Concurrency-aware object-oriented programming with roles

Concurrency-Aware Object-Oriented Programming with Roles

MICHAEL FAES, ETH Zurich, Switzerland
THOMAS R. GROSS, ETH Zurich, Switzerland

Object-oriented Programming has been effective in reducing code complexity in sequential programs, but in current practice, *concurrent* programs still present a number of challenges. We present here a model of object-oriented programming that identifies concurrent tasks and the *relationship between objects and tasks*, effectively making objects *concurrency-aware*. This awareness is formalized in a parallel programming model where every object *plays a role* in every task (e.g., the READONLY role). When an object is shared with a new task, it adapts to the new sharing pattern by changing its roles, and therefore its behavior, i.e., the operations that can be performed with this object. This mechanism can be leveraged to prevent interfering accesses from concurrently executing tasks, and therefore makes parallel execution deterministic.

To this end, we present a role-based programming language that includes several novel concepts (role transitions, guarding, slicing) to enable practical, object-oriented deterministic parallel programming. We show that this language can be used to *safely* implement programs with a range of different parallel patterns. The implementations to 8 widely used programming problems achieve substantial parallel speedups and demonstrate that this approach delivers performance roughly on par with manually synchronized implementations.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; **Object oriented languages**; **Concurrent programming languages**; Concurrent programming structures; • **Theory of computation →** *Parallel computing models*;

Additional Key Words and Phrases: programming models, concurrency, parallelism, determinism

ACM Reference Format:
Michael Faes and Thomas R. Gross. 2018. Concurrency-Aware Object-Oriented Programming with Roles. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 130 (November 2018), 30 pages. https://doi.org/10.1145/3276500

1 INTRODUCTION

The object-oriented programming paradigm has proven to be effective in taming complexity in software systems. By encapsulating related *data* and *code* into objects, object-oriented programming (OOP) enables programmers to reason about complex software systems in a modular way. For example, such encapsulation makes it simple to express and maintain *invariants* about the state of an object, because direct access to that state is restricted to that object's methods. Modular reasoning in turn leads to higher maintainability and lower risk of introducing software bugs due to insufficient code understanding.

However, many advantages of OOP are severely limited in a concurrent setting. When there are multiple concurrently executing threads in a system, it becomes difficult to maintain object invariants, because the assumption that there is exactly one executing method at a time is violated.

Authors' addresses: Michael Faes, Department of Computer Science, ETH Zurich, Switzerland, michael.faes@inf.ethz.ch; Thomas R. Gross, Department of Computer Science, ETH Zurich, Switzerland, trg@inf.ethz.ch.

2475-1421/2018/11-ART130
https://doi.org/10.1145/3276500

Simple approaches to mitigate this problem, such as manual locking, have shown to be deadlock-prone, difficult to get right, and impossible to reason about in a modular way, as explained by Lee [2006]. Lee illustrates that simple and commonly used OOP patterns, such as the Observer Pattern, are difficult to implement correctly in a multi-threaded setting and require intricate thinking about possible interleavings. Given that most of today's computing devices rely on multi-core CPUs to achieve high performance, it is clear that we need techniques that bring the advantages of OOP to the concurrent world.

Many of the current OOP practices are questionable for concurrent programs, especially for parallel programming (PP), where concurrency is not needed *per se* but is used solely as a means to increase performance through parallel execution. Instead, we describe how to adapt the object-oriented paradigm to the concurrent setting by *making objects aware of how they are used concurrently*, that is, how they are shared among threads. This approach stands in contrast to most mainstream OOP languages, where an object can be shared with another thread simply by passing it a reference (or "pointer") to that object, or even by passing it a reference to another object that has a reference to that object, or so forth. Passing references makes sharing objects among tasks straightforward and efficient, but it comes with a significant disadvantage: the information about which threads have access to which objects is only implicit and can be scattered around many objects and pieces of code. This violates the basic OO principle of encapsulation.

The approach presented here also differs from other concurrent OOP models such as the Actor Model [Agha 1986, 1990; Hewitt et al. 1973] and the Active Object Model [Yonezawa et al. 1986], where objects are not just *aware* of concurrency, but are concurrent entities themselves.

The building blocks for the presented approach to OOP are *objects*, *tasks* (which are like threads), and the *explicit relationship between objects and tasks*. More concretely, we present a programming model where every object *plays a role* in every task and *may change the roles it plays* whenever it is shared with a task (or unshared). In other words, objects are aware of their concurrent use by tasks and can adapt their behavior whenever the sharing pattern changes. This awareness enables programmers or programming language designers to guarantee system-wide invariants such as mutual exclusion, deadlock freedom, or even determinism. In fact, the programming model we present guarantees that the results produced by any parallel execution of a program that follows this model are *deterministic* and *equal to those produced by a sequential execution*.

We build upon our previous work [Faes and Gross 2017], which suggested the use of roles to ease object-oriented parallel programming using the *Rolez* language, but which failed to address many important issues in PP. Here we present several novel language concepts that make role-based parallel programming in Rolez simpler and more efficient. We also present the complete formal programming model that backs the Rolez language, including a formal proof of determinism. Every Rolez program implicitly follows the rules of this model and profits from the guarantees it provides.

Most importantly, Rolez enables programmers to efficiently implement data-parallel programs through the concept of *slicing*. With slicing, a programmer can partition an object, for example an array, into multiple disjoint *slices*, and then share and use these slices independently. By starting multiple tasks and assigning a different slice of data to each of them, data can be processed in parallel. Since these slices (like all objects in Rolez) are concurrency-aware, they detect if the programmer erroneously shares one of them with multiple tasks in a way that could cause nondeterminism. In addition, slices are *partitioning-aware*: That is, slices are aware of all the other slices of the same array (and of their roles) and also prevent interfering concurrent access that may result if the programmer's partitioning erroneously produced non-disjoint slices.

By default, the Rolez runtime system prevents any interfering access by transparently blocking those tasks that should *logically* execute later, essentially falling back to sequential execution where

necessary. However, Rolez also features novel role-based parallelism constructs that check for interference *eagerly* and report an error in case the programmer made a mistake.

The main contributions of this paper are the following:

(1) **Formal Model Definition.** We present a formal definition of the Parallel Roles programming model, including the sketch of a formal proof of determinism (Section 3). The full proof is available separately [Faes and Gross 2018].
(2) **Language Definition.** We have designed a language called Rolez that implements Parallel Roles at the language level and features a set of novel role-based PP language concepts that enable programmers to safely and efficiently implement a range of parallel patterns (Sections 4–6). We have implemented a prototype compiler and runtime system for Rolez, which are based on the Java platform.
(3) **Empirical Evaluation**. We study 8 parallel programs implemented in Rolez. We explain the parallel patterns they contain, present how they make use of the novel Rolez constructs, and analyze their performance, comparing them to manually synchronized implementations, written in Java (Section 7). Our results show that all Rolez programs achieve substantial parallel speedups (one of them up to 24× on a 32-core machine) and that the overhead compared to Java, which is caused by the maintenance of roles and by the interference checks, is moderate (below 20% for many programs).

2 PARALLEL ROLES OVERVIEW

This section gives an informal overview of the main concepts in Parallel Roles, as presented previously [Faes and Gross 2017].

The main idea behind Parallel Roles is to use *objects* as the basis to reason about concurrent effects and parallelism, by making them aware of the tasks with which they are shared. A task is simply a method that is supposed to execute in a separate thread, in parallel to the code that calls that method. In the standard OOP model, an object is a collection of fields plus a collection of methods. In the Parallel Roles model, every object has a third component: the *roles* it currently plays for the different tasks in the program, as illustrated in Figure 1.

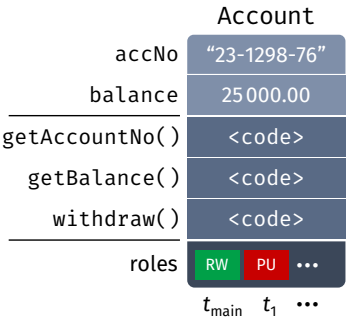


Fig. 1. The components of an object: fields, methods, and roles

The roles of an object determine which interactions are legal in which tasks and what happens when an illegal interaction occurs. Parallel Roles is a simple model, with only three roles: READWRITE, READONLY, and PURE. The READWRITE role permits both field read and field write operations, while READONLY permits only read operations. PURE permits neither (except if a field is final; then it may be read). The roles an object plays may change when the object is shared with a new task or when a task it was shared with finishes; this is how an object adapts to different sharing patterns. There are strict rules, called *role transition rules*, that define when and how the roles of an object change and, as a consequence, which *combinations* of roles an object may play at any point in time. Using the formal definition in Section 3, we prove that these rules guarantee noninterference and, by extension, determinism.

To control which role an object plays in a certain task, the programmer provides a *role declaration* for that object, which can be thought of as an annotation for the task parameter that corresponds to that object. Once a task is started, all the objects that are shared with that task perform a *role transition*, such that their roles in the new task match the ones declared by the programmer. This is one of the role transition rules.

A role transition may also change the role an object plays in the parent task (the task that starts the new task). For example, this is the case if the new task declares the READONLY role for an object. In that case, the object becomes READONLY also in the parent task, to prevent interference. Hence, while an object is guaranteed to play the declared role at the *beginning* of a task, a role declaration does not mean that the object plays this role for the whole duration of the task; it only means that the declared role is the *most permissive* role that this object may play in the task. For example, if the declared role of an object is READONLY, this object may play the READONLY or the PURE role in that task, but never the READWRITE role, since READWRITE is more permissive than READONLY.

Because objects may play a less permissive role than the declared one, some operations may temporarily become illegal, despite being legal under the declared role. To prevent errors that would result from temporarily illegal operations, Parallel Roles has the concept of *guarding*. The idea of guarding is to delay the execution of the task that attempts to perform the operation until the object plays the declared role again. This way, two tasks, the parent and its child task, can execute in parallel until the parent attempts to perform an interfering operation, at which point the parent is blocked until the child finishes and another set of role transitions takes place.

Role transitions and guarding are illustrated in Figure 2. The left side shows two pieces of code written in Rolez. The top piece defines a task `computeInterest` with an `Account` parameter that is declared as READONLY. The code below, which we assume is executed in a t_{parent} task, first creates an `Account` instance. This instance initially plays the READWRITE role in t_{parent}, as shown in the illustration on the right, at Number 1. Then, t_{parent} starts an instance of the `computeInterest` task, which we call t_{child}, and shares the `Account` object with it. This causes a role transition after which the object plays the READONLY role in both tasks (Number 2). While t_{child} executes the code in `computeInterest`, t_{parent} continues to execute the code after the **`start`** statement, i.e., the `print` and the `deposit` methods. To execute `print`, t_{parent} only needs to *read* from the object, which is permitted by the READONLY role that the object currently plays in t_{parent}. Therefore, `print` is executed in parallel with `computeInterest`. This is desirable, since the two methods do not interfere (both only read from the object). On the other hand, the `deposit` method *writes* to the object and therefore interferes with `computeInterest`: if the two were executed concurrently, then `computeInterest` would take the balance *before or after* the deposit operation as a basis for the computation, depending on the scheduling of the tasks. To prevent such nondeterminism, a compiler-inserted *guard* in the `deposit` method checks if the object actually plays the READWRITE role, blocking the execution if this is not the case. Once t_{child} finishes, the object performs another role transition and becomes again READWRITE for t_{parent}, at which point the `deposit` method may continue to execute (Number 3).

3 FORMAL DEFINITION OF PARALLEL ROLES

While it may be *intuitively* clear how role transition rules and guarding prevent interference and guarantee determinism, the description in the previous section is informal and therefore unsuitable for any detailed reasoning or even a formal proof. In this section, we present a formal definition of the Parallel Roles model, for which determinism is proven. A sketch of the determinism proof is given in Appendix A, while the full proof is available as a technical report [Faes and Gross 2018].

Our formal definition comprises the core concepts of role transitions and guarding, as presented above, while attempting to be as minimal as possible, including only those elements that are essential to the understanding of the model and for the determinism proof. For example, while we model read and write operations from and to objects, the actual names and contents of an object's fields are not modeled. Instead, we model only the component that makes an object concurrency-aware, i.e., the roles that it plays. Similarly, we do not model the precise execution state of a task, i.e., the list of instructions a task executes or the expressions it evaluates. Instead, we focus on the

```
task computeInterest(
    acc: readonly Account): int {
    ...
}
```

```
var acc = new Account(20000);

var interest = start computeInterest(acc);

print(acc.balance);
acc.deposit(5000);
```

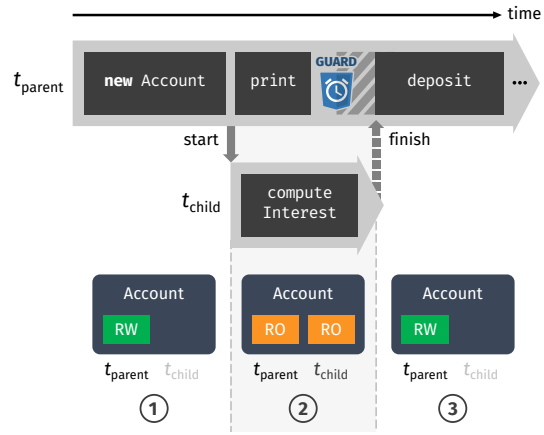


Fig. 2. An example of role transitions and guarding

$$
\begin{array}{llll}
\varrho & & \in \textbf{Role} = \{\textsc{rw}, \textsc{ro}, \textsc{pu}\} & \text{(I)} \\
r & & \in \textbf{Reference} & \\
t, \tau & & \in \textbf{Task ID} & \\
R & ::= \{r_i \,^{i \in 1..n}\} & \in \textbf{References} = \mathcal{P}(\textbf{Reference}) & \\
T & ::= \langle t, \tau, R^{\textsc{rw}}, R^{\textsc{ro}}, c \rangle & \in \textbf{Task} = \textbf{Task ID} \times (\textbf{Task ID} \cup \{\bullet\}) \times \textbf{References}^2 \times (\mathbb{N}^0 \cup \{\bullet\}) & \text{(II)} \\
Ts & ::= \{T_i \,^{i \in 1..n}\} & \in \textbf{Tasks} = \mathcal{P}(\textbf{Task}) & \\
O & ::= \{t_i :: \varrho_i \,^{i \in 1..n}\} & \in \textbf{Object} = \textbf{Task ID} \rightarrow \textbf{Role} & \text{(III)} \\
H & ::= \{r_i \mapsto O_i \,^{i \in 1..n}\} & \in \textbf{Heap} = \textbf{Reference} \rightarrow \textbf{Object} & \\
S & ::= \langle H, Ts \rangle & \in \textbf{State} = (\textbf{Heap} \times \textbf{Tasks}) \cup \{S_{\text{error}}\} &
\end{array}
$$

Fig. 3. Formal domain of the Parallel Roles model. The important parts are highlighted: the three roles (I), role declarations of tasks (II), and the definition of objects, which are mappings from tasks to roles (III).

role-related aspects of task, i.e., its role declarations, and model the execution state of a task as a single counter, which is incremented whenever the task executes an operation.

3.1 Domain

The formal domain of Parallel Roles, which describes the "structure" of the entities that we model, is shown in Figure 3. The parts that are special for the Parallel Roles model are highlighted in Blue and we refer to these highlighted parts using the roman numerals on the right. We first define some basic entities: the three roles READWRITE, READONLY and PURE (see Highlight I); references, which uniquely identify objects on the heap; and task IDs, which uniquely identify tasks in a program. We assume that references and task IDs are disjoint infinite sets.

Next, we define the interesting entities in the model: tasks and objects. First, a task T is a tuple that contains (1) the ID t of the task itself, (2) the ID τ of t's parent task, (3) the role declarations $R^{\textsc{rw}}$ and $R^{\textsc{ro}}$, and (4) the instruction counter c. The most important components are the role declarations, which are represented as two sets of references (II). The $R^{\textsc{rw}}$ set contains the references to all the objects that are supposed to be shared with t as READWRITE, while $R^{\textsc{ro}}$ contains those shared as

$$
\begin{array}{lcll}
\text{obj}_H(r) & \triangleq & O, \text{ s.t. } (r \mapsto O) \in H & \\
\text{task}_{Ts}(t) & \triangleq & T = \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, c\rangle, \text{ s.t. } T \in Ts & \\
\text{parent}_{Ts}(t) & \triangleq & \tau, \text{ s.t. } \text{task}_{Ts}(t) = \langle t, \tau, \_, \_, \_\rangle & \\
\text{ancest}^i_{Ts}(t) & \triangleq & \begin{cases} t, & \text{if } i = 0 \vee \text{parent}_{Ts}(t) = \bullet \\ \text{ancest}^{i-1}_{Ts}(\text{parent}_{Ts}(t)), & \text{otherwise} \end{cases} & \\
\text{ancests}_{Ts}(t) & \triangleq & \{\text{ancest}^i_{Ts}(t) \mid i \in 1..\infty\} & \\
\text{step}(\langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, c\rangle) & \triangleq & \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, c+1\rangle & \\
\text{mayRead}_H(t, r) & \triangleq & (t :: \textsc{rw}) \in \text{obj}_H(r) \vee (t :: \textsc{ro}) \in \text{obj}_H(r) & \text{(I)} \\
\text{mayWrite}_H(t, r) & \triangleq & (t :: \textsc{rw}) \in \text{obj}_H(r) & \text{(II)}
\end{array}
$$

Fig. 4. Helper functions for the state transition relation. The most important are $\text{mayRead}_H(t, r)$ and $\text{mayWrite}_H(t, r)$, which indicate whether a task is permitted to read from or to write to an object.

READONLY. Note that a R^{PU} set is not required, as we explain in Section 3.2. Second, an object is modeled as a mapping from task IDs to roles (III). This reflects the fundamental idea of Parallel Roles that every object plays a specific role for each task in the program, at any point in time. Also, as explained before, the rest of an object's state is not modeled.

Finally, the state of a complete program comprises the heap, which is a mapping from references to objects, and the set of tasks. Note that there are a few special values in the model. We use $\tau = \bullet$ to denote that a task has no parent task (which is true exactly for the "main" task). Further, we use $c = \bullet$ for tasks that have finished (instead of removing them from the set of tasks in the program). Finally, we use S_{error} to denote the program state that results from an erroneous operation.

3.2 State Transition Relation

We describe the semantics of the model as a state transition relation. Whenever a task in a program performs an operation, the program transitions from its current state to the next one, according to the respective state transition rule. If some operation violates the rules of the model, the program transitions to the special S_{error} state. For example, this is the case when a task attempts to write to an object that it declared as READONLY. On the other hand, temporarily illegal operations do *not* cause an error, as explained in Section 2. These operations are illegal under the role that an object *currently* plays for a task t, but will become legal later, when the object plays a more permissive role again for t. Hence, such operations are modeled using a state transition back to the same state, effectively preventing t from making any progress.

Helper Functions. To define the state transition relation, we first need a few helper functions, which are defined in Figure 4. The function $\text{obj}_H(r)$ denotes the object that the reference r refers to (for a given heap H), while $\text{task}_{Ts}(t)$ denotes the task with the ID t (for a given set of tasks Ts). Further, $\text{ancests}_{Ts}(t)$ denotes the set of all ancestor tasks of t, that is, t's parent, t's parent's parent, and so on. (Note that we use the symbol _ as a wildcard for values that are not used.) In addition, $\text{step}(T)$ returns a "copy" of task T, with an incremented instruction counter.

The two most interesting functions are $\text{mayRead}_H(t, r)$ and $\text{mayWrite}_H(t, r)$ (see I and II). These predicates indicate whether a task t is permitted to read from or to write to the object referred to by reference r, according to the *current* roles of the object on the heap H. If the object contains the mapping $(t :: \textsc{rw})$, i.e., it plays the READWRITE role in t, then t may write to the object. Similarly, if the object contains the mapping $(t :: \textsc{rw})$ or $(t :: \textsc{ro})$, then t may read from the object. Note that

The initial state at the beginning of a program:

$$S_0 \triangleq \langle \emptyset, \{\langle t_{\text{main}}, \bullet, \emptyset, \emptyset, 0\rangle\}\rangle$$

Task $\langle t, \_, \_, \_, c\rangle \in Ts$, $c \neq \bullet$ creates a new object, which is identified by a fresh reference $r \in \textbf{Reference} - \{r_i \mid (r_i \mapsto \_) \in H\}$:

$$\langle H, Ts\rangle \xrightarrow{t \text{ create } r} \langle H \cup \{r \mapsto \{t :: \text{RW}\}\}, \{\text{upd}(T_i) \mid T_i \in Ts\}\rangle, \text{where} \tag{I}$$

$$\text{upd}(\langle t_i, \tau_i, R_i^{\text{RW}}, R_i^{\text{RO}}, c_i\rangle) \triangleq \begin{cases} \langle t_i, \tau_i, R_i^{\text{RW}} \cup \{r\}, R^{\text{RO}}, c_i + 1\rangle, & \text{if } t_i = t \\ \langle t_i, \tau_i, R_i^{\text{RW}} \cup \{r\}, R^{\text{RO}}, c_i\rangle, & \text{else, if } t_i \in \text{ancests}_{Ts}(t) \\ \langle t_i, \tau_i, R_i^{\text{RW}}, R^{\text{RO}}, c_i\rangle, & \text{otherwise} \end{cases} \tag{II}$$

Task $T = \langle t, \_, R^{\text{RW}}, R^{\text{RO}}, c\rangle \in Ts$, $c \neq \bullet$ reads from a field with reference r as the target:

$$\langle H, Ts\rangle \xrightarrow{t \text{ read } r} \begin{cases} \langle H, Ts - \{T\} \cup \{\text{step}(T)\}\rangle, & \text{if } \text{mayRead}_H(t, r) \\ \langle H, Ts\rangle, & \text{else, if } r \in R^{\text{RW}} \cup R^{\text{RO}} \\ S_{\text{error}}, & \text{otherwise} \end{cases} \tag{III}$$

Task $T = \langle t, \_, R^{\text{RW}}, R^{\text{RO}}, c\rangle \in Ts$, $c \neq \bullet$ writes to a field with reference r as the target:

$$\langle H, Ts\rangle \xrightarrow{t \text{ write } r} \begin{cases} \langle H, Ts - \{T\} \cup \{\text{step}(T)\}\rangle, & \text{if } \text{mayWrite}_H(t, r) \\ \langle H, Ts\rangle, & \text{else, if } r \in R^{\text{RW}} \\ S_{\text{error}}, & \text{otherwise} \end{cases} \tag{IV}$$

Fig. 5. Definition of the initial state S_0 and first part of the state transition relation $\longrightarrow$.

objects that play the PURE role for a task t simply contain no mapping for t, instead of a mapping $(t :: \text{PU})$.

We use these functions, especially $\text{mayRead}_H(t, r)$ and $\text{mayWrite}_H(t, r)$, to define the state transition relation $\longrightarrow$, which is given in Figures 5 and 6. First, a few notational remarks: In addition to the standard $\cup$ and $\cap$ set operators, we use $-$ for the set difference. Further, given two mappings A and B, we write $A \,/\, B$ for the union of A and B, where B's entries override A's when there is a key clash. Finally, all operators $\cup, \cap, -, \,/\,$ have the same precedence and evaluate from left to right.

The first part in Figure 5 defines the initial state S_0, which is the same for every program (as we do not model the program instructions themselves). S_0 is simply the tuple that contains the empty head and a set of one single task with some ID t_{main}, with no parent task, empty role declarations, and a instruction counter of 0. The rest of Figure 5 defines the first three out of five cases for the $\longrightarrow$ relation, which correspond to object creation and access operations:

Object Creation: $\xrightarrow{t \text{ create } r}$. When a task t creates a new object, the object is READWRITE for t and, implicitly, PURE for all other tasks (I). In addition, the object is added to the R^{RW} set of t and of all of t's ancestor tasks (II). As explained in Section 3.1, the R^{RW} and R^{RO} sets represent the *declared* roles of objects and are, in addition to the *current* roles, required to determine whether an operation is *legal*, *temporarily illegal*, or *erroneous*. By adding a newly created object to t's and t's ancestors' R^{RW} sets, this object's implicitly declared role for these tasks is READWRITE, which means that it is legal or at least only temporarily illegal for them to access it. In fact, because the current role of

Task $T = \langle t, \_, R^{\text{RW}}, R^{\text{RO}}, c\rangle \in Ts$, $c \neq \bullet$ starts a new task with a fresh ID of $t_{\text{ch}} \in$ **Task ID** $- \{t_i \mid \langle t_i, \_, \_, \_, \_\rangle \in Ts\}$ and with role declarations $R^{\text{RW}}_{\text{ch}}$, $R^{\text{RO}}_{\text{ch}}$:

$$\langle H, Ts\rangle \xrightarrow{t \text{ start } t_{\text{ch}}(R^{\text{RW}}_{\text{ch}}, R^{\text{RO}}_{\text{ch}})} \begin{cases} \langle H', Ts'\rangle, & \text{if } \forall r \in R^{\text{RW}}_{\text{ch}} : \text{mayWrite}_H(t, r) \wedge \\ & \quad \forall r \in R^{\text{RO}}_{\text{ch}} : \text{mayRead}_H(t, r) \\ \langle H, Ts\rangle, & \text{else, if } R^{\text{RW}}_{\text{ch}} \subseteq R^{\text{RW}} \wedge R^{\text{RO}}_{\text{ch}} \subseteq R^{\text{RO}} \cup R^{\text{RW}} \\ S_{\text{error}}, & \text{otherwise} \end{cases}, \text{where} \tag{I}$$

$$H' \triangleq H \,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{t :: \text{RO}, t_{\text{ch}} :: \text{RO}\} \mid r \in R^{\text{RO}}_{\text{ch}}\} \tag{II}$$

$$\,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{t :: \text{PU}, t_{\text{ch}} :: \text{RW}\} \mid r \in R^{\text{RW}}_{\text{ch}}\}, \tag{III}$$

$$Ts' \triangleq Ts - \{T\} \cup \{\text{step}(T), \langle t_{\text{ch}}, t, R^{\text{RW}}_{\text{ch}}, R^{\text{RO}}_{\text{ch}}, 0\rangle\}$$

Task $T = \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, c\rangle \in Ts$, $c \neq \bullet$ is about to finish:

$$\langle H, Ts\rangle \xrightarrow{t \text{ finish}} \begin{cases} \langle H' Ts'\rangle, & \text{if } \forall \langle t_i, \tau_i, \_, \_, c_i\rangle \in Ts : \tau_i \neq t \vee c_i = \bullet \\ \langle H, Ts\rangle, & \text{otherwise} \end{cases}, \text{where} \tag{IV}$$

$$H' \triangleq H \,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{\tau :: \varrho(r)\} - \{t :: \_\} \mid r \in R^{\text{RO}}\} \tag{V}$$

$$\,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{\tau :: \text{RW}\} - \{t :: \_\} \mid r \in R^{\text{RW}}\}, \tag{VI}$$

$$\varrho(r) \triangleq \begin{cases} \text{RO}, & \text{if } \exists t_i \neq t : \text{mayRead}_H(t_i, r) \\ \text{RW}, & \text{otherwise} \end{cases}, \tag{VII}$$

$$Ts' \triangleq Ts - \{T\} \cup \{\langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, \bullet\rangle\}$$

Fig. 6. Second part of the definition of the state transition relation $\longrightarrow$.

the object is PURE for all tasks but t, only t may immediately access the object. The parent of t, for example, would be blocked by guarding until t has finished, as explained below.

Note that for this case of the state transition to apply, the instruction counter c must be $\neq \bullet$, like for all other cases. This means that only tasks that have not yet finished can perform any operation. Also note that c is incremented, indicating that the operation was successful.

Read or Write Operation: $\xrightarrow{t \text{ read } r}$ *or* $\xrightarrow{t \text{ write } r}$. When a task t attempts to read from or write to an object referred to by a reference r, that object's *current* and *declared* role for t determine whether that operation is legal, temporarily illegal, or erroneous (III and IV). Hence, there are *three* possible outcomes for such an operation: (1) If the *current* role of the object permits the operation, as determined by $\text{mayRead}_H(t, r)$ or $\text{mayWrite}_H(t, r)$, the operation is successful. This is represented by incrementing t's instruction counter c. Since we do not model the fields of objects or the contents of local variables, no other change of program state takes place. (2) If the current role of the object does not permit the operation, but the *declared* role does, as determined by t's R^{RW} and R^{RO} sets, then the operation is delayed, but no error occurs. This is an instance of guarding, and is represented by a transition back to the same state. (3) Finally, if the declared role for the object does not permit the operation, then this is a programmer error and the program transitions to the S_{error} state.

The final two cases for the state transition relation, which describe the starting and finishing of tasks, are shown in Figure 6:

Task Start Operation: $\xrightarrow{t \text{ start } t_{\text{ch}}(R^{\text{RW}}_{\text{ch}}, R^{\text{RO}}_{\text{ch}})}$. Whenever a new child task t_{ch} is started, the programmer supplies the *role declarations* for that task, which we model as two sets $R^{\text{RW}}_{\text{ch}}$ and $R^{\text{RO}}_{\text{ch}}$. These sets

contain the references to all objects that should be shared with t_{ch} and play the READWRITE role or, respectively, the READONLY role. When the task is successfully started, then those objects adapt to their new sharing pattern by performing a *role transition*: (1) Objects referred to by the R^{RW}_{ch} set become READWRITE in t_{ch} and PURE in t (III), which means that t may not access those objects anymore (until t_{ch} finishes; see below) and (2) objects referred to by R^{RO}_{ch} become READONLY in both t and t_{ch} (II), meaning that both tasks can still read from, but cannot write to them anymore (again, until t_{ch} finishes). R^{RW} takes precedence, in case a reference is in both sets. Note that objects shared as PURE keep playing the same role in t and need not be explicitly declared, e.g., in a R^{PU} set. Since objects are implicitly PURE for tasks in which they play no explicit role, no role transition is required for them to play the PURE role in t_{ch}.

Like for read and write operations, there are three possible outcomes for a start operation (I). The successful case, as described above, only applies if the *current* roles for t of all involved objects match the roles declared with R^{RW}_{ch} and R^{RO}_{ch}: Objects in R^{RW}_{ch} must play the READWRITE role in t before t_{ch} starts and those in R^{RO}_{ch} must play the READONLY or the READWRITE role. This restriction ensures that t cannot, for example, share an object as READWRITE with more than one task at a time. Otherwise, if the *declared* roles for t of all involved objects are at least as permissive as those for t_{ch}, the operation is delayed. This is another instance of guarding. Otherwise, the start operation results in S_{error}, because t may not share an object such that its declared role is more permissive in t_{ch} than it is in t.

Task Finish Operation: $\xrightarrow{t \text{ finish}}$. When a task t finishes, the sharing patterns for the objects that were declared in t's R^{RW} and R^{RO} sets (or, if created in t or one of t's child tasks, were added later) changes again, so these objects perform another role transition. The transitions make sure that the objects that t's parent task τ shared with t can be accessed by τ again once t finishes. All objects in the R^{RW} set become READWRITE in τ (VI). These objects were either shared with t as READWRITE, or newly created in t or one of t's children. On the other hand, an object in R^{RO} (and not R^{RW}) was shared as READONLY and its roles depend on whether there are any other tasks left in which the object plays the READONLY role (V and VII). The reason for this is that τ could have shared an object as READONLY with multiple tasks and such an object should only become READWRITE for τ again once *all* of these tasks have finished.

A finish operation has *two* possible outcomes (IV). Task t can only finish once all of its own child tasks have finished, which is expressed using the condition $\forall \langle t_i, \tau_i, \_, \_, c_i \rangle \in Ts : \tau_i \neq t \vee c_i = \bullet$. This restriction is a simplification to ensure that objects only ever play the READWRITE role in one single task, which is a key invariant in the model. Thus, as long as there are still any active child tasks, t cannot finish and no progress happens. However, unlike a read, write, or start operation, a finish operation can never result in an error, because all of t's child tasks will finish eventually.

4 ROLEZ LANGUAGE OVERVIEW

The following three sections present the design of the Rolez language. This section provides an overview of the basic features that we described previously [Faes and Gross 2017] and explains how they relate to the formal definition of the Parallel Roles model in the previous section. Sections 5 and 6 focus on the advanced concepts of slicing and eager interference checking.

4.1 Basic Language Features

Rolez is a Java-like language that enforces the Parallel Roles model on the language level, which means that *every* program written in Rolez implicitly follows this model and benefits from the guarantees it provides. In other words, every Rolez program is guaranteed to be free of interference

```
1  class App {
2     def calcInterest(balance: int): int {
3        return (0.015 * balance) as int;
4     }
5     task pure payInterest(acc: readwrite Account): {
6        val intrst = this.calcInterest(acc.getBalance);
7        acc.deposit(intrst);
8     }
9     task pure main: {
10       val acc = new Account;
11       acc.deposit(1000);
12       this start payInterest(acc);
13    }
14 }
```

```
15 class Account {
16    var balance: int
17    var owner: User
18
19    def deposit(amount: int): {
20       this.balance += amount;
21    }
22    def getBalance: int {
23       return this.balance;
24    }
25 }
```

Fig. 7. Rolez code example for tasks and role declarations. Note that `void` return types can be omitted.

and is therefore deterministic (unless nondeterminism is introduced explicitly, for example, with a random number generator).

Because Rolez enforces the model on the language level, most role-specific entities defined in Section 3.1 correspond directly to explicit language constructs, including tasks and role declarations. However, the roles that an object plays at runtime do not correspond to any explicit language construct (unlike fields or methods). Instead, the roles of all Rolez objects are maintained implicitly by the runtime system, which updates them whenever tasks start or finish.

Tasks. Tasks in Rolez are declared in the same way as methods. Two different keywords, `def` and `task`, are used to distinguish the two. Likewise, starting a task is expressed in the same way as invoking a method, except for the keyword `start`, which replaces the dot. When an object is supposed to be shared with a task, the programmer simply creates a corresponding parameter for that task and passes the object as an argument when starting it. Figure 7 shows a Rolez example program that illustrates these points. Lines 2 to 8 contain the declarations of a method and a task, while Lines 11 and 12 show how these are called or started, respectively.

Role Declarations. To declare the role of an object in a task, the programmer annotates the corresponding task parameter with that role, as shown on Line 5. This line indicates that the `payInterest` task requires an `Account` object that plays the READWRITE role to be shared with it. In terms of the model in Section 3, this means that, at runtime, the R^{RW} set of the `payInterest` task contains the object reference that is passed as an argument to `payInterest`. The parameter needs to be declared as READWRITE because `payInterest` modifies the balance of the given account when calling `deposit` on Line 7. When this task is started on Line 12, the `Account` object that is passed as an argument performs a role transition and becomes READWRITE for the `payInterest` task and PURE for the `main` task, as defined in Section 3.

Note that both the `payInterest` task and the `main` task have another, implicit, parameter: the receiver, i.e., the "`this`". The role for the receiver is declared right after the `task` keyword and is PURE for both of these tasks. This means that an `App` instance does not perform any role transition when used as the receiver for the `payInterest` or `main` task.

4.2 Joint Role Transitions

An important practical language concept in Rolez are *joint role transitions*. This concept makes it simpler to work with objects whose functionality depends on other objects. Such objects often store references to the objects they depend on. For example, a `Bank` object, with a `payAllInterest` method, may store references to all `Account` objects of that bank. When `payAllInterest` is called, it computes and deposits the yearly interest for each of its accounts. With joint role transitions, if a `Bank` object is shared with a task t, then all of the reachable `Account` objects *implicitly* have the same declared role and *automatically* perform the same role transition as the `Bank` itself, to ensure that the `payAllInterest` method works in t as well. The Rolez example in Figure 7 also contains a joint role transition: when an `Account` object is shared with the `payInterest` task, the `User` object referred to by this `Account`'s `owner` field performs the same transition as the `Account` itself (even though the `owner` field is not accessed inside `payInterest`).

The idea of joint role transitions was described previously as the "Object Graphs" extension to the Parallel Roles model [Faes and Gross 2017]. That extension is a set of additional rules that describe explicitly what happens when an object with references to other objects is shared with a task.

In this paper, we take a different approach: As mentioned earlier, our formal definition in Section 3 is as minimal as possible, to make it simple to understand and to reason about. In particular, the formal model simply uses two sets of references, R^{RW} and R^{RO}, to represent the objects that are shared with a task. This simplicity also makes the model more general and lets us define joint role transitions without actually modifying the definition in Section 3. Instead, we redefine the *meaning* of R^{RW} and R^{RO} with respect to the concrete language we want to describe. This makes it possible to map different language designs onto the model, by defining how language-level task arguments translate into the model-level sets R^{RW} and R^{RO}.

In the case of Rolez, the R^{RW} set is defined to be the set of all objects that are reachable from any of the arguments that correspond to a READWRITE task parameter. Similarly, the R^{RO} set is defined to be the set of objects reachable from READONLY arguments.

4.3 Rolez Type System

Rolez features a static type system that includes a notion of roles, to report certain programmer errors at compile time. The type system is *complementary* to guarding, but it does not *replace* guarding or role transitions, which are dynamic concepts. When a task τ has started a child task t, guarding, on the one hand, prevents *temporarily* illegal operations in the parent task τ, potentially blocking τ's execution until t has finished. The type system, on the other hand, prevents *permanently* illegal operations in t, which cannot be handled by guarding.

An operation is permanently illegal in a task if it is prohibited by the *declared* role of an object in that task, which is the most permissive role that this object may play in that task. For example, if a task declares a parameter as READONLY, then a corresponding object can never play a more permissive role than READONLY, and thus must never be modified in this task. This check, that the declared role of an object is respected, is necessary for determinism; therefore, such checks are part of the formal model in Section 3: errors resulting from permanently illegal operations correspond to state transitions to S_{error}. While the model in Section 3 may imply that these errors, called *role errors*, may occur at runtime, this is not necessarily the case; in Rolez, all role errors are reported by the compiler, using the static type system designed specifically for this purpose.

Since the type system is irrelevant for the main concepts presented in this paper, including the language concepts presented in the following sections, we only discuss the most important features

of the type system here. In addition, all Rolez code shown in this paper (except in Figure 9) is simplified, leaving away the role type annotations (except for task parameters—see below).

Role Types. The Rolez type system is an extension of the class-based type system known from Java and other OOP languages. The type of every (non-primitive) variable in Rolez contains the class of the object that this variable refers to (or a superclass thereof), plus the declared role of the object in the currently executing task (or a *superrole* thereof—see *Subtyping* below). Thus, all reference types in Rolez, called *role types*, contain two parts, the *static role* and the *class part*. For example, a role type `readwrite Account` consists of the static role `readwrite` and the class part `Account`. If a class has a type parameter, the class part may also contain a type argument, as in the role type `readwrite Array[int]`. If the type argument itself is a reference type, it contains again a static role, as in the type `readwrite Array[readonly Account]`.

Using role types, the type system tracks the declared role of every object that is shared with a task and ensures that this declared role is respected. This is possible because, unlike the actual current role, the *declared* role of an object never changes during the execution of a task. Thus, the static role of a variable always corresponds to the declared role of the object this variable refers to or, due to subtyping, to a *superrole* thereof.

Subtyping. Subtyping for role types takes into account not only the class part, but also the static role: A role type $r_1 C_1$ is a subtype of $r_2 C_2$, written $r_1\ C_1 <:\ r_2\ C_2$, if and only if the class C_1 is a subclass of C_2 and the role r_1 is a *subrole* of r_2, written $r_1 <: r_2$. The subrole relation $<:$ is transitive and reflexive and includes `readwrite` $<:$ `readonly` $<:$ `pure`. For example, the type `readwrite Account` is a subtype of `readonly Account`, and also of `readwrite SavingsAccount` (given that the class `SavingsAccount` extends the class `Account`). This is illustrated in Figure 8, which shows the complete type hierarchy for three classes, where `SavingsAccount` extends `Account`, which in turn extends `Object`. Note that every type is also technically a subtype of itself, though not shown in the figure.

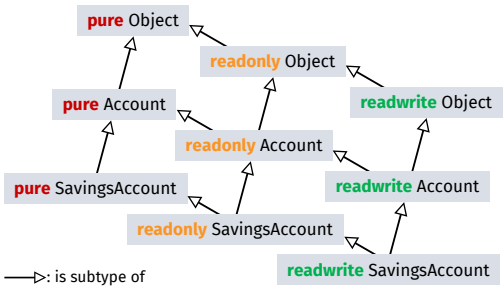


Fig. 8. Rolez subtyping example.

Just as class-based subtyping enables code reuse, for example by allowing `SavingsAccount` instances to be used where `Account` objects are expected, role-based subtyping enables safe code reuse with respect to roles. For example, a method with a `readonly Account` parameter can be safely called with an `readwrite Account` argument, as `readwrite` permits everything that `readonly` permits.

Example. Role declarations for task parameters, which are explained in Section 4.1, are integrated into the type system. For example, the `acc` parameter of the `foo` task in Figure 9 is declared as `readonly` and hence has the type `readonly Account`. Note that role declarations for task parameters serve two purposes: On the one hand, they are regular role type annotations that are required by the Rolez type system. On the other hand, they serve as the role declarations of the task they belong to and define the role transitions that are performed at the beginning of that task (see Section 4.1). Thus, role declarations for task parameters are a fundamental requirement of the Parallel Roles model, while role type annotations for other constructs, like local variables, fields, or method parameters, are required merely to report role errors at compile time.

In the example in Figure 9, the type system provides two guarantees: 1) the object that `acc` refers to is never assigned to a variable of a role type with a more permissive static role and 2) variables

```
1  task pure foo(acc: readonly Account): {
2     val ro:  readonly Account = acc; // OK
3     val rw: readwrite Account = acc; // ERROR
4
5     println(ro.balance);     // OK
6     ro.balance += 1000;      // ERROR
7     println(ro.getBalance); // OK
8     ro.deposit(1000);        // ERROR
9
10    val userRw: readwrite User = ro.owner; // ERROR
11    val userRo:  readonly User = ro.owner; // OK
12 }
```

```
13 class Account {
14    var balance: int
15    var owner: readwrite User
16
17    def readwrite deposit(amount: int): {
18       this.balance += amount;
19    }
20    def readonly getBalance: int {
21       return this.balance;
22    }
23 }
```

Fig. 9. Code example for role types. The left side shows permitted and prohibited uses of `Account` variables; the right side shows the `Account` class from Figure 7 with role type annotations. Note that the types of local variables (Lines 2, 3, 10, 11) could be inferred by the compiler, but are declared here for illustration purposes.

with a static role of `readonly` cannot be used as targets for field write operations. Together, these two ensure that `acc` is never modified inside the `foo` task.

The first guarantee is illustrated on Lines 2 and 3. While assigning `acc` to variable `ro` of type `readonly Account` is permitted, assigning it to `rw` is prohibited, because the type of `acc` is not a subtype of `rw`'s type `readwrite Account`. Note that the standard *right-hand side type* <: *left-hand side type* rule applies here. Lines 5–8, together with the definition of the `Account` class on the right, illustrate the second guarantee. Reading from the `balance` field of the `ro` variable (Line 5) is permitted, as `ro`'s static role is `readonly`. However, *writing to* a field is not permitted by this static role, so Line 6 is an error. Lines 7 and 8 illustrate how this guarantee is maintained across method boundaries: Line 7 reads the balance of `ro` by calling `getBalance`. This method declares the receiver as `readonly`, so the type of "`this`" inside the method is `readonly Account` and thus reading from the `balance` field is permitted. When calling a method, the compiler checks that the target, in this case the variable `ro`, has a static role that is a subrole of the declared role of the receiver. This is the case on Line 7, but not on Line 8, because the `deposit` method declares its receiver as `readwrite`. Hence, it is impossible to modify the balance of the `Account` object that `acc` and `ro` refer to—both directly or indirectly by calling a method. To fix the `foo` task, the programmer would need to declare the `acc` parameter (and all the variables it is assigned to) as `readwrite` instead. This would eliminate the errors reported by the type system and, at runtime, would cause the corresponding object to play the READWRITE role in `foo`, allowing to modify it.

The type system not only guarantees that `acc` itself, but also that all the objects that `acc` refers to remain unmodified in the `foo` task. This is required because of joint role transitions; as explained in Section 4.2, all objects that are reachable from an object that is shared with a task implicitly have the same declared role as that object itself. In the example in Figure 9, this means that the declared role of the `User` object that is stored in `acc`'s `owner` field is `readonly`, like the declared role of `acc` and disregarding the static `readwrite` role of the `owner` field. To ensure that this `User`'s declared role is respected too, the Rolez type system includes a special typing rule that concerns field reads: The resulting static role of a field read expression $t.f$ is not simply the role of the field f, but the *less permissive* static role of f and of the target t itself. This is illustrated on Lines 10 and 11 in Figure 9: When reading the field `ro.owner`, it is an error to assign the result to the `readwrite` variable `userRw`, as the less permissive static role of `ro` and `owner` is `readonly`. Thus, `ro.owner` can only be assigned to a variable with a static role of "at most" `readonly`, which prevents any modification.

5 SLICING

A common pattern in parallel programs is data parallelism. In data-parallel algorithms, some data items can be processed independently of other data items, and therefore parallel processing of these items is possible. Coarse-grain data parallelism is usually exploited by starting multiple tasks and assigning different partitions of the data space to each of them. Rolez enables programmers to implement parallel algorithms safely and efficiently using *slicing*.

5.1 Overview

Slicing enables programmers to "split" an object, e.g., an array, into multiple *slices* and share each of them with a different task. Slices, like all objects in Rolez, are aware of the tasks they are shared with, and adapt by changing their *individual* roles. In addition, slices are *partitioning-aware*. That is, slices of the same array are aware of *each other's* roles, and this awareness allows them to detect and prevent interfering concurrent access due to an erroneous partitioning, as explained in Section 5.2.

Slicing is completely virtual: an array that is "split" into slices is not actually divided; a slice merely acts as a view onto an array, through which only the data elements that are part of the slice can be accessed and which may play its own role. Hence, slicing is much more efficient than copying data into individual arrays (and possibly back again after a parallel section), which is the only way to achieve data parallelism in the absence of slicing.

Since the slices of an array are just views onto the same underlying data, slices need not be disjoint. As an example, for an array with six elements (with indices 0 to 5), there could be two slices $[0:2]$ and $[3:5]$, which do not overlap. However, there could also be a slice $[2:3]$, which overlaps with both of the two former. In fact, the array itself can be considered a slice $[0:5]$, which covers the entire range of indices and overlaps with all other slices of that array. Not forcing all slices of an array to be mutually disjoint gives programmers more flexibility, allowing them to access the same array through different sets of mutually disjoint slices in different sections of the program. This includes also the trivial, but ubiquitous case in which one of these sets of slices consists of one single slice, which is the entire array.

In terms of roles, two disjoint slices are completely independent. For example, a task t may share an array slice $[0:2]$ with another task t_{ch}, while it keeps a (non-overlapping) slice $[3:5]$ for itself. This way, the first slice might be PURE in t and READWRITE in t_{ch}, while the second slice is READWRITE in t and PURE in t_{ch}. Thus, both tasks can safely read and modify their slice. On the other hand, if two slices overlap, their roles are coupled, to prevent interference between tasks. If this were not the case, for example, if a slice $[0:1]$ were READWRITE for one task and a slice $[1:2]$ of the same array were READWRITE for another task, both tasks could read and write the element with index 1, causing interference and nondeterminism.

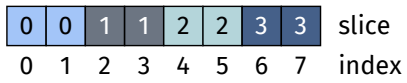
5.2 Formal Definition and Determinism

As with joint role transitions, we define slicing without extending or modifying the formal model definition in Section 3. Instead, we redefine how language-level objects (in particular slices) and references are mapped onto model-level objects and references.

So far, we have implied that there is a one-to-one mapping from Rolez objects and references to model-level objects and references. To define the semantics of slices, we use a different mapping: For every Rolez object O that contains n elements and can be sliced, e.g., an array of length n, there is a set of model-level objects $\{O_i \ ^{i \in 1..n}\}$, where O_i corresponds to the i-th element of O, e.g., the i-th array cell. For each of these O_i, there is a separate reference r_i, such that $(r_i \mapsto O_i) \in H$.

A reference to a slice $\mathcal{S}$ of O that includes the elements $i \in \text{elems}(\mathcal{S}) \subseteq \{1..n\}$ is mapped to a set of references $\{r_i \ ^{i \in \text{elems}(\mathcal{S})}\}$. This means that, whenever a reference to a slice is shared with a task in

```
val data = new Array[int](n);
...
val s = n / m; // n is a multiple of m
for (var i = 0; i < m; i++) {
    val slice = data.slice(i * s, (i+1) * s, 1);
    ...
}
```




```
val data = new Array[int](n);
...

for (var i = 0; i < m; i++) {
    val slice = data.slice(i, n, m);
    ...
}
```

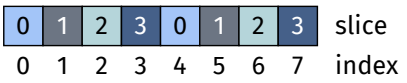


Fig. 10. Example for contiguous and striped array slicing, for `n = 8` and `m = 4`.

Rolez, the complete set of references r_i is shared with the corresponding task in the formal model. Similarly, reading from (or writing to) an element of a slice $\mathcal{S}$ in Rolez corresponds to a complete set of read (or write) operations in the model, with *all* the references r_i as targets. The operation succeeds only if all r_i permit the operation, i.e., if $\forall r_i : \text{mayRead}_H(t, r_i)$ (or $\forall r_i : \text{mayWrite}_H(t, r_i)$).

It follows that every single element of a sliceable Rolez object plays its own roles and that, strictly speaking, there are no roles for entire slices or sliceable objects. Instead, by guaranteeing noninterference and determinism on the level of individual elements, Rolez guarantees these properties also for programs with slicing. However, although there are technically no roles for entire slices, a slice can be thought of as playing a set of "aggregate" roles. The aggregate role of a slice in a task t is the *least permissive role* that any of its elements play in t: an operation is only permitted for the slice as a whole if *all* the elements permit that operation.

5.3 Array Slicing

Array slicing in Rolez is supported though the `rolez.lang.Array` class, which is the built-in class that all arrays are instances of. This class provides a method with the signature

```
def slice(begin: int, end: int, step: int): rolez.lang.Slice[T]
```

which creates a new slice, i.e., an instance of `rolez.lang.Slice`. The returned slice type has a type parameter `T` that defines the type of the elements in the slice. `T` corresponds to the element type of the array the method is invoked on. The `slice` method has three parameters, `begin`, `end`, and `step`, which specify the set of elements that the returned slice covers. The parameters `begin` and `end` define the first and "one-after-last" index of the slice. In addition, the `step` parameter can be used to create non-contiguous slices that skip elements. Such slices are useful when not all array elements are associated with the same amount of work and a contiguous partitioning scheme would lead to work-imbalanced partitions. Note that the `slice` method does not copy any data in the array, but only creates a proxy object through which the original array is accessed. This makes working with slices almost as efficient as working with the original array.

Figure 10 illustrates how the `slice` method can be used to split an array into both *contiguous* and non-contiguous, *striped* slices. From a programmer's perspective, slices are normal objects and can be passed around and shared with tasks like any other object. Only when accessing an array element, the Rolez runtime system checks whether the element is actually covered by the slice it is accessed through. Since these checks can incur a substantial overhead, they can be switched on for testing and debugging and off for production (if desired).

```
1  class Body {
2      slice position {
3          var x: double
4          var y: double
5          var z: double
6      }
7      slice velocity {
8          var vx: double
9          var vy: double
10         var vz: double
11     }
12 }
```

```
13 val body: Body = new Body;
14
15 val position: Body\position = body slice position;
16 val velocity: Body\velocity = body slice velocity;
17
18 // only the fields that are included
19 // in the slice can be accessed:
20
21 val x  = position.x;  // OK
22 val vx = velocity.vx; // OK
23 val vy = position.vy; // ERROR
```

Fig. 11. Class slicing example with slice declarations on the left and two slicing operations of the right.

Note that programmers do not actually have to write code like in Figure 10. Instead, the `Array` class provides a convenience method that simply takes the number of slices to be created, plus the partitioning scheme, and returns a whole set of slices that are guaranteed to be mutually disjoint.

5.4 Class Slicing

Array slicing deals with arrays. To support a wide range of parallel algorithms, Rolez also includes *class slicing* to define partitions of an object's fields. While array slicing is restricted to the built-in class `Array`, class slicing allows programmers to slice objects that are instances of their own classes. The underlying concept is identical to that of array slicing, but applied to the *fields* of an object instead of array cells. Thus, the formal definition in Section 5.2 applies for class slicing as well, with elems($\mathcal{S}$) representing the set of fields of an object that the slice $\mathcal{S}$ includes.

While class slicing is not as widely applicable as array slicing, it similarly enables efficient data parallelism, but on tree-like structures instead of flat arrays. In addition, it gives programmers more flexibility and fine-grained control when sharing objects with tasks, independent of data parallelism. This is especially helpful in conjunction with the concept of joint role transitions explained earlier: since objects are shared together with the objects they depend on (i.e., store references to), class slicing can help to more precisely define the set of objects that are shared with a certain task.

To slice an object of a programmer-defined class, the programmer first defines which fields of the class belong to which slice. The left side of Figure 11 shows an example of a class `Body` with two slices `position` and `velocity`. The `position` slice contains the fields `x`, `y`, `z`, while the `velocity` slice contains the fields `vx`, `vy`, `vz`. Then, to slice an instance of a class with slices, the programmer uses the **`slice`** operator, as shown on Lines 15 and 16. Like the `slice` method on arrays, the **`slice`** operator does not create a copy of the fields in the respective slice, but only creates a proxy object that delegates to the original object when accessed.

Support for class slicing is built into the type system of Rolez, as so-called *slice types*. A slice type has the form "`Class\slice`", and only permits access to those fields and methods that are declared inside the respective class slice. In the example in Figure 11, this means that the expressions "`position.x`" and "`velocity.vx`" are legal, but the expression "`position.vy`" causes a compile-time error, because `position` is of type `Body\position`, which does not permit access to the `vy` field. Note that the verbose type declarations on Lines 13–16 are not actually necessary, as Rolez infers the types of local variables automatically; they are shown for illustrative purposes.

6 EAGER INTERFERENCE CHECKING

In this section, we present two new language constructs in Rolez that give the programmer more control over when the runtime system checks for interfering accesses in concurrent tasks.

6.1 Overview

In Rolez, any kind of parallelism is achieved by starting tasks: after a task t has started another task t_{ch}, t simply continues to execute in parallel to t_{ch}, resulting in a degree of parallelism (DOP) of 2. To increase the DOP, task t (or t_{ch}) may simply start more child tasks. For example, to execute the `foo` method (which we assume takes some objects as arguments) in 8 tasks in parallel, the following piece of code could be used:

```
for (var i = 0; i < 8; i++)
    this start foo(...);
```

Another option is to start only 7 child tasks and execute one `foo()` call in the parent task, which prevents the overhead of starting one more task, but leads to some code duplication.

This approach of forking off tasks is simple and has straightforward semantics: if starting one of the `foo` tasks interfered with a previously started task, this operation would be blocked by guarding and the execution would be (partially) serialized. This mechanism of "silently" reverting to sequential execution is useful if interference may or may not occur, depending on the input of a program. And at any rate, it is more desirable than the "silent" concurrency bugs that could manifest in manually synchronized languages.

However, often the programmer knows that a certain set of tasks should never interfere (if implemented correctly) and would like to be informed in case this assumption does not hold. In other words, the programmer wants to check for interference *eagerly*, before these tasks are started, and not rely on the lazy checks that are performed by guarding. In addition, eager interference checking can have significant performance benefits: after checking for interference *once* before starting all the tasks, no guarding is necessary inside these tasks anymore. For some programs, this optimization reduces the runtime overhead of Rolez substantially.

Therefore, to give the programmers the option of eager interference checking, Rolez provides two new constructs: the **`parfor`** loop and the **`parallel`**–**`and`** statement.

6.2 Parfor Loop

The **`parfor`** loop is a drop-in replacement for the **`for`** snippet shown above:

```
parfor (var i = 0; i < 8; i++)
    this.foo(...);
```

While this snippet looks similar, the semantics (and performance characteristics) are very different from using a plain **`for`** loop. The execution of a **`parfor`** loop has four stages:

(1) All task arguments are evaluated. The resulting object references are collected and grouped according to the task they are shared with and their declared roles. To do this, the parent task (i.e., the currently executing task) executes the loop sequentially like a normal **`for`** loop, but without actually starting the tasks in the loop body.
(2) The role transitions for all objects are performed. In the same turn, the interference checks take place: if there is a reference that is shared with multiple tasks in a way that could lead to interference, the program terminates with an error that indicates the mistake.
(3) If no interference is detected, then all the tasks are started at once. As an optimization, the task that corresponds to the last loop iteration can be executed in the same underlying system thread as the parent task, to avoid some overhead.

```
1  task sort(b: readwrite Slice[int], a: readwrite Slice[int], begin: int, end: int): {
2      if (begin == end) return;
3      // split and recursively sort both runs from a into b
4      val m = (begin + end) / 2;
5      parallel
6          this.sort(a.slice(begin, m), b.slice(begin, m), begin, m);
7      and
8          this.sort(a.slice(m, end), b.slice(m, end), m, end);
9      // merge the resulting runs from b into a
10     this.merge(b, a, begin, m, end);
11 }
```

Fig. 12. Example usage of the **parallel–and** statement: a (simplified) parallel Mergesort implementation.

(4) The parent task waits for all child tasks to finish and only then continues to execute the code that follows the **parfor** loop.

This execution scheme gives the programmer considerable flexibility: because the "skeleton" of the loop is executed sequentially, there are no restrictions about the loop header. In particular, the number of iterations, i.e., the number of started tasks, does not have to be known at compile time.

6.3 Parallel–And Statement

The second eager checking construct in Rolez is the **parallel–and** statement. Conceptually, it works in the same way as the **parfor** loop, but is tailored towards starting exactly two tasks.

Figure 12 shows how the **parallel–and** statement can be used, in a simplified parallel Mergesort implementation. The `sort` task has two parameters of type `Slice`, which are array slices (see Section 5.3). One contains the data to be sorted, while the other is a work slice in which the data is sorted into. First, `sort` splits the two slices in half and uses the **parallel–and** statement to recursively sort the two sides, in parallel (Lines 5–8). Then, it calls a `merge` method to merge the two sorted sides, effectively sorting the whole range of the slice. Note that, to achieve reasonable performance, sorting should not be parallelized "all the way to the bottom". Instead, tasks should only be started until a desired DOP is reached, from which point on sorting should be done sequentially, to avoid the unnecessary overhead of starting tasks.

The **parallel–and** statement is well-suited for divide-and-conquer-style parallel algorithms. Because both task calls are written explicitly (as opposed to the **parfor** loop, where there is only one call in the code), this statement makes it simple to pass different arguments to each call, as is the case in the Mergesort example, or to call two altogether different tasks.

The execution of the **parallel–and** statement is equivalent to that of the **parfor** loop: (1) Task arguments are evaluated; (2) role transitions and interference checks are performed; (3) the two child tasks are executed in parallel; and (4), after both calls have finished, the parent task continues execution. This means that the **parallel–and** statement, like the **parfor** loop, can have a significant performance advantage when compared to just forking off child tasks: First, no guarding is required inside the child tasks, because interference is checked for beforehand. Second, in many situations, guarding is also not needed for the code that *follows* the **parallel–and** statement (or the **parfor** loop). Given that there are no child tasks *before* the statement, there cannot be any child tasks *afterwards*, because the parent task does not continue to execute until both tasks from the **parallel–and** statement have finished. Thus, if there are no child tasks, no guarding is required, because only child tasks can possibly make an operation in the parent task temporarily illegal. For our Mergesort

Task $T = \langle t, \_, R^{\text{RW}}, R^{\text{RO}}, c\rangle \in Ts$, $c \neq \bullet$ starts n new tasks t_i with role declarations $\{R_i^{\text{RW}}{}^{i\in 1..n}\}, \{R_i^{\text{RO}}{}^{i\in 1..n}\}$, checking that they cannot interfere with each other:

$$\langle H, Ts\rangle \xrightarrow{t \text{ start } \{t_i\}(\{R_i^{\text{RW}}\},\{R_i^{\text{RO}}\})} \begin{cases} S_{\text{error}}, & \text{if } \exists a, b \in 1..n : a \neq b \wedge \exists r : r \in R_a^{\text{RW}} \wedge r \in R_b^{\text{RW}} \cup R_b^{\text{RO}} \\ \langle H', Ts'\rangle, & \text{else, if } \forall i \in 1..n : (\forall r \in R_i^{\text{RW}} : \text{mayWrite}_H(t, r) \wedge \\ & \qquad\qquad \forall r \in R_i^{\text{RO}} : \text{mayRead}_H(t, r)) \\ \langle H, Ts\rangle, & \text{else, if } \forall i \in 1..n : R_i^{\text{RW}} \subseteq R^{\text{RW}} \wedge R_i^{\text{RO}} \subseteq R^{\text{RO}} \cup R^{\text{RW}} \\ S_{\text{error}}, & \text{otherwise} \end{cases},$$

where

$$H' \triangleq H \,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{t :: \text{RO}, t_0 :: \text{RO}\} \mid r \in R_0^{\text{RO}}\} \,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{t :: \text{PU}, t_0 :: \text{RW}\} \mid r \in R_0^{\text{RW}}\}$$
$$\,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{t :: \text{RO}, t_1 :: \text{RO}\} \mid r \in R_1^{\text{RO}}\} \,/\, \{r \mapsto \text{obj}_H(r) \,/\, \{t :: \text{PU}, t_1 :: \text{RW}\} \mid r \in R_1^{\text{RW}}\}$$
$$\,/\, \ldots$$

$$Ts' \triangleq Ts - \{T\} \cup \{\text{step}(T)\} \cup \{\langle t_i, t, R_i^{\text{RW}}, R_i^{\text{RO}}, 0\rangle \mid i \in 1..n\},$$
$$\{t_i{}^{i\in 1..n}\} \subset \textbf{Task ID} - \{t \mid \langle t, \_, \_, \_, \_\rangle \in Ts\}$$

Fig. 13. Eager checking extension for the state transition relation in Figures 5 and 6. The part that corresponds to the eager interference checks is highlighted.

example, this means that the whole `merge` method can be executed without guarding, reducing the runtime overhead substantially.

6.4 Formal Definition

To define the **`parfor`** and **`parallel-and`** constructs formally, we extend the definition of the state transition relation $\longrightarrow$ from Figures 5 and 6 in Section 3 with a new version of the start operation. This version starts multiple new tasks at once and checks that they cannot interfere, by comparing their role declarations.

The new start operation is given in Figure 13; it is defined in a similar way as the original start operation: There are different possible outcomes, depending on the current and declared roles of the objects in the R_i^{RW} and R_i^{RO} sets of the new tasks. However, there is one new case, which is on the very top: if the role declarations of any two tasks may cause interference, then the start operation immediately results in S_{error}, meaning that the program terminates with an error. This is the case if one task t_a declares a reference r as READWRITE while another task t_b declares the same r as READWRITE or READONLY. Note that the definition of the start operation in Figure 13 is actually a generalization of the earlier one: for the case $n = 1$, the two definitions are equivalent.

Also note that the new definition fails to capture one aspect of the informal descriptions of **`parfor`** and **`parallel-and`** above: in the formal definition, after task t has successfully started all tasks t_i, it may continue to execute other operations, while the informal description states that the parent task waits for all the child tasks to finish first. However, this aspect is only relevant with respect to the guarding optimization, which is why we abstracted it away.

7 EVALUATION

This section presents our evaluation of the role-based parallel OOP approach. It addresses two main questions:

(1) **Expressiveness.** Can a role-based OOP language like Rolez, including the novel concepts like slicing and eager interference, be used to express important parallel algorithms and applications?
(2) **Performance.** How much performance can be gained by using such a language? This question includes two aspects: how much parallelism can be expressed and how much overhead is added by the runtime mechanisms like role transitions and guarding?

To answer these questions, we have created a prototype implementation of the Rolez language and used it to implement 8 parallel programs from various benchmark suites.

7.1 Expressiveness

To understand what kinds of parallel patterns and algorithms can be expressed with Rolez, we implemented the following programs: parallel Quicksort and Mergesort; IDEA encryption and Monte Carlo financial simulation, both adapted from the Parallel Java Grande benchmark suite [Smith et al. 2001]; a k-means clustering algorithm, as in the STAMP Benchmark Suite [Cao Minh et al. 2008]; a simple n-body physics simulation; and two programs based on our own implementation of a ray tracer, one that renders static images (simply called Ray Tracer) and one that renders animated scenes (called Animator). These programs use a range of parallel patterns, which we discuss next.

Divide and Conquer. Both Quicksort and Mergesort are based on the divide-and-conquer pattern, which can be exploited for parallel execution. In Rolez, parallel divide-and-conquer algorithms can be expressed naturally using a combination of slicing and the **`parallel-and`** statement, as shown in Section 6.3. In each recursion step, the data array is divided into two disjoint slices, each of which is then shared as READWRITE with one of the two child tasks.

Data Parallelism. Data parallelism is present in the IDEA program, where each 8-byte block of data can be encrypted independently; in the two ray tracer programs, where each image pixel can be computed independently; in the k-means program during the assignment phase, where each data point is independently assigned to a cluster; and also in the n-body simulation, where the position of each body can be updated independently. In all these programs, the data is arranged in a linear or 2D fashion, both of which can be represented with arrays.

To parallelize these programs with Rolez, the data arrays are partitioned into slices, each of which is passed to a different task as READWRITE. Rolez has built-in support for three different partitioning schemes, all of which are needed to efficiently parallelize some of these programs. The *contiguous* and *striped* schemes have already been discussed in Section 5.3. The former is used for k-means and n-body, where every data point or body involves the same amount of work, while the latter is used for ray tracing, where some pixels can be much more expensive to compute than others, because the rays they correspond to are reflected more often before hitting the background. Thus, to distribute the workload evenly among tasks, striped partitioning is used. The third scheme, *block* partitioning, is similar to the contiguous scheme, but ensures that the size of each partition is a multiple of a programmer-specified block size. This scheme is required for the IDEA program, which encrypts data in 8-byte blocks.

Partially Shared Objects. In some data-parallel programs, a task that processes one object needs to read data from objects that are processed by other tasks. This pattern is still deterministic, as long as the fields that are updated are distinct from those that are only read. This is the case in the n-body simulation, where the step that computes the force that acts on a body and updates that body's velocity depends on all (or most) other bodies' positions.

With class slicing, such programs can be parallelized in Rolez without the need for workarounds like splitting a class into two different classes. The n-body simulation can be implemented in Rolez

by declaring the position and velocity fields in separate class slices, as shown previously in Figure 11. Then, the "velocity" slices of all bodies are partitioned and shared with the tasks as READWRITE, while the "position" slices are shared with them as READONLY.

Read-only Data. Many programs involve data that is read by multiple tasks in parallel, but not modified. In *k*-means, this applies to the cluster centroids, which are read by all tasks during the assignment step. And in the ray tracer programs, it applies to all objects that represent the 3D scene that is rendered. In Rolez, this pattern is supported by means of the READONLY role. In the Animator program, e.g., the root object of the scene is shared with all rendering tasks, which declare it as READONLY. Thanks to joint role transitions (Section 4.2), this causes all objects that belong to the scene to become READONLY in these tasks. Once all rendering tasks have finished, the scene becomes READWRITE again in the main task, which can then modify it to prepare it for the next frame.

Task-local Data. Another common pattern is that every task uses its own private copy of some data to perform a computation. For example, in the Monte Carlo simulation, every task generates random fluctuations for a stock and computes prices and expected return rates based on these. This pattern is straightforward in Rolez: every object that is created inside a task is automatically task-local, as it plays the READWRITE role in that task and the PURE role in all other tasks.

One specific pattern that uses task-local data is parallel reduction. A reduction combines all elements in a data set into a single one, using an associative operator. For example, in the *k*-means algorithm, the data points in a cluster are reduced to a single vector, the new centroid, using vector addition. A reduction can be parallelized by partitioning the data and having each task perform the reduction locally for its own partition. The partial results of all tasks are then reduced to a single element in the main task.

In Rolez, reductions can be expressed naturally. During a local reduction in a child task, all task-local objects are READWRITE inside the child task and PURE for the main task. Then, when a child task finishes, it may return the object that represents its partial result to the main task, where it becomes READWRITE.

Task Parallelism. Finally, another form of parallelism is task parallelism, where tasks execute *different* pieces of code in parallel. Task parallelism is present in the Animator program, where the rendering of one frame can be done in parallel to the encoding of the previous one. Task parallelism is trivially expressed in Rolez: a piece of code that can be executed in parallel to the next one can simply be forked off as a separate task. Guarding ensures that the execution of the second piece of code is blocked as soon as it performs the first interfering operation, thus guaranteeing that the result is equivalent to a sequential execution.

7.2 Experimental Setup

We have implemented a Rolez prototype on top of the Java platform. The runtime system, which performs guarding and the role transitions, is implemented as a Java library. This library also contains implementations of the built-in Rolez classes like `Array` and `Slice`. The compiler is implemented with the Xtext framework [Eclipse Foundation 2006] and transforms Rolez source code into Java source code, inserting role transition and guarding operations as calls to the runtime library where necessary. The generated Java code is compiled using a standard Java compiler and executed on a standard Java Virtual Machine (JVM).

We measured the performance of the programs listed above on a machine with four Intel Xeon E7-4830 processors with a total of 32 cores and 64 GB of main memory, running Ubuntu Linux. As the Java platform we used OpenJDK 7, using the default values for all VM options (heap size, etc.). To minimize warm-up effects from the JIT compiler in the JVM, we executed every program 5 to 10

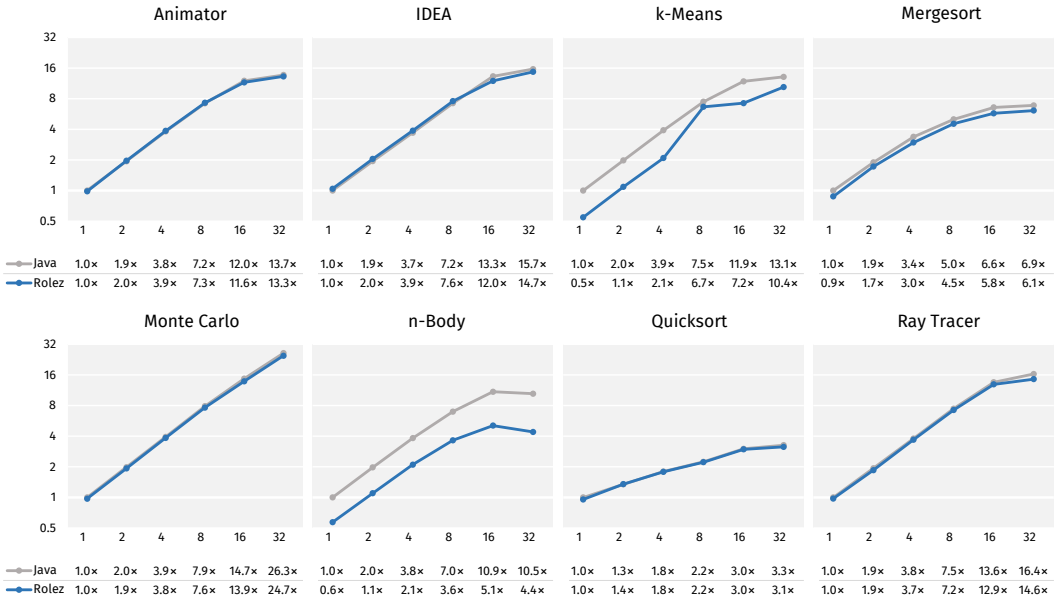


Fig. 14. Comparison of the parallel speedups achieved by the Rolez (Blue) and Java (Gray) implementations, for different numbers of tasks. All speedups are relative to the *single-threaded Java implementations*.

times before measuring. Then, we repeated every experiment 30 times inside the same JVM, taking the arithmetic mean. We also study different input sizes.

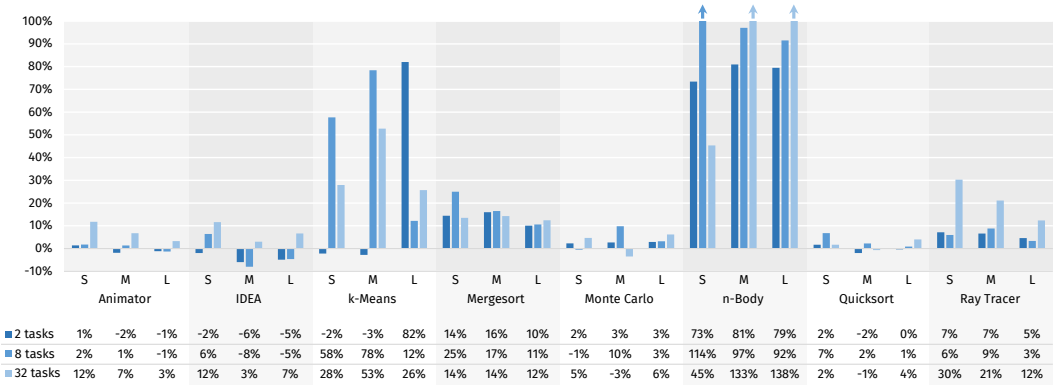
Note that the slice coverage checks, explained in Section 5.3, were turned off for all experiments, to enable a fairer comparison with the manually synchronized Java implementations. These checks are independent of the concept of roles and could just as well make sense in the Java implementations.

The numbers reported here differ from those reported in our previous work [Faes and Gross 2017], for two reasons: First, we use a slightly different experimental setup than before: instead of explicitly configuring the memory management of the JVM, we use the default values here. Especially the maximum heap size influences the performance of some programs, both for the Rolez and the Java implementations. Second, and more importantly, we have since improved the Rolez runtime system and have implemented a range of role-specific optimizations in the Rolez compiler. Thus, the performance of Rolez has improved substantially for some programs.

7.3 Performance

We compare the performance of all Rolez programs to that of equivalent sequential and manually parallelized Java versions. Note that the Rolez programs reuse some Java classes, such as `String`, `System`, and `Math`, which contain native parts, and also classes like `java.util.Random` and `java.util.Scanner`, to avoid the effort of porting these classes to Rolez. We manually ensured that the use of these classes is deterministic.

Parallel Speedup. Figure 14 shows the parallel speedups that the Rolez programs achieve and compares them to those achieved by the Java implementations. Both the Rolez and the Java speedups are relative to the *single-threaded Java implementations*, to show that the performance of Rolez is quite comparable to that of Java. For this experiment, we used the largest input size for all programs. Also, note the logarithmic scale of both axes.



	Animator			IDEA			k-Means			Mergesort			Monte Carlo			n-Body			Quicksort			Ray Tracer		
2 tasks	1%	-2%	-1%	-2%	-6%	-5%	-2%	-3%	82%	14%	16%	10%	2%	3%	3%	73%	81%	79%	2%	-2%	0%	7%	7%	5%
8 tasks	2%	1%	-1%	6%	-8%	-5%	58%	78%	12%	25%	17%	11%	-1%	10%	3%	114%	97%	92%	7%	2%	1%	6%	9%	3%
32 tasks	12%	7%	3%	12%	3%	7%	28%	53%	26%	14%	14%	12%	5%	-3%	6%	45%	133%	138%	2%	-1%	4%	30%	21%	12%

Fig. 15. Runtime overhead of the Rolez implementations compared to the Java implementations, for the same input sizes and numbers of tasks. "S", "M", and "L" stand for the small, medium, and large sizes.

Five out of the eight programs (Animator, IDEA, *k*-means, Monte Carlo, and Ray Tracer) achieve substantial speedups, ranging from 10.4× to 24.7× for 32 tasks. The Rolez implementations of Mergesort and Quicksort achieve slightly less substantial speedups of 3.1–6.1× for 32 tasks, but since the Java implementations perform very similarly, this can be attributed to the limited scalability that is inherent in these programs.

Only for the *n*-body program does the Java implementation achieve substantially higher speedups for all numbers of tasks (about twice as high). This is due to a limitation in the current Rolez prototype, which is related to class slicing: As explained in the previous section, updating a body's velocity requires reading the positions of other bodies, which is possible by separately sharing the "velocity" slices as READWRITE and the "position" slices as READONLY. However, Rolez currently lacks a way to share a collection of object slices without storing them in a new array, which is why the current Rolez implementation of *n*-body exhibits some overhead.

Note that the Rolez *k*-means implementation behaves somewhat irregularly on the machine we used for the evaluation. This is probably due to effects caused by the NUMA (non-uniform memory access) architecture, as we observed a more regular behavior on a different, non-NUMA machine.

Runtime Overhead. To better understand the runtime overhead that is caused by role transitions and guarding, Figure 15 shows the relative execution time differences between the Rolez and the Java implementations, for the same input sizes and for various numbers of tasks. A positive percentage means that the Rolez implementation took longer to execute than the Java version, while a negative one means that the Rolez version took actually less time.

The numbers show that, for the majority of the programs, input sizes, and numbers of tasks, the Rolez overhead is moderate and ranges from 0% to 30%, but mostly stays below 20%. For some configurations, the Rolez versions are even slightly faster: since Rolez objects are larger, due to the fields required for keeping track of the roles, differences in memory allocation and caching may sometimes cancel the Rolez overhead out. We can also see that, for most programs, the input size and the number of tasks have only a small effect on the overhead, which means that Rolez is not only suitable for large-scale parallel applications, but can also be used to speed up smaller computations on personal devices like laptops or smart phones.

As explained before, the Rolez *k*-means implementation behaves somewhat erratically, which leads to large differences in execution time, depending on the input size and numbers of tasks. Further, Figure 15 shows again the issue with the *n*-body implementation, which exhibits an

overhead of 45% to 138%. However, as discussed earlier, even *k*-means and *n*-body still achieve substantial speedups compared to a sequential Java implementation, for large numbers of tasks.

8 RELATED WORK

Many approaches have been proposed to adapt the object-oriented paradigm to the challenges of concurrent systems and, in particular, to parallel programming.

8.1 Concurrent Object-Oriented Programming

A well-known concurrent OOP model is the Actor Model, which was originally proposed by Hewitt et al. [1973] and developed further by Agha [1986, 1990]. Actors are isolated objects that communicate with each other exclusively via asynchronous messages. Due to this strict isolation, low-level concurrency bugs like data races or deadlocks are inherently impossible. The Actor model has been implemented in numerous libraries, such as Akka [Allen 2013] or the Scala Actor Library [Haller and Odersky 2007] for Scala, or Kilim [Srinivasan and Mycroft 2008] for Java. The Erlang language [Armstrong et al. 1996] is also based on actors, even though it uses the name "processes" instead.

A closely-related model, called Active Objects, was introduced by Yonezawa et al. [1986]. In the classic Actor model, messages are processed mostly in a functional style, where an actor effectively *replaces* its behavior with a new one. On the other hand, active objects have their own thread of control and their own local memory, which can be updated imperatively. Some libraries based on active objects are ProActive [Baduel et al. 2006], SALSA [Varela and Agha 2001], and Orleans [Bernstein et al. 2014].

In the Actor and Active Objects Model, the concepts of objects and concurrency are closely linked: objects are not only *used* concurrently, objects *are* the concurrent entities. In contrast, in our model, objects are *aware* of concurrency, but we use the separate concept of tasks. This brings our "concurrency-aware" model closer to the sequential OOP model, and also to mainstream OOP languages like Java, C#, and C++, where concurrent threads can communicate via shared objects. We see this as an opportunity to make concurrency more accessible to non-expert programmers, at least for applications where concurrency is not inherent, but only used for performance reasons. Another advantage of the "concurrency-aware" model is that it can provide strong, system-wide guarantees, such as determinism. In contrast, the isolation provided by actors and active objects only guarantees to prevent low-level bugs like data races and deadlocks, but higher-level race conditions are still possible.

8.2 Deterministic Parallel Programming

While nondeterminism is inherent in *some* concurrent systems, many applications use concurrency merely for parallel execution. In such applications, nondeterminism is almost always undesirable and makes it harder to test and debug them. As a result, many approaches to Deterministic Parallel Programming (DPP) have been proposed.

Functional Approaches. The earliest DPP approaches use functional languages as a basis. For instance, in Multilisp [Baker and Hewitt 1977; Halstead 1985], parallelism is expressed using *futures*, which are basically annotations to evaluate expressions in parallel. Because of the general lack of side effects, the evaluation of one expression does not interfere with the parallel evaluation of any other expression. Another notable example are *I-structures* [Arvind et al. 1989], imperative data structures in an otherwise functional language. Individual parts of I-structures can be written no more than once by a single *producer*, and concurrently read by one or more *consumers*. If a part has not been written to yet, the corresponding read operations will block, which can be seen as a simple form of guarding. I-structures, or *IVars*, have recently appeared in other languages, for example,

in Haskell [Marlow et al. 2011] or, as part of the Concurrent Collections system [Budimlić et al. 2010], in C++ and Java. Recently, *LVars* have been proposed as a generalization of IVars [Kuper and Newton 2013; Kuper et al. 2014b]. LVars allow multiple writes to a particular variable, as long as its state is *monotonically increasing*, with respect to a user-defined lattice. Every variable update ("put") takes the *least upper bound* of the variable's current and new state with respect to the lattice. The *LVish* Haskell library [Kuper et al. 2014a] implements the LVars programming model and extends it with other deterministic parallel patterns, such as atomically incrementing a counter.

Imperative Approaches. Many of these functional approaches include constructs that allow imperative updates, but in entirely imperative languages, determinism is much harder to guarantee, because any piece of code can have *effects* on shared mutable state. If not restricted, the nondeterministic interleaving of such effects leads to nondeterministic results [Lee 2006]. And while an increasing number of mainstream languages now enable functional-*style* programming, e.g., using lambda expressions and the Stream API in Java 8 [Oracle Corporation 2014], these languages are still imperative at their core and cannot guarantee determinism. As an answer to this problem, the *deterministic-by-default* approach for imperative OOP languages has been proposed [Bocchino et al. 2009a; Devietti et al. 2009; Lee 2006; Lu and Scott 2011].

An early DPP language is Jade [Lam and Rinard 1991; Rinard and Lam 1998]. In Jade, the programmer specifies the effects of a task using arbitrary code, which enables the runtime system to check that tasks with interfering effects are not executed concurrently. Though extremely flexible, this approach comes with a substantial drawback: the correctness of effect specifications can only be checked at runtime. Such checks impact performance and may lead to unexpected errors. The same applies to Prometheus [Allen et al. 2009], where the programmer writes code that assigns operations to different *serialization sets*, and to Yada [Gay et al. 2011], where *sharing types* restrict how tasks may access shared data. In contrast, the correctness of role declarations in Rolez can be checked at compile time, using a static type system.

Static Effect Systems. To avoid these problems, *static effect systems* enable checking the correctness of effect specifications at compile time. In fact, these systems typically even check *noninterference* statically, avoiding runtime checks altogether. The effect system used in Deterministic Parallel Java (DPJ) [Bocchino and Adve 2011; Bocchino et al. 2009b] and TweJava [Heumann et al. 2013] brings statically checked effects to OOP languages. To support a wide range of parallel patterns, the effect system includes many features that are relatively complex and are based on the concept of memory regions. Rolez, on the other hand, aims to strike a balance between static guarantees and simplicity: The three roles READWRITE, READONLY, and PURE can be seen as simple, object-oriented effect specifications, designed for compile-time checking of the *correctness* of a task's effect specification, but not for compile-time *noninterference* checking. Instead, noninterference is enforced or checked at runtime, using either guarding or the eager interference constructs (or a mix of both). A Rolez program that uses only guarding is guaranteed to be deterministic and to execute without errors related to parallelism, but tasks are not guaranteed to execute in parallel. On the other hand, with eager interference checking, parallel execution of tasks is guaranteed, but the program may abort with a runtime error in case there are parallel tasks that would interfere.

Other effect systems have been proposed to make parallel programming less error-prone, e.g., by enforcing a locking discipline or by preventing data races or deadlocks [Boyapati et al. 2002; Jacobs et al. 2006]. These systems combine effects with ownership types [Clarke et al. 2001, 1998] and generally couple the regions and effects of an object with those of its owner. This idea resembles our concept of joint role transitions, which can be interpreted as coupling the role of an object with that of its "owners", i.e., the objects that have a reference to it.

Permission Systems. An alternative to effects are systems based on *permissions* [Bierhoff and Aldrich 2007; Boyland 2003; Boyland and Retert 2005]. Permissions accompany object references and define how an object is shared and how it may be accessed. In Æminium [Stork et al. 2009, 2014] for instance, permissions like *unique*, *immutable*, or *shared* keep track of how may references to an object exist and specify the permitted operations. The system then automatically extracts and exploits concurrency. Similarly, the Rust language [Matsakis and Klock 2014] features *mutable* or *immutable* references and guarantees at any time either a single mutable reference or multiple immutable references to an object. Permissions are more object-oriented than effects and conceptually similar to our roles. However, like static effect systems, permission systems aim to guarantee noninterference at compile time, to avoid any runtime checking. This approach may result in more efficient execution and guarantees the absence of runtime errors related to parallelism, but these guarantees are based on restricting aliasing, to specific patterns that can be tracked statically. To enable more complex aliasing and sharing patterns, some permission-based languages include special built-in types that can be used to work around these restrictions. For example, Rust's `std::sync` module [The Rust Project Developers 2011] contains wrappers like `Mutex` and `RwLock` that can be used to concurrently access data in arbitrary patterns. In contrast, the Parallel Roles model sacrifices the ability to guarantee noninterference at compile time, to foster simpler DPP languages like Rolez, which allow arbitrary aliasing. Instead, noninterference is enforced or checked dynamically, leading to a noticeable but modest runtime overhead, as our evaluation shows.

Speculative Execution. Another approach to DPP is *speculative execution*, where the effects of tasks are buffered by a runtime component and rolled back in case they interfere. Two well-known speculative approaches, Thread Level Speculation [Rauchwerger and Padua 1995; Sohi et al. 1995; Steffan and Mowry 1998] and Transactional Memory [Harris and Fraser 2003; Herlihy and Moss 1993; Shavit and Touitou 1995] are not DPP models in a strict sense: the former *automatically* parallelizes sequential programs, while the latter usually provides no determinism guarantees. Actual speculative DPP models are Safe Futures for Java [Welc et al. 2005] and Implicit Parallelism with Ordered Transactions [von Praun et al. 2007]. In both models, the programmer defines which parts of a sequential program should execute asynchronously and the runtime then executes them as speculative tasks, enforcing their sequential order. Speculation, which often comes with a significant overhead due to buffering and rollbacks, is not necessary in the Parallel Roles model, because interfering operations are either delayed by guarding or cause an error (in the case of Rolez, at compile time).

9 CONCLUSION

We have presented an approach to concurrent OOP that demands that objects are *aware* of concurrency through the concept of roles, but in which concurrency is still expressed using the separate concept of tasks. We argue that this "concurrency-aware" paradigm is a viable alternative to "concurrent objects" approaches like the Actor Model. While the latter may be more suitable for inherently concurrent or large-scale systems, our approach has several advantages for deterministic applications: First, because concurrency is expressed using a concept similar to threads, which most programmers should be familiar with, some programmers may find this approach more accessible. Second, because our role-based programming model is deterministic by default, it does not only prevent low-level concurrency bugs like data races but guarantees that any parallel execution produces the same results as a sequential execution, saving programmers from hard-to-debug, high-level concurrency bugs as well. Finally, the "concurrency-aware" approach is based on communication via shared objects, which can be implemented very efficiently on shared-memory architectures. As

our evaluation shows, a role-based language like Rolez can be used to implement a range of parallel programs that achieve almost the same performance as manually synchronized implementations.

A SKETCH OF DETERMINISM PROOF

The Parallel Roles model, as defined in Section 3, guarantees a number of properties, most importantly determinism. Our proof of determinism [Faes and Gross 2018], which we sketch here, uses a property we call *child task priority*. It states the following: From the moment a task τ starts a child task t, τ may not write to any object O as long as t may still read from or write to O, and τ may not read from any object O as long as t may still write to O. This property is expressed more simply using the inverse:

Theorem A.1 (Child Task Priority). *For two tasks, t and t's parent τ, and for all objects O on the current heap, the following* always *holds: If O's roles permit τ to write to O, then O's roles* never *permit t to read from or write to O again, and if O's roles permit τ to read from O, then O's roles* never *permit t to write to O again.*

The formal definition of child task priority involves a significant formal infrastructure, which is outside of the scope of this proof sketch.

Child task priority can be proved using induction over legal program states. These are states that can be reached from the initial state S_0 by a sequence of state transitions, as defined by the state transition relation $\longrightarrow$, and that are not equal to the S_{error} state. For every legal program state, we can show that if the child task priority property holds for that state, then after any of the operations in Figures 5 and 6, the property still holds. In particular, we can show that after a start operation $\xrightarrow{\tau \text{ start } t(R^{\text{RW}}, R^{\text{RO}})}$, task τ may not read from any object in R^{RW} and may not write to any object in R^{RO}, until the newly started task t has finished. This is again shown using induction, over the legal program states that can be reached from the state after the start operation.

From child task priority follows *noninterference*:

Theorem A.2 (Noninterference). *Whenever a task τ starts a task t, all read or write operations in t* happen before *any interfering operation in τ that follows the starting of t.*

More precisely, for every legal state S_{start} that results from a successful task start operation, there cannot be two states S_τ and S_t, where S_τ is reachable from S_{start} and results from a successful read or write operation in τ, and S_t is reachable from S_τ and results from a successful read or write operation in t, and such that these two operations interfere, i.e., they access the same object and at least one of them is a write operation. Noninterference can be shown by proof via contradiction, using child task priority. Again, the formal definition is outside the scope of this sketch.

Finally, noninterference actually implies that all read and write effects in a program *logically* take place as if the program were executed sequentially, i.e., as if every task start operation were replaced by the sequence of operations executed in that started task. Therefore, parallel execution of any program is deterministic, given that the sequential execution of that program is deterministic.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their careful reading of the paper and their detailed comments and suggestions for improvement. We also thank Remi Meier for the countless insightful discussions and Aristeidis Mastoras for helping to prepare the Rolez implementation for the artifact evaluation. Finally, we kindly thank Alain Senn, who wrote the first implementation of the eager interference checking constructs and revised the Rolez programs used in the evaluation.

REFERENCES

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Gul Agha. 1990. Concurrent Object-Oriented Programming. *Commun. ACM* 33, 9 (Sept. 1990), 125–141. https://doi.org/10.1145/83880.84528

Jamie Allen. 2013. *Effective Akka.* O'Reilly and Associates, Sebastopol, Calif.

Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. 2009. Serialization Sets: A Dynamic Dependence-Based Parallel Execution Model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1504176.1504190

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG* (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-Structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632. https://doi.org/10.1145/69558.69562

Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. 2006. Programming, Composing, Deploying for the Grid. In *Grid Computing: Software Environments and Tools*. Springer, London, 205–229. https://doi.org/10.1007/1-84628-339-6_9

Henry C. Baker, Jr. and Carl Hewitt. 1977. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. ACM, New York, 55–59. https://doi.org/10.1145/800228.806932

Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. Microsoft Research. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 301–320. https://doi.org/10.1145/1297027.1297050

Robert L. Bocchino and Vikram S. Adve. 2011. Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP '11)*. Springer-Verlag, Berlin, Heidelberg, 306–332. https://doi.org/10.1007/978-3-642-22655-7_15

Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009a. Parallel Programming Must Be Deterministic by Default. In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar '09)*. USENIX Association, Berkeley. http://dl.acm.org/citation.cfm?id=1855591.1855595

Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009b. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, 97–116. https://doi.org/10.1145/1640089.1640097

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 211–230. https://doi.org/10.1145/582419.582440

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis (SAS '03)*. Springer, Berlin, Heidelberg, 55–72. https://doi.org/10.1007/3-540-44898-5_4

John Tang Boyland and William Retert. 2005. Connecting Effects and Uniqueness with Adoption. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/1040305.1040329

Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. 2010. Concurrent Collections. *Scientific Programming* 18, 3 (Jan. 2010), 203–217. https://doi.org/10.3233/SPR-2011-0305

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of The IEEE International Symposium on Workload Characterization (IISWC '08)*. https://doi.org/10.1109/IISWC.2008.4636089

David G. Clarke, James Noble, and John Potter. 2001. Simple Ownership Types for Object Containment. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer-Verlag, London, UK, 53–76. https://doi.org/10.1007/3-540-45337-7_4

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1508244.1508255

Eclipse Foundation. 2006. Xtext. http://www.eclipse.org/Xtext/

Michael Faes and Thomas R. Gross. 2017. Parallel Roles for Practical Deterministic Parallel Programming. In *Proceedings of the 30th International Workshop on Languages and Compilers for Parallel Computing (LCPC '17)*.

Michael Faes and Thomas R. Gross. 2018. *Formal Proof of Determinism for the Parallel Roles Model.* Technical Report. Department of Computer Science, ETH Zurich, Zürich, Switzerland. https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Publications/parallel-roles-determinism-proof.pdf

David Gay, Joel Galenson, Mayur Naik, and Kathy Yelick. 2011. Yada: Straightforward Parallel Programming. *Parallel Comput.* 37, 9 (Sept. 2011), 592–609. https://doi.org/10.1016/j.parco.2011.02.005

Philipp Haller and Martin Odersky. 2007. Actors That Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION'07)*. Springer-Verlag, Berlin, Heidelberg, 171–190. http://dl.acm.org/citation.cfm?id=1764606.1764620

Robert H. Halstead, Jr. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

Tim Harris and Keir Fraser. 2003. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '03)*. ACM, New York, 388–402. https://doi.org/10.1145/949305.949340

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, 289–300. https://doi.org/10.1145/165123.165164

Stephen T. Heumann, Vikram S. Adve, and Shengjie Wang. 2013. The Tasks with Effects Model for Safe Concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, 239–250. https://doi.org/10.1145/2442516.2442540

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)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245. http://dl.acm.org/citation.cfm?id=1624775.1624804

Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. 2006. A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering (ICFEM '06)*. Springer-Verlag, Berlin, Heidelberg, 420–439. https://doi.org/10.1007/11901433_23

Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-Based Data Structures for Deterministic Parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC '13)*. ACM, New York, 71–84. https://doi.org/10.1145/2502323.2502326

Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014a. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, 2–14. https://doi.org/10.1145/2594291.2594312

Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014b. Freeze After Writing: Quasi-Deterministic Parallel Programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, 257–270. https://doi.org/10.1145/2535838.2535842

Monica S. Lam and Martin C. Rinard. 1991. Coarse-Grain Parallel Programming in Jade. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '91)*. ACM, New York, 94–105. https://doi.org/10.1145/109625.109636

Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (May 2006), 33–42. https://doi.org/10.1109/MC.2006.180

Li Lu and Michael L. Scott. 2011. Toward a Formal Semantic Framework for Deterministic Parallel Programming. In *Proceedings of the 25th International Conference on Distributed Computing (DISC '11)*. Springer-Verlag, Berlin, Heidelberg, 460–474. https://doi.org/10.1007/978-3-642-24100-0_43

Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, 71–82. https://doi.org/10.1145/2034675.2034685

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Oracle Corporation. 2014. API Reference of the Java.Util.Stream Package (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Lawrence Rauchwerger and David Padua. 1995. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, 218–232. https://doi.org/10.1145/207110.207148

Martin C. Rinard and Monica S. Lam. 1998. The Design, Implementation, and Evaluation of Jade. *ACM Trans. Program. Lang. Syst.* 20, 3 (May 1998), 483–545. https://doi.org/10.1145/291889.291893

Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, 204–213. https://doi.org/10.1145/224964.224987

L. A. Smith, J. M. Bull, and J. Obdrzálek. 2001. A Parallel Java Grande Benchmark Suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*. ACM, New York, NY, USA, 8–8. https://doi.org/10.1145/582034.582042

Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, 414–425. https://doi.org/10.1145/223982.224451

Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP '08)*. Springer-Verlag, Berlin, Heidelberg, 104–128. https://doi.org/10.1007/978-3-540-70592-5_6

J. Steffan and T Mowry. 1998. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA '98)*. IEEE Computer Society, Washington, DC, 2–13. https://doi.org/10.1109/HPCA.1998.650541

Sven Stork, Paulo Marques, and Jonathan Aldrich. 2009. Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 933–940. https://doi.org/10.1145/1639950.1640060

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. 2014. Æminium: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Trans. Program. Lang. Syst.* 36, 1 (March 2014), 2:1–2:42. https://doi.org/10.1145/2543920

The Rust Project Developers. 2011. Documentation of the Std::Sync Rust Module. https://doc.rust-lang.org/std/sync/

Carlos Varela and Gul Agha. 2001. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Not.* 36, 12 (Dec. 2001), 20–34. https://doi.org/10.1145/583960.583964

Christoph von Praun, Luis Ceze, and Calin Caşcaval. 2007. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM, New York, 79–89. https://doi.org/10.1145/1229428.1229443

Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe Futures for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, 439–453. https://doi.org/10.1145/1094811.1094845

Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. 1986. Object-Oriented Concurrent Programming in ABCL/1. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*. ACM, New York, NY, USA, 258–268. https://doi.org/10.1145/28697.28722