

DISS. ETH No. 26132

A DYNAMIC APPROACH
TO DETERMINISTIC
PARALLEL PROGRAMMING

A dissertation submitted to attain the degree of
DOCTOR OF SCIENCES OF ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

MICHAEL FAES

MSc ETH CS
born on 04.11.1987
citizen of Aesch (BL), Switzerland

accepted on the recommendation of

Prof. Dr. Thomas R. Gross
Prof. Dr. Peter Müller
Prof. Dr. Jonathan Aldrich

2019

ABSTRACT

Deterministic parallel programming models are a beacon of hope in the stormy, concurrency-bug-ridden seas of parallel programming. These models guarantee that any parallel execution of a program produces the same results as a sequential execution, implying the absence of any kind of concurrency bugs. However, most existing deterministic models rely on compile-time techniques and on complex program annotations to achieve this goal. While such an approach ensures high performance, it negatively affects the simplicity of a language, making it unsuitable for non-expert programmers.

This dissertation explores a relatively uncharted area in the design space of parallel programming models and presents an approach that is deterministic, but relies primarily on runtime checking to guarantee this property. In this programming model, every object *plays a role* in every concurrent task, for example, the `READWRITE` role or 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, namely, by changing the set of permitted operations that can be performed with this object. This mechanism can be leveraged to prevent interfering accesses from concurrently executing tasks and makes parallel execution deterministic.

To this end, the dissertation presents a role-based programming language that includes several novel concepts (role transitions, guarding, slicing) to enable practical deterministic parallel programming. Unlike previous deterministic languages, this language can express a variety of parallel patterns without relying on complex language constructs or rigorous restrictions.

A prototype implementation demonstrates that this dynamic approach yields high performance for a range of programs, despite the overhead that stems from the necessary runtime checks. In particular, the implementations of 8 widely used programming problems achieve substantial parallel speedups, as a result of a combination of optimizations specifically tailored towards this programming model.

In summary, this dissertation demonstrates that deterministic parallel programming can be realized with a dynamic instead of a static approach, yielding vastly different—and much simpler—language designs.

ZUSAMMENFASSUNG

Deterministische Modelle für paralleles Programmieren sind Hoffnungsträger in der Concurrency-Bug-verseuchten Welt der parallelen Programmierung. Diese Modelle garantieren, dass jede parallele Ausführung eines Programms die selben Resultate wie eine sequenzielle Ausführung produziert, was die Abwesenheit jeglicher Art von Concurrency Bugs bedeutet. Jedoch basieren die meisten bisherigen deterministischen Modelle auf Kompilierzeit-Techniken und auf komplexen Programm-Annotationen. Ein solcher Ansatz sorgt für hohe Leistung, aber er beeinflusst die Einfachheit einer Programmiersprache negativ und ist deshalb ungeeignet für Nicht-Experten.

Diese Dissertation erkundet ein relativ unbekanntes Gebiet im Design-Raum von parallelen Programmiermodellen und präsentiert einen Ansatz, welcher deterministisch ist, aber in erster Linie Laufzeit-Checks verwendet um Determinismus zu garantieren. In diesem Modell *spielt* jedes Objekt *eine Rolle* in jeder gleichzeitig ausgeführten Task, zum Beispiel die READ-WRITE-Rolle oder die READONLY-Rolle. Wenn ein Objekt mit einer neuen Task geteilt wird, dann passt es sich an das neue Sharing-Muster an, indem es seine gespielten Rollen und damit sein Verhalten ändert. Genauer gesagt, wenn ein Objekt geteilt wird, ändern die für dieses Objekt erlaubten Operationen. Mit diesem Mechanismus werden Zugriffe verhindert, die von zwei gleichzeitig ausgeführten Tasks gemacht werden und sich gegenseitig störend beeinflussen, und damit wird die Ausführung deterministisch.

Zu diesem Zweck präsentiert diese Dissertation eine Rollen-basierte Programmiersprache, welche verschiedene neue Konzepte (“role transitions”, “guarding”, “slicing”) enthält, welche praktisches und deterministisches paralleles Programmieren ermöglichen. Im Gegensatz zu früheren deterministischen Sprachen kann diese Sprache eine Vielfalt von parallelen Programm-Muster ausdrücken, ohne sich dabei auf komplexe Sprachkonstrukte oder strikte Einschränkungen zu verlassen.

Eine Prototyp-Implementierung zeigt, dass dieser dynamische Ansatz hohe Leistung für verschiedene Programme liefert, trotz des Overheads der nötigen Laufzeit-Checks. Insbesondere erreichen die Implementierungen von 8 verbreiteten Programmierproblemen beträchtliche parallele Speedups, unter anderem dank einer Kombination von Optimierungen, welche speziell auf dieses Programmiermodell zugeschnitten wurden.

Insgesamt zeigt diese Dissertation auf, dass deterministisches paralleles Programmieren anstelle eines statischen auch mit einem dynamischen Ansatz möglich ist, welcher ganz andere—deutlich einfachere—Sprach-Designs ermöglicht.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my creator and Father in heaven, to whom I owe everything. I would like to thank my mother and my father for their unconditional and diverse support and my whole (extended) family for never doubting my talent or perseverance. I am also grateful to Domi for supporting me throughout most of my studies and for countless moments of happiness.

Special thanks go to my advisor, Prof. Thomas Gross, who taught me countless lessons about computer science, writing, presenting, and life, and who gave me all the freedom and support that I needed to become an independent researcher. And I am deeply grateful to Michael Pradel, who did a superb job of advising me and working with me on my master's thesis—and who talked me into pursuing a PhD in the first place.

I want to thank Prof. Peter Müller and Prof. Jonathan Aldrich for the constructive and insightful discussions on the day of my defense. In addition, I thank Prof. Aldrich for traveling to Switzerland and for compiling an extremely detailed and accurate feedback on the first draft of this document.

I would also like to thank Stephanie Balzer and Angela Wilhelm for their valuable feedback on some of my paper drafts.

I thank all the past and current members of the LST and ACL groups at ETH who are responsible for countless valuable or fun moments at conferences or during lunch and coffee breaks: Zoltán Majó, Faheem Ullah, Daniele Spampinato, Georg Ofenbeck, Victoria Caparrós Cabezas, Stefan Schmid, Martin Bättig, François Serre, Alen Stojanov, Virág Varga, Ivana Unkovic, Eliza Wszola, Chris Wendler, Tyler Smith, and Joao Rivera.

I am immensely grateful for the friendship with my fellow LLT members, Luca, Aristeidis, and Remi. I dare not imagine how dull my PhD time would have been without the (double) dinners of various nationalities and ethnicities, the fruitful technical and endless non-technical discussions, and all the fun we shared.

Finally, thank you, Manu, for your support, your positivity, and for making me the happiest man on the planet.

CONTENTS

ABSTRACT	iii
ZUSAMMENFASSUNG	iv
1 INTRODUCTION	1
1.1 Thesis	1
1.2 Effect- and Permission-based Languages	4
1.3 Parallel Roles	7
1.4 Organization of this Thesis	10
2 PROGRAMMING MODEL	11
2.1 Core Concepts	11
2.2 Example	12
2.3 Formal Description	14
2.3.1 Domain	15
2.3.2 State Transition Relation	16
2.4 Determinism	21
2.5 Related Work	22
3 LANGUAGE	23
3.1 Basic Language Features	23
3.2 Joint Role Transitions	25
3.3 Slicing	26
3.3.1 Overview	26
3.3.2 Formal Definition and Determinism	27
3.3.3 Array Slicing	28
3.3.4 Class Slicing	29
3.4 Eager Interference Checking	30
3.4.1 Parfor Loop	31
3.4.2 Parallel-And Statement	32
3.4.3 Formal Definition	33
3.5 Expressiveness Evaluation	34
3.6 Related Work	36
4 ROLE TYPE SYSTEM	41
4.1 Role Errors	41
4.2 Role Types	42
4.3 Example	43
4.4 Formal Description	45
4.4.1 Syntax	46
4.4.2 Subtyping	47
4.4.3 Typing Rules	48
4.5 Role Parameters	49
4.5.1 Example	49
4.5.2 Formal Description	50
4.6 Related Work	51

5	IMPLEMENTATION	53	
5.1	Prototype Overview	53	
5.2	Optimizations	56	
5.2.1	Code-Managed Runtime Data	56	
5.2.2	Concurrency Analysis	58	
5.3	Performance Evaluation	65	
5.3.1	Experimental Setup	65	
5.3.2	Parallel Speedup	66	
5.3.3	Overhead	68	
5.3.4	Impact of Optimizations	68	
5.3.5	Impact of Slice Coverage Checks	71	
5.4	Related Work	73	
6	CONCLUSION	77	
6.1	Summary	77	
6.2	Implications	78	
APPENDIX			
A	DETERMINISM PROOF	83	
A.1	Outline	83	
A.2	Legal Program States	83	
A.3	Soundness of Role Declarations	84	
A.4	Child Task Priority	91	
A.5	Noninterference	96	
BIBLIOGRAPHY		99	
CURRICULUM VITAE		107	

INTRODUCTION

1.1 THESIS

A DETERMINISTIC PARALLEL PROGRAMMING MODEL CAN BE REALIZED WITH A SMALL NUMBER OF NEW LANGUAGE CONCEPTS AND FEW RESTRICTIONS, WHILE YIELDING HIGH PERFORMANCE.

Most widely-used programming models and languages today provide little assistance in engineering bug-free parallel software. They follow an approach that—for the sake of performance or language simplicity—employs no safety checks with respect to correct parallel execution, neither at run-time nor at compile time. Thus, parallel programs often contain various concurrency bugs, leading to race conditions or deadlock.

Such bugs have been studied extensively, and a myriad of models, languages, programming methodologies, and software engineering tools have been created to prevent or at least detect them. Yet, concurrency bugs are only the symptoms of a much more fundamental issue: *nondeterminism*. A parallel programming model is nondeterministic if a program that follows this model can produce different results, even when all programmer- and user-controlled factors, such as input or operating environment, remain the same. In other words, the result of a program execution is affected by uncontrollable factors, which depend on the concrete programming model.

Nondeterminism

A well-known nondeterministic model is the *thread-based* model, in which parallel execution is achieved by distributing the workload among multiple concurrently executing threads. These threads share a common memory space that all of them can access. To ensure maximum performance, many thread-based languages by default provide extremely weak guarantees about the visibility, ordering, and atomicity of concurrently executed operations on this shared memory. For example, the Java language gives no guarantee that multiple read or write operations performed by a thread are executed atomically* or even that a normal write operation is ever seen by other threads. It is the programmer's responsibility to ensure such properties, for example, by using locks or specific memory operations such as *compare-and-swap*. Should the programmer fail to employ such explicit synchronization correctly, the program will be nondeterministic and may crash, deadlock, or—worst of all—silently produce incorrect results.

*Thread-Based
Concurrency Model*

* that is, they cannot be interleaved with operations from a different thread

Other programming models, such as the Actor Model (Agha, 1986, 1990; Hewitt et al., 1973) or the Active Objects Model (Yonezawa et al., 1986), address some of the issues in the thread-based model by abandoning the shared memory concept and completely isolating concurrently executing *actors* from each other. The only way for actors to communicate is by sending and receiving messages. This model precludes many low-level concurrency bugs, such as data races or deadlock, and frees the programmer from reasoning about memory model intricacies. However, the Actor Model suffers from nondeterminism too: actors may behave differently depending on the order of the messages they receive, which in turn depends on various uncontrollable factors, like the scheduling of other actors' execution, or the time that passes between when a message is sent and received.

Nondeterminism poses serious challenges to software engineering. If a program can produce different results when given the same inputs, standard testing techniques cannot guarantee that a program always behaves as expected. Simple approaches like executing tests repeatedly are unhelpful, as many nondeterministic concurrency bugs manifest rarely—and sometimes only in production, where they can cause incomparably higher costs. Nondeterminism also hampers the ability to debug a program, as the presence of a debugger often changes the scheduling of operations, preventing the programmer from analyzing a “typical” execution.

What makes the problem of nondeterminism even more delicate is that it is specific to parallel (or more precisely, concurrent) programming. Sequential programs are typically deterministic, because they are unaffected by the effects of scheduling and weak memory models. Thus, correctly parallelizing an existing sequential program under any nondeterministic model can be extremely difficult and error-prone, and often requires expert knowledge about memory models and intricate reasoning about possible interleavings of operations. In addition, achieving high performance on modern multi-core CPUs is impossible without parallel execution, making nondeterminism an ever more pressing issue.

While many programming models address individual symptoms of nondeterminism, for example, by preventing data races, one approach aims at the very disease and makes parallel programming *deterministic by default* (Lee, 2006; Bocchino et al., 2009a). In such a *deterministic parallel programming model* a program is guaranteed to produce results that are affected only by programmer- or user-controlled factors such as the program input—unless the programmer uses nondeterministic operations explicitly. Usually, the results are not only deterministic, but equal to those that are produced if the program is executed in a sequential instead of a parallel manner.

Deterministic parallel programming (DPP) restores the effectiveness of testing, the ability to debug, and—most importantly—opens the parallel programming door to non-expert programmers who have no training in reasoning about concurrent access to shared memory or messaging. In addition, it may ease the process of parallelizing existing sequential programs,

as the semantics of the program is preserved as long as nondeterminism is not introduced explicitly.

Many existing DPP models build on top of the thread-based model, because of its simplicity and popularity. Thread-based determinism can be guaranteed using a property called *noninterference*, which asserts that the execution of one thread cannot influence the computation of another thread; we say that two threads cannot *interfere*. If threads cannot interfere, then each thread performs the same computation—and thus produces the same results—as if it were the only thread in the program. Consequently, the entire program produces the same results as if the computations in all threads were performed in a single thread, which amounts to sequential execution.

Noninterference

The difficulty of ensuring noninterference depends heavily on the properties of a programming language. For purely functional languages, noninterference is trivial: since the evaluation of any expression is free from side effects, it cannot interfere with the evaluation of any other expression. By contrast, in an imperative language, any segment of code can have side effects on some mutable state; thus, noninterference requires all pairs of operations from two concurrent segments of code to *commute with each other*, i.e., that neither of the operations writes to a memory location that the other reads from or writes to as well. For a language with object references, this property is usually difficult to check, as it requires *aliasing information*, i.e., information about whether two given references are guaranteed to refer to different objects.

Commuting Operations

Aliasing

Most of today's widely-used languages fall into the far end of this difficulty spectrum: they are imperative, object-oriented languages that allow unrestricted aliasing. While it is debatable whether such a design is the best starting point for parallel languages for a future generation of programmers, it is a fact that many *existing* programmers are unfamiliar with different language designs, for example, purely functional ones. Therefore, designing imperative, object-oriented deterministic languages is an ongoing research effort.

Most imperative DPP approaches explore one of two models to ensure noninterference. In the first model, the programmer declares the *effects* (Lucassen and Gifford, 1988; Greenhouse and Boyland, 1999; Leino et al., 2002) of each segment of code (e.g., of each method), by summarizing the read and write operations that are performed by this segment. Then, noninterference is checked in two steps: first, by verifying that all effect annotations are correct, and second, by asserting that the effects of concurrently executing code are free of conflict. In the second model, noninterference is ensured using *permissions* (Boyland, 2003; Boyland and Retert, 2005; Bierhoff and Aldrich, 2007): A segment of code can execute a certain operation on an object only if it has the respective permission, which is specific to the object and the kind of operation, i.e., read or write. Noninterference is established by a set of rules that prevent the creation of two or more conflicting permissions for the same object.

Effects

Permissions

Both models affect programming language design. First, in addition to the core concepts (effects or permissions, respectively), most languages based on one of these models also include some concept of ownership or grouping. As an example for an effect-based language, in Deterministic Parallel Java (DPJ) (Bocchino et al., 2009b; Bocchino and Adve, 2011) the programmer declares the effects of a method using a rich language based on *regions*, which partition the heap into disjoint parts. On the other hand, $\text{\AE} \text{MINIUM}$ (Stork et al., 2009, 2014), a permission-based language, has the concept of *data groups* (which, like regions, partition the heap into disjoint parts) and the related concept of *group permissions*. Both languages require the programmer to master a substantial number of new concepts to write a parallel program. Second, due to this concept of grouping, effect- and permission-based languages usually impose rigid aliasing restrictions to ensure the disjointness of regions or data groups, respectively.

Static DPP Approaches

The fundamental reason why these two models have such strong effects on language design is that they are both static approaches, that is, they check noninterference *at compile time*. A major reason is performance: if noninterference is checked at compile time, then runtime checking and the associated runtime overhead can be avoided. However, the focus on this *performance-first* approach also means that almost every existing deterministic language suffers from at least one of two problems that affect simplicity: the language contains a large number of new concepts that are required for checking noninterference, or it imposes major restrictions that affect the productivity of programmers—especially of those only familiar with mainstream languages.

By consciously abandoning the requirement of compile-time interference checking, and instead employing a dynamic, *simplicity-first* approach, this dissertation shows that *a deterministic parallel programming model can be realized with a small number of new language concepts and few restrictions*. Nevertheless, by using a combination of model-specific optimizations, simplicity is achieved *while yielding high performance* for programs with a range of parallel patterns.

1.2 EFFECT- AND PERMISSION-BASED LANGUAGES

The complexity imposed by static effect- or permission-based languages is better understood using examples of program code written in such languages.

The first example concerns DPJ (Bocchino et al., 2009b; Bocchino and Adve, 2011), a Java-like language with a static effect system that supports a wide range of parallel patterns. In DPJ, the programmer writes *method effect summaries* that state which parts of the object heap, called *regions*, are read and written by each method. Regions are declared and named by the programmer, much like fields or local variables. By additionally declaring the region that each field resides in, the programmer effectively partitions the heap into disjoint parts that the DPJ effect system can reason about.

```

1 class TreeNode<region P> {
2   region Links, L, R;
3   double value = 0 in P;
4   TreeNode<P:L> left in Links;
5   TreeNode<P:R> right in Links;
6   void computeSum() reads Links, writes P {
7     cobegin {
8       /* reads Links, writes P:L */
9       if (left != null) left.computeSum();
10      /* reads Links, writes P:R */
11      if (right != null) right.computeSum();
12    }
13    this.value += left != null ? left.value : 0;
14    this.value += right != null ? right.value : 0;
15  }
16 }

```

Figure 1.1: Example for regions and effects in DPJ. Region and effect declarations are highlighted.

The DPJ example is presented in Figure 1.1 and shows a class that represents a node in a binary tree. Line 2 contains the declaration of three regions, `Links`, `L`, and `R`, while Lines 3 to 5 show how fields are assigned to regions. The example also shows more features of the effect system, including *region parameters* (first line) and *region path lists*, which are used to specify nested regions (Lines 4 and 5). Finally, the figure contains an example of a method effect summary, on Line 6, and shows how to specify parallel execution in DPJ, using the `cobegin` block. All statements in such a block may be executed in parallel, and the DPJ compiler checks that the effects of these statements are conflict-free. In this example, this is indeed the case: First, both statements access the `Links` region (where the `left` and `right` fields reside in), but only in a reading manner. Second, both statements modify the `value` field of a number of `TreeNode` instances, but the two statements operate on different subtrees, i.e., different regions. The effects that the compiler computes for the two statements are given in comments, but the details of the noninterference check that the compiler performs are not discussed here.

Region Parameters,
Region Path Lists

The DPJ example illustrates that a static effect system brings with it a host of new concepts (regions, region parameters, region path lists) that a programmer needs to master to effectively use such a language. In fact, the concepts used in the example (with the exception of region path lists) are considered “basic capabilities” of the language (Bocchino et al., 2009b); to support other patterns, like data parallelism with arrays, the language includes many more concepts, including *disjointness constraints*, *index-parameterized arrays*, *subarrays*, and *invocation effects*.

Interestingly, an effect summary by itself is not necessarily complex; in the example in Figure 1.1, it is simply “reads `Links`, writes `P`”, which is simple enough to understand. The complexity lies elsewhere, namely in

```

1  state LinkedListNode<internal, data> {
2      ...
3  }
4  state LinkedList<data> {
5      group<internal>
6      shared<internal> LinkedListNode<internal, data> head;
7      method void add<exclusive owner, shared data>
8          (shared<data> Object<data> o): shared<owner> {
9          /* owner: exclusive, data: shared */
10         ...
11     }
12     ...
13 }

```

Figure 1.2: Example for data groups in \mathcal{A} MINIUM, based on the example presented by Stork et al. (2014). Group and permission annotations are highlighted.

the declaration, the parameterization, and the instantiation of the various regions that are required to express the effects of a method precisely enough. The structure of the regions is really a compile-time reflection of the actual runtime structure of the object graph, i.e., how objects are connected to each other with references. By specifying this information in the program text, the programmer enables the compiler to check that two concurrent statements access two disjoint sets of objects (or that objects are only read from but not written to). On a conceptual level, regions ultimately encode the aliasing information that is required to determine noninterference.

The second example deals with \mathcal{A} MINIUM (Stork et al., 2009, 2014), a recent deterministic-by-default language that is based on permissions. In \mathcal{A} MINIUM, every object reference is accompanied by an *access permission*, which explicitly expresses constraints on aliasing and is also a simple form of effect specification. Permissions allow the compiler to build a dependence graph among the statements in a program and then to *automatically* parallelize the program. There are two basic permissions for references: *unique* and *immutable*; the *unique* permission guarantees the absence of aliasing and grants read as well as write access to an object, while *immutable* references can be aliased, but grant only read access. In addition, a *shared* permission indicates that an object may be aliased, but is owned by a *data group*, for which an additional *group permission* determines the permitted access.

Data Groups

The \mathcal{A} MINIUM example is given in Figure 1.2; it shows the skeleton of a linked list implementation. Both states* have data group parameters (Lines 1 and 4) that define to which data group each object that is part of the data structure belongs. The objects that are actually stored in the list are in the “data” group, while the internal list nodes belong to an “internal” group that is declared on Line 5. The latter fact is visible on Line 6, which states that the head node has the permission `shared<internal>`. Lines 7–11 show the declaration of an `add` method, which has four parameters: two group

* States are similar to classes in Java.

parameters (in angle brackets), one regular parameter that corresponds to the object *o* to be added to the list, and an implicit receiver parameter. The parameter *o* has the permission `shared<data>`, while the receiver has the permission `shared<owner>` (declared after the colon). This method declaration expresses two conditions: first, that the object *o* must belong to the same data group that the list is parameterized with, and second, that the list itself be in a data group with the `exclusive` permission, which is a group permission similar to the `unique` permission for single objects.

This example shows that a permission-based language comes with many of the same concepts found in effect systems—ÆMINIUM’s data groups and group parameters closely correspond to DPJ’s regions and region parameters—and with many of the same complexities. While ÆMINIUM’s data groups may be less complex than DPJ’s region path lists, they still require the programmer to put a substantial effort into designing a static abstraction of the heap structure that enables parallel execution—and into expressing this abstraction with the syntax that the language provides.

1.3 PARALLEL ROLES

The fact that static effect- and permission systems check noninterference at compile time—and thus require virtually no runtime checking—makes such languages an attractive option for high-performance applications and for programs with very fine-grained parallelism, for which the gains of parallel execution could easily be negated by any runtime checking overhead. But with the spreading of multi-core CPUs in personal devices, parallel programming has become relevant for a much broader range of applications—and for programmers who would like to get the largest possible performance boost without learning a new and complex language.

This dissertation goes beyond pure compile-time checking of noninterference, and instead explores a programming model that is fundamentally based on runtime checking. However, as an optimization to avoid excessive runtime overhead, and also to avoid unexpected runtime errors, the model can be combined with a lightweight static type system. The combination of runtime and compile-time checking results in a programming language that has fewer and less complex new concepts that a programmer needs to master; it also has much fewer aliasing restrictions in comparison with statically-checked languages such as DPJ or ÆMINIUM.

The building blocks for this programming model are *objects*, *tasks* (which are comparable to threads), and the *explicit relationship between objects and tasks*. More concretely, every object *plays a role* in every task and *may change the roles it plays* whenever it is shared with a task (or unshared). This model, called the *Parallel Roles* model, defines three simple roles that define the permitted operations on an object: `READWRITE`, `READONLY`, and `PURE`. The first two roles resemble ÆMINIUM’s *unique* and *immutable* permissions: `READWRITE` permits read *and* write access, but an object can play this role only in a single task at every point in time; the `READONLY` role permits only

read access, but can be played in multiple tasks at the same time. However, there is a fundamental difference between roles and permissions: roles are a runtime concept and accompany *objects*, not references. Therefore, the number of references to an object is basically irrelevant in this model; instead, the crucial point is which tasks an object has been shared with (and with which role).

To specify the role that an object plays in a task, the programmer provides *role declarations*, which take the form of annotations on the parameters of the task. When a task is started, an object that is passed as an argument to this task performs a *role transition*, such that the role the object plays in this task matches the declared one. Noninterference is ensured by simultaneously changing the role that the object plays in the *parent* task (the task that started the new task) as well. For example, if an object plays the READWRITE role in a task and is then shared as READONLY with a new task (called the *child task*), it performs a role transition and then plays the READONLY role *in both tasks*. When the child task finishes, the inverse role transition takes place and the object plays the READWRITE role for the parent task again.

This means that a programmer should provide role declarations that are as precise as possible, but any less precise declarations will not lead to nondeterminism, as the runtime system ensures that an object never plays a pair of conflicting roles at the same time. Moreover, less precise declarations will not lead to compile-time errors either; instead, the program may simply be serialized to a certain degree.

Rolez Language

The dynamic nature of roles allows for much fewer aliasing restrictions—and thus simpler language designs. As a case in point, this dissertation presents the programming language *Rolez*, which enforces the rules of the Parallel Roles model for every program written in that language; role transitions and guarding are performed implicitly by the runtime system. Besides role declarations for task parameters (which are used to determine the role transitions at the beginning and the end of a task), *Rolez* also requires role annotations for other kinds of variables, e.g., method parameters and fields. These additional annotations are required by the type system mentioned earlier and are used by the *Rolez* compiler to verify that tasks respect their role declarations; for example, if a task declares one of its parameters as READONLY, it may assign it only to other variables that are declared as READONLY. However, these additional role annotations do not enforce any kind of aliasing restrictions and have no effect on the roles of an object; in particular, an object may be assigned to any number of variables annotated as READWRITE. Instead, these annotations are used to reduce the amount of runtime checking and to prevent errors that would occur if a task disrespected a role declaration.

To illustrate the simplicity of *Rolez*, Figure 1.3 shows how the `LinkedList` example in Figure 1.2 looks like when written in *Rolez* instead of *ÆMINIUM*. While the *ÆMINIUM* implementation includes group parameters for both `LinkedList` and `LinkedListNode` and for the `add` method, requires the declaration of an internal group, and contains somewhat complex permission


```

1  class LinkedListNode {
2      ...
3  }
4  class LinkedList {
5      var head: readwrite LinkedListNode
6      def readwrite add(o: readwrite Object): void {
7          ...
8      }
9      ...
10 }

```

Figure 1.3: Example for role declarations in the Rolez language. Role declarations are highlighted.

declarations, the Rolez version requires only three simple role declarations, for the head field (Line 5) and for the two method parameters (the explicit parameter `o` and the receiver, both on Line 6). In fact, `READWRITE` could be defined as the implicit default role; then this example would need no annotations at all.

Despite the small number of annotations, Rolez enables the parallel—and guaranteed deterministic—use of such a linked list. When an instance of `LinkedList` is shared with a task (for example, as `READWRITE`), not only the `LinkedList` object itself, but also all involved `LinkedListNodes` and all objects stored in the list perform a role transition, changing the permitted operations for the parent and child task. This behavior is due to a concept called *joint role transitions*, which essentially causes all objects that are reachable from a root object to perform the same role transition as the root object itself. Thus, Rolez frees the programmer from designing and expressing a *static abstraction* of the heap structure (as is done with regions or data groups) and instead uses the *actual* runtime heap structure as the basis for enforcing noninterference.*

This dynamic approach to the problem of grouping or ownership still requires programmers to consider aliasing when writing a parallel program, and may impose restrictions on the design or selection of data structures. However, it enables reasoning about the problem in simple and familiar terms—object references—and greatly simplifies the use of some aliasing patterns that are difficult (or impossible) to describe statically.

The main challenge that comes with the Parallel Roles model is performance. When an object is shared with a task, the runtime system needs to perform all involved role transitions, i.e., it needs to update the roles of that object on the heap—and, due to joint role transitions, the roles of all objects that are reachable from that object. In addition, Rolez prevents interference between tasks using a concept called *guarding*, which dynamically prevents a parent task from performing operations that interfere with its children. Both guarding and role transitions cause a runtime overhead when compared with statically checked languages like `DPJ` and `ÆMINIUM`, and also when compared with nondeterministic languages like Java. Thus,

* This example assumes that any list instance is accessed only by a single task at any point in time, or that multiple tasks access it in a read-only manner. Permitting multiple tasks to (deterministically) modify the same list instance would be more involved—both in Rolez and in `ÆMINIUM`.

this dissertation also addresses the research question of how large this overhead is—and by how much it can be reduced using roles-specific compiler optimizations.

1.4 ORGANIZATION OF THIS THESIS

The dissertation is organized as follows. Chapter 2 presents the Parallel Roles model, including a formal proof of determinism and a comparison with other object-oriented models of concurrency. Chapter 3 discusses the design of the Rolez programming language and evaluates its expressiveness, using a range of patterns found in parallel programs. Chapter 4 presents the role-based type system, which complements the dynamic concept of guarding and increases the safety and efficiency of Rolez, by checking for a specific class of errors at compile time. Chapter 5 discusses the implementation of Rolez, focusing on two optimizations that help reducing the runtime overhead. This chapter also features an empirical performance evaluation, which illustrates the potential of the approach and discusses the impact of the optimizations presented earlier. Finally, the conclusions of this dissertation are presented in Chapter 6.

The material presented in these chapters draws upon, refines, and extends our earlier published work (Faes and Gross, 2017, 2018a,b).

2

PROGRAMMING MODEL

2.1 CORE CONCEPTS

The Parallel Roles model consists of four core concepts: *roles*, *role declarations*, *role transitions*, and *guarding*. Only one of these concepts—role declarations—is a static concept that corresponds to a programming language construct; all others are dynamic concepts that correspond to runtime information and actions. This dynamic nature of the model distinguishes it from most other deterministic models.

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 object-oriented programming 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.

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 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. By restricting the possible combinations of roles and by enforcing each role's access restrictions, the model guarantees noninterference and, by extension, determinism. We formally prove this statement in Section 2.4.

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.

Role Declarations

Role Transitions









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 *child task* (the newly started task) declares the READONLY role for an object. In that case, the object becomes READONLY also in the parent task (if it is not already). Once the child task finishes, the object performs another role transition and may become READWRITE again in the parent task. 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 become *temporarily* illegal, despite being legal under the declared role. Temporarily illegal operations are handled differently from *erroneous* operations, i.e., from operations that are illegal under the *declared* role of an object: While performing an erroneous operation is considered a programmer mistake, performing a temporarily illegal operation is allowed and does not result in any compile-time or runtime error. Instead, a temporarily illegal operation is handled by *delaying* the execution of the task that performs the operation—until the target object plays its declared role again and the operation becomes legal once more. This way, two tasks, a parent and its child task, can execute in parallel as long as there is no interference between them; as soon as the parent attempts to perform an interfering operation (as determined by the roles of the involved objects), it is being blocked. Only once the child task has finished, and another set of role transitions has taken place, the parent may continue to execute. This concept of automatically delaying the execution of a task due to a temporarily illegal operation is called *guarding*.

Guarding

2.2 EXAMPLE

Role transitions and guarding are illustrated using a code segment written in Rolez that is shown in Figure 2.1. The illustration in Figure 2.2 demonstrates how role transitions and guarding influence the execution of this code.

The illustration uses various symbols and colors: The  symbols stand for tasks; in this example, there is a parent task t_{parent} , which is displayed using the blue symbol , and a child task t_{child} with the purple symbol . The roles that an object plays are displayed using the three symbols , , and , which stand for READWRITE, READONLY, and PURE, respectively. To indicate to which task a role belongs, a role symbol is displayed in the respective task's color; for example, the symbol  means that an object plays the READONLY role in the blue task .

The code in Figure 2.1 contains a method `foo` and a task `computeInterest`. The `foo` method, which we assume is executed in a t_{parent} task, first creates

```

1  def foo(): void {
2      var acc = new Account;
3      acc.deposit(10000);
4
5      var interest = start computeInterest(acc);
6
7      print(acc.balance);
8      acc.deposit(5000);
9      ...
10 }
11
12 task computeInterest(acc: readonly Account): int {
13     ...
14 }

```

Figure 2.1: Rolez code example for role transitions and guarding. The parts that are related to roles and parallelism are highlighted. The execution of this code is illustrated in Figure 2.2.

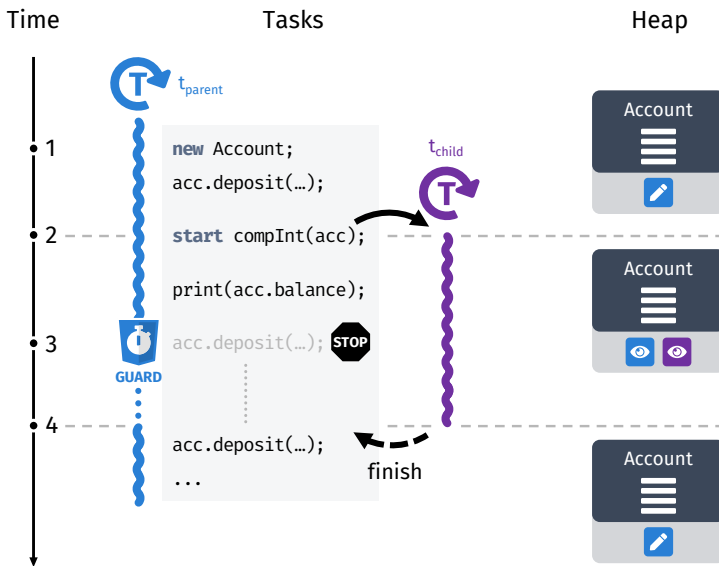


Figure 2.2: Execution illustration of the code in Figure 2.1. The left part shows the key points on the time line, the center part illustrates the execution of the two task, and the right part shows the same `Account` object and the roles it plays during different phases of the execution.

an Account instance. This instance initially plays the READWRITE role in t_{parent} , as shown in the illustration at Point 1. Therefore, t_{parent} is permitted to mutate the object and the call to the `deposit` method is legal.

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 (Point 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 would therefore interfere with `computeInterest` if the two were executed concurrently. In that case, `computeInterest` would take the balance *either before or after* the `deposit` operation as a basis for the computation—depending on the scheduling of the tasks. This nondeterminism is prevented by guarding: As long as `computeInterest` has not finished, the write operation is temporarily illegal; it is legal under the declared role of the object, READWRITE (as explained in the next section), but illegal under the actual role, READONLY. Therefore, before t_{parent} may perform the write operation, a *guard* checks if the operation happens to be temporarily illegal. If this is the case, the execution of t_{parent} is blocked, as illustrated at Point 3.*

* Note that guards are implicit and not visible in the program code.

Once t_{child} finishes, the object performs another role transition and becomes again READWRITE for t_{parent} (Point 4). Now, the write operation in t_{parent} is legal again and the `deposit` method may continue to execute.

The example illustrates how little information the programmer needs to provide in this model; in this example, only the simple `readwrite` annotation for the `computeInterest` task. Interference is automatically prevented by the runtime system, which keeps track of the roles of every object, performs role transitions at the beginning and the end of a task's execution, and enforces access restrictions using guarding.

2.3 FORMAL DESCRIPTION

The formal definition of the Parallel Roles model 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 proof of determinism. 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 roles that an object 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 role-related aspects of task, i.e., its role declarations, and model the execution state of a task

ϱ	\in Role = {RW, RO, PU}	(I)
r	\in Reference	
t, τ	\in Task ID	
$R ::= \{r_i^{i \in 1..n}\}$	\in References = $\mathcal{P}(\text{Reference})$	
$T ::= \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, c \rangle$	\in Task = Task ID \times (Task ID \cup {•}) \times References ² \times ($\mathbb{N}^0 \cup$ {•})	(II)
$Ts ::= \{T_i^{i \in 1..n}\}$	\in Tasks = $\mathcal{P}(\text{Task})$	
$O ::= \{t_i :: \varrho_i^{i \in 1..n}\}$	\in Object = Task ID \rightarrow Role	(III)
$H ::= \{r_i \mapsto O_i^{i \in 1..n}\}$	\in Heap = Reference \rightarrow Object	
$S ::= \langle H, Ts \rangle$	\in State = (Heap \times Tasks) \cup {S _{error} }	

Figure 2.3: 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).

as a single counter, which is incremented whenever the task executes an operation.

2.3.1 Domain

The formal domain of Parallel Roles, which describes the “structure” of the entities that we model, is shown in Figure 2.3. The parts that are specific to 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^{RW} and R^{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^{RW} set contains the references to all the objects that are supposed to be shared with t as READWRITE, while R^{RO} contains those shared as READONLY. Note that a R^{PU} set is not required, as we explain in Section 2.3.2. 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

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

Figure 2.4: 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.

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.

2.3.2 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 the previous sections. 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.

To define the state transition relation, we first need a few helper functions, which are defined in Figure 2.4. The function $\text{obj}_H(r)$ denotes the object that the reference r refers to (for a given heap H), while $\text{task}_{T_S}(t)$ denotes the task with the ID t (for a given set of tasks T_S). Further, $\text{ancests}_{T_S}(t)$ 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.

Most interesting are the functions $\text{mayRead}_H(t, r)$ 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 $(t :: \text{RW})$, i.e., it plays the READWRITE role in t , then t may write to the object. Similarly, if the object contains the mapping $(t :: \text{RW})$ or $(t :: \text{RO})$, then t may read from the object. Note that 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 2.5 and 2.6. 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.

INITIAL STATE The first part in Figure 2.5 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 heap and a set of one single task with some ID t_{main} , with no parent task, empty role declarations, and an instruction counter of 0.

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, as shown at Highlight I in Figure 2.5. 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 2.3.1, 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 for them to access it, or at least only temporarily illegal. In fact, because the current role of 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 (Figure 2.5, 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

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 $r \in \text{Reference} - \{r_i \mid (r_i \mapsto _) \in H\}$:

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

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

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:

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

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:

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

Figure 2.5: Definition of the initial state S_0 and first part of the state transition relation \longrightarrow .

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 $R_{\text{ch}}^{\text{RW}}, R_{\text{ch}}^{\text{RO}}$:

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

where

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

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

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

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

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

where

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

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

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

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

Figure 2.6: Second part of the definition of the state transition relation \longrightarrow .

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.

TASK START OPERATION: $\xrightarrow{t \text{ start } t_{\text{ch}}(R_{\text{ch}}^{\text{RW}}, R_{\text{ch}}^{\text{RO}})}$ Whenever a new child task t_{ch} is started, the programmer supplies the *role declarations* for that task, which we model as two sets $R_{\text{ch}}^{\text{RW}}$ and $R_{\text{ch}}^{\text{RO}}$. 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_{ch}^{RW} set become READWRITE in t_{ch} and PURE in t (Figure 2.6, III), which means that t may not access those objects anymore (until t_{ch} finishes; see below) and (2) objects referred to by R_{ch}^{RO} 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_{ch}^{RW} and R_{ch}^{RO} : Objects in R_{ch}^{RW} must play the READWRITE role in t before t_{ch} starts and those in R_{ch}^{RO} 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 this operation does not require any specific declared role for any involved object.

2.4 DETERMINISM

While the formal description in the previous section leaves away many aspects of a real execution of a program, including the actual program instructions and the content of object fields, the description is nevertheless complete enough to formally prove that the model is deterministic. This section presents a sketch of the proof, while the full proof is given in Appendix A.

The proof uses a property we call *child task priority*, which 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:

Child Task Priority

THEOREM 2.4.1 (CHILD TASK PRIORITY). *For any task t and for t 's parent τ , and for all objects O on the current heap, the following holds, for any legal program state: If O 's roles permit τ to write to O , then O 's roles never again permit t to read from or write to O , and if O 's roles permit τ to read from O , then O 's roles never again permit t to write to O .*

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 2.5 and 2.6, the property still holds. In particular, we can show that after a start operation $\tau \text{ start } t(R^{\text{RW}}, R^{\text{RO}})$, task τ may not read from any object in R^{RW} and may not write to any object in R^{RW} or 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 2.4.2 (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., such that 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.

2.5 RELATED WORK

While one of the main strengths of the Parallel Roles model is determinism, it is more generally a model of concurrency for Object-Oriented Programming (OOP). As such, we compare it with other, not necessarily deterministic, models here. The next chapter, which presents a *language* that is based on this model, extends the discussion to concrete languages, including specifically deterministic ones, such as DPJ and ÆMINIUM.

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.

Yonezawa et al. (1986) presented a closely-related model called Active Objects. 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 (Badel 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 of course 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.

3

LANGUAGE

3.1 BASIC LANGUAGE FEATURES

Rolez is a Java-like language that enforces the rules of the Parallel Roles model in every program that is written in it, meaning that every program benefits from the guarantees the model provides. In particular, every Rolez program is guaranteed to be free of interference and is therefore deterministic (unless nondeterminism is introduced explicitly, for example, with a random number generator).

Most role-specific entities defined in Section 2.3.1 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.

OBJECTS AND VALUES Like Java, Rolez programs are hierarchically structured using packages, classes, and then fields, methods, and constructors. Besides objects, which are instantiated using constructors, there are values of primitive types like `int`, `double`, etc., which are created using literals. As in Java, primitive values cannot be shared or mutated (they are always copied) and thus have no roles.

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 3.1 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 adds the role to the corresponding task parameter, 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

```

1 class App {
2   def pure 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: readwrite User
18
19   def readwrite deposit(amount: int): {
20     this.balance += amount;
21   }
22   def readonly getBalance: int {
23     return this.balance;
24   }
25 }

```

Figure 3.1: Rolez code example for tasks and role declarations. Note that void return types can be omitted.

the model in Section 2.3, 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 2.3.2.

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.

As mentioned earlier, the Rolez language includes a type system that detects errors that result from tasks disrespecting their role declarations; for example, if a task declares an object as `READONLY`, it would be an error to write to this object (because an object declared as `READONLY` can never play the `READWRITE` role). To detect such errors, not only task parameters, but all kinds of variables in Rolez that refer to objects (as opposed to primitive values) need to be annotated with a role. In fact, these additional

role annotations are part of the static type of a variable and are called the variable's *static role*. In the case of task parameters, their static roles serve both as part of their type and as their role declarations, as defined by the Parallel Roles model. In Figure 3.1, besides the role declarations mentioned before, the code contains a static role for the receiver of the `calcInterest` method (specified after the `def` keyword on Line 2) and for the `owner` field of the `Account` class (Line 17). Note that local variables generally require no type annotations (and thus no static role), as these can be inferred by the compiler. A more detailed explanation of static roles and the type system in general are provided in Chapter 4.

3.2 JOINT ROLE TRANSITIONS

What differentiates Rolez from many other deterministic languages is how it handles groups of objects. As outlined in Section 1.2, languages that guarantee determinism using compile-time techniques often require the programmer to abstract and declare the heap structure of the objects involved in the execution of the program; for example, DPJ uses the concept of regions for this purpose, while `ÆMINIUM` has data groups. Rolez, on the other hand, does not depend on any static abstraction for grouping or ownership; instead, it employs a dynamic approach that uses the actual runtime structure of the heap, i.e., the actual references between objects.

The basic underlying assumption behind the approach is that an object o_1 which stores a reference to another object o_2 “depends” on o_2 and requires this object for its own functionality. Thus, when o_1 is shared with a task, it is usually desirable to share o_2 with that task as well (and in addition any other object that o_1 can reach via its references). In terms of role transitions, this means that whenever o_1 changes its role, all objects that are reachable from o_1 change their roles in the same way. This concept is called *joint role transitions*.

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 3.1 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. Note that, in this example, the owner of the account is not actually accessed in `payInterest`, so this extra role transition is technically unnecessary. In case such unnecessary role transitions prove to be a problem, either because they inhibit parallel execution or because they cause excessive runtime overhead, Rolez provides some language constructs to refine the set of

objects that are shared together with an object. This refinement is achieved using *slicing*, which is explained in Section 3.3.

How does the concept of joint role transitions relate to the formal definition of Parallel Roles in Section 2.3? As explained earlier, the definition 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 modifying any definition in Section 2.3. 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. (What happens if an object is reachable from a READWRITE *and* a READONLY argument, i.e., if it is in both the R^{RW} and the R^{RO} set, is already defined in Section 2.3.2.) Therefore, objects that are reachable from a task argument not only perform the same role transitions as that argument, but they also have the same (implicitly) declared role.

3.3 SLICING

As explained in the previous section, it is often desirable that a group of reachable objects performs a role transition as if it were a single object. However, this is not always the case. For example, 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. However, with joint role transitions, if all data items are stored in a single data structure (e.g., an array), there is no efficient way to share only a subset of the data items with a task: if the data structure that contains the items is shared with one task, *all* items perform the same role transition and become inaccessible for other tasks.* Therefore, to give the programmer more control over which objects are shared with a task, and in particular to permit efficient implementations of data-parallel algorithms, Rolez contains the concept of *slicing*.

* We assume the general case where full READWRITE access to the data items is required. If the READONLY role is sufficient, then sharing the entire set of data items with multiple tasks is possible anyway.

3.3.1 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 3.3.2.

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.

3.3.2 Formal Definition and Determinism

As with joint role transitions, we define slicing without extending or modifying the formal model definition in Section 2.3. 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 \mathcal{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 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.

3.3.3 Array Slicing

Array slicing in Rolez is supported through 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 3.2 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 or debugging and off for production (if desired).

```

val data = new Array[int](n);
...
val s = n / m; // n be 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);
  ...
}

```

Figure 3.2: Example for contiguous and striped array slicing, for $n = 8$ and $m = 4$.

Note that programmers do not actually have to write code like in Figure 3.2. 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.

3.3.4 Class Slicing

Array slicing deals with arrays; to support a wider 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 programmer-defined 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 3.3.2 applies for class slicing as well, with $\text{elems}(\mathcal{S})$ 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 with tree-like structures instead of with 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 3.3 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 16 and 18. 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.

```

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

```

Figure 3.3: Class slicing example with slice declarations on the left and two slicing operations of the right.

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 3.3, 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 13–18 are not actually necessary, as Rolez infers the types of local variables automatically; they are shown for illustrative purposes.

3.4 EAGER INTERFERENCE CHECKING

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 additional 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.

3.4.1 *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.
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.

```

1  task sort(b: readwrite Slice[int], a: readwrite Slice[int],
2      begin: int, end: int): {
3      if (begin == end) return;
4      // split and recursively sort both runs from a into b
5      val m = (begin + end) / 2;
6      parallel
7          this.sort(a.slice(begin, m), b.slice(begin, m), begin, m);
8      and
9          this.sort(a.slice(m, end), b.slice(m, end), m, end);
10     // merge the resulting runs from b into a
11     this.merge(b, a, begin, m, end);
12 }

```

Figure 3.4: Example usage of the `parallel-and` statement: a (simplified) parallel Mergesort implementation.

3.4.2 *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 3.4 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 3.3.3). 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 6–9). 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 with 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` state-

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

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

where

$$\begin{aligned} H' &\triangleq H / \{r \mapsto \text{obj}_H(r) / \{t :: \text{RO}, t_0 :: \text{RO}\} \mid r \in R_0^{RO}\} \\ &\quad / \{r \mapsto \text{obj}_H(r) / \{t :: \text{PU}, t_0 :: \text{RW}\} \mid r \in R_0^{RW}\} \\ &\quad / \{r \mapsto \text{obj}_H(r) / \{t :: \text{RO}, t_1 :: \text{RO}\} \mid r \in R_1^{RO}\} \\ &\quad / \{r \mapsto \text{obj}_H(r) / \{t :: \text{PU}, t_1 :: \text{RW}\} \mid r \in R_1^{RW}\} \\ &\quad / \dots \\ Ts' &\triangleq Ts - \{T\} \cup \{\text{step}(T)\} \cup \{\langle t_i, t, R_i^{RW}, R_i^{RO}, 0 \rangle \mid i \in 1..n\}, \\ &\quad \{t_i \mid i \in 1..n\} \subset \text{Task ID} - \{t \mid \langle t, _, _, _ \rangle \in Ts\} \end{aligned}$$

Figure 3.5: Eager checking extension for the state transition relation in Figures 2.5 and 2.6. The part that corresponds to the eager interference checks is highlighted.

ment (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 example, this implies that the whole merge method can be executed without guarding, reducing the runtime overhead substantially.

3.4.3 Formal Definition

To define the `parfor` and `parallel`-and constructs formally, we extend the definition of the state transition relation \longrightarrow from Figures 2.5 and 2.6 in Section 2.3 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 3.5; 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 3.5 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.

3.5 EXPRESSIVENESS EVALUATION

To evaluate the expressiveness of Rolez, i.e., what kinds of parallel patterns and algorithms can be expressed with this language, we created a suite of programs, analyzed the parallel patterns they contain, and if and how these patterns can be implemented in Rolez.

The suite consists of a range of programs, including some from well-known benchmark suites: 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). We identified 7 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 3.4.2. 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 indepen-

dently; 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 3.3.3. 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 3.3. 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 3.2), 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.

3.6 RELATED WORK

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 non-deterministic 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 (Lee, 2006; Bocchino et al., 2009a; Lu and Scott, 2011; Devietti et al., 2009).

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 runtime 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 Parallel Roles model has been designed with compile-time checking in mind, as demonstrated by the Rolez type system presented in this chapter.

STATIC EFFECT SYSTEMS To address the problems of dynamic effect specifications, *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 DPJ (Bocchino et al., 2009b; Bocchino and Adve, 2011) 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 specifica-

tions, designed for compile-time checking of the *correctness* of a task’s effect specification, but not for compile-time *noninterference* checking. Instead, noninterference is checked or enforced at runtime, using either guarding or 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., 1998, 2001) 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* (Boyland, 2003; Boyland and Retert, 2005; Bierhoff and Aldrich, 2007). 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 many 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 checked or enforced dynamically, leading to a noticeable but modest runtime overhead, as the performance evaluation in Section 5.3 will show.

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 (Sohi et al., 1995; Rauchwerger and Padua, 1995; Steffan and Mowry, 1998) and Transactional Memory (Herlihy and Moss, 1993; Shavit and Touitou, 1995; Harris and Fraser, 2003) 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 prevented by guarding and, in the case of Rolez, by the type system.

4

ROLE TYPE SYSTEM

4.1 ROLE ERRORS

When compared with languages like DPJ and ÆMINIUM, which check non-interference at compile time, it might seem that Rolez is at a substantial disadvantage in terms of safety. Indeed, the model presented in Chapter 2 contains several state transitions that lead to the S_{error} state, which may seem to imply that there is a class of runtime errors that the programmer needs to carefully avoid. For example, if a task declares one of its parameters as `READONLY`, then it is an error if the corresponding object is ever written to in this task—which applies to the code inside the task body, but also to any code inside methods that are called by the task. Furthermore, when taking joint role transitions into account, this rule applies not only to the very object that is passed as an argument, but also to any *reachable* object. Errors that result from disrespecting the declared role of an object are called *role errors*.

Role Errors

First, note that this issue of role errors is technically distinct from the problem of nondeterminism: role errors are deterministic, i.e., if the programmer wrote code that mistakenly reads from an object declared as `READONLY`, then the program would always fail if this read statement is reached, independently of the scheduling of tasks. The reason is that whether or not a role error occurs depends only the *declared* role of an object, but not on its *current* role. This is not obvious from the definition of the state transition relation in Section 2.3.2, which states, for example, that a write operation causes a transition to S_{error} if neither $\text{mayWrite}_H(t, r)$ (which indeed depends on the current role of the object at r) nor $r \in R^{\text{RW}}$ holds. However, $\text{mayWrite}_H(t, r)$ actually implies $r \in R^{\text{RW}}$, as the proof for Theorem A.3.2 in Appendix A shows. Thus, a write operation would lead to S_{error} if and only if $r \notin R^{\text{RW}}$, i.e., if and only if the object's *declared* role is less permissive than `READWRITE`. Nevertheless, even if role errors are deterministic, they could still be an issue in practice, as they may only be triggered by specific inputs and could therefore be missed during testing.

For this reason, Rolez includes a type system that checks for role 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

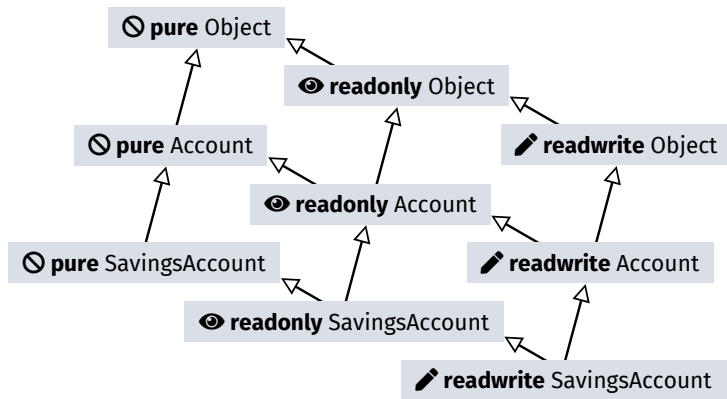


Figure 4.1: Rolez subtyping example.

τ 's execution until t has finished. The type system, on the other hand, detects *erroneous* operations in t , that is, it detects role errors. With this combination of static and dynamic checking, Rolez provides roughly the same safety guarantees as purely statically checked languages like `DPJ` and `ÆMINIUM`, even though Rolez's type system is much more lightweight.

4.2 ROLE TYPES

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 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 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 $r_2 C_2$ 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 , i.e., if r_1 is at least as permissive as r_2 . 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 4.1, 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 this is not shown in the figure.

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 a `readwrite Account` argument, as `readwrite` permits everything that `readonly` permits.

Note that with role types, there are now *three* concepts of roles to distinguish: the *current* role of an object, the *declared* role of an object, and the *static* role of a variable. On the one hand, the current role of an object is always *at most* as permissive as the object's declared role, but it could be *less* permissive, if the object had been shared with another task. On the other hand, the declared role of an object is always *at least* as permissive as the static role of any variable that refers to this object, but it could be *more* permissive, due to subtyping. This leads to the slightly counterintuitive fact that the current role of an object could be *more or less* permissive than the static role of a variable that refers to the object. However, this is not an actual issue, as the type system is only concerned with the declared role of an object, while the discrepancy between the declared and the current role is handled by guarding.

As outlined in Section 3.1, the role declarations of task parameters are integrated into the parameters' type annotations. For example, the `acc` parameter of the `foo` task in Figure 4.2 is declared as `readonly` and hence has the type `readonly Account`. Thus, 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 3.1). This means that 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 exclusively to report role errors at compile time.

4.3 EXAMPLE

In the example in Figure 4.2, 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 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. On the one hand, assigning `acc` to variable `ro` of type `readonly Account` is permitted; on the other hand, assigning it to `rw` is prohibited, because the type of `acc` is not a

```

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 }

```

Figure 4.2: Code example for role types, showing permitted and prohibited uses of variables of class `Account` (defined in Figure 3.1). 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.

subtype of `rw`'s type `readwrite Account`. Note that the standard “right-hand side is a subtype of the left-hand side” rule applies here.

Lines 5–8, together with the definition of the `Account` class in Figure 3.1, 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 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 3.2, 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 4.2, 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

$CL ::= \text{class } C \text{ extends } C \{ \overline{\text{var } f: T} \overline{\text{val } f: T} \overline{M} \}$	classes
$M ::= \text{def } r \ m[\overline{p \text{ incl } q}](\overline{x: T}): T \{ \text{return } e; \}$	methods
$T ::= r \ C$	role types
$r ::=$	roles:
q	built-in role
p	role parameter
$q ::= RW \mid RO \mid PU$	built-in roles
$e ::=$	expressions:
x	variable reference
$e.f$	field read
$e.f = e$	field write
$e.m[\overline{f}](\overline{e})$	method invocation
$\text{new } C$	object creation

Figure 4.3: Syntax of Featherweight Rolez. The parts in blue are related to role parameters and are addressed in Section 4.5.

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 $e.f$ is not simply the role of the field f , but the *less permissive* static role of f and of the target expression e itself. This is illustrated on Lines 10 and 11 in Figure 4.2: 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.

4.4 FORMAL DESCRIPTION

We describe the typing rules of the role type system using a minimal subset of Rolez. The description is inspired by Featherweight Java (FJ), a “minimal core calculus for Java” (Igarashi et al., 2001), and roughly follows the presentation by Pierce (2002). Thus, we call this subset Featherweight Rolez (FR).

Featherweight Rolez

Note that FJ is a “functional” subset of Java, i.e., it includes no notion of assignment and thus no side effects, like the lambda calculus. On the other hand, FR *does* include side effects in the form of field write operations, because side effects are at the core of the problem that Rolez attempts to solve. Therefore, any formal semantics for FR would be much more complex than that for FJ, and we do not provide one here. Consequently, we provide no soundness proof either; however, we argue that a formal description of the type system is valuable nonetheless, especially if it follows a well-known description such as FJ’s.

$$\begin{array}{c}
\hline
\Gamma \vdash r <: r \quad \Gamma \vdash rW <: r \quad \Gamma \vdash r <: \text{PU} \quad \frac{p \text{ incl } p \in \Gamma \quad \rho <: \varrho}{\Gamma \vdash p <: \varrho} \\
\Gamma \vdash C <: C \quad \frac{\Gamma \vdash C <: D \quad \Gamma \vdash D <: E}{\Gamma \vdash C <: E} \\
\frac{CT(C) = \text{class } C \text{ extends } D \{ \dots \}}{\Gamma \vdash C <: D} \quad \frac{\Gamma \vdash r <: s \quad \Gamma \vdash C <: D}{\Gamma \vdash rC <: sD} \\
\hline
\end{array}$$

Figure 4.4: FJ subtyping rules. The rule in blue is related to role parameters, which are addressed in Section 4.5.

4.4.1 Syntax

The syntax of FR is given in Figure 4.3. The metavariables C , D , and E range over class names, f and g range over field names, m ranges over method names, p and q range over role parameter names (as explained later), and x ranges over parameter names (i.e., variables). CL ranges over class declarations, M ranges over method declarations, T , U , V range over role types, r , s , t range over roles, ϱ and ρ range over the built-in roles READWRITE, READONLY, and PURE, and e ranges over expressions. We assume that the set of variables includes the special variable `this`, but that `this` is never used as the name of an explicit parameter of a method; instead, it is implicitly bound in every method declaration. Note that we write “ $\overline{\text{val } f : T}$ ” as shorthand for “ $\text{val } f_1 : T_1 \dots \text{val } f_n : T_n$ ” (similarly $\overline{\text{var } f : T}$ and \overline{M}) and we write “ $\overline{p \text{ incl } \varrho}$ ” for “ $p_1 \text{ incl } \varrho_1, \dots, p_n \text{ incl } \varrho_n$ ” (with commas; similarly $\overline{x : T}$, \overline{r} , \overline{e}). Sequences of field declarations, role parameters, parameters, and method declarations are assumed to contain no duplicate names.

A class declaration CL includes a reference to a superclass, a set of non-final fields (declared using “`var`”), a set of final fields (declared using “`val`”), and a set of methods. A method declaration M first includes the role of the receiver (the `this`), then a list of role parameters, a list of normal parameters, a return type, and finally a body that includes a single return statement with an expression. For now, we ignore role parameters and thus all parts in Figure 4.3 that are given in blue. A role type T includes a role and a reference to a class, as explained in Section 4.2. Finally, an expression e is either a reference to a variable, a field read or write operation, a method invocation, or a constructor invocation.

Note that FR contains no tasks, even though these are a fundamental construct in Rolez. The reason is that, from the point of view of the type system, tasks and methods behave exactly the same; the difference between them is only apparent at runtime: starting a task causes role transitions and changes the declared roles of objects, while a method invocation has no effect on any role.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)} \quad \Gamma \vdash \text{new } C : \text{rw } C \text{ (T-NEW)}$$

$$\frac{\Gamma \vdash e : s C \quad \text{vals}(C) = \overline{f : r C} \quad \Gamma \vdash r_i <: r \quad \Gamma \vdash s <: r}{\Gamma \vdash e.f_i : r C_i} \text{ (T-FREAD)}$$

$$\frac{\Gamma \vdash e : s C \quad \Gamma \vdash s <: \text{ro} \quad \text{vars}(C) = \overline{f : r C} \quad \Gamma \vdash r_i <: r \quad \Gamma \vdash s <: r}{\Gamma \vdash e.f_i : r C_i} \text{ (T-NREAD)}$$

$$\frac{\Gamma \vdash e : \text{RW } C \quad \text{vars}(C) = \overline{f : U} \quad \Gamma \vdash e_R : T \quad \Gamma \vdash T <: U_i}{\Gamma \vdash e.f_i = e_R : T} \text{ (T-WRITE)}$$

$$\frac{\Gamma \vdash e : r C \quad \text{mtype}(m, C) = s \overline{U} \rightarrow T \quad \overline{\Gamma \vdash e : T} \quad \overline{\Gamma \vdash T <: U} \quad \Gamma \vdash r <: s}{\Gamma \vdash e.m(\overline{e}) : T} \text{ (T-INVOKE)}$$

$$\frac{\overline{x : T}, \text{this} : r C \vdash e : U \quad \emptyset \vdash U <: T \quad \text{CT}(C) = \text{class } C \text{ extends } D \{ \dots \} \quad \text{override}(m, D, r \overline{T} \rightarrow T)}{\text{def } r \overline{m(x : T)} : T \{ \text{return } e; \} \text{ OK in } C} \text{ (OK-METHOD)}$$

$$\frac{\text{mtype}(m, D) = s \overline{U} \rightarrow U \text{ implies } r = s \wedge \overline{T} = \overline{U} \wedge T = U}{\text{override}(m, D, r \overline{T} \rightarrow T)} \text{ (OK-OVERRIDE)}$$

Figure 4.5: FR typing rules.

4.4.2 Subtyping

Figure 4.4 shows the formal subtyping rules for FR. Γ is a typing environment that contains the declared types of local variables and the bounds of role parameters, which are used in one of the subtyping rules. Formally, $\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, p \text{ incl } \varrho$.

We use $<:$ for the subrole relation: every role is a subrole of itself, READWRITE is a subrole of every role, and every role is a subrole of PURE. The rule in blue concerns role parameters, which are discussed later. Further, \subset stands for the subclass relation, which is defined as follows: subclassing is reflexive, i.e., every class is a subclass of itself; subclassing is transitive, i.e., if C is a subclass of D and D is a subclass of E , then C is a subclass of E ; and C is a subclass of D if C is declared to extend D . This is formalized using a *class table* CT , which maps class names to their declarations. Finally, $<:$ and \subset are used to define the subtype relation $<:$, as described earlier.

4.4.3 *Typing Rules*

Finally, the typing rules for FR are given in Figure 4.5. The type of a variable is simply that variable's declared type (T-VAR). When an object is instantiated, the resulting type is READWRITE C , where C is the instantiated class (T-NEW). This corresponds to the fact that the declared role of a newly created object is READWRITE, as defined in Figure 2.5 in Chapter 2.

When reading from a field of an object, there are two cases: reading from a final or from a non-final field. In both cases, the target expression e needs to be of a type $s C$ and the field f_i must exist in C (we use again the notation $f : r C$ as a shorthand for $f_1 : r_1 C_1 \dots f_n : r_n C_n$). The terms $\text{vals}(C)$ and $\text{vars}(C)$ denote the set of final and non-final fields in C , respectively. Notably, the resulting role r of a field read expression is not simply equal to the declared role r_i of the field; instead, it is a (possibly non-strict) superrole of r_i and, more importantly, it is also a superrole of s , the role of the target expression. This typing rule reflects Rolez's concept of joint role transitions: An object that is reachable from a task argument has the same declared role as that argument. Thus, the type system needs to ensure that such an object cannot be referred to by an expression that has a more permissive role than that of the argument. By ensuring that the static role of a field read expression is always a superrole of that of the target expression, this property holds. For example, when reading a field from a READONLY target, the resulting role is *at most* READONLY (but it could be PURE, if the field is declared as PURE). The only difference when reading from a non-final field (T-NREAD) as opposed to reading from a final field (T-FREAD) is that the role s of the target expression must be at least READONLY. Reading a final field is always legal, even when the target is PURE.

The typing rule for writing to a field (T-WRITE) includes the standard "right-hand side is a subtype of the left-hand side" rule, as well as the requirement that the target field must be non-final. In addition, the role of the target expression e must be READWRITE.

Typing method invocations is slightly more involved. Given that the target expression is of type $r C$, the rule uses the function $\text{mtype}(\dots)$ to determine the "type signature" of the method. This signature $s \bar{U} \rightarrow T$ includes the receiver role s , the types \bar{U} of all method parameters, and the return type T . Then, the rule states that each of the types \bar{T} of the method arguments is a subtype of the corresponding parameter type, and that the role r of the target expression is a subrole of s .

Rule OK-METHOD defines well-typed methods and "bootstraps" the typing of an expression e , defining the typing environment Γ that is used in all typing rules. Γ contains the declared types \bar{T} of all explicit method parameters \bar{x} , plus the type $r C$ of the implicit *this* parameter, where r is the declared role for *this* and C is the class in which the method is declared. In addition, the rule includes two premises concerning method overriding. Valid overriding is defined in OK-OVERRIDE, stating that if a method has

the same name as a method in a superclass, then all parameter types, the return type, and the `this` role must be the same as in that method.

We omit the definitions of the various auxiliary functions like `mtype(...)`, as these differ only insignificantly from their counterparts in FJ (Pierce, 2002).

4.5 ROLE PARAMETERS

Figures 4.3 and 4.4 contain elements that are still unexplained and are related to *role parameters*. (These elements are given in blue.) Role parameters are an advanced feature of the Rolez type system that allows methods to operate on objects with different roles without sacrificing type safety (or introducing code duplication). Role parameters share some similarities with type parameters in Java and other object-oriented languages, but they address a much more specific, but ubiquitous problem.

Role Parameters

4.5.1 *Example*

As explained above, the static role of a field read expression depends on the role of the target. For example, if the declared type of a field `f` is `readwrite A`, then the expression `e.f` has the same static role as the target expression `e` itself. If a programmer wanted to encapsulate this field access in a getter method `get`, it is unclear what roles to use for `get`: On the one hand, if the method is declared like this:

```
def readwrite get(): readwrite A {
    return this.f;
}
```

then the method can only be invoked on `READWRITE` targets. On the other hand, if the `this` role is changed to `READONLY`, then the return type needs to be changed to `READONLY` too, otherwise the method body would not be well-typed. One could work around this problem using casts or even by defining two (or three) versions of the method, but such measures seem disproportionate for a simple getter method.

With role parameters, the `get` method can be defined in a generic and type-safe way:

```
def r get[r](): r A {
    return this.f;
}
```

The square brackets enclose the role parameters of a method, like the parentheses enclose the regular parameters. Here, a single role parameter `r` is defined, and used both as the `this` role and as the role of the return type. Thus, when this method is invoked on a `READWRITE` target, its return type is `readwrite A`, when invoked on a `READONLY` target, the return type is `readonly A`, etc.

$$\begin{array}{c}
\Gamma \vdash e : r C \quad \text{mtype}(m, C) = t \overline{p \text{ incl } \varrho} \overline{V} \rightarrow U \\
\overline{\Gamma \vdash e : T} \quad \overline{U = V \setminus \overline{p \mapsto r}} \quad \overline{\Gamma \vdash T <: U} \\
s = t \setminus \overline{p \mapsto r} \quad \Gamma \vdash r <: s \quad T = U \setminus \overline{p \mapsto r} \quad \overline{\Gamma \vdash r <: \varrho} \\
\hline
\Gamma \vdash e.m[\overline{r}](\overline{e}) : T \quad (\text{T-INVOKE})
\end{array}$$

$$\begin{array}{c}
\overline{p \text{ incl } \varrho}, x : \overline{T}, \text{this} : r C \vdash e : U \quad \overline{p \text{ incl } \varrho} \vdash U <: T \\
CT(C) = \text{class } C \text{ extends } D \{ \dots \} \\
\text{override}(m, D, r \overline{p \text{ incl } \varrho} \overline{T} \rightarrow T) \\
\hline
\text{def } r \ m[\overline{p \text{ incl } \varrho}](x : \overline{T}) : T \{ \text{return } e; \} \text{ OK in } C \quad (\text{OK-METHOD})
\end{array}$$

$$\begin{array}{c}
\text{mtype}(m, D) = s \overline{q \text{ incl } \rho} \overline{U} \rightarrow U \text{ implies} \\
r = s \wedge \overline{p} = \overline{q} \wedge \overline{\varrho} = \overline{\rho} \wedge \overline{T} = \overline{U} \wedge T = U \\
\hline
\text{override}(m, D, r \overline{p \text{ incl } \varrho} \overline{T} \rightarrow T) \quad (\text{OK-OVERRIDE})
\end{array}$$

Figure 4.6: FR typing rules with role parameters, which replace the respective rules in Figure 4.5.

However, this generic definition of `get` is only well-typed if the field `f` is final; otherwise the `T-NREAD` rule applies, which states that the role of the target must be at least `READONLY`, which does not apply to the role parameter `p`, which could be instantiated with any role. To address this issue, role parameters can have *upper bounds*. For example, `r` can be defined to be *at least as permissive* as `READONLY`:

Role Parameter Bounds

```

def r get[r includes readonly](): r A {
  return this.f;
}

```

Now, `get` can only be called on a `READWRITE` or `READONLY` target, making the field read operation safe, even when `f` is non-final. Role parameter bounds work in the same way as type parameter bounds known from other OOP languages, in that they restrict the set of arguments that a parameters can be instantiated with.

4.5.2 Formal Description

Role parameters are the final missing piece in Featherweight Rolez. To describe them formally, we include the blue parts in Figures 4.3 and 4.4, and we replace `T-INVOKE`, `OK-METHOD`, and `OK-OVERRIDE` in Figure 4.5 with the versions shown in Figure 4.6.

The complete syntax of FR now includes role parameters (including bounds) in method declarations and role arguments in method invocations. In addition, wherever a role `r` can occur, a role parameter `p` can be used instead of a built-in role `q`. The additional subtyping rule in Figure 4.4 states

that a role parameter p is a subrole of a built-in role ρ if p 's upper bound is a subrole of ρ . The upper bounds of all declared role parameters are included in the typing environment Γ .

The new T-INVOKE rule in Figure 4.6 shows how method invocations with role parameters are typed. First, the type signature of a method now also includes all role parameter names and their upper bounds. Second, since the `this` role, the parameter types, and the return type can contain references to a role parameter, there are three new premises that relate the parameterized types to the instantiated types, i.e., the types where all occurrences of a role parameter p have been replaced by the respective role argument r . For example, the premise $T = U \setminus \overline{p} \mapsto r$ states that T is equal to U with all occurrences of p_1, \dots, p_n replaced with the respective r_1, \dots, r_n . Finally, the last premise states that each role argument r must respect the bound of the respective parameter p .

OK-METHOD is updated to include the role parameters and their bounds in the typing environment when typing the expression e and when checking that $U <: T$, because both U and T could contain references to a role parameter. In addition, the override rule now includes premises stating that all role parameter names and bounds must be equal to the corresponding ones of the overridden method. Since role parameters are instantiated based on their positions, the names could actually be chosen differently, but this would complicate the rest of the premises.

Note that in FR, role arguments must be specified explicitly for every method invocation. However, in Rolez, the compiler can infer them automatically, much like type arguments to methods are inferred automatically in Java. Thus, the syntactic overhead for role-parameterized methods is relatively low.

4.6 RELATED WORK

One of the main distinguishing features of the Rolez type system is the sound handling joint role transitions, which is accomplished with a special typing rule for field read expressions and, for getter-like methods, with role parameters. In essence, the Rolez type system guarantees that the restrictions imposed by the static role of a reference r apply not only to the state of the object represented by r itself, but to the *transitive state* of r , i.e., to the state of all objects reachable from r .

Such a guarantee is also useful for other reasons than safe parallel execution, for example, for protecting the state of an object from unauthorized modifications by untrusted code, or simply for ensuring encapsulation. Recognizing this fact, an extension for Java, called Java with Transitive Readonly Access Control (JAC) (Kniesel and Theisen, 1999), employs a type system where for every class type C , there is also a variant `readonly C`, which prohibits modification. As the name of the language suggests, `readonly` is transitive, i.e., when reading from a field of a `readonly` type, the resulting type is again `readonly`. While this type system bears a certain resemblance

to Rolez’s, it is less expressive: First, it includes only two types of “roles”, `readonly` or full access, while Rolez includes an even stricter role, `PURE`. Second, JAC’s handling of method calls differs substantially from Rolez’s: When calling a method on a `readonly` target in JAC, by default, the result type is `readonly` as well, assuming that methods generally return objects that belong to their transitive state. Programmers may override this behavior, but only if the returned object is unconditionally mutable. Rolez, on the other hand, allows more flexibility: using role parameters, a programmer can link the role of a method’s return type not only to the role of the method target, but also to the role of any other method parameter.

The more general idea behind Rolez’s and JAC’s type systems is that the way a reference may be used depends on the “context” from where the reference was obtained: for example, if the reference was read from the field of a `READONLY` reference, it may not be used to modify the object; but if it was obtained, for example, by creating a new object, it grants full access. This context-dependent behavior was explored in the Accessibility-Based Encapsulation (ACE) model (Kniesel, 1996), which JAC is based on. In a broader context, the idea is also used in the Java Security Architecture (Oracle Corporation, 2018), where “the permission set of an execution thread is considered to be the intersection of the permissions of all protection domains traversed by the execution thread.”

The idea of “adapting” the role of a field read expression $e.f$ to the role of the target e is also similar to the concept of *viewpoint adaptation* known from ownership type systems, in particular Generic Universe Types (Dietl et al., 2007). That type system includes *ownership modifiers*, which accompany reference types, similar to static roles in Rolez. The ownership modifier of a field read $e.f$ depends on the declared modifier of field f as well as on the modifier of e . In contrast to Rolez, where static role “adaptation” applies only to field reads and (using role parameters) to method results, viewpoint adaptation in Generic Universe Types is a more ubiquitous concept that applies also to method parameters and type parameters.

5

IMPLEMENTATION

The main challenge that comes with the Parallel Roles model is performance. In contrast to static DPP approaches, this model relies on runtime mechanisms like role transitions and guarding. These mechanisms cause a runtime overhead in Rolez programs when compared to statically checked languages like DPJ or ÆMINIUM, or nondeterministic languages like Java. To evaluate this overhead, and the performance of a Rolez-like language in general, we implemented a prototype compiler and runtime system for Rolez, based on the Java platform (Section 5.1).

This overhead depends heavily on the details of the implementation. A straight-forward implementation would be to store both the declared and the current roles of each object (for each task) alongside the object's fields, and to insert a check before every field access operation. Such a check would retrieve the object's declared and current role for the currently executing task and then determine whether the operation is legal, temporarily illegal, or erroneous, as defined in Section 2.3.2 (Figure 2.5, III and IV). Such an implementation would incur an enormous overhead, easily negating the gains from parallel execution.

However, there are various avenues to improve over such an inefficient implementation: Section 5.2 presents two specific optimizations that improve the performance of our prototype substantially. For many programs, the combination of these optimization raises the Rolez performance almost to the level of unchecked languages like Java, as we show in the performance evaluation in Section 5.3.

5.1 PROTOTYPE OVERVIEW

The Rolez compiler prototype is implemented using the Xtext framework (Eclipse Foundation, 2006) and takes as in input a set of Rolez source files which it transforms into a set of Java source files. These Java files can be compiled with a standard Java compiler and executed on a standard, unmodified Java Virtual Machine (JVM). The runtime system is implemented as a Java library.

OBJECTS AND ROLES For every Rolez class, the compiler generates a corresponding Java class, with the same fields and methods. To keep track of the roles of an object, the Rolez runtime library contains a class called

```

class Account {
  var balance: int
  def readwrite deposit(amount: int): {
    this.balance += amount;
  }
  def readonly getBalance(): int {
    return this.balance;
  }
}

class Account extends Guarded {
  int balance;
  void deposit(int amount) {
    this.guardReadWrite();
    this.balance += amount;
  }
  int getBalance() {
    this.guardReadOnly();
    return this.balance;
  }
}

```

Figure 5.1: Rolez and generated Java class.

rolez.lang.Guarded, which every generated Java class extends (directly, if the corresponding Rolez class extends the root class rolez.lang.Object, or else indirectly).

The Guarded class contains a set of fields that represent the roles of an object. As explained above, a straight-forward implementation would store both the current and the declared roles in every object. However, in combination with the Rolez type system, a more efficient scheme is possible: to store only the *current* roles inside an object. The *current* role of an object for any task is required to perform guarding in that task, i.e., to differentiate whether an operation is legal or temporarily illegal. By contrast, the *declared* role is not required for guarding or any other runtime check. According to Section 2.3.2, it is used to determine whether an operation is *erroneous*, i.e., to detect role errors. But since role errors are detected by Rolez’s type system, the program is guaranteed to be free of role errors once it has passed the Rolez compiler, and thus no checks for role errors need to be performed at runtime. Consequently, the fields of the Guarded class store only the *current* role of an object.

GUARDING Ignoring for now the optimizations presented later, the Rolez compiler inserts a guard for *every* read or write access to an object. Because all generated classes extend Guarded, all objects are instances of that class and can be used as targets for the methods of that class. Thus, guards are implemented simply as calls to Guarded methods, with the same target as that of the guarded operation.

Figure 5.1 illustrates these points using a simple Account class. The original Rolez class is shown on the left side and the generated Java class on the right. Note that the generated class is simplified, leaving away various aspects that are discussed in the next section. In both methods, one guard is inserted. Two kinds of guards exist: `guardReadWrite` for operations that require the READWRITE role and `guardReadOnly` for operations that require the READONLY role. (PURE never requires guarding, as it is the least permissive role possible.) Technically, the `+=` and `-=` operators correspond to *two*

```

task computeInterest(acc: readonly Account): int {
    // task code
}
def foo(): int {
    var acc = new Account(20000);
    var interest = this start computeInterest(acc);
    // ...
    return interest.get();
}

```

```

1 Task<Integer> computeInterest(final Account acc) {
2     return new Task<>(new Object[] {}, new Object[] {acc}) {
3         protected Integer runRolez() {
4             // generated task code
5         }
6     }
7 }
8 int foo() {
9     Account acc = new Account(20000);
10    Task<Integer> interest = this.computeInterest(acc);
11    TaskSystem.start(interest);
12    // ...
13    return interest.get();
14 }

```

Figure 5.2: Example of a declaration and invocation of a Rolez task and the corresponding generated Java code.

operations, a read and a write. However, since the `READWRITE` role permits both reads and writes, only one `guardReadWrite` call is inserted.

TASKS In addition to fields, methods, and constructors, Rolez classes may contain tasks. Tasks are invoked using the usual method invocation syntax, except that the `start` keyword replaces the dot. A `start` expression results in an object of type `rolez.lang.Task[V]`, which acts as a wrapper around the (future) result of the task. The type parameter `V` corresponds to the return type of the task.

A task is transformed into Java by creating a method that explicitly returns a `rolez.lang.Task` object, which is an instance of a class in the Rolez runtime library. This `Task` class is similar to the `FutureTask` class in the Java standard library, in the sense that it represents asynchronous computations. However, when a `Task` object is created, all (non-primitive) arguments to that task must be passed to the constructor, grouped by the roles that the programmer declared for the corresponding task parameters. This enables the `Task` constructor to perform the role transitions for these arguments. Finally, a `start` expression is transformed into Java simply by calling the corresponding method and starting the code in the resulting `Task` object in a different thread.

Figure 5.2 illustrates these points. The top part shows the declaration of a task and a method that starts it, while the bottom part shows the generated Java code. Note that the `Task` constructor that is invoked on Line 2 takes two arrays as arguments. The first contains all arguments declared as `READWRITE`, while the second contains those declared as `READONLY`. In this example, the first array is empty and the second contains the `acc` argument. Also note that tasks are started using a method in a `TaskSystem` class (Line 11). This class is also part of the Rolez runtime library and can be used to execute a task in a new thread or in a thread from a thread pool. A task is executed by calling the `runRolez` method, which contains the generated code of the task (Line 3). As soon as the `runRolez` method has finished, its return value is stored in the `Task` instance and can be retrieved using the `get` method, as shown on Line 13.

5.2 OPTIMIZATIONS

The Rolez prototype is designed to be independent of any particular JVM. In particular, it does not rely on any JVM-implementation-specific feature, and it requires no modifications to the JVM itself. This design has a number of advantages: First, it allows full compatibility with other languages that run on the JVM. Since Rolez classes are transformed into Java classes, it is possible to write the parallel parts of a program in Rolez, and implement the rest in, e.g., Java or Scala. Second, the approach enables the use of existing software tools; for example, Rolez code can be tested using frameworks like JUnit or TestNG and analyzed using Java debuggers or code coverage tools. Finally, the approach results in great portability. Because the implementation does not require any modifications to a JVM component, such as the just-in-time (JIT) compiler, Rolez programs that are compiled to Java can be executed on any standard JVM.

However, since a VM-independent implementation cannot rely on custom JIT optimizations, achieving efficiency is much more difficult. The following optimization techniques address this issue. While they were developed specifically for Rolez, some of them are also applicable to VM-independent implementations of other parallel languages.

5.2.1 *Code-Managed Runtime Data*

To achieve high performance with Rolez, efficient guarding is paramount, because a guard is conceptually required for every single read or write access to an object. And even though many redundant guards can be eliminated using concurrency analysis (Section 5.2.2), some guards may remain.

GUARDING OVERVIEW Guarding is implemented in the `Guarded` class of the Rolez runtime library, which contains fields and methods for maintaining and checking the roles of an object. As all Rolez objects are instances of

a subclass of `Guarded`, they inherit these fields and methods. The following snippet shows how the `Guarded` class represents the roles of an object, using two fields. Note that the actual implementation is more complex, but this simplified version is more suitable for illustration purposes.

```
public abstract class Guarded {
    private volatile int ownerId;
    private AtomicInteger readers;
    // ...
}
```

First, the `ownerId` field stores the ID of the current *owner* task of the object. This is the single task in which the object currently plays the `READWRITE` role or, if the object was shared as `READONLY`, the last task in which it played the `READWRITE` role.* Second, `readers` stores the number of tasks in which the object currently plays the `READONLY` role. This information is sufficient to determine all roles of an object.

Besides the fields that store an object's roles, the `Guarded` class contains the methods that implement role transitions and, most importantly, guarding. A guard operation checks whether the role of the target object in the current task permits the given operation and, if not, blocks the execution of the current task. Using the above representation of roles, guarding for `READWRITE` can be implemented as follows:

```
public final void guardReadWrite() {
    while(!isReadWrite()) { LockSupport.park(); }
}
private boolean isReadWrite() {
    long taskId = /* get ID of current task */;
    return this.ownerId == taskId && this.readers.get() == 0;
}
```

The `isReadWrite` method checks if “this” object plays the `READWRITE` role by comparing the ID of the current task with that in the object's `ownerId` field. In addition, it checks that the object is not currently shared as `READONLY`, in which case it would currently also play the `READONLY` role in the owner task. The `guardReadWrite` method simply parks the currently executing thread, using the `LockSupport.park` method from the Java standard library, until `isReadWrite` returns true. Whenever a role transition happens, parked threads are unparked so that they can check the role again. The `guardReadOnly` method is implemented similarly.

CODE GENERATION FOR THE CURRENT TASK The challenging part in the guarding implementation is to efficiently retrieve the “current” task, i.e., the task in which the given code is being executed. This information could be stored in a thread-local variable, e.g., using the `ThreadLocal` class from the Java standard library. However, we found that reading such a thread-local variable is relatively slow in comparison with a normal heap read or write. Thus, reading a thread-local variable for every guarded operation would be inefficient.

* There is always a task in which an object last played the `READWRITE` role, as every object initially plays the `READWRITE` role in the task that creates it.

Instead of solving this issue purely in the runtime system, we leverage a code generation technique: To each Java method generated from a Rolez method, the compiler adds an additional parameter for the ID of the current task. Then, each generated method invocation passes the current task in the enclosing method on to the invoked method. Finally, when a guard is inserted, the current task in the enclosing method is passed on to the `guard*` method.

Figure 5.3 shows how this mechanism works, using two example classes written in Rolez and the corresponding generated Java classes. Lines 11 and 16 show how an additional `$task` parameter is added to methods, while Lines 6 and 7 show how these methods are called. The actual ID of the currently executing task is retrieved exactly once, at the beginning of a task (Line 5). Finally, Line 17 shows how the ID is ultimately passed to `guardReadOnly`, which guards the read operation in the `getBalance` method. The `guardReadOnly` method simply passes the ID on to `isReadOnly`, where it is compared with the owner of the object:

```
public final void guardReadOnly(int $task) {
    while(!isReadOnly($task)) { LockSupport.park(); }
}
private boolean isReadOnly(int $task) {
    return this.ownerId == $task || this.readers.get() > 0;
}
```

Adding a method parameter to *every* method in the program may seem inefficient as well. In addition, the code snippets may imply that guarding could be implemented using, e.g., the ID of the current native Java thread, which is typically efficient to retrieve. However, the actual Guarded implementation is more complex and needs to store not only the *number* of readers, but the *set* of readers. To query this set efficiently, it is implemented as a bit set, relying on specific properties of Rolez task IDs. Since Rolez code could be executed in arbitrary threads, using the thread ID would not work. In addition, we have conducted experiments on the JVM that showed that adding a parameter to all methods of a program has no significant performance impact and that this approach is at least as efficient as using the thread ID.

Once inlined into the code that performs the guarded operation, a guard consists only of at most three field reads and a few comparisons. And because the `ownerId` and `readers` fields are in the same object as the field that is accessed by the guarded operation, they are likely to be stored in memory close-by and will benefit from CPU caching. Thus, guards result in very little overhead, despite being implemented without the explicit help of a JIT compiler.

5.2.2 Concurrency Analysis

Even though guards can be implemented efficiently, actually guarding every object access would still result in poor performance. Fortunately, this is

```

class Banking {
  task computeInterest(acc: readonly Account): int {
    val balance = acc.getBalance();
    return complexComputation(balance);
  }
  def complexComputation(balance: int): int { /*...*/ }
}
class Account {
  var balance: int
  // ...
  def getBalance(): int { return this.balance; }
}

```

```

1 class Banking {
2   Task<Integer> computeInterest(final Account acc) {
3     return new Task<>(new Object[]{}, new Object[]{acc}) {
4       protected Integer runRolez() {
5         int $task = this.id();
6         int balance = acc.getBalance($task);
7         return complexComputation(balance, $task);
8       }
9     }
10  }
11  int complexComputation(int balance, int $task) { /*...*/ }
12 }
13 class Account {
14   int balance;
15   // ...
16   int getBalance(int $task) {
17     this.guardReadOnly($task);
18     return this.balance;
19   }
20 }

```

Figure 5.3: Code generation for passing the current task from method to method. On the top are two Rolez classes and on the bottom the corresponding generated Java code.



not required, as many guards are redundant and can be eliminated, using *concurrency analysis*.

Guards prevent *temporarily illegal* operations. Such operations are illegal because the target object has been shared with a child task and currently plays a less permissive role than its declared one. The main insight behind concurrency analysis is that an operation in some task t can only be temporarily illegal if t has started a child task before and that child task has not yet finished. Or conversely, if there exists no child task, all objects are guaranteed to play the role that was declared for them. Concurrency analysis statically determines if there possibly exists any child task, for every point in the program. Where this is not the case, the compiler does not emit any guards.

MODULAR INTERPROCEDURAL ANALYSIS To be useful, concurrency analysis needs to be interprocedural, as a sound *intra*-procedural analysis would have to assume that there already exists a child task at the beginning of a method's execution. However, interprocedural analysis is usually expensive and precludes modular compilation, a standard feature for Java-like languages. The concurrency analysis is interprocedural *and* modular, offering the best of both worlds. This is achieved again using a co-design with code generation.

The key insight to make concurrency analysis both interprocedural and modular is that the analysis computes only a single bit of information per program point: whether there is at least one child task or not. This boolean information can be propagated through the program without actually analyzing the whole program at once. Instead, every method is analyzed using an intraprocedural version of the analysis, but twice: once under the assumption that there *are* child tasks at the beginning of the method, and once under the opposite assumption. Propagating the information through the program is done in two steps: First, the compiler generates two versions of every method, one for each assumption. And second, for every method call, the compiler generates a call to the version that matches the information computed for that program point by the intraprocedural analysis.

Because two different versions are generated for every method, the analysis is not only interprocedural, but also context-sensitive. When a method is called from a context with child tasks, the version with guarding is used, and when called from contexts without child tasks, the unguarded version. This is important for programs written in languages similar to Java, because these languages typically come with an extensive standard library that includes classes and methods that are used in many different parts of a program.

EXAMPLE We illustrate concurrency analysis using the program in Figure 5.4. On the left, the figure shows the results of the intraprocedural version of the concurrency analysis. The  symbol means that there possibly exist child tasks and thus guarding is necessary, while the  symbol

	N	Y	
1			task main(): {
2	—		process(/*...*/);
3			}
4			def process(accounts: Array[Account]): {
5	—	⊙	printTotal(accounts);
6		⊙	for (var i = 0; i < accounts.length; i++)
7	⊙	⊙	start depositInterest(accounts[i]);
8	⊙	⊙	printTotal(accounts);
9			}
10			def printTotal(accounts: Array[Account]): {
11	—	⊙	var total = 0;
12	—	⊙	for (var i = 0; i < accounts.length; i++)
13		⊙	total += accounts[i].balance;
14	—	⊙	println(total);
15			}
16			task depositInterest(acc: readwrite Account): {
17	—		val balance = acc.balance;
18	—		acc.deposit(/*...*/);
19			}

Figure 5.4: Concurrency analysis example.

means the opposite. There are two columns: N shows the result of the analysis under the assumption that there are *no* child tasks at the beginning of a method, and Y shows the results for the opposite assumption. Column Y is empty for tasks; because it is impossible for new tasks to have any child tasks, there is no need to analyze a task under this assumption.

When the program begins execution in the `main` task, there are no child tasks, so no guarding is necessary on Line 2. The `process` method is analyzed twice. Assuming no child tasks exist initially, no guarding is necessary on Line 5, as shown in Column N. However, since tasks are started inside the loop (Line 7), guarding is necessary for the whole loop and subsequently on Line 8. Under the opposite assumption, that child tasks initially exist, guarding is necessary throughout the whole method, as shown in Column Y. The `printTotal` method is also analyzed twice. Since no tasks are started inside the method, guarding is necessary or unnecessary, respectively, throughout the whole method, depending on the initial assumption. Finally, `depositInterest` is again analyzed only once, because it is a task. Even though there may be other tasks in the program by the time a `depositInterest` task is started, every instance of the task is guaranteed to have no *child* tasks when it starts execution. And because `depositInterest` does not start any tasks itself, no guarding is required throughout the whole task body.

The resulting generated code for this program is shown in Figure 5.5.* Two versions are generated for each method, an `$Unguarded` and a `$Guarded` one, while there is only one version for tasks. Whenever a method is called, the compiler generates a call to the version that matches the results of the analysis shown in Figure 5.4. For example, in `process$Unguarded` on Line 10,

* This code is simplified, leaving away the `$task` parameters introduced in Section 5.2.1 and simplifying the handling of arrays: the compiler wraps Java arrays in instances of the `GuardedArray` class (a subclass of `Guarded`) and inserts guards where necessary.

```

1 Task<Void> main() {
2     return new Task<Void>(new Object[] {}, new Object[] {}) {
3         protected Void runRolez() {
4             process$Unguarded(/*...*/);
5             return null;
6         }
7     }
8 }
9 void process$Unguarded(Account[] accounts) {
10    printTotal$Unguarded(accounts);
11    for (int i = 0; i < accounts.length; i++)
12        TaskSystem.start(depositInterest(accounts[i]));
13    printTotal$Guarded(accounts);
14 }
15 void process$Guarded(Account[] accounts) {
16    printTotal$Guarded(accounts);
17    for (int i = 0; i < accounts.length; i++)
18        TaskSystem.start(depositInterest(accounts[i]));
19    printTotal$Guarded(accounts);
20 }
21 void printTotal$Unguarded(Account[] accounts) {
22    int total = 0;
23    for (int i = 0; i < accounts.length; i++)
24        total += accounts[i].balance;
25    System.out.println(total);
26 }
27 void printTotal$Guarded(Account[] accounts) {
28    int total = 0;
29    for (int i = 0; i < accounts.length; i++) {
30        accounts[i].guardReadOnly();
31        total += accounts[i].balance;
32    }
33    System.out.println(total);
34 }
35 Task<Void> depositInterest(Account acc) { /* ... */ }

```

Figure 5.5: Generated code for the program in Figure 5.4.

$T ::=$	<code>task $t()$ { \bar{s} }</code>	task
$M ::=$	<code>def [async] $m()$ { \bar{s} }</code>	method
$S ::=$		statements:
	<code>start $t()$;</code>	task start
	<code>$m()$;</code>	method invocation

Figure 5.6: Simplified Rolez syntax for the dataflow analysis.

the `printTotal$Unguarded` is called, because, for this version of the process method, the analysis assumed that there are no child tasks at the beginning of the method. However, after process has started some tasks, the guarded version of `printTotal` is called (Line 13), as the roles of objects may now differ from their declared ones. On the other hand, in `process$Guarded`, both calls to `printTotal` use the guarded version (Lines 16 and 19), because the analysis assumed that there exist child tasks at the beginning of this version of the process method. Note that the `process$Guarded` method is not actually used anywhere in the program, but is still generated in case the process method is used in another Rolez program, where it may be called from a guarded context.

Finally, the two generated versions of `printTotal` illustrate how the analysis actually eliminates guards. While the guarded version contains a guard that protects the access to the `Account.balance` field (Lines 30 and 31), the unguarded version accesses the field without a guard (Line 24), because for this version there cannot be any child tasks. Hence, the information that there are no child tasks is propagated all the way from the main task to this field access, even though every method is analyzed and compiled in isolation.

DATAFLOW ANALYSIS The intraprocedural analysis that is illustrated in Figure 5.4 can be expressed as a forward dataflow analysis (Nielson et al., 1999), which is performed for every method (twice) and every task (once). The analysis tracks the program state of interest, i.e., whether there are child tasks, along the edges of the control flow graph. To deal with loops, the dataflow analysis follows an iterative fixed-point algorithm.

To present the analysis concisely, we focus on a very small subset of the Rolez language (much smaller than Featherweight Rolez). This subset includes only tasks and methods, and as statements just task starts and method invocations, as shown in Figure 5.6. Statements that affect control flow, like `if-else` statements or loops are not included explicitly, but are handled generally by combining program states when control flow joins. All other kinds of statements that Rolez supports have no effect on the program state computed by the analysis and are omitted here.

As described in the literature (Nielson et al., 1999), the dataflow analysis computes for each statement in the control flow graph (CFG) an *in-state* and

Program state domain: $s \in \{\top, \perp\}$

Transfer functions:

$$f_{\text{start } t();}(s) \triangleq \top$$

$$f_{m();}(s) \triangleq \begin{cases} \top, & \text{if } \text{async}(m) \\ s, & \text{otherwise} \end{cases}$$

Initial and entry state:

$$s_{\text{init}} \triangleq \perp$$

$$s_{\text{entry}} \triangleq \begin{cases} \top, & \text{assuming there are} \\ & \text{initially child tasks} \\ \perp, & \text{otherwise} \end{cases}$$

Combination operator: $\sqcup \triangleq \vee$

Figure 5.7: Dataflow analysis definition.

an *out-state*, which represent the state of the program before and after a statement, respectively. We use $s \in \{\top, \perp\}$ to denote the program state, as shown in Figure 5.7. If $s = \top$, then there are possibly some child tasks; if $s = \perp$, there are none. A set of dataflow equations defines how a statement S transforms the state of the program, using a set of transfer functions f_s . The out-state of a statement is the result of applying the transfer function to the in-state.

For a start statement, the program state changes to \top , as there is now at least one child task. For a method invocation, the program state depends on the declaration of the method. By default, a Rolez method implicitly joins all newly started tasks at the end of the method. Therefore, if there were no child tasks before the method invocation, there are again none afterwards. To override this behavior, a method can be declared as `async`, which allows newly started tasks to continue to execute after the method has returned. Thus, the program state depends on the presence of the `async` keyword, which is expressed using the $\text{async}(m)$ function.

The dataflow analysis performs a fixed-point iteration over all statements in a method's CFG, applying the dataflow equations repeatedly until all in- or out-states are stable. In- and out-states are initialized with the s_{init} state, except for the in-state of the first statement in a method, which is initialized with s_{entry} instead. The entry state s_{entry} depends on the initial assumption of the analysis: for methods, the analysis is performed once with \top and once with \perp , while tasks are analyzed once, using \perp . When control flow joins, e.g., after an `if-else`, the combination operator \sqcup combines the out-states

of the predecessor statements. Since the analysis is conservative, \sqcup is the logical disjunction, i.e, if there may exist child tasks on at least one incoming path, the analysis must assume that they may also exist after the join.

5.3 PERFORMANCE EVALUATION

The performance evaluation of the Rolez prototype addresses the following questions:

1. How much performance can be gained using a dynamically-checked, deterministic language like Rolez, when compared with an (unchecked) *sequential* implementation? One aspect of this question, how much parallelism can be expressed using Rolez, has already been addressed qualitatively in Section 3.5.
2. How much runtime overhead is caused by runtime checking due to role transitions and guarding, when compared with an unchecked *parallel* implementation, e.g., in Java. This question also addresses the simplicity–performance trade-off when comparing Rolez with statically-checked languages like DPJ and ÆMINIUM: because statically-checked languages add no (or very little) runtime overhead, we can assume that their performance is virtually the same as for unchecked languages. In addition, we investigate the overhead caused by the runtime checks due to array slicing, even though these checks are not specific to Rolez, but would affect also other languages that guarantee safe access to any kind of “sub-array”.
3. How effective are the presented optimizations in reducing the runtime overhead of the Rolez prototype?

To answer these questions, the parallel programs described in Section 3.5 were implemented in Rolez and, as a representative of an unchecked language, in Java. The comparison with Java is most suitable: because the Rolez prototype transforms Rolez programs into Java, any difference in performance is likely to be related to Rolez’s runtime checking.

5.3.1 *Experimental Setup*

As described earlier, the Rolez compiler transforms Rolez source code into Java source code. The generated Java code, as well as the code for the separate Java implementations, is compiled using a standard Java compiler and executed on a standard JVM.

We measured the performance of the aforementioned programs 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 8, 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 15 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 3.3.3, were turned off for the experiments in Sections 5.3.2–5.3.4, to enable a fairer comparison with the manually synchronized Java implementations. As explained above, these checks are independent of the main concepts in Rolez and could just as well make sense in the Java implementations. We evaluate their impact separately in Section 5.3.5.

5.3.2 Parallel Speedup

We compare the performance of all Rolez programs to that of equivalent but unchecked sequential and parallel 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.

Figure 5.8 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 execution*, to show that the performance of Rolez is mostly on par with that of Java. For this experiment, we used the largest input size for all programs. Also, note the logarithmic scale of both axes.

Five out of the eight programs (Animator, IDEA, *k*-means, Monte Carlo, and Ray Tracer) achieve substantial speedups, ranging from $9.5\times$ to $25.4\times$ for 32 tasks. The Rolez implementations of Mergesort and Quicksort achieve slightly less substantial speedups of 3.3 – $5.1\times$ 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 Section 3.5, 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.

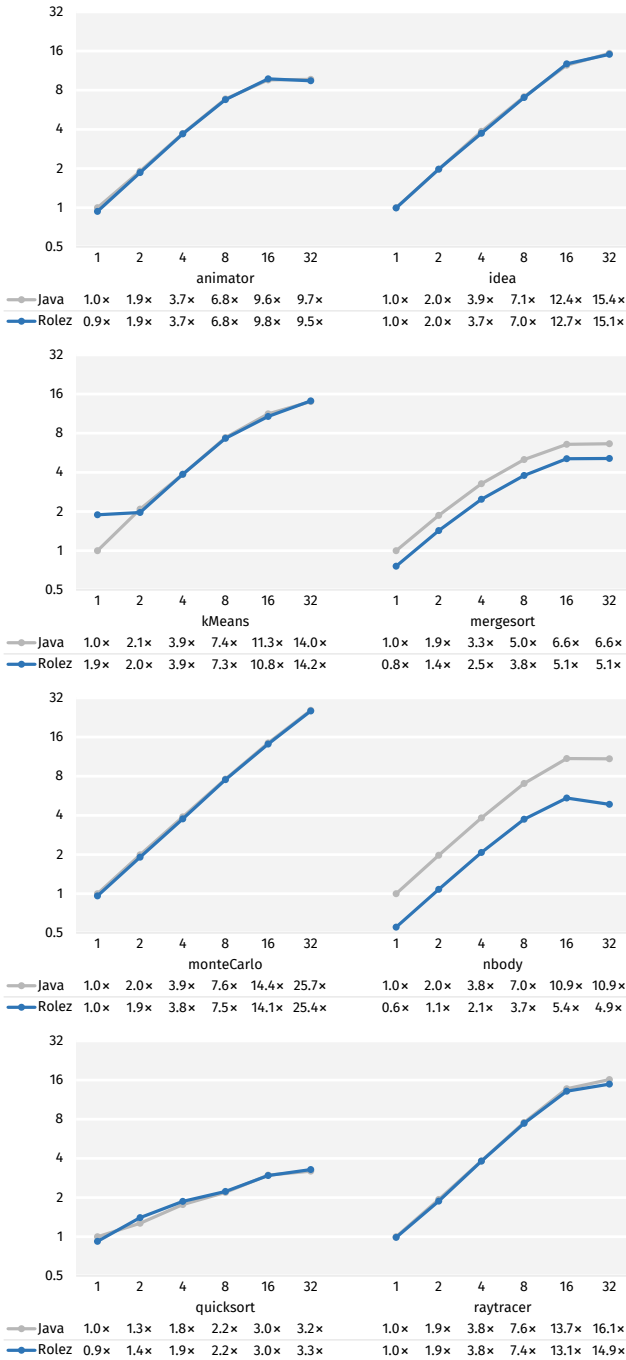


Figure 5.8: Comparison of the parallel speedups achieved by the Rolez (blue) and Java (gray) implementations, for different numbers of tasks. All speedups are relative to the *single-threaded Java execution*.

5.3.3 *Overhead*

To better understand the runtime overhead that is caused by role transitions and guarding, Figure 5.9 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%. 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 5.9 shows again the issue with the n -body implementation, which exhibits an overhead of 80% to 124%. 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.

5.3.4 *Impact of Optimizations*

To understand the impact of the presented optimizations, we also compiled and executed all programs with four different levels of optimizations.

In addition to the two optimizations discussed in Section 5.2, we implemented another optimization, called *role analysis*. This is an intra-procedural dataflow analysis that conservatively approximates the current (dynamic) role of every object in the program. If the analysis determines that the role of an object is guaranteed to be at least `READONLY` at some point in the program, then the compiler will not emit any field *read* guards for this point; similarly, if an object is guaranteed to be `READWRITE`, no field *write* guards are emitted. Role analysis is similar to concurrency analysis, in that they both reduce redundant guarding. However, on the one hand, role analysis is more fine-grained, considering a program state on the level of individual objects, while concurrency analysis considers a “thread-global” program state. On the other hand, concurrency analysis is more precise, due to its inter-procedural nature, while role analysis is strictly intra-procedural.

The four levels of optimizations are: 1) no optimizations enabled, 2) task parameters enabled, 3) task parameters and role analysis enabled, and 4) task

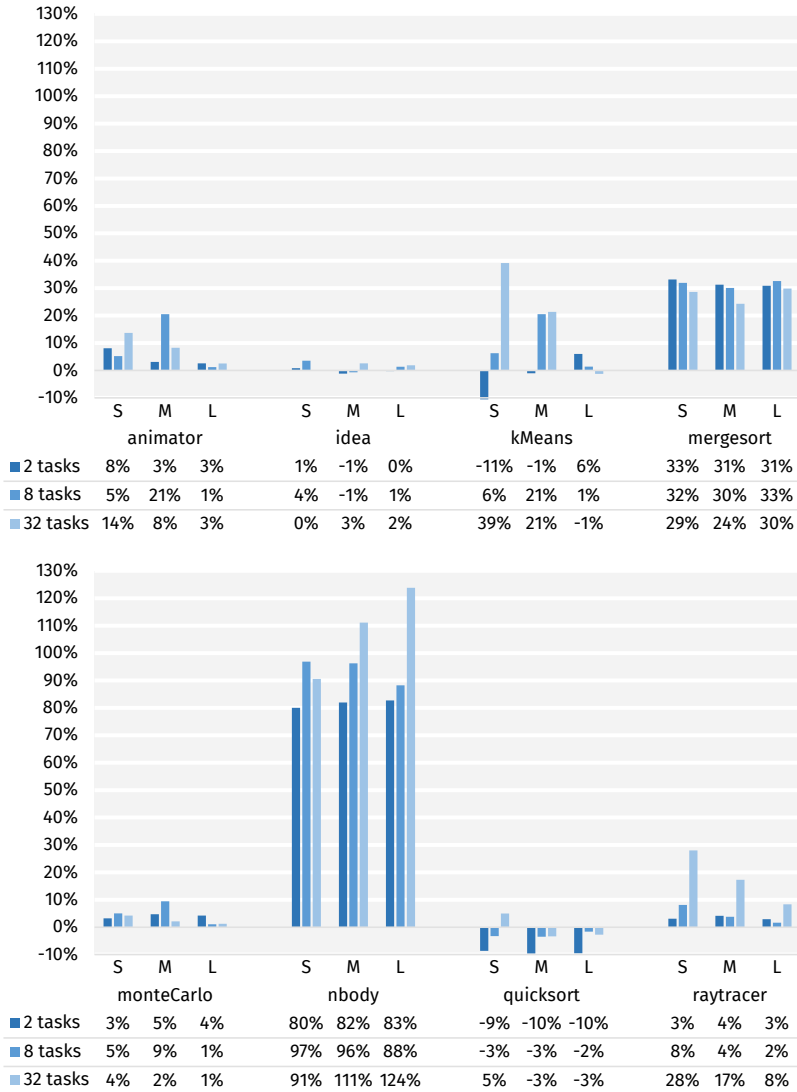


Figure 5.9: Execution time overhead of the Rolez implementations compared to the Java implementations, for the same input sizes and numbers of tasks. “S”, “M”, and “L” stand for the small, medium, and large sizes.

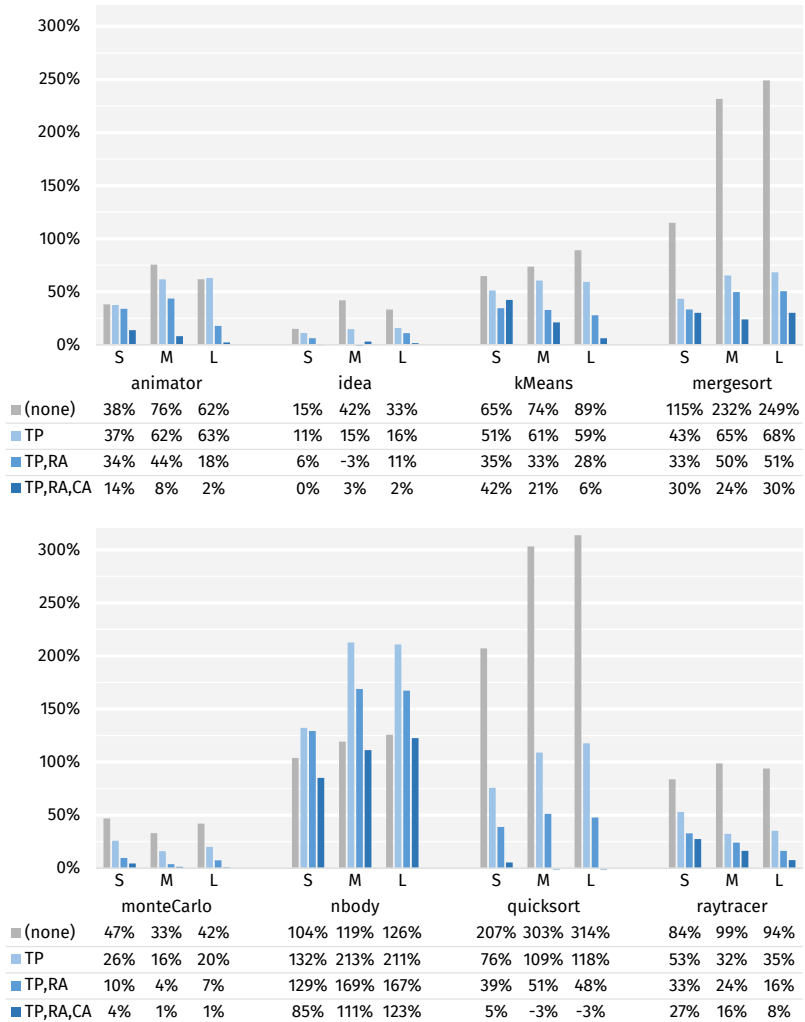


Figure 5.10: Execution time overhead of Rolez compared to Java, with 32 tasks, for three input sizes, and for three levels of optimization: 1) no optimizations, 2) with task parameters (TP), 3) with task parameters and role analysis (RA), and 4) with task parameters, role analysis, and concurrency analysis (CA).

parameters, role analysis, and concurrency analysis enabled. For the first level, every guard retrieves the current task using a `ThreadLocal` variable.*

Figure 5.10 shows the execution time overhead of the Rolez programs compared to the equivalent Java implementations, when run with 32 tasks and for the three different input sizes. For some programs, like IDEA and Monte Carlo, the overhead is moderate (mostly below 20%), for all input sizes and optimization levels. In these programs, there is relatively little access to guarded objects, so few checks are required. For others, like Mergesort and Quicksort, the overhead is more pronounced and can be as high as 300% without optimizations. These programs perform little computation per access to a guarded object (in this case the array to be sorted), so the runtime checks comprise a larger share of execution time.

When comparing the different levels of optimizations, the figure shows that the task parameter technique alone can already reduce the runtime overhead substantially. For example, for Mergesort and Quicksort, the overhead is reduced roughly by 3×.

Adding role analysis and concurrency analysis, the overhead is again reduced substantially for some programs. For example, for the Ray Tracer program, the overhead is reduced from 32–53% (with task parameters but without the other optimizations) to 8–27%. This program is structured like the example program in Figure 5.4: there is a single level of concurrent tasks that execute almost all the work in parallel. Since none of these tasks have child tasks, concurrency analysis is able to remove almost all guards. IDEA is structured similarly.

All three techniques combined reduce the overhead of Rolez over Java substantially, for almost all programs. For example, for Ray Tracer, the overhead is decreased by a factor of about 3–12× and, for Quicksort, it is reduced from over 200% to almost 0%.

5.3.5 Impact of Slice Coverage Checks

For all the results discussed above, the slice coverage checks were disabled. These runtime checks ensure that an operation on an array slice accesses only elements that are actually covered by this slice.* Even though these checks have little to do with the main concepts in Rolez (roles, transitions, guarding) and would just as well make sense in a language like Java, they are technically required for guaranteeing determinism. Thus, strictly speaking, there is an additional overhead to pay for Rolez's guarantee of determinism.

This overhead is shown in Figure 5.11. The figure contains pairs of bars, one pair for each combination of program and number of tasks. For each pair, the gray bar represents the execution time of the Rolez program *without* slice coverage checks, normalized to 100%, and the blue bar shows the execution time of the Rolez program *with* checks, relative to the gray bar. This figure only includes results for the large input sizes, but the numbers for the small and medium sizes are similar.

* When disabling these optimizations, the compiler still generates task parameters and unguarded methods, but these are not used during execution. As explained earlier, we have found that adding a method parameter has no significant impact on performance.

* Note that only array slices require runtime checks, while class slices are checked statically and incur no runtime overhead.

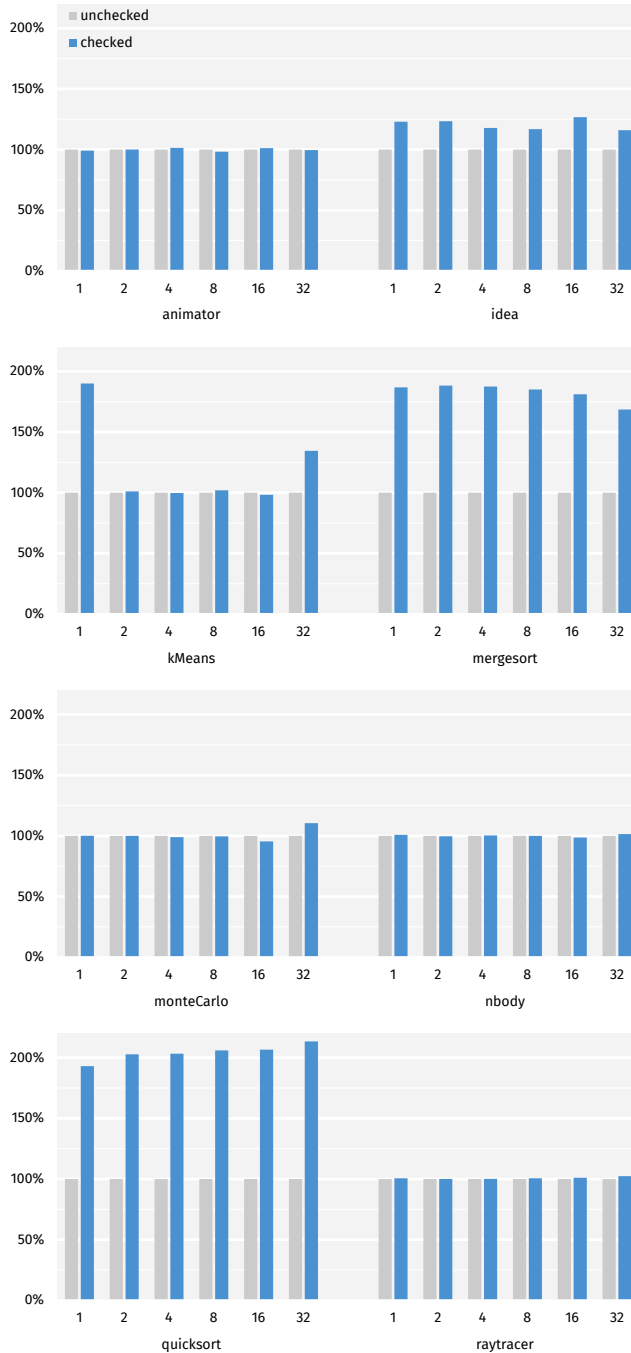


Figure 5.11: Impact of the slice coverage checks on execution time. The blue bars show the execution time when slice coverage checks are enabled, relative to the execution time of the same program and the same number of tasks when the checks are disabled (the gray bars). The large input size is shown.

First, the figure shows that the overhead of enabling slice coverage checks depends heavily on the kind of program: it is almost negligible for the Animator, Monte Carlo, n -body, and Ray Tracer programs, staying mostly below 2%, but it is substantial for the Mergesort and Quicksort programs, ranging from around 70% to over 100%. This difference is explained by the immense number of accesses to sliced arrays performed by the two sort programs. The IDEA program exhibits a modest overhead of around 25%, while the k -means program again shows irregular behavior on the machine used for the evaluation.

The results also imply that the overhead is largely independent of the number of tasks. (The only significant exception is the k -means program, where the overhead is almost negligible for 2–16 tasks, but substantially higher for 1 and 32 tasks.) This suggests that scalability is mostly unaffected by these checks.

DISCUSSION Figure 5.11 might suggest that some array-heavy programs are impossible to implement efficiently with Rolez (while maintaining the determinism guarantee), because of the high overhead inflicted by slice coverage checks. However, there are two arguments against this conclusion.

First, our Rolez implementation includes no specific optimizations for coverage checks, i.e., the code that the compiler produces includes a check for each and every slice access. A static analysis in the Rolez compiler, or even more so, a dynamic analysis in a customized JIT compiler, could conceivably reduce the number of checks performed at runtime.

More importantly, slice coverage checks can be turned on and off in different situations. A practical approach would be to enable the checks during testing and for debugging purposes, and to disable them in production, in case this has a significant impact on the overall performance of the application. This is an attractive option, especially when combined with existing testing techniques like measuring test coverage: The determinism guarantee of Rolez, combined with a sufficiently high test coverage score, would ensure that any illegal slice access is detected during testing. Then, the checks can confidently be disabled for production, because (unlike, for example, guarding) they are not required for ensuring determinism, under the assumption that all slices accesses are legal.

5.4 RELATED WORK

CODE-MANAGED RUNTIME DATA Besides Rolez, there are many other languages and systems implemented on top of VMS that rely on a VM-independent approach to do runtime checking for parallelism and concurrency. For example, there are several software transactional memory (STM) implementations that rely on this approach. Hindman and Grossman (2006) add an atomic block to Java using source-to-source translation and thus keep the implementation of the atomic block “quite separate from other concerns”. Their checks directly use the current Java thread, which is more

efficient to retrieve than a `ThreadLocal` value, so their implementation may not benefit from using dedicated method parameters.

By contrast, *DEUCE* (Korland et al., 2010) is a Java STM framework, implemented using bytecode instrumentation, which uses the same technique as described in Section 5.2.1 to pass a “transaction context” from method to method. While the reason for using method parameters is not explicitly stated, we assume that this is done for performance reasons. However, their evaluation does not discuss the impact of this optimization; to the best of our knowledge, we are the first to describe this idea as a general technique for efficient concurrency runtime checking, and to evaluate its performance impact.

Another STM-based language implemented using bytecode transformation is presented by Bättig and Gross (2017). Although not mentioned in the paper, their implementation also makes use of a transaction context that is passed using method parameters. In this chapter, we demonstrated that this technique can be used as a powerful optimization for non-transactional parallel languages as well.

CONCURRENCY ANALYSIS The analysis presented in Section 5.2.2 is an interprocedural dataflow analysis. Such analyses have been researched for a long time, but are traditionally considered whole-program analyses (Sharir and Pnueli, 1981). While there are approaches to make dataflow analysis “modular”, e.g., by Rountev (2005) and Rountev et al. (2006), these approaches consider much larger “modules”, e.g., whole libraries, whereas the presented concurrency analysis is modular on the method level, thanks to the integration with the code generator.

The idea of generating two copies of methods and switching between them has been employed in other contexts before. For example, Pizlo et al. (2008) present a compiler optimization that generates multiple versions of the program code, each optimized for a specific garbage collection (GC) phase. In contrast to the technique presented here, switching between different code versions is not based on a static analysis, but is determined at runtime, based on GC progress. The idea is also used in an STM system by Yoo et al. (2008), where two versions of a function are generated: one with memory barriers that is used inside atomic blocks and an optimized version for calls outside atomic blocks. While the technique is similar to ours, the work presented here shows that the idea can be applied more generally to perform optimization based on interprocedural analysis in a modular fashion.

Another technique that is used to perform interprocedural optimization is method inlining (Richardson and Ganapathi, 1989; Davidson and Holler, 1992), and is performed by many JIT compilers. Inlining a method can result in more optimization opportunities, as its body can be optimized in the context of the caller code. However, the VM-independent approach for implementing concurrency runtime checks poses challenges to this technique of performing interprocedural optimizations using method inlining. The

reason is that inlining is done by the JIT compiler, whereas the concurrency checking optimizations have to be applied on the source or bytecode level, *before* inlining has been performed. A trick that is used by Bättig and Gross (2017) is to compile and execute the program once without optimizations, record the inlining decisions made by the JIT compiler, and then compile the program again, using this information to inline ahead of time.

6

CONCLUSION

6.1 SUMMARY

This dissertation introduced a new parallel programming model that is deterministic-by-default, but substantially different from previous, statically-checked DPP models.

Chapter 2 introduced the Parallel Rolez model, including the core concepts of roles, role transitions, and guarding. The novelty of this model is that noninterference is not checked ahead of time, i.e., at compile time or just before a parallel section is started, but instead is enforced *during* the execution of the parallel section: in case two computations interfere, one of them is automatically being paused just before the first interfering operation is performed, and resumed as soon as the other computation has finished. This mechanism, which is realized by the combination of role transitions and guarding, guarantees that any parallel execution produces the same results as if the entire computation was executed sequentially. This property was formally described (and is proven in Appendix A).

This dynamic DPP approach provides several advantages, including the ability to parallelize two computations just as much as a given input allows. More importantly, it addresses one of the main challenges of deterministic parallelism: the handling of aliasing and object groups. To demonstrate this, Chapter 3 presented the Rolez language, which introduces two key concepts, *joint role transitions* and *slicing*, which simplify the handling of object groups when compared with static DPP languages. In these languages, the programmer needs to develop and express a static abstraction of the heap structure such that the compiler understands the aliasing patterns in the program well enough to prove noninterference. With joint role transitions and slicing, such an abstraction is not needed; instead, the runtime system uses the *actual* heap structure to enforce noninterference. Joint role transitions cause objects that are reachable from a task argument to automatically perform the same role transitions as the argument itself, serving as a sensible default behavior. Slicing can then be used to refine the set of objects that are shared alongside a task argument; for example, only part of an array may be shared with a given task, while other parts of the same array are shared with different task.

Despite its dynamic nature, Rolez provides roughly the same safety guarantees as static DPP languages. This is due to Rolez's type system, which checks for mistakes in the role declarations that the programmer provides, as presented in Chapter 4. This type system is much more lightweight than, e.g., DPJ's effect system and imposes no aliasing restrictions whatsoever. Yet, it prevents programmer mistakes that would otherwise lead to runtime errors and, in addition, it enables more efficient execution, as fewer checks need to be performed at runtime.

Chapter 5 further addressed the performance aspect by presenting an efficient prototype of the Rolez compiler and runtime system. In particular, the chapter described two optimizations that greatly reduce the runtime overhead otherwise caused by performing role transitions and guarding. The prototype is implemented on top of the Java platform and yields high performance for many Rolez programs. When compared with equivalent Java implementations without runtime checking, most of the studied Rolez programs are no more than 30% slower, which means most of them achieve substantial parallel speedups relative to a sequential execution.

In summary, the dissertation has demonstrated that deterministic parallel programming can indeed be realized without the drawbacks that many existing DPP models carry, by stepping outside the framework of pure compile-time checking and moving to a dynamic or mixed static–dynamic approach.

6.2 IMPLICATIONS

When moving from a purely static to a dynamic or mixed static–dynamic DPP approach, a simplicity–performance trade-off arises.

On the one hand, performing checks at runtime implies that there is a cost in terms of performance. Indeed, our evaluation shows that there are programs for which the overhead caused by these checks is significant, and may be intolerable for some applications.

On the other hand, we argue that the Parallel Roles model, or the dynamic DPP approach in general, provides a much simpler and especially beginner-friendly way to do parallel programming when compared with statically-checked DPP languages. Dynamic DPP frees the programmer from the burden of expressing aliasing and grouping patterns explicitly, and instead allows a programmer to take a sequential program and convert it to a deterministic parallel program simply by adding role declarations and constructs for parallel execution (e.g., by converting a method to a task). Of course, such a simple transformation may not result in any substantial performance improvement; often, more transformations may be required, for example, partitioning of data or restructuring of the computation. But the model allows the programmer to iteratively transition a program from a completely sequential implementation to an increasingly parallel version, without worrying about determinism and without understanding any complex new language constructs.

Note that his simplicity–performance trade-off may not necessarily concern raw performance in terms of total execution time. Considering that the Rolez implementation we evaluated is a merely research prototype, and that it employs only static optimizations (as opposed to JIT optimizations), future efforts may reduce the execution time overhead to insignificant levels for many applications. Then, other performance aspects may become more critical: the *variation* and the *predictability* of execution time. Because the precise functioning and performance impact of the runtime checks may depend on implementation details, the environment, and the scheduling of tasks, they are beyond the direct control of the programmer. Therefore, performance variation and predictability may be more severe than those of static DPP approaches, which would make the dynamic approach unsuitable for some applications (e.g., in embedded, real-time, or high-performance settings).

When comparing Rolez’s dynamic DPP approach with nondeterministic mainstream languages, a different trade-off emerges. The simplicity of Rolez is comparable to that of mainstream languages like Java; besides role declarations, there are only few language constructs and concepts that differentiate the two. However, on the one hand, a deterministic language like Rolez provides substantially more safety: if a Rolez program compiles, the programmer can be sure that it contains no concurrency bugs and that its execution will be deterministic. On the other hand, there is again the performance overhead of the runtime checks that the dynamic DPP approach involves. Thus, we have a safety–performance trade-off.

This safety–performance trade-off is known from other, tried-and-tested techniques that involve some form of runtime checking, including automatic memory management like garbage collection or array bounds checking. Again, the trade-off may concern performance variation and predictability rather than total execution time, as is often the case for garbage collection. While the cost of such techniques may be intolerable for some scenarios, many organizations or individuals are willing to pay this cost in exchange for safety, which in turn implies maintainability and increased programmer productivity. As a result, languages that perform some of these runtime checking techniques, like Java, C#, Python, Javascript, etc., have largely superseded unchecked languages like C and C++ for many applications, and they have become the first choice for the majority of today’s software projects. Given that the dynamic approach presented here brings the simplicity of DPP languages much closer to mainstream languages, the results in this dissertation suggest that a similar shift in the field of parallel programming languages may be possible.

APPENDIX



DETERMINISM PROOF

A.1 OUTLINE

To formalize the determinism property, we use the concept of *noninterference*, which states that concurrently executing tasks may not access the same object, unless both accesses are only read operations. In Parallel Roles, noninterference more specifically means that whenever a parent task starts a child task, all read or write operations in the child *happen before* any subsequent interfering operation in the parent.

Noninterference 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.

The proof is organized as follows. After defining legal program states (Section A.2), we first prove *soundness of role declarations*, which roughly states that the (current) role of an object in a task is never more permissive than its *declared* role (as defined by the R^{RW} and R^{RO} sets) in that task and all ancestor tasks (Section A.3). Then, we show that the role transitions at the start of a new child task guarantee that the roles of the objects that are shared between the child and the parent always “give priority to the child”. More precisely, as long as the roles of an object permit the child task to read from (or write to) the object, they do not permit the parent to write to (or read from) that object (Section A.4). This *child task priority* property is finally used to show noninterference (Section A.5).

A.2 LEGAL PROGRAM STATES

For the whole determinism proof, we only consider *legal program states*. These are all states that a program in this model can possibly be in, except the special S_{error} state, which corresponds to errors that result from programmer mistakes. (Chapter 4 discusses how these errors are handled in Rolez.) We define legal program states using the transitive state transition

relation, which is in turn based on the state transition relation in Figure 2.5 and 2.6.

Definition A.2.1 (\longrightarrow^*). The relation \longrightarrow^* describes all program states that can be reached from a given one. It is the transitive closure of the state transition relation \longrightarrow :

$$S \longrightarrow^* S'' \triangleq S = S'' \vee \exists S' : S \longrightarrow S' \wedge S' \longrightarrow S''.$$

We write $\longrightarrow^*(S)$ to denote the set of all states that are reachable from S .

Definition A.2.2 (*Legal Program States*). A program state S is legal if and only if it can be reached from the initial state S_0 and is not the error state:

$$\text{legal}(S) \triangleq S_0 \longrightarrow^* S \wedge S \neq S_{\text{error}}.$$

We write “ $\forall \text{legal}(S) : \dots$ ” as a shorthand for “ $\forall S : \text{legal}(S) \rightarrow \dots$ ” in the rest of this chapter. Note that the short arrow \rightarrow stands for the logical implication and is not to be confused with the state transition relation \longrightarrow . Further, we write $\xrightarrow{\text{legal}}^*(S)$ for $\longrightarrow^*(S) \cap \{S_i \mid \text{legal}(S_i)\}$.

A.3 SOUNDNESS OF ROLE DECLARATIONS

We begin the proof by showing *soundness of role declarations*. For this, we need a few more helper functions, which should be self-explanatory:

$$\begin{aligned} \text{refs}(H) &\triangleq \{r \mid (r \mapsto _) \in H\}, \\ \text{ids}(Ts) &\triangleq \{t \mid \langle t, _, _, _, _ \rangle \in Ts\}, \\ \text{finished}_{Ts}(t) &\triangleq \text{task}_{Ts}(t) = \langle _, _, _, \bullet \rangle. \end{aligned}$$

Then, we also need to prove a preliminary lemma:

LEMMA A.3.1. *Whenever an object is declared as READONLY in a task t (which has not finished), then this object plays the READONLY role in at least one other task, which is an ancestor of t :*

$$\begin{aligned} \forall \text{legal}(\langle H, Ts \rangle) : \forall \langle t, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in Ts : \text{finished}_{Ts}(t) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_i \in \text{ancests}_{Ts}(t) : \text{mayRead}_H(t_i, r). \end{aligned}$$

PROOF. Shown using induction over all legal program states $S = \langle H, Ts \rangle$. In the base case, where $S = S_0$, the set of references $\text{refs}(H_0)$ is empty, so the property trivially holds. For the inductive step, there is a predecessor state $S_p = \langle H_p, Ts_p \rangle$, such that $S_p \longrightarrow S$. We assume the property holds for S_p :

$$\begin{aligned} \forall \langle t, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in Ts_p : \text{finished}_{Ts_p}(t) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_i \in \text{ancests}_{Ts_p}(t) : \text{mayRead}_{H_p}(t_i, r). \end{aligned} \quad (1)$$

Then, we show that the property also holds for S , for each of the five cases for the state transition $S_p \longrightarrow S$, which correspond to the five operations in

Figures 2.5 and 2.6. Note that some operations have multiple possible outcomes. However, we only really need to consider the “successful” outcomes. In the other outcomes, either $S = S_p$, where the property holds trivially for S , or $S = S_{\text{error}}$, where $\neg\text{legal}(S)$.

$S_p \xrightarrow{t \text{ create } r} S$: According to the definition of the create operation in Figure 2.5, the new heap H equals $H_p \cup \{r \mapsto \{t :: \text{rw}\}\}$. This means that the roles of all existing objects remain unchanged. Thus, from (i) follows (note the change from H_p to H):

$$\begin{aligned} \forall \langle t, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_{S_p} : \text{finished}_{T_{S_p}}(t) \vee \\ \forall r_i \in R^{\text{RO}} - R^{\text{RW}} : \exists t_i \in \text{ancests}_{T_{S_p}}(t) : \text{mayRead}_H(t_i, r_i). \end{aligned}$$

Further, all child–parent relationships remain the same, and no task starts or finishes during the state transition. In addition, all the R^{RO} sets are unchanged, while the R^{RW} sets are unchanged or even grow. Therefore, it follows (note the change from T_{S_p} to T_s):

$$\begin{aligned} \forall \langle t, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_s : \text{finished}_{T_s}(t) \vee \\ \forall r_i \in R^{\text{RO}} - R^{\text{RW}} : \exists t_i \in \text{ancests}_{T_s}(t) : \text{mayRead}_H(t_i, r_i). \end{aligned}$$

$S_p \xrightarrow{-\text{read}} S$: There are no changes in any roles or in any R^{RW} or R^{RO} set. Therefore, since the property holds for S_p , it holds for S as well.

$S_p \xrightarrow{-\text{write}} S$: Again, no changes in any roles or in any R^{RW} or R^{RO} set.

$S_p \xrightarrow{t \text{ start } t_{\text{ch}}(R_{\text{ch}}^{\text{RW}}, R_{\text{ch}}^{\text{RO}})} S$: All tasks that are not t or t_{ch} are not affected by any role transition or any change in R^{RW} or R^{RO} sets, so from (i) follows:

$$\begin{aligned} \forall \langle t_i, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_s - \{\text{task}_{T_s}(t_i) \mid t_i = t \vee t_i = t_{\text{ch}}\} : \\ \text{finished}_{T_s}(t_i) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_j \in \text{ancests}_{T_s}(t_i) : \text{mayRead}_H(t_j, r). \end{aligned}$$

The R^{RW} and R^{RO} sets of t are also unaffected by the state transition. In addition, if t has any ancestor task t_j , then the roles in t_j are unaffected as well. Therefore, it further follows:

$$\begin{aligned} \forall \langle t_i, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_s - \text{task}_{T_s}(t_{\text{ch}}) : \text{finished}_{T_s}(t_i) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_j \in \text{ancests}_{T_s}(t_i) : \text{mayRead}_H(t_j, r). \end{aligned}$$

Finally, the role transitions in Figure 2.6 define that all objects in $R_{\text{ch}}^{\text{RO}} - R_{\text{ch}}^{\text{RW}}$ play the READONLY role in t after the transition. Because t is an ancestor of t_{ch} , the term $\forall r \in R_{\text{ch}}^{\text{RO}} - R_{\text{ch}}^{\text{RW}} : \exists t_j \in \text{ancests}_{T_s}(t_{\text{ch}}) : \text{mayRead}_H(t_j, r)$. is true. It finally follows:

$$\begin{aligned} \forall \langle t_i, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_s : \text{finished}_{T_s}(t_i) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_j \in \text{ancests}_{T_s}(t_i) : \text{mayRead}_H(t_j, r). \end{aligned}$$

$S_p \xrightarrow{t \text{ finish}} S$: All tasks that are not t or the parent of t are not affected by any role transition or any change in R^{RW} or R^{RO} sets, so from (1) follows:

$$\begin{aligned} \forall \langle t_i, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_S - \{\text{task}_{T_S}(t_i) \mid t_i = t \vee t_i = \text{parent}_{T_S}(t)\} : \\ \text{finished}_{T_S}(t_i) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_j \in \text{ancests}_{T_S}(t_i) : \text{mayRead}_H(t_j, r). \end{aligned}$$

Since t finishes during the state transition, the property trivially holds for t . Thus:

$$\begin{aligned} \forall \langle t_i, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_S - \{\text{task}_{T_S}(\text{parent}_{T_S}(t))\} : \text{finished}_{T_S}(t_i) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_j \in \text{ancests}_{T_S}(t_i) : \text{mayRead}_H(t_j, r). \end{aligned}$$

Finally, the R^{RW} and R^{RO} sets of t 's parent are not affected by the transition, and neither are the roles for any of t 's parent's ancestors. It follows:

$$\begin{aligned} \forall \langle t_i, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_S : \text{finished}_{T_S}(t_i) \vee \\ \forall r \in R^{\text{RO}} - R^{\text{RW}} : \exists t_j \in \text{ancests}_{T_S}(t_i) : \text{mayRead}_H(t_j, r). \end{aligned}$$

□

Now, we can prove *soundness of role declarations*, which roughly states that the (current) role of an object in a task is never more permissive than its *declared* role in that task and all ancestor tasks.

THEOREM A.3.2 (SOUNDNESS OF ROLE DECLARATIONS). *An object can only be read in a task t if it is in t 's R^{RW} or R^{RO} set, and it can only be written in t if it is in t 's R^{RW} set. In addition, the R^{RW} set of a task t is always a subset of t 's parent's R^{RW} set, and the R^{RO} set of t is always a subset of the union of t 's parent's R^{RW} and R^{RO} sets:*

$$\begin{aligned} \forall \text{legal}(\langle H, T_S \rangle) : \forall \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_S : \text{finished}_{T_S}(t) \vee \\ \left(\forall r \in \text{refs}(H) : (\text{mayRead}_H(t, r) \rightarrow r \in R^{\text{RW}} \cup R^{\text{RO}}) \wedge \right. \\ \left. (\text{mayWrite}_H(t, r) \rightarrow r \in R^{\text{RW}}) \right) \wedge \\ (\tau = \bullet \vee R^{\text{RW}} \subseteq R_\tau^{\text{RW}} \wedge R^{\text{RO}} \subseteq R_\tau^{\text{RW}} \cup R_\tau^{\text{RO}}), \end{aligned}$$

where

$$\text{task}_{T_S}(\tau) = \langle \tau, _, R_\tau^{\text{RW}}, R_\tau^{\text{RO}}, _ \rangle.$$

In the following, we call the part on the second and third line the “first part” and the rest the “second part”.

PROOF. Again shown using induction over all legal program states $S = \langle H, T_S \rangle$. In the base case, where $S = S_0$, the set of references $\text{refs}(H_0)$ is empty, so the first part of the property trivially holds. Since the only task in the program has no parent, i.e., $\tau = \bullet$, the second part also holds.

For the inductive step, there is a predecessor state $S_p = \langle H_p, Ts_p \rangle$, such that $S_p \longrightarrow S$. We assume the property holds for S_p :

$$\begin{aligned} & \forall \langle t, \tau, R^{RW}, R^{RO}, _ \rangle \in Ts_p : \text{finished}_{Ts_p}(t) \vee \\ & \left(\forall r \in \text{refs}(H_p) : (\text{mayRead}_{H_p}(t, r) \rightarrow r \in R^{RW} \cup R^{RO}) \wedge \right. \\ & \quad \left. (\text{mayWrite}_{H_p}(t, r) \rightarrow r \in R^{RW}) \right) \wedge \\ & (\tau = \bullet \vee R^{RW} \subseteq R_\tau^{RW} \wedge R^{RO} \subseteq R_\tau^{RW} \cup R_\tau^{RO}). \end{aligned} \quad (1)$$

Note that we leave away the “where” clause, always just writing R_τ^{RW} and R_τ^{RO} for τ 's R^{RW} and R^{RO} sets.

Then, we show that the property also holds for S , for each of the five cases for the state transition $S_p \longrightarrow S$, which correspond to the five operations in Figures 2.5 and 2.6. Like in the previous proof, some operations have multiple possible outcomes, of which we only consider the “successful” ones.

$S_p \xrightarrow{t \text{ create } r} S$: First, the state transition leaves the roles of all existing objects, i.e., all objects referred to by $\text{refs}(H_p) = \text{refs}(H) - \{r\}$, unchanged. Also, the R^{RW} and R^{RO} of all tasks remain unchanged with respect to the existing objects. Therefore, from (1) follows:

$$\begin{aligned} & \forall \langle t, _, R^{RW}, R^{RO}, _ \rangle \in Ts : \text{finished}_{Ts}(t) \vee \\ & \forall r_i \in \text{refs}(H) - \{r\} : (\text{mayRead}_H(t, r_i) \rightarrow r_i \in R^{RW} \cup R^{RO}) \wedge \\ & (\text{mayWrite}_H(t, r_i) \rightarrow r_i \in R^{RW}). \end{aligned} \quad (2)$$

The newly created object with reference r plays the READWRITE role in t and the PURE role in all other tasks. Since r is added to t 's R^{RW} set, the following also holds:

$$\begin{aligned} & \forall \langle t, _, R^{RW}, R^{RO}, _ \rangle \in Ts : \text{finished}_{Ts}(t) \vee \\ & (\text{mayRead}_H(t, r) \rightarrow r \in R^{RW} \cup R^{RO}) \wedge (\text{mayWrite}_H(t, r) \rightarrow r \in R^{RW}). \end{aligned}$$

Together with (2), we have:

$$\begin{aligned} & \forall \langle t, _, R^{RW}, R^{RO}, _ \rangle \in Ts : \text{finished}_{Ts}(t) \vee \\ & \forall r_i \in \text{refs}(H) : (\text{mayRead}_H(t, r_i) \rightarrow r_i \in R^{RW} \cup R^{RO}) \wedge \\ & (\text{mayWrite}_H(t, r_i) \rightarrow r_i \in R^{RW}). \end{aligned} \quad (3)$$

The new reference r is not only added to t 's R^{RW} set, but to all of t ancestor's R^{RW} sets as well. Thus, the second part of the property also holds after the transition:

$$\begin{aligned} & \forall \langle t, \tau, R^{RW}, R^{RO}, _ \rangle \in Ts : \text{finished}_{Ts}(t) \vee \\ & (\tau = \bullet \vee R^{RW} \subseteq R_\tau^{RW} \wedge R^{RO} \subseteq R_\tau^{RW} \cup R_\tau^{RO}). \end{aligned}$$

Combined with (3), we finally have:

$$\begin{aligned} \forall \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in Ts : \text{finished}_{Ts}(t) \vee \\ (\forall r \in \text{refs}(H) : (\text{mayRead}_H(t, r) \rightarrow r \in R^{\text{RW}} \cup R^{\text{RO}}) \wedge \\ (\text{mayWrite}_H(t, r) \rightarrow r \in R^{\text{RW}})) \wedge \\ (\tau = \bullet \vee R^{\text{RW}} \subseteq R^{\text{RW}}_t \wedge R^{\text{RO}} \subseteq R^{\text{RW}}_t \cup R^{\text{RO}}_t). \end{aligned}$$

$S_p \xrightarrow{\text{-read}} S$: There are no changes in any roles or in any R^{RW} or R^{RO} set. Therefore, since the property holds for S_p , it holds for S as well.

$S_p \xrightarrow{\text{-write}} S$: Again, no changes in any roles or in any R^{RW} or R^{RO} set.

$S_p \xrightarrow{t \text{ start } t_{\text{ch}}(R^{\text{RW}}_{\text{ch}}, R^{\text{RO}}_{\text{ch}})} S$: During a start transition, all object's roles and all R^{RW} and R^{RO} sets for all tasks $t_i \notin \{t, t_{\text{ch}}\}$ remain the same. Thus, from (1) follows:

$$\begin{aligned} \forall \langle t_i, _, R_i^{\text{RW}}, R_i^{\text{RO}}, _ \rangle \in Ts - \{\text{task}_{Ts}(t), \text{task}_{Ts}(t_{\text{ch}})\} : \text{finished}_{Ts}(t_i) \vee \\ \forall r \in \text{refs}(H) : (\text{mayRead}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}} \cup R_i^{\text{RO}}) \wedge \\ (\text{mayWrite}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}}). \end{aligned} \quad (4)$$

In the original task, all object play either the same or a less permissive role after the transition than before, so from (1) also follows:

$$\begin{aligned} \forall r \in \text{refs}(H) : (\text{mayRead}_H(t, r) \rightarrow r \in R^{\text{RW}} \cup R^{\text{RO}}) \wedge \\ (\text{mayWrite}_H(t, r) \rightarrow r \in R^{\text{RW}}), \end{aligned} \quad (5)$$

where $\text{task}_{Ts}(t) = \langle t, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle$. In t_{ch} , all objects that play the READ-WRITE role are in $R^{\text{RW}}_{\text{ch}}$ and all objects that play the READONLY role are in $R^{\text{RO}}_{\text{ch}}$. All other objects implicitly play the PURE role, because there cannot be any role mapping ($t_{\text{ch}} :: _$) already present in an object, because “ t_{ch} ” is a fresh task ID that has never been used before. Therefore, it follows, again from (1):

$$\begin{aligned} \forall r \in \text{refs}(H) : (\text{mayRead}_H(t_{\text{ch}}, r) \rightarrow r \in R^{\text{RW}}_{\text{ch}} \cup R^{\text{RO}}_{\text{ch}}) \wedge \\ (\text{mayWrite}_H(t_{\text{ch}}, r) \rightarrow r \in R^{\text{RW}}_{\text{ch}}). \end{aligned} \quad (6)$$

(4), (5), and (6) together imply:

$$\begin{aligned} \forall \langle t_i, _, R_i^{\text{RW}}, R_i^{\text{RO}}, _ \rangle \in Ts : \text{finished}_{Ts}(t_i) \vee \\ \forall r \in \text{refs}(H) : (\text{mayRead}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}} \cup R_i^{\text{RO}}) \wedge \\ (\text{mayWrite}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}}). \end{aligned} \quad (7)$$

According to the the definition of the start transition, only those objects that t may read (in the program state S_p) may be in the $R^{\text{RO}}_{\text{ch}}$ set and only

those that t may write may be in the $R_{\text{ch}}^{\text{RW}}$ set. Otherwise, the start operation would not be successful. Combined with the first part of (1), it follows:

$$R_{\text{ch}}^{\text{RW}} \subseteq R^{\text{RW}} \wedge R_{\text{ch}}^{\text{RO}} \subseteq R^{\text{RW}} \cup R^{\text{RO}}, \quad (8)$$

where $\text{task}_{T_S}(t) = \langle t, _, R^{\text{RW}}, R^{\text{RO}}, _ \rangle$. Because all existing task's R^{RW} and R^{RO} sets are unchanged, it further follows from (1):

$$\begin{aligned} & \forall \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_S - \{\text{task}_{T_S}(t_{\text{ch}})\} : \text{finished}_{T_S}(t) \vee \\ & (\tau = \bullet \vee R^{\text{RW}} \subseteq R_{\tau}^{\text{RW}} \wedge R^{\text{RO}} \subseteq R_{\tau}^{\text{RW}} \cup R_{\tau}^{\text{RO}}). \end{aligned}$$

And together with (8):

$$\begin{aligned} & \forall \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_S : \text{finished}_{T_S}(t) \vee \\ & (\tau = \bullet \vee R^{\text{RW}} \subseteq R_{\tau}^{\text{RW}} \wedge R^{\text{RO}} \subseteq R_{\tau}^{\text{RW}} \cup R_{\tau}^{\text{RO}}). \end{aligned}$$

Finally, together with (7):

$$\begin{aligned} & \forall \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_S : \text{finished}_{T_S}(t) \vee \\ & \left(\forall r \in \text{refs}(H) : (\text{mayRead}_H(t, r) \rightarrow r \in R^{\text{RW}} \cup R^{\text{RO}}) \wedge \right. \\ & \quad \left. (\text{mayWrite}_H(t, r) \rightarrow r \in R^{\text{RW}}) \right) \wedge \\ & (\tau = \bullet \vee R^{\text{RW}} \subseteq R_{\tau}^{\text{RW}} \wedge R^{\text{RO}} \subseteq R_{\tau}^{\text{RW}} \cup R_{\tau}^{\text{RO}}). \end{aligned}$$

$S_p \xrightarrow{t \text{ finish}} S$: When task t finishes, all object's roles and all R^{RW} and R^{RO} sets for all tasks $t_i \notin \{t, \tau\}$, where τ is the parent task of t , remain the same. It follows from (1):

$$\begin{aligned} & \forall \langle t_i, _, R_i^{\text{RW}}, R_i^{\text{RO}}, _ \rangle \in T_S - \{\text{task}_{T_S}(t), \text{task}_{T_S}(\tau)\} : \text{finished}_{T_S}(t_i) \vee \\ & \forall r \in \text{refs}(H) : (\text{mayRead}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}} \cup R_i^{\text{RO}}) \wedge \\ & (\text{mayWrite}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}}), \end{aligned}$$

and since t is finished after the transition:

$$\begin{aligned} & \forall \langle t_i, _, R_i^{\text{RW}}, R_i^{\text{RO}}, _ \rangle \in T_S - \{\text{task}_{T_S}(\tau)\} : \text{finished}_{T_S}(t_i) \vee \\ & \forall r \in \text{refs}(H) : (\text{mayRead}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}} \cup R_i^{\text{RO}}) \wedge \\ & (\text{mayWrite}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}}). \end{aligned} \quad (9)$$

In the parent task τ , all objects in t 's R^{RW} set become READWRITE. However, because of the second part of (1), we know that $R^{\text{RW}} \subseteq R_{\tau}^{\text{RW}}$ (where $\text{task}_{T_S}(t) = \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, \bullet \rangle$). Hence, all objects in R^{RW} are also in R_{τ}^{RW} :

$$\begin{aligned} & \forall r \in R^{\text{RW}} : (\text{mayRead}_H(\tau, r) \rightarrow r \in R_{\tau}^{\text{RW}} \cup R_{\tau}^{\text{RO}}) \wedge \\ & (\text{mayWrite}_H(\tau, r) \rightarrow r \in R_{\tau}^{\text{RW}}). \end{aligned} \quad (10)$$

From the second part of (1) also follows $R^{\text{RO}} \subseteq R_{\tau}^{\text{RW}} \cup R_{\tau}^{\text{RO}}$, so every object in R^{RO} must be in R_{τ}^{RW} or R_{τ}^{RO} . According to the finish transition definition, these

objects either play the READWRITE or the READONLY role in τ afterwards. For those not in $R_\tau^{\text{RO}} - R_\tau^{\text{RW}}$, i.e., those in R_τ^{RW} , the following trivially holds:

$$\begin{aligned} \forall r \in R^{\text{RO}} - (R_\tau^{\text{RO}} - R_\tau^{\text{RW}}) : & \quad (11) \\ (\text{mayRead}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}} \cup R_\tau^{\text{RO}}) \wedge (\text{mayWrite}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}}). \end{aligned}$$

On the other hand, objects that are in R^{RO} and also in $R_\tau^{\text{RO}} - R_\tau^{\text{RW}}$ can only play the READONLY role in τ afterwards. This follows from Lemma A.3.1: For all references r in $R_\tau^{\text{RO}} - R_\tau^{\text{RW}}$, there must be an ancestor of τ that may read from r . (Task τ cannot have finished before t has finished, which follows from the precondition of the successful case of the finish transition, but applied to τ instead of t .) Since there does exist another task $t_i \notin \{t, \tau\}$ that may read from r , the role of the object that r refers to becomes READONLY in τ . Therefore, the following holds as well:

$$\begin{aligned} \forall r \in R^{\text{RO}} \cap (R_\tau^{\text{RO}} - R_\tau^{\text{RW}}) : & \quad (12) \\ (\text{mayRead}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}} \cup R_\tau^{\text{RO}}) \wedge (\text{mayWrite}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}}). \end{aligned}$$

Finally, for all r not in R^{RW} or R^{RO} , the roles in τ remain the same. Hence, (1) implies:

$$\begin{aligned} \forall r \in \text{refs}(H) - (R^{\text{RW}} \cup R^{\text{RO}}) : & \quad (13) \\ (\text{mayRead}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}} \cup R_\tau^{\text{RO}}) \wedge (\text{mayWrite}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}}). \end{aligned}$$

Combining (10)–(13), we can infer:

$$\begin{aligned} \forall r \in \text{refs}(H) : & (\text{mayRead}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}} \cup R_\tau^{\text{RO}}) \wedge \\ & (\text{mayWrite}_H(\tau, r) \rightarrow r \in R_\tau^{\text{RW}}), \end{aligned}$$

and, together with (9):

$$\begin{aligned} \forall \langle t_i, _, R_i^{\text{RW}}, R_i^{\text{RO}}, _ \rangle \in T_s : & \text{finished}_{T_s}(t_i) \vee \\ \forall r \in \text{refs}(H) : & (\text{mayRead}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}} \cup R_i^{\text{RO}}) \wedge \\ & (\text{mayWrite}_H(t_i, r) \rightarrow r \in R_i^{\text{RW}}). \end{aligned}$$

The second part of the property holds trivially, because the R^{RW} and R^{RO} sets of all tasks stay the same. Therefore, we finally arrive at:

$$\begin{aligned} \forall \langle t, \tau, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in T_s : & \text{finished}_{T_s}(t) \vee \\ & \left(\forall r \in \text{refs}(H) : (\text{mayRead}_H(t, r) \rightarrow r \in R^{\text{RW}} \cup R^{\text{RO}}) \wedge \right. \\ & \left. (\text{mayWrite}_H(t, r) \rightarrow r \in R^{\text{RW}}) \right) \wedge \\ & (\tau = \bullet \vee R^{\text{RW}} \subseteq R_\tau^{\text{RW}} \wedge R^{\text{RO}} \subseteq R_\tau^{\text{RW}} \cup R_\tau^{\text{RO}}). \end{aligned}$$

□

A.4 CHILD TASK PRIORITY

The final stepping stone for the determinism proof is *child task priority*, which formalizes the following property: From the moment a task τ starts a child task t , τ may not write to any object O as long as t may read still from or write to O , and τ may not read from any object O as long as t may still write to O . We define this property formally using the inverse:

THEOREM A.4.1 (CHILD TASK PRIORITY). *For two tasks, t and t 's parent τ , and for all objects O on the current heap (and accordingly, for all corresponding references r), 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. Formally:*

$$\begin{aligned} & \forall \text{legal}(\langle H, Ts \rangle) : \forall t \in \text{ids}(Ts) : \forall r \in \text{refs}(H) : \\ & \quad (\text{mayWrite}_H(\tau, r) \rightarrow \text{mayNeverRead}_{\langle H, Ts \rangle}(t, r)) \wedge \\ & \quad (\text{mayRead}_H(\tau, r) \rightarrow \text{mayNeverWrite}_{\langle H, Ts \rangle}(t, r)), \\ & \text{where} \\ & \quad \text{mayNeverRead}_S(t, r) \triangleq \forall \langle H', Ts' \rangle \in \xrightarrow{\text{legal}}*(S) : \\ & \quad \quad \text{finished}_{Ts'}(t) \vee \neg \text{mayRead}_{H'}(t, r), \\ & \quad \text{mayNeverWrite}_S(t, r) \triangleq \forall \langle H', Ts' \rangle \in \xrightarrow{\text{legal}}*(S) : \\ & \quad \quad \text{finished}_{Ts'}(t) \vee \neg \text{mayWrite}_{H'}(t, r), \\ & \quad \tau \triangleq \text{parent}_{Ts}(t). \end{aligned}$$

This theorem really expresses the same property as described above: For example, if τ may not read from an object as long as t may write to it, then that means that once τ may indeed read from it, then t may not write to it anymore. Note that $\neg \text{mayRead}_{H'}(t, r)$ implies $\neg \text{mayWrite}_{H'}(t, r)$, like $\text{mayWrite}_H(t, r)$ implies $\text{mayRead}_H(t, r)$. Also note that from now on, we simply write τ_i for $\text{parent}_{Ts}(t_i)$, when Ts is clear from the context.

PROOF. We prove child task priority using induction over legal program states $S = \langle H, Ts \rangle$. The base case, where $\langle H, Ts \rangle = S_0$, is trivial, since H is empty. For the inductive step, if $\langle H, Ts \rangle$ is legal (and $\neq S_0$), then there exists a legal $S_p = \langle H_p, Ts_p \rangle$, such that $S_p \longrightarrow S$. We assume that child task priority holds for S_p :

$$\begin{aligned} & \forall t_i \in \text{ids}(Ts_p) : \forall r_i \in \text{refs}(H_p) : \\ & \quad (\text{mayWrite}_{H_p}(\tau_i, r_i) \rightarrow \text{mayNeverRead}_{S_p}(t_i, r_i)) \wedge \\ & \quad (\text{mayRead}_{H_p}(\tau_i, r_i) \rightarrow \text{mayNeverWrite}_{S_p}(t_i, r_i)). \end{aligned} \tag{1}$$

Then, there are five cases for the transition $S_p \longrightarrow S$, corresponding to the five possible operations (for which we again assume that they are successful):

OBJECT CREATION: $S_p \xrightarrow{t \text{ create } r} S$ Following the definition of $\xrightarrow{t \text{ create } r}$, $\text{ids}(Ts) = \text{ids}(Ts_p)$. Also, because the state transition does not change the roles of any previously existing object or causes a task to finish, (1) implies:

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) : \forall r_i \in \text{refs}(H_p) : \\ (\text{mayWrite}_H(\tau_i, r_i) \rightarrow \text{mayNeverRead}_{S_p}(t_i, r_i)) \wedge \\ (\text{mayRead}_H(\tau_i, r_i) \rightarrow \text{mayNeverWrite}_{S_p}(t_i, r_i)). \end{aligned} \quad (2)$$

Further, because $\xrightarrow{\text{legal}}_*(S) \subset \xrightarrow{\text{legal}}_*(S_p)$, everything that holds for states in $\xrightarrow{\text{legal}}_*(S_p)$ also holds for states in $\xrightarrow{\text{legal}}_*(S)$. Thus, since $\text{mayNeverRead}_{S_p}(t_i, r_i)$ is defined as $\forall \langle H'_p, T'_p \rangle \in \xrightarrow{\text{legal}}_*(S_p) : \text{finished}_{T'_p}(t_i) \vee \neg \text{mayRead}_{H'_p}(t_i, r_i)$, we can see that $\text{mayNeverRead}_{S_p}(t_i, r_i)$ implies $\text{mayNeverRead}_S(t_i, r_i)$, for all t_i and r_i . Similarly, $\text{mayNeverWrite}_{S_p}(t_i, r_i)$ implies $\text{mayNeverWrite}_S(t_i, r_i)$. Together with (2), this implies:

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) : \forall r_i \in \text{refs}(H_p) : \\ (\text{mayWrite}_H(\tau_i, r_i) \rightarrow \text{mayNeverRead}_S(t_i, r_i)) \wedge \\ (\text{mayRead}_H(\tau_i, r_i) \rightarrow \text{mayNeverWrite}_S(t_i, r_i)). \end{aligned} \quad (3)$$

Because the newly created object, which corresponds to the reference r , implicitly plays the PURE role for all tasks except t , the terms $\text{mayRead}_H(\tau_i, r)$ and $\text{mayWrite}_H(\tau_i, r)$ are both false for all t_i that are not children of t . The implication is therefore true:

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) - \{t_i \mid \text{parent}_{Ts}(t_i) = t\} : \\ (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_S(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_S(t_i, r)). \end{aligned} \quad (4)$$

On the other hand, for t_i that are children of t , the terms $\text{mayRead}_H(\tau_i, r)$ and $\text{mayWrite}_H(\tau_i, r)$ are true. Therefore, we show that $\text{mayNeverRead}_S(t_i, r)$ and $\text{mayNeverWrite}_S(t_i, r)$ are also true. Since $\text{mayNeverRead}_S(t_i, r)$ actually implies $\text{mayNeverWrite}_S(t_i, r)$, we only need to prove the former. Theorem A.3.2 states $\text{mayRead}_H(t, r) \rightarrow r \in R^{\text{RW}} \cup R^{\text{RO}}$, for all t and r in a legal program state $\langle H, Ts \rangle$. Therefore, the inverse holds as well: $r \notin R^{\text{RW}} \cup R^{\text{RO}} \rightarrow \neg \text{mayRead}_H(t, r)$. As $\text{mayNeverRead}_S(t_i, r)$ is defined as $\forall \langle H', T'_s \rangle \in \xrightarrow{\text{legal}}_*(S) : \text{finished}_{T'_s}(t_i) \vee \neg \text{mayRead}_{H'}(t_i, r)$, we can instead prove the following:

$$\begin{aligned} \forall \langle H', T'_s \rangle \in \xrightarrow{\text{legal}}_*(S) : \\ \forall \langle t_i, t, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in \{\text{task}_{T'_s}(t_i) \mid \text{parent}_{T'_s}(t_i) = t\} : \\ \text{finished}_{T'_s}(t_i) \vee r \notin R^{\text{RW}} \cup R^{\text{RO}}. \end{aligned} \quad (5)$$

We show this again using induction. In the base case $S' = S$, this property holds, as the new object is added only to the R^{RW} sets of t and t 's ancestors, but not to t 's children. For the inductive step, where $S' \neq S$, there exists a

predecessor state $S'_p = \langle H'_p, Ts'_p \rangle \in \xrightarrow{\text{legal}}*(S)$, such that $S'_p \longrightarrow S'$. We assume that the property holds for $\langle H'_p, Ts'_p \rangle$:

$$\forall \langle t_i, t, R^{\text{RW}}, R^{\text{RO}}, _ \rangle \in \{\text{task}_{Ts'_p}(t_i) \mid \text{parent}_{Ts'_p}(t_i) = t\} : \\ \text{finished}_{Ts'_p}(t_i) \vee r \notin R^{\text{RW}} \cup R^{\text{RO}}.$$

Then, we have again five cases, which correspond to the five operations:

$S'_p \xrightarrow{\text{-create } r'} S'$: Since the property holds for S'_p , and because the create operation only changes the R^{RW} sets with respect to the new reference r' , but not with respect to r (and does not change the R^{RO} sets at all), the property holds for S' as well.

$S'_p \xrightarrow{\text{-read}} S'$: There are no changes in any R^{RW} or R^{RO} sets, so the property still holds.

$S'_p \xrightarrow{\text{-write}} S'$: Again, no changes in any R^{RW} or R^{RO} sets.

$S'_p \xrightarrow{t' \text{ start } t'_{\text{ch}}(R^{\text{RW}}, R^{\text{RO}})} S'$: The start operation only “changes” the R^{RW} and R^{RO} sets of the new task, but not of any existing task in $\{t_i \mid \text{parent}_{Ts}(t_i) = t\}$. Note that we only have to consider tasks that are children of t when r is created, i.e., for the program state S , but not child tasks of t that are started later, i.e., in state S' . Hence, because the property holds for S_p , it holds trivially for S'_p .

$S'_p \xrightarrow{t' \text{ finish}} S'$: There are again no changes in any R^{RW} or R^{RO} sets, so the property still holds.

Having shown (5), it follows (as discussed above):

$$\forall t_i \in \{t_i \mid \text{parent}_{Ts}(t_i) = t\} : \\ (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_S(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_S(t_i, r))$$

and, together with (4):

$$\forall t_i \in \text{ids}(Ts) : (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_S(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_S(t_i, r)). \quad (6)$$

Finally, because $H' = H'_p \cup \{r\}$, it follows from (3) and (6):

$$\forall t_i \in \text{ids}(Ts) : \forall r_i \in \text{refs}(H) : \\ (\text{mayWrite}_H(\tau_i, r_i) \rightarrow \text{mayNeverRead}_S(t_i, r_i)) \wedge \\ (\text{mayRead}_H(\tau_i, r_i) \rightarrow \text{mayNeverWrite}_S(t_i, r_i)).$$

READ OPERATION: $S_p \xrightarrow{t \text{ read } r} S$ Because child task priority holds for S_p , and because $\text{ids}(S) = \text{ids}(S_p)$ and $H = H_p$, child task priority holds for S too.

WRITE OPERATION: $S_p \xrightarrow{t \text{ write } r} S$ Like above, $\text{ids}(S) = \text{ids}(S_p)$ and $H = H_p$, therefore, because child task priority holds for S_p , it holds for S .

TASK START: $S_p \xrightarrow{t \text{ start } t_{\text{ch}}(R_{\text{ch}}^{\text{RW}}, R_{\text{ch}}^{\text{RO}})} S$ According to the definition of the start operation, $\text{refs}(H) = \text{refs}(H_p)$ and $\text{ids}(Ts) = \text{ids}(Ts_p) \cup \{t_{\text{ch}}\}$. Also, because the state transition does not change the roles of any object for any task t_i 's parent τ_i , except for t_{ch} 's parent t , the following follows from (1):

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) - \{t_{\text{ch}}\} : \forall r \in \text{refs}(H) : \\ (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_{S_p}(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_{S_p}(t_i, r)). \end{aligned} \quad (7)$$

Further, as explained in the object creation case, everything that holds for states in $\xrightarrow{\text{legal}}*(S_p)$ also holds for states in $\xrightarrow{\text{legal}}*(S)$. Thus, $\text{mayNeverRead}_{S_p}(t_i, r)$ implies $\text{mayNeverRead}_S(t_i, r)$, for all t_i and r , and $\text{mayNeverWrite}_{S_p}(t_i, r)$ implies $\text{mayNeverWrite}_S(t_i, r)$. Together with (7), it follows:

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) - \{t_{\text{ch}}\} : \forall r \in \text{refs}(H) : \\ (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_S(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_S(t_i, r)). \end{aligned} \quad (8)$$

To show that child task priority also holds for t_{ch} , we need to separately consider the objects that are referred to by the $R_{\text{ch}}^{\text{RW}}$ set, by the $R_{\text{ch}}^{\text{RO}}$ set (but not by $R_{\text{ch}}^{\text{RW}}$), and by none of them.

For $r \in R_{\text{ch}}^{\text{RW}}$, both $\text{mayRead}_H(t, r)$ and $\text{mayWrite}_H(t, r)$ are false; therefore, the following is trivially true:

$$\begin{aligned} \forall r \in R_{\text{ch}}^{\text{RW}} : (\text{mayWrite}_H(t, r) \rightarrow \text{mayNeverRead}_S(t_{\text{ch}}, r)) \wedge \\ (\text{mayRead}_H(t, r) \rightarrow \text{mayNeverWrite}_S(t_{\text{ch}}, r)). \end{aligned} \quad (9)$$

For $r \in R_{\text{ch}}^{\text{RO}} - R_{\text{ch}}^{\text{RW}}$, $\text{mayWrite}_H(t, r)$ is false, but $\text{mayRead}_H(t, r)$ is true. Therefore, we show that $\text{mayNeverWrite}_S(t_{\text{ch}}, r)$ is also true. Theorem A.3.2 states $\text{mayWrite}_H(t, r) \rightarrow r \in R^{\text{RW}}$, for all t and r in a legal program state $\langle H, Ts \rangle$. Therefore, the inverse holds as well: $r \notin R^{\text{RW}} \rightarrow \neg \text{mayWrite}_H(t, r)$. As $\text{mayNeverWrite}_S(t_{\text{ch}}, r)$ is defined as $\forall \langle H', Ts' \rangle \in \xrightarrow{\text{legal}}*(S) : \text{finished}_{Ts'}(t_{\text{ch}}) \vee \neg \text{mayWrite}_{H'}(t_{\text{ch}}, r)$, we can instead prove the following:

$$\begin{aligned} \forall \langle H', Ts' \rangle \in \xrightarrow{\text{legal}}*(S) : \forall r \in R_{\text{ch}}^{\text{RO}} - R_{\text{ch}}^{\text{RW}} : \text{finished}_{Ts'}(t_{\text{ch}}) \vee r \notin R_{\text{ch}}^{\text{RW}}, \\ \text{where } \text{task}_{Ts'}(t_{\text{ch}}) = \langle t_{\text{ch}}, t, R_{\text{ch}}^{\text{RW}}, _ , _ \rangle. \end{aligned} \quad (10)$$

We show this again using induction. In the base case $S' = S$, this property holds, because $R_{\text{ch}}^{\text{RW}} = R_{\text{ch}}^{\text{RW}}$. For the inductive step, where $S' \neq S$, there

exists a predecessor state $S'_p = \langle H'_p, Ts'_p \rangle \in \xrightarrow{\text{legal}}^*(S)$, such that $S'_p \longrightarrow S'$. We assume that the property holds for $\langle H'_p, Ts'_p \rangle$:

$$\forall r \in R_{\text{ch}}^{\text{RO}} - R_{\text{ch}}^{\text{RW}} : \text{finished}_{Ts'_p}(t_{\text{ch}}) \vee r \notin R_{\text{ch}}^{\text{RW}},$$

$$\text{where task}_{Ts'_p}(t_{\text{ch}}) = \langle t_{\text{ch}}, t, R_{\text{ch}}^{\text{RW}}, \rightarrow, _ \rangle.$$

Then, we have again five cases, which correspond to the five operations:

$S'_p \xrightarrow{\text{-create } r'} S'$: Since the property holds for S'_p , and because the create operation only changes the R^{RW} sets with respect to the new reference r' , but not with respect to any existing $r \in R_{\text{ch}}^{\text{RO}} - R_{\text{ch}}^{\text{RW}}$, the property holds for S' as well.

$S'_p \xrightarrow{\text{-read}} S'$: There are no changes in any R^{RW} set, so the property still holds.

$S'_p \xrightarrow{\text{-write}} S'$: Again, no changes in any R^{RW} set.

$S'_p \xrightarrow{t' \text{ start } t'_{\text{ch}}(R^{\text{RW}}, R^{\text{RO}})} S'$: The start operation only “changes” the R^{RW} set of the new task, but not of any existing task such as t_{ch} . Hence, because the property holds for S_p , it holds trivially for S'_p .

$S'_p \xrightarrow{t' \text{ finish}} S'$: There are again no changes in any R^{RW} set, so the property still holds.

Having shown (10), it follows:

$$\forall r \in R_{\text{ch}}^{\text{RO}} - R_{\text{ch}}^{\text{RW}} : (\text{mayWrite}_H(t, r) \rightarrow \text{mayNeverRead}_S(t_{\text{ch}}, r)) \wedge$$

$$(\text{mayRead}_H(t, r) \rightarrow \text{mayNeverWrite}_S(t_{\text{ch}}, r)). \quad (11)$$

Finally, for all $r \notin R_{\text{ch}}^{\text{RW}} \cup R_{\text{ch}}^{\text{RO}}$, the roles stay the same. Therefore, it follows from (1):

$$\forall r \in \text{refs}(H) - (R_{\text{ch}}^{\text{RW}} \cup R_{\text{ch}}^{\text{RO}}) :$$

$$(\text{mayWrite}_H(t, r) \rightarrow \text{mayNeverRead}_S(t_{\text{ch}}, r)) \wedge$$

$$(\text{mayRead}_H(t, r) \rightarrow \text{mayNeverWrite}_S(t_{\text{ch}}, r)),$$

and, when combined with (9) and (11):

$$\forall r \in \text{refs}(H) : (\text{mayWrite}_H(t, r) \rightarrow \text{mayNeverRead}_S(t_{\text{ch}}, r)) \wedge$$

$$(\text{mayRead}_H(t, r) \rightarrow \text{mayNeverWrite}_S(t_{\text{ch}}, r)),$$

At last, when combined with (8), it follows:

$$\forall t_i \in \text{ids}(Ts) : \forall r_i \in \text{refs}(H) :$$

$$(\text{mayWrite}_H(\tau_i, r_i) \rightarrow \text{mayNeverRead}_S(t_i, r_i)) \wedge$$

$$(\text{mayRead}_H(\tau_i, r_i) \rightarrow \text{mayNeverWrite}_S(t_i, r_i)).$$

TASK FINISH: $S_p \xrightarrow{t \text{ finish}} S$ According to the definition of the $\xrightarrow{t \text{ finish}}$ operation, $\text{refs}(H) = \text{refs}(H_p)$ and $\text{ids}(Ts) = \text{ids}(Ts_p)$. Also, because the state transition does not change the roles of any object for any task t_i 's parent τ_i , except for t 's parent τ , and for t itself, the following follows from (1):

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) - \{t\} - \{t_i \mid \text{parent}_{Ts}(t_i) = t\} : \forall r \in \text{refs}(H) : \\ (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_{S_p}(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_{S_p}(t_i, r)). \end{aligned} \quad (12)$$

As explained before, everything that holds for states in $\xrightarrow{\text{legal}}*(S_p)$ also holds for states in $\xrightarrow{\text{legal}}*(S)$. Thus, $\text{mayNeverRead}_{S_p}(t_i, r)$ implies $\text{mayNeverRead}_S(t_i, r)$, for all t_i and r , and $\text{mayNeverWrite}_{S_p}(t_i, r)$ implies $\text{mayNeverWrite}_S(t_i, r)$. Together with (12), it follows:

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) - \{t\} - \{t_i \mid \text{parent}_{Ts}(t_i) = t\} : \forall r \in \text{refs}(H) : \\ (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_S(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_S(t_i, r)). \end{aligned} \quad (13)$$

As $\text{mayNeverRead}_S(t, r)$ is defined as $\forall \langle H', Ts' \rangle \in \xrightarrow{\text{legal}}*(S) : \text{finished}_{Ts'}(t) \vee \neg \text{mayRead}_{H'}(t_{\text{ch}}, r)$, and similarly $\text{mayNeverWrite}_S(t, r)$, both are true in S , as t has finished. Child tasks of t have also finished, otherwise t could not finish, as defined by the finish operation. Thus, from (13) follows:

$$\begin{aligned} \forall t_i \in \text{ids}(Ts) : \forall r \in \text{refs}(H) : \\ (\text{mayWrite}_H(\tau_i, r) \rightarrow \text{mayNeverRead}_S(t_i, r)) \wedge \\ (\text{mayRead}_H(\tau_i, r) \rightarrow \text{mayNeverWrite}_S(t_i, r)). \end{aligned}$$

□

A.5 NONINTERFERENCE

Having shown child task priority, we can finally prove *noninterference*. First, we define interference formally. Since interference concerns operations, for which there is no explicit notion in the definition in Section 2.3, we represent an operation “ x ” implicitly using the pair of states S_x and S'_x before and after the operation.

Definition A.5.1 (Interference). Two operations a and b in tasks t_a and t_b are said to interfere if one of them is a write operation and the other is a read or a write operation and the two operations have the same reference r as the target (and are successful):

$$\begin{aligned} \text{interfere}(S_a, S'_a, t_a, S_b, S'_b, t_b) \triangleq S'_a \notin \{S_a, S_{\text{error}}\} \wedge S'_b \notin \{S_b, S_{\text{error}}\} \wedge \\ (\exists r : S_a \xrightarrow{t_a \text{ writes } r} S'_a \wedge S_b \xrightarrow{t_b \text{ reads } r} S'_b \vee \\ S_a \xrightarrow{t_a \text{ writes } r} S'_a \wedge S_b \xrightarrow{t_b \text{ writes } r} S'_b \vee \\ S_a \xrightarrow{t_a \text{ reads } r} S'_a \wedge S_b \xrightarrow{t_b \text{ writes } r} S'_b) \end{aligned}$$

THEOREM A.5.2 (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 . Formally, if any legal state S' is the result of a successful start operation, then S' cannot be followed by an operation in τ and then by an operation in t , such that these operations interfere:*

$$\forall \text{legal}(S') : (\exists S : S \xrightarrow{\tau \text{ start } t(R^{RW}, R^{RO})} S' \wedge S \neq S') \rightarrow \neg \exists S_\tau, S'_\tau, S_t, S'_t : \\ S' \longrightarrow^* S_\tau \wedge S'_\tau \longrightarrow^* S_t \wedge \text{interfere}(S_\tau, S'_\tau, \tau, S_t, S'_t, t).$$

For the following proof, we assume $S = \langle H, Ts \rangle$, $S' = \langle H', Ts' \rangle$, $S_\tau = \langle H_\tau, Ts_\tau \rangle$, and so on.

PROOF. We prove noninterference by contradiction, assuming that some successful start operation is indeed followed by two interfering operations, as described above. Formally, we assume:

$$\exists S, S', S_\tau, S'_\tau, S_t, S'_t : \text{legal}(S') \wedge S \xrightarrow{\tau \text{ start } t(R^{RW}, R^{RO})} S' \wedge S \neq S' \wedge \\ S' \longrightarrow^* S_\tau \wedge S'_\tau \longrightarrow^* S_t \wedge \text{interfere}(S_\tau, S'_\tau, \tau, S_t, S'_t, t). \quad (1)$$

From the definition of interference (Definition A.5.1) follows that interference takes place for a reference r in one of the three following ways:

$$\exists r : S_\tau \xrightarrow{\tau \text{ writes } r} S'_\tau \wedge S_t \xrightarrow{t \text{ reads } r} S'_t \quad \vee \\ S_\tau \xrightarrow{\tau \text{ writes } r} S'_\tau \wedge S_t \xrightarrow{t \text{ writes } r} S'_t \quad \vee \\ S_\tau \xrightarrow{\tau \text{ reads } r} S'_\tau \wedge S_t \xrightarrow{t \text{ writes } r} S'_t. \quad (2)$$

In addition, it follows:

$$S'_\tau \notin \{S_\tau, S_{\text{error}}\} \wedge S'_t \notin \{S_t, S_{\text{error}}\}. \quad (3)$$

We continue the proof for the three cases in (2) separately.

$S_\tau \xrightarrow{\tau \text{ writes } r} S'_\tau \wedge S_t \xrightarrow{t \text{ reads } r} S'_t$: Given that the write operation in τ is successful, as implied by (3), the definition of the write operation implies $\text{mayWrite}_{Ts_\tau}(\tau, r)$. Further, because of child task priority (Theorem A.4.1), it follows $\text{mayNeverRead}_{S'_t}(t, r)$, which is defined as:

$$\forall \langle H_i, Ts_i \rangle \in \xrightarrow{\text{legal}}^*(S_\tau) : \text{finished}_{Ts_i}(t) \vee \neg \text{mayRead}_{H_i}(t, r).$$

Because S_t is actually in $\xrightarrow{\text{legal}}^*(S_\tau)$, as assumed in (1), it follows:

$$\text{finished}_{Ts_t}(t) \vee \neg \text{mayRead}_{H_t}(t, r). \quad (4)$$

However, given that the read operation in t is assumed to be successful (implied by (3)), it follows from the definition of the read operation:

$$\neg \text{finished}_{Ts_t}(t) \wedge \text{mayRead}_{H_t}(t, r),$$

which contradicts (4).

$S_\tau \xrightarrow{\tau \text{ writes } r} S'_\tau \wedge S_t \xrightarrow{t \text{ writes } r} S'_t$: As in the case above, Equation (3) and Theorem A.4.1 together imply $\text{mayNeverRead}_{S'_\tau}(t, r)$, which in turn implies:

$$\text{finished}_{T_{S'_\tau}}(t) \vee \neg \text{mayRead}_{H_t}(t, r),$$

and, according to the definitions of $\text{mayRead}_{H_t}(t, r)$ and $\text{mayWrite}_{H_t}(t, r)$, also:

$$\text{finished}_{T_{S'_\tau}}(t) \vee \neg \text{mayWrite}_{H_t}(t, r). \quad (5)$$

Yet, it follows from the definition of the write operation:

$$\neg \text{finished}_{T_{S'_\tau}}(t) \wedge \text{mayWrite}_{H_t}(t, r),$$

which contradicts (5).

$S_\tau \xrightarrow{\tau \text{ read } r} S'_\tau \wedge S_t \xrightarrow{t \text{ writes } r} S'_t$: Similarly to the previous two cases, given that the read operation in τ is successful, as implied by (3), the definition of the read operation implies $\text{mayRead}_{T_{S'_\tau}}(\tau, r)$. Further, because of child task priority (Theorem A.4.1), it follows $\text{mayNeverWrite}_{S'_\tau}(t, r)$, which is defined as:

$$\forall \langle H_i, T_{S'_\tau} \rangle \in \xrightarrow{\text{legal}} *(S'_\tau) : \text{finished}_{T_{S'_\tau}}(t) \vee \neg \text{mayWrite}_{H_i}(t, r).$$

Because S_t is actually in $\xrightarrow{\text{legal}} *(S_\tau)$, as assumed in (1), it follows:

$$\text{finished}_{T_{S_t}}(t) \vee \neg \text{mayWrite}_{H_t}(t, r). \quad (6)$$

However, given that the write operation in t is assumed to be successful, it follows from the definition of the write operation:

$$\neg \text{finished}_{T_{S_t}}(t) \wedge \text{mayWrite}_{H_t}(t, r),$$

which contradicts (6).

As all three cases lead to a contradiction, the assumption we made in (1) is false. Therefore, the opposite, which is Theorem A.5.2, must be true. \square

BIBLIOGRAPHY

- 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.
- Jamie Allen. 2013. *Effective Akka*. O’Reilly and Associates, Sebastopol, CA, USA.
- 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.
- 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.
- 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.
- 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.
- Martin Bättig and Thomas R. Gross. 2017. Synchronized-by-Default Concurrency for Shared-Memory Systems. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’17)*. ACM, New York, NY, USA, 299–312.
- 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.
- 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.
- 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.
- 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.
- 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.
- John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis (SAS '03)*. Springer, Berlin, Heidelberg, 55–72.
- 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.
- Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Saĝnak Taşırlar. 2010. Concurrent Collections. *Scientific Programming* 18, 3 (Jan. 2010), 203–217.
- 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)*.
- 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.

- 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.
- Jack W. Davidson and Anne M. Holler. 1992. Subprogram Inlining: A Study of Its Effects on Program Execution Time. *IEEE Trans. Softw. Eng.* 18, 2 (Feb. 1992), 89–102.
- 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.
- Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *Proceedings of the 21th European Conference on Object-Oriented Programming (ECOOP '07)*, Erik Ernst (Ed.). Springer-Verlag, Berlin, Heidelberg, 28–53.
- 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. 2018a. Concurrency-Aware Object-Oriented Programming with Roles. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 130:1–130:30.
- Michael Faes and Thomas R. Gross. 2018b. Efficient VM-Independent Runtime Checks for Parallel Programming. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2018)*. ACM, New York, NY, USA, 5–15.
- David Gay, Joel Galenson, Mayur Naik, and Kathy Yelick. 2011. Yada: Straightforward Parallel Programming. *Parallel Comput.* 37, 9 (Sept. 2011), 592–609.
- Aaron Greenhouse and John Boyland. 1999. An Object-Oriented Effects System. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*. Springer-Verlag, London, UK, 205–229.
- 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.
- Robert H. Halstead, Jr. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538.

- Tim Harris and Keir Fraser. 2003. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*. ACM, New York, 388–402.
- 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.
- 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.
- 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.
- Benjamin Hindman and Dan Grossman. 2006. Atomicity via Source-to-Source Translation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC '06)*. ACM, New York, NY, USA, 82–91.
- Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450.
- 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.
- Günter Kniesel. 1996. *Encapsulation = Visibility + Accessibility*. Technical Report IAI-TR-96-12. Institut für Informatik, Universität Bonn, Bonn, Germany. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.1428>
- Günter Kniesel and Dirk Theisen. 1999. JAC: Java with Transitive Readonly Access Control. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems (IWA00S '99)*. <http://janvitek.org/events/ecoopws/iwa00s/papers/index.html>
- Guy Korland, Nir Shavit, and Pascal Felber. 2010. Noninvasive Concurrency with Java STM. In *Programmability Issues for Heterogeneous Multicores (MultiProg 2010)*. <https://sites.google.com/site/deucestm/documentation>

- 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.
- 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.
- 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.
- 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.
- Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (May 2006), 33–42.
- K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. 2002. Using Data Groups to Specify and Check Side Effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 246–257.
- 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.
- J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 47–57.
- 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.
- 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.
- Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

- 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>
- Oracle Corporation. 2018. Java SE Platform Security Architecture. <https://docs.oracle.com/en/java/javase/11/security/java-se-platform-security-architecture.html>
- Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA.
- Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008. Path Specialization: Reducing Phased Execution Overheads. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, NY, USA, 81–90.
- 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.
- Stephen Richardson and Mahadevan Ganapathi. 1989. Interprocedural Analysis vs. Procedure Integration. *Inf. Process. Lett.* 32, 3 (Aug. 1989), 137–142.
- 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.
- Atanas Rountev. 2005. Component-Level Dataflow Analysis. In *Proceedings of the 8th International Conference on Component-Based Software Engineering (CBSE'05)*. Springer-Verlag, Berlin, Heidelberg, 82–89.
- Atanas Rountev, Scott Kagan, and Thomas Marlowe. 2006. Interprocedural Dataflow Analysis in the Presence of Large Libraries. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 2–16.
- Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. 189–234.
- 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.
- L. A. Smith, J. M. Bull, and J. Obdržálek. 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.

- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. 2008. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th Annual*

Symposium on Parallelism in Algorithms and Architectures (SPAA '08).
ACM, New York, NY, USA, 265–274.