_j, \text{eff}, \text{ctx}^? \rangle \cup \text{task}_*, \mu, \tau, \sigma$
 $\rightarrow A, T \cup \langle f, a, \mathcal{E}[v], F_c, F_j \cup F_j^*, \text{eff}_+, \text{ctx}_+^? \rangle \cup \text{task}_*, \mu, \tau, \sigma$

where $\text{task}_* = \langle f_*, a, v, F_c^*, F_j^*, \text{eff}_*, \text{ctx}_*^? \rangle$ (same actor)

if $f_* \notin F_j$ and $F_c^* \subseteq F_j^*$ (first join; must have joined its children)

with $\text{eff}_+ = \text{eff} += \text{eff}_*$

$$\text{ctx}_+^? = \begin{cases} \bullet & \text{if } \text{ctx}^? = \bullet \text{ and } \text{ctx}_*^? = \bullet \\ \text{ctx}^? & \text{if } \text{ctx}^? \neq \bullet \text{ and } \text{ctx}_*^? = \bullet \\ \text{ctx}^? += \text{ctx}_*^? & \text{if } \text{ctx}^? \neq \bullet \text{ and } \text{ctx}_*^? \neq \bullet \end{cases} \quad \begin{array}{l} \text{(both non-transactional)} \\ \text{(non-tx'al into transactional)} \\ \text{(both transactional)} \end{array}$$

join₂|c

$A, T \cup \langle f, a, \mathcal{E}[\text{join } f_*], F_c, F_j, \text{eff}, \text{ctx}^? \rangle \cup \text{task}_*, \mu, \tau, \sigma \rightarrow$

$A, T \cup \langle f, a, \mathcal{E}[v], F_c, F_j, \text{eff}, \text{ctx}^? \rangle \cup \text{task}_*, \mu, \tau, \sigma$

where $\text{task}_* = \langle f_*, a_*, v, F_c^*, F_j^*, \text{eff}_*, \text{ctx}_*^? \rangle$

if $f_* \in F_j$ or $a \neq a_*$ (subsequent join or different actor)

Fig. 10. Rules concerning futures.

task's set of forked tasks (F_c) is empty, as it has not forked any futures. However, its joined tasks F_j are copied from its parent: When these are joined again, their effects should not be applied again.

join. There are two rules to join futures: when joining a future for the first time within the same actor ($\text{join}_1|_c$), and when joining a future subsequently or within another actor ($\text{join}_2|_c$).

In $\text{join}_1|_c$, when a future from the same actor is joined for the first time ($f_* \notin F_j$), its effects on actors and the transactional context are merged into the task performing the join, using the operator $+=$ defined in Section B.5. We require that the task f_* has joined all of its children ($F_c^* \subseteq F_j^*$), so the effects of its children have been merged into f_* and are now present in its eff and ctx , allowing us to simply merge these. If the task has not joined all its children, no rule is applicable in PureChocola; in the actual implementation of Chocola an error is raised.

Joining a transactional task into a non-transactional task is not allowed: The transactional task has effects on transactional state that the non-transactional task cannot handle. In practice, this will raise an error. The opposite, joining a non-transactional task into a transactional task, is no problem, as the non-transactional task does not have any side effects.

The rule $\text{join}_2|_c$ simply resolves to the future's value without merging effects and occurs in two cases. First, in the case of subsequent joins: The effects are already present and do not need to be merged again; second, when joining a future from a different actor, which can occur when an actor sends a future in a message. Merging the effects of the task f_* from actor a_* into the task f of actor a is not desirable: Effects on actors should not be "transferred" between actors, as this could cause them to be duplicated. Instead, these effects will be merged into the parent of f_* , which exists in the same actor a_* , as a result of the rule that requires parent tasks to join all of their children (a condition on the rules $\text{join}_1|_c$ and $\text{turn-end}|_c$).

$\text{atomic}_c|_c$

$$\frac{\Lambda, T \cup \langle f, a, \mathcal{E}[\text{atomic } e], F_c, F_j, \text{eff}, \bullet \rangle, \mu, \tau, \sigma}{\rightarrow \Lambda, T \cup \langle f, a, \mathcal{E}[\text{atomic}\star e], F_c, F_j, \text{eff}, \text{ctx} \rangle, \mu, \tau[n \mapsto \langle \triangleright, e \rangle], \sigma}$$

with $\text{ctx} = \langle n, \sigma, \emptyset, \langle \emptyset, \bullet \rangle \rangle$ and n fresh

 $\text{atomic}_{\text{tx}}|_c$

$$\frac{\mathcal{T}\langle f, a, \mathcal{E}[\text{atomic } e], F_c, F_j, \text{eff}, \text{ctx} \rangle}{\rightarrow \mathcal{T}\langle f, a, \mathcal{E}[e], F_c, F_j, \text{eff}, \text{ctx} \rangle}$$

 $\text{ref}_c|_c$

$$\frac{\mathcal{T}\langle f, a, \mathcal{E}[\text{ref } v], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle}{\rightarrow \mathcal{T}\langle f, a, \mathcal{E}[r], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta[r \mapsto v], \text{eff}_{\text{tx}} \rangle \rangle}$$

with r fresh

 $\text{deref}_c|_c$

$$\frac{\mathcal{T}\langle f, a, \mathcal{E}[\text{deref } r], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle}{\rightarrow \mathcal{T}\langle f, a, \mathcal{E}[(\tilde{\sigma} :: \delta)(r)], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle}$$

 $\text{ref-set}_c|_c$

$$\frac{\mathcal{T}\langle f, a, \mathcal{E}[\text{ref-set } r v], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle}{\rightarrow \mathcal{T}\langle f, a, \mathcal{E}[v], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta[r \mapsto v], \text{eff}_{\text{tx}} \rangle \rangle}$$

 $\text{commit}_{\checkmark}|_c$

$$\frac{\Lambda \cup \text{act}, T \cup \langle f, a, \mathcal{E}[\text{atomic}\star v], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle, \mu, \tau[n \mapsto \langle \triangleright, \tilde{e} \rangle], \sigma}{\rightarrow \Lambda \cup \text{act}, T \cup \langle f, a, \mathcal{E}[v], F_c, F_j, \text{eff}_+, \bullet \rangle, \mu, \tau[n \mapsto \langle \checkmark, \tilde{e} \rangle], \sigma :: \delta}$$

where $\text{act} = \langle a, f_{\text{root}}, \text{beh}, n_{\text{dep}}^? \rangle$

- if $\forall r \in \text{dom}(\delta) : \sigma(r) = \tilde{\sigma}(r)$ (no conflicts)
- $\forall f_* \in \text{tx-futs}(T, n) : f_* \in F_j$ (all futures forked in the tx must be joined)
- $n_{\text{dep}}^? = \bullet$ or $\tau(n_{\text{dep}}^?) = \langle \checkmark, \tilde{e} \rangle$ (in a definitive or a successful tentative turn)

with $\text{eff}_+ = \text{eff} += \text{eff}_{\text{tx}}$

 $\text{commit}_{\times}|_c$

$$\frac{\Lambda, T \cup \langle f, a, \mathcal{E}[\text{atomic}\star v], F_c, F_j, \text{eff}, \langle n, \tilde{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle, \mu, \tau[n \mapsto \langle \triangleright, \tilde{e} \rangle], \sigma}{\rightarrow \Lambda, T \cup \langle f, a, \mathcal{E}[\text{atomic } \tilde{e}], F_c, F_j, \text{eff}, \bullet \rangle, \mu, \tau[n \mapsto \langle \times, \tilde{e} \rangle], \sigma}$$

if $\exists r \in \text{dom}(\delta) : \sigma(r) \neq \tilde{\sigma}(r)$ (a conflict occurred)

 $\text{commit}_{\times}|_c$

$$\frac{\Lambda \cup \text{act}, T \cup \langle f, a, \mathcal{E}[\text{atomic}\star v], F_c, F_j, \text{eff}, \text{ctx} \rangle, \mu, \tau, \sigma}{\rightarrow \Lambda \cup \text{act}', T', \mu, \tau', \sigma}$$

where $\text{act} = \langle a, f_{\text{root}}, \text{beh}, n_{\text{dep}} \rangle$

- if $\tau(n_{\text{dep}}) = \langle \times, \tilde{e} \rangle$ (in a failed tentative turn)
- with $\text{act}' = \langle a, \bullet, \text{beh}, \bullet \rangle$ (reset actor to idle state)
- $T' = T \setminus \text{actor-tasks}(T, a)$ (abort and remove all tasks in this turn)
- $\tau'(n) = \begin{cases} \langle \times, \text{nil} \rangle & \text{if } n \in \text{actor-txs}(a) \\ \tau(n) & \text{otherwise} \end{cases}$ (abort all transactions in this turn, including current)

Fig. 11. Rules concerning transactions.

9.2.3 Transactions. The rules handling transactions are shown in Figure 11. We describe their changes compared to Section 2.2.3.

atomic, ref, deref, ref-set. The rule $\text{atomic}_c|_c$ has been modified to store the transactional state in ctx instead of directly in the task. The other rules work almost exactly like before.

commit. The commit rules have been modified to take into account whether the current turn is definitive or tentative. Before a transaction can successfully commit, in the rule $\text{commit}_{\checkmark}|_c$, the current turn must either be definitive or its dependency must have succeeded ($n_{\text{dep}}^? = \bullet$ or $\tau(n_{\text{dep}}^?) = \langle \checkmark, \tilde{e} \rangle$). This ensures that changes that are the result of a tentative message are persisted to the transactional heap only when it is sure the dependency succeeded. In $\text{commit}_{\times}|_c$, this condition is

not necessary: After a conflict, the transaction will not write any changes to the transactional heap anyway. We also add the rule $\text{commit}_{|c}$, which matches when the current transaction runs in a tentative turn whose dependency failed ($\tau(n_{\text{dep}}) = \langle \times, \tilde{e} \rangle$). Then:

- The current turn is abandoned and the actor returns to an idle state. Any changes that occurred in the current turn are thus discarded, as they are the result of an invalid message.
- All other tasks that were active in this turn are also stopped and removed.
- This transaction and all other transactions in this turn are marked as aborted in τ . This ensures that tentative messages sent by these transactions are now invalid. Note that other transactions in this turn cannot have committed yet, as they must also wait for the dependency.

No rule applies when the dependency is still running, i.e., $n_{\text{dep}} \neq \bullet$ and $\tau(n_{\text{dep}}) = \langle \triangleright, \tilde{e} \rangle$. As a result, the reduction of the current task will be stuck until its dependency commits or aborts, at which point either $\text{commit}_{\surd|c}$ or $\text{commit}_{|c}$ applies and the current task can proceed.

Additionally, the rules $\text{commit}_{\surd|c}$ and $\text{commit}_{\times|c}$ have two other modifications. First, all tasks that were forked in a transaction must have been joined before the commit. This ensures that all effects have been merged into the root task of this transaction, and can therefore be applied atomically. Second, the effects on actors that occurred in the transaction are merged into the task upon a successful commit or discarded if the transaction aborts.

9.2.4 Actors. Figure 12 shows the operations on actors. Section 6.3 described how these work in each of three contexts: in a transaction, outside a transaction in a tentative turn, and outside a transaction in a definitive turn; the reduction rules likewise differentiate between these cases.

spawn. `spawn` creates a new actor, but it is not immediately active: it is not added to A in the program state. Outside a transaction, the new actor is stored in the effects of the current task, eff (both in definitive and tentative turns). The actor will become active if and when the current turn ends successfully (rule $\text{turn-end}_{|c}$). In a transaction, the new actor is stored in the effects of the transaction, eff_{tx} . If the transaction commits successfully, these effects will be merged into the effects of the task upon commit, which will eventually be executed at the end of the turn. If the transaction aborts, the effects are discarded. The inbox is created immediately, though, as it should be able to receive (possibly tentative) messages immediately.

become. `become` similarly distinguishes these two cases: if no transaction is active, the effect is stored in the task; otherwise, it is stored in the transaction.

send. Messages are always sent immediately, but can be tentative. This is indicated through an additional parameter $n_{\text{msg}}^?$, which refers (1) to the current transaction in a transaction, (2) to the current dependency in a tentative turn, or (3) is \bullet in a definitive turn.

Receive (start of turn). The rule $\text{receive}_{|c}$ is triggered when an actor takes a message from its inbox. As before, it binds the parameters of the code in the behavior to the respective values in the internal memory and the message. A root task is created to evaluate this expression and its future is stored in the actor. If the message is tentative, its dependency is copied to the actor, causing the turn to be tentative as well. ($n_{\text{dep}}^?$ can be \bullet , too, then the turn is definitive.)

End of turn. Finally, the rule $\text{turn-end}_{|c}$ is evaluated when the turn ends, i.e., when its root task has been reduced to a single value. The actor moves to an idle state and loses its dependency. We require that the root task has joined all its children. In turn, these children must have joined their children (required by rule $\text{join}_1|c$). (In contrast to the formal semantics, the implementation raises

spawn_c

$$A, T \cup \langle f, a, \mathcal{E}[\text{spawn } b, \bar{v}], F_c, F_j, \text{eff}, \text{ctx}^? \rangle, \mu, \tau, \sigma$$

$$\rightarrow A, T \cup \langle f, a, \mathcal{E}[a_*], F_c, F_j, \text{eff}', \text{ctx}' \rangle, \mu[a_* \mapsto []], \tau, \sigma$$

with $\text{act}_* = \langle a_*, \bullet, \langle b_*, \bar{v} \rangle, \bullet \rangle$ and a_* fresh

$$\begin{cases} \text{if } \text{ctx}^? = \bullet: & \text{eff}' = \text{eff} += \langle \text{act}_*, \bullet \rangle & \text{ctx}' = \bullet & \text{(outside tx)} \\ \text{if } \text{ctx}^? = \langle n, \bar{\delta}, \delta, \text{eff}_{\text{tx}} \rangle: & \text{eff}' = \text{eff} & \text{ctx}' = \langle n, \bar{\delta}, \delta, \text{eff}_{\text{tx}} += \langle \text{act}_*, \bullet \rangle \rangle & \text{(in tx)} \end{cases}$$

become_c

$$\mathcal{T} \langle f, a, \mathcal{E}[\text{become } b, \bar{v}], F_c, F_j, \text{eff}, \text{ctx}^? \rangle \rightarrow \mathcal{T} \langle f, a, \mathcal{E}[\text{nil}], F_c, F_j, \text{eff}', \text{ctx}' \rangle$$

$$\text{with } \begin{cases} \text{if } \text{ctx}^? = \bullet: & \text{eff}' = \text{eff} += \langle \emptyset, \langle b_*, \bar{v} \rangle \rangle & \text{ctx}' = \bullet & \text{(outside tx)} \\ \text{if } \text{ctx}^? = \langle n, \bar{\delta}, \delta, \text{eff}_{\text{tx}} \rangle: & \text{eff}' = \text{eff} & \text{ctx}' = \langle n, \bar{\delta}, \delta, \text{eff}_{\text{tx}} += \langle \emptyset, \langle b_*, \bar{v} \rangle \rangle \rangle & \text{(in tx)} \end{cases}$$

send_c

$$A \cup \text{act}, T \cup \langle f, a, \mathcal{E}[\text{send } a_{\text{to}} \bar{v}], F_c, F_j, \text{eff}, \text{ctx}^? \rangle, \mu[a_{\text{to}} \mapsto \overline{\text{msg}}], \tau, \sigma$$

$$\rightarrow A \cup \text{act}, T \cup \langle f, a, \mathcal{E}[\text{nil}], F_c, F_j, \text{eff}, \text{ctx}^? \rangle, \mu[a_{\text{to}} \mapsto \overline{\text{msg}} \cdot \langle \bar{v}, n_{\text{msg}}^? \rangle], \tau, \sigma$$

where $\text{act} = \langle a, f_{\text{root}}, \text{beh}, n_{\text{dep}}^? \rangle$

$$\text{with } n_{\text{msg}}^? = \begin{cases} n_{\text{tx}} & \text{if } \text{ctx}^? = \langle n_{\text{tx}}, \bar{\delta}, \delta, \text{eff}_{\text{tx}} \rangle & \text{(in transaction)} \\ n_{\text{dep}}^? & \text{if } \text{ctx}^? = \bullet \text{ and } n_{\text{dep}}^? \neq \bullet & \text{(in tentative turn)} \\ \bullet & \text{otherwise} & \text{(in definitive turn)} \end{cases}$$

receive_c

$$A \cup \langle a, \bullet, \text{beh}, \bullet \rangle, T, \mu[a \mapsto \langle \bar{v}_{\text{msg}}, n_{\text{dep}}^? \rangle \cdot \overline{\text{msg}}], \tau, \sigma \rightarrow A \cup \langle a, f_*, \text{beh}, n_{\text{dep}}^? \rangle, T \cup \text{task}, \mu[a \mapsto \overline{\text{msg}}], \tau, \sigma$$

with $\text{task} = \langle f_*, a, e_*, \emptyset, \emptyset, \langle \emptyset, \bullet \rangle, \bullet \rangle$ and f_* fresh and $e_* = \text{bind}(\text{beh}, \bar{v}_{\text{msg}})$

turn-end_c

$$A \cup \langle a, f_{\text{root}}, \text{beh}, n_{\text{dep}}^? \rangle, T \cup \text{task}_{\text{root}}, \mu, \tau, \sigma \rightarrow A \cup \langle a, \bullet, \text{beh}', \bullet \rangle \cup \bar{\mathcal{A}}', T \cup \text{task}_{\text{root}}, \mu, \tau, \sigma$$

where $\text{task}_{\text{root}} = \langle f_{\text{root}}, a, v, F_c, F_j, \langle \bar{\mathcal{A}}, \overline{\text{beh}}^? \rangle, \bullet \rangle$

if $F_c \subseteq F_j$

(all futures forked in the turn must be joined)

$$\text{with } \begin{cases} \text{if } n_{\text{dep}}^? = \bullet \text{ or } \tau(n_{\text{dep}}^?) = \langle \checkmark, \bar{e} \rangle: & \text{beh}' = \overline{\text{beh}}^? \mid \text{beh} & \bar{\mathcal{A}}' = \bar{\mathcal{A}} & \text{(definitive, or successful tentative)} \\ \text{if } \tau(n_{\text{dep}}^?) = \langle \times, \bar{e} \rangle: & \text{beh}' = \text{beh} & \bar{\mathcal{A}}' = \emptyset & \text{(failed tentative turn)} \end{cases}$$

Fig. 12. Rules concerning actors.

an error if this is not the case.) As a result, the effects of all tasks created in this turn have been merged, ultimately becoming part of the root task's effects.

The turn can succeed or fail. When it is definitive ($n_{\text{dep}}^? = \bullet$) or when it is tentative and its dependency committed ($\tau(n_{\text{dep}}^?) = \langle \checkmark, \bar{e} \rangle$), the turn succeeds so its delayed effects are executed: The actor's behavior is updated (if necessary) and newly spawned actors become active. When a turn is tentative and its dependency has aborted ($\tau(n_{\text{dep}}^?) = \langle \times, \bar{e} \rangle$), it fails, so its delayed effects are discarded. When the turn is tentative and its dependency is still executing (\triangleright), this rule will not be applicable: The reduction of this actor is stuck until the dependency finishes.

9.3 Formalization of Guarantees

We formalize Chocla's guarantees here and try to give the intuition behind these theorems. Corresponding formal proofs are given in the appendices.

9.3.1 Intratransactional Determinacy. We first introduce several definitions. In Section 2.1.3, we defined the equivalence of two program states up to renaming; here, we define the notion of equivalent up to message ordering.

Definition 9.1 (Equivalence Up to Message Ordering \cong). Two program states are **equivalent up to message ordering** when they are equivalent except for the order of the messages in the inboxes of the actors. Formally, we write $p_1 \cong p_2$ for $p_1 = A, T, \mu, \tau, \sigma$ and $p_2 = A', T', \mu', \tau', \sigma'$ if:

- $A, T, \mu, \tau, \sigma \cong A', T', \mu, \tau', \sigma'$ (the states are equivalent except for the inboxes), and
- $\exists \alpha : \text{dom}(\mu) \rightarrow \text{dom}(\mu')$ (a renaming of equivalent actor addresses), so
- $\forall a \in \text{dom}(\mu) : \mu(a)$ is a permutation of $\mu'(\alpha(a))$ after transaction numbers have been renamed (inboxes of equivalent actors contain the same messages, but not necessarily in the same order).

Definition 9.2 (Intratransactional Reduction \xrightarrow{n}). The reduction of the expression e in a transaction with number n looks as follows: $A, T \cup \langle f, a, e, F_c, F_j, \text{eff}, \langle n, \bar{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle, \mu, \tau, \sigma \rightarrow A', T' \cup \langle f, a, e', F'_c, F'_j, \text{eff}', \langle n, \bar{\sigma}', \delta', \text{eff}'_{\text{tx}} \rangle \rangle, \mu', \tau', \sigma'$. We refer to this as an **intratransactional reduction** and will sometimes annotate it with the notation \xrightarrow{n} .

Definition 9.3 (Ready-to-commit States). We say a program state $A, T \cup \langle f, a, \mathcal{E}[\text{atomic}\star v], F_c, F_j, \text{eff}, \langle n, \bar{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle, \mu, \tau, \sigma$ is **ready to commit transaction n** . This state is running the transaction n in task f , which has been fully reduced to a value v . It corresponds to the left-hand side of the commit rules, i.e., the next rule to trigger in the task f is one of the commit rules.

THEOREM 9.4 (INTRATRANSACTIONAL DETERMINACY). *Given $p_0 = \mathcal{P}[\text{atomic}\star e]$, if $p_0 \xrightarrow{n^*} p_1$ and $p_0 \xrightarrow{n^*} p_2$, with p_1 and p_2 states ready to commit transaction n , then $p_1 \cong p_2$.*

Intuitively, this theorem can be understood as follows: A transaction starts with a given state and then evaluates until the point just before it commits. Any state it can reach is equivalent, except for the order in which messages are sent. In other words, different evaluations might take different steps during the reduction of the transaction, but eventually they will all lead to equivalent end results just before the commit. This corresponds to maintaining determinacy *within* each transaction. We prove this theorem in Appendix C.

9.3.2 Snapshot Isolation. PureChocola guarantees snapshot isolation as defined in Theorem 2.4. A proof is given in Appendix D. Intuitively, it is quite easy for Chocola to maintain snapshot isolation: While multiple tasks can be forked within each transaction, all their changes are first combined into the transaction's root task before they are committed in a single step. Hence, each transaction still starts from one snapshot and still commits in one step: Chocola's commit protocol is unchanged. Moreover, in a tentative turn, a transaction can commit only if the dependency has committed, but this affects only the order in which transactions are committed and not the actual commit protocol.

9.3.3 Consistent Turn Principle. The isolated turn principle was defined in Section 2.3.1 as a combination of three constituent guarantees: continuous message processing, consecutive message processing, and isolation. As PureChocola allows memory to be shared between actors (using transactions), the isolated turn principle clearly no longer holds. Instead, we define a slightly less restrictive guarantee, which we call the **consistent turn principle**. The consistent turn principle replaces the isolated turn principle's requirement for the isolation of all memory with a requirement of the isolation of actor memory only $\boxed{\text{Iso}} \rightarrow \boxed{\text{IsoAM}}$.

LEMMA 9.5 (ISOLATION OF ACTOR MEMORY). *The internal memory of an actor is isolated: After the actor’s creation, its internal memory can be read and written only by the actor itself.*

THEOREM 9.6 (CONSISTENT TURN PRINCIPLE). *PureChocola provides the consistent turn principle: It guarantees continuous message processing, consecutive message processing, and isolation of actor memory.*

Continuous and consecutive message processing were defined in Lemmas 2.6 and 2.7 of Section 2.3.3 and still hold for Chocola. We prove the three lemmas and the theorem in Appendix E.

9.3.4 *Deadlock Freedom and Continuous Message Processing.* In Definition 2.5 (in Section 2.3.3), we said that “a set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause” [59]. In PureChocola, the following transition rules “wait” for events, i.e., these rules trigger only when some non-local state conforms a certain pattern:

- The rules $\text{join}_1|_c$ and $\text{join}_2|_c$ wait until the task that is being joined has finished its reduction.
- The different commit rules and the rule $\text{turn-end}|_c$, if there is a dependency on another transaction, wait until the dependency has finished reducing and has committed or aborted.
- The rule $\text{receive}|_c$ waits until a message is available in the current actor’s inbox.

The last case can lead to a deadlock when two actors are blocked waiting for a message from the other. The actor model is inherently prone to such deadlocks at the level of messages—it guarantees deadlock freedom inside a turn only—and we will therefore not further consider this issue. We formalize the absence of deadlocks for the two other cases. We prove these theorems in Appendix F.

THEOREM 9.7 (DEADLOCK FREEDOM OF FUTURES). *Assuming no futures are stored in transactional memory, the tasks created within a turn cannot deadlock. A set of tasks is deadlocked if each task in the set is waiting for the future of another task created in the same turn to be resolved.*

The intuition behind this theorem is that the tasks within a turn form a “fork tree.” In such a tree, no cycles are possible, thereby avoiding the cyclical dependencies that lead to deadlocks.

THEOREM 9.8 (DEADLOCK FREEDOM (CONTINUOUS MESSAGE PROCESSING) OF TRANSACTIONAL ACTORS). *Deadlocks between transactions in tentative turns are impossible, as dependencies always go from newer to older transaction attempts, defining a total order on the transaction attempts.*

The intuition behind this theorem is that an actor acquires a dependency when receiving a message, i.e., at the start of a turn, and this dependency must therefore necessarily point to an older transaction. That older transaction cannot acquire a new dependency, because while it is running, it cannot receive a message. Hence, cyclical dependencies between transactions are impossible.

Note on Deadlock Freedom of Transactions. In Section 2.2.5, we discussed deadlocks in transactions. The same remark applies here: Deadlocks are prevented using the MVCC algorithm.

10 IMPLEMENTATION

We have implemented Chocola on top of Clojure.¹² Like Clojure, Chocola is partially implemented in Clojure itself, and partially in Java. Clojure already provides futures and transactions. To use Chocola, programmers must load it as an extension of Clojure when their program starts, at which

¹²It is available online at <http://soft.vub.ac.be/~jswalens/chocola/> and <https://github.com/jswalens/chocolalib>.

point it will replace parts of Clojure’s implementation of futures and transactions and add actors, to provide the semantics discussed in this article. We briefly summarize our implementation here.¹³

While the implementation provides the semantics as specified in Section 9, it is not a translation of the reduction rules. The differences with PureChocola are listed at the end of this section.

Actors. Chocola adds a simple implementation of actors to Clojure. A behavior definition is converted into a nested function, with the parameters of the outer function corresponding to the internal memory and those of the inner function to the message’s values. An actor stores three components: its behavior definition, its internal memory, and its inbox (using Java’s `LinkedListBlockingDeque`). Every actor consists of a thread that executes an infinite loop that takes a message from the inbox and evaluates the behavior definition with the internal memory and the message’s values.

Transactional Actors. We made these changes to support transactional actors:

- At the start of a turn, it is marked tentative if the processed message has a dependency.
- When a message is sent, a dependency may be attached: In a transaction, the dependency is that transaction; in a tentative turn but outside a transaction, the dependency of the turn is used; and in a definitive turn it is null.
- Depending on the context, the effects of `become` and `spawn` are delayed.
- At the end of a tentative turn, the actor waits until the dependency has finished. If it succeeded, newly spawned actors are started; otherwise, they are discarded and the old behavior is restored.

Transactional Futures. Most of the data structures that Clojure stores in the class `Transaction` were moved to a new class `TransactionalContext`, which is associated with a (transactional) future. Each transaction contains one “root” `TransactionalContext`, which contains its root future’s transactional data structures. We explain how transactional futures are forked and joined (in `TransactionalContext`) and which changes were made to running and aborting a transaction.

In Section 5.3, we explained that *forking a future* conceptually copies its parent’s snapshot and local store. In our implementation, we avoid creating duplicates, using the technique described by Swalens et al. [56]. Instead of representing these data structures as maps, they are represented as a tree of maps. The root future’s snapshot and local store start empty. When forking a future: (1) the child’s snapshot is a reference to the parent’s current local store; (2) the child’s local store is an empty wrapper around the parent’s local store; (3) the parent’s local store is modified to be a wrapper around its current local store, so further modifications in the parent are not visible by the child. Looking up a value consists of first looking it up in the current map, then in the map higher up the tree, and so on until the root of the tree.

When *joining a future*, the current task first waits until the child has finished and then merges its data structures into those of the parent, as in the formal semantics.

Transactions mostly *run* as before and use the data structures from the transactional context of the root future during commit. There is one important change: there are now two kinds of *aborts*. On the one hand, `RetryException` (which already exists in Clojure) is raised when a transaction encounters a conflict, and corresponds to the rule `commitx|c` in the formal semantics. This causes the transaction to abort and retry. On the other hand, `AbortException` (new in Chocola) is raised when a tentative message is invalid. This is checked at the end of a tentative turn (as in rule

¹³This section omits many details of our implementation. We have also renamed some constructs for clarity or brevity, e.g., `future` to `fork`, `dosync` to `atomic`, and `LockingTransaction` to `Transaction`.

turn-end_{|c} of the formal semantics) and during the commit of a transaction in a tentative turn (as in rule commit_{|c}). After an `AbortException`, the whole turn is aborted and the actor proceeds to the next message. (These two exceptions cannot be raised by the programmer; they are internal to our implementation.)

Compatibility with Clojure. These features of Clojure are safe/unsafe to use in Chocola:

- ✓ The **functional subset** of Clojure, i.e., all its purely functional built-in functions, will not break Chocola’s guarantees.
- ✗ In general, any **functions with side effects** cannot safely be used in Chocola; e.g., the use of non-deterministic input breaks determinacy, or waiting for the response to an HTTP request can break deadlock freedom.
- ✓ Clojure provides **futures** and **transactions**. When used separately, Chocola is backwards compatible. When they are combined, Chocola changes Clojure’s semantics as in Section 5.
- ✗ Any **other concurrency model** provided by Clojure is incompatible with Chocola: the guarantees can be broken, as in regular Clojure.

Differences with PureChocola. We list the key differences with the formal semantics PureChocola:

- Chocola is built on top of Clojure, while the base language of PureChocola is a functional λ calculus. It is generally unsafe to use functions from Clojure with side effects in Chocola.
- In Chocola intratransactional conflicts may be resolved using custom “conflict resolution functions.” This functionality has been omitted from PureChocola.
- PureChocola implements snapshot isolation straightforwardly, by taking a copy of the transactional memory. Chocola instead relies on the MVCC algorithm (as explained in Section 2.2). A notable consequence is that in Chocola, a transaction can abort early, while in PureChocola conflicts are only detected during commit.
- In PureChocola, commit_{|c} immediately stops all tasks and aborts all transactions in the current turn. In Chocola, this is implemented using `AbortException` (see above). Hence, in PureChocola all tasks seem to abort at exactly the same moment, while in Chocola they abort at different moments.
- PureChocola requires that tasks and turns join all their children. When this is not the case in Chocola, an exception is raised. In PureChocola, we did not make these errors explicit.

11 EVALUATION

In this section, we evaluate the performance of Chocola (quantitatively) and the effort required to use it (qualitatively). We first describe our methodology and experimental setup (Section 11.1). Next, we look at three applications that use transactions and show that introducing transactional futures or actors improves their performance by exploiting additional parallelism (Sections 11.2, 11.3, and 11.4). Finally, we discuss the developer effort required to make these changes (Section 11.5).

11.1 Methodology and Experimental Setup

In this section, we describe the goal and criteria of our evaluation, how we selected the benchmarks, and how we transformed them to use Chocola.

Evaluation Goal and Criteria. The goal of this evaluation is to demonstrate that, in existing programs that use transactions, additional parallelism can be exploited within those transactions by using Chocola, without fundamentally changing the design of the program. Hence, we select existing applications that use transactions and compare the original application with a transformation

Table 1. Characterization of the STAMP Applications from Minh et al. [48], on a Simulated 16-core System

Application	Tx length (instructions/tx)	Average time in tx	Contention (retries/tx)	Domain
Labyrinth	219,571 ■	100% ■	0.94 □	Engineering
Bayes	60,584 ■	83% ■	0.59 □	Machine learning
Yada	9,795 ■	100% ■	2.51 ■	Scientific
Vacation-high	3,223 □	86% ■	0.00 □	Transaction processing
Genome	1,717 □	97% ■	0.14 □	Bioinformatics
Intruder	330 □	33% □	3.54 ■	Security
Kmeans-high	117 □	7% □	2.73 ■	Data mining
SSCA2	50 □	17% □	0.00 □	Scientific

of the program that introduces transactional futures or actors. Our evaluation focuses specifically on programs with transactions, because the combination of actors and futures, discussed in Section 7, did not require major changes to the semantics of the separate models.

We compare the original and transformed programs using two criteria:

Performance The ultimate goal of introducing additional parallelism is to increase performance. Hence, we calculated the speed-up by comparing the execution time of (a part of) the transformed program with a varying number of threads with that of the original program.

Developer effort We (qualitatively) assess the effort that is required from the developer to use our techniques by discussing the changes that were made to the programs (in Section 11.5).

Selection of Benchmarks. Our evaluation is based on the STAMP benchmark suite [48]: eight applications, based on real-world use cases of transactions, commonly used to compare the performance of transactional systems and covering a range of characteristics (some shown in Table 1).

To evaluate transactional futures, we are interested in two characteristics in particular. First, the time spent in transactions: When most of the program’s execution occurs in transactions, further parallelization of these applications will have to occur *within* the transactions. Second, the transaction length: Long-running transactions may benefit from more fine-grained parallelism. As shown in Table 1, three applications exhibit these characteristics: Labyrinth, Bayes, and Yada.

To evaluate transactional actors, we look at a slightly reduced version of the Vacation benchmark, called Vacation2. This application already served as the basis for many of the code examples throughout this article. It is suited to the use of transactional actors, as the actor model naturally encodes the event-driven way in which a travel reservation system processes requests from customers. It is also an application that spends much of its time in transactions, and therefore may benefit from offloading parts of these transactions to different actors.

Transformation of Benchmarks. We started by porting the STAMP benchmarks from C to Clojure, retaining the design and algorithms of the original program. We then introduced futures and actors where applicable using the following steps. First, using profiling tools, we search for the atomic block in which the largest proportion of the program’s execution time is spent. Next, we search for the part of that transaction in which most time is spent. In our cases, this is always a loop. We try to parallelize this loop. To do this, we examine whether there are dependencies between the

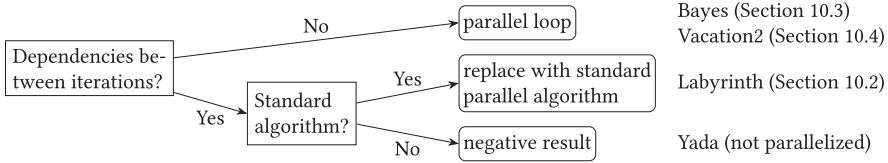


Fig. 13. The steps taken to transform the STAMP benchmarks into a version that uses Chocla.

iterations of the loop, which occur when an iteration uses the result of a previous iteration. There are three cases (illustrated in Figure 13):

- When the iterations are *independent*, we simply parallelize the loop. This occurs for Bayes and Vacation2.
- When there are *dependencies* between the iterations, but the program follows a *standard algorithm* for which a parallel version exists in literature, we replace the sequential algorithm with a parallel equivalent. This occurs for Labyrinth, in which a sequential breadth-first search is changed into a parallel search algorithm.
- When there are *dependencies* between the iterations and the program uses a *custom algorithm*, we reach a negative result and do not introduce futures or actors. This is the case for Yada: It uses a custom algorithm for Delaunay mesh refinement that either cannot be parallelized or requires specific domain expertise to do so. Hence, we do not further consider this application.

The code of all benchmarks is available online, both the original and the transformed versions.¹⁴ Each benchmark in the STAMP suite further has several parameters; we note their values for each experiment as we go along. All data points of the graphs in this section correspond to the median of 30 runs, with the error bars depicting the interquartile ranges. We show medians and interquartile ranges, because the results are usually not normally distributed. Note also that interquartile ranges depict the spread of the results and should be independent of sample size. We found a sample size of 30 to strike the balance between sufficiently fast experiments and sufficient precision.

In all experiments, the transformed benchmarks are compared with the original benchmark in Clojure. We do not compare to the implementations in C, as the performance characteristics of Clojure and C are so wildly different that they render a comparison meaningless.¹⁵

Hardware and Software Set-up. The experiments ran on a machine with four AMD Opteron 6376 processors, each containing 16 cores with a clock frequency of 2.3 GHz and a last-level cache of 16 MB, resulting in a total of 64 cores. The machine has 128 GB memory. We used Chocla 2.0.1 and Clojure 1.10.1, running on the OpenJDK 64-Bit Server VM (build 25.222-b10) for Java 1.8.0.¹⁶

¹⁴ <https://github.com/jswalens/labyrinth> • .../bayes • .../vacation2 • .../yada.

¹⁵ Just to name a few: a statically typed programming language with manual memory management vs. a dynamically typed language with garbage collection, an imperative vs. functional programming paradigm, and STMs using different algorithms.

¹⁶ Performing statistically rigorous performance experiments on virtual machines with just-in-time compilation, such as the Java Virtual Machine in our case, is notoriously difficult [7, 26]. As we are interested in the execution time of ephemeral benchmark programs that process some input to produce some output, and not the steady-state performance of continually running applications, our experiments all measure the execution time for all or part of the program from start to finish. Each measurement corresponds to a new execution of the program, for which a new instance of the JVM is started.

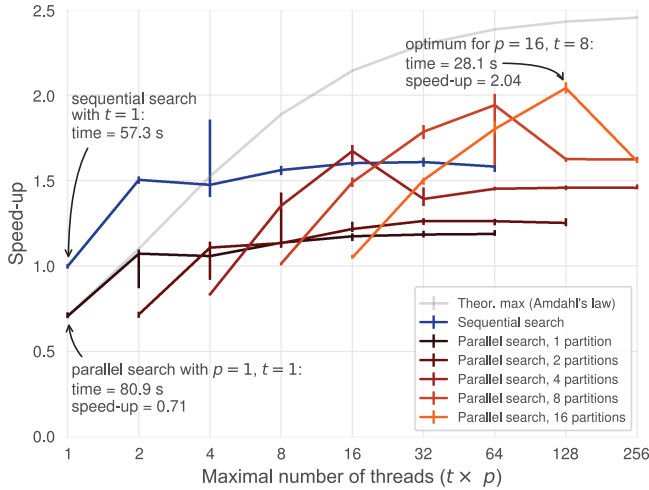


Fig. 14. Speed-up of the Labyrinth benchmark for the version with sequential search (blue line) and parallel search (other lines), as the total number of threads ($t \times p$) increases (logarithmic scale). The gray line indicates the theoretical maximal speed-up of the version using a parallel search, per Amdahl’s law.

11.2 Labyrinth

The Labyrinth benchmark connects points in a grid using non-overlapping paths. The program finds the cheapest path using a breadth-first search algorithm. This occurs in a transaction to ensure that paths do not overlap: If two transactions find overlapping paths, one will roll back and search an alternative. The program spends almost all its time performing these searches. We transformed the program to replace this with a standard parallel breadth-first search algorithm [68].

We ran the experiments on a three-dimensional grid of $50 \times 50 \times 50$ with 10 input pairs. Figure 14 shows the speed-up when varying two parameters: t : the number of worker threads that process input points in parallel, and p : the maximal number of partitions created in each iteration of the parallel search (only for the version with parallel search).¹⁷ The x axis denotes the (maximal) number of threads, which is t for the sequential search and $t \times p$ for the parallel search. The y axis shows the speed-up, calculated relative to the version with sequential search and one worker thread, which takes 57.3 s.

The blue line depicts the results of the original benchmark with the sequential search algorithm. Increasing the number of threads produces only a modest speed-up, because they find overlapping paths and consequently need to roll back and reexecute. This curtails any potential speed-up.

For the version with parallel search, both the number of partitions (different lines) and the number of worker threads (different points on the same line) are varied. As the number of partitions p increases, the speed-up improves: Each transaction forks p tasks and consequently finishes faster. On the tested hardware, an optimal speed-up of 2.04 is reached, when 8 worker threads process elements and create up to 16 partitions.

The parallel search with $p = 1$, which does not actually search in parallel as only one partition is created, is slower than the sequential search: The difference between the blue and the black line reflects the cost of switching to a parallel algorithm. We also annotated the graph with the

¹⁷To minimize the overhead of forking futures, we ensure that each partition contains at least 20 elements.

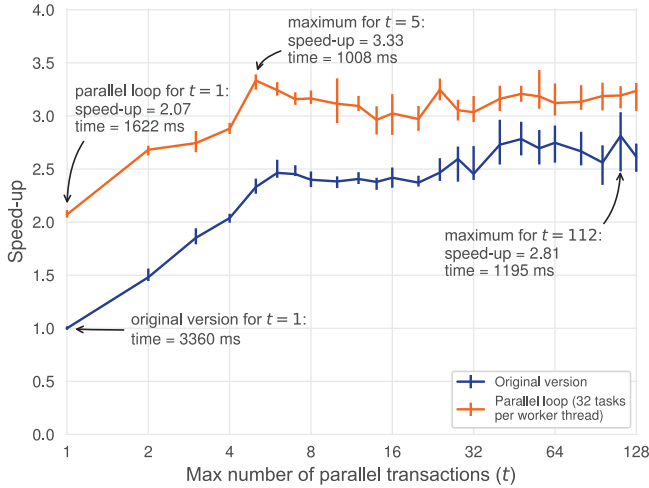


Fig. 15. Speed-up of the part of the Bayes benchmark that learns the structure of a network with 32 variables as the number of workers increases (logarithmic scale). The blue line shows the original version. The orange line shows the version with a `parallel-for` loop that executes (up to) 32 iterations in parallel.

theoretical maximum speed-up (gray line), calculated using Amdahl’s law.¹⁸ Profiling shows that the transformed program spends 71.4% of its execution time performing the search algorithm; the other 28.6% is spent in code that is not parallelized. Hence, if the program ran on a theoretical machine with an infinite number of cores, its maximal speed-up would be 3.5. The gray line indicates this theoretical limit for varying number of threads.

These results demonstrate two ways in which transactional futures improve performance. First, transactions run faster by exploiting parallelism in the transaction. Second, conflicts are cheaper, because each attempt runs faster. By varying the two parameters, we can find an optimum between running several transactions simultaneously but risking conflicts (t) and speeding up the transactions internally but with more costly fine-grained parallelism (p).

11.3 Bayes

The Bayes benchmark implements an algorithm that learns the structure of a Bayesian network: Starting from a network of v variables without dependencies, it adds dependencies to maximize its ability to predict the input data. Each variable of the network is represented as a transactional variable. t worker threads process a shared work queue in parallel: They insert a dependency into the network, then calculate which dependencies (if any) could be inserted next, and append the best candidate to the work queue. This is encapsulated in a transaction to ensure that conflicting threads cannot introduce cycles in the network. This transaction contains a `for` loop that calculates the score for each candidate. As each iteration of the loop is independent, it can be parallelized by replacing `for` with `parallel-for`, a construct of Chocla that executes the loop in parallel using transactional futures.

We ran the benchmark on a network of 32 variables with an input of 512 records, generating up to 8 parents per variable. Figure 15 shows the speed-up of the learning phase of this benchmark as the number of worker threads (t) increases, compared to the original version with one worker.

¹⁸See https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=917796330.

The blue line plots the speed-up of the original version. Initially, as the number of workers that run transactions in parallel increases, the speed-up increases. However, when using more than 6 workers, the speed-up plateaus. (The maximal speed-up is 2.81.) By examining the execution of the program, we find that the program generates a large initial amount of work, but that after a certain point in the execution not enough work is available to keep all threads busy. Hence, after that point, only a limited number of worker threads actually perform any work.

The orange line shows the version with `parallel-for`. Here, there are up to t transactions running in parallel and in each up to $v (= 32)$ transactional tasks run in parallel. When there is only one worker processing one transaction at a time ($t = 1$), the parallelization of this loop produces a speed-up of 2.07. By increasing the number of workers, a maximal speed-up of 3.33 is achieved when up to 5 transactions run in parallel, each containing a `parallel-for` with up to 32 iterations. Again, the speed-up plateaus, as not enough work is available for all worker threads. However, the reached speed-up is higher than the original version, as more fine-grained parallelism is exploited in each work item.

In conclusion, while in the original version the parallelism is limited to the number of transactions, we can improve performance by using transactional futures to exploit more fine-grained parallelism within the transactions.

11.4 Vacation2

Our Vacation2 benchmark is based on the Vacation benchmark from STAMP.¹⁹ At the start of the program, r flights, hotel rooms, and cars are generated, collectively called *items*, with a random price and a random number of seats. The input consists of c customers that want to book a holiday for one to five people. For every item type, customers randomly select a subset of q items, pick the cheapest with sufficient available seats, and reserve seats. Additionally, each customer generates a password using a cryptographically secure hash. We ran the experiments with $c = 1000$, $r = 50$, and $q = 10$.

Each item and each customer is stored in a transactional variable. In the original benchmark, there are p worker actors, over which the customers are evenly distributed. Each customer's reservation is encapsulated in a transaction, ensuring seats cannot be double-booked. We transformed the original benchmark to parallelize this transaction by reserving the different items in parallel, in separate actors. In the transformed version, next to p primary worker actors, there are s secondary worker actors. Now, each customer will send messages to secondary worker actors, which each reserve one item in a new transaction. Because the transactions in the secondary workers have a dependency on those in the primary workers, correctness remains guaranteed.

In Figure 16, the blue line depicts the speed-up of the original version. As the number of worker actors p increases, the speed-up increases up to 2.5 for 42 worker actors. On a machine with 64 cores this is very limited: This is due to the fact that increasing the number of transactions that run in parallel increases the chance of conflicts and thus the number of retries.

The other lines in the figure show the speed-up of the transformed program. Both the number of primary (the x axis) and secondary (different lines) worker actors are varied. When using only one secondary worker actor, customers are processed in parallel but the reservation of individual items is not. Increasing the number of primary worker actors results in a maximal speed-up of 6.1 for 32 primary worker actors: a better result than the original version. This is because there are far fewer conflicts: there is only one secondary actor reserving items, so there can never be any conflicts on the items.

¹⁹We add the suffix "2" to clearly indicate that, in contrast to the other benchmarks, we omitted some functionality: In the original STAMP benchmark, customers can be deleted and items can be changed, while Vacation2 does not support this.

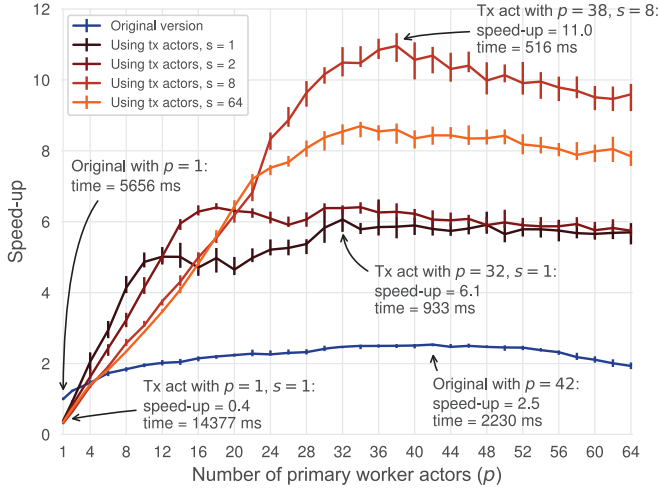


Fig. 16. Speed-up of the Vacation2 benchmark. The blue line shows the original version for an increasing number of worker actors (p). The other lines show the version using transactional actors for varying numbers of primary (p) and secondary (s) worker actors.

By increasing the number of secondary worker actors, the speed-up can be improved further. We see that a maximum speed-up of 11.0 is reached for 38 primary and 8 secondary worker actors on this machine. At this point, the balance between increased parallelism and a low chance of conflicts is optimal. Using more than 8 secondary worker actors will again lower the performance due to a higher chance of conflicts. This result demonstrates that this application benefits from being parallelized in two places: Instead of parallelizing only the processing of customers (as in the original version) or parallelizing only the reservation of items (the results of $p = 1$), the optimum is found by combining both.

This experiment capitalizes on another benefit of transactional actors as well: They allow a transaction to be split up into multiple transactions with dependencies. Every transaction in the original version was split into one primary and several dependent transactions. If the primary transaction aborts, the dependent transactions are aborted, too. However, if a dependent transaction aborts, no other transaction must abort. Hence, by using transactional actors, we lower the cost of a conflict in a dependent transaction, as only this part needs to retry.

Finally, we note the high overhead of transactional actors: while the original version took 5,656 ms when using a single worker actor, the version with transactional actors takes 14,377 ms when $p = 1$ and $s = 1$, more than twice as slow. We suspect this is due to our relatively simplistic implementation of actors and we believe further optimizations could improve performance.

11.5 Developer Effort

In this section, we discuss the effort required to use Chocla. We focus on the qualitative aspect: which changes were necessary to introduce transactional futures and transactional actors in a program with transactions, and how do they compare to the changes necessary to introduce regular futures and actors in a program without transactions. Table 2 summarizes the quantitative aspect: the number of lines of code that were changed.

11.5.1 Labyrinth. To introduce transactional futures in the Labyrinth application, 20 lines (3%) were removed and 72 lines (10%) were added out of 748 lines in total. Almost all of these changes

Table 2. The Number of Lines of Code Added and Removed to Introduce Transactional Futures or Actors

Benchmark	Added	Removed	Total lines of code
Labyrinth	72 (10%)	20 (3%)	748
Bayes	1 (<1%)	1 (<1%)	1,256
Vacation2	25 (7%)	17 (5%)	359

are a result of swapping the sequential search algorithm with a more complex parallel equivalent. Besides that, the developer needs to define a suitable conflict resolution function for the transactional variables that represent the grid. Thus, we observe that the effort required to introduce transactional futures in this program is similar to the effort required to parallelize any sequential code, with the definition of the conflict resolution function as a notable exception.

11.5.2 Bayes. To transform the original version of the Bayes application into the one that uses transactional futures, actually only one line (out of 1,256) had to be changed: `for` was replaced with `parallel-for`. This is possible, because each iteration of the loop is independent, exhibiting a type of parallelism sometimes described as “embarrassingly parallel” [36]. This benchmark thus epitomizes the small developer effort required to introduce transactional futures: While in a naive combination of transactions and futures, read operations on transactional state inside `parallel-for` would not be possible or give inconsistent results, here transactional futures easily deliver the expected result.

11.5.3 Vacation2. While in the original version of the benchmark a worker actor processes a reservation in a single transaction, in the modified version some of this work is split off to “secondary” worker actors. Accordingly in the code, we defined the behavior of these new actors and a part of the original transaction was moved to them. In total, out of 359 lines of code, 25 lines (7%) were added and 17 lines (5%) were removed. As transactional actors do not introduce new constructs, the same techniques that are used to introduce actors in a sequential program without transactions also apply here. Hence, using Chocola developers can reuse their existing knowledge of the models, because the guarantees of the separate models are maintained.

11.6 Conclusions

Based on these experiments, we draw the following conclusions:

- Of the four benchmarks of the STAMP suite with the longest transactions, in three cases, we found that we could improve performance by introducing futures or actors. (For the fourth case, further parallelization is impossible or requires more domain expertise.)
- The improved performance was a result of faster (internally parallel) transactions (Labyrinth and Bayes), a lower chance and cost of conflicts (Labyrinth and Vacation2), and the ability to exploit more fine-grained parallelism (Bayes).
- The required effort is mostly due to the introduction of additional parallelism, which would be necessary even outside a transaction and not due to specific requirements of our techniques. (A notable exception is the definition of the conflict resolution function in Labyrinth.)

These results demonstrate that Chocola enables developers to improve the performance of their transactional applications with only limited effort. Because Chocola does not introduce any new constructs, developers can now reuse their existing knowledge of the separate models even when

they are combined. Moreover, as our implementation is a relatively simple prototype in Clojure, further optimizations could decrease its overheads and improve its performance.

It should be possible to apply transactional futures and actors to other STM systems, such as Haskell or ScalaSTM. In those systems, the studied applications can benefit from parallelism in the transaction as well and the development effort to introduce them should be similar, but depending on the implementation the speed-up may be different.

12 RELATED WORK

To the best of our knowledge, no existing literature studies the semantics of combinations of more than two concurrency models. Below, we describe work related to the pairwise combinations of the models studied in this article.

Transactions and Futures. We describe existing work that combines transactions and futures/threads. The most prominent difference between these techniques lies in how they handle conflicts between tasks in a transaction, which we refer to as “intratransactional conflicts.”

Multithreaded transactions [29] are transactions in which threads are forked, and they work like the naive combination of Section 5.2: Threads do not run within their parent’s transactional context, permitting race conditions. *Nested transactions* [8, 50, 51] are transactions created in the context of another transaction, and several nested transactions may run in parallel. In *Transactional Featherweight Java* [62], transactions can fork threads that can start a nested transaction, and when the nested transaction commits, its changes are written to its parent. However, intratransactional conflicts are explicitly forbidden. Both (NPTs) *Nested Parallel Transactions* (NPTs) [3, 4, 6, 63] and our transactional futures (TFs) go a step further and solve such conflicts. NPTs resolve conflicts using the traditional serializability of transactions: When two nested transactions conflict, one of both will roll back and retry—which of the two is non-deterministic. In contrast, TFs rely on conflict resolution functions to deterministically solve intratransactional conflicts. Hence, NPTs forsake determinacy but arguably provide a more consistent semantics.

Independently from our work, Zeng et al. [67] developed (JTFs) *Java Transactional Futures* (JTFs). Here, an intratransactional conflict causes a future to roll back. Which future is rolled back is deterministic, as JTFs maintain the semantic transparency of futures [25]. Hence, JTFs guarantee intratransactional determinacy, like our TFs and unlike NPTs. The three models also have different performance characteristics: When intratransactional conflicts are prevalent (e.g., in the Labyrinth benchmark from Section 11.2), TFs will outperform NPTs and JTFs, as rollbacks are avoided. Hence, our transactional futures forsake semantic transparency to improve performance in these cases.

Finally, *Concurrent Revisions* [15] and *Worlds* [64] do not provide transactions, but use a future-like model to guarantee determinacy for the whole program. *Chocola’s* semantics of futures in transactions is inspired by these models: *Chocola* combines serializable transactions at the top level with determinate futures within the transactions.

Transactions and Actors. We first look at three approaches that add communication to a transactional system. *Transactions with Isolation and Cooperation* [55] can temporarily “suspend” their atomicity and isolation to exchange data, making communication between transactions possible but breaking the isolation guarantee. Similarly, *Transaction Communicators* [47] are a special type of object through which transactions can communicate, but again breaking isolation. Third, *Communicating Memory Transactions* [45] combine transactions with message passing over channels. When a message is sent, a dependency is introduced from the receiver to the sender, similar to our approach. However, cyclical dependencies are not prevented and lead to a deadlock. In contrast, *Chocola* prevents cyclical dependencies by making it impossible to receive a message in a transaction.

Next, we highlight approaches to share memory between actors. *Sharing actors* [44] and *passive processors* [49] can share memory between multiple readers, but only one writer is allowed. *Domains* [19] are containers that can be accessed from multiple actors, but writing must happen asynchronously. Finally, Pony [17] and Encore [13] are languages that allow memory to be shared between actors, statically preventing race conditions using capabilities.

Futures and Actors. Imam and Sarkar [41] combine actors with the async–finish model (AFM), which is similar to futures. They allow a task to “escape” its turn (as in Figure 8) and prevent races by prohibiting this task from modifying the internal memory of its actor. In this model, it is also possible to coordinate the termination of actors, enabled by the AFM’s finish construct.

13 CONCLUSIONS AND FUTURE WORK

Many programming languages support a wide variety of concurrency models and these are often combined by developers. In this article, we studied the combination of futures, transactions, and actors. We found that naive combinations of these models can invalidate the guarantees that they normally provide, thereby breaking the assumptions of programmers.

We presented Chocola, a language that integrates futures, transactions, and actors into a unified model that maintains the guarantees of all three models wherever possible, even when they are combined. We formalized its semantics and proved its guarantees. We also implemented Chocola as an extension of Clojure and demonstrated that it can improve the performance of three benchmark applications for relatively little effort from the developer. Hence, using Chocola developers can freely pick and mix different concurrency models in their program, in each part using the model that fits best.

To the best of our knowledge, Chocola is the first programming language that combines more than two concurrency models. It is a first exploration in the design space of such combinations, focused on preserving the guarantees of separate concurrency models when they are combined. We highlight some high-level observations and ideas for future research:

- In our work, we have noticed that certain properties of concurrency models make combinations especially problematic, in particular non-determinism and the presence of constructs that retry or block. However, other properties facilitate combinations, such as determinism or the absence of side effects. Further exploration of different concurrency models can lead to a list of such problematic and helpful properties. This can then lead to a table of properties that do or do not combine well and possible solutions. For instance, using a model with side effects in a model with retrying operations is a problematic combination, which can be solved by either forbidding the side effects in those contexts, delaying them, or making it possible to roll them back.
- Our current base language lacks exceptions: The interactions between concurrency and exceptions have been the topic of previous research (e.g., for futures [53] and transactions [33, 34]) and can be challenging due to their non-local control flow, in some cases allowing an exception to “escape” a future or transaction.
- Our actor model is based on the original actor model of Agha [1], in which actor state can be updated only using become, which takes effect in the next turn. When using an actor model that allows mutable state, changes to the actor state in a transaction would need to either be forbidden (possibly enforced by the type system) or be possible to roll back.
- Some actor models feature an explicit receive statement. To avoid cyclic dependencies, it should not be possible to receive a (tentative) message while a transaction is active. In Chocola, whose actor model does not have an explicit receive statement, this is enforced

	Clojure	Scala	Java	C++	Haskell (GHC)
<i>Deterministic models</i>					
Futures	✓	✓	✓	✓	• ¹¹
Promises	✓	✓	✓	✓	• ¹¹
Fork/Join	✓ _J	✓ _J	✓	• ⁷	
Parallel collections	✓ _J	✓	✓	• ⁷	• ¹²
Dataflow	• ¹	• ²	• ⁴		• ¹³
<i>Shared-memory models</i>					
Threads	✓ _J	✓ _J	✓	✓	✓
Locks	✓ _J	✓ _J	✓	✓	✓ ¹⁴
Atomic variables	✓	✓ _J	✓	✓	✓
Transactional memory	✓	• ³	• ⁵	• ⁸	✓
<i>Message-passing models</i>					
Actors	• ¹	• ⁴	• ⁴	• ⁹	• ¹⁵
Channels	✓	✓	• ⁶	• ¹⁰	✓
Agents	✓				

✓	Built into the language or its standard library	• ⁷	Threading Building Blocks: https://www.threadingbuildingblocks.org
✓ _J	Part of Java's standard library, and are therefore available in Clojure and Scala too.	• ⁸	TinySTM: http://www.tmware.org/tinystm.html
• ₁	Available as a library (we list one, often several exist):	• ⁹	C++ Actor Framework: http://actor-framework.org
• ¹	Pulsar: http://docs.paralleluniverse.co/pulsar , supports actors and dataflow	• ¹⁰	Boost channels (Fiber library): https://www.boost.org
• ²	Ozma: https://github.com/sjrd/ozma	• ¹¹	future package for Haskell: https://hackage.haskell.org/package/future
• ³	ScalaSTM: https://nbronson.github.io/scala-stm	• ¹²	DPH: https://wiki.haskell.org/GHC/Data_Parallel_Haskell
• ⁴	Akka: https://akka.io , supports actors (Akka Actors) and dataflow concurrency (Akka Streams)	• ¹³	Etage: https://hackage.haskell.org/package/Etage
• ⁵	DeuceSTM: https://sites.google.com/site/deucestm	✓ ¹⁴	QSem semaphores.
• ⁶	JCSP: https://www.cs.kent.ac.uk/projects/ofa/jcsp	• ¹⁵	thespian: https://hackage.haskell.org/package/thespian

Fig. 17. Concurrency models supported by selected programming languages.

syntactically. In actor models with a receive statement, such a restriction would need to be enforced in a different manner, e.g., by prohibiting it using the type system.

- Choccola explores only one point in the design space of combinations of concurrency models, combining futures, transactions, and actors. In future research, it would be interesting to explore combinations with other concurrency models, such as Concurrent Revisions [15] (instead of transactions) or Communicating Sequential Processes [38] (instead of actors).

APPENDICES

A LANGUAGE AND LIBRARY SUPPORT FOR CONCURRENCY MODELS

Many programming languages have built-in support for multiple concurrency models. Moreover, when a concurrency model is not built into the language, developers often build libraries instead. These claims are supported by the table in Figure 17.

B HELPER FUNCTIONS AND FULL DEFINITIONS OF OPERATIONAL SEMANTICS

This Appendix contains some additional definitions for the formal operational semantics.

$$\begin{aligned}
\text{apply}|_b & \mathcal{E}[(\text{fn } [\bar{x}] e) \bar{v}] \rightarrow_b \mathcal{E}[[\bar{v}/\bar{x}] e] \\
\text{if-true}|_b & \mathcal{E}[\text{if true } e_1 e_2] \rightarrow_b \mathcal{E}[e_1] \\
\text{if-false}|_b & \mathcal{E}[\text{if false } e_1 e_2] \rightarrow_b \mathcal{E}[e_2] \\
\text{let}|_b & \mathcal{E}[\text{let } [x v] e] \rightarrow_b \mathcal{E}[[v/x] e] \\
\text{do}|_b & \mathcal{E}[\text{do } \bar{v}; e] \rightarrow_b \mathcal{E}[e]
\end{aligned}$$

Fig. 18. The reduction rules of L_b , the base language.

B.1 Reductions

A **reduction** of a program e is a sequence of program states $p_0 \rightarrow p_1 \rightarrow \dots \rightarrow p_n$, with the **initial state** p_0 as defined in Figure 9. A state p is **final** if it cannot be further reduced, i.e., $\nexists p' : p \rightarrow p'$. The notation $\rightarrow^?$ indicates a reduction in zero or one step; \rightarrow^* in zero, one or more steps.

B.2 Base Language

Figure 18 defines the reduction relation \rightarrow_b of the base language L_b . The syntax “[v/x] e ” denotes the expression e with all free occurrences of the variable x replaced by the value v . Note that the `do` construct is not useful in the functional base language; it becomes useful only when combined with the side effects introduced in the extensions of the language.

B.3 Syntax and Evaluation Contexts

The full syntax of PureChocola and its evaluation contexts are shown in Figure 19.

B.4 Helper Functions on the Program State

Figure 20 defines three helper functions that extract elements out of the program state:

- `actor-tasks(T, a)` returns all tasks in the actor with address a .
- `actor-txs(T, a)` returns the numbers of all transactions that are active (in a task) within actor a . (Tasks in which no transaction is active do not match the pattern and are thus ignored.)
- `tx-futs(T, n)` returns the futures of all tasks forked within the transaction with number n .

B.5 Operations to Merge Effects

Figure 21 defines four operations that are used to merge the effects of tasks.

- The operator `::` concatenates two maps and is right-preferential.
- $\overline{\text{beh}}_1^? \mid \overline{\text{beh}}_2^?$ (read “behavior 1 otherwise behavior 2”) combines two optional behaviors: It returns the first if it exists, otherwise the second.
- $\text{eff}_1 += \text{eff}_2$ merges effects on actors from two tasks: The sets of spawned actors are joined and the behaviors are combined (preferring the second over the first if both exist). This operation will occur when a task is joined into another.
- $\text{ctx}_1 += \text{ctx}_2$ merges the transactional context of a second (“child”) task into a first (“parent”) task, which occurs when a transactional task is joined into another. We define that:
 - The transaction numbers need to be the same: A transactional task can be merged only by another task in the *same* transaction.
 - The first task is performing the join, so it keeps its snapshot.
 - The local store of the second is added to the first, solving conflicts by preferring the version in the second task. In PureChocola, we do not consider custom conflict resolution functions (as in Section 5.3), but instead, we always use the default conflict resolution function that takes the value from the child task.

$c \in \text{Constant}$	$::= \text{nil} \mid \text{true} \mid \text{false} \mid 0 \mid 1 \mid \dots \mid "" \mid "a" \mid \dots$	
$x \in \text{Variable}$		
$f \in \text{Future}$		
$r \in \text{TVar}$		
$a \in \text{Address}$		
$b \in \text{BehaviorDef}$	$::= \text{behavior } [\bar{x}_{\text{beh}}] [\bar{x}_{\text{msg}}] e$	Behavior definition
$v \in \text{Value}$	$::= c$	
	$\mid x$	
	$\mid \text{fn } [\bar{x}] e$	Anonymous function
	$\mid f$	Future
	$\mid r$	Transactional variable
	$\mid a$	Address of actor
	$\mid b$	Behavior
$e \in \text{Expression}$	$::= v$	
	$\mid e \bar{e}$	Function application
	$\mid \text{if } e e e$	
	$\mid \text{let } [x e] e$	
	$\mid \text{do } \bar{e}; e$	
	$\mid \text{fork } e$	Fork a future
	$\mid \text{join } e$	Join a future
	$\mid \text{atomic } e$	Transaction
	$\mid \text{atomic}\star e$	In transaction (intermediate state)
	$\mid \text{ref } e$	Create a TVar
	$\mid \text{deref } e$	Read a TVar
	$\mid \text{ref-set } e e$	Write to a TVar
	$\mid \text{spawn } e \bar{e}$	Spawn an actor
	$\mid \text{become } e \bar{e}$	Become a behavior
	$\mid \text{send } e \bar{e}$	Send a message
\mathcal{P}	$::= A, T \cup \langle f, a, \mathcal{E}, F_c, F_j, \text{eff}, \text{ctx}^2 \rangle, \mu, \tau, \sigma$	
\mathcal{E}	$::= \square \mid (\bar{v} \mathcal{E} \bar{e}) \mid \text{if } \mathcal{E} e e \mid \text{let } [x \mathcal{E}] e \mid \text{do } \bar{v}; \mathcal{E} \mid \text{join } \mathcal{E} \mid \text{atomic}\star \mathcal{E}$	
	$\mid \text{ref } \mathcal{E} \mid \text{deref } \mathcal{E} \mid \text{ref-set } \mathcal{E} e \mid \text{ref-set } r \mathcal{E} \mid \text{spawn } \mathcal{E} \bar{e} \mid \text{spawn } b \bar{v} \mathcal{E} \bar{e}$	
	$\mid \text{become } \mathcal{E} \bar{e} \mid \text{become } b \bar{v} \mathcal{E} \bar{e} \mid \text{send } \mathcal{E} \bar{e} \mid \text{send } a \bar{v} \mathcal{E} \bar{e}$	

Fig. 19. The syntax and evaluation contexts of PureChocola.

$\text{actor-tasks}(T, a) = \{\text{task} \mid \text{task} = \langle f, a, e, F_c, F_j, \text{eff}, \text{ctx}^2 \rangle \in T\}$	(tasks in actor a)
$\text{actor-txs}(T, a) = \{n \mid \langle f, a, e, F_c, F_j, \text{eff}, \langle n, \bar{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle \in T\}$	(id's of tx's in actor a)
$\text{tx-futs}(T, n) = \{f \mid \langle f, a, e, F_c, F_j, \text{eff}, \langle n, \bar{\sigma}, \delta, \text{eff}_{\text{tx}} \rangle \rangle \in T\}$	(futures of tasks in tx n)

Fig. 20. Helper functions to extract elements out of the program state's set of tasks T .

- The effects on actors of the second are added to the first. In case of conflicting becomes, the one from the second task is preferred.

C PROOF OF INTRATRANSACTIONAL DETERMINACY

Before proving Intratransactional Determinacy, we first introduce some definitions and then establish three additional lemmas that prove local determinism, strong local confluence, and confluence.

$$\begin{aligned}
 (\tilde{\sigma} :: \delta)(r) &= \begin{cases} \delta(r) & \text{if } r \in \text{dom}(\delta) \\ \tilde{\sigma}(r) & \text{otherwise} \end{cases} \\
 \overline{\text{beh}}_1^? \mid \overline{\text{beh}}_2^? &= \begin{cases} \overline{\text{beh}}_1^? & \text{if } \overline{\text{beh}}_1^? \neq \bullet \\ \overline{\text{beh}}_2^? & \text{otherwise} \end{cases} \\
 \langle \mathbb{A}_1, \overline{\text{beh}}_1^? \rangle += \langle \mathbb{A}_2, \overline{\text{beh}}_2^? \rangle &= \langle \mathbb{A}_1 \cup \mathbb{A}_2, \overline{\text{beh}}_1^? \mid \overline{\text{beh}}_2^? \rangle \\
 \langle n, \tilde{\sigma}_1, \delta_1, \text{eff}_1 \rangle += \langle n, \tilde{\sigma}_2, \delta_2, \text{eff}_2 \rangle &= \langle n, \tilde{\sigma}_1, \delta_1 :: \delta_2, \text{eff}_1 += \text{eff}_2 \rangle
 \end{aligned}$$

Fig. 21. The operator $::$ combines transactional heaps, snapshots, or local stores; \mid combines optional behaviors; $+=$ merges effects on actors or transactional contexts.

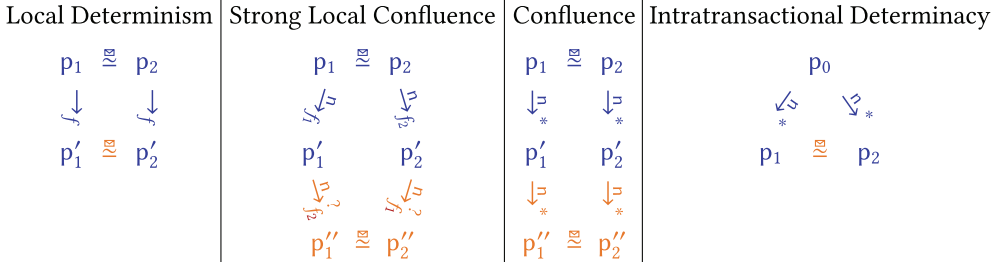


Fig. 22. A visualization of the four properties proven in Appendix C: with the states and relations in blue given, the states and relations in orange must hold.

Figure 22 visualizes these properties. This is based on the work of Burckhardt and Leijen [16] and Budimlić et al. [14].

Definition C.1 (Annotation of Task \rightarrow_f). Every rule in our semantics except $\text{receive}|_c$ and $\text{turn-end}|_c$ corresponds to the reduction of an expression in a single task. We can therefore annotate these reductions, using \rightarrow_f , to indicate that they correspond to a reduction in the task with identifier f .

Definition C.2 (Local and Non-local Reductions). A reduction is **local** if it accesses (reads or writes) only information stored in the task that is being reduced. **Non-local reductions** read from and/or write to shared data structures (A, μ, τ, σ) or other tasks (in T).

In our reduction rules, local reductions were written using the shorthand \mathcal{T} syntax. There are 11 non-local rules: $\text{fork}|_c, \text{join}_1|_c, \text{join}_2|_c, \text{atomic}|_c$, the three commit rules, $\text{spawn}|_c, \text{send}|_c, \text{receive}|_c$, and $\text{turn-end}|_c$.

LEMMA C.3 (LOCAL DETERMINISM). *If $p_1 \cong p_2$ and $p_1 \rightarrow_f p'_1$ and $p_2 \rightarrow_f p'_2$, then $p'_1 \cong p'_2$.*

Informally: Any reduction of the program state in a given task leads to equivalent results, as at each point there is only one possible step to take.

PROOF. The program evaluation context \mathcal{P} allows a reduction in any task in which one is possible, representing the non-determinism inherent to the execution of multithreaded programs. It contains at most one evaluation context \mathcal{E} corresponding to the task identified with the future f . Furthermore, by construction each evaluation context \mathcal{E} contains at most one hole. Thus, for a given future, a program state contains a single redex.

Given this redex, there is at most one rule that reduces it. First, rules match on the construct used in the redex. Second, if there are multiple rules for the same construct (which is the case for `join`, `atomic`, and `atomic★`), they have mutually exclusive conditions. In other words, for a given p_1 and f , at most one reduction rule applies, thereby uniquely defining p'_1 up to renaming.

As $p_1 \cong p_2$, both program states reduce using the same transition rule. Because each reduction rule relies solely on its initial state, besides the creation of identifiers, these rules lead to equivalent states $p'_1 \cong p'_2$. \square

Note: the same theorem and reasoning holds for the equivalence \cong .

LEMMA C.4 (STRONG LOCAL CONFLUENCE). *Given $p_1 \cong p_2$ and $p_1 \xrightarrow{n}_{f_1} p'_1$ and $p_2 \xrightarrow{n}_{f_2} p'_2$, there exist equivalent states $p''_1 \cong p''_2$ such that $p'_1 \xrightarrow{n}_{f_2} p''_1$ and $p'_2 \xrightarrow{n}_{f_1} p''_2$.*

Informally: We have a start state in a transaction and can either do a reduction in task 1 or task 2 (both part of the same transaction). We can find a reduction in task 2 to follow the one in task 1 and one in task 1 to follow the one in task 2, so both cases lead to an equivalent end result.

PROOF. In the case of $f_1 = f_2$, this follows directly from the lemma of local determinism. No transitions are necessary, i.e., $p''_1 = p'_1$ and $p''_2 = p'_2$. We consider the case $f_1 \neq f_2$.

We distinguish different cases based on the two rules triggered in the reductions. For each combination of a rule triggered in task 1 and one triggered in task 2, we demonstrate that reducing task 1 followed by task 2 leads to end states equivalent to reducing them in the opposite order. This is shown in Table 3.

Some rules are grouped or ignored:

- We group all local transition rules (as these do not access shared state).
- The rules `atomicc`, `receivec`, and `turn-endc` apply only when no transaction is running, while the lemma applies only to reductions in transactions (indicated with \xrightarrow{n}), so we do not need to consider them here.
- The commit rules trigger only when a transaction is fully reduced, in which case p_1 and p_2 are states *ready to commit*, and no subsequent reductions within the transaction (\xrightarrow{n}) are possible. Hence, they also do not need to be considered.
- Finally, the cases below the diagonal of the table are equivalent to those above the diagonal with f_1 and f_2 swapped.

Table 3 considers all remaining cases and states why triggering the rule in the column followed by the one in the row (i.e., reducing task 1 and then task 2) leads to equivalent end states as reducing them in the opposite order, thus proving the theorem. \square

Note that this proof relies on the equivalence \cong , indicating that p''_1 and p''_2 are equivalent up to the ordering of messages only. This exception is needed in the case both tasks trigger the rule `sendc`.

LEMMA C.5 (CONFLUENCE). *Given $p_1 \cong p_2$ and $p_1 \xrightarrow{n}^* p'_1$ and $p_2 \xrightarrow{n}^* p'_2$, there exist equivalent states $p''_1 \cong p''_2$ such that $p'_1 \xrightarrow{n}^* p''_1$ and $p'_2 \xrightarrow{n}^* p''_2$.*

Informally: In a transaction, we can interleave reductions from different tasks and they can all lead to the same end result. We can prove this by repeatedly applying the previous lemma.

PROOF. The standard technique of Huet [40] to prove confluence from strong local confluence applies here. \square

THEOREM C.6 (INTRATRANSACTIONAL DETERMINACY). *Given $p_0 = \mathcal{P}[\text{atomic★ } e]$, if $p_0 \xrightarrow{n}^* p_1$ and $p_0 \xrightarrow{n}^* p_2$, with p_1 and p_2 states ready to commit transaction n , then $p_1 \cong p_2$.*

Table 3. Combinations of Rules

$\downarrow 2 \rightarrow 1$	local	$join_{1 2 c}$	$fork _c$	$send _c$	$spawn _c$
Accessed shared state	None	Joined task is read	Forked task is added	Message is added to receiver's inbox	Inbox is created
local	The rules affect independent parts of the state and thus commute.	As it is impossible to join a task that has not been fully reduced to a value, join in task 1 cannot be joining task 2, in which a further reduction is possible. Both reductions are therefore independent.	The reduction in task 2 is local.	The reduction in task 2 is local.	spawn creates an inbox with a unique identifier, thereby not affecting other tasks.
$join_{1 2 c}$	(commutative)		join in task 1 cannot join the new task created by fork in task 2, as it does not have access to its future. Hence, both rules must affect different tasks.	They affect independent parts of the state: send writes to an inbox; fork and join read and write to different tasks.	
$fork _c$	(commutative)	(commutative)	Both tasks independently create a task (each with a unique future).		
$send _c$	(commutative)	(commutative)	(commutative)	(!) Both rules append a message to an inbox. If they send to the same actor, the order in which messages are added may differ. This is allowed by the equivalence \cong .	
$spawn _c$	(commutative)	(commutative)	(commutative)	(commutative)	

We show that first reducing task 1 using the rule in the column, followed by reducing task 2 using the rule in the row, leads to equivalent results as reducing them in the opposite order (except for the order in which messages are sent). We also list which shared state is accessed in each transition.

Informally: A transaction starts and then evaluates until the point just before it commits. Any state it can reach is equivalent, except for the order in which messages are sent. In other words, different evaluations might take different steps during the reduction of the transaction, but eventually they will all lead to equivalent end results just before the commit.

PROOF. Given $p_0 \xrightarrow{n}^* p_1$ and $p_0 \xrightarrow{n}^* p_2$, the confluence theorem states that there exist equivalent states $p'_1 \cong p'_2$ such that $p_1 \xrightarrow{n}^* p'_1$ and $p_2 \xrightarrow{n}^* p'_2$. However, as p_1 and p_2 are states ready to commit transaction n , no further intratransactional reductions within the transaction n are possible, so it must be that $p_1 = p'_1$ and $p_2 = p'_2$, and thus $p_1 \cong p_2$ as claimed. \square

Note that $\mathcal{P}[\text{atomic}\star e]$ must not necessarily be the start of the transaction: It can be any intermediate state during the reduction of the transaction.

D PROOF OF SNAPSHOT ISOLATION

Like Clojure, Chocla provides snapshot isolation. As mentioned in Section 2.2.4, Berenson et al. [9] define snapshot isolation as the absence of five anomalies. These anomalies are defined as patterns that may not appear in the program's transactional history in Table 4, based on the definitions of Berenson et al. [9].

Definition D.1 (Transactional History Pattern). A **transactional history pattern** is a pattern that describes a set of transactional histories. It is a sequence of transactional operations in which:

- x and y refer to any two different variables,
- v_i and v_j refer to any two different values (for $i \neq j$),
- 1 and 2 refer to any two different transaction attempts,
- the construct “...” elides parts of the history, and
- the construct “or” indicates a choice between operations.

A transactional history **matches** a pattern if any part of the history is described by the pattern.

Table 4. Five Anomalies and Their Corresponding Transactional History Pattern

Anomaly	Transactional history pattern
Dirty read	$x \leftarrow_1 v_1 \dots x \rightarrow_2 v_1 \dots \surd_1$ or \times_1
Dirty write	$x \leftarrow_1 v_1 \dots x \rightarrow_2 v_1 \dots x \leftarrow_2 v_2 \dots \surd_1$ or \times_1
Non-repeatable read	$x \rightarrow_1 v_1 \dots x \leftarrow_2 v_2 \dots x \rightarrow_1 v_2$
Lost update	$x \leftarrow_1 v_1 \dots x \leftarrow_2 v_2 \dots \surd_2 \dots \surd_1$
Read skew	$x \rightarrow_1 v_{x,1} \dots x \leftarrow_2 v_{x,2} \dots y \leftarrow_2 v_{y,2} \dots \surd_2 \dots y \rightarrow_1 v_{y,2}$

THEOREM D.2 (SNAPSHOT ISOLATION). L_t provides snapshot isolation, i.e., the transactional history of any program reduction does not contain dirty reads, dirty writes, non-repeatable reads, lost updates, or read skew.

We prove this theorem by proving each constituent property separately.

LEMMA D.3 (NO DIRTY READS). Program reductions contain no dirty reads: Their history cannot match the pattern $x \leftarrow_1 v_1 \dots x \rightarrow_2 v_1 \dots (\surd_1$ or $\times_1)$.

PROOF. The write operation of v_1 in transaction 1 is stored in its local store (rule $\text{ref-set}|_c$), which is made visible to other transactions only when transaction 1 commits successfully (rule $\text{commit}|_c$). When transaction 2 starts, it copies the current transactional heap into a snapshot

(rule $\text{atomic}|_c$), which contains the previous value of x and not yet v_1 . The read operation $x \rightarrow_2 v_1$ is therefore impossible. \square

LEMMA D.4 (NO DIRTY WRITES). *Program reductions contain no dirty writes: Their history cannot match the pattern $x \leftarrow_1 v_1 \dots x \rightarrow_2 v_1 \dots x \leftarrow_2 v_2 \dots (\checkmark_1 \text{ or } \times_1)$.*

PROOF. The write in transaction 1 is stored in its local store (rule $\text{ref-set}|_c$) and is therefore not visible to transaction 2 before transaction 1 committed. The read operation $x \rightarrow_2 v_1$ is therefore impossible. \square

LEMMA D.5 (NO NON-REPEATABLE READS). *Program reductions contain no non-repeatable reads: Their history cannot match the pattern $x \rightarrow_1 v_1 \dots x \leftarrow_2 v_2 \dots x \rightarrow_1 v_2$.*

PROOF. When transaction 1 starts, it copies the transactional heap into a snapshot (rule $\text{atomic}|_c$). All read operations in the transaction look up values in the transaction's local store and this snapshot (rule $\text{deref}|_c$). Hence, the value v_2 is not visible in transaction 1. \square

LEMMA D.6 (NO LOST UPDATES). *Program reductions contain no lost updates: Their history cannot match the pattern $x \leftarrow_1 v_1 \dots x \leftarrow_2 v_2 \dots \checkmark_2 \dots \checkmark_1$.*

PROOF. When transaction 1 starts, it copies the transactional heap into a snapshot (rule $\text{atomic}|_c$), containing the original value of variable x . When transaction 2 commits, it stores the value v_2 in the heap. When transaction 1 attempts to commit afterwards, it encounters a conflict: The value v_2 in the heap no longer matches the value in transaction 1's snapshot. The rule $\text{commit}_\times|_c$ therefore applies and transaction 1 aborts. \square

LEMMA D.7 (NO READ SKEW). *Read skew is prevented: The history of any program reduction cannot match the pattern $x \rightarrow_1 v_{x,1} \dots x \leftarrow_2 v_{x,2} \dots y \leftarrow_2 v_{y,2} \dots \checkmark_2 \dots y \rightarrow_1 v_{y,2}$.*

PROOF. When transaction 1 starts, it copies the transactional heap into a snapshot (rule $\text{atomic}|_c$), in which variables x and y have the original values $v_{x,1}$ and $v_{y,1}$, respectively. Even after transaction 2 commits, this snapshot remains unmodified. When transaction 1 reads y (final operation in the pattern), it looks up its value in its snapshot (rule $\text{deref}|_c$), which returns $v_{y,1}$ and not $v_{y,2}$, preventing read skew. \square

PROOF OF SNAPSHOT ISOLATION. As each constituent property holds, snapshot isolation holds, too. \square

E PROOF OF CONSISTENT TURN PRINCIPLE

THEOREM E.1 (CONSISTENT TURN PRINCIPLE). *PureChocola provides the consistent turn principle, i.e., it guarantees continuous message processing, consecutive message processing, and isolation of actor memory.*

We prove this theorem by proving each constituent property separately.

LEMMA E.2 (CONTINUOUS MESSAGE PROCESSING). *An actor's turn is free from deadlocks.*

PROOF. See Deadlock Freedom of Transactional Actors in Section F.2. \square

LEMMA E.3 (CONSECUTIVE MESSAGE PROCESSING). *For each actor, at each step in the reduction at most one turn is active.*

PROOF. A turn corresponds to the processing of a message, which starts when the rule $\text{receive}|_c$ is triggered. We will prove that at most one turn is active per actor. We do this in two steps: First, we show that each turn has fully ended when the rule $\text{turn-end}|_c$ triggers (so no tasks “escape” the turn in which they were forked); second, we show no new turn starts during the reduction of a previous turn.

For the first step, observe that the rule $\text{turn-end}|_c$ applies only when the root task has been fully reduced to a value (v in definition of $\text{task}_{\text{root}}$) and all futures forked in the root task have been joined (condition $F_c \subseteq F_j$). Furthermore, these futures could have been joined only if they joined their respective children (condition $F_c^* \subseteq F_j^*$ in rule join). This applies recursively, ensuring that all tasks created in the current actor during the current turn have been joined. A task can be joined only if it has been reduced to a value, after which no further reductions of that task are possible. Hence, when the rule $\text{turn-end}|_c$ triggers, no further reductions are possible in any task created in the turn. (This proves the absence of the problem of tasks “escaping” the turn they were forked in, described in Section 7.)

For the second step, observe that the rule $\text{receive}|_c$ requires an actor to be in an idle state, by requiring its root task ($f_{\text{root}}^?$) to be \bullet . There are only two rules that cause an actor to be idle: $\text{spawn}|_c$ and $\text{turn-end}|_c$. The rule $\text{receive}|_c$ is thus never applicable while another turn is active, meaning that it is impossible for a new turn to start while another is active. \square

LEMMA E.4 (ISOLATION OF ACTOR MEMORY). *The internal memory of an actor is isolated: After the actor’s creation, its internal memory can be read and written only by the actor itself.*

PROOF. The internal memory of an actor consists of the values \bar{v} stored in its behavior beh . It is accessed in the following transition rules:

- The rule $\text{receive}|_c$ reads the internal memory, replacing all corresponding variables in the turn’s body with these values in a single step (function bind). The actor thus reads its own memory.
- In the rule $\text{turn-end}|_c$, the effect of become operations that occurred during the turn are stored in the internal memory. (Note that any become during a turn is first stored locally in the task or transaction, and only effectively stored in the actor at the end of the turn.) Again, the actor is modifying its own memory.
- In the rule $\text{spawn}|_c$, the internal memory is initialized as the actor is created. As indicated in the theorem, we discount operations that take place during the actor’s creation.

Hence, after an actor is created, its internal memory is read and modified only by the actor itself. \square

This lemma can be generalized: Not only the internal memory of an actor is isolated, but all of its behavior beh , so also its behavior definition (its code).

PROOF OF THE CONSISTENT TURN PRINCIPLE. As each constituent property holds, the consistent turn principle holds, too. \square

F PROOF OF DEADLOCK FREEDOM

In this Appendix, we prove the deadlock freedom of futures and transactional actors, as defined in Section 9.3.4.

F.1 Deadlock Freedom of Futures

A task that encounters join waits until the given future has resolved, so we must ensure this can never lead to a deadlock. In this section, we look at individual turns and prove that there is a total

order on the futures created in a turn, thereby preventing deadlocks. We assume the program does not exchange futures in transactional memory. We first introduce a few definitions.

Definition F.1 (Deadlock). A program state $\rho = A, T, \mu, \tau, \sigma$ is in **deadlock** if there exists a subset of “deadlocked” futures $F_{dl} \subset \text{Future}$ and a permutation $\pi : F_{dl} \rightarrow F_{dl}$ representing the dependencies between them, such that $\forall f_i \in F_{dl} : \langle f_i, a_i, \mathcal{E}[\text{join } \pi(f_i)], F_{c,i}, F_{j,i}, \text{eff}_i, \text{ctx}_i^? \rangle \in T$ (i.e., each task is joining the next future in the permutation).

Note that the existence of such a permutation implies a cycle in the dependencies between futures. If there is no cycle in the dependencies between futures, no such permutation π exists and therefore no deadlock can occur.

Note also that $\pi \neq \text{id}$, because this implies each future is joining itself, but a task does not have access to its own future. (This is in fact also proven by the lemma below.)

A task can join only futures it can reach. We define the notion of reachable futures:

Definition F.2 (Reachable Futures \tilde{F}_i). Given a program state $\rho = A, T, \mu, \tau, \sigma$, we say the set of **reachable futures** \tilde{F}_i is the set of futures to which $\text{task}_i \in T$ can refer.

This definition does not say *which* futures a task can reach: It is not immediately obvious how, given a program state, the set \tilde{F}_i can be constructed. We first introduce two more definitions.

Definition F.3 (Root Task and Fork Tree). Each turn starts with the creation of one task, in the rule receive_c , which we refer to as the **root task** of that turn. (The future of this root task is stored in the actor for the duration of the turn.) Each task can fork child tasks. Hence, all tasks forked during a turn can be represented as a tree, which we refer to as the turn’s **fork tree**.²⁰

Definition F.4 (Post-order Notation). Writing a tree in **post-order notation** consists of performing a depth-first traversal of the tree and writing down a node after having visited its children. The post-order notation of a tree thus defines a unique order of the nodes in the tree so: (1) a node appears after all of its children, (2) later siblings appear after earlier siblings.

The following lemma states how \tilde{F}_i can be constructed: Within a turn, a task can reach the futures of tasks from the same turn that precede it in the post-order notation of the turn’s fork tree.

LEMMA F.5. *If a program does not read futures from transactional memory, then task_i ’s set of reachable futures \tilde{F}_i consists of (at most) the futures of the tasks that precede task_i in the post-order notation of the fork tree of the turn in which task_i was forked.*

PROOF. By considering the rules in our language, we observe that the following rules modify the set \tilde{F}_i (by making futures from other tasks reachable in task_i):

- (1) fork_c : A newly forked task is reachable by its parent, i.e., when task_i forks a new future f_j , its set of reachable futures is extended with f_j . Hence $\tilde{F}_i' = \tilde{F}_i \cup \{f_j\}$. The set of reachable futures \tilde{F}_j of the new task j is a copy of the reachable futures \tilde{F}_i of its parent before it was extended. Hence, $\tilde{F}_j' = \tilde{F}_i$. This implies the new task has access to all previous children of its parent, i.e., its earlier siblings. The same is true recursively up the fork tree: It has access to the earlier siblings of all its ancestors.

²⁰Blumofe et al. [11] refer to this as a spawn tree; Lee and Palsberg [43] call it an execution tree.

- (2) $\text{join}_1|_c$ and $\text{join}_2|_c$: When task_{*i*} joins the future f_j , its set of reachable futures is extended with \vec{F}_j , the set from the joined task. This is because the task can return any future to which it had access. Hence: $\vec{F}'_i = \vec{F}_i \cup \vec{F}_j$.
- (3) $\text{receive}|_c$: As we consider only the fork tree formed within a turn in this lemma, no receive rules are triggered for the current actor for the reduction of the current turn.
- (4) $\text{atomic}|_c$: The set \vec{F}_i can be extended by reading futures from the transactional memory. However, we explicitly assume this does not occur for this lemma.

We now prove that these observations result in the set of reachable futures as stated in the lemma. The fork rule (1) implies that the set \vec{F}_i contains at least all children, earlier siblings, and earlier siblings of ancestors of task_{*i*}. The join rule (2) implies that, as task_{*i*} can join any of its reachable futures, it can reach any of those futures' reachable futures, i.e., \vec{F}_i can be extended with any \vec{F}_j for which $f_j \in \vec{F}_i$. This applies transitively, allowing \vec{F}_i to be extended with its transitive closure.

This all corresponds exactly to the order indicated in the post-order notation of the fork tree, which is the unique order of tasks within the turn such that (1) a task appears after all of its children, (2) siblings appear in the order they are forked. \square

THEOREM F.6 (DEADLOCK FREEDOM OF FUTURES). *Assuming no futures are stored in transactional memory, the tasks created within a turn cannot deadlock. A set of tasks is deadlocked if each task in the set is waiting for the future of another task created in the same turn to be resolved.*

PROOF. A task can only wait for a future to be resolved if it can reach the future. A task has access only to futures of tasks that precede it in the post-order notation of the fork tree. This defines a total order on the futures within a turn, therefore there cannot be a cyclical dependency between tasks and thus deadlocks cannot occur. \square

F.2 Deadlock Freedom of Transactional Actors

Remember that messages sent in a transaction are tentative, carrying a dependency n_{dep} on a transaction attempt. (A dependency does not point to a transaction, but to an individual attempt of a transaction.) The turn that processes such a message is a tentative turn.

In a tentative turn, the rules $\text{commit}_\surd|_c$, $\text{commit}_\cdot|_c$, and $\text{turn-end}|_c$ wait until the dependency has finished its reduction and has either committed or aborted. We first discuss the conditions in which these rules block and then prove that deadlocks are impossible, because cyclical dependencies are impossible.

$\text{turn-end}|_c$. An actor can block at the end of a tentative turn, waiting for its dependency to be resolved. This is encoded in the rule $\text{turn-end}|_c$ by checking whether the dependency has succeeded or failed ($\tau(n_{\text{dep}}^?) = \langle \surd, \vec{e} \rangle$ or $\langle \times, \vec{e} \rangle$). While the transaction is still running ($\tau(n_{\text{dep}}^?) = \langle \triangleright, \vec{e} \rangle$), none of both conditions is satisfied, so this rule is not applicable, nor is any other rule.

When the rule $\text{turn-end}|_c$ is triggered, the turn's root task has been fully reduced to a value, and therefore no transaction is active (the root task's transactional context is \bullet). Hence, cyclical dependencies are impossible: The reduction of the current task depends on the reduction of a transaction, but vice versa the reduction of that transaction can not depend on the reduction of the current task, as there is no transaction in the current task.

$\text{commit}_\surd|_c$ and $\text{commit}_\cdot|_c$. After a transaction has finished its reduction, one of three commit rules applies:

- the rule $\text{commit}_\times|_c$ applies if the transaction conflicts with another;

- the rule $\text{commit}_{|c}$ applies if the transaction does not conflict, and has either no dependency or the dependency has succeeded;
- the rule $\text{commit}_{|c}$ applies if the dependency aborted.

If there is a dependency but it is still running ($\tau(n_{\text{dep}}) = \langle \triangleright, \tilde{\tau} \rangle$), none of the three rules apply and therefore the current task waits until the dependency has either committed or aborted.

Below, we prove that cyclical dependencies are impossible, as the current transaction attempt (with number n in the rule) is always newer than the dependency (n_{dep}).

At the start of every transaction attempt, in the rule $\text{atomic}_{|c}$, a fresh transaction number n is generated. We suppose that this process generates strictly increasing numbers.²¹ Newer transaction attempts thus have higher numbers. We will show that dependencies always point from higher (newer) to lower (older) transaction numbers.

THEOREM F.7 (DEADLOCK FREEDOM OF TRANSACTIONAL ACTORS). *Deadlocks between transactions in tentative turns are impossible, as dependencies always go from newer to older transaction attempts, defining a total order on the transaction attempts.*

PROOF. First, observe that a transaction starts and commits within the same turn: When a transaction attempt is started (rule $\text{atomic}_{|c}$) it inserts the expression $\text{atomic} \star e$ in the current evaluation context, in the following reduction steps e is reduced, after which the transaction commits (using one of the commit rules).

Second, observe that a dependency is always introduced at the start of a turn: In the rule $\text{receive}_{|c}$, a potential dependency $n_{\text{dep}}^?$ from the received message is stored in the current actor. It is removed at the end of the turn (rule $\text{turn-end}_{|c}$).

Hence, when a transaction commits in a tentative turn, the number n of the current transaction was created in the current turn, while the number n_{dep} of the dependency already existed before the turn started. The dependency must therefore be older than the current transaction. \square

REFERENCES

- [1] Gul A. Agha. 1985. *Actors: A Model of Concurrent Computation in Distributed Systems*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [2] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. A foundation for actor computation. *J. Funct. Prog.* 7, 1 (1997), 1–72.
- [3] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. 2008. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. 163–174. DOI: <https://doi.org/10.1145/1345206.1345232>
- [4] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. 253–262. DOI: <https://doi.org/10.1145/1810479.1810528>
- [5] Henry C. Baker and Carl Hewitt. 1977. The incremental garbage collection of processes. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*. 55–59. DOI: <https://doi.org/10.1145/800228.806932>
- [6] João Barreto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. 2010. Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. 10. DOI: <https://doi.org/10.1145/1693453.1693466>
- [7] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proc. ACM Prog. Lang.* 1, OOPSLA (Oct. 2017).
- [8] Catriel Beerli, Philip A. Bernstein, and Nathan Goodman. 1989. A model for concurrency in nested transactions systems. *J. ACM* 36, 2 (Apr. 1989), 230–269. DOI: <https://doi.org/10.1145/62044.62046>

²¹Something similar happens in the implementation of the MVCC algorithm: At the start of each transaction attempt, the value of a logical clock is stored.

- [9] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD ’95)*. 1–10. DOI : <https://doi.org/10.1145/223784.223785>
- [10] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *Comput. Surv.* 13, 2 (June 1981), 185–221. DOI : <https://doi.org/10.1145/356842.356846>
- [11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP’95)*. 207–216. DOI : <https://doi.org/10.1145/209936.209958>
- [12] Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel programming must be deterministic by default. In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar’09)*.
- [13] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel objects for multicores: A glimpse at the parallel language encore. In *Proceedings of the 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM’15)*. 1–56.
- [14] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Saĝnak Taşlılar. 2010. Concurrent collections. *Sci. Prog.* 18, 3–4 (Aug. 2010), 203–217. DOI : <https://doi.org/10.1155/2010/521797>
- [15] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA’10)*. 691–707. DOI : <https://doi.org/10.1145/1869459.1869515>
- [16] Sebastian Burckhardt and Daan Leijen. 2011. Semantics of concurrent revisions. In *Proceedings of the European Symposium on Programming (ESOP’11)*. 116–135.
- [17] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE’15)*. 1–12. DOI : <https://doi.org/10.1145/2824815.2824816>
- [18] Joeri De Koster. 2015. *Domains: Language Abstractions for Controlling Shared Mutable State in Actor Systems*. Ph.D. Dissertation. Vrije Universiteit Brussel.
- [19] J. De Koster, S. Marr, T. Van Cutsem, and T. D’Hondt. 2016. Domains: Sharing state in the communicating event-loop actor model. *Comput. Lang., Syst. Struct.* 45 (2016), 132–160. DOI : <https://doi.org/10.1016/j.cl.2016.01.003>
- [20] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: A taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE’16)*. 31–40. DOI : <https://doi.org/10.1145/3001886.3001890>
- [21] Peter J. Denning and Jack B. Dennis. 2010. The resurgence of parallelism. *Commun. ACM* 53, 6 (June 2010), 30–32.
- [22] E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (Sept. 1965), 569. DOI : <https://doi.org/10.1145/365559.365617>
- [23] Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent bug patterns and how to test them. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS’03)*. DOI : <https://doi.org/10.1109/IPDPS.2003.1213511>
- [24] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. The MIT Press.
- [25] Cormac Flanagan and Matthias Felleisen. 1995. The semantics of future and its use in program optimization. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’95)*. 209–220.
- [26] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA’07)*. 57–76.
- [27] Patrice Godefroid and Nachi Nagappan. 2008. *Concurrency at Microsoft—An Exploratory Survey*. Technical Report. Retrieved from: <https://www.microsoft.com/en-us/research/publication/concurrency-at-microsoft-an-exploratory-survey/>.
- [28] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP’08)*. 175–184. DOI : <https://doi.org/10.1145/1345206.1345233>
- [29] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. 1994. Composing first-class transactions. *ACM Trans. Prog. Lang. Syst.* 16, 6 (Nov. 1994), 1719–1736. DOI : <https://doi.org/10.1145/197320.197346>
- [30] Stuart Halloway. 2009. *Programming Clojure* (1st ed.). Pragmatic Bookshelf.
- [31] Robert H. Halstead. 1985. MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. DOI : <https://doi.org/10.1145/4472.4478>

- [32] Tim Harris and Keir Fraser. 2003. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. 388–402. DOI : <https://doi.org/10.1145/949305.949340>
- [33] Tim Harris, James R. Larus, and Ravi Rajwar. 2010. *Transactional Memory* (2nd ed.). Morgan & Claypool.
- [34] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. 48–60. DOI : <https://doi.org/10.1145/1065944.1065952>
- [35] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA'93)*. 289–300.
- [36] Maurice Herlihy and Nir Shavit. 2011. *The Art of Multiprocessor Programming*. Morgan Kaufmann.
- [37] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*. 235–245. Retrieved from: <http://dl.acm.org/citation.cfm?id=1624775.1624804>.
- [38] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. DOI : <https://doi.org/10.1145/359576.359585>
- [39] David Hovemeyer and William Pugh. 2004. Finding concurrency bugs in Java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*. Retrieved from: <https://www.cs.jhu.edu/~daveho/pubs/csfp2004.pdf>.
- [40] Gérard Huet. 1980. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM* 27, 4 (Oct. 1980), 797–821. DOI : <https://doi.org/10.1145/322217.322230>
- [41] Shams M. Imam and Vivek Sarkar. 2012. Integrating task parallelism with actors. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA'12)*. 753–772.
- [42] Edward A. Lee. 2006. The problem with threads. *Computer* 39, 5 (May 2006), 33–42. DOI : <https://doi.org/10.1109/MC.2006.180>
- [43] Jonathan K. Lee and Jens Palsberg. 2010. Featherweight X10: A core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. 25–36. DOI : <https://doi.org/10.1145/1693453.1693459>
- [44] M. Lesani and A. Lain. 2013. Semantics-preserving sharing actors. In *Proceedings of the Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE'13)*. 69–80.
- [45] M. Lesani and J. Palsberg. 2011. Communicating memory transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. 157–168.
- [46] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes—A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. 329–339. DOI : <https://doi.org/10.1145/1346281.1346323>
- [47] V. Luchangco and V. J. Marathe. 2011. Transaction communicators: Enabling cooperation among concurrent transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. 169–178.
- [48] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 35–46. DOI : <https://doi.org/10.1109/IISWC.2008.4636089>
- [49] B. Morandi, S. Nanz, and B. Meyer. 2014. Safe and efficient data sharing for message-passing concurrency. In *Proceedings of the 16th International Conference on Coordination Models and Languages (COORDINATION'14)*. 99–114.
- [50] J. Eliot B. Moss. 1981. *Nested Transactions: An Approach to Reliable Distributed Computing*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [51] J. Eliot B. Moss and Antony L. Hosking. 2006. Nested transactional memory: Model and architecture sketches. *Sci. Comput. Prog.* 63, 2 (2006), 186–201.
- [52] Michael Nash and Wade Waldron. 2016. *Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications* (1st ed.). O'Reilly Media, Inc.
- [53] Armand Navabi and Suresh Jagannathan. 2009. Exceptionally safe futures. In *Proceedings of the 11th International Conference on Coordination Models and Languages (COORDINATION'09)*. 47–65.
- [54] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distrib. Comput.* 10, 2 (Feb. 1997), 99–116. DOI : <https://doi.org/10.1007/s004460050028>
- [55] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. 2007. Transactions with isolation and cooperation. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA'07)*. 191–210.

- [56] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2016. Transactional tasks: Parallelism in software transactions. In *Proceedings of the 30th European Conference on Object-oriented Programming (ECOOP'16)*. 23:1–23:28. DOI : <https://doi.org/10.4230/LIPIcs.ECOOP.2016.23>
- [57] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2017. Transactional actors: Communication in transactions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems (SEPS'17)*. 31–41. DOI : <https://doi.org/10.1145/3141865.3141866>
- [58] Janwillem Swalens, Stefan Marr, Joeri De Koster, and Tom Van Cutsem. 2014. Towards composable concurrency abstractions. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES'14)*. DOI : <https://doi.org/10.4204/EPTCS.155.8>
- [59] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall Press.
- [60] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why do Scala developers mix the actor model with other concurrency models? In *Proceedings of the 27th European Conference on Object-oriented Programming (ECOOP'13)*. 302–326. DOI : https://doi.org/10.1007/978-3-642-39038-8_13
- [61] Peter Van Roy and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press.
- [62] Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. 2004. A semantic framework for designer transactions. In *Proceedings of the 13th European Symposium on Programming (ESOP'04)*. 249–263.
- [63] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. 2009. NePalTM: Design and implementation of nested parallelism for transactional memory systems. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. 291–292. DOI : <https://doi.org/10.1145/1504176.1504220>
- [64] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. 2011. Worlds: Controlling the scope of side effects. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. 179–203.
- [65] Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe futures for Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. 439–453. DOI : <https://doi.org/10.1145/1094811.1094845>
- [66] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. 1986. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA'86)*. 258–268. DOI : <https://doi.org/10.1145/28697.28722>
- [67] Jingna Zeng, Joao Barreto, Seif Haridi, Luís Rodrigues, and Paolo Romano. 2016. The future(s) of transactional memory. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP'16)*. 442–451. DOI : <https://doi.org/10.1109/ICPP.2016.57>
- [68] Yang Zhang and Eric A. Hansen. 2006. Parallel breadth-first heuristic search on a shared-memory architecture. In *Proceedings of the Workshop on Heuristic Search, Memory-based Heuristics and Their Applications (AAAI'06)*.

Received January 2020; revised July 2020; accepted September 2020