KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

A Type System for Ensuring Safe, Structured Concurrency in Scala

SEBASTIAN WILLENBRINK

Stockholm, Sweden, 2024

A Type System for Ensuring Safe, Structured Concurrency in Scala

SEBASTIAN WILLENBRINK

Degree Programme in Computer Science and Engineering
Date: November 19, 2024

Supervisor: Philipp Haller
Examiner: Roberto Guanciale
School of Electrical Engineering and Computer Science
Swedish title: Ett typsystem för säker och strukturerad samtidighet

Abstract

Concurrent programming brings many pitfalls, such as data races, deadlocks and nondeterminism. Designing programming languages to eliminate these classes of bugs is an active area of research. Previous attempts at this focused on creating completely new languages, hampering adoption and reuse of existing code, or were either insufficient to address all of data races, deadlocks and nondeterminism or required annotating code, complicating reuse of existing libraries.

In this thesis, we present a new approach to integrating safe concurrency into an existing mainstream programming language, Scala. To do so, we combine two approaches: We represent concurrency in code using the Async Finish Model, a variant of the widely used Fork Join Model. It is inherently free of deadlocks and deterministic. To ensure data race freedom, we isolate subgraphs of the heap by building on the work of LaCasa [HL16]. LaCasa requires no annotations as the compiler can automatically verify that this isolation is preserved.

We provide an informal overview of a language combining these two approaches along with a formalization of the operational semantics. In addition, we show the progress of our reduction and preservation of the necessary invariants as well as isolation of asynchronous tasks. Progress implies deadlock freedom and isolation of asynchronous tasks implies data race freedom. We also formally state and provide an argument for determinism.

Actually write preservation down

Keywords

Concurrent programming, Object-oriented programming, Operational semantics, Scala

Sammanfattning

Samtidig programmering medför många fallgropar, t.ex. samtidiga åtkomst till samma minne, baklås och nondeterminism. Att utforma programmeringsspråk för att eliminera dessa typer av buggar är ett aktivt forskningsområde. Tidigare försök har fokuserat på att skapa helt nya språk, vilket försvårar användning och återanvändning av befintlig kod, eller så har de antingen inte varit tillräckliga för att hantera alla nämnda problem eller så har de krävt att koden annoteras, vilket försvårar återanvändning av befintliga bibliotek.

I den här avhandlingen presenterar vi en ny metod för att integrera säker samtidighet i ett befintligt vanligt programmeringsspråk, Scala. För att göra detta kombinerar vi två metoder: Vi representerar samtidighet i kod med hjälp av det så kallade Async Finish Model, en variant av den allmänt använda Fork Join Model. Den är i sig själv fri från baklås och deterministisk. För att säkerställa frihet från samtidiga minneståkomster isolerar vi delgrafer av heapen genom att bygga vidare på LaCasas arbete [HL16]. LaCasa kräver inga annotationer eftersom kompilatorn automatiskt kan verifiera att denna isolering bevaras.

Vi ger en informell översikt över ett språk som kombinerar dessa två tillvägagångssätt tillsammans med en formalisering av den operativa semantiken. Dessutom visar vi reduktionens framsteg och bevarande av semantikens invarianter samt isolering av asynkrona trådar. Reduktionens framsteg innebär frihet från baklås och isolering av asynkrona trådar innebär frihet från samtidiga minnesåtkomster. Vi formaliserar och ger dessutom ett argument för determinism.

Nyckelord

Samtidig programmering, Objekt-orienterad programmering, Operationell semantik, Scala

Zusammenfassung

Parallele Programmierung birgt viele Stolperfallen, wie zum Beispiel gleichzeitige Speicherzugriffe, Deadlocks und Nichtdeterminismus. Die Entwicklung von Programmiersprachen zur Vermeidung dieser Fehlerklassen ist ein aktives Forschungsgebiet. Bisherige Versuche konzentrierten sich auf die Entwicklung völlig neuer Sprachen, was die Übernahme und Wiederverwendung von bestehendem Code erschwert, oder sie waren entweder unzureichend, um alle genannten Probleme zu verhindern, oder sie erforderten die Annotierung von Code, was die Wiederverwendung bestehender Bibliotheken erschwert.

In dieser Arbeit stellen wir einen neuen Ansatz zur Integration von sicherer Parallelität in eine bestehende Mainstream-Programmiersprache, Scala, vor. Zu diesem Zweck kombinieren wir zwei Ansätze: Wir stellen die Nebenläufigkeit im Code mithilfe des Async Finish Model dar, einer Variante des weit verbreiteten Fork Join Model. Es ist von Natur aus frei von Deadlocks und deterministisch. Um die Freiheit von gleichzeitigen Speicherzugriffen zu gewährleisten, isolieren wir Teilgraphen des Heaps, indem wir auf der Arbeit von LaCasa [HL16] aufbauen. LaCasa erfordert dabei keine Anmerkungen, da der Compiler automatisch überprüfen kann, dass diese Isolation erhalten bleibt.

Wir geben einen informellen Überblick über eine Sprache, die diese beiden Ansätze kombiniert, sowie eine Formalisierung der operativen Semantik dieser Sprache. Darüber hinaus zeigen wir den Fortschritt der Reduktion und die Erhaltung der Invarianten der Semantik sowie die Isolierung asynchroner Ausführungseinheiten. Fortschritt der Reduktion impliziert Deadlock-Freiheit und die Isolierung asynchroner Aufgaben impliziert die Abwesenheit paralleler Speicherzugriffe. Wir formalisieren und liefern außerdem ein Argument für Determinismus.

Schlüsselwörter

Parallele Programmierung, Objekt-orientierte Programmierung, Operative Semantik, Scala

Acknowledgments

I would like to thank my supervisor at KTH, Philipp Haller, for his reliable and extensive support over the course of the thesis. I would also like to thank my supervisor at TUM, Kevin Kappelmann, for the valuable discussions.

Stockholm, November 2024
Sebastian Willenbrink

Contents

1 Introduction **1**
1.1 Contribution 2

2 Background **5**
2.1 Scala 5
2.2 Object Capabilities 5
2.3 Boxes 6
2.4 Affinity through CPS 6
2.5 Concurrency Models 7

3 Overview of Parallel LaCasa **9**
3.1 First Step: Async Finish Model 9
3.2 Second Step: Boxes and Object Capabilities 10
3.3 Third Step: Permissions 12
3.4 Fourth Step: Unique Fields 13
3.5 Merging Boxes 14

4 Formalization **18**
4.1 Syntax 18
4.2 Dynamic Semantics 23
4.2.1 Single Frame Reduction 23
4.2.2 Frame Stack Reduction 25
4.2.3 Task Set Reduction 26
4.2.4 Initial state 30
4.3 Static Semantics 31
4.3.1 Type Checking 31
4.3.2 Well-Formed Programs 31
4.3.3 Guarded Types and Permissions 32
4.3.4 Object Capabilities 33
4.3.5 Term and Expression Typing 34

4.3.6 Subtyping . . . 36
4.3.7 Well-Formedness . . . 36

5 Properties **40**
5.1 Progress . . . 40
5.2 Isolation . . . 42
5.3 Preservation . . . 44

6 Related Work **46**
6.1 LaCasa . . . 46
6.2 X10 . . . 47
6.3 Habanero-Java . . . 48
6.4 Deterministic Parallel Java . . . 48
6.5 Rust . . . 49
6.6 LVars . . . 49

7 Conclusion **51**
7.1 Future Work . . . 51

A Definitions **53**

B Proofs **56**
B.1 Progress . . . 56
B.1.0.0.1 Finish . . . 56
B.1.0.0.2 Async . . . 56
B.1.0.0.3 Capture . . . 58
B.1.0.0.4 Swap . . . 58
B.1.0.0.5 Select . . . 59
B.1.0.0.6 Assign . . . 60
B.1.0.0.7 Remaining rules . . . 60
B.2 Isolation . . . 60
B.2.0.0.1 E-TASK-DONE . . . 61
B.2.0.0.2 E-RESUME . . . 61
B.2.0.0.3 E-FINISH . . . 62
B.2.0.0.4 E-ASYNC . . . 62
B.2.0.0.5 E-CAPTURE . . . 63
B.2.0.0.6 E-SWAP . . . 63
B.2.0.0.7 E-INVOKE . . . 63
B.2.1 E-STACK . . . 64
B.2.1.0.1 E-BOX . . . 64

B.2.1.0.2 E-RETURN1 64
B.2.1.0.3 E-RETURN2 64
B.2.2 E-FRAME 64
B.2.2.0.1 E-NULL 64
B.2.2.0.2 E-VAR 64
B.2.2.0.3 E-SELECT 64
B.2.2.0.4 E-ASSIGN 65
B.2.2.0.5 E-NEW 65

References **66**

References **66**

List of Figures

1.1 Sequential tree sum and a safe parallel translation 3

3.1 A sequential tree sum implementation 10
3.2 Tree sum using the basic Async Finish Model 11
3.3 Tree sum using boxes to improve isolation 11
3.4 Tree sum with explicit permissions and continuations. Normally, they are implicit. 13
3.5 Safe and parallel tree sum with explicit permissions and continuations 15
3.6 Safe and parallel tree sum with implicit permissions and continuations 16

4.1 Syntax of **PLC** 19
4.2 Terms and expressions of **PLC** 21
4.3 Single frame evaluation rules 24
4.4 Frame stack evaluation rules 25
4.5 Task set evaluation rules introduced by **PLC** 26
4.6 Task set evaluation rules adapted from LaCasa 27
4.7 Rules for well-formed programs 32
4.8 Object capability rules 33
4.9 Standard term typing rules 34
4.10 Non-standard term typing rules 35
4.11 Subtyping rules 37
4.12 Well-formedness rules 37
4.13 Frame typing rules 38

5.1 Definitions for isolation 43

6.1 Properties of related work 46

A.1 Predicates 54

A.2 Invariants 55

Chapter 1

Introduction

Concurrent programming brings many pitfalls, such as data races, deadlocks and livelocks. In addition, common software development tasks, including reproducing program executions and finding sources of bugs, are typically harder due to various sources of nondeterminism introduced by concurrent programming models. For example, shared-memory threads commonly permit large numbers of thread interleavings. [Goe+06]

In order to increase the safety of concurrent programs, a substantial body of prior work has explored the static prevention of data races [BR01; HL16] and deadlocks, using session types [HYC16] or other type systems [BLR02].

Pervasive mutability and aliasing as typical in imperative object-oriented programming languages pose a major challenge. To prevent data races, we either limit mutation, which negatively impacts performance and is thus not desirable, or control aliasing which does not have this drawback. Doing this statically requires more advanced type systems.

Many approaches to alias control rely on creating a new language such as X10 [Cha+05] or Rust [MK14] while others require extensive annotations of the classes and methods used in concurrent programs [Boc+09]. Due to the wealth of existing programs and the impact of existing libraries on developer productivity, these approaches severely impact adoption.

Prior work on lightweight concurrency extensions of existing programming languages exists but has some shortcomings. For example, while LaCasa [HL16] ensures data race freedom, it does not ensure deadlock freedom or provide a deterministic concurrency model. Habanero-Java [Cav+11] does provide deadlock freedom and determinism in the absence of data races but only integrates with diagnostic tools to analyse data race freedom. Deadlocks and nondeterminism, especially in combination with each other, result in bugs that are hard to reason about. Data races are even worse as they are sensitive to minor changes and may remain undetected.

Finally, a language providing safe concurrency requires trust in its soundness. A formalization of core concepts can help understand the mechanisms involved. Proofs of properties such as progress and isolation are also important as they establish deadlock freedom and data-race freedom, respectively.

In this thesis, we present a new approach to integrating safe concurrency into the existing mainstream programming language Scala. Our language uses the Async Finish model for concurrency that is inherently free of deadlocks and deterministic [Cha+05]. Furthermore, we build on LaCasa [HL16] to ensure isolation and uniqueness without requiring annotations of program code. We successfully combine (a) the semantics of unique references as introduced by LaCasa's boxes and (b) concurrent tasks that temporarily have exclusive ownership of them.

Figure 1.1 shows how a simple sequential programming calculating the sum of a tree can be safely parallelized by wrapping the children in boxes as in LaCasa, starting isolated parallel tasks in these boxes via *async* and awaiting their termination via *finish*. Note that a class used in a concurrent task requires no annotations to indicate its safety. Instead, it only has to avoid global variables.

1.1 Contribution

This thesis makes the following contributions:

- We extend CLC^2, a core language formalizing LaCasa, with the Async Finish model to obtain a language that is deterministic, data-race free and deadlock free without requiring annotations of existing code. We provide a formalization of the operational semantics for this language.

- We provide a type system using object capabilities and unique

```
class Node {
 var children: Option[(Node, Node)]
 var v: Int
 var sum: Int

 def compute() {
  children match
   case None =>
    sum = v
   case Some (left, right) =>
    left.compute()
    right.compute()
    sum = v + left.sum + right.sum
} }

class Node {
 var children: Option[(Box[Node], Box[Node])]
 var v: Int
 var sum: Int

 def compute() {
   children match
   case None =>
    sum = v
   case Some (left, right) =>
    var boxL = swap(left, null)
    var boxR = swap(right, null)

    finish {
     async(boxL, node => node.compute())
     async(boxR, node => node.compute())
    }

    sum = v + boxL.sum + boxR.sum
    swap(left, boxL)
    swap(right, boxR)
} }
```

Figure 1.1: Sequential tree sum and a safe parallel translation

pointers to ensure encapsulation. We emulate affine types using continuation-passing-style.

- We make formal statements of key properties such as progress, preservation, isolation and confluence. We also provide complete proofs of progress and isolation.

state confluence

Chapter 2

Background

2.1 Scala

Scala is a mainstream programming language that is statically typed and supports both object-oriented and functional programming. Its most commonly used execution environment is the Java Virtual Machine (JVM) and its design decisions provide interoperability with Java. Its type system includes subtyping to properly support inheritance. This poses a challenge when extending the type system as it increases the complexity of the type system.

2.2 Object Capabilities

The object capabilities model [Mil06] is a model originating in computer security research. It places constraints on the interaction with objects by requiring a user to hold a capability giving this right. This requires that a user cannot circumvent these capabilities by ignoring the reference graph as this would allow an object to obtain a reference to the protected object without holding the capability.

In the context of this thesis, we say that a class C or a method m adheres to the object capabilities model, written as *ocap(C)* or *ocap(m)*, if it only accesses explicitly passed objects and their transitive references. In particular, global variables are not accessible unless explicitly passed. In addition, an *ocap* method may not create non-*ocap* objects as this would then allow it to call a method accessing global variables.

2.3 Boxes

LaCasa [HL16] introduced the concept of boxes where a box encapsulates an object. This means that it holds an externally unique reference of that object where externally refers to the fact that the object may have an internal reference to itself, i.e. a circular reference. Unique states that no other external reference to this object exists, i.e. it prevents aliasing. External uniqueness is sufficient for the guarantees required in this thesis [CW03].

A box dominates all transitive references of the encapsulated object. This means that a sequence of references from an object outside the box must go “through” the box to reach any object inside. An object is inside a box if it is reachable from the box. No reference is allowed to cross the boundary of a box, thereby ensuring that the box itself remains the only reference pointing into its isolated graph.

Accessing global variables would break this encapsulation. As such, every object contained in a box must adhere to the *ocap* constraint. A box can be thought of as a capability describing the right to interact with the encapsulated object and its transitive references.

2.4 Affinity through CPS

In mainstream type systems, values cannot be consumed. A variable x containing some value will continue to be accessible even if we refer to x somewhere else. Even if x is mutable and its value changed, the variable itself is still available.

Affine types provide a notion of consumption, allowing some operations to invalidate variables and make them inaccessible. This is enforced at compile time. A common example where affine types provide safety are file descriptors which should be consumed by closing them, preventing invalid accesses. We use affine types to ensure safe concurrency: Starting a concurrent task invalidates the object on which it operates in the callers scope.

Most mainstream programming languages, among them Scala, do not support affine types. Nevertheless, we can simulate them by using continuation passing style. Every expression that consumes a value takes a continuation within which this value is no longer available. It is possible to implement this as a compiler plugin for Scala. For more details, refer to LaCasa [HL16].

2.5 Concurrency Models

There exist multiple ways of representing parallel programs. One of the oldest is the Fork-Join Model (FJM) which allows the creation of a process via the *fork* systemcall which can then access the same memory while running in parallel. Later, the original process, termed parent, can await the termination of the copy, or child, via the *join* systemcall [23]. To this day, the FJM is widely spread due to its conceptual simplicity and expressiveness.

At the same time, it is in its most common form unsafe as all memory is shared between the two processes, making data races an omnipresent threat. It is also fundamentally unstructured: A process spawned via *fork* can spawn its own children who might outlive their parent. This creates hard-to-follow concurrency patterns as a *fork* followed by a *join* might leave some grandchild process alive.

In the FJM, processes are identified by ids which can be passed around. This allows any process to await any others termination, providing great flexibility. At the same time, this makes the expected termination order of processes opaque. A child is not guaranteed to terminate before its parent and cycles of processes awaiting each others termination, that is a deadlock, are hard to spot from only analysing the code.

An alternative is the Async Finish Model (AFM), a variant of the Fork Join model (FJM). The AFM mostly keeps the semantics of fork, called *async*, modifying it to take a closure that is evaluated in parallel. In the following we call these parallel processes tasks as it better represents their intended usage and to distinguish them from the FJM. The AFM also modifies join, named *finish*. Instead of awaiting a single task by id, it delimits a scope and awaits all tasks started within, making it obvious when a specific task will have terminated. Furthermore, instead of only blocking until all tasks started in the block terminate, it awaits the termination of the descendants of these tasks too. This guarantees that a data structure modified by a task can be accessed safely after exiting the enclosing *finish*.

This approach has two main advantages over the FJM. It is deterministic, as *finish* awaits all tasks spawned inside of it once, making it impossible for the execution time of a task to affect the parent task as long as concurrent reads and writes are prevented [LP10]. Furthermore, it guarantees deadlock freedom as a task cannot await a task spawned earlier, ensuring that children always terminate before parents.

At the same time, these advantages can be seen as disadvantages. The FJM can be seen as a generalization of the AFM, making it capable of representing patterns that the AFM does not support. Sometimes, non-deterministic concurrency is desired or the structure enforced by the AFM too restrictive. We nevertheless think that these tradeoffs are worth the safety and structure provided by the AFM and use it in our language.

Chapter 3

Overview of Parallel LaCasa

We explain Parallel LaCasa, the language we have designed, via a toy problem. We solve it sequentially in a Scala-like base language and add stepwise onto this base until we obtain a language that supports safe and structured concurrency.

Assume we have a tree with integer values stored in each node. We want to compute the sum of all values and, while doing so, we also want to store the sum of each subtree in the corresponding node. We assume for the sake of simplicity that each node either has no or two children as it reduces the number of pattern matches. While the example is quite simple, it is similar to many parallel algorithms with a tree structure and expensive operations at each node. It also showcases the most important parts of Parallel LaCasa.

A straightforward sequential algorithm is shown in Fig. 3.1. The `compute` method sequentially descends into both the left and right subtree, computing their results first, before adding the results together with the node's own value. This code is not as fast as it could be as it does not take advantage of the problem structure to execute the program in parallel. Each subtree is inherently isolated from the other due to the tree structure which enables trivial parallelization.

3.1 First Step: Async Finish Model

As a first step, we parallelize the example using the Async Finish Model (AFM) introduced in the background. Figure 3.2 shows the adapted code. Instead of two direct calls to the left and right node, we use *async* to create tasks that perform the computation in parallel. In

```
class Node {
 var children: Option[(Node, Node)]
 var v: Int
 var sum: Int

 def compute() {
  children match
   case None =>
    sum = v
   case Some (left, right) =>
    left.compute()
    right.compute()
    sum = v + left.sum + right.sum
 }
}
```

Figure 3.1: A sequential tree sum implementation

addition, we wrap both calls in a *finish* to ensure that they terminate before we finally access their results and sum them up.

3.2 Second Step: Boxes and Object Capabilities

This form of concurrency is not safe in general: a task may operate on different subgraphs of the heap. In particular, global variables make it possible for a task to mutate data in unexpected places. We eliminate this problem by restricting parallel tasks to subgraphs of the heap and completely forbidding global variables inside them.

Note that if a task has only access to one object, it can only affect this object and all its transitive references, isolating the rest of the heap. Although a new object may be created, it is isolated from all preexisting objects. As described in the background, we use boxes in combination with the object capabilities model to provide this guarantee.

Figure 3.3 shows the example with boxes. The children field now contains a tuple of boxed nodes. More importantly, *async* now takes a box whose content is made available in the task. While not shown in the example, the expression *box[C]* creates a guarded box containing an object of class *C*. The expressions is similar to and exists in addition to *new*, which can only be used to create classes and not boxes.

```
class Node {
 var children: Option[(Node, Node)]
 var v: Int
 var sum: Int

 def compute() {
  children match
   case None =>
    sum = v
   case Some (left, right) =>
    finish {
     async(() => left.compute())
     async(() => right.compute())
    }
    sum = v + left.sum + right.sum
} }
```

Figure 3.2: Tree sum using the basic Async Finish Model

```
class Node {
 var children: Option[(Box[Node], Box[Node])]
 var v: Int
 var sum: Int

 def compute() {
  children match
   case None =>
    sum = v
   case Some (left, right) =>
    finish {
     async(left, node => node.compute())
     async(right, node => node.compute())
    }
    sum = v + left.sum + right.sum
} }
```

Figure 3.3: Tree sum using boxes to improve isolation

3.3 Third Step: Permissions

Two major problems are still open: First, two asynchronously spawned tasks may be started in the same box, interfering with each other and leading to data races. Second, it is possible for a parent task to forget wrapping an *async* in a *finish*, leaving the child task free to mutate data while the parent accesses it.

To prevent the first problem, we ensure that only one task can operate on a certain box at once. Every box is guarded by one permission that is created together with the box. An *async* requires the permissions of the box it acts on to be available in the context. In addition, it consumes the permission, preventing another *async* from being started on the same box. Note that *async* may not capture variables from the environment for the task as this would also break isolation. Accessing the contents of a box to read or write non-reference values requires the permission to be available, to ensure that no other task can modify the contents of the box, but does not consume the permission.

Permissions consumed within the scope introduced by *finish* are still available outside. *finish* ensures that the scope is only left after all tasks started inside have terminated, guaranteeing that all boxes that were available before the *finish*-scope are safe to be accessed afterwards.

Permissions are affine types, as they can be consumed at most once after their creation. Unfortunately, Scala does not support affine types. Instead, we can use CPS to simulate permissions, entering a new scope that does not contain a consumed permission when necessary.

Figure 3.4 shows an example of how CPS and implicits can be used to implement permissions in Scala. We create two boxes using the *box* expression, which takes a continuation in which the respective box is available and an implicit permission for that box. *finish* wraps a scope and takes a continuation. Inside of the scope, we start two tasks with *async*. The permissions for the boxes are provided and made unavailable in the continuations of the *asyncs*. These permissions are available after the scope introduced by *finish* is left, allowing us to access the integer fields of the boxes afterwards to compute the sum.

Note that we do not use the `children` field as there are no *implicits* to introduce their permissions. This makes fields containing boxes impossible to use without explicitly passing permissions, which we want to avoid. In the next section, we will explore which changes are

```
class Node {
 var children: Option[(Box[Node], Box[Node])]
 var v: Int
 var sum: Int

 def compute(contAfterMethod:  Continuation) {
  box[Node] { boxL =>
   implicit boxLPerm = ...
   box[Node] { boxR =>
    implicit boxRPerm = ...
    finish { contAfterFinish =>
     async(boxL, node => node.compute())(boxLPerm) {
          async(boxR, node => node.compute())(boxRPerm) {
       contAfterFinish()
      }
     }
    }
    sum = v + (boxL.sum)(boxLPerm) + (boxR.sum)(boxRPerm)
    contAfterMethod()
} } } }
```

Figure 3.4: Tree sum with explicit permissions and continuations. Normally, they are implicit.

necessary to make fields containing boxes work with permissions.

3.4 Fourth Step: Unique Fields

In the previous example, we did not store boxes in fields. Since permissions cannot be stored in fields (they could easily be duplicated through multiple reads), boxes in fields are impractical: we can only obtain the box, not the permission, when reading the fields content.

To improve this, we want to ensure that a field containing a box is the only reference to that box. We do so by consuming the permission on writes to a field. As this permission cannot be recreated, this invalidates all variables that hold a reference to the box. To access the field, we employ destructive reads which at the same time guards the box with a new permission. These so-called unique fields are able to contain boxes without requiring explicit permission passing to access these boxes.

We introduce a new expression for reads and writes of unique fields, called *swap*. This expression takes a unique field and a box, replacing the content of the field by the box and returning the previous content as a box with a newly created permission, consuming the permission for the written box in the process.

Figure 3.5 shows the final solution for the example problem. We first obtain the boxes from the fields by first swapping them with `null`. Afterwards, we proceed as previously, ultimately storing the boxes in the fields again. Figure 3.6 shows the same solution but without the implicit permissions as they can be inferred by the compiler.

Note that *swap* cannot be nested inside of *finish* as one would expect. Instead of the *finish* returning once all tasks have terminated, it will never return. This is necessary as the *swap* permanently consumes the permission of the box it stored in the field. Normally, the *finish* scope is exited after the expressions inside of it are evaluated and all tasks have terminated. This is sound as all permissions consumed inside of the scope are once again safe to use at this point. This does not apply if a permission was permanently consumed by *swap*. This is not a major problem as the *swap* can instead be performed outside of the *finish* or in one of the tasks spawned inside of the *finish*.

3.5 Merging Boxes

An object stored in a box cannot be easily accessed. Firstly, a permission is always required. Secondly, the manner of access depends on the type of the field. We distinguish three different types of fields. For non-reference typed fields, accessing the object inside of a box and reading or writing the fields is safe as we hold the permission and no other task may access the box at the same time. Unique fields, containing boxes, cannot be modified as freely and have to be replaced by another box to obtain them via a *swap*. The last category of fields, containing unboxed objects, cannot be read or modified at all while contained within a boxed object. Instead, the box has to be first opened through an *async*. This is due to the fact that boxes represent isolated areas of the heap and adding or copying references might break that isolation.

Sometimes, it is necessary to access the contents of a box x anyway. This can be done by removing the box completely, merging its isolated

```
class Node {
 var children: Option[(Box[Node], Box[Node])]
 var v: Int
 var sum: Int

 def compute(contAfterMethod:  Continuation) {
   children match
   case None =>
    sum = v
   case Some (left, right) =>
    swap(left, null) { boxL =>
     implicit boxLPerm = ...
     swap(right, null) { boxR =>
      implicit boxRPerm = ...


      finish { contAfterFinish =>
       async(boxL, node => node.compute())(boxLPerm) {
            async(boxR, node => node.compute())(boxRPerm) {
         contAfterFinish()
      } } }
      sum = v + (boxL.sum)(boxLPerm) + (boxR.sum)(boxRPerm)
      swap(left, boxL)(boxLPerm) {
       swap(right, boxR)(boxRPerm) {
        contAfterMethod()
} } } } } }
```

Figure 3.5: Safe and parallel tree sum with explicit permissions and continuations

```
class Node {
 var children: Option[(Box[Node], Box[Node])]
 var v: Int
 var sum: Int

 def compute() {
   children match
   case None =>
    sum = v
   case Some (left, right) =>
    var boxL = swap(left, null)
    var boxR = swap(right, null)

    finish {
     async(boxL, node => node.compute())
     async(boxR, node => node.compute())
    }

    sum = v + boxL.sum + boxR.sum
    swap(left, boxL)
    swap(right, boxR)
} }
```

Figure 3.6: Safe and parallel tree sum with implicit permissions and continuations

area with the isolated area of another box. After starting a task via *async* within this other box, this task can freely access the contents of the box it was started in and the box that was merged into it. To merge two boxes, we introduce another expression, $\mathit{capture}(y.f, x)$. It stores the content of box x in a normal class-typed field f inside of box y, effectively unwrapping box x.

Chapter 4

Formalization

In this section, we formalize the core concepts of this extension. For this, we use a typed object-oriented core language with subtyping based on LaCasa called Parallel LaCasa or **PLC**. First, we describe the syntax. Then, we describe the operational semantics in three steps, starting with the reduction of single frames, then frame stacks and finally sets of tasks. Lastly, we describe the static semantics, i.e. the typing rules and wellformedness.

We formulate our language using small-step semantics and syntax-directed typing rules. We base our language directly on CLC_2 of the LaCasa paper. Nontrivial differences are highlighted with a gray background.

4.1 Syntax

Figure 4.1 shows the syntax of programs and types in **PLC**. A type is either a surface type, which can occur in the surface syntax, or a guarded type, which can only be inferred. Guarded types consist of some Q and some `Box`$[C]$ where each Q is associated with a permission during type-checking to ensure that guarded boxes are only accessed in the presence of their corresponding permission. Surface types encompass return types and boxes. Return types consist of class types, the null type and bottom. Type `Null` is a subtype of all classes and used to assign a type to the *null* value. The bottom type $\perp$ is used as the return type of continuation terms and indicates that they do not return.

The type system has only classes as primitive types as we focus on

$\rho ::=$		return type
	C, D	class type
	$\mid \texttt{Null}$	null type
	$\mid \bot$	bottom type
$\sigma, \tau ::=$		surface type
	ρ	return type
	$\mid \mathrm{Box}[C]$	box type
$\pi ::=$		type
	σ, τ	surface type
	$\mid Q \triangleright \mathrm{Box}[C]$	guarded type
$p ::=$	$\overline{cd}\ \overline{cfd}\ t$	program
$cd ::=$	$\text{class } C \text{ extends } D\ \{\overline{fd}\ \overline{md}\}$	class
	$\mid\ AnyRef$	anyref
$cfd ::=$	$\text{var } f : C$	class field
$ufd ::=$	$\text{var } f :\ \mathrm{Box}[C]$	unique field
$fd ::=$	$cfd \mid ufd$	field
$md ::=$	$\text{def } m(x : \tau) : \rho = t$	method

Figure 4.1: Syntax of **PLC**

the behaviour of references in an execution environment with shared memory. We do not include, for example, integers to simplify the formalization.

A program consists of a sequence of class definitions and global variable definitions, followed by the "main" term, the entry point of the program. Classes themselves define a name C, a superclass D (which might be the superclass of all classes, *AnyRef*, which does not contain any methods or fields) followed by a sequence of fields and methods. Both the sequences in the program and the classes can be empty.

A class field can only store class types. A field storing a box is called a unique field and provides externally unique references to boxes. Externally unique references are the only references pointing from outside a heap area into it. This means that without this reference the pointed-to area would be fully separated from the remaining heap. Unique fields are only externally unique as the object referenced by the field may contain an (indirect) reference to itself. External uniqueness is required to ensure that a box stored in a unique field remains isolated.

Methods have a term as body and take a single argument of arbitrary surface type. The argument object can contain an arbitrary number of fields, simulating multiple arguments. Methods can only return a subset of types, so-called return types. Preventing the return of boxes is not a significant limitation as either the box originated in the callers context or was created inside the method. In the former case, returning the box isn't necessary. In the latter case, the permission to the box is not available to the caller, thus making the box inaccessible and therefore meaningless to return. Both the restriction on the number of arguments and the return type exist to simplify the formalization.

Figure 4.2 shows the terms and expressions of **PLC**. All terms are in the administrative normal form (ANF), that is, all arguments to methods must be constants or variables and the results of expressions must be immediately bound by a variable. The terms are divided into three categories: variables, expressions and continuation terms. Variables and expressions are mostly standard. Due to the ANF, each expression is immediately bound to a variable. This can lead to confusing syntax such as "let $x = y.f = z$ in t" which assigns the value of z to $y.f$ and stores the result of the assignment (defined as y) in the variable x.

$t ::=$	terms
x	variable
$\mid \text{let } x = e \text{ in } t$	let binding
$\mid t^c$	continuation term
$e ::=$	expressions
`null`	null reference
$\mid x$	variable
$\mid x.f$	selection
$\mid x.f = y$	assignment
$\mid \text{new } C$	instance creation
$\mid x.m(y)$	invocation
$\mid \text{finish}\{t\}$	finish
$t^c ::=$	continuation terms
$\text{box}[C]\{x \Rightarrow t\}$	box creation
$\mid \text{capture}(x.f, y)\{z \Rightarrow t\}$	capture
$\mid \text{swap}(x.f, y)\{z \Rightarrow t\}$	swap
$\mid \text{async}(x, y \Rightarrow t)\{s\}$	async

Figure 4.2: Terms and expressions of **PLC**

Continuation terms represent terms that do not return, i.e. have type bottom. Expressions and terms that come after a continuation term will thus never be evaluated, e.g. "$\mathit{capture}(...)\{...\};\text{new } C$" where "new C" is never executed. While the syntax of **PLC** does not allow this specific construct as continuation terms are always in the final position of a term, bottom is still relevant as method calls might not return.

$\mathit{capture}(x.f, y)\{z \Rightarrow t\}$ takes a field f of box x and a box y, storing the content of box y inside the field $x.f$. This consumes the permission of y as the box was captured and is no longer isolated from the box x. Therefore, accessing box y must be impossible to ensure that the contents of x are only accessed through x's permission. t is a continuation that is evaluated without the permission of y and with the variable z containing the same box as x. z is not strictly necessary but kept for consistency with CLC^2. As *capture* returns bottom, the permission for y remains inaccessible.

capture is an irreversible action. After capturing y, x and y are no longer isolated. Alternatively, $\mathit{swap}(x.f, y)\{z \Rightarrow t\}$ stores the content of a box y inside of a unique field f of another box x while simultaneously returning the previous content of $x.f$ wrapped in a new box z. The content of y can then later be retrieved by another swap. Due to the creation and consumption of boxes *swap* also returns bottom and a continuation t.

$\mathit{async}(x, y \Rightarrow t)\{s\}$ takes a box x, a term t and a continuation s. t is evaluated in isolation from the variable environment of *async*; instead it only has access to the content of box x, made available as variable y. As the name implies, the term t is evaluated in parallel to s. *async* consumes the permission for box x to ensure that no two terms can simultaneously access the contents of x. s is evaluated in this environment without the permission for x.

While *async* returns bottom, it functions differently from *capture* and *swap*. Instead of terminating after the continuation is fully evaluated, *async* only skips the evaluation of terms between itself and the immediately enclosing *finish*, similar to an exception skipping evaluation until reaching an exception handler. This *finish* will then block until its argument term has finished evaluation, ensuring that the parallel task is done before continuing. The permission for the box x, temporarily consumed by *async*, might still be available after *finish* is evaluated. This makes it possible to access the produced result of t.

This assumes of course that box x was created outside of the *finish*

Note that LaCasa provided an expression called $x.open(y \Rightarrow t\}$, used for evaluating a term t on y, the contents of box x. In **PLC**, this is just syntactic sugar for an async wrapped in a finish, i.e. *"finish*$\{async(x, y \Rightarrow t)\{\text{let } z = \texttt{null} \text{ in } z\}\}$*"*. Note that the continuation of *async* is irrelevant here as no value is returned by async. let $z = \texttt{null}$ in z is simply the shortest possible continuation in ANF.

4.2 Dynamic Semantics

We formalize the dynamic semantics through three small-step reduction relations. The simplest, $H, F \rightarrow H', F'$, reduces a single stack frame F and heap H and is used for standard rules that only modify the current frame. $H, FS \twoheadrightarrow H', FS'$ reduces a complete frame stack FS. Lastly, $H, TS, WS \rightsquigarrow H', TS', WS'$ reduces a set of active tasks TS and a set of waiting tasks WS together with heap H to H', TS', WS'.

A heap H represents a mapping of references o to objects $\langle C, FM \rangle$, where C is a class type and FM a mapping of fields f to their values. Fields may only contain other heap values or null, i.e. the domain of FM is $dom(H) \cup \{\text{null}\}$.

A box $x = b(o, p)$ encapsulates its content and prevents ordinary access to o by requiring special expressions to supply a matching permission p. For example $async(x, y \Rightarrow t)\{s\}$ makes the content o available as y in a limited scope t in a separate task while continuing to evaluate s in a scope where p is no longer available. $capture(x.f, y)\{z \Rightarrow t\}$ stores the content o' of another box $y = b(o', p')$ in the field f of box $b(o, p)$, that is $o.f$. It then evaluates t in an environment without the permission p'. Permissions can be temporarily or permanently consumed, depending on the expression, to prevent concurrent access to the same box or to permanently destroy a box.

4.2.1 Single Frame Reduction

A frame $F = \langle L,\ t,\ P \rangle^{l}$ consists of a variable environment L, a term t, a set of permissions P and a return annotation l. In detail:

- L is a mapping of variables to heap values or boxes, i.e. $L(x) \in$

$$\begin{array}{c} H, \langle L,\ \text{let } x = \text{null in } t,\ P\rangle^l \\ \to H, \langle L[x \to \text{null}],\ t,\ P\rangle^l \end{array} \qquad \text{(E-NULL)}$$

$$\frac{H(L(y)) = \langle C, FM\rangle \qquad f \in dom(FM)}{\begin{array}{c} H, \langle L,\ \text{let } x = y.f \text{ in } t,\ P\rangle^l \\ \to H, \langle L[x \to FM(f)],\ t,\ P\rangle^l \end{array}} \qquad \text{(E-SELECT)}$$

$$\begin{array}{c} H, \langle L,\ \text{let } x = y \text{ in } t,\ P\rangle^l \\ \to H, \langle L[x \to L(y)],\ t,\ P\rangle^l \end{array} \qquad \text{(E-VAR)}$$

$$\frac{\begin{array}{c} L(y) = o \qquad H(o) = \langle C, FM\rangle \\ H' = H[o \to \langle C, FM[f \to L(z)]\rangle] \end{array}}{\begin{array}{c} H, \langle L,\ \text{let } x = y.f = z \text{ in } t,\ P\rangle^l \\ \to H', \langle L,\ \text{let } x = z \text{ in } t,\ P\rangle^l \end{array}} \qquad \text{(E-ASSIGN)}$$

$$\frac{\begin{array}{c} o \notin dom(H) \qquad fields(C) = \overline{f} \\ H' = H[o \to \langle C, \overline{f \to \text{null}}\rangle] \end{array}}{\begin{array}{c} H, \langle L,\ \text{let } x = \text{new } C \text{ in } t,\ P\rangle^l \\ \to H', \langle L[x \to o],\ t,\ P\rangle^l \end{array}} \qquad \text{(E-NEW)}$$

Figure 4.3: Single frame evaluation rules

$$\frac{}{\begin{array}{l} H, \langle L,\ x,\ P\rangle^{y} \circ \langle L',\ t',\ P'\rangle^{l} \circ FS \\ \twoheadrightarrow H, \langle L'[y \to L(x)],\ t',\ P'\rangle^{l} \circ FS \end{array}} \quad \text{(E-RETURN1)}$$

$$\frac{}{\begin{array}{l} H, \langle L,\ x,\ P\rangle^{\epsilon} \circ \langle L',\ t',\ P'\rangle^{l} \circ FS \\ \twoheadrightarrow H, \langle L',\ t',\ P'\rangle^{l} \circ FS \end{array}} \quad \text{(E-RETURN2)}$$

$$\frac{H, F \to H', F'}{H, F \circ FS \twoheadrightarrow H', F' \circ FS} \quad \text{(E-FRAME)}$$

Figure 4.4: Frame stack evaluation rules

$dom(H) \cup \{\text{null}\} \cup \{b(o, p) \mid o \in dom(H)\}$. We write an updated mapping that maps x to v and $y \neq x$ to $L(y)$ as $L[x \to v]$.

- t is the term that is being reduced by the evaluation rules.
- P is a set of permissions available within this frame. A permission p represents the ability to access a box $b(o, p)$.
- The return annotation l is used when returning to another frame $G = \langle L',\ s,\ P'\rangle^{m}$ to indicate in which variable of G, if any, the return value of F should be stored. l is either a variable or ϵ to indicate the lack of a return value.

Figure 4.3 shows the single frame evaluation rules. As they are identical to the ones presented in LaCasa and standard, we will not discuss them further.

4.2.2 Frame Stack Reduction

A frame stack FS is a stack of frames representing a call stack. We write $FS = F \circ FS'$ to indicate that FS consists of a top-most frame F and a remaining stack FS'. The empty stack is written as $FS = \epsilon$

As in LaCasa, $\mathit{fields}(C)$ is the set of fields of class C. Similarly, $mbody(C, m)$ is the body $x \to t$ of method $m(x : \sigma) = t : \rho$ of class C.

Figure 4.4 shows the evaluation rules for frame stacks. E-RETURN1 removes the top-most stack frame by assigning its result to the variable y in the second top-most frame as described earlier. E-RETURN2 discards the value instead. E-FRAME applies the frame evaluation

$$\frac{\begin{array}{c} T_1 = (i_f, j_a, \langle [x \to o],\ t,\ \emptyset\rangle^{\epsilon}) \qquad L(b) = b(o,p) \qquad j_a \mathit{fresh} \\ T_2 = (i_f, i_a, \langle L,\ s,\ P \setminus \{p\}\rangle^{\epsilon}) \qquad p \in P \end{array}}{\begin{array}{l} H, \{(i_f, i_a, \langle L,\ \mathit{async}(b, x \Rightarrow t)\{s\},\ P\rangle^{l} \circ FS)\} \uplus TS, WS \\ \rightsquigarrow H, \{T_1, T_2\} \uplus TS, WS \end{array}} \quad (\text{E-ASYNC})$$

$$\frac{\begin{array}{c} T_1 = (j_f, i_a, \langle L,\ t,\ P\rangle^{\epsilon}) \qquad j_f \mathit{fresh} \\ T_2 = (i_f, i_a, \langle L,\ \text{let}\ \ x = \texttt{null}\ \text{in}\ s,\ P\rangle^{l} \circ FS) \end{array}}{\begin{array}{l} H, \{(i_f, i_a, \langle L,\ \text{let}\ \ x = \mathit{finish}\{\ t\ \}\ \ \text{in}\ \ s,\ P\rangle^{l} \circ FS)\} \uplus TS, WS \\ \rightsquigarrow H, \{T_1\} \uplus TS, \{(T_2, j_f)\} \uplus WS \end{array}} \quad (\text{E-FINISH})$$

$$\frac{\nexists (i_f, i_a, FS) \in TS \cup \mathit{tasks}(WS)}{H, TS, \{(T, i_f)\} \uplus WS \rightsquigarrow H, \{T\} \uplus TS, WS} \quad (\text{E-RESUME})$$

$$\frac{FS = \langle L,\ x,\ P\rangle^{l} \circ \epsilon \lor FS = \epsilon}{H, \{(i_f, i_a, FS)\} \uplus TS, WS \rightsquigarrow H, TS, WS} \quad (\text{E-TASK-DONE})$$

$$\frac{H, FS \twoheadrightarrow H', FS'}{H, \{(i_f, i_a, FS)\} \uplus TS, WS \rightsquigarrow H', \{(i_f, i_a, FS')\} \uplus TS, WS} \quad (\text{E-STACK})$$

Figure 4.5: Task set evaluation rules introduced by **PLC**

rules to the top frame of a frame stack, thereby connecting the two reduction relations.

4.2.3 Task Set Reduction

Tasks represent an isolated computation on a separate thread. A task $T = (i_f, i_a, FS)$ consists of a frame stack FS being evaluated, a finish id i_f to track from which *finish* a task originated from and an async id i_a to track which tasks share permissions with T. Each *finish* introduces a fresh i_f different from all previous finish ids. An asynchronous task spawned by an *async* has a fresh id $j_a \neq i_a$ as it does not inherit the context or permissions from its parent. The continuation term of an *async* shares the id with its parent as it inherits the context and with it the permissions.

The tasks are grouped into two sets TS and WS. TS contains active tasks T while WS contains waiting tasks together with a finish id. A

$$\frac{\begin{array}{c} H(L(y)) = \langle C, FM \rangle \qquad mbody(C, m) = x \to t' \\ L' = \begin{cases} L_0[\text{this} \to L(y), x \to L(z)] & \text{if } i_a = i_{a0} \\ [\text{this} \to L(y), x \to L(z)] & \text{otherwise} \end{cases} \end{array}}{\begin{array}{l} H, \{(i_f, i_a, \langle L,\ \text{let } x = y.m(z) \text{ in } t,\ P\rangle^l \circ FS)\} \uplus TS, WS \\ \rightsquigarrow H, \{(i_f, i_a, \langle L',\ t',\ \text{P} \rangle^x \circ \langle L,\ t,\ P\rangle^l \circ FS)\} \uplus TS, WS \end{array}}$$

(E-INVOKE)

$$\frac{\begin{array}{c} o \notin dom(H) \qquad fields(C) = \overline{f} \\ H' = H[o \to \langle C, \overline{f \to \texttt{null}} \rangle] \qquad p \ fresh \\ WS' = dropAncestorsFrames(WS, i_a) \end{array}}{\begin{array}{l} H, \{(i_f, i_a, \langle L,\ box[C]\{x \Rightarrow t\},\ P\rangle^l \circ FS)\} \uplus TS, WS \\ \rightsquigarrow H', \{(i_f, i_a, \langle L[x \to b(o, p)],\ t,\ P \cup \{p\}\rangle^l \circ \epsilon)\} \uplus TS, WS' \end{array}} \text{(E-BOX)}$$

$$\frac{\begin{array}{c} L(x) = b(o, p) \qquad L(y) = b(o', p') \qquad \{p, p'\} \subseteq P \\ H(o) = \langle C, FM \rangle \qquad H' = H[o \to \langle C, FM[f \to o'] \rangle] \\ WS' = dropAncestorsFrames(WS, i_a) \end{array}}{\begin{array}{l} H, \{(i_f, i_a, \langle L,\ capture(x.f, y)\{z \Rightarrow t\},\ P\rangle^l \circ FS)\} \uplus TS, WS \\ \rightsquigarrow H', \{(i_f, i_a, \langle L[z \to L(x)],\ t,\ P \setminus \{p'\}\rangle^\epsilon \circ \epsilon)\} \uplus TS, WS' \end{array}}$$

(E-CAPTURE)

$$\frac{\begin{array}{c} L(x) = b(o, p) \qquad L(y) = b(o', p') \qquad \{p, p'\} \subseteq P \\ H(o) = \langle C, FM \rangle \qquad FM(f) = o'' \qquad p'' \ fresh \\ H' = H[o \to \langle C, FM[f \to o'] \rangle] \\ WS' = dropAncestorsFrames(WS, i_a) \end{array}}{\begin{array}{l} H, \{(i_f, i_a, \langle L,\ swap(x.f, y)\{z \Rightarrow t\},\ P\rangle^l \circ FS)\} \uplus TS, WS \\ \rightsquigarrow H', \{(i_f, i_a, \langle L[z \to b(o'', p'')],\ t,\ (P \setminus \{p'\}) \cup \{p''\}\rangle^\epsilon \circ \epsilon)\} \uplus TS, WS' \end{array}}$$

(E-SWAP)

Figure 4.6: Task set evaluation rules adapted from LaCasa

pair $(U, i_f) \in WS$ indicates that task U can only be resumed, that is, moved to TS, once all tasks with id i_f are finished. Note that finish ids are not unique to tasks; a single task might spawn multiple tasks with finish id i_f which are awaited by a single finish.

Figure 4.5 shows the evaluation rules for task sets introduced in **PLC**. **PLC** adds four additional rules to create tasks, await their completion, resume them and complete their reduction. E-ASYNC reduces one task by replacing it by two new tasks T_1, T_2. The first one evaluates the function $x \Rightarrow t$ using only the content of box b without any additional variables or permissions. The second task evaluates the continuation s and inherits the variables and permissions (with the exception of box permission p) from the original task. As T_1 does not inherit any permissions from the parent, it is assigned a fresh async id j_a; every permission consumed within T_1 originated in T_1 and is thus not present in other tasks. T_2, on the other hand, does inherit permissions and thus must remove all tasks potentially holding the same permission when consuming one. It inherits the async id i_a.

Both tasks start active and may be immediately reduced. The waiting set WS is not modified by E-ASYNC. Both tasks also inherit the same finish id i_f from the original task. This ensures that the immediately enclosing finish awaits the completion of both tasks. Note that this is the core difference between the async-finish model and the fork-join model as joins only await one specific task.

E-ASYNC replaces T by two new tasks, essentially skipping the frames created between the *async* and the most recent *finish*. This is a form of non-local control flow, similar to an exception skipping to a handler. Async skips to the most recent finish, as the permission it consumes only has to be unavailable as long as the asynchronously started task is running. After this parent task resumes, this is no longer the case.

E-FINISH creates a new task T_1 with a new id j_f evaluating the term t and a suspended task T_2 waiting on the completion of all j_f-tagged tasks. This way, any async within t inherits this id j_f and T_2 will only be resumed once T_1 and all its spawned tasks are done. As T_1 shares P with T_2, it has to remove T_2 upon permission consumption; thus i_a is inherited.

E-RESUME resumes a waiting task blocking on id i_f. For this it checks that no task with this id is currently being executed or itself waiting on another task. This ensures no descendant is alive.

E-TASK-DONE completes the execution of a task. Once a task has been fully evaluated to an empty stack or its term is an irreducible variable, it can simply be dropped. Tasks cannot directly return values, instead they must use the box passed as argument to store the result for later retrieval. Note that the only irreducible term in a wellformed program of **PLC** is a variable.

E-STACK reduces the frame stack of one task according to the frame stack evaluation rules, thereby embedding that reduction relation in the task set reduction relation.

PLC also modifies E-INVOKE, E-BOX, E-CAPTURE and E-SWAP as their semantics must be adapted to the presence of tasks. The adapted rules are shown in Fig. 4.6.

E-INVOKE calls method m of an object y with argument z, creating a new frame within which the methods body is evaluated. It differs from its LaCasa equivalent in two ways: Instead of passing only the necessary subset of permissions, it now copies the permissions from the previous frame. This avoids a case distinction on whether the argument is a box or not. It is sound because all permissions valid in the caller are also valid in the callee, the relevant variables are simply not available making the permissions superfluous.

The other difference is that E-INVOKE in LaCasa always created the new frame with the variable environment L_0, which includes the global variables. This allows *ocap* methods to access global variables. This is nevertheless sound in LaCasa as *ocap* methods are type-checked without global variables, ensuring that they do not access them at runtime. We instead only pass L_0 if m is allowed to access them; this will later become necessary for the isolation proof which requires that certain tasks share no variables.

E-BOX creates a new box of class C and continues execution in a continuation t with the newly created box $b(o, p)$ being available in variable x. In theory, it would be sound to not drop any frames. But encoding permissions in Scala is done using implicits which cannot be introduced by a variable binding, preventing `let` $x = box[C]$ `in` t as a construct, and instead requiring the creation of a new scope via the continuation t. We decide to retain this limitation.

E-CAPTURE stores the content of box y in the field $x.f$ and is analogous to assignment. To do so, it requires permission for both x, which is modified, and y, which is unwrapped, permanently consuming the permission for box y in the process.

E-SWAP similarly takes a box y and a unique field $x.f$. It stores y in $x.f$, returning the previous content of $x.f$ in a new box z. While E-CAPTURE and E-SWAP are similar in evaluation, E-SWAP acts only on unique fields. As unique fields cannot be accessed or modified by any expression besides *swap*, the content of the unique field will stay isolated and must be retrieved via another *swap* before it can be interacted with.

box, *capture* and *swap* are continuation terms and do not return, which is equivalent to dropping all frames below the current one, i.e. FS in $(i_f, i_a, F \circ FS)$. This is done to ensure that the consumed permissions are permanently unavailable. But this is not sufficient in the presence of tasks: if a task inherited permissions from its parent, it must also drop this parent and in general all ancestors that may have the same permission. This is indicated by the async id i_a as the parent of a task has exactly then the same async id as its child when it handed down permissions to its child. This yields the set WS'.

Another complication prevents us from simply removing WS' from WS: A task not only stores a frame stack but encodes also the relationship between tasks. We illustrate the problem with an example: Assume that task T spawned an asynchronous task U. U then calls *finish* followed by *async*, creating a task U' with $id_f(T) \neq id_f(U')$ and another asynchronous task V. If U' now calls *capture*, it erases the frame stacks of U and U'. Because U' remains, even with an empty frame stack, T cannot be resumed before U is resumed which in turn cannot be resumed before V is resumed. Preserving this relation is important because V might not be completely isolated from T, for example if T created U with box x and U created V with box $y = x.f$.

To satisfy these constraints, we define a function *dropAncestors-Frames* that replaces the frame stacks of all ancestors of an id i_a with the empty frame stack FS:

$$dropAncestorsFrames(WS, i_a) = (WS \setminus WS') \cup \{((j_f, j_a, \epsilon), k_f) \mid ((j_f, j_a, FS'), k_f) \in WS' \wedge j_a = i_a\}$$

$$\text{where } WS' = \{(T, j_f) \in WS \mid id_a(T) = i_a\}$$

4.2.4 Initial state

For a program $p = \overline{cd}\ \overline{cfd}\ t$ the initial state is defined as $H_0, \{(i_{f0}, i_{a0}, \langle L_0,\ t,\ \emptyset\rangle^{\epsilon})\}, \emptyset$.

Global variables are encoded through a special global object o_g with a field for every variable and the single element of the initial heap: $H_0 = \{o_g \rightarrow \langle C_g, FM_g \rangle\}$ where $FM_g = \{x \rightarrow \text{null} \mid \text{var } x : C \in \overline{cfd}\}$ and class C_g extends `AnyRef`$\{\overline{cfd}\}$.

$L_0 = [global \rightarrow o_g]$ describes the initial variable environment and is used both for the main term and in E-INVOKE.

As the initial task is not awaited by any other task, the value of i_{f0} is arbitrary as long as it is fixed. The actual value of i_{a0} is similarly arbitrary as it only influences E-INVOKE which checks for equality.

4.3 Static Semantics

4.3.1 Type Checking

A statement of the form $\Gamma; a \vdash t : \tau$ states that term t is assigned type τ in type environment Γ and effect a. a might either be ϵ, representing no effect, or *ocap* which requires that all classes C in t satisfy $ocap(C)$ which later will be described in more detail.

4.3.2 Well-Formed Programs

Figure 4.7 shows the rules for well-formed programs. A program p is well-formed under WF-PROGRAM if the class definitions $\overline{cd}$ are well-typed and the main term t under the type environment $\Gamma_0 = \{\text{global} : C_g\}$ with C_g being the global object encompassing all global variables: "`class` C_g `extends AnyRef`$\{\overline{cfd}\}$".

WF-CLASS describes classes C that extend `AnyRef` or another well-formed class D. The methods of the class must be well-formed. In addition, all methods must have well-formed overrides as defined by WF-OVERRIDE. Fields defined within C must not be defined in D as overriding fields is not possible.

WF-OVERRIDE states that a method m defined in C is a valid override if it either is not defined in D or has the same type in both classes.

WF-METHOD1 states that a method m is well-formed if its body is well-typed under an environment consisting of only globals, the class containing the method and its argument. The type of the term must furthermore be a subtype of the stated return type. WF-METHOD2 differs from WF-METHOD1 in the method argument being a box. The

$$\frac{\begin{array}{c} C \vdash \overline{md} \qquad D = \texttt{AnyRef} \vee p \vdash \text{class } D \ldots \\ \forall(\text{def } m \ldots) \in \overline{md}.\ \mathit{override}(m, C, D) \\ \forall \text{var } f : \sigma \in \overline{fd}.\ f \notin \mathit{fields}(D) \end{array}}{p \vdash \text{class } C \text{ extends } D\ \{\overline{fd}\ \overline{md}\}} \text{ (WF-CLASS)}$$

$$\frac{\mathit{mtype}(m, D) \text{ not defined} \vee \mathit{mtype}(m, D) = \mathit{mtype}(m, C)}{\mathit{override}(m, C, D)} \text{ (WF-OVERRIDE)}$$

$$\frac{\Gamma_0, \mathit{this} : C, x : D; \epsilon \vdash t : E' \qquad E' <: E}{C \vdash \text{def } m(x : D) : E = t} \text{ (WF-METHOD1)}$$

$$\frac{\Gamma = \Gamma_0, \mathit{this} : C, x : Q \rhd \mathit{Box}[D], \mathit{Perm}[Q] \qquad Q \ \mathit{fresh} \qquad \Gamma; \epsilon \vdash t : E' \qquad E' <: E}{C \vdash \text{def } m(x : \texttt{Box}[D]) : E = t} \text{ (WF-METHOD2)}$$

$$\frac{p \vdash \overline{cd} \qquad p \vdash \Gamma_0 \qquad \Gamma_0; \epsilon \vdash t : \sigma}{p \vdash \overline{cd}\ \overline{vd}\ t} \text{ (WF-PROGRAM)}$$

Figure 4.7: Rules for well-formed programs

argument x is annotated by the programmer with type $\texttt{Box}[D]$. WF-METHOD2 introduces a guarded type $Q \rhd \texttt{Box}[D]$ based on this, in addition to a fresh permission $\texttt{Perm}[Q]$ corresponding to the guarded box. As stated earlier, guarded types do not appear in the surface syntax and are only introduced during type-checking, specifically this rule.

4.3.3 Guarded Types and Permissions

To help ensure that no expressions break the isolation of boxes we use guarded types. A guarded type $Q \rhd \texttt{Box}[\texttt{C}]$ consists of a Q, an abstract type taken from a countably infinite supply, and the to-be-guarded box. In addition, we have permission types $\texttt{Perm}[Q]$ that give permission to access the guarded box. All operations on guarded types require that this associated permission type $\texttt{Perm}[Q]$ is available in the environment Γ.

We will later show that each permission type $\texttt{Perm}[Q]$ corresponds to one permission p in the evaluation semantics. We will also show that the corresponding permission is always available at evaluation if

$$\frac{}{ocap(\texttt{AnyRef})} \quad \text{(OCAP-ANYREF)}$$

$$\frac{\begin{array}{c} p \vdash \text{class } C \text{ extends } D \ \{\overline{fd}\ \overline{md}\} \\ C \vdash_{ocap} \overline{md} \qquad ocap(D) \\ \forall \text{var } f : \sigma \in \overline{fd}.\ ocap(\sigma) \lor \sigma = \text{Box}[E] \land ocap(E) \end{array}}{ocap(C)} \quad \text{(OCAP-CLASS)}$$

$$\frac{\begin{array}{c} \Gamma = \text{this} : C, x : D \\ \Gamma; \text{ocap} \vdash t : E' \qquad E' <: E \end{array}}{C \vdash_{ocap} \text{def } m(x : D) : E = t} \quad \text{(OCAP-METHOD1)}$$

$$\frac{\begin{array}{c} \Gamma = \text{this} : C, x : Q \rhd \text{Box}[D], \text{Perm}[Q] \qquad Q \textit{ fresh} \\ \Gamma; \text{ocap} \vdash t : E' \qquad E' <: E \end{array}}{C \vdash_{ocap} \text{def } m(x : \text{Box}[D]) : E = t} \quad \text{(OCAP-METHOD2)}$$

Figure 4.8: Object capability rules

the term is well-typed, thereby showing that the permissions in the evaluation semantics need not actually be tracked. They are thus only an aid for the proofs.

4.3.4 Object Capabilities

The rules in Fig. 4.8 define how the object capabilities constraint affects class and method definitions. They are quite similar to their corresponding well-formedness rules. OCAP-ANYREF states that the class containing no methods and fields obeys the constraints, i.e. `AnyRef`, is *ocap*. OCAP-CLASS requires that the parent class D is *ocap* as well as all methods. In addition, fields may only contain *ocap* objects or boxes. OCAP-METHOD1 and OCAP-METHOD2 differ only slightly from WF-METHOD1 respective 2. They remove the global variables Γ_0 from the type environment in which t is checked, in addition to using effect *ocap* instead of ϵ. The first change prevents t from accessing global variables directly whereas checking under effect *ocap* is later used in the typing rules to ensure that t does not call non-ocap methods, thereby gaining indirect access to the global variables.

$$\frac{}{\Gamma; a \vdash \text{null} : \texttt{Null}} \quad \text{(T-NULL)}$$

$$\frac{x \in dom(\Gamma)}{\Gamma; a \vdash x : \Gamma(x)} \quad \text{(T-VAR)}$$

$$\frac{\Gamma; a \vdash e : \tau \qquad \Gamma, x : \tau; a \vdash t : \sigma}{\Gamma; a \vdash \text{let } x = e \text{ in } t : \sigma} \quad \text{(T-LET)}$$

$$\frac{\Gamma; a \vdash x : C \qquad \mathit{ftype}(C, f) = D}{\Gamma; a \vdash x.f : D} \quad \text{(T-SELECT)}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : C \qquad \mathit{ftype}(C, f) = D \\ \Gamma; a \vdash y : D' \qquad D' <: D\end{array}}{\Gamma; a \vdash x.f = y : D} \quad \text{(T-ASSIGN)}$$

Figure 4.9: Standard term typing rules

4.3.5 Term and Expression Typing

Figure 4.9 shows the typing rules for standard terms. As they are not further interesting, we do not describe them in detail. Instead, we focus on Fig. 4.10 which shows the typing rules related to the changes in **PLC**.

T-NEW requires the object to contain only class typed fields. If it is typed under effect *ocap* it requires the instantiated object C to satisfy $\mathit{ocap}(C)$.

T-INVOKE calls a method, passing it an argument y. If this argument has a guarded type $Q \triangleright \texttt{Box}[D]$, T-INVOKE checks that the method takes a box with matching class as argument and that permission $\texttt{Perm}[Q]$ corresponding to the guarded type is available in the callers environment.

T-BOX creates a new box, requiring that its content is *ocap* (as it would otherwise defeat the purpose of an isolated box). It also creates a new permission and checks the continuation term with both box and permission in the environment. As it changes the set of available permissions, it has to create a new environment using the continuation term t. Note that it returns bottom for the sake of simplicity, all permissions in Γ are still valid and could be accessed without soundness issues.

$$\frac{a = ocap \Longrightarrow ocap(C) \qquad \forall\, var\ \ f : \sigma \in \overline{fd}.\ \exists D.\ \sigma = D}{\Gamma; a \vdash \text{new } C : C} \text{ (T-NEW)}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : C \qquad mtype(C, m) = \sigma \rightarrow \tau \qquad \Gamma; a \vdash y : \sigma' \\ \sigma' <: \sigma\ \vee (\sigma = Box[D] \wedge \sigma' = Q \rhd Box[E] \wedge Perm[Q] \in \Gamma \wedge E <: D)\end{array}}{\Gamma; a \vdash x.m(y) : \tau} \text{ (T-INVOKE)}$$

$$\frac{ocap(C) \qquad Q\ fresh \qquad \Gamma, x : Q \rhd Box[C], Perm[Q]; a \vdash t : \sigma}{\Gamma; a \vdash box[C]\{x \Rightarrow t\} : \bot} \text{ (T-BOX)}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : Q \rhd Box[C] \qquad \Gamma; a \vdash y : Q' \rhd Box[D] \\ \{Perm[Q], Perm[Q']\} \subseteq \Gamma \qquad D <: ftype(C, f) \\ \Gamma \setminus \{Perm[Q']\}, z : Q \rhd Box[C]; a \vdash t : \sigma\end{array}}{\Gamma; a \vdash capture(x.f, y)\{z \Rightarrow t\} : \bot} \text{ (T-CAPTURE)}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : Q \rhd Box[C] \qquad \Gamma; a \vdash y : Q' \rhd Box[D'] \\ \{Perm[Q], Perm[Q']\} \subseteq \Gamma \qquad ftype(C, f) = Box[D] \\ D' <: D \qquad R\ fresh \\ \Gamma \setminus \{Perm[Q']\}, z : R \rhd Box[D], Perm[R]; a \vdash t : \sigma\end{array}}{\Gamma; a \vdash swap(x.f, y)\{z \Rightarrow t\} : \bot} \text{ (T-SWAP)}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : Q \rhd Box[C] \qquad Perm[Q] \in \Gamma \\ y : C; ocap \vdash t : \tau \qquad \Gamma \setminus Perm[Q]; a \vdash s : \sigma\end{array}}{\Gamma; a \vdash async(x, y \Rightarrow t)\{s\} : \bot} \text{ (T-ASYNC)}$$

$$\frac{\Gamma; a \vdash t : \tau}{\Gamma; a \vdash finish\{t\} : null} \text{ (T-FINISH)}$$

Figure 4.10: Non-standard term typing rules

T-CAPTURE checks that both arguments are boxes, requires that both their permissions are available and that the target field $x.f$ has the correct type to contain the content of box y. The target field must be class field and cannot be a box. It also checks the continuation term in an environment that does not contain the permission for y is it was captured by $x.f$.

T-SWAP similarly checks both arguments, their permissions and the target field. In contrast to T-CAPTURE, the target field must be a unique field. A new permission is also created as the previous content of the field is made available in the continuation term.

T-ASYNC checks that the first argument is a box with available permission and that the term t is well-typed in an *ocap* environment containing only the contents of that box. We require *ocap* as all parallel tasks must be isolated from each other. It also checks the continuation term without the permission of the box. The continuation term is checked under the same effect as *async* itself.

T-FINISH requires only that the term is valid in the same environment. Note that it returns type `null` instead of τ, the type of the enclosed term. Due to *async* returning bottom and jumping directly to the immediately enclosing finish, similar to an exception jumping to its handler, type checking becomes significantly more difficult.

4.3.6 Subtyping

Figure 4.11 shows the subtyping rules used in **PLC**. They are based on Featherweight Java with <:-ID, <:-EXT and <:-TRANS being directly based on it and <:-BOT, <:-BOX and <:-NULL being introduced in LaCasa. The **PLC** extension does not add any additional subtyping rules and is unaffected by subtyping besides subtypes of boxes. We include them as they are a standard feature of object oriented languages and were included in CLC^2.

4.3.7 Well-Formedness

Figure 4.12 show the rules for wellformed variable environments and permissions. WF-VAR requires that a variable is either `null`, an object for which the actual value on the heap is a subtype of the type assigned by Γ. Lastly, the variable can be a box with a guarded type in the environment Γ.

$$\frac{}{C <: C} \quad \text{(<:-ID)}$$

$$\frac{\text{class } C \text{ extends } D \text{ \{...\}}}{C <: D} \quad \text{(<:-EXT)}$$

$$\frac{C <: D \qquad D <: E}{C <: E} \quad \text{(<:-TRANS)}$$

$$\frac{}{\perp <: \tau} \quad \text{(<:-BOT)}$$

$$\frac{C <: D}{\texttt{Box}[C] <: \texttt{Box}[D]} \quad \text{(<:-BOX)}$$

$$\frac{}{\texttt{Null} <: \tau} \quad \text{(<:-NULL)}$$

Figure 4.11: Subtyping rules

$$\frac{\begin{array}{c} L(x) = \mathit{null} \ \vee \\ L(x) = o \wedge \mathit{typeof}(H, o) <: \Gamma(x) \vee \\ L(x) = b(o, p) \wedge \Gamma(x) = Q \rhd \mathit{Box}[C] \wedge \mathit{typeof}(H, o) <: C \end{array}}{H \vdash \Gamma; L; x} \quad \text{(WF-VAR)}$$

$$\frac{\begin{array}{c} \gamma : \mathit{permTypes}(\Gamma) \longrightarrow P \ \ \mathit{injective} \\ \forall x \in \mathit{dom}(\Gamma). \\ \Gamma(x) = Q \rhd \mathit{Box}[C] \wedge L(x) = b(o, p) \wedge \texttt{Perm}[Q] \in \Gamma \Longrightarrow \gamma(Q) = p \end{array}}{\vdash \Gamma; L; P} \quad \text{(WF-PERM)}$$

$$\frac{\begin{array}{c} \mathit{dom}(\Gamma) \subseteq \mathit{dom}(L) \\ \forall x \in \mathit{dom}(\Gamma).\ H \vdash \Gamma; L; x \end{array}}{H \vdash \Gamma; L} \quad \text{(WF-ENV)}$$

Figure 4.12: Well-formedness rules

$$\frac{}{H \vdash \epsilon} \quad \text{(T-EMPFS)}$$

$$\frac{\begin{array}{cc}\Gamma; a \vdash t : \sigma & l \neq \epsilon \Longrightarrow \sigma <: C \\ H \vdash \Gamma; L & \vdash \Gamma; L; P\end{array}}{H \vdash \langle L,\ t,\ P\rangle^{l} : \sigma} \quad \text{(T-FRAME1)}$$

$$\frac{\begin{array}{cc}\Gamma, x : \tau; a \vdash t : \sigma & l \neq \epsilon \Longrightarrow \sigma <: C \\ H \vdash \Gamma; L & \vdash \Gamma; L; P\end{array}}{H \vdash^{\tau}_{x} \langle L,\ t,\ P\rangle^{l} : \sigma} \quad \text{(T-FRAME2)}$$

$$\frac{H \vdash F^{\epsilon} : \sigma \qquad H \vdash FS}{H \vdash F^{\epsilon} \circ FS} \quad \text{(T-FS-NA)}$$

$$\frac{H \vdash^{\tau}_{x} F^{\epsilon} : \sigma \qquad H \vdash FS}{H \vdash^{\tau}_{x} F^{\epsilon} \circ FS} \quad \text{(T-FS-NA2)}$$

$$\frac{H \vdash F^{y} : \sigma \qquad H \vdash^{\sigma}_{y} FS}{H \vdash F^{y} \circ FS} \quad \text{(T-FS-A)}$$

$$\frac{H \vdash^{\tau}_{x} F^{y} : \sigma \qquad H \vdash^{\sigma}_{y} FS}{H \vdash^{\tau}_{x} F^{y} \circ FS} \quad \text{(T-FS-A2)}$$

$$\frac{\begin{array}{c}\forall (i_f, i_a, FS) \in TS\,.\ H \vdash FS \\ \forall ((i_f, d_a, FS), j_f) \in WS\,.\ H \vdash FS\end{array}}{H \vdash TS, WS} \quad \text{(T-TS)}$$

Figure 4.13: Frame typing rules

We obtain the type of an object as follows:

Definition 4.3.1 (Object Type)**.** For an object identifier $o \in dom(H)$ where $H(o) = \langle C, FM\rangle$, $typeof(H, o) := C$

WF-PERM defines a mapping γ of static permission types to runtime permissions. It requires firstly that the permission types present in Γ, defined by $permTypes(\Gamma) = \{Q \mid \texttt{Perm}[Q] \in \Gamma\}$, only map to at most one permission in P each. Furthermore, for each box x defined in Γ with available permission $\texttt{Perm}[Q]$, we map the permission type Q to the runtime permission p that guards the box, i.e. $L(x) = b(o, p)$.

Figure 4.13 shows the rules for wellformed frames, framestacks and task sets. T-FS-NA checks a frame that does not return a value in addition to recursively checking the rest of the frame stack. If a frame does return a value, i.e. $l \neq \epsilon$, we check the frame below it in the stack with the variable in which it will be stored and the type of the expected argument. This shown by T-FS-A that takes a frame with label $l = y$ and type σ and checks the remaining frame stack with the annotation $\vdash_y^\sigma$. T-FS-NA2 and T-FS-A2 handle the remaining two combinations of argument and no argument.

T-FRAME1 states that a frame $\langle L, t, P \rangle^l$ is wellformed if its term t is welltyped. Furthermore, if the return label l is a variable, the return type must be a class type as we do not allow e.g. boxes to be returned. Finally, the variable environment and the permissions must be wellformed. T-FRAME2 is similar but adds the return value of expected type τ in variable x to the environment Γ when checking t.

Finally, T-EMPFS handles the empty frame stack and T-TS extends the wellformedness rules to task sets.

Chapter 5

Properties

5.1 Progress

The progress theorem states that we can always take a step as long as we are in a well-formed state or that we have reached a final state. A consequence of progress is that there are no deadlocks as a *finish* never blocks forever. One exception to progress are *null* values as calling the method or assigning to fields of a null object is not well-defined. We define the final state as both the active and waiting task sets TS and WS being empty. We can thus state Progress follows:

Theorem ?? (Progress)**.**

$$\vdash H : \star \tag{5.1}$$
$$\wedge \;\; H \vdash TS, WS \tag{5.2}$$
$$\wedge \;\; H \vdash TS, WS \,\mathbf{ok} \tag{5.3}$$
$$\Rightarrow H, TS, WS \rightsquigarrow H', TS', WS' \tag{5.4}$$
$$\vee \;\; TS = \emptyset \wedge WS = \emptyset \tag{5.5}$$
$$\vee \;\; \exists (i_f, i_a, FS) \in TS.\; FS = F \circ FS' \text{where } F = \langle L, \text{let } x = t \text{ in } t', P\rangle^l \text{ and} \tag{5.6}$$

$t \in \{y.f, y.f = z, y.m(z), y.async(b, z \to s)\{t''\},$
$capture(y.f, x)\{z \Rightarrow t''\}, swap(y.f, x)\{z \Rightarrow t''\}\}$ and $L(y) = \texttt{null}$.

To prove this, we first must have a task in TS that can be reduced or that we can resume a waiting task to obtain one. To resume a waiting task, we must show that, assuming TS is empty, at least one task in

WS can be resumed. This is equivalent to stating that at least one task in WS has no task it waits on. We call this property deadlock freedom as it states that there cannot be a cycle of tasks waiting on each other.

To show this, we rely on the *idOrdering* invariant which states that a waiting task always waits on tasks created after it or equivalently: $i_f < j_f$ for a waiting task $((i_f, i_a, FS), j_f) \in WS$.

Definition 5.1.1 (ID Ordering)**.** We say that a set of waiting tasks WS has ordered IDs iff all tasks await IDs created after their own ID, that is:

$$idOrdering(WS) := \forall((i_f, i_a, FS), j_f) \in WS.\ i_f < j_f$$

Lemma 1 (Deadlock Freedom)**.**

$$\begin{aligned}
& H \vdash TS, WS\ \boldsymbol{ok} && (5.7)\\
\wedge\ & idOrdering(WS) && (5.8)\\
\wedge\ & WS \neq \emptyset && (5.9)\\
\wedge\ & TS = \emptyset && (5.10)\\
\Rightarrow\ & \exists(T, i_f) \in WS\ .\ H, TS, \{(T, i_f)\} \uplus WS \rightsquigarrow H, \{T\} \uplus TS, WS && (5.11)
\end{aligned}$$

Proof: Deadlock Freedom. We proceed with a proof by contradiction:

Assume that no such $(T, i_f) \in WS$ exists. It follows that every task in WS is waiting on another task in either TS or WS:

$$\forall(T, i_f) \in WS.\ \exists(j_f, i_a, FS) \in TS \cup tasks(WS).\ i_f = j_f \tag{5.12}$$

From Eqs. (5.8) and (5.10) we obtain that every task awaits a task with a higher id:

$$\forall(T, i_f) \in WS.\ \exists((i_f, i_a, FS), j_f) \in WS.\ i_f < j_f \tag{5.13}$$

At the same time, Eq. (5.9) and the fact that WS is finite implies that there must exist a maximal element. Assuming finiteness of WS is valid as a finite program can only create a finite number of tasks:

$$\exists(T_{max}, i_f) \in WS.\ \forall(T', j_f) \in WS.\ j_f \leq i_f \tag{5.14}$$

We obtain a contradiction as this maximal element (T_{max}, i_f) has an id i_f larger than the id of all other tasks. $\square$

We can now proceed with progress.

Proof: Progress. We distinguish three cases for TS and WS:

1. $TS = \emptyset$ and $WS = \emptyset$: We can immediately conclude with Eq. (5.5).
2. $TS = \emptyset$ and $WS \neq \emptyset$: We apply E-RESUME to one of the waiting tasks obtained via Lemma 1.
3. $TS \neq \emptyset$: We continue with an arbitrary task $(i_f, i_a, FS) \in TS$

We distinguish three cases for FS:

1. $FS = \epsilon$: We can apply rule E-TASK-DONE
2. $FS = \langle L,\ x,\ P\rangle^l \circ \epsilon$: We can apply rule E-TASK-DONE
3. $FS = F \circ FS'$ where $F = \langle L,\ t,\ P\rangle^l$

For the last case, we continue with a case distinction on t. This is relatively straightforward as the evaluation rules are syntax directed. A detailed proof for the premises of each rule with all intermediate steps can be found in the appendix. □

5.2 Isolation

The isolation theorem states that no two active tasks can access the same object in the heap. Intuitively, this ensures that no data race can occur as a prerequisite for data races is two tasks concurrently accessing the same object. Note that waiting tasks are not strictly required to be isolated from the active tasks as they are not able to access the memory while waiting. One issue with this is the resumption of waiting tasks. If a waiting task is not isolated from all active tasks, resuming it will break isolation. We thus require each waiting task to be isolated from, or awaiting the termination of, each active task.

Based on these requirements, we can define isolation as shown in Fig. 5.1. ISO-TS states that the current program state is isolated, $\mathit{isolated}(H, TS, WS)$, if each pair of tasks, whether active or waiting, is isolated from each other. ISO-T, in turn, defines two tasks to be isolated from each other if the set of objects they can reach is disjoint

$$\frac{\forall T, T' \in TS \cup tasks(WS).\ isolated(H, WS, T, T')}{isolated(H, TS, WS)} \quad (\text{ISO-TS})$$

$$\frac{T = (i_f, i_a, FS) \qquad T' = (j_f, j_a, FS') \\ reachables(H, FS) \cap reachables(H, FS') = \emptyset \lor awaits(WS, T, T') \lor awaits(WS, T', T)}{isolated(H, WS, T, T')}$$

$$(\text{ISO-T})$$

$$\frac{(T, i_f) \in WS \land id_f(T') = i_f}{awaits(WS, T, T')} \quad (\textsc{Awaits-Base})$$

$$\frac{(U, k_f) \in WS \qquad awaits(WS, T, U) \qquad awaits(WS, U, T')}{awaits(WS, T, T')}$$

$$(\textsc{Awaits-Step})$$

Figure 5.1: Definitions for isolation

or one awaits the other. Finally, AWAITS-BASE and AWAITS-STEP define the transitive $awaits(WS, T, T')$ relation. Either T directly awaits the termination of T' by its id or T awaits a task U that in turn awaits the task T.

The initial state is trivially isolated as we only have one task. By showing that isolation is preserved when taking a reduction step we can show that any reachable program state is isolated.

We define the isolation preservation as follows: Assuming we take a step from a state H, TS, WS to H', TS', WS' and that the initial state is well-typed, fulfills all invariants and is isolated the final state will also be isolated. This can be formulated as follows:

Theorem 2 (Isolation)**.**

$$
\begin{aligned}
& H, TS, WS \rightsquigarrow H', TS', WS' && (5.15)\\
\land\ & H \vdash TS, WS && (5.16)\\
\land\ & H \vdash TS, WS\ \boldsymbol{ok} && (5.17)\\
\land\ & isolated(H, TS, WS) && (5.18)\\
\Rightarrow\ & isolated(H', TS', WS') && (5.19)
\end{aligned}
$$

Before we can prove the theorem, we make three observation about *isolated*. Firstly, removing a task T that does not await any other task preserves *isolated*. This follows from the definition of *isolated* and *awaits* as a task U awaiting a task U' before removing T will also await U' after T has been removed and the *reachables* of other tasks unaffected by T.

Secondly, if a task T is *isolated* from all other tasks in TS and WS, adding this task to either TS or WS, with some finish id i_f, preserves *isolated*. This follows immediately from the definition of *isolated*.

We can now prove isolation by analysing the reduction rules individually. The rules can affect the proof in two ways: (a) adding or removing tasks from either TS or WS and (b) affect the *reachables* set of a preexisting task. The first is covered by the two observations about task sets. The second is, for the most part, trivial as the *reachables* set is preserved by most rules. We provide a detailed analysis of the rules in the appendix.

5.3 Preservation

Preservation states that welltyped terms and tasks retain the same type after a reduction step. Furthermore, a set of invariants is preserved by the reduction. Due to time constraints, we do not provide a proof of preservation. Preservation can be stated as three theorems, one for each reduction relation:

Theorem 3 (Preservation: Frame Reduction)**.**

$$\vdash H : \star \tag{5.20}$$
$$\wedge\ H \vdash F : \sigma \tag{5.21}$$
$$\wedge\ H; a \vdash F\ \textit{\textbf{ok}} \tag{5.22}$$
$$\wedge\ H, F \rightarrow H', F' \tag{5.23}$$
$$\Longrightarrow \vdash H' : \star \tag{5.24}$$
$$\wedge\ H' \vdash F' : \sigma \tag{5.25}$$
$$\wedge\ H'; a \vdash F'\ \textit{\textbf{ok}} \tag{5.26}$$

Theorem 4 (Preservation: Frame Stack Reduction)**.**

$$\vdash H : \star \tag{5.27}$$
$$\land\ H \vdash FS \tag{5.28}$$
$$\land\ H; a \vdash FS\ \boldsymbol{ok} \tag{5.29}$$
$$\land\ H, FS \twoheadrightarrow H', FS' \tag{5.30}$$
$$\Longrightarrow \vdash H' : \star \tag{5.31}$$
$$\land\ H' \vdash FS' \tag{5.32}$$
$$\land\ H'; a \vdash FS'\ \boldsymbol{ok} \tag{5.33}$$

Theorem 5 (Preservation: Task Set Reduction)**.**

$$\vdash H : \star \tag{5.34}$$
$$\land\ H \vdash TS, WS \tag{5.35}$$
$$\land\ H \vdash TS, WS\ \boldsymbol{ok} \tag{5.36}$$
$$\land\ H, TS, WS \rightsquigarrow H', TS', WS' \tag{5.37}$$
$$\Longrightarrow \vdash H' : \star \tag{5.38}$$
$$\land\ H' \vdash TS', WS' \tag{5.39}$$
$$\land\ H' \vdash TS', WS'\ \boldsymbol{ok} \tag{5.40}$$

Most of the invariants we require are identical to those of LaCasa and can be found in the appendix. We define only a single additional invariant for the progress theorem, $\mathit{idOrdering}(WS)$, which is a property on a specific waiting task set WS. As such, to show that it is preserved, we only need to examine rules that modify WS, i.e. E-RESUME, E-CAPTURE, E-SWAP, E-FINISH. Of these, the first three only remove tasks from WS, that is $H, TS, WS \rightsquigarrow H, TS, WS'$ where $WS' \subseteq WS$. It follows that $\mathit{idOrdering}(WS) \Longrightarrow \mathit{idOrdering}(WS')$. E-FINISH also preserves ID ordering as the ID j_f is fresh and therefore larger than all other defined IDs.

Chapter 6

Related Work

In this chapter, we compare **PLC** with a selected number of different approaches. Figure 6.1 summarizes the differences. Note that Habanero-Java is only deadlock-free if a limited set of constructs is used and only deterministic if it is data race free.

6.1 LaCasa

LaCasa [HL16] is an extension to the Scala programming language that uses object capabilities, unique pointers and affine types to provide encapsulation and data-race freedom in an actor-based concurrency model. LaCasa focuses on being a minimal extension to the Scala language, requiring only a single bit of information per class to encode whether it follows the object capability model or not. In addition, it interacts soundly with the local type inference and requires no annotations of existing code, supporting a gradual transition. It has

	Data race freedom	Deadlock freedom	Determinism	Requires annotations
PLC	Yes	Yes	Yes	No
LaCasa	Yes	No	No	No
X10	Yes	Yes	No	Yes
Habanero-Java/Scala	Diagnostic	Yes*	Yes*	No
DPJ	Yes	Yes	Yes	Yes
Rust	Yes	No	No	Yes

Figure 6.1: Properties of related work

also been implemented as a compiler plugin.

LaCasa and, more specifically, CLC^2 form the basis for this thesis and **PLC**. The main difference is that LaCasa adds an actor-based extension in CLC^3, whereas **PLC** uses the Async Finish model. As a consequence, the formalization of **PLC** adds sets of active and waiting tasks with the main difficulty being *swap* and *capture* and how they interact with tasks. We improve on it in two significant ways.

First, LaCasa opts for an actor-based concurrency model which is inherently nondeterministic due to asynchronous message passing. Furthermore, deadlocks are possible and difficult to reason about due to the decentralized program logic in the actors. Finally, translating sequential algorithms to using actors is often not trivial. We solve these problems by adopting the Async Finish Model.

Second, we provide a proof of isolation, showing that tasks stay isolated and do not interfere with each other. This shows the absence of data races. Although LaCasa also ensures isolation, it only gives a high-level intuition for its correctness.

6.2 X10

X10 [Cha+05] is a programming language designed for systems with Non-Uniform Memory Access, that is, systems where memory access time depends on which processor is accessing it, and has three goals: (a) achieve high performance by colocating computations with the data they operate on; (b) achieve high developer productivity by supporting a variety of lightweight concurrency primitives such as async, finish and foreach; and (c) ensure deadlock freedom.

X10 uses a partitioned global address space, separating the heap into different areas, called *places*, each associated with a different compute node. Each asynchronous expression also takes a place in which it should be executed. By running asynchronous tasks in the same area as the memory they access performance can be increased at the cost of a few annotations.

Note that places have some similarity with boxes in that both partition the heap. But in contrast to boxes, places do not enforce strict isolation and separation as they are primarily a tool for ensuring that a program accesses data in an efficient manner. Furthermore, the number of places is arbitrary but fixed on program start and objects residing in one place permanently reside there.

6.3 Habanero-Java

Habanero-Java [Cav+11] is a programming language based on X10 which transpiles to standard Java classfiles. It focuses on safe concurrent programming with flexible constructs. It supports the Async-Finish model and some additional constructs such as *futures* in addition to integrating actors. A Habanero Scala version also exists [IS12b; IS12a].

Programs limiting themselves to the AFM are guaranteed to be deadlock free. Furthermore, if a program is data-race free and uses only the AFM, it is also deterministic. In a way, this is the best of both worlds, as it makes nondeterminism a choice for specific problems ill-suited to the AFM while making the majority of programs deterministic. To ensure data-race freedom, it supports diagnostic tools but does not provide static guarantees. Unfortunately, these tools can only show the presence of data races, not their absence.

Note that Habanero-Java is heavily inspired by X10 and aims to stay closer to Java, being almost a superset of the Java 5 language. This move aims to ease adoption. Since then, Habanero-Java has even been ported to a Java library called HJlib, lowering the hurdles to adoption even further. This comes with a trade-off, as HJlib does not possess the same safeguards due to not having special compiler support. Some lexical and scoping errors, for example, are not detected statically [IS14].

6.4 Deterministic Parallel Java

Deterministic Parallel Java (DPJ) [Boc+09] is an extension of the Java programming language that supports the Fork-Join model. The primary focus of the project is ensuring deterministic behavior. This requires that DPJ is also data-race free.

DPJ partitions the heap into regions and annotates classes with the region in which they reside. Regions share many similarities with boxes such as being hierarchical and non-overlapping. The key difference is that boxes encapsulate their content, whereas regions do not require this. Instead, a class might have different member variables that reside in different regions.

Explicit effect annotations of methods describe the regions accessed by the method. These annotations are then checked to ensure that no two concurrently executed methods both write or write and read to/from the same region. Concurrent reads are possible, enabling DPJ to implement some algorithms efficiently that **PLC** cannot, for example the Barnes-Hut algorithm presented in [Boc+09].

Both region and method effect annotations are necessary. The annotations are challenging to write and difficult to infer. Although some progress has been made towards inferring method effect annotations, it has not yet been possible to completely eliminate them and region annotations [Vak+09].

6.5 Rust

Rust [MK14] is probably the most well-known attempt at realizing memory safety, including data races. It is a completely new programming language with significant differences to any other mainstream language. Deadlock freedom and determinism are not guaranteed by the language. Instead, the language includes a future-based concurrency model.

The safety of Rust is based on a notion of *ownership* where the owner of an object encapsulates it similar to boxes. One difference is that the owner can create *borrowed references* to the owned object or any of its transitive references. This is different from **PLC** that only allows transferring ownership via e.g. *swap* of complete boxes.

One other important difference is that a borrow can be immutable: This allows the user to turn a mutable object temporarily immutable, which in turn relaxes the need for alias control. This is not possible in **PLC** as the permission protecting a box may never be duplicated.

6.6 LVars

LVars [KN13] pursues a completely different approach to determinism. Instead of controlling aliasing, it instead limits the operations on shared memory as operations that commute, producing the same result irrespective of order. More specifically, LVars requires writes to be monotonic, that is, shared data structures may grow but not shrink and that previous information is not invalidated. As an intuitive example, this means that a dictionary might be shared and that entries can be added but that no entries can be removed. In addition, the shared data cannot be read in its entirety; only blocking queries for specific keys are allowed.

This approach is quite powerful for suitable problems as it is not coupled to any specific concurrency model. Actors, futures or the AFM can theoretically all be used in combination with such a system. Unfortunately, it places heavy restrictions on the kind of operation as many operations are not monotonic. In fact, mutability often refers to overwriting of data which LVars does not support. Instead, monotonic writes can be seen as a middle ground between immutability and mutability as no key-value pair in a dictionary can be made inaccessible by monotonic writes.

These shared data structures allowing only monotonic writes, called *cells*, may depend on other cells through e.g. blocking reads awaiting a write. This allows even cyclic dependencies which where not handled correctly by LVars. Reactive Async [Hal+16] builds on LVars and handles cyclic dependencies and fallbacks correctly, but introduces nondeterminism through shared state. Reactive Async was, in turn, built upon by RACL [Arv18] to reintroduce determinism by combining it with LaCasa boxes and object capabilities.

Chapter 7

Conclusion

In this thesis, we introduced **PLC**, a programming language based on LaCasa [HL16], that supports the Async Finish model and guarantees data race freedom as well as deadlock freedom and determinism. We furthermore formalized a core language. While we also proved the properties progress and isolation, we fell short of proving preservation and confluence. This means that we cannot be certain that the system is completely sound.

7.1 Future Work

Naturally, finishing the proofs of preservation and confluence is an obvious next step. Defining the invariants necessary for showing both can also provide additional insights into how the system can be further adapted and simplified. For example, tasks are currently divided into two sets, one active and one waiting. Alternatively, the tasks could be organised in a tree structure where the leaves are active tasks. We did not adopt this structure as we were not certain whether or not the tree structure could be as easily adapted as changes were made.

An interesting extension would be integrating actors into the model. While determinism is a very desirable property, the AFM places some constraints on the program structure. A less structured model such as actors might be able to complement the AFM nicely. Even though LaCasa already introduced actors unforeseen complications might arise from combining them with the AFM.

Another observation we made is that a box cannot be unwrapped in **PLC**: Its content will never be available within the main task. Instead, we can *capture* a box in another box. This is more limiting than it should be. Instead, replacing *capture* by an *unwrap(x :* `Box`*[C]) : C* expression would allow the contents of a box to be made available even in the main task without requiring another box into which the contents are merged. It is not clear whether other complications arise.

Last but not least, this thesis is purely theoretical. Implementing the language as a compiler plugin for Scala is necessary before the language can actually be used. It would also allow gathering more real-world data on its strengths and weaknesses compared to other approaches. Building on the work of LaCasa, which is already implemented as a plugin, should reduce the effort required.

Appendix A

Definitions

Definition A.0.1 (Object Type)**.** For an object identifier $o \in \mathit{dom}(H)$ where $H(o) = \langle C, FM \rangle$, $\mathit{typeof}(H, o) := C$

Definition A.0.2 (Well-typed Heap)**.** A heap H is well-typed, written $\vdash H : \star$, iff

$$\begin{aligned}&\forall o \in \mathit{dom}(H).\ H(o) = \langle C, FM \rangle \implies (\mathit{dom}(FM) = \mathit{fields}(C)\ \land\\ &\qquad\qquad \forall f \in \mathit{dom}(FM).\ FM(f) = \texttt{null} \lor \mathit{typeof}(H, FM(f)) <: D \land \mathit{ftype}(C, f) \in \{\end{aligned}$$

Definition A.0.3 (Reachables)**.** The set of objects that can be reached by following an arbitrary number of references is called the *reachables* set. We define it for both objects and frame stacks:

$$\mathit{reachables}(H, o) = \{x \mid \mathit{reach}(H, o, x)\}$$

$$\mathit{reachables}(H, FS) = \bigcup\{\mathit{reachables}(H, o) \mid \mathit{accRoot}(o, FS)\}$$

Definition A.0.4. $\mathit{permTypes}(\Gamma)$ is the set of *permissions* in a typing context Γ,
$\mathit{permTypes}(\Gamma) = \{Q \mid \texttt{Perm}[Q] \in \Gamma\}$

Definition A.0.5 (WS-Tasks)**.** The tasks within a set of waiting tasks WS are denoted by $\mathit{tasks}(WS) = \{T.\ \exists i.(T, i) \in WS\}$

$$\frac{x \rightarrow o \in L \vee (x \rightarrow b(o,p) \in L \wedge p \in P)}{accRoot(o, \langle L, t, P\rangle^l)} \quad \text{(ACC-F)}$$

$$\frac{accRoot(o,F) \vee accRoot(o,FS)}{accRoot(o, F \circ FS)} \quad \text{(ACC-FS)}$$

$$\frac{o \in dom(H)}{reach(H,o,o)}$$

$$\frac{\begin{array}{c} o \in dom(H) \qquad H(o) = \langle C, FM\rangle \\ \exists (f \rightarrow o'') \in FM.\ reach(H,o'',o') \end{array}}{reach(H,o,o')}$$

$$\frac{x \rightarrow b(o,p) \in L \qquad p \in P}{boxRoot(o, \langle L,\ t,\ P\rangle^l)}$$

$$\frac{boxRoot(o,F)}{boxRoot(o, F \circ \epsilon)}$$

$$\frac{boxRoot(o,F) \vee boxRoot(o,FS)}{boxRoot(o, F \circ FS)}$$

$$\frac{x \rightarrow b(o,p) \in L \qquad p \in P}{boxRoot(o, \langle L,\ t,\ P\rangle^l, p)}$$

$$\frac{boxRoot(o,F,p)}{boxRoot(o, F \circ \epsilon, p)}$$

$$\frac{boxRoot(o,F,p) \vee boxRoot(o,FS,p)}{boxRoot(o, F \circ FS, p)}$$

$$\frac{boxRoot(o,FS) \qquad x \rightarrow o' \in env(F) \qquad reach(H,o,o')}{openbox(H,o,F,FS)}$$

Figure A.1: Predicates

$$\frac{\begin{array}{c} \mathit{boxSep}(H,F) \qquad \mathit{boxObjSep}(H,F) \qquad \mathit{boxOcap}(H,F) \\ a = \mathit{ocap} \implies \mathit{globalOcapSep}(H,F) \qquad \mathit{fieldUniqueness}(H,F) \end{array}}{H; a \vdash F \ \mathbf{ok}} \ \text{(F-OK)}$$

$$\frac{H; a \vdash F \ \mathbf{ok}}{H; a \vdash F \circ \epsilon \ \mathbf{ok}} \quad \text{(SINGFS-OK)}$$

$$\frac{\begin{array}{c} H; a \vdash F^l \ \mathbf{ok} \qquad H; a \vdash FS \ \mathbf{ok} \\ \mathit{boxSeparation}(H,F,FS) \\ \mathit{uniqueOpenBox}(H,F,FS) \\ \mathit{openBoxPropagation}(H,F^l,FS) \end{array}}{H; a \vdash F^l \circ FS \ \mathbf{ok}} \quad (\text{ FS-OK })$$

$$\frac{\begin{array}{c} \forall (i_f, i_a, FS) \in TS \cup \mathit{tasks}(WS).\ \begin{cases} H; \epsilon \vdash FS \, \mathbf{ok} & \text{if } i_a = i_{a0} \\ H; \mathit{ocap} \vdash FS \, \mathbf{ok} & \text{otherwise} \end{cases} \\ \mathit{idOrdering}(WS) \qquad \mathit{finishUniqueness}(WS) \\ \mathit{asyncTaskUniqueness}(TS) \qquad \mathit{asyncGroupUniqueness}(TS, WS) \\ \mathit{isolated}(H, TS, WS) \end{array}}{H \vdash TS, WS \, \mathbf{ok}}$$

$$(\text{ TS-OK })$$

Figure A.2: Invariants

Appendix B

Proofs

B.1 Progress

Proof: Case Distinction for Progress. We continue with a case distinction on t assuming we have reduced a task $(i_f, i_a, FS) \in TS$ where $FS = F \circ FS'$ and $F = \langle L,\ t,\ P\rangle^l$.

B.1.0.0.1 Finish Assume $t =$ let $\mathit{finish}\{s\}$ in s. We define T_1, T_2 and j_f according to E-FINISH and can directly apply the rule.

B.1.0.0.2 Async Assume $t = \mathit{async}(b, x \Rightarrow s)\{u\}$. To apply E-ASYNC, we must show that $L(b) = b(o, p)$ and $p \in P$:

Proof.

$$H \vdash F : \tau \quad \text{by T-TS, T-FS-A, T-FS-NA, Eq. (5.2)} \tag{B.1}$$

$$H \vdash \Gamma; L \quad \text{by previous, T-FRAME1} \tag{B.2}$$

$$\forall x \in dom(\Gamma).\ H \vdash \Gamma; L; x \quad \text{by previous, WF-ENV} \tag{B.3}$$

$$\Gamma; a \vdash async(b, x \Rightarrow s)\{u\} : \tau \quad \text{by T-FRAME1, Eq. (B.1)} \tag{B.4}$$

$$\Gamma; a \vdash b : Q \triangleright \text{Box}[C] \quad \text{by previous, T-ASYNC} \tag{B.5}$$

$$b \in dom(\Gamma) \quad \text{by previous, T-VAR} \tag{B.6}$$

$$H \vdash \Gamma; L; b \quad \text{by previous, Eq. (B.3)} \tag{B.7}$$

$$\begin{aligned} & L(b) = null \\ & \ \vee L(b) = o \wedge typeof(H, o) <: \Gamma(b) \\ & \ \vee (L(b) = b(o, p) \wedge \Gamma(b) = Q \triangleright \text{Box}[C] \\ & \ \wedge typeof(H, o) <: C) \quad \text{by previous, WF-VAR} \end{aligned} \tag{B.8}$$

$$L(b) = null \vee L(b) = b(o, p) \quad \text{by previous, T-VAR, Eq. (B.5)} \tag{B.9}$$

We conclude that either $L(b)$ is null and we are done by **??** or that $L(b) = b(o, p)$ in which case we proceed to show $p \in P$:

$$\forall\ \text{Perm}[Q] \in \Gamma.\ \gamma(Q) \in P \quad \text{by WF-PERM, injective } \gamma \tag{B.10}$$

$$\begin{aligned} & (\Gamma(b) = Q \triangleright \text{Box}[C] \\ & \wedge L(b) = b(o, p) \wedge \text{Perm}[Q] \in \Gamma) \Rightarrow p \in P \quad \text{by previous, WF-PERM} \end{aligned} \tag{B.11}$$

$$L(b) = b(o, p) \Rightarrow p \in P \quad \text{by previous, T-VAR, T-ASYNC, Eq. (B.1), Eq.} \tag{B.12}$$

□

We define T_1, T_2 and j_a according to E-ASYNC and apply the rule.

B.1.0.0.3 Capture Assume $t = capture(x.f, y)\{z \Rightarrow t'\}$. To apply rule E-CAPTURE, we must show the following:

$$L(x) = b(o, p) \tag{B.13}$$
$$\wedge\, L(y) = b(o', p') \tag{B.14}$$
$$\wedge\, \{p, p'\} \subseteq P \tag{B.15}$$

Note that we do not show that $H(o) = \langle C, FM \rangle$ as o is by definition of L not `null`. As such, this premise is trivial to fulfill by defining C and FM accordingly. This proof is directly based on LaCasa [HL16].

Proof.

$$\Gamma; a \vdash capture(x.f, y)\{z \Rightarrow t' : \sigma\} \tag{B.16}$$
$$\wedge\, H \vdash \Gamma; L \tag{B.17}$$
$$\wedge \vdash \Gamma; L; P \quad \text{by Eq. (5.2), T-FS-A, T-FS-NA} \tag{B.18}$$

$$\Gamma; a \vdash x : Q \rhd \texttt{Box}[C] \tag{B.19}$$
$$\wedge\, \Gamma; a \vdash y : Q' \rhd \texttt{Box}[D] \tag{B.20}$$
$$\wedge\, \{\texttt{Perm}[Q], \texttt{Perm}[Q']\} \subseteq \Gamma \quad \text{by T-CAPTURE, Eq. (B.16)} \tag{B.21}$$
$$\wedge\, \forall x \in dom(\Gamma).\ H \vdash \Gamma; L; x \quad \text{by WF-ENV, Eq. (B.17)} \tag{B.22}$$

$$L(x) = b(o, p) \wedge L(y) = b(o', p') \vee L(x) = \texttt{null}$$
$$\vee\, L(y) = \texttt{null} \quad \text{by WF-VAR, Eqs. (B.19), (B.20) and (B.22)} \tag{B.23}$$

Equation (B.23) states that either one of $L(x)$ and $L(y)$ are `null`, in which case we conclude by **??**, or that both are boxes as in Eqs. (B.13) and (B.14). The availability of their permissions as required by Eq. (B.15) is shown by WF-PERM and Eqs. (B.18) to (B.21) and (B.23). □

We define C, FM, H' and WS' according to E-CAPTURE and apply the rule.

B.1.0.0.4 Swap Analogous to Capture.

Lemma 6. *If $C' <: C$ and $f \in \mathit{fields}(C)$, then $f \in \mathit{fields}(C') \wedge \mathit{ftype}(C, f) = \mathit{ftype}(C', f)$*

Proof. Directly by <:-Ext and WF-CLASS. □

B.1.0.0.5 Select Assume $t = \text{let } x = y.f \text{ in } t'$. To apply rule E-SELECT with help of E-STACK and E-FRAME, we have to show that $f \in \mathit{dom}(FM)$ where $H(L(y)) = \langle C, FM, \rangle$. Alternatively, $L(y) =$ `null` in which case evaluation gets stuck. This proof is directly based on LaCasa [HL16]

Proof.

$$H \vdash F : \sigma \quad \text{by T-TS, T-FS-NA, T-FS-A, Eq. (5.2)} \tag{B.24}$$

$$\Gamma; a \vdash t : \sigma \wedge H \vdash \Gamma; L \quad \text{by T-FRAME1, previous} \tag{B.25}$$

$$\Gamma(y) = C \wedge \mathit{ftype}(C, F) = D \quad \text{by T-LET, T-SELECT, T-VAR, previous} \tag{B.26}$$

$$\mathit{dom}(\Gamma) \subseteq \mathit{dom}(L) \wedge \forall x \in \mathit{dom}(\Gamma).\ H \vdash \Gamma; L; x \text{ by WF-ENV, Eq. (B.25)} \tag{B.27}$$

$$L(y) = \texttt{null} \vee L(y) = o \wedge \mathit{typeof}(H, o) <: C \quad \text{by WF-VAR, previous, Eq. (B.26)} \tag{B.28}$$

If $L(y) =$ `null`, we immediately conclude by **??**. Otherwise, we continue with proving our original goal:

$$L(y) = o \wedge \mathit{typeof}(H, o) <: C \text{ by previous} \tag{B.29}$$

$$f \in \mathit{fields}(C) \quad \text{by Eq. (B.26)} \tag{B.30}$$

$$o \in \mathit{dom}(H) \quad \text{by Eq. (B.29)} \tag{B.31}$$

$$\text{Define } \langle C', FM \rangle := H(o) \quad \text{by previous} \tag{B.32}$$

$$\mathit{dom}(FM) = \mathit{fields}(C') \quad \text{by previous, Eq. (5.1) and Definition A.0.2} \tag{B.33}$$

$$f \in \mathit{dom}(FM) \quad \text{by previous, Eqs. (B.29) and (B.30) and Lemma 6} \tag{B.34}$$

□

B.1.0.0.6 Assign Assume $t = \text{let } x = y.f = z \text{ in } t'$. To apply rule E-ASSIGN with help of E-STACK and E-FRAME, we must show that $L(y)$ is not null. The proof is analogous to the previous case.

B.1.0.0.7 Remaining rules For the following terms nothing has to be shown. We can trivially define the necessary variables and then apply the corresponding rule in combination with E-STACK and/or E-FRAME.

- $t = \text{let } x = \text{ null in } t'$: E-NULL
- $t = \text{let } x = y \text{ in } t'$: E-VAR
- $t = \text{let } x = \text{ new } C \text{ in } t'$: E-NEW
- $t = x$: E-RETURN1 or E-RETURN2
- $t = box[C]\{x \Rightarrow t'\}$: E-BOX

□

B.2 Isolation

Lemma 7. $TS' \subseteq TS \wedge WS' \subseteq WS \wedge isolated(H, TS, WS) \Rightarrow isolated(H, TS', WS')$

Proof. The proof follows from the definition of isolation. □

Lemma 8 (Isolation-Step-TS)**.** *Assuming we have isolated sets of tasks TS, WS and that a task U is isolated from each task in these sets, we know that the task sets $TS \cup \{U\}, WS$ are also isolated.*

$$\begin{aligned}
&isolated(H, TS, WS)\\
&\wedge \forall T \in TS \cup tasks(WS).\ isolated(H, T, U)\\
&\Rightarrow isolated(H, TS \cup \{U\}, WS)
\end{aligned}$$

Proof. This follows directly from the definition of isolation. □

Lemma 9 (Isolation-Step-WS)**.** *Assuming we have isolated sets of tasks TS, WS and that a waiting task (U, i_f) is isolated from each task in these sets* *we know that the task sets $TS, WS \cup \{(U, i_f)\}$ are also isolated.*

$$\begin{aligned} & isolated(H, TS, WS) \\ \wedge\ & \forall T \in TS \cup tasks(WS).\ isolated(H, T, U) \\ \Rightarrow\ & isolated(H, TS, WS \cup \{(U, i_f)\}) \end{aligned}$$

Proof. This follows directly from the definition of isolation. □

Theorem 10 (Isolation)**.**

$$H, TS, WS \rightsquigarrow H', TS', WS' \tag{B.35}$$
$$\wedge\ H \vdash TS, WS \tag{B.36}$$
$$\wedge\ H \vdash TS, WS\ \textbf{\textit{ok}} \tag{B.37}$$
$$\wedge\ isolated(H, TS, WS) \tag{B.38}$$
$$\Rightarrow\ isolated(H', TS', WS') \tag{B.39}$$

Proof. We analyse the rules individually to proof the theorem. The rules can be divided into two categories: Firstly, the rules added by **PLC** that handle tasks and have to ensure that no two unisolated tasks run concurrently. Secondly, the rules from LaCasa which modify the *reachables* set of frame stacks.

B.2.0.0.1 E-TASK-DONE Assume that Eq. (B.35) uses rule E-TASK-DONE and that $TS = \{T\} \uplus TS'$, i.e. that the task T is removed by application of rule E-TASK-DONE, $H = H'$ and $WS = WS'$. $isolated(H, TS', WS)$ immediately follows from Lemma 7.

B.2.0.0.2 E-RESUME Assume that Eq. (B.35) uses rule E-RESUME and that $WS = \{(T, j_f)\} \uplus WS'$, that $TS' = \{T\} \uplus TS$ and $H = H'$. Let $T = (i_f, i_a, FS)$. We know from E-RESUME that T does not await any task, i.e. $\nexists (i_f, j_a, FS) \in TS\, tasks(WS)$. Combined with $isolated(H, TS, WS)$, we obtain:

$$\forall T' \in TS \cup tasks\,.\ T' = (j_f, j_a, FS') \Rightarrow isolated(H, FS, FS') \vee awaits(WS, T', T)$$

Furthermore, we know from isolation that:

$$\forall T' \in TS \cup tasks(WS').\ isolated(H, T, T')$$

Isolation follows from Lemmas 7 and 8.

B.2.0.0.3 E-FINISH Assume that Eq. (B.35) uses rule E-FINISH and that $TS = \{T\} \uplus TS''$, $TS' = \{T_1\} \uplus TS''$ and $WS' = \{(T_2, j_f)\} \uplus WS$. Let $T_1 = (j_f, i_a, \langle L,\ t,\ P\rangle^l)$ and $T_2 = (i_f, i_a, \langle L[x = null],\ s,\ P\rangle^m \circ FS)$.

To show that E-FINISH preserves isolation, we first show that both T_1 and T_2 are isolated from all other tasks and secondly that T_1 and T_2 are isolated from each other (as one awaits the other).

Due to Lemma 7 and Eq. (B.38), we obtain $isolated(H, TS'', WS)$.

Using $reachables(H, T_1) = reachables(H, T_2) = reachables(H, T)$ and $id_a(T) = id_a(T_1) = id_a(T_2)$ and $id_f(T_1) > id_f(T_2)$, we obtain that any task U whose frame stack isolated from T's frame stack is also isolated from T_1, T_2 as they have the same *reachables*. If U instead awaited T, it will now await T_2 (and T_1 transitively).:

$$isolated(H, T_1, T_2) \wedge \forall U \in TS'' \cup tasks(WS).\ isolated(H, U, T_2) \wedge isolated(H, U, T_1)$$

We can thus apply Lemmas 8 and 9 to obtain $isolated(H, TS', WS')$.

B.2.0.0.4 E-ASYNC Assume that Eq. (B.35) uses rule E-ASYNC and that $TS = \{T\} \uplus TS''$, $TS' = \{T_1, T_2\} \uplus TS''$ and $WS' = WS$. Let $T_1 = (i_f, j_a, \langle[x \to o],\ t,\ \emptyset\rangle^\epsilon)$ and $T_2 = (i_f, i_a, \langle L,\ s,\ P \setminus \{p\}\rangle^\epsilon)$.

Similar to E-FINISH, we can show isolation of T_1 and T_2 with all other tasks and that T_1 and T_2 are isolated from each other.

Due to Lemma 7 and Eq. (B.38), we obtain $isolated(H, TS'', WS)$.

As $accRoots(T_1) = \{o\}$, we know that $reachables(H, T_1) \subseteq reachables(H, T)$. Using this and ISO-FS, we obtain that all tasks U whose frame stack is isolated from T are also isolated from T_1. If U awaited T it will also await T_1 as $id_f(T) = id_f(T_1)$:

$$\forall U \in TS'' \cup tasks(WS).\ isolated(H, U, T_1)$$

As T_2 has only the variable bindings from T's top frame, we know that $reachables(H, T_2) \subseteq reachables(H, T)$. Furthermore, a task awaiting T will also await T_2 as $id_f(T_2) = id_f(T)$. Using this, we obtain:

$$\forall U \in TS'' \cup tasks(WS).\ isolated(H, U, T_2)$$

Furthermore, $reachables(H, T_1) \cap reachables(H, T_2) = \emptyset$ as T_2 does not have the permission p. This implies $isolated(H, T_1, T_2)$. Thus we can apply Lemmas 8 and 9 to obtain $isolated(H, TS', WS)$.

B.2.0.0.5 E-CAPTURE Assume that Eq. (B.35) uses rule E-CAPTURE and that $TS = \{T\} \uplus TS''$, $TS' = \{T'\} \uplus TS''$ and $WS'' = \{((i_f, i_a, GS), j_f) \mid ((i_f, i_a, FS) \in WS \wedge (i_a = id_a(T) \implies GS = \epsilon) \wedge (i_a \neq id_a(T) \implies GS = FS)\}$. Furthermore, let $T = (i_f, i_a, \langle L,\ capture(x.f, y)\{z \Rightarrow t\},\ P\rangle^l \circ FS$ and $T' = (i_f, i_a, \langle L[z \to L(x)],\ t,\ P \setminus \{p'\}\rangle^\epsilon \circ \epsilon)$.

Firstly, every task in WS'' has the same ids as some task in WS while their frame stack is either identical or ϵ. As each pair of tasks in WS is isolated, the corresponding pair in WS'' is also isolated, i.e. $isolation(H, TS, WS) \implies isolation(H, TS, WS'')$.

As $reachables(H, T') \subseteq reachables(H, T)$ and $id_a(T) = id_a(T')$, it follows that $isolated(H, TS', WS'')$.

B.2.0.0.6 E-SWAP Analogous to E-CAPTURE. Although o'' is a box root after applying E-SWAP, it was already in the reachables set before E-SWAP.

B.2.0.0.7 E-INVOKE Assume that Eq. (B.35) uses rule E-INVOKE and that $TS = \{T\} \uplus TS''$, $TS' = \{T'\} \uplus TS''$ and $WS' = WS$. Let $T = (i_f, i_a, F \circ FS)$, $T' = (i_f, i_a, F' \circ \langle L,\ t,\ P\rangle^l \circ FS)$, $F = \langle L,\ \text{let } x = y.m(z) \text{ in } t,\ P\rangle^l$ and $F' = \langle L',\ t',\ P\rangle^x$.

We distinguish two cases: First, assume that $i_a = 0$, i.e.:

$$L' = [this \to L(y), x \to L(z)]$$

It follows that $roots(F') \subseteq roots(F)$ as F' only contains two of the variables of L. Isolation is preserved.

For the second case, assume that $i_a \neq 0$, i.e.:

$$L' = L_0[this \to L(y), x \to L(z)]$$

Due to Eq. (B.37) we know that all tasks (j_a, j_f, FS) with $j_a \neq 0$ are checked as $ocap$, i.e. $H; ocap \vdash FS\,\mathbf{ok}$ which in turn implies $globalOcapSep$ for every frame in FS. Thus, $\forall o \in reachables(H, FS), x \to o' \in L_0.\ sep(H, o, o')$.

For all tasks U' with $id_a(U') = 0$ we obtain through $asyncTaskUniqueness$ and $asyncGroupUniqueness$ that $awaits(H, WS, U', T)$. Together with the previous paragraph, this implies $isolated(H, TS', WS')$.

B.2.1 E-STACK

In this section, we analyse E-STACK and all the possible rules that can reduce the stack of a single task. The following rules have in common that they only affect a single tasks frame stack and the heap. The ids and the waiting task set WS are fixed and thus the only property in isolation that can possibly be violated by E-STACK is the isolation of two frame stacks.

Assume that Eq. (B.35) uses rule E-STACK and that $TS = \{(i, d, FS)\} \uplus TS''$, $TS' = \{(i, d, FS')\} \uplus TS''$ and $WS' = WS$. We continue by analysing the frame stack evaluation rules.

B.2.1.0.1 E-BOX $\mathit{reachables}(H, FS') \subseteq \mathit{reachables}(H, FS) \cup \{o\}$. As $o \notin \mathit{dom}(H)$ it follows that o is not in any reachables set before E-BOX is applied. Consequently, isolation is preserved.

B.2.1.0.2 E-RETURN1 $\mathit{reachables}(H, FS') \subseteq \mathit{reachables}(H, FS)$ as $L(x)$ already was a root.

B.2.1.0.3 E-RETURN2 $\mathit{reachables}(H, FS') \subseteq \mathit{reachables}(H, FS)$ as L is completely discarded.

B.2.2 E-FRAME

We continue with the frame evaluation rules. Let $FS = F \circ FS''$ and $FS' = F' \circ FS''$.

B.2.2.0.1 E-NULL $\mathit{reachables}(H, FS') \subseteq \mathit{reachables}(H, FS)$ as $L(x)$ might be an object which is now no longer reachable.

B.2.2.0.2 E-VAR $\mathit{reachables}(H, FS') \subseteq \mathit{reachables}(H, FS)$ as $L(y)$ already was a root.

B.2.2.0.3 E-SELECT $\mathit{reachables}(H, FS') \subseteq \mathit{reachables}(H, FS)$ as $FM(f)$ already was a root.

B.2.2.0.4 E-ASSIGN $reachables(H, FS') \subseteq reachables(H, FS)$ as $L(z)$ already was a root. Note that H' differs from H only in o and that o is not in the reachables set of any other task besides ancestors due to isolation. As such, isolation is preserved.

B.2.2.0.5 E-NEW $reachables(H, FS') = reachables(H, FS) \cup \{o\}$. As $o \notin dom(H)$ it follows that o is not in any reachables set before E-NEW is applied. Consequently, isolation is preserved.

$\square$

References

[23] *Fork–join model*. In: *Wikipedia*. Page Version ID: 1157279085. May 27, 2023.

[Arv18] E. Arvidsson. *Deterministic Concurrency Using Lattices and the Object Capability Model*. 2018.

[BLR02] C. Boyapati, R. Lee, and M. C. Rinard. "Ownership types for safe programming: preventing data races and deadlocks". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*. Ed. by M. Ibrahim and S. Matsuoka. ACM, 2002, pp. 211–230. DOI: 10.1145/582419.582440.

[Boc+09] R. L. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. "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. New York, NY, USA: Association for Computing Machinery, Oct. 25, 2009, pp. 97–116. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640097.

[BR01] C. Boyapati and M. C. Rinard. "A Parameterized Type System for Race-Free Java Programs". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*. Ed. by L. M. Northrop and J. M. Vlissides. ACM, 2001, pp. 56–69. DOI: 10.1145/504282.504287.

[Cav+11] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. "Habanero-Java: the new adventures of old X10". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. New York, NY, USA: Association for Computing Machinery, Aug. 24, 2011, pp. 51–61. ISBN: 978-1-4503-0935-6. DOI: 10.1145/2093157.2093165.

[Cha+05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: an object-oriented approach to non-uniform cluster computing". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, Oct. 12, 2005, pp. 519–538. ISBN: 978-1-59593-031-6. DOI: 10.1145/1094811.1094852.

[CW03] D. Clarke and T. Wrigstad. "External Uniqueness Is Unique Enough". In: *ECOOP 2003 – Object-Oriented Programming*. Ed. by L. Cardelli. Berlin, Heidelberg: Springer, 2003, pp. 176–200. ISBN: 978-3-540-45070-2. DOI: 10.1007/978-3-540-45070-2_9.

[Goe+06] B. Goetz, T. Peierls, J. J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006. ISBN: 978-0-321-34960-6.

[Hal+16] P. Haller, S. Geries, M. Eichberg, and G. Salvaneschi. "Reactive Async: expressive deterministic concurrency". In: *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*. Ed. by A. Biboudis, M. Jonnalagedda, S. Stucki, and V. Ureche. ACM, 2016, pp. 11–20. DOI: 10.1145/2998392.2998396.

[HL16] P. Haller and A. Loiko. "LaCasa: lightweight affinity and object capabilities in Scala". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, Oct. 19, 2016, pp. 272–291. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984042.

[HYC16] K. Honda, N. Yoshida, and M. Carbone. “Multiparty Asynchronous Session Types”. In: *J. ACM* 63.1 (2016), 9:1–9:67. DOI: 10.1145/2827695.

[IS12a] S. Imam and V. Sarkar. “Habanero-scala: Async-finish programming in scala”. In: *The Third Scala Workshop (Scala Days 2012)*. 2012.

[IS12b] S. M. Imam and V. Sarkar. “Integrating task parallelism with actors”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. OOPSLA ’12. New York, NY, USA: Association for Computing Machinery, Oct. 19, 2012, pp. 753–772. ISBN: 978-1-4503-1561-6. DOI: 10.1145/2384616.2384671.

[IS14] S. Imam and V. Sarkar. “Habanero-Java library: a Java 8 framework for multicore programming”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. PPPJ ’14. New York, NY, USA: Association for Computing Machinery, Sept. 23, 2014, pp. 75–86. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647514.

[KN13] L. Kuper and R. R. Newton. “LVars: lattice-based data structures for deterministic parallelism”. In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. FHPC ’13. New York, NY, USA: Association for Computing Machinery, Sept. 23, 2013, pp. 71–84. ISBN: 978-1-4503-2381-9. DOI: 10.1145/2502323.2502326.

[LP10] J. K. Lee and J. Palsberg. “Featherweight X10: a core calculus for async-finish parallelism”. In: *SIGPLAN Not.* 45.5 (Jan. 9, 2010), pp. 25–36. ISSN: 0362-1340. DOI: 10.1145/1837853.1693459.

[Mil06] M. Miller. *Robust composition: Towards a unified approach to access control and concurrency control*. Johns Hopkins University, 2006.

[MK14] N. D. Matsakis and F. S. Klock. “The rust language”. In: *ACM SIGAda Ada Letters* 34.3 (Oct. 18, 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188.

[Vak+09] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. “Inferring method effect summaries for nested heap regions”. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 421–432.

TRITA-EECS-EX- 2022:00

€€€€ For DIVA €€€€

{
"Author1": { "Last name": "Willenbrink",
"First name": "Sebastian",
"Local User Id": "u1oy2pa8",
"E-mail": "stwi@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Cycle": "2",
"Course code": "DA250X",
"Credits": "30.0",
"Degree1": {"Educational program": "Degree Programme in Computer Science and Engineering"
,"programcode": "CDATE"
,"Degree": "Degree of Master of Science in Engineering"
,"subjectArea": "Computer Science and Engineering"
},
"Title": {
"Main title": "A Type System for Ensuring Safe, Structured Concurrency in Scala",
"Language": "eng" },
"Alternative title": {
"Main title": "Ett typsystem för säker och strukturerad samtidighet",
"Language": "swe"
},
"Supervisor1": { "Last name": "Haller",
"First name": "Philipp",
"Local User Id": "u100003",
"E-mail": "phaller@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"Examiner1": { "Last name": "Guanciale",
"First name": "Roberto",
"Local User Id": "u1d13i2c",
"E-mail": "robertog@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"National Subject Categories": "10201, 10205",
"Other information": {"Year": "2024", "Number of pages": "1,69"},
"Copyrightleft": "copyright",
"Series": { "Title of series": "TRITA-EECS-EX" , "No. in series": "2022:00" },
"Opponents": { "Name": ""},
"Number of lang instances": "3",
"Abstract[eng]": €€€€
Concurrent programming brings many pitfalls, such as data races, deadlocks and nondeterminism. Designing programming languages to eliminate these classes of bugs is an active area of research. Previous attempts at this focused on creating completely new languages, hampering adoption and reuse of existing code, or were either insufficient to address all of data races, deadlocks and nondeterminism or required annotating code, complicating reuse of existing libraries.
In this thesis, we present a new approach to integrating safe concurrency into an existing mainstream programming language, Scala. To do so, we combine two approaches: We represent concurrency in code using the Async Finish Model, a variant of the widely used Fork Join Model. It is inherently free of deadlocks and deterministic. To ensure data race freedom, we isolate subgraphs of the heap by building on the work of LaCasa [HL16]. LaCasa requires no annotations as the compiler can automatically verify that this isolation is preserved.
We provide an informal overview of a language combining these two approaches along with a formalization of the operational semantics. In addition, we show the progress of our reduction and preservation of the necessary invariants as well as isolation of asynchronous tasks. Progress implies deadlock freedom and isolation of asynchronous tasks implies data race freedom. We also formally state and provide an argument for determinism.
€€€€,

Actually write preservation down

"Keywords[eng]": €€€€
Concurrent programming, Object-oriented programming, Operational semantics, Scala €€€€,
"Abstract[swe]": €€€€
Samtidig programmering medför många fallgropar, t.ex. samtidiga åtkomst till samma minne, baklås och nondeterminism. Att utforma programmeringsspråk för att eliminera dessa typer av buggar är ett aktivt forskningsområde. Tidigare försök har fokuserat på att skapa helt nya språk, vilket försvårar användning och återanvändning av befintlig kod, eller så har de antingen inte varit tillräckliga för att hantera alla nämnda problem eller så har de krävt att koden annoteras, vilket försvårar återanvändning av befintliga bibliotek.
I den här avhandlingen presenterar vi en ny metod för att integrera säker samtidighet i ett befintligt vanligt programmeringsspråk, Scala. För att göra detta kombinerar vi två metoder: Vi representerar samtidighet i kod med hjälp av det så kallade Async Finish Model, en variant av den allmänt använda Fork Join Model. Den är i sig själv fri från baklås och deterministisk. För att säkerställa frihet från samtidiga minnesåtkomster isolerar vi delgrafer av heapen genom att bygga vidare på LaCasas arbete [HL16]. LaCasa kräver inga annotationer eftersom kompilatorn automatiskt kan verifiera att denna isolering bevaras.
Vi ger en informell översikt över ett språk som kombinerar dessa två tillvägagångssätt tillsammans med en formalisering av den operativa semantiken. Dessutom visar vi reduktionens framsteg och bevarande av semantikens invarianter samt isolering av asynkrona trådar. Reduktionens framsteg innebär frihet från baklås och isolering av asynkrona tråder innebär frihet från samtidiga minnesåtkomster. Vi formaliserar och ger dessutom ett argument för determinism. €€€€,
"Keywords[swe]": €€€€
Samtidig programmering, Objekt-orienterad programmering, Operationell semantik, Scala €€€€,
"Abstract[ger]": €€€€
Parallele Programmierung birgt viele Stolperfallen, wie zum Beispiel gleichzeitige Speicherzugriffe, Deadlocks und Nichtdeterminismus. Die

Entwicklung von Programmiersprachen zur Vermeidung dieser Fehlerklassen ist ein aktives Forschungsgebiet. Bisherige Versuche konzentrierten sich auf die Entwicklung völlig neuer Sprachen, was die Übernahme und Wiederverwendung von bestehendem Code erschwert, oder sie waren entweder unzureichend, um alle genannten Probleme zu verhindern, oder sie erforderten die Annotierung von Code, was die Wiederverwendung bestehender Bibliotheken erschwert.
In dieser Arbeit stellen wir einen neuen Ansatz zur Integration von sicherer Parallelität in eine bestehende Mainstream-Programmiersprache, Scala, vor. Zu diesem Zweck kombinieren wir zwei Ansätze: Wir stellen die Nebenläufigkeit im Code mithilfe des Async Finish Model dar, einer Variante des weit verbreiteten Fork Join Model. Es ist von Natur aus frei von Deadlocks und deterministisch. Um die Freiheit von gleichzeitigen Speicherzugriffen zu gewährleisten, isolieren wir Teilgraphen des Heaps, indem wir auf der Arbeit von LaCasa [HL16] aufbauen. LaCasa erfordert dabei keine Anmerkungen, da der Compiler automatisch überprüfen kann, dass diese Isolation erhalten bleibt.
Wir geben einen informellen Überblick über eine Sprache, die diese beiden Ansätze kombiniert, sowie eine Formalisierung der operativen Semantik dieser Sprache. Darüber hinaus zeigen wir den Fortschritt der Reduktion und die Erhaltung der Invarianten der Semantik sowie die Isolierung asynchroner Ausführungseinheiten. Fortschritt der Reduktion impliziert Deadlock-Freiheit und die Isolierung asynchroner Aufgaben impliziert die Abwesenheit paralleler Speicherzugriffe. Wir formalisieren und liefern außerdem ein Argument für Determinismus. €€€,
"Keywords[ger]": €€€
Parallele Programmierung, Objekt-orientierte Programmierung, Operative Semantik, Scala €€€,
}