Parallel Roles for Practical Deterministic Parallel Programming

Michael Faes(✉) and Thomas R. Gross(✉)

Department of Computer Science, ETH Zurich, Zürich, Switzerland
{mfaes,trg}@inf.ethz.ch

Abstract. Deterministic parallel programming languages are attractive as they do not allow data races, deadlocks, or similar kinds of concurrency bugs that are caused by unintended (or poorly understood) parallel execution. We present here a simple programming model for deterministic parallel programming that is based on *roles*. The programmer specifies the role that an object plays for a task (e.g., the READONLY role), and compiler and runtime system *together* ensure that only those object accesses are performed that are allowed by this role. An object may play different roles in the course of a program's execution, giving the programmer considerable flexibility in expressing a parallel program.

The model has been implemented in a Java-like language with references and object sharing. Preliminary results indicate that the runtime overhead is moderate (compared to standard Java programs), and that the compiled programs achieve substantial parallel speedups.

1 Introduction

Deterministic parallel programming languages avoid bugs caused by the unintended or poorly understood parallel execution of programs. These languages attempt to make concurrency bugs impossible by design [5,23,24,37,38].

Recently, several projects proposed static effect systems to support deterministic parallel programming (DPP) for imperative and object-oriented languages [6,18,20,25]. In such systems, the programmer declares the side effects of tasks and methods by indicating the *memory regions* that are read or modified. These effect specifications are then used by the compiler or the runtime system to check that tasks with interfering effects are not executed in parallel.

Memory regions as used in effect systems may allow a precise description of which memory locations are read or modified by a program unit. However, object-oriented programs are not structured (or documented) based on memory locations but instead use *objects* as the unit of reasoning. Memory locations provide little abstraction and are at too low a level. Since objects are the foundation of object-oriented programs, our approach to DPP is based on objects. The first idea is to leverage the concept of *roles*, which have a long-standing tradition in sequential object-oriented programming and modeling, where they are used to characterize the different "roles" an object may assume when collaborating with

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 163–181, 2019.
https://doi.org/10.1007/978-3-030-35225-7_12

other objects [15,21,28,30,31]. Our work builds on this foundation and uses roles as the key abstraction to specify and reason about parallelism. Together with the concept of *role transitions*, roles form the basis for a new object-oriented DPP model.

In this model, every object plays a role in each task, and these roles change dynamically when tasks start or finish. Because the role of an object defines the legal interactions with that object, roles provide a concise way to reason about, document, and specify the effects of concurrent tasks. In contrast to effect systems, the model does not focus on pieces of code and their effects on memory regions; instead, it focuses on objects and the roles they play in parallel – hence the name *Parallel Roles*. By employing a specific set of roles and *role transition rules*, the model guarantees that tasks do not interfere. Noninterference is not checked at compile time or before a task is started, like in effect systems; instead, it is enforced *during* the execution of tasks. However, unlike in speculative systems, noninterference is enforced deterministically and without rollback.

This dynamic approach makes it possible to design DPP languages with simple program annotations, without the need for special syntactic constructs for parallel execution, and without any kind of aliasing restriction. To illustrate these points, we give an overview of a roles-based, Java-like language we call *Rolez*. This language enables programmers to parallelize a program by simply marking a method as a "task" and declaring one of three possible roles for its parameters: READWRITE, READONLY, or PURE. When a task is invoked, it is executed in parallel to the invoking code, while the runtime system prevents the two concurrent parts of the program from interfering, based on the declared roles.

Figure 1 illustrates the simplicity of Rolez in a snippet of an encryption program we use in our evaluation. The encryption scheme is block-based, so different parts of the data can be encrypted in parallel. Note that for the sake of clarity, some annotations are left out; Sect. 3.2 explains what additional annotations are required. The `encrypt` task has two main parameters: `src` and `dst`, both of type `Array`. The task declares the READONLY role for the `src` array, which the task only *reads*, and the READWRITE role for the `dst` array, to which the task *writes* the encrypted data. In addition, the `encrypt` task has a parameter that defines the part of the `src` array that should be encrypted. The `parallelEncrypt` method achieves parallelism by creating multiple destination arrays and starting a separate `encrypt` task for each of them. Noninterference is guaranteed in two ways: First, the `plaintext` array plays the READONLY role in all tasks, which means that it cannot be modified by any of them. Second, every

```
def parallelEncrypt(plaintext: Array[byte],
                    tasks: int): Array[byte] {
  val encrypted: Array[Array[byte]] = ...
  for(var i = 0; i < tasks; i++)
    start encrypt(plaintext, encrypted.get(i), i);
  return merge(encrypted);
}

task encrypt(src: readonly  Array[byte],
             dst: readwrite Array[byte],
             partition: int): void {
  ...
}
```

Fig. 1. Rolez example. The role declarations are highlighted in green and orange. (Color figure online)

task writes to a separate destination array. In terms of roles, a destination array that plays the READWRITE role in one task plays the PURE role in all other tasks (including the parent task), meaning that it is inaccessible. However, as soon as all tasks have finished, all destination arrays are READWRITE again in the parent task, so they can be merged into a single array. When the `merge` method in the parent task tries to read from the destination arrays, it is automatically blocked until all `encrypt` tasks have finished.

To demonstrate the viability of roles-based languages, we implemented a prototype compiler and runtime system for Rolez and use a suite of parallel programs to assess its effectiveness. These programs contain a range of parallel patterns that are expressible with the three mentioned roles. All programs achieve substantial speedups over a sequential Java version and exhibit a reasonable runtime overhead compared to a manually parallelized Java programs.

To summarize, the key contributions of this paper are the following:

1. an object-oriented parallel programming model, based on three roles: READWRITE, READONLY, and PURE that guarantee determinism (Sect. 2);
2. an overview of the design of Rolez, a roles-based, Java-like DPP language that requires only simple *role declarations* from a programmer (Sect. 3);
3. a preliminary evaluation of the Rolez prototype for 4 parallel programs. Rolez can express many parallel patterns found in these programs and achieves substantial speedups over sequential Java for most of them (Sect. 4).

2 The Parallel Roles Model

This section presents the Parallel Roles programming model. We first present a simple core version for single objects and then extend it to cover object graphs.

2.1 Core Parallel Roles

The main idea behind Parallel Roles is to use the *object*, the key concept of object-oriented programming (OOP) as the basis to reason about concurrent effects and parallelism. In the standard OOP model an object is a collection of fields, which contain the object's state, plus a collection of methods, which define the object's functionality. In the Parallel Roles model, every object has a third component: the *roles* it currently plays for the different *tasks* in the program. This is illustrated in Fig. 2.

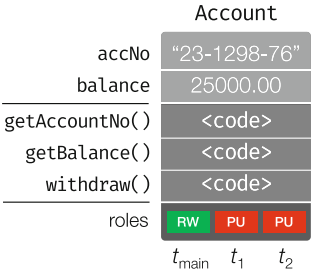

Fig. 2. The components of an object: fields, methods, roles

The fields and methods of an object define the object's *sequential behavior*. That is, they define how the object behaves when other objects interact with it in a single task. On the other hand, the roles of an object define the object's *concurrent behavior*. Specifically, they define which interactions are legal in which

tasks and what happens when an illegal interaction occurs. Like the content of an object's fields, the roles an object plays may change over time. However, in contrast to the fields' contents, which (in general) can be modified arbitrarily, the changing of roles follows strict rules. These *role transition rules* restrict the combinations of roles an object may play in different tasks at the same time. Those restrictions in turn guarantee noninterference and, by extension, determinism. In the following paragraphs, we explain these core concepts, roles, tasks, and role transitions, in more detail.

Roles. The role of an object defines how other objects may interact with that object, i.e., which kinds of field operations they may perform and, by extension, which methods they may invoke. There are three roles: READWRITE, READONLY, and PURE. READWRITE permits both field read and field write operations, while READONLY permits only read operations. PURE permits neither, except if a field is final (i.e., it cannot be modified, as in Java); then it may be read. Final fields are treated specially because they can never be the source of interference. Figure 3 summarizes these rules.

	PURE	READONLY	READWRITE
final field read	✓	✓	✓
any field read	✗	✓	✓
field write	✗	✗	✓

Fig. 3. Operations permitted by the roles

The set of permitted field operations also defines the set of permitted methods. READWRITE permits calls to any method, while READONLY permits only calls to methods that do not modify the target object. PURE permits only calls to *pure methods*, which are the object-oriented counterpart of a *pure function*: They are side-effect free (i.e., they do not write to any of the target object's fields) and their result is always the same, given the same target object (i.e., they do not read any of the target object's non-final fields). As an example, PURE for some `Account` object would only permit calls to `getAccountNo()` (assuming account numbers are immutable), READONLY would also permit calls to `getBalance()`, and READWRITE would permit calls to all methods, including `withdraw()`.

Tasks and Role Declarations. Tasks are execution contexts, like threads. When the execution of a program begins, all objects interact with each other in the *main task*. A task may start other tasks (called *child tasks*) and thereby create multiple concurrent execution contexts. While tasks are similar to threads, there is a key difference: When defining a task, the programmer needs to declare the role that each object is supposed to play in that task. With these *role declarations*, the programmer controls the role transitions that objects perform, as described next.

Role Transitions. As mentioned earlier, there are rules about when and how the roles of an object change, i.e., when and how an object performs a role transition. Most importantly, role transitions only take place when a task starts or finishes. When a new task starts, every object for which the task declares a role performs a role transition such that its role in that task matches the declared one. Hence,

at the beginning of a task, every object plays the declared role in that task. However, 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 READWRITE role for an object. In such a case, the object becomes PURE in the parent task, to prevent interference. Therefore, while an object is guaranteed to play the declared role at the *beginning* of a task, a role declaration does not state that the object plays this role for the whole duration of the task. What a role declaration *does* state is that the object may never play a *more permissive* role than the one declared, in either that task itself or any task that is (transitively) started by it. That is, an object may never play a role that permits an operation the declared role does not permit. For example, if the declared role of an object is READONLY, this object can never play the READWRITE role in that task, since READWRITE is more permissive than READONLY.

1. When an object is created in a task t, it plays the READWRITE role in t and the PURE role in all other tasks.
2. When an object o plays the READWRITE role in a task t, t may share o with a new task t_n that declares the READWRITE role for o. When t_n is started, o becomes READWRITE in t_n and PURE in t.
3. When an object o plays the READWRITE or READONLY role in a task t, t may share o with a new task t_n that declares the READONLY role for o. When t_n is started, o becomes READONLY in t and t_n.
4. Any object o may be shared with a task t that declares the PURE role for it. No transition takes place for o when t is started.
5. When a task t that declared the READWRITE role for an object o is about to finish, t waits until o is READWRITE; then, t finishes and o becomes READWRITE in t's parent task.
6. When a task t that declared the READONLY role for an object o is about to finish, t waits until o is READONLY. After t has finished, if t's parent t_p is the only task left in which o is READONLY, o becomes READWRITE in t_p. Otherwise, o stays READONLY in t_p.
7. When a task that declared an object o as PURE finishes, o performs no role transition.
8. When a task t is about to finish, t waits until every object o created in t (or a child of t) plays the READWRITE role; then, t finishes and o becomes READWRITE in t's parent t_p (if t_p may reach o via some reference).

Fig. 4. The core role transition rules

The rules in Fig. 4 define when and how the roles of an object can change. As we explain shortly, these rules are designed such that they guarantee noninterference for every object. Rule 1 concerns newly created objects, while Rules 2–4 concern the starting of tasks and Rules 5–8 the finishing of tasks.

Figure 5 illustrates these rules by showing a series of role transitions an object can go through. Initially, when the object is created in task t_1, it is READWRITE in t_1 and PURE in the t_{main} task. It is then shared with two tasks: t_2, which declares it as READWRITE, and later t_3, which declares it as READONLY. When t_2 and t_3 start and finish, the object performs a role transition. After t_3 has finished, it is again READWRITE in t_1. Finally, t_1 finishes and the object becomes READWRITE in t_{main}.

Guarding. An object may never play a role that is more permissive than the role declared in a given task. However, the object may temporarily play a *less* permissive role. When this happens, some operations may become illegal, despite being

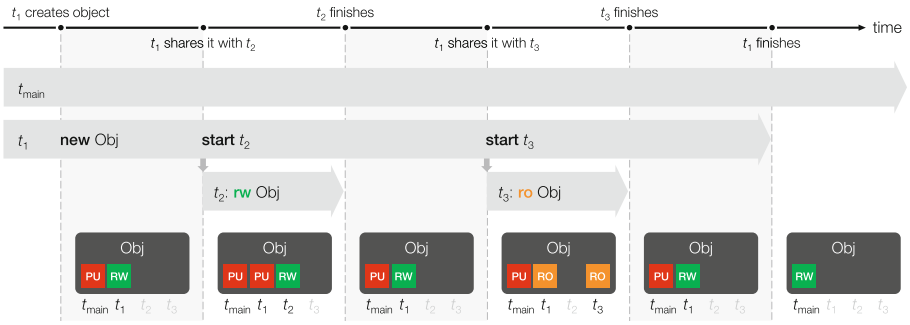

Fig. 5. Illustration of the role transition rules for an object. The gray arrows from the left to the right are tasks, the black boxes represent the same object in different points in time, and the small colored boxes show the roles the object plays in each task. (Color figure online)

legal under the declared role. For example, if an object is declared READWRITE in a task, it might play the READONLY role for some time, because it was shared with another task. This discrepancy between declared and current role is the subject of *guarding*. The idea of guarding is to wait until the current role equals the declared role: When an operation is performed that is legal under the declared but not under the current role of the target object, this operation is not an error but instead is blocked until the object plays its declared role again.

We illustrate guarding with a simplified Rolez snippet (from a program we later use for the evaluation) and a corresponding illustration, in Fig. 6. This program renders animated 3D scenes and encodes the rendered images as frames in a video file. The main loop consists of three steps: First, the scene is rendered for a fixed point in (animation) time, then the resulting image is encoded as a video frame, and finally an animation step is performed to update the scene for the next frame. The encoding and the animation step can be done in parallel, which is why `encode` is declared and invoked as a task. Because `encode` only needs to *read* the image, it declares it as READONLY. When the `encode` task starts, the image performs a role transition and becomes READONLY also in the "main" task. While `animateStep` does not modify the image, the rendering in the next iteration does. In case the `render` method begins execution before the `encode` task has finished, guarding blocks the execution of `render` to prevent it from interfering with the encoding. Once `encode` finishes, the `render` method resumes execution. Note that in the version of the program used for the evaluation, *two* image buffers are used, to enable the `encode` task to also execute in parallel to `render`.

Properties. We now examine the properties of Parallel Roles. First of all, the transition rules ensure the *soundness of role declarations*, i.e., that no object may play a more permissive role than its declared role in both the task that declared it and any task it (transitively) starts. This follows from two observations: First,

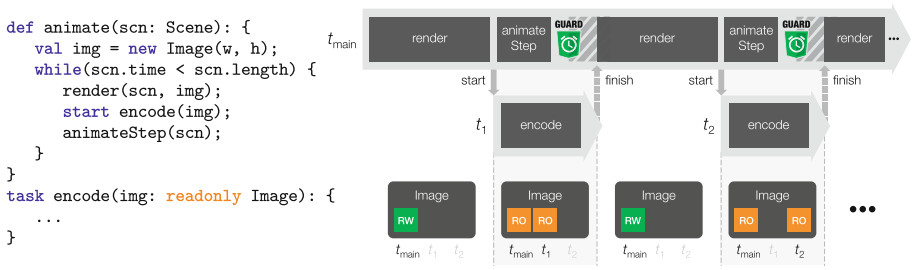

Fig. 6. Guarding example. The left side shows (simplified) Rolez code and the right side illustrates how guarding prevents `encode` from interfering with the `render` method.

no transition rule permits an object with a declared role to play a role it has not played before in a given task. And second, none of the rules permit an object to be shared with a task that declares a more permissive role than the object currently plays. Note that Rule 8 *does* permit objects to play a more permissive role (READWRITE) in the parent task than before (PURE), but since these objects were newly created, they do not have a declared role in the parent task.

Second, the transition rules guarantee that no object ever plays the READWRITE role in one task while it plays the READWRITE or the READONLY role in another task. We call this property *exclusiveness of* READWRITE and we show it using induction: When an object is created, it is READWRITE for the creator task and PURE for all other tasks (Rule 1). This is the base case. For the inductive step, we assume the object is either READWRITE in a single task or READONLY in a number of tasks, but in both cases PURE for all other tasks. After any start transition (Rules 2 or 3), this rule still holds. After any transition at the end of a task (Rules 5, 6, or 8), the condition also still holds. In particular, Rule 6 ensures that an object that is READONLY in any task can only become READWRITE again once there is no task left in which it is READONLY. Therefore, no series of transitions may ever violate the exclusiveness of READWRITE.

Exclusiveness of READWRITE, combined with guarding and the definitions of permitted operations in Fig. 3, implies that if an object can be modified in one task, then the mutable parts of it cannot be accessed by any other task until the modifying task has finished. Thus, the model guarantees *noninterference*. Note that two mechanisms to prevent interference are combined: (i) An operation that is illegal with respect to the *declared* role of an object results in an error. This could be a runtime or a compile-time error, depending on the language. (ii) An operation that is illegal with respect to the *current* role of an object, but not with respect to its declared role, is blocked by guarding until the object plays a role under which the operation is legal.

Note that noninterference is much stricter than data race freedom. Since the exclusiveness of READWRITE holds for all objects in the program, no modification of a task t can be observed by any other task, as long as t is running. Therefore, tasks cannot communicate, except for passing arguments and waiting for each

other's results. This restriction is the key to guarantee determinism. However, Parallel Roles could be extended with nondeterministic roles to enable inter-task communication for parallel applications that profit from nondeterminism.

Since noninterference is achieved in part by blocking the execution of operations, it may seem like the model is prone to deadlock. However, this is not the case: Whenever an operation is blocked in a task t_1, it is because the target object currently plays a less permissive role than its declared role. This can only be the case if t_1 shared the object with another task t_2. Since objects can only be shared when a task is started, t_2 must be a child task of t_1. Therefore, tasks can be blocked only by child tasks, and this property precludes cyclic dependences. Thus, Parallel Roles not only guarantees noninterference, but also *deadlock freedom*. Together, these two properties imply that Parallel Roles guarantees determinism.

To summarize, Parallel Roles combines roles, which determine the legal operations for an object, with transition rules, which determine the possible combinations of roles an object may play in parallel. Tasks are prevented from interfering using a combination of runtime or compile-time checking and guarding.

2.2 Object Graphs

A shortcoming of the transition rules presented so far is that they do not consider objects with references to other objects. That is, they do not define what happens to objects that are *reachable* from an object that performs a role transition.

A safe but impractical definition would be that objects are simply unaffected by the role transitions of their referrers. However, with such a definition, an object could easily break when shared with another task, because objects it depends on would play a different role than itself. For example, consider a `Bank` object, which contains references to all `Accounts` of that bank. The `Bank` has a method `payInterest`, which computes and deposits the yearly interest for each of its accounts. If such a `Bank` object was shared with a task t that declares it as READWRITE, calling the `payInterest` method in t would fail, since all of its `Account` objects would be PURE and their balance could not be accessed in t.

We employ a practical, but simple and safe way to handle object graphs. Expressed as two additional role transition rule, it states:

9. Whenever an object o is about to perform a role transition, all objects that are reachable from o perform the same transition. The transitions only take place once all these objects play one of the roles o is required to play. The implicitly declared role of these objects is the same as for o. In case an object is reachable from multiple objects that perform different role transitions at the start of a task, that object performs the transition that makes it play the most permissive role in the new task.
10. When a task t that declared the READWRITE role for an object o is about to finish, t waits until all objects *that were reachable from* o *w*hen t started are READWRITE. Then, t finishes and all these objects become READWRITE in t's parent task.

With Rule 9, when an object is shared with a task, the task will not start until that object *and all objects that are reachable from it* play the required role. For example, when a `Bank` object is shared with a task that declares it as READWRITE, not only the `Bank` itself, but also all of its `Accounts` must play the READWRITE role before the task may start. Once they do, all these objects perform a transition and become READWRITE for the new task. Now, `payInterest` can be successfully invoked in that task, because all required objects play the READWRITE role.

Finally, Rule 10 concerns object graphs that are shared with a task that unlinks some objects in the graph. Since these objects may still be used in the parent task later, they also revert to their previous roles once the task finishes.

3 Rolez Language Overview

This section gives an informal description of a concrete programming language, *Rolez*, which implements the Parallel Roles model presented in the previous section. It is a Java-like language with a roles-based type system.

```
1  class App {
2     def pure calcInterest(balance: int): int {
3        return (the Consts.intrstRate * 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    def readwrite deposit(i: int): {
18       this.balance += i;
19    }
20    def readwrite withdraw(i: int): {
21       this.balance -= i;
22    }
23    def readonly getBalance: int {
24       return this.balance;
25    }
26 }
27 object Consts {
28    val intrstRate: double = 0.015
29 }
```

Fig. 7. Rolez code example for tasks, role declarations, and global singleton objects

3.1 Tasks and Role Declarations

Declaring and Starting Tasks. In Rolez, tasks 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. Note that `void` return types can be omitted.

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 a single object to be shared with it, namely an `Account` object that plays the READWRITE role. The parameter is declared as READWRITE because the `payInterest` modifies the balance of the given account when calling `deposit` on Line 7. So 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.

Incidentally, both the `payInterest` and the `main` task have another parameter: the "`this`". The role for "`this`" is declared right after the `task` keyword and is PURE for both of these tasks. This means that the `App` instance does not perform any role transition (see Rule 4). This instance is created implicitly before the program starts and is the target (the "`this`") of both task start invocations (including the implicit start of the `main` task at the start of the program execution).

Note that, in Rolez, not only task parameters but also method parameters and other constructs have role declarations. Section 3.2 elaborates these aspects.

Global Objects. How can Rolez guarantee that only objects that have been shared with a task are accessed in that task? Simply, a task can only access objects that were passed to it as arguments (including "`this`"), or that are reachable from such. (As per Rule 9, such reachable objects perform the same transitions as their referrers and implicitly have the same declared role.) That is, no objects can be globally accessed in Rolez, in contrast to, e.g., objects in static fields in Java.

However, there is one exception: A programmer may define global singleton objects, using the `object` keyword instead of `class`. To prevent tasks from interfering when they access such global objects, these objects are immutable. In other words, they are (conceptually) initialized at the beginning of the program and then they permanently play the READONLY role for all tasks. An example for the declaration of such a singleton is shown in Fig. 7 on Lines 27 to 29, while Line 3 shows how this singleton is accessed using the keyword "`the`".

3.2 Role Type System

Rolez uses a static type system to report erroneous operations at compile time. Recall that there are two kinds of *illegal* operations with regard to roles, only one of which is considered *erroneous*. The first kind is a *temporarily illegal* operation, which is illegal only with respect to an object's *current* role. Such an operation is not considered an error, but is delayed until it becomes legal, using guarding. The second kind of an illegal operation is illegal with respect to an object's *declared* role. Such an operation can never become legal and must be reported as a *role error*. In Rolez, role errors are reported at compile-time, using a roles-based type system. In this section, we give a brief, informal overview of this type system.

Note that the Rolez type system does not guarantee noninterference on its own, unlike static effect systems. Only in combination with guarding can Rolez guarantee that tasks do not interfere. Thus, the Rolez type system is much less complex than static effect systems or permission-based type systems (see Sect. 5) and does not, e.g., impose any aliasing restrictions.

Role Types. The Rolez type system is an extension of the class-based type system known from Java and other OOP languages. Every variable in such a language has a type that corresponds to a class. A sound type system guarantees that, at runtime, a variable always refers to an object that is an instance of the class that corresponds to the variable's type (or a subclass thereof). Therefore, when accessing a field or calling a method on a variable, the compiler can check whether this member exists in that class, or else report a type error. Likewise, by including an object's declared role in the static type of variables that refer to that object, the Rolez type system enables the compiler to report role errors.

A static type in Rolez, called a *role type*, consists of two parts, the *class part* and the *static role*. The class part corresponds to the class of an object, while the static role corresponds to the declared role of an object in the currently executing task. An example for a role type is `readwrite Account`, where `readwrite` is the static role and `Account` is the class part.

```
class Object {
    ...
}
class Account extends Object {
    ...
}
class SavingsAccount extends Account {
    ...
}
```

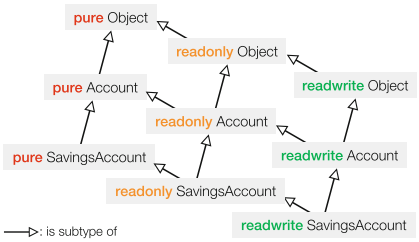

Fig. 8. Rolez type hierarchy example: source code and corresponding type hierarchy

In Java-like languages, a variable may not only refer to instances of the very class that corresponds to the variable's type, but also to instances of subclasses thereof. In Rolez, the same applies to the static role: A variable may refer to objects whose declared role is a *subrole* of the variable's static role. A role is a subrole of another role if it is the same or a more permissive role. Hence, subtyping applies to both the class part and the static role.

Figure 8 illustrates the subtype relation with an example consisting of three classes. In Java, this code would lead to a type hierarchy with a linear structure and three types that correspond to the three classes. On the other hand, in Rolez the code results in a lattice containing nine role types that correspond to all possible combinations of roles and classes.

Type Declarations and Type Checks. In Rolez, like in other languages with a static type system, all local variables, parameters, fields, and methods need a type declaration, in general. However, Fig. 7 shows that *type inference* is applied to local variables to reduce the programmer's annotation burden. If a variable is assigned right when it is declared, the variable's type is inferred from the right-hand side of the assignment (Lines 6 and 10). For method parameters, type inference is not possible under modular compilation, therefore types must be fully declared. This is true also for the "`this`" parameter of methods (and tasks), although the class part of the type is implicit, because it corresponds to the method's class. The role part is still necessary though (Lines 17, 20, and 23). These type declarations are used by the compiler to perform type checks, with the ultimate purpose of preventing operations that are not permitted under the declared role of an object.

Most type checks in Rolez are standard, like "the right-hand side type of an assignment must be a subtype of the left-hand side type". The roles-specific checks concern field accesses. A field may only be read if the target's role is "at least" `readonly` (or if the field is final). Likewise, a field may only be written to if the target is `readwrite`. Another difference between the field access rules in Rolez and other OOP languages is that the type of a field read expression depends on the role of the target expression, and is not simply the declared type of the field. The reason for this difference is the object graphs extension introduced in Sect. 2.2. With this extension, the declared role of an object that is reachable from a task parameter corresponds to the declared role of that parameter. To reflect this in the type system, the role of a field-read expression must always be a superrole (the same or a less permissive role) of that of the target expression.

```
1  task pure getOwnerName(a: readonly Account): pure String {
2      val owner: readonly Client = a.owner;
3      return owner.name;
4  }
```

```
5  class Account {
6      var owner: readwrite Client
7      ...
8  }
```

Fig. 9. Rolez example to illustrate the field-read type check

The example in Fig. 9 illustrates how this last rule ensures that the static role of an object that is reachable from a task parameter is always a superrole of that object's implicitly declared role. The `getOwnerName` task declares an `Account` parameter with the `readonly` role. When an `Account` object is shared with this task, it becomes READONLY, like the `Client` object that the `owner` field on Line 6 refers to. When this field is read on Line 2, the role of the `a.owner` expression is `readonly`, even though the type of the `owner` field is `readwrite Client`. Therefore, this expression can only be assigned to a variable of type `readonly Client`, making sure that the `Client` object's implicitly declared READONLY role is respected.

4 Evaluation

In this section, we present a preliminary evaluation of the Rolez language that shows that (i) parallel programs for non-trivial problems can be written in Rolez,

and (ii) parallel Rolez programs realize a speedup over both sequential Rolez and Java programs, despite the runtime overhead of role transitions and guarding.

4.1 Experimental Setup

We implemented a Rolez prototype, i.e., a compiler and a runtime system, on top of the Java platform. The runtime system is implemented as a Java library, while the compiler, implemented with Xtext [1], transforms Rolez source code into Java source code, inserting role transition and guarding operations as method calls to the runtime library where necessary. The generated code is compiled using a standard Java compiler and executed on a standard Java Virtual Machine (JVM).

The following programs were implemented in Rolez: IDEA encryption and Monte Carlo financial simulation, both adapted from the Parallel Java Grande benchmark suite [34]; a k-means clustering algorithm, as in the STAMP Benchmark Suite [10]; and a ray tracer that renders animated scenes (called animator). These programs contain the following parallel patterns, all of which can be expressed in Rolez: data parallelism, task parallelism, read-only data, and task-local data.

We measured the performance of each program 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. To eliminate warm-up effects from the JIT compiler in the JVM, we executed every program 5 to 10 times before measuring. Then we repeated every experiment 30 times inside the same JVM, taking the arithmetic mean.

Fig. 10. Speedup of parallel Rolez programs, compared to speedup of parallel Java programs, for different numbers of tasks. All numbers are relative to single-threaded Java. Error bars are omitted since the variation is insignificant for all programs.

4.2 Results

First, we focus on the parallel speedup of the Rolez programs and compare it to that of equivalent Java programs. Note that the Rolez programs reuse some Java classes, such as `System` and `Math`, which contain native code, and also classes like

`String` and `Random`, to avoid the porting effort to Rolez. We manually ensured that the use of these classes is deterministic. Figure 10 shows the speedups of the Rolez and Java programs, relative to the single-threaded Java version, for different numbers of tasks. Note the logarithmic scale of both axes.

All Rolez programs achieve substantial speedups. They outperform single-threaded Java already with two tasks, and achieve maximum speedups of 7–20×. The speedup they achieve is practically linear with up to 8 tasks, and for IDEA and Monte Carlo even with 32 tasks. The plots also give a first idea about the Rolez overhead. While for IDEA and Monte Carlo the speedup lines are mostly equal, the overhead is clearly visible for animator and *k*-means, where the Java versions achieve substantially higher performance.

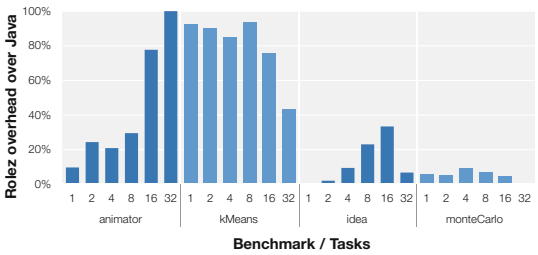

Fig. 11. Relative Rolez overhead when compared to the Java version of the same program and with the same number of tasks. Again, error bars are omitted due to insignificant variation.

Figure 11 shows this overhead in more detail. For IDEA, the overhead stays below 35% and for Monte Carlo even below 10%. In both of these programs, there is a modest amount of sharing and, due to static analysis in the Rolez compiler, almost no need for guarding. While there is more sharing in the animator program, the overhead stays low for up to 8 tasks. With more tasks, a limitation of the current incarnation of Parallel Roles shows: Since there is no built-in support for data partitioning, data sets need to be split and merged explicitly, which may result in a substantial overhead. Finally, *k*-means contains the most sharing and therefore suffers most from the overhead caused by role transitions.

To summarize, while the runtime concepts of Parallel Roles may inflict a non-negligible performance overhead, our prototype still delivers substantial parallel speedups. We expect that the performance of Rolez could be significantly improved by a more advanced compiler with access to global program information or runtime data (such as a JIT compiler), or by more optimized guarding and role transitions. However, we argue that the current Rolez prototype already provides good performance for many applications, especially on personal devices, where the number of cores has remained relatively small.

5 Related Work

Many approaches have been proposed to make parallel programming in some way safer than with explicit synchronization. Recently, the deterministic-by-default approach for imperative, object-oriented languages has sparked the interest of the research community [5,13,23,24]. In imperative languages, DPP is hard because tasks may have effects on shared mutable state. If not restricted, the nondeterministic interleaving of such effects leads to nondeterministic results [23].

The first imperative DPP language is Jade [22,32], where the programmer specifies the effects of a task using arbitrary code that is executed at runtime. 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 [2], where the programmer writes code that assigns operations to different *serialization sets*, and to Yada [14], where *sharing types* restrict how tasks may access shared data. Yada's sharing types are similar in spirit to role types, but they were not designed with compile-time checking as a goal.

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. While early systems like FX [26] can only express limited forms of parallelism, recent systems like Liquid Effects [20] or Deterministic Effects [25] can handle many kinds of parallelism, although not necessarily in an object-oriented setting. The effect system used in Deterministic Parallel Java (DPJ) [4,6] and TWEJava [18] brings statically checked effects to Java-like languages. To support a wide range of parallel patterns, it includes many features: region parameters, disjointness constraints, region path lists, indexed-parameterized arrays, subarrays, and invocation effects. This formidable list shows that DPJ and TWEJava require a programmer to understand many and potentially complex concepts. Parallel Roles aims to simplify DPP by using the concepts of roles and role transitions to specify the effects of tasks. In addition, the concept of guarding enables parallelization by simply marking methods as tasks and invoking them like normal methods.

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 [7,19]. These systems combine effects with ownership types [11,12] and generally couple the regions and effects of an object with those of its owner. This idea resembles our handling of object graphs, 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. Even though this simple idea of "referrer as owner" has the advantage that no additional notion of ownership is involved, combining roles with a more advanced concept of ownership would be interesting future work.

An alternative to effects are systems based on *permissions* [3,8,9]. Permissions accompany object references and define how an object is shared and how it may be accessed. In ÆMINIUM [37,38] 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 [27] features *mutable* or *immutable* references and guarantees that there are either a single mutable or multiple immutable references to an object at any time. Permissions are more object-based than effects and conceptually similar to our roles. However, roles and particularly guarding are dynamic concepts and enable simpler language designs, at the cost of some runtime overhead. For instance, while Æminium and Rust rigorously restrict aliasing, Rolez is a simpler language that permits arbitrary aliasing.

Another approach for DPP is speculative execution, where the effects of tasks are buffered by a runtime component and rolled back in case they interfere. The two most well-known such approaches, Thread Level Speculation [29,35,36] and Transactional Memory [16,17,33] are not DPP models in a strict sense: The former *automatically* parallelizes sequential programs and the latter usually provides no determinism guarantees. However, there *are* speculative approaches that constitute DPP models: Safe Futures for Java [40] and Implicit Parallelism with Ordered Transactions [39]. In both models, the programmer defines which parts of a sequential program should execute asynchronously. The runtime then executes them as speculative tasks, enforcing their sequential order. In Parallel Roles, speculation is not necessary, because interfering operations are either delayed by guarding or cause an error (in the case of Rolez, at compile time).

6 Conclusion

During the last few years, much research about deterministic parallel programming has focused on static effect or permission systems. In this paper, we presented Parallel Roles to leverage *roles* to express the kinds of access that are permitted for an object. Parallel Roles puts the focus on objects and presents a simple object-oriented way to specify and reason about effects of parallel computations. This paper explores parallel programming with just three simple roles; these are powerful enough to express a wide range of parallel patterns and applications without the burden of complex program annotations. While a certain runtime overhead seems to be the necessary toll for this simplicity, a preliminary evaluation indicates that the overhead is moderate: The implementation of a roles-based language achieves substantial speedups over the corresponding sequential Java version. Furthermore, past programming language innovations such as garbage collection or runtime type checking have shown that a modest runtime overhead is a small price to pay for more safety, simplicity and programmer productivity.

References

1. Xtext. http://www.eclipse.org/Xtext/
2. Allen, M.D., Sridharan, S., Sohi, G.S.: 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 2009), pp. 85–96. ACM, New York (2009)

3. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA 2007), pp. 301–320. ACM, New York (2007)
4. Bocchino, R.L., Adve, V.S.: Types, regions, and effects for safe programming with object-oriented parallel frameworks. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 306–332. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22655-7_15
5. Bocchino, R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel programming must be deterministic by default. In: Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar 2009). USENIX Association, Berkeley (2009). http://dl.acm.org/citation.cfm?id=1855591.1855595
6. Bocchino, R.L., et al.: 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 2009), pp. 97–116. ACM, New York (2009)
7. Boyapati, C., Lee, R., Rinard, M.: 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 2002), pp. 211–230. ACM, New York (2002)
8. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
9. Boyland, J.T., Retert, W.: Connecting effects and uniqueness with adoption. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), pp. 283–295. ACM, New York (2005)
10. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: stanford transactional applications for multi-processing. In: Proceedings of The IEEE International Symposium on Workload Characterization (IISWC 2008), September 2008
11. Clarke, D.G., Noble, J., Potter, J.M.: Simple ownership types for object containment. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 53–76. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45337-7_4
12. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1998), pp. 48–64. ACM, New York (1998)
13. Devietti, J., Lucia, B., Ceze, L., Oskin, M.: DMP: deterministic shared memory multiprocessing. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV), pp. 85–96. ACM, New York (2009)
14. Gay, D., Galenson, J., Naik, M., Yelick, K.: Yada: straightforward parallel programming. Parallel Comput. **37**(9), 592–609 (2011)
15. Gottlob, G., Schrefl, M., Röck, B.: Extending object-oriented systems with roles. ACM Trans. Inf. Syst. **14**(3), 268–296 (1996)
16. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2003), pp. 388–402. ACM, New York (2003)
17. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA 1993), pp. 289–300. ACM, New York (1993)

18. Heumann, S.T., Adve, V.S., Wang, S.: The tasks with effects model for safe concurrency. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013), pp. 239–250. ACM, New York (2013)
19. Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A statically verifiable programming model for concurrent object-oriented programs. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 420–439. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_23
20. Kawaguchi, M., Rondon, P., Bakst, A., Jhala, R.: Deterministic parallelism via liquid effects. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), pp. 45–54. ACM, New York (2012)
21. Kristensen, B.B.: Object-Oriented Modeling with Roles. In: Murphy, J., Stone, B. (eds.) OOIS 1995, pp. 57–71. Springer, London (1996)
22. Lam, M.S., Rinard, M.C.: Coarse-grain parallel programming in Jade. In: Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 1991), pp. 94–105. ACM, New York (1991)
23. Lee, E.A.: The problem with threads. Computer **39**(5), 33–42 (2006)
24. Lu, L., Scott, M.L.: Toward a formal semantic framework for deterministic parallel programming. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 460–474. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24100-0_43
25. Lu, Y., Potter, J., Zhang, C., Xue, J.: A type and effect system for determinism in multithreaded programs. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 518–538. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_26
26. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988), pp. 47–57. ACM, New York (1988)
27. Matsakis, N.D., Klock II, F.S.: The rust language. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT 2014), pp. 103–104. ACM, New York (2014)
28. Pernici, B.: Objects with roles. In: Proceedings of the ACM SIGOIS and IEEE CS TC-OA Conference on Office Information Systems (COCS 1990), pp. 205–215. ACM, New York (1990)
29. Rauchwerger, L., Padua, D.: 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 1995), pp. 218–232. ACM, New York, June 1995
30. Reenskaug, W., Wold, P., Lehne, O.A.: Working with Objects: OORAM Software Engineering Method. J a Majors, Greenwich, June 1995
31. Riehle, D., Gross, T.: Role model based framework design and integration. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1998), pp. 117–133. ACM, New York (1998)
32. Rinard, M.C., Lam, M.S.: The design, implementation, and evaluation of Jade. ACM Trans. Program. Lang. Syst. **20**(3), 483–545 (1998)
33. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC 1995), pp. 204–213. ACM, New York (1995)

34. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC 2001), pp. 8. ACM, New York (2001)
35. Sohi, G.S., Breach, S.E., Vijaykumar, T.N.: Multiscalar processors. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA 1995), pp. 414–425. ACM, New York, June 1995
36. Steffan, J., Mowry, T.: 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 1998), pp. 2–13. IEEE Computer Society, Washington DC (1998)
37. Stork, S., Marques, P., Aldrich, J.: 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 2009), pp. 933–940. ACM, New York (2009)
38. Stork, S., et al.: Æminium: a permission-based concurrent-by-default programming language approach. ACM Trans. Program. Lang. Syst. **36**(1), 2:1–2:42 (2014)
39. von Praun, C., Ceze, L., Caşcaval, C.: Implicit parallelism with ordered transactions. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2007), pp. 79–89. ACM, New York (2007)
40. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), pp. 439–453. ACM, New York (2005)