DEGREE PROJECT IN THE FIELD OF TECHNOLOGY
ENGINEERING PHYSICS
AND THE MAIN FIELD OF STUDY
COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Deterministic Concurrency Using Lattices and the Object Capability Model

ELLEN ARVIDSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Deterministic Concurrency Using Lattices and the Object Capability Model

ELLEN ARVIDSSON

Master in Computer Science
Date: December 5, 2018
Supervisor: Philipp Haller
Examiner: Mads Dam
Swedish title: Determinism i parallelliserade program med hjälp av gitterstrukturer och objektsförmågor
School of Electrical Engineering and Computer Science

Abstract

Parallelization is an important part of modern data systems. However, the non-determinism of thread scheduling introduces the difficult problem of considering all different execution orders when constructing an algorithm. Therefore deterministic-by-design concurrent systems are attractive. A new approach called LVars consists of using data which is part of a lattice, with a predefined join operation. Updates to shared data are carried out using the join operation and thus the updates commute. Together with limiting the reads of shared data, this guarantees a deterministic result. The Reactive Async framework follows a similar approach but has several aspects which can cause a non-deterministic result. The goal with this thesis is to explore how we can ammend Reactive Async in order to guarantee a deterministic result. First an exploration into the subtleties of lattice based data combined with concurrency is made. Then a formal model based on a simple object-oriented language is constructed. The constructed small-step operational semantics and type system are shown to guarantee a form of determinism. This shows that LVars-similar system can be implemented in an object-oriented setting. Furthermore the work can act as a basis for future revisions of Reactive Async and similar frameworks.

Sammanfattning

Parallellisering är en viktig del i moderna datasystem. Flertrådade applikationer innebär dock en svårighet i och med att programmerare måste ta alla exekveringsordningar i beaktning. Därför är beräkningsmodeller vars resultat är garanterat deterministiskt en attraktiv utväg. En ny modell, kallad LVars, använder gitterstrukturer tillsammans med en supremum-operation för att garantera att uppdateringar av delad data kommuterar. Detta tillsammans med begränsningar av läsning av datan garanterar ett deterministiskt resultat. Reactive Async är ett programmeringsramverk som följer en liknande strategi. Det finns dock flera delar i dess konstruktion som i en oförsiktig programmerares händer kan orsaka att ett programs resultat blir icke-deterministiskt. Målet med detta examensarbete är att utforska vilka modifikationer som skulle kunna göras av Reactive Async för att garantera determinism. Först görs en undersökning av de mer svårförståeliga delarna i kombinationen av gitterbaserad data med flertrådad exekvering. Sedan konstrueras en formell beräkningsmodell baserad på ett enkelt objektorienterat språk. Konstruktionens småstegade operationella semantik tillsammans med dess typsystem visas kunna garantera en form av determinism. Detta visar att ett system liknande LVars kan implementeras i ett objektorienterat språk. Därmed skulle detta arbete kunna ligga till grund för framtida versioner av Reactive Async.

Contents

Chapter 1

Introduction & Motivation

The main approach to achieving concurrency in general applications has long been to use locks and similar constructs in order to locally synchronize data accesses. The major drawback with this is that it demands a lot of effort from the programmer in order to achieve good parallelization, even of simple tasks. Furthermore, the likelihood of concurrency-related errors such as deadlocks, data races and livelocks naturally becomes higher. A simple example of a data race is illustrated in Figure 1.1a where `thread1` and `thread2` are two threads with access to a common variable `x`. Depending on the execution order, `x` will have a different final value.

Concurrent deterministic programs are concurrent programs which always produce the same results for the same inputs. This is attractive, since it provides reproducible computations. Deterministic-by-design concurrent programming models are systems which guarantee this property at compile time and thus significantly simplify concurrent programming. This greatly eases the effort of the programmer. As an example, if in a similar setting to the program in Figure 1.1a, say you were only interested in the maximum value of `x`. You could make the program deterministic by introducing a special `maxWrite(x,v)` operation. `maxWrite` takes a variable `x` and a value `v` and sets the new value of `x` to the maximum of its current value and `v`. The modified program can be seen in Figure 1.1b. It is not hard to see that this is deterministic. Now, by somehow statically enforcing that all writes to shared variables should be through this special write operation, we could at compile time guarantee that a program has deterministic results.

```
var x: Int = 0

thread1 = () => {
  /* doing cool stuff */
  ...

  x = 1

  /* doing more cool stuff */
  ...
}

thread2 = () => {
  /* doing wicked stuff */
  ...

  x = 2

  /* doing more wicked stuff */
  ...
}
```

(a) Example of a data race.

```
var x: Int = 0

thread1 = () => {
  /* doing cool stuff */
  ...

  maxWrite(x,1)

  /* doing more cool stuff */
  ...
}

thread2 = () => {
  /* doing wicked stuff */
  ...

  maxWrite(x,2)

  /* doing more wicked stuff */
  ...
}
```

(b) The deterministic program.

Figure 1.1: Program examples.

A novel deterministic-by-design model called LVish [12, 13] has been introduced. This generalizes the `maxWrite` from above. It uses data types that fulfill the condition of being part of a lattice. More explicitly, this means the shared data has some partial order defined on its values, and a so-called join operation (a generalized maximum operation) defined for all value pairs. By only allowing writes to the data in form of this join operation, you can achieve concurrent determinism with very high parallelism using event-based computation. This is also the approach of the Reactive Async project [8], a Scala-based framework inspired by LVish. Both LVish and Reactive Async have shown potential to speed up computations for algorithms concerning graphs and static code analysis [13, 8].

The main problem with Reactive Async as of now, is that it does not include any construct to ensure that data races between event handlers do not occur. There are also more fundamental problems with callback spawning, which could also lead to non-determinism. Furthermore, there are problems with LVish and its proof of determinism. Therefore, the main concern of this thesis is to analyze and find a solution to these problems.

1.1 Research Question & Goal

This thesis aims to construct a formal model which could later be adopted by future revisions of Reactive Async. Constructing a model similar to that of LVish but in an object-oriented setting, the final goal will be to mathematically prove a form of determinism for the system.

Doing this will hopefully give us an answer to the research question:

> How can we enforce a deterministic concurrent model similar to LVish in an object-oriented setting?

1.2 Methodology

We use the theory of programming language semantics and type systems to build a simple object-oriented language. The type system incorporates theory that relates to the object capability model from computer security, also utilized in previous work by Haller and Loiko [9].

After defining the property of a well-typed program state, we prove that this is preserved during execution. Furthermore, we prove that an execution of a well-typed program will not get stuck unless there is a null-pointer exception. Finally, we prove a form of determinism for the system.

1.3 Contributions

With this work, the following technical contributions are made:

- Identification and exemplification of problems with previous work on deterministic concurrency.
- Formalization of a core object-oriented language for deterministic concurrency.
- Design and formalization of an accompanying type system.
- A proof of soundness for the core language and type system.
- A proof of quasi-determinism, a form of determinism, for our formalized system.

1.4 Report Structure

In Chapter 2 some technical background is described. This includes some mathematical definitions, an introduction to programming language formalization and an informal description of the object capability model. Chapter 3 describes some selected related work, upon which our model builds. In Chapter 4 we describe and exemplify some problems with existing deterministic-by-design concurrent systems. In Chapter 5 we introduce our core language and formalization. Chapter 6 describes the theorems and proofs of soundness and quasi-determinism. Finally, in Chapter 7 we discuss extensions and implementation of our formal system as well as ethical aspects and sustainability. We finally include a short conclusion.

Chapter 2

Background & Preliminaries

This chapter is an introduction to some mathematical concepts as well as some language and type system theory. In Section 2.1, we will initially give some mathematical definitions. In Section 2.2, some basics of language syntax will be explained. Section 2.3 gives a brief introduction to programming language semantics. Section 2.4 explains some basics of programming language type systems. Finally, Section 2.5 explains the object capability security model.

2.1 Mathematical Preliminaries

Definition 2.1. Let $(V, \sqsubseteq)$ be a partial order. Then, the *least upper bound* of two elements $v, v' \in V$ is a an element $u \in V$ such that

$$v \sqsubseteq u \qquad v' \sqsubseteq u$$

and for all $v'' \in V$ such that $v \sqsubseteq v'', v' \sqsubseteq v''$

$$u \sqsubseteq v''$$

It is easy to show that if a least upper bound exists, it is unique.

Definition 2.2. A *join-semilattice* is a partially ordered set $(\mathscr{L}, \sqsubseteq)$ such that for any two $l, l' \in \mathscr{L}$, there is a least upper bound $u \in \mathscr{L}$. u is also called the *join* of l and l' and we can write this as

$$u = l \sqcup l'$$

The greatest element of $(\mathscr{L}, \sqsubseteq)$ is called the top element, written $\top_{\mathscr{L}}$. If the join-semilattice also has a bottom (or least) element, written $\bot_{\mathscr{L}}$, it is called a *bounded* join-semilattice.

From now on we will refer to a bounded join-semilattice simply as a *lattice*.

Definition 2.3. which is A *partial function* or *partial map* f from A to B, written

$$f \in A \rightharpoonup B$$

is a function defined on some subset of A, i.e.

$$f \in A' \to B \qquad A' \subseteq A.$$

Notation. If a map $g \in A \to B$ is a bijection we write

$$g \in A \hookrightarrow B.$$

2.2 Language Syntax

A language syntax or grammar is a formal way to describe the syntactical structure of a programming language. It gives us a basis on which to build the language semantics and type systems, as will be clear later. This section will give a brief introduction to the Backus-Naur-form, and exemplify this with a simple programming language.

2.2.1 The Backus-Naur-form

Backus-Naur-form (BNF) is a very common way to describe the grammar of programming languages. BNF describes a grammar using rules of the form

$$s ::= \text{<expression>}.$$

The left side symbol s is refered to as a non-terminal. <expression> could also be a list of the possible forms of s, separated by |, i.e.,

$$s ::= \text{<expression1>} \mid \ldots \mid \text{<expressionN>}$$

The expressions can themselves contain non-terminals and terminals. Terminals are symbols which does not occur on the left side of any rule.

As an example, the While language can be written in BNF as described by Nielson and Nielson [19]. For reference we give a similar description here, with the addition of a very simple type annotation. The grammar is presented in Figure 2.1.

c	$::=$		Program code
		ε	Empty code
	$\mid$	$s; c$	Statement concatenation
s	$::=$		Program statement
		`var` x `:` τ	Variable declaration
	$\mid$	x `:=` e	Assignment
	$\mid$	`skip`	Skip
	$\mid$	`if` e `then` c `else` c	If branch
	$\mid$	`while` e `do` c	While loop
e	$::=$		Expressions
		x	Variable
	$\mid$	v	Value literal
	$\mid$	$e + e$	Addition
	$\mid$	$e * e$	Multiplication
	$\mid$	$e - e$	Subtraction
	$\mid$	$e = e$	Equality comparison
	$\mid$	$e \leq e$	$\leq$ comparison
	$\mid$	$\neg e$	Logical not
	$\mid$	$e \wedge e$	Logical and
τ	$::=$		Types
		Bool	Boolean type
	$\mid$	Int	Integer type
v	$::=$		Values
		n	Integer
	$\mid$	`true`	Boolean true
	$\mid$	`false`	Boolean false

Figure 2.1: Grammar of While

2.3 Small-step Operational Semantics

In order to prove properties of a programming language, you must be able to formally describe the "meaning", or semantics, of the program, i.e., to describe how programs statements modify state and what execution steps can be taken. A common approach to this is small-step operational semantics, also called structural operational semantics (SOS). Other approaches includes big-step operational semantics, also called natural semantics, and denotational semantics. The presentation here will focus on SOS since this can model concurrency well.

Small-step operational semantics define rules which can be used to derive single execution steps, also called transitions, between program states. We exemplify this with the While language defined in the above section. We let Var be the set of all variable names and Val the set of all values, which for the While language will be the integers $\mathbb{Z}$, and the boolean values `true` and `false`:

$$\mathrm{Val} = \mathbb{Z} \cup \{\mathtt{true}, \mathtt{false}\} .$$

In order to define rules for state transitions, a state must be defined. Therefore we make the following definition for the While language.

Definition 2.4. A *state* S for the While language is a value of the form

$$\langle V, c\rangle$$

where c is program code to be executed and V is a partial map,

$$V : \mathrm{Var} \rightharpoonup \mathrm{Val},$$

representing program memory.

We can now define rules for state transitions. All rules are of the form

$$\frac{\text{preconditions}}{S \to S'} \qquad (\textsc{Rule name})$$

This basically says *if the preconditions holds then we can step from S to S'*. If there are no preconditions we simply write

$$S \to S' \qquad (\textsc{Rule name})$$

$$\langle V, \mathsf{var}\ x : \mathsf{Int}; c\rangle \;\rightarrow\; \langle V[x \mapsto 0], c\rangle \qquad (\textsc{Whe-Decl1})$$

$$\langle V, \mathsf{var}\ x : \mathsf{Bool}; c\rangle \;\rightarrow\; \langle V[x \mapsto \mathtt{false}], c\rangle \qquad (\textsc{Whe-Decl2})$$

$$\langle V, x := e; c\rangle \;\rightarrow\; \langle V[x \mapsto \mathcal{E}[\![V]\!](e)], c\rangle \qquad (\textsc{Whe-Assign})$$

$$\langle V, \mathtt{skip}; c\rangle \;\rightarrow\; \langle V, c\rangle \qquad (\textsc{Whe-Skip})$$

$$\frac{\mathcal{E}[\![V]\!](e) = \mathtt{true}}{\langle V, \mathtt{if}\ e\ \mathsf{then}\ c_1\ \mathtt{else}\ c_2; c\rangle \;\rightarrow\; \langle V, c_1; c\rangle} \qquad (\textsc{Whe-IfTrue})$$

$$\frac{\mathcal{E}[\![V]\!](e) = \mathtt{false}}{\langle V, \mathtt{if}\ e\ \mathsf{then}\ c_1\ \mathtt{else}\ c_2\rangle \;\rightarrow\; \langle V, c_2; c\rangle} \qquad (\textsc{Whe-IfFalse})$$

$$\langle V, \mathtt{while}\ e\ \mathsf{do}\ c'; c\rangle \;\rightarrow\; \langle V, \mathtt{if}\ e\ \mathsf{then}\ (c'; \mathtt{while}\ e\ \mathsf{do}\ c')\ \mathtt{else}\ \mathtt{skip}; c\rangle \qquad (\textsc{Whe-While})$$

Figure 2.2: Small-step operational semantics for While

All rules for state transistions are defined in Figure 2.2 and most of them should be intuitive. For example, rules WHE-DECL1 and WHE-DECL2 executes a variable declaration by extending the state map V with x and a default value corresponding to the declared type. Rule WHE-ASSIGN executes a variable assignment by evaluating the expression e and mapping x to the result. Rules WHE-IFTRUE and WHE-IFFALSE take an if-statement and evaluate its expression e. If the result is `true` or `false`, the code of the corresponding branch is prepended to the rest of the program. We can also see that WHE-WHILE is just an expansion of a while loop into an if statement. Finally we note that we can not step from $\langle V, \varepsilon\rangle$, which means that this is a halting state.

The SOS rules of While heavily relies on the *expression evaluation function* $\mathcal{E}[\![\cdot]\!](\cdot)$. This is a function which takes a state map V and an expression e, and returns a value in Val. We define this recursively as follows in equation (2.1).

$$\begin{array}{lcl}
\mathcal{E}[\![V]\!](v) & = & v \\
\mathcal{E}[\![V]\!](x) & = & V(x) \\
\mathcal{E}[\![V]\!](e_1 + e_2) & = & \mathcal{E}[\![V]\!](e_1) + \mathcal{E}[\![V]\!](e_2) \\
\mathcal{E}[\![V]\!](e_1 * e_2) & = & \mathcal{E}[\![V]\!](e_1) * \mathcal{E}[\![V]\!](e_2) \\
\mathcal{E}[\![V]\!](e_1 - e_2) & = & \mathcal{E}[\![V]\!](e_1) - \mathcal{E}[\![V]\!](e_2) \\
\mathcal{E}[\![V]\!](e_1 = e_2) & = & \mathcal{E}[\![V]\!](e_1) = \mathcal{E}[\![V]\!](e_2) \\
\mathcal{E}[\![V]\!](e_1 \leq e_2) & = & \mathcal{E}[\![V]\!](e_1) \leq \mathcal{E}[\![V]\!](e_2) \\
\mathcal{E}[\![V]\!](e_1 \wedge e_2) & = & \mathcal{E}[\![V]\!](e_1) \wedge \mathcal{E}[\![V]\!](e_2) \\
\mathcal{E}[\![V]\!](\neg e_1) & = & \neg\mathcal{E}[\![V]\!](e_1)
\end{array} \tag{2.1}$$

Here we implicitly rely on that the operations $+, -, *, =, \leq$ are only defined for integer values, and that $\neg, \wedge$ are only defined for boolean values. If we encounter a state map V and an expression e where the corresponding integer or boolean operation above is undefined, the value for $\mathcal{E}[\![V]\!](e)$ is undefined. Since the definition of $\mathcal{E}[\![\cdot]\!](\cdot)$ is recursive, the occurrence of an undefined value should also propagate upwards. For example, if $\mathcal{E}[\![V]\!](e_1)$ is undefined then $\mathcal{E}[\![V]\!](e_1 + e_2)$ is undefined aswell. This is implicit in our definition.

Remark. Note that in [19] the conversion between the syntactical (numerals/boolean literals) and semantic (integers/boolean values) versions of values are explicit whereas here this conversion is implicit.

The evaluation function $\mathcal{E}[\![\cdot]\!](\cdot)$ together with the derivation rules in Figure 2.2 gives us a complete description of what steps are allowed. It is implicit in the description that if the evaluation function is undefined at some point in the derivation of a step, execution cannot progress and we get stuck.

2.4 Type Systems

A type system is a mathematical construct that consists of elements called types, and a set of rules that assign types to parts of a programming language, e.g., statements, variables and expressions. Type systems are most commonly used to prevent programming errors such as undefined data operations, e.g., feeding a data structure to a function for which it was not made. This can be enforced both using static checking at compile time or dynamic checking at runtime. Type systems are found in many modern programming languages such as Java, Scala, Haskell, Python and C++.

The focus here will be on static type systems. We will explain the basics using our previous example of the While language. Then some uses and extensions will be discussed.

2.4.1 A Type System for While

For the While language we have the type Int representing integer values, the boolean type Bool, and the special non-value type Void. We call the set of all types

$$\mathrm{Tpe} = \{\mathsf{Int}, \mathsf{Bool}, \mathsf{Void}\}\,.$$

Our typing rules will define a relation of the form

$$\Gamma \vdash r : \tau. \tag{2.2}$$

Here Γ is a *typing environment*, i.e. a partial map

$$\Gamma : \mathrm{Var} \rightharpoonup \mathrm{Tpe}.$$

r is either code or an expression and $\tau \in \mathrm{Tpe}$. Equation (2.2) basically reads: *r is typeable as τ under the typing environment Γ.* A program c is *well-formed* if

$$\emptyset \vdash c : \mathsf{Void}$$

where $\emptyset$ is the empty typing environment, i.e. $\emptyset(x)$ is undefined for all $x \in \mathrm{Var}$.

Notation. Sometimes you would like to remap a key of, or extend a partial map M. One common notation for this is

$$M[k \mapsto v],$$

which means that $M(l) = M[k \mapsto v](l)$ for all $l \neq k$ and $M(k) = v$. For typing environments we will also use the notation

$$\Gamma, x : \tau,$$

which is equivalent to $\Gamma[x \mapsto \tau]$. We will also see the notation

$$(x : \tau) \in \Gamma,$$

which is equivalent to $\Gamma(x) = \tau$.

$$\Gamma \vdash \varepsilon : \mathsf{Void} \qquad \text{(WHT-EMPTY)}$$

$$\frac{\Gamma(x) \text{ undefined} \qquad \Gamma, x : \tau \vdash c : \mathsf{Void}}{\Gamma \vdash \mathsf{var}\ x\ :\ \tau; c : \mathsf{Void}} \qquad \text{(WHT-DECL)}$$

$$\frac{(x : \tau) \in \Gamma \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash c : \mathsf{Void}}{\Gamma \vdash x\ := e; c : \mathsf{Void}} \qquad \text{(WHT-ASSIGN)}$$

$$\frac{\Gamma \vdash c : \mathsf{Void}}{\Gamma \vdash \mathtt{skip}; c : \mathsf{Void}} \qquad \text{(WHT-SKIP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash c_1 : \mathsf{Void} \\ \Gamma \vdash c_2 : \mathsf{Void} \qquad \Gamma \vdash c : \mathsf{Void}\end{array}}{\Gamma \vdash \mathtt{if}\ e\ \mathsf{then}\ c_1\ \mathtt{else}\ c_2; c : \mathsf{Void}} \qquad \text{(WHT-IF)}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash c' : \mathsf{Void} \qquad \Gamma \vdash c : \mathsf{Void}}{\Gamma \vdash \mathtt{while}\ e\ \mathsf{do}\ c'; c : \mathsf{Void}} \qquad \text{(WHT-WHILE)}$$

Figure 2.3: While program code typing rules.

Typing rules for program code are defined in Figure 2.3. Generally, While program code can only be typed as Void. Intuitively, this is because the execution of program code does not have a resulting value. Instead, it can only modify the state map V according to the semantics of the previous section. The rules for typing program code should not be hard to understand. For example, WHT-ASSIGN states that for code that starts with an assignment, the expression in the assignment must be of the same type as the variable. WHT-IF requires that the expression of an if-statement is typeable as Bool, and that both branches should be typeable as Void.

Finally the rules for typing expressions are found in Figure 2.4. The possible types for an expression are Int and Bool. The rules themselves should be quite self-explanatory, and reflects the fact that our binary operators are only defined for a subset of all values in Val.

2.4.2 Uses & Extensions

So what are type systems used for? As mentioned earlier, they are commonly put in place to prevent errors such as undefined data operations. For example, in Java you cannot feed a List to a method which

$$\Gamma \vdash n : \mathsf{Int} \quad \text{(WHT-NUM)} \qquad \Gamma \vdash \texttt{true} : \mathsf{Bool} \quad \text{(WHT-TRUE)} \qquad \Gamma \vdash \texttt{false} : \mathsf{Bool} \quad \text{(WHT-FALSE)}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{(WHT-VAR)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}} \quad \text{(WHT-ADD)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 * e_2 : \mathsf{Int}} \quad \text{(WHT-TIMES)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 - e_2 : \mathsf{Int}} \quad \text{(WHT-MINUS)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}} \quad \text{(WHT-EQ)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 \leq e_2 : \mathsf{Bool}} \quad \text{(WHT-LEQ)}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool}}{\Gamma \vdash \neg e : \mathsf{Bool}} \quad \text{(WHT-NOT)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash e_1 \wedge e_2 : \mathsf{Bool}} \quad \text{(WHT-AND)}$$

Figure 2.4: While expression typing rules.

is denoted to take an Integer as an argument. This will result in a compile time error.

2.4.3 Preservation & Progress

A formal type system can furthermore be used to prove properties like *preservation* and *progress* for programs that are properly type checked. These are important properties since they can beforehand ensure that programs terminate properly, or at least that if it terminates erroneously, it must be due to , e.g., a null-pointer exception. Progress and preservation together is often refered to as *soundness* of a type system, and to prove this is a standard approach. For a more in depth explanation, see [21].

As an example, we can state preservation and progress properties of While as follows. Let $\Gamma(V)$ be defined as

$$\Gamma(V)(x) = \begin{cases} \mathsf{Int} & \text{if } V(x) \in \mathbb{Z} \\ \mathsf{Bool} & \text{if } V(x) \in \{\mathtt{true}, \mathtt{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Proposition 2.5 (Preservation of While)**.** *Let*

$$\langle V, c\rangle \ \rightarrow \ \langle V', c'\rangle \qquad \Gamma(V) \vdash c : \mathsf{Void}.$$

Then

$$\Gamma(V') \vdash c' : \mathsf{Void}$$

Proposition 2.6 (Progress of While)**.** *Let $\langle V, c\rangle$ be a state and let*

$$\Gamma(V) \vdash c : \mathsf{Void}.$$

Then either $c = \varepsilon$ or there is a state $\langle V', c'\rangle$ such that

$$\langle V, c\rangle \ \rightarrow \ \langle V', c'\rangle.$$

2.4.4 Extensions

The example type system for While is of course very simple. This is mostly due to the simplicity of the language itself. More complicated languages have more complicated type systems, and a type system is oftenmost designed together with the language itself.

One common construct in type systems is *subtyping*. This introduces a a lattice based on a partial order relation between types called the subtyping relation, denoted $<:$. It is oftenmost used to say that *if $\sigma <: \tau$, then values of type σ can be used in the same way as values of type τ*. This is the case for Java, where a class C can extend a class D. This means that the fields and methods of D are also availible in C. We will see an example of how subtyping can be used in Chapter 5.

In Chapter 5 the typing relation will also include an effect a. In short this means that the typing rules are more or less strict depending on the value of a.

2.5 The Object Capability Model

The object capability (OCAP) model is a computer security model that was introduced in 1966 by Dennis and VanHorn [5]. In this model, the access rights to an object are equivalent to a reference to the object itself. This means is that if we somehow acquire a reference to the object, we are free to use it as we see fit. While this might seem like a severe loosening of security, implementing access restrictions similar to other security models can be done using proxy-like objects [17].

In the OCAP model, we formalize the concept of an object. This could be of two types: either a primitive object or an instance. A primitive object is, for example, data such as a character string or number. In a setting where communication with some outside world is modeled, a primitive object could also be, for example, a device. An instance contains state and code. For example, a class object in an object oriented language could be modeled as an instance.

Instances can only interact by sending messages over references. This implies that for an instance to interact with another one, it needs a reference to the other. Continuing the example with the class object above, a message corresponds to a method call. In essence, we send a command (specify the method), together with some information (the method parameters). The method can only be called if we have a reference to the corresponding class object beforehand.

We can model references and objects using a directed graph, called a object reference graph. If object A has has a reference to object B, then edge (A, B) is part of the graph. In the OCAP model, references can only be obtained in four ways:

1. Initial connectivity: The system must have some initial references in the connectivity graph.
2. Parenthood: If some instance A creates an object B, at creation it has a reference to B.
3. Endowment: If some instance A has access to B and creates C with a reference to B, then C has a reference to B.
4. Introduction: If instance A sends a message to (calls a method on) instance B, B gets all references A sends in the message (gives as parameters).

The above four rules imply that two disjoint components in a reference graph can never become connected [17].

A language implementing the OCAP model is called an OCAP language. Most common object-oriented languages are not OCAP. This is partly because they allow global references, i.e., objects referable from all instances.

Chapter 3

Related Work

In this chapter, a few approaches to deterministic concurrency as well as other related works will be summarized. Section 3.1 quickly introduces a few existing models of deterministic concurrency. In the rest of the chapter we describe some existing work, upon which the model presented in Chapter 5 will build on: Section 3.2 describes the LVars system and its extension LVish. Section 3.3 describes Reactive Async, a programming framework inspired by LVish. In Section 3.4, the LaCasa system is introduced. Finally, the concept of spores is briefly introduced in Section 3.5.

3.1 Approaches to Concurrent Determinism

One of the first works on concurrent determinism was made by Kahn [11]. They suggest a simple system of computing units connected by channels. A channel is a FIFO construct to send data of a specified data type. Each computing unit is a sequential program with separate memory, for which data channels are registered as either incoming or outgoing. It has access to the incoming and outgoing channels through the `wait` and `send` operations. `wait` takes an incoming channel identifier, and pops and returns the first object in the channel. If no objects are currently in the channel, `wait` blocks until a data object is sent. `send`, as the name suggests, sends a data object on an outgoing channel. At most one computing unit can use a channel as outgoing and incoming channel respectively. A computing unit can be interpreted as a tuple of deterministic functions of the incoming channel data sequences, generating sequences for outgoing data channels. This in turn makes a system of these computing units deterministic [11].

In in an article Bocchino et al. [3] argues that concurrent and parallel programs should be deterministic by default. This is of course in contrast to the common approach that concurrency is handled directly with locks and similar constructs, which leaves no guarantees of determinism other than vague promises from the programmer. The solutions suggested by Bocchino et al. [3] includes language-based approaches as well as automatically parallelizing compilers and runtime-based approaches. They identify language-based approaches as the most promising, and exemplify this with their experimental language Deterministic Parallel Java (DPJ) [6]. As the name suggests, DPJ extends Java with an object-oriented effect system. Using class annotations this divides the heap into sets of memory locations called regions, and stipulates that all methods must be annotated with what regions are written and read. This allows them to check the program at compile time and give guarantees of determinism. DTJ also allows use of non-determinism. This, however, needs to be explicitly stated in code [6, 2].

FlowPools [22] is a deterministic concurrent programming model using the collection abstraction together with the dataflow programming model. The result is a collection-like data structure that can handle asynchronous adds of data, together with dataflow operations being both executed and registered concurrently. The model also guarantees determinism using a sealing operation, effectively freezing the collection and allowing no more writes. This, however, still requires the programmer to manually freeze the result, which has to be done with care since any subsequent attempt of data modification will result in an error. This is a problem since we would like to place as much of the burden of concurrency control in the model itself.

Write-once data structures, also called IVars, is also a concept that has been explored thoroughly. It was first introduced by Nikhil, Pingali, et al. [20]. An IVar is a data type which allows at most one write through a put operation. After the first one, any subsequent put results in an error. There is also a get operation, which blocks until a value has been put into the IVar. Some models guaranteeing determinism using language constructs, and IVars or similar structures are for example the Intel Concurrent Collections for Haskell [18] and the Haskell Par monad [15]. Both of these rely on the strictly functional nature of Haskell together with the semantics of IVars to ensure determinism.

3.2 LVars

LVars [12] is a programming model that was introduced as a solution to the problem of guaranteed deterministic concurrent programs. It generalizes the concept of IVars [20] with the ability to write more than once but limiting update operations to being monotonic. In LVars, all variables shared between threads are part of a programmer specified lattice, and all writes are done through a join operation of the old and new values. This ensures that writes are monotonic and thus that they commute [12].

3.2.1 Stores & Lattice

At the foundation of the LVars system lie lattices. The values resulting from a computation are elements from a lattice Lat, specified by the programmer. These lattice values are stored in a *store*. This is a set of pairs, consisting of a location and a lattice element. For one store location, there is at most one value. Letting Loc be the set of locations, a store Σ can be represented using a partial map

$$\Sigma \in \mathrm{Loc} \rightharpoonup \mathrm{Lat}.$$

3.2.2 LVars Operations

The LVars model supports three main operations [12].

- Extending the store with a new location. This obtains a fresh location and sets its value to $\bot_{\mathrm{Lat}}$, the bottom element of Lat. Given a store Σ and a fresh location r, the resulting store is $\Sigma[r \mapsto \bot_{\mathrm{Lat}}]$.
- Updating a store location, also called a *put* operation. This operation takes a store location r and a lattice value l. Given a store Σ, the resulting store is $\Sigma[r \mapsto l \sqcup S(r)]$. To ensure determinism in conjunction with the next operation, any put operation that takes a store location to $\top_{\mathrm{Lat}}$ results in an error.

- An LVar read operation, also referred to as *get*. This operation is specified with a store location and a threshold set, i.e. a set of lattice values. The operation blocks until the store location has passed one of the values in the threshold set, according to the ordering $\sqsubseteq$ of the lattice. Upon passing a threshold, the get operation returns the threshold value. In order to ensure determinism, the elements in the threshold set are required to be mutually *incompatible*. Two elements l_1, l_2 are incompatible if

$$l_1 \sqcup l_2 = \top_{\mathrm{Lat}}$$

where $\top_{\mathrm{Lat}}$ is the top element of Lat.

3.2.3 LVish: Extending LVars

LVish [13] is an extension of LVars. It introduces a new operation called freezing, a system to create dependencies between store locations, and the concepts of quasi-determinism and quiescence. In LVish, the store is modified such that every location also has an associated *freeze bit* with a default value of false. The *freeze* operation takes a location and changes its associated bit to true. Whenever a freeze bit is true, it prevents the associated lattice value, i.e., store location, from being changed. If a put operation tries to modify a freezed store location, the computation ends up in the error state **error**.

LVish also permits the programmer to create dependencies between store locations. To do this, the programmer specifies a store location, a set of threshold lattice values, and a callback function. The callback is scheduled to be executed as soon as the given location passes a threshold value. Upon execution, the callback is given access to the threshold value passed. Unlike the threshold sets for the get operation, the elements do not have to be incompatible.

With the introduction of freezing, a program can take different execution paths, where one halts in **error** and the other halts in a non-erroneous state. It is thus obvious that the LVish system is no longer deterministic. Therefore the authors introduce *quasi-determinism*, a weaker form of determinism. Instead of requiring the same computation result for all execution paths, quasi-determinism only requires that results be the same for non-erroneous halting states.

Finally, the LVish system makes use of a concept called *quiescence*, in order to decide when to freeze a location. In short, a computation

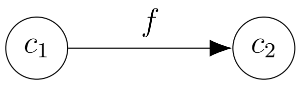

Figure 3.1: Simple Reactive Async dependency graph.

reaches quiescence when no more changes to the store are expected to take place. For example, this could be when there are no more callback threads running.

The proof of quasi-determinism for LVish presented by Kuper et al. [14] is flawed. This will be touched upon further in chapter 4.

3.3 Reactive Async

Reactive Async [8] is a Scala language programming framework inspired by LVish. It is based around objects called *cells*. Cells are similar to the store locations from LVish: they hold a value from a lattice, the value is updated through a join operation, and you are able to create dependencies between them using callbacks.

Similarly to LVish, the cells are updated through a put operation. The cells and dependencies can be illustrated as a graph. Figure 3.1 is a very simple example of a dependency graph. There are two cells c_1 and c_2, and a dependency callback f between them. This should be interpreted as c_2 being dependent on c_1. f is a function which takes a lattice value and returns a new lattice value. As soon as the lattice value of c_1 is updated to a new lattice value l, the callback $f(l)$ is scheduled. If $f(l)$ returns a lattice value l', c_2 is updated with l' though a put operation, equivalent to the put of LVars above. The return value of $f(l)$ can also be a special value `NoOutcome`, representing no put operation should be done on c_2.

If a programmer is not careful, there are several aspects of Reactive Async which can make a computation non-deterministic. This will be discussed further in Chapter 4.

3.4 LaCasa

LaCasa [9] is a programming model for ensuring determinism in an actor model [10] setting. It draws parallels to the OCAP model to ensure deterministic execution of threads. In this section we give a brief introduction to the type system of LaCasa, and the intuition behind it.

The formalization of LaCasa is based on a simple object oriented language. Haller and Loiko [9] present several models, the last of which models concurrency with several parallel processes. In order to ensure determinism, all data interchanged between processes must be encapsulated in *boxes*. A box is a container for a class reference with some accompanying methods to access and perform calculations with the encapsulated object. The type system ensures that all calculations made with objects encapsulated in a box are isolated, i.e., that at any given time, at most one thread can access the data and perform calculations with it.

A big part of ensuring isolation is requiring all objects encapsulated in boxes are of classes typed as *ocap*. This basically means three things.

1. Methods never access global global objects.

2. All field types are ocap.

3. Methods can only create new instances of ocap classes.

To ensure this holds in the presence of a subtyping relation, superclasses are furthermore required to be ocap [9].

These rules ensure that all references acquired by an ocap class object will have been explicitly given to it. The ocap classification for classes will later be incorporated in the system presented in this thesis.

3.5 Spores

In recent years there has been an increasing demand for distributed and concurrent data processing. Highly concurrent or distributed frameworks like Akka [1] and Spark [24] have become more and more popular. As their popularity grows, concerns have been raised over hazards that relate to closures, one of the basic and most important constructs in modern programming languages. A closure is basically the notion of capturing and using an externally defined variable in a function definition. In a concurrent or distributed setting, it could be fatal if we capture, e.g., mutable or non-serializable references [16].

An experimental solution for Scala is *spores* [16]. Basically, a spore allows the programmer to clearly declare what external objects are being captured by a closure and, furthermore, limit what types that can be captured. The use of spores can be forced by annotating code with the spore type instead of the normal Scala function type.

Chapter 4

Challenges of Deterministic Concurrency

As noted in the introduction, achieving a high level of parallelization and ensuring determinism is a difficult task. The inherent nondeterminism of concurrent operations leaves the programmer with the hard task of considering all paths of execution and making sure that they all lead to the same result.

Systems like Reactive Async and LVish both have the ambition of moving this burden off the programmer and onto the programming model and type system. Both however have problems. In this chapter, we describe these problems in an informal way. As a running example, we will use the lattice of integers $(\mathbb{Z}, \leq)$, where $\leq$ is the standard less-than-or-equal comparison.

4.1 Syntax Explanation

In order to construct examples, we will use a language akin to Reactive Async. This will have a syntax similar to Scala function syntax [23]. In this syntax we write a callback function as in Figure 4.1. This takes a lattice value `l` as an argument. In our example callback code we will use common programming language constructs such as conditionals (**`if`**-statements) and field accesses. Furthermore we will use the `put` and `freeze` operations, which will be explained below.

```
(l) => {
  /* callback code here */
}
```

Figure 4.1: Example of callback syntax.

4.1.1 Cells

An integral part of both LVish and Reactive Async, is a data type that holds the resulting value of a computation. Callbacks also need to be registered somehow. Although the implementations differ, we will use a unified representation for our examples below. All examples should be reconstructable in their respective setting of Reactive Async or LVish.

We call our data type a *cell*, in alignment with Reactive Async. A cell holds two values: A lattice element (also called cell value below) and a freeze bit. This is similar to an LVar in LVish. Registration and spawning of callbacks will be explained for each example separately.

4.1.2 The `put` and `freeze` Statements

There are two main operations used in our examples below. We can see an example of both in Figure 4.2. The freeze statement `frz(...)` takes one argument of cell type and freezes the cell specified, i.e., sets its associated freeze bit to true. Furthermore, it returns the associated cell value. The put statement `put(...)` takes two arguments, one of cell type and one lattice value. This updates the cell value with the specified one using the join operation, similar to how an LVar is updated [12]: if the freeze bit is true and the join result is not the same as the current value, the put operation results in an error. Otherwise, the cell value is updated to the join result and execution continues. The return value of `put` is the cell reference itself. The behaviour of both `freeze` and `put` mimic their LVish counterpart closely.

```
x = put(c, l)
```

(a) Example put operation. We put the lattice value assigned to l into cell c. This may result in an error.

```
x = frz(c)
```

(b) Example freeze operation. We freeze the cell c and assign its cell value to variable x.

Figure 4.2: Example cell operations.

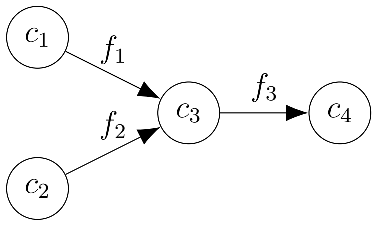

Figure 4.3: Reactive Async dependency graph.

4.2 Problems of Reactive Async

For Reactive Async, many put operations are implicit due to the programming framework structure [8]. In order to reduce complexity, however, all put operations in the examples below will be explicit.

Reactive Async has two main problems. To highlight these we use a dependency graph similar to the one in Figure 3.1. Figure 4.3 describes a dependency graph which is a little more complex. Here we have three callback functions f_1, f_2 and f_3 registered to cells c_1, c_2 and c_3 respectively. The arrows annotated with the above callbacks indicate to what cell the corresponding function will make a put operation.

4.2.1 Callback Spawning Semantics

Callbacks in Reactive Async are spawned as follows: Let a callback f be registered to a cell c, and say the cell value of c changes to l. Then a new callback thread running $f(l)$, i.e., function f with parameter l, will be spawned.

```
(l) => {
  Global.someInt = 1
  put(c3,0)
}
```

(a) Code of f_1.

```
(l) => {
  if (Global.someInt == 1)
    put(c3, 1)
  else put(c3,0)
}
```

(b) Code of f_2.

Figure 4.4: Two callback functions sharing state.

4.2.2 Problematic State Sharing

The first problem of Reactive Async is that it is not prohibited for callbacks to share mutable state. To exemplify this let f_1 and f_2 from above be as in Figure 4.4. We ignore f_3 and c_4 for now. Say both f_1 and f_2 are scheduled to run and that the initial value of `Global.someInt` is 0. The lattice is $(\mathbb{Z}, \leq)$ and we let the initial cell value of all cells be 0. It should be clear that depending on the order in which f_1 and f_2 are scheduled to execute, c_3 will have different final cell values. If f_1 runs first, c_3 will have a final cell value of 1. If f_2 runs first, c_3 will have a final value of 0. This is due to that the callbacks share mutable state through the `Global` object, and that put operations updates a cell using the join operation.

It would seem that restricting direct access to any global object would be sufficient to rectify this problem. However, as the example functions of Figure 4.5 shows, this is not the case. Here we instansiate new class objects which in turn have access to a global object. This will lead to the same problem, even though we never access the global object directly within the callbacks.

The problem can, however, be remedied using the OCAP model in a similar manner to LaCasa. By ensuring that callback threads are unconnected in the object reference graph, we ensure that they do not share any state. We do this by ensuring that for each pair of threads, the intersection of their connected components in the reference graph only contains cell objects. The sharing of cells is safe since it should be impossible for any two running callback threads to transmit information through a cell object. In Chapter 5, the type system is used to enforce this isolation of threads.

```
(l) => {
  val x = new C
  x.setGlobal
  put(c₃,0)
}
```

(a) Code of f_1.

```
(l) => {
  val x = new C
  if ( x.isGlobalSet )
    put(c₃, 1)
  else put(c₃,0)
}
```

(b) Code of f_2.

```
class C {
  ⋮
  setGlobal : Unit = {
    Global.someInt = 1
  }

  isGlobalSet : Boolean = {
    if ( Global.someInt == 1 )
      true
    else
      false
  }
  ⋮
}
```

(c) Code of class `C`, which is instansiated in the two callbacks.

Figure 4.5: Two callback functions sharing state through a reference.

```
(l) => put(c3,2)
```

(a) Code of f_1.

```
(l) => put(c3,3)
```

(b) Code of f_2.

```
(l) => {
  if (l == 2)
    put(c4,1)
  else put(c4,0)
}
```

(c) Code of f_3.

Figure 4.6: Simple callback functions breaking determinism using callback spawning semantics of Reactive Async.

4.2.3 Problematic Callback Spawning Semantics

Remedying the above problem does not, however, yield a deterministic system. There is a flaw inherent in how Reactive Async decides to spawn a callback thread. A thread is only spawned when a cell value actually changes, which means that all put operations does not necessarily spawn a callback. To see that this is a problem, let f_1, f_2 and f_3 from Figure 4.3 be as in Figure 4.6. As before, let the lattice be $(\mathbb{Z}, \leq)$ and all cells have an initial cell value of 0. Furthermore let both f_1 and f_2 be scheduled to run.

Say f_1 is scheduled to run before f_2. The result of running f_1 will be a put of 2 to cell c_3. Since $2 \sqcup 0 = 2$, the cell value of c_3 will be changed to 2. This will spawn the callback $f_3(2)$. This callback will result in a put of 1 to c_4, i.e., c_4 will have its cell value updated to 1 by $1 \sqcup 0 = 1$. Running f_2 now results in the value of c_3 being updated to 3 which will spawn the callback $f_3(3)$. The resulting put of 0 to c_4 will not update c_4 since $0 \leq 1$. The final cell value of c_4 in this case will be 1.

Now let f_2 execute before f_3. This results in a put of 3 to c_3, which results in c_3 being updated to 3. The callback $f_3(3)$ will be spawned and result in a put of 0 to c_4. Since $0 \sqcup 0 = 0$ this will not change the value of c_4. Now, if f_1 executes, this will result in a put of 2 to c_3. Since the cell value of c_3 is already 3 and $2 \sqcup 3 = 3$, this will not result in an update of c_3. Thus the callback $f_3(2)$ is never spawned. The final cell value of c_4 for this execution order is 0, clearly different from before.

The problem of callback spawning can be remedied with the use of the threshold sets of LVish. Instead of only spawning a callback thead

Callback	Cell	Threshold Set
f_1	c	$\{1\}$
f_2	c	$\{1\}$

Table 4.1: Cell registration and threshold sets for f_1 and f_2.

```
(l) => {
  x = frz(c₁)
  if ( x == 0 )
    put(c₂, 1)
}
```

(a) Callback code for f_1.

```
(l) => {
  x = frz(c₂)
  if ( x == 0 )
    put(c₁, 1)
}
```

(b) Callback code for f_2.

Figure 4.7: Callback code to break determinism in LVish.

when a cell value changes, a callback will be spawned for each lattice value passed in the specified threshold set.

4.3 A Problem with LVish

As hinted in the end of Section 3.2, the proof of quasi-determinism of LVish is flawed. While this leaves the quasi-determinism of LVish an open problem, this is not really interesting. This is due to the fact that by adding a simple if-statement to the language, we can easily construct a counterexample.

4.3.1 Callback Spawning Semantics

The callback thread spawning semantics of the below example are as follows. Let c be a cell, f a callback registered to c with the associated threshold set T. If the cell value of c has passed some $l \in T$ according to the lattice ordering $\sqsubseteq$, the callback $f(l)$ will be spawned eventually. Furthermore, l will be removed from T. This is equivalent to the semantics of LVish.

Execution point	c_1 cell val	c_1 frz bit	c_2 cell val	c_2 frz bit
f_1 start	0	false	0	false
f_1 termination	0	true	1	false
f_2 start	0	true	1	false
f_2 termination	0	true	1	true

(a) f_1 executes before f_2.

Execution point	c_1 cell val	c_1 frz bit	c_2 cell val	c_2 frz bit
f_2 start	0	false	0	false
f_2 termination	1	false	0	true
f_1 start	1	false	0	true
f_1 termination	1	true	0	true

(b) f_2 executes before f_1.

Table 4.2: Two different execution paths contradicting LVish quasi-determinism.

4.3.2 LVish Freezing Problem

Say we have two cells c_1 and c_2, and two callbacks f_1 and f_2. Let the lattice be $(\mathbb{Z}, \leq)$, let the initial cell values of both cells be 0 and let both associated freeze bits be false. The callbacks f_1 and f_2 are registered to some auxilliary cell c according to Table 4.1. The threshold sets are not really important for this example, but are still included for completeness.

Now let f_1 and f_2 be as in Figure 4.7. Furthermore, let both $f_1(1)$ and $f_2(1)$ be scheduled (due to some put of an integer value greater than 1 to auxilliary cell c).

If f_1 executes before f_2, we end up with an execution as described by Table 4.2a. It is clear that we never get an error since we never make a put to a frozen cell. If f_2 executes before f_1, we can similarly describe the execution with Table 4.2b. Clearly the results differ and are non-erroneous, a non-quasi-deterministic result.

This shows that as soon as we introduce the if-statement, a basic part of any programming language, we can easily break quasi-determinism. The problem is inherent in the freeze statement. By freezing a variable we can attain information about the value and freeze bit of another cell. In the example above, by freezing cell c_1 (c_2) and inspecting its value we attain information about whether c_2 (c_1) has been frozen or not. By conditioning on this information, we can skip a put operation that would otherwise lead to an error.

A simple solution to this problem is to simply remove the capability of freezing a cell. This is the approach of the system presented in Chapter 5. It should however be possible to add other forms of freezing, which will be discussed in Chapter 7.

Chapter 5

Core Language

In this chapter, a basic core language is introduced. It will build a lot upon the core language of LaCasa and incorporate features of LVish. In the end, we will have an object-oriented language with many of the features of LVish, accompanied by a type system enforcing OCAP properties, much like the system of LaCasa.

5.1 Syntax

The language which we shall call Reactive Async Core Language (RACL) is very similar to the core language from LaCasa. Many expressions are the same, except for a few removals and additions. The language grammar in BNF can be found in Figure 5.1. It is a simple object-oriented language parameterized on the lattice $(\mathscr{L}, \sqsubseteq)$.

Looking at Figure 5.1 we can see that a program p consists of a sequence of class definitions $\overline{cd}$, a sequence of variable declarations $\overline{vd}$ and a term t. A class definition cd consists of a name specifier C, inheritance specifier D, field declarations $\overline{fd}$ and method definitions $\overline{md}$. Variable and field declarations both have the same form: they consist of a name specifier f and a type τ. Method definitions are also standard, with one thing to note being that all methods takes exactly one input. More complicated inputs can be constructed using a container class.

The type specifiers σ and τ can take on the values as specified. Note that we have the special Cell and $\mathcal{L}$ types, which are meant to represent the cells of Reactive Async and lattice values respectively. Types will be discussed more in Section 5.3.

p	$::=$	$\overline{cd}\,\overline{vd}\,t$	Program
cd	$::=$	$\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\,\overline{md}\}$	Class
vd, fd	$::=$	$\texttt{var}\ f\ :\ \tau$	Variable/Field
md	$::=$	$\texttt{def}\ m(x : \sigma) :\ \tau = t$	Method
σ, τ	$::=$		Types
		C, D	Class types
	$\mid$	Cell	Cell type
	$\mid$	$\mathcal{L}$	Lattice value type
t	$::=$		Terms
		x	Variable
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ t$	Let binding
e	$::=$		Expression
		l	Lattice value
	$\mid$	$\texttt{null}$	Null reference
	$\mid$	x	Variable
	$\mid$	$x.f$	Field selection
	$\mid$	$x.f = y$	Field assignment
	$\mid$	$\texttt{new}\ C$	Class instance creation
	$\mid$	$\texttt{new}\ \texttt{Cell}$	Cell instance creation
	$\mid$	$x.m(y)$	Method invocation
	$\mid$	$x\ \texttt{put}\ y$	Cell value update
	$\mid$	$\texttt{when}\ x\ \texttt{pass}\ y\ \texttt{then}\ (\overline{cap}, z \Rightarrow t)$	Dependency creation
cap	$::=$	$x = y$	Variable capture

Figure 5.1: Grammar of RACL

As in LaCasa, the terms of RACL are written in A-normal form (i.e. every subexpression is named). Most of the expressions should be self-explanatory. However, note for example that we have a separate instance creation expression for cells, an expression for updating the value of a cell, as well as an expression for creating dependencies between cells. The dependency creation expression is probably the most interesting since it mimics the syntax of spores [16]. All captured variables are clearly stated in the sequence of captures $\overline{cap}$.

5.2 Semantics

In this section a small-step operational semantic of RACL will be introduced. First a brief overview is made, and then a few of the more interesting or non-standard execution rules will be explained.

In short, the definitions in this section says that the state of the execution of a RACL program consists of a *heap* and a *thread set*. The heap is represented by a partial map from object identifiers to objects. The thread set is a set of threads, each of which consisting of call frame stacks. Each call frame holds a local variable environment and a term to be executed. Steps between states can be made according to rules on either frame, frame stack or thread set level. All steps taken on lower levels are propagated to thread set level using auxilliary rules.

5.2.1 Semantical Definitions

We first make a few definitions.

Definition 5.1 (Basic Definitions)**.** We define the following:

- We let Var be the set of all allowed variable names.
- We let $\mathscr{F}$ be the set of all allowed field names.
- We let $\mathscr{L}$ be the set of all lattice elements.
- We let $\mathcal{O}$ be the set of all object identifiers.
- We let $\mathcal{D}$ be the set of all thread identifiers.
- We let `null` be the special null value.

- We let $\mathscr{V}$ be the set of all possible runtime values

$$\mathscr{V} = \mathscr{L} \cup \mathcal{O} \cup \{\texttt{null}\}\,.$$

- We let OStat be the set of OCAP statuses

$$\mathrm{OStat} = \{ocap, \epsilon\}\,.$$

Definition 5.2. A *local environment* L is a partial map

$$L \in \mathrm{Var} \rightharpoonup \mathscr{V}.$$

Definition 5.3 (Heap Objects)**.** We let Obj be the set of all *heap objects*, i.e., objects of the form

$$\langle C, FM\rangle \text{ or } \langle \mathsf{Cell}, DEP, l\rangle.$$

$\langle C, FM\rangle$ is a *class object* where C is a class name. The *field map* FM is a partial map

$$FM \in \mathscr{F} \rightharpoonup \mathscr{V}.$$

In $\langle \mathsf{Cell}, DEP, l\rangle$ is a *cell object* where $l \in \mathscr{L}$. The *dependency set* DEP is a set of elements of the form

$$(l', (L_{\text{env}}, z \Rightarrow t))^{\iota}$$

where $l' \in \mathscr{L}$, L_{env} is a local environment, $z \in \mathrm{Var}$, t is a term and $\iota \in \mathcal{D}$ is a unique thread identifier. The uniqueness of ι is global to the state of a program.

Definition 5.4. A *heap* H is a partial map

$$H \in \mathcal{O} \rightharpoonup \mathrm{Obj}.$$

A heap must always contain the global object from definition 5.9.

Definition 5.5. A *frame* F is an object of the form

$$\langle L, t\rangle^{s}$$

where L is a local environment, t is a term and $s \in \mathrm{Var}$ is a *return tag*.

A *frame stack* FS is a finitely large stack of frames. We can write all of these recursively as either the empty stack $FS = \varepsilon$, or as $FS = F \circ GS$ where GS is also a frame stack.

A *thread set* P is a finite set

$$P = \left\{FS_i|^{\iota_i}_{a_i}\right\}^n_{i=1}$$

of non-empty frame stacks FS_i tagged with a unique thread id $\iota_i \in \mathcal{D}$ and an OCAP status $a_i \in \mathrm{OStat}$. Note that P can be the empty set. We call a thread tagged with *ocap* an *ocap thread*

Definition 5.6. We define a *state* to be either the error state **error** or a pair H, P, where H is a heap and P is a thread set.

Definition 5.7. The set of all valid types $\mathscr{T}_\mathrm{v}$ for a given program p consists of AnyRef, Null, Cell, $\mathcal{L}$ and all defined class types in p, $\mathscr{C}$. I.e.

$$\mathscr{T}_\mathrm{v} = \mathscr{C} \cup \{\mathsf{AnyRef}, \mathsf{Null}, \mathsf{Cell}, \mathcal{L}\}$$

Definition 5.8. The *default value function* $\mathrm{default} \in \mathscr{T}_\mathrm{v} \to \mathscr{V}$ is defined as follows.

$$\mathrm{default}(\tau) = \begin{cases} \bot_{\mathscr{L}} & \text{if } \tau = \mathcal{L} \\ \texttt{null} & \text{otherwise} \end{cases}$$

Definition 5.9. Execution of a program starts from state $S_0 = H_0, P_0$. H_0 and P_0 are defined as follows for a program $p = \overline{cd}\ \overline{vd}\ t$.

$$H_0 = [o_g \mapsto \langle C_g, FM_g\rangle]$$
$$FM_g = [f \mapsto \mathrm{default}(\sigma) \mid (\texttt{var}\ f\ \texttt{:}\ \sigma) \in \overline{vd}]$$

where o_g is a reserved object identifier for the *global object* $\langle C_g, FM_g\rangle$. C_g is the *global class name*.

$$P_0 = \{\langle L_0, t\rangle \circ \varepsilon|^{\iota_\mathrm{main}}_{\epsilon}\} \qquad L_0 = [\texttt{global} \mapsto o_g].$$

where ι_main is a reserved thread identifier.

Definition 5.10. For a program $p = \overline{cd}\ \overline{vd}\ t$, let the partial function mbody be defined as follows:

$$\mathrm{mbody}(m, C) = \begin{cases} x \to t & \text{if } \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \in \overline{cd} \wedge \\ & \quad \texttt{def}\ m\texttt{(}x\ \texttt{:}\ \sigma\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ t \in \overline{md} \\ \mathrm{mbody}(m, D) & \text{if } \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \in \overline{cd} \wedge \\ & \quad \texttt{def}\ m\texttt{(}x\ \texttt{:}\ \sigma\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ t \notin \overline{md} \\ \text{undefined} & \text{o.w.} \end{cases}$$

In our reduction rules below, we refer to fresh identifiers. The following is one way to define this.

Definition 5.11. For each state $S = H, P$, there are two related sets of currently used object identifiers $\mathcal{O}(S)$, and thread identifiers $\mathcal{D}(S)$. Respectively, these contain all object and thread identifiers occuring in H and P.

We say that an object (thread) identifier o (d) is *fresh* at a state S if $o \notin \mathcal{O}(S)$ ($d \notin \mathcal{D}(S)$).

5.2.2 Reduction Rules

An execution step from state S to S' in RACL is expressed using the relation $\Rrightarrow$, as in

$$S \Rrightarrow S'.$$

RACL reduction rules are expressed at three different levels: Thread set, frame stack and frame level. Therefore, we will also see the relations $\rightarrow$ and $\twoheadrightarrow$, e.g.,

$$H, F \rightarrow H', F' \quad \text{and} \quad H, FS \twoheadrightarrow H', FS'.$$

This is to allow expression of, e.g., single frame execution, thread creation and method calls, as will be explained below. A reduction on frame or frame stack level is propagated up to thread set level with the rules E-FPROP and E-FSPROP which are defined in figures 5.4 and 5.5.

In Figure 5.2, reduction rules for single frames are defined. These are rules that advance the state of a single frame F and possibly changes the heap H. We have very simple ones like the rule E-LVAL, which assigns the specified lattice value to a local variable, and the rule E-VAR, which assigns the value of one local variable to another. The rules E-SELECT and E-ASSIGN respectively fetches and sets the value of a field f of an object on the heap. Rules E-NEW and E-NEWCELL creates a new object on the heap, of either class or cell type, and assigns the corresponding new object identifier to a local variable. All fields of a new class object are initiated with a default value according to the $\mathrm{default}$ function. The rule E-PUT updates the cell value of a cell object on the heap through a join operation. Finally the rule E-WHEN is responsible for adding a new dependency callback. Looking at this rule more closely, we see that it captures the variables specified by $\overline{cap}$ and creates a local environment L_{env}. This is then stored, together with the

$$H, \langle L, \texttt{let}\ x = \texttt{null}\ \texttt{in}\ t\rangle^s \rightarrow H, \langle L[x \mapsto \texttt{null}], t\rangle^s \quad \text{(E-NULL)}$$

$$H, \langle L, \texttt{let}\ x = l\ \texttt{in}\ t\rangle^s \rightarrow H, \langle L[x \mapsto l], t\rangle^s \quad \text{(E-LVAL)}$$

$$H, \langle L, \texttt{let}\ x = y\ \texttt{in}\ t\rangle^s \rightarrow H, \langle L[x \mapsto L(y)], t\rangle^s \quad \text{(E-VAR)}$$

$$\frac{L(y) = o \qquad H(o) = \langle C, FM\rangle \qquad f \in \mathrm{dom}(FM)}{\begin{array}{c} H, \langle L, \texttt{let}\ x = y\texttt{.}f\ \texttt{in}\ t\rangle^s \rightarrow \\ H, \langle L[x \mapsto FM(f)], t\rangle^s \end{array}} \quad \text{(E-SELECT)}$$

$$\frac{\begin{array}{c} L(y) = o \qquad H(o) = \langle C, FM\rangle \qquad f \in \mathrm{dom}(FM) \\ FM' = FM[f \mapsto L(z)] \qquad H' = H[o \mapsto \langle C, FM'\rangle] \end{array}}{\begin{array}{c} H, \langle L, \texttt{let}\ x = y\texttt{.}f = z\ \texttt{in}\ t\rangle^s \rightarrow \\ H', \langle L[x \mapsto L(z)], t\rangle^s \end{array}} \quad \text{(E-ASSIGN)}$$

$$\frac{\begin{array}{c} o \text{ fresh object identifier} \\ FM = [f \mapsto \mathrm{default}(\sigma) : (\texttt{var}\ f : \sigma) \in \mathrm{fdecls}(C)] \\ H' = H[o \mapsto \langle C, FM\rangle] \end{array}}{\begin{array}{c} H, \langle L, \texttt{let}\ x = \texttt{new}\ C\ \texttt{in}\ t\rangle^s \rightarrow \\ H', \langle L[x \mapsto o], t\rangle^s \end{array}} \quad \text{(E-NEW)}$$

$$\frac{o \text{ fresh object identifier} \qquad H' = H[o \mapsto \langle \mathsf{Cell}, \emptyset, \bot_{\mathscr{L}}\rangle]}{\begin{array}{c} H, \langle L, \texttt{let}\ x = \texttt{new Cell in}\ t\rangle^s \rightarrow \\ H', \langle L[x \mapsto o], t\rangle^s \end{array}} \quad \text{(E-NEWCELL)}$$

$$\frac{\begin{array}{c} L(y) = o \qquad H(o) = \langle \mathsf{Cell}, DEP, l\rangle \\ L(z) = l' \qquad c' = \langle \mathsf{Cell}, DEP, l \sqcup l'\rangle \\ H' = H[o \mapsto c'] \end{array}}{\begin{array}{c} H, \langle L, \texttt{let}\ x = y\ \texttt{put}\ z\ \texttt{in}\ t\rangle^s \rightarrow \\ H', \langle L[x \mapsto L(y)], t\rangle^s \end{array}} \quad \text{(E-PUT)}$$

$$\frac{\begin{array}{c} L(y) = o \qquad H(o) = \langle \mathsf{Cell}, DEP, l\rangle \qquad L(z) = l' \\ L_{\text{env}} = [u \mapsto L(u') \mid (u = u') \in \overline{cap}] \qquad cb = (L_{\text{env}}, w \Rightarrow t') \\ \iota \text{ fresh thread identifier} \qquad DEP' = DEP \cup (l', cb)^{\iota} \\ H' = H[o \mapsto \langle \mathsf{Cell}, DEP', l\rangle] \end{array}}{\begin{array}{c} H, \langle L, \texttt{let}\ x = \texttt{when}\ y\ \texttt{pass}\ z\ \texttt{then}\ (\overline{cap}, w \Rightarrow t')\ \texttt{in}\ t\rangle^s \\ \rightarrow H', \langle L[x \mapsto L(y)], t\rangle^s \end{array}} \quad \text{(E-WHEN)}$$

Figure 5.2: RACL single frame reduction rules.

$$\frac{L(y) = \texttt{null}}{H, \langle L, \texttt{let}\ x = y\texttt{.}f\ \texttt{in}\ t,\ \rangle^s \to \textbf{error}} \quad (\text{E-NullSelect})$$

$$\frac{L(y) = \texttt{null}}{H, \langle L, \texttt{let}\ x = y\texttt{.}f = z\ \texttt{in}\ t,\ \rangle^s \to \textbf{error}} \quad (\text{E-NullAssign})$$

$$\frac{L(y) = \texttt{null}}{H, \langle L, \texttt{let}\ x = y\texttt{.}m\texttt{(}z\texttt{)}\ \texttt{in}\ t,\ \rangle^s \to \textbf{error}} \quad (\text{E-NullCall})$$

$$\frac{L(y) = \texttt{null}}{H, \langle L, \texttt{let}\ x = y\ \texttt{put}\ z\ \texttt{in}\ t,\ \rangle^s \to \textbf{error}} \quad (\text{E-NullPut})$$

$$\frac{L(y) = \texttt{null}}{\begin{array}{c}H, \langle L, \texttt{let}\ x = \texttt{when}\ y\ \texttt{pass}\ z\ \texttt{then}\ (\overline{cap}, w \Rightarrow t')\ \texttt{in}\ t\rangle^s \\ \to \textbf{error}\end{array}} \quad (\text{E-NullWhen})$$

Figure 5.3: RACL error spawning rules.

closure $w \Rightarrow t'$, a threshold value l' and a fresh thread identifier ι, in the new dependency set DEP'. E-When mimics the capture semantics of spores [16]. Note that all rules in Figure 5.2 assigns something to the variable x.

In Figure 5.3, rules that lead to **error** are defined. These are all what would be called null-pointer exceptions in a language like Java. That is, that we try to access an object through the null identifier. Errors are propagated to frame stack, and then thread set level using rules E-ErrorFS and E-ErrorP from figures 5.4 and 5.5.

In Figure 5.4, rules for frame stack reductions are defined. Here we find the aforementioned E-FProp and E-ErrorFS, together with rules for method calls and returns. E-Call handles method calls by creating a new frame with the corresponding term and local environment. Note that the new local environment differs depending on the OCAP status of the working thread. If the thread is tagged with $a = \epsilon$, the new frame has access to the global environment, while if $a = ocap$ it will not. Note also that the new frame stack is tagged with the variable name x, the variable to which the method result will be assigned to upon method return. E-Ret corresponds to a method return and utilizes the aforementioned variable name tag. Note that none of these rules changes the thread identifier or OCAP status of a thread.

$$\frac{\begin{array}{c} L(y) = o \qquad H(o) = \langle C, FM\rangle \\ \text{mbody}(m, C) = w \to t' \\ L_{\text{base}} = \begin{cases} \emptyset & \text{if } a = ocap \\ L_0 & \text{if } a = \epsilon \end{cases} \\ L' = L_{\text{base}}[\texttt{this} \mapsto L(y), w \mapsto L(z)] \end{array}}{\begin{array}{c} H, \langle L, \texttt{let}\ x \texttt{ = } y\texttt{.}m\texttt{(}z\texttt{)}\ \texttt{in}\ t\rangle^s \circ FS|_a^\iota \twoheadrightarrow \\ H, \langle L', t'\rangle^x \circ \langle L, t\rangle^s \circ FS|_a^\iota \end{array}} \quad \text{(E-CALL)}$$

$$\begin{array}{c} H, \langle L', x\rangle^y \circ \langle L, t\rangle^s \circ FS|_a^\iota \twoheadrightarrow \\ H, \langle L[y \mapsto L'(x)], t\rangle^s \circ FS|_a^\iota \end{array} \quad \text{(E-RET)}$$

$$\frac{H, F \to H', F'}{H, F \circ FS|_a^\iota \twoheadrightarrow H', F' \circ FS|_a^\iota} \quad \text{(E-FPROP)}$$

$$\frac{H, F \to \textbf{error}}{H, F \circ FS|_a^\iota \twoheadrightarrow \textbf{error}} \quad \text{(E-ERRORFS)}$$

Figure 5.4: RACL frame stack reduction rules.

Finally, in Figure 5.5, the rules for thread set reductions are defined. Here we find E-FSPROP and E-ERRORP, which were mentioned above. Furthermore, we find rule E-SPAWN, which spawns a new callback thread for the threshold value l' in the dependency set of the cell specified by o. Note that in order to use E-SPAWN, the cell value l must have passed the value l' according to the lattice ordering $\sqsubseteq$. Furthermore, the spawned thread uses the thread identifier with which the dependency was tagged. E-TERM is a rule to remove any thread stack finished with its execution.

5.3 Type System

The type system will now be introduced, starting with the types themselves. The types and type lattice of RACL is summarized in Figure 5.6. The lattice structure reflects the subtyping relation defined in Figure 5.7. Except for standard types like the top, bottom, and class types, we find the Cell type, which is a subtype of AnyRef This is similar to the class types. Intuitively, this is due to that cell objects are stored on the heap. We also have a separate lattice value type $\mathcal{L}$.

$$\frac{\begin{array}{c} o \in \mathrm{dom}(H) \qquad H(o) = \langle \mathsf{Cell}, \mathit{DEP}, l \rangle \\ l' \sqsubseteq l \qquad (l', cb)^{\iota} \in \mathit{DEP} \qquad cb = (L_{\text{env}}, z \Rightarrow t) \\ L = L_{\text{env}}[z \mapsto l'] \\ H' = H[o \mapsto \langle \mathsf{Cell}, \mathit{DEP} - (l', cb)^{\iota}, l \rangle] \end{array}}{H, P \Rrightarrow H', P \cup \left\{ \langle L, t \rangle^{-} \circ \varepsilon|^{\iota}_{ocap} \right\}} \quad \text{(E-SPAWN)}$$

$$\frac{P = P' \cup_D \{\langle L, x \rangle^{s} \circ \varepsilon|^{\iota}_{a}\}}{H, P \Rrightarrow H, P'} \quad \text{(E-TERM)}$$

$$\frac{H, \mathit{FS} \twoheadrightarrow H', \mathit{FS}'}{H, P \cup_D \{\mathit{FS}\} \Rrightarrow H', P \cup \{\mathit{FS}'\}} \quad \text{(E-FSPROP)}$$

$$\frac{H, \mathit{FS} \twoheadrightarrow \textbf{error}}{H, P \cup_D \{\mathit{FS}\} \Rrightarrow \textbf{error}} \quad \text{(E-ERRORP)}$$

Figure 5.5: RACL thread set reduction rules.

$$\begin{tikzcd}
 & \top_{\text{RACL}} \arrow[ddl, dash] \arrow[dr, dash] & & \\
 & & \mathsf{AnyRef} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathcal{L} \arrow[ddr, dash] & \mathsf{Cell} \arrow[dr, dash] & & C, D \arrow[dl, dash] \\
 & & \mathsf{Null} \arrow[dl, dash] & \\
 & \bot_{\text{RACL}} & &
\end{tikzcd}$$

Figure 5.6: Type lattice of RACL

$\tau <: \top_{\text{RACL}}$ (ST-TOP) $\qquad C <: \mathsf{AnyRef}$ (ST-C-ANYREF)

$\bot_{\text{RACL}} <: \tau$ (ST-BOT) $\qquad \mathsf{Null} <: \mathsf{Cell}$ (ST-NULL-CELL)

$\mathsf{Cell} <: \mathsf{AnyRef}$ (ST-CELL-ANYREF) $\qquad \mathsf{Null} <: C$ (ST-NULL-C)

$$\frac{p \vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\dots\ \dots\}}{C <: D} \quad \text{(ST-C-D)}$$

Figure 5.7: Subtyping relation of RACL

Remark. Note that the subtyping judgement $\sigma <: \tau$ is implicitly dependent on p due to the inclusion of p in the precondition of rule ST-C-D. In many rules for both execution and typing this holds true aswell. E.g. the use of mbody, a shorthand for the method body, in rule E-CALL also implicitly levarages the program p. We will keep this dependency implicit in order to simplify presentation.

5.3.1 Terms & Expressions

The basic building block of our type system is the typing of expressions and terms. Our typing relation is written

$$\Gamma; a \vdash t : \tau \quad \text{or} \quad \Gamma; a \vdash e : \tau.$$

Apart from the usual components of typing environment Γ, term t or expression e, and type τ, we note that it includes a designator a, which can take the values ϵ and $ocap$. The latter indicates that the term or expression is typed under OCAP constraints. This is equivalent to the OCAP typing of LaCasa [9].

In order to define the typing rules, we need two auxilliary shorthands for the field and method types of classes:

Definition 5.12. For a program $p = \overline{cd}\,\overline{vd}\,t$, let the partial function ftype be defined as follows:

$$\mathrm{ftype}(f, C) = \begin{cases} \tau & \text{if } C = C_g \wedge \texttt{var}\ f\ \texttt{:}\ \tau \in \overline{vd} \\ \sigma & \text{if } \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \in \overline{cd} \wedge \\ & \quad \texttt{var}\ f\ \texttt{:}\ \sigma \in \overline{fd} \\ \mathrm{ftype}(f, D) & \text{if } \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \in \overline{cd} \wedge \\ & \quad \texttt{var}\ f\ \texttt{:}\ \sigma \notin \overline{fd} \\ \text{undefined} & \text{o.w.} \end{cases}$$

Definition 5.13. For a program $p = \overline{cd}\ \overline{vd}\ t$, let the partial function mtype be defined as follows:

$$\mathrm{mtype}(m, C) = \begin{cases} \sigma \rightarrow \tau & \text{if } \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \in \overline{cd} \wedge \\ & \quad \texttt{def}\ m\texttt{(}x\ \texttt{:}\ \sigma\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ t \in \overline{md} \\ \mathrm{mtype}(m, D) & \text{if } \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \in \overline{cd} \wedge \\ & \quad \texttt{def}\ m\texttt{(}x\ \texttt{:}\ \sigma\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ t \notin \overline{md} \\ \text{undefined} & \text{o.w.} \end{cases}$$

All typing rules for terms and expressions can be found in Figure 5.8. Most of the type system rules are standard. For example, rule T-LET types a let-term if the subexpression e is typeable as τ under $\Gamma; a$, and the subterm t is typeable under the extended environment $\Gamma, x : \tau$ and a. Rule T-VAR types a variable x under Γ provided that $x \in \mathrm{dom}(\Gamma)$. T-NEW types new C under effect $a = ocap$ only if the class C is typeable as $ocap$. The rules for typing a class as $ocap$ is defined in Figure 5.10. Typing rule T-CALL states that a method call $x.m(y)$, is only typeable if the type of y is a subtype of the method parameter type σ. T-PUT types the expression x put y if x is typeable as Cell and y is of the lattice type $\mathcal{L}$.

As a final example, the rule T-WHEN describes typing of the dependency creation expression. It says that in order to register a callback for lattice value y in cell x, first x and y must be typeable as Cell and $\mathcal{L}$ respectively. Furthermore, all captured variables in $\overline{cap}$ must be typeable as Cell. This is to ensure that all objects shared between threads are of cell type, which is necessary to prove preservation of thread isolation in the reference graph later. The restricting of capturable types is similar to capturing-constraints in spores [16]. Finally, the term t from callback closure $z \Rightarrow t$ must be typeable in an environment containing the captured cells and $z : \mathcal{L}$.

5.3.2 Well-Formed Programs

Intuitively, a well-formed program is a program which obeys our type system. We define well-formedness in Figure 5.9. We also need the definition of the global typing environment Γ_0:

Definition 5.14. Let the *global typing environment* Γ_0 be defined as

$$\Gamma_0 = \texttt{global} : C_g,$$

where C_g is the global class.

Rule WF-PROG says that in order for a program $p = \overline{cd}\ \overline{vd}\ t$ to be well-formed, all classes $\overline{cd}$ and global variables $\overline{vd}$ must be well-formed, and the program term t must be well-typed. Rule WF-GLOBAL says that in order for a global variable declaration to be well-formed, the denoted type must either be $\mathcal{L}$, Cell, or be of class type C where the class definition of C is well-formed. In order for a class definition of C to be well-formed, rule WF-CLASS declares that all methods must

$$\frac{\Gamma; a \vdash e : \tau \qquad \Gamma, x : \tau; a \vdash t : \sigma}{\Gamma; a \vdash \texttt{let}\ x = e\ \texttt{in}\ t : \sigma} \tag{T-Let}$$

$$\Gamma; a \vdash \texttt{null} : \mathsf{Null} \tag{T-Null}$$

$$\Gamma; a \vdash l : \mathcal{L} \tag{T-LVal}$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma; a \vdash x : \Gamma(x)} \tag{T-Var}$$

$$\frac{\Gamma; a \vdash x : C \qquad \mathrm{ftype}(f, C) = \tau}{\Gamma; a \vdash x\texttt{.}f : \tau} \tag{T-Select}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : C \qquad \mathrm{ftype}(f, C) = \tau \\ \Gamma; a \vdash y : \tau' \qquad \tau' <: \tau\end{array}}{\Gamma; a \vdash x\texttt{.}f = y : \tau} \tag{T-Assign}$$

$$\frac{a = ocap \Longrightarrow ocap(C)}{\Gamma; a \vdash \mathsf{new}\ C : C} \tag{T-New}$$

$$\Gamma; a \vdash \mathsf{new}\ \texttt{Cell} : \mathsf{Cell} \tag{T-NewCell}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : C \qquad \mathrm{mtype}(C, m) = \sigma \to \tau \\ \Gamma; a \vdash y : \sigma' \qquad \sigma' <: \sigma\end{array}}{\Gamma; a \vdash x\texttt{.}m\texttt{(}y\texttt{)} : \tau} \tag{T-Call}$$

$$\frac{\Gamma; a \vdash x : \mathsf{Cell} \qquad \Gamma; a \vdash y : \mathcal{L}}{\Gamma; a \vdash x\ \texttt{put}\ y : \mathsf{Cell}} \tag{T-Put}$$

$$\frac{\begin{array}{c}\Gamma; a \vdash x : \mathsf{Cell} \qquad \Gamma; a \vdash y : \mathcal{L} \\ \forall u = u' \in \overline{cap}.\ \Gamma; a \vdash u' : \mathsf{Cell} \\ \Gamma_{\text{cells}} = [u \mapsto \mathsf{Cell} \mid (u = u') \in \overline{cap}] \\ \Gamma_{\text{cells}}, z : \mathcal{L}; ocap \vdash t : \sigma\end{array}}{\Gamma; a \vdash \mathsf{when}\ x\ \mathsf{pass}\ y\ \mathsf{then}\ (\overline{cap}, z \Rightarrow t) : \mathsf{Cell}} \tag{T-When}$$

Figure 5.8: RACL typing rules for expressions and terms.

$$\frac{p \vdash \overline{cd} \qquad p \vdash \overline{vd} \qquad \Gamma_0; \epsilon \vdash t : \tau}{p \vdash \overline{cd}\,\overline{vd}\,t} \quad \text{(WF-Prog)}$$

$$\frac{\begin{array}{c}\sigma = \mathcal{L} \vee \sigma = \mathsf{Cell} \vee \\ (\sigma = C \wedge p \vdash \texttt{class}\ C\ \texttt{extends}\ \ldots\ \{\ldots\ \ldots\})\end{array}}{p \vdash \texttt{var}\ f\ :\ \sigma} \quad \text{(WF-Global)}$$

$$\frac{\begin{array}{c}C \vdash \overline{md} \\ D = \mathsf{AnyRef} \vee p \vdash \texttt{class}\ D\ \texttt{extends}\ \ldots\ \{\ldots\ \ldots\} \\ \forall (\texttt{def}\ m(\ldots\ :\ \ldots)\ :\ \ldots = \ldots) \in \overline{md}.\ \mathrm{override}(m, C, D) \\ \forall (\texttt{var}\ f\ :\ \tau) \in \overline{vd}.\ f \notin \mathrm{fields}(D)\end{array}}{p \vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{vd}\ \overline{md}\}} \quad \text{(WF-Class)}$$

$$\frac{\mathrm{mtype}(m, D)\ \text{not def.} \vee \mathrm{mtype}(m, C) = \mathrm{mtype}(m, D)}{\mathrm{override}(m, C, D)} \quad \text{(WF-Override)}$$

$$\frac{\begin{array}{c}\Gamma_0, \texttt{this} : C, x : \sigma; \epsilon \vdash t : \tau' \\ \tau' <: \tau\end{array}}{C \vdash \texttt{def}\ m(x\ :\ \sigma)\ :\ \tau = t} \quad \text{(WF-Method)}$$

Figure 5.9: RACL rules for well-formedness of programs.

be well-formed under C, the extended class D must either be AnyRef or must also be well-formed. Furthermore, all methods must obey the rules of overriding, and C cannot redeclare any fields that has been declared in an extended class D. WF-OVERRIDE declares that for a method to be correctly overriden it must either not be declared in any extended class, or it must have the same declared type as in the extended class. Finally WF-METHOD states that in order for a method to be well-formed, its term must be typable as a type τ' under an environment consisting of the global environment, the class itself, `this` $: C$, and the method parameter, $x : \sigma$. Furthermore, $\tau' <: \tau$, where τ is the declared return type of the method.

5.3.3 OCAP Typing

In Figure 5.10, we find the rules for classifying a type as *ocap*. Immediately, we see that unconditionally, both AnyRef and Cell are *ocap*. To type a class C as *ocap*, OCAP-CLASS says that apart from being well-formed, the superclass D must be *ocap*. Furthermore, all methods must be typable under the special judgement $\vdash_{ocap}$ and all fields must be *ocap*. For a method to be typed under $\vdash_{ocap}$, rule OCAP-METHOD declares that the method term must be typable under effect *ocap*, without access to the global environment Γ_0. Being typable under effect *ocap* means that the expression `new` C is only allowed if C is *ocap*, as stated in rule T-NEW. In short, an ocap term can only instansiate ocap classes. This part of the type system is similar to the corresponding part of LaCasa [9]. In short, an *ocap* class can only reference other *ocap* types, and only instansiate other *ocap* classes within its methods. Thus an *ocap* object can never access another object unless it is explicitly passed a reference to it, the global object included.

Definition 5.15. For a program $p = \overline{cd}\ \overline{vd}\ t$, let the partial function $\mathrm{classdef}$ be defined as follows:

$$\mathrm{classdef}(C) = \begin{cases} cd & \text{if } cd \in \overline{cd} \\ & \quad cd = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \\ \text{undefined} & \text{o.w.} \end{cases}$$

$$ocap(\mathsf{AnyRef}) \qquad \text{(OCAP-ANYREF)}$$

$$ocap(\mathsf{Cell}) \qquad \text{(OCAP-CELL)}$$

$$\frac{\begin{array}{c}\mathrm{classdef}(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fd}\ \overline{md}\} \\ ocap(D) \qquad C \vdash_{ocap} \overline{md} \\ \forall(\texttt{var}\ f : \sigma) \in \overline{fd}.\ ocap(\sigma)\end{array}}{ocap(C)} \qquad \text{(OCAP-CLASS)}$$

$$\frac{\begin{array}{c}\texttt{this} : C, x : \sigma; ocap \vdash t : \tau' \\ \tau' <: \tau\end{array}}{C \vdash_{ocap} \texttt{def}\ m(x : \sigma) : \tau = t} \qquad \text{(OCAP-METHOD)}$$

Figure 5.10: RACL OCAP typing rules.

5.4 State Properties

In order to state and prove things such as progress and preservation we need a few more definitions. Many of them are just auxilliary properties of things like states and types and build up to the final definition of a well-typed state.

5.4.1 Well-Typed Heap

This first definition is a straightforward partial function defining the dynamic type of a value.

Definition 5.16. The partial function typeOf is defined as follows:

$$\mathrm{typeOf}(k, H) = \begin{cases} \mathcal{L}, & \text{if } k \in \mathscr{L} \\ \mathsf{Null}, & \text{if } k = \texttt{null} \\ \mathsf{Cell}, & \text{if } k \in \mathrm{dom}(H) \text{ and } H(k) = \langle \mathsf{Cell}, ... \rangle \\ C, & \text{if } k \in \mathrm{dom}(H) \text{ and } H(k) = \langle C, ... \rangle \end{cases} \tag{5.1}$$

In order to simplify the definition of a well-typed heap, we define the following.

Definition 5.17. The typing environment $\Gamma_{\mathsf{Cell}}(L)$ is defined as follows:

$$\Gamma_{\mathsf{Cell}}(L) = [(x : \mathsf{Cell}) \mid x \in \mathrm{dom}(L)] \tag{5.2}$$

Next comes the definition of a well-typed heap. To say that a heap is well-typed means that for all class objects, all field values are of a subtype of the declared type. Furthermore, our definition includes a statement about cell objects. We say that for all callbacks stored in dependency sets, all captured values must be of cell type and the callback term must be typeable under an environment containing the captured variables and closure parameter z.

Definition 5.18 (Well-Typed Heap)**.** A heap H is well-typed, written $\vdash H$, if $\forall o \in \mathrm{dom}(H)$:

If $H(o) = \langle C, FM \rangle$ then

$$\begin{aligned} &\forall f \in \mathrm{fields}(C). \\ &\quad f \in \mathrm{dom}(FM) \wedge \\ &\quad \mathrm{typeOf}(FM(f), H) <: \mathrm{ftype}(f, C) \end{aligned} \tag{5.3}$$

and if $H(o) = \langle \mathsf{Cell}, DEP, l \rangle$ then

$$\begin{aligned} &\forall (l', (L_{\text{env}}, z \Rightarrow t))^{\iota} \in DEP. \\ &\quad \forall (x \mapsto k) \in L_{\text{env}}.\ \mathrm{typeOf}(k, H) <: \mathsf{Cell} \wedge \\ &\quad \Gamma_{\mathsf{Cell}}(L_{\text{env}}), z : \mathcal{L}; ocap \vdash t : \gamma \end{aligned} \tag{5.4}$$

5.4.2 Well-Typed Threads

A non erroneous state $S = H, P$ has two parts, the heap H and thread set P. In order to define and prove preservation properties, we furthermore need to put restrictions on the thread set P. We do this with the relation $H \vdash P$, meaning P is well-typed under heap H. This relation is defined in Figure 5.11. T-PROCS and T-EMPTY basically says that in order for a thread set P to be well-typed under heap H, all threads need to be well-typed under H and its OCAP status a. For a thread $GS|_a^{\iota}$ we write this as $H; a \vdash GS$. This relation is defined in Figure 5.12.

For $H; a \vdash GS$ to hold, either $FS = \varepsilon$ as in rule T-FSEMPTY1 (this is actually impossible since frame stacks of a state cannot be empty, see definition 5.5), or the term t of top frame F is typeable with some environment Γ which is conformant with its local variable map L and H. This conformancy is expressed through the relation $H \vdash \Gamma; L$, which is defined in Figure 5.13. Simply stated, $H \vdash \Gamma; L$ says that all types

$$H \vdash \emptyset \qquad \text{(T-EMPTY)}$$

$$\frac{H; a \vdash FS \qquad H \vdash P}{H \vdash P \cup \{FS|^{\iota}_{a}\}} \qquad \text{(T-PROCS)}$$

Figure 5.11: Rules for typing thread sets under some heap H.

$$H; a \vdash \varepsilon \qquad \text{(T-FSEMPTY1)}$$

$$H; a \vdash^{x:\sigma} \varepsilon \qquad \text{(T-FSEMPTY2)}$$

$$\frac{\begin{array}{c} F = \langle L, t\rangle^{x} \qquad H \vdash \Gamma; L \\ \Gamma; a \vdash t : \sigma' \qquad \sigma' <: \sigma \qquad H; a \vdash^{x:\sigma} FS \end{array}}{H; a \vdash F \circ FS} \qquad \text{(T-FS1)}$$

$$\frac{\begin{array}{c} F = \langle L, t\rangle^{y} \qquad H \vdash \Gamma; L \\ \Gamma, x : \tau; a \vdash t : \sigma' \qquad \sigma' <: \sigma \qquad H; a \vdash^{y:\sigma} FS \end{array}}{H; a \vdash^{x:\tau} F \circ FS} \qquad \text{(T-FS2)}$$

Figure 5.12: Rules for typing frame stacks under some heap H and effect a.

$$\frac{\text{typeOf}(L(x), H) <: \Gamma(x)}{H \vdash \Gamma; L; x} \qquad \text{(WF-ENVVAR)}$$

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

Figure 5.13: Rules for classifying local environments L as well-typed.

$$\frac{o \in \mathrm{dom}(H)}{\mathrm{reach}(H, o, o)} \qquad (\textsc{Reach1})$$

$$\frac{\begin{array}{cc} o \in \mathrm{dom}(H) & H(o) = \langle C, FM \rangle \\ o'' \in \mathrm{codom}(FM) & \mathrm{reach}(H, o'', o') \end{array}}{\mathrm{reach}(H, o, o')} \qquad (\textsc{Reach2})$$

Figure 5.14: Definition of heap reachability.

specified in Γ align with the dynamic types of the values in L. Furthermore T-FS1 declares that the rest of the frame stack FS must be typable under the judgement $\vdash^{x:\sigma}$. This judgement is defined by rules T-FSEMPTY2 and T-FS2. The latter rule reflects that the top frame returns some value to its underlying frame stack. It is closely connected with the well-formedness of methods and call/return execution semantics, as defined by rules WF-METHOD and E-CALL/E-RET respectively.

5.4.3 Isolation

The next two definitions are used when defining isolation of threads in the reference graph of the heap. The ruling $\mathrm{reach}(H, o, o')$ is defined in Figure 5.14, and this is basically a formal definition of object o' being reachable from o in heap H using field accesses.

Definition 5.19 (Class Object Separation)**.** For any heap H and heap identifiers o, o' we have class object separation, $\mathrm{csep}(H, o, o')$ iff

$$\begin{aligned} &\forall q, q' \in \mathrm{dom}(H). \\ &\quad \mathrm{reach}(H, o, q) \wedge \mathrm{reach}(H, o', q') \implies \\ &\quad q \neq q' \vee \mathrm{typeOf}(q, H) = \mathsf{Cell} \end{aligned} \tag{5.5}$$

Definition 5.20 (Accessible Roots)**.** For a heap H and a frame stack FS we define $\mathrm{accRoots}(FS, H)$ as

$$\mathrm{accRoots}(FS, H) = \left\{ o \in \mathrm{dom}(H) \;\middle|\; \begin{array}{c} \exists \langle L, t \rangle^s \in \mathrm{frames}(FS), x \in \mathrm{Var} \\ \text{s.t. } (x \mapsto o) \in L \end{array} \right\}$$

The function frames above is defined as

$$\begin{aligned} \mathrm{frames}(\varepsilon) &= \emptyset \\ \mathrm{frames}(F \circ GS) &= F \cup \mathrm{frames}(GS) \end{aligned}$$

$$\frac{\forall o \in \mathrm{accRoots}(FS, H), o' \in \mathrm{accRoots}(GS, H).\ \mathrm{csep}(H, o, o')}{\mathrm{isolated}(H, FS, GS)} \quad \text{(ISO-FS)}$$

$$\frac{\begin{array}{c}\forall FS|^{\iota}_{a}, GS^{\iota'}_{b} \in P \text{ where } FS|^{\iota}_{a} \neq GS|^{\iota'}_{b}.\\ a = ocap \vee b = ocap \implies \mathrm{isolated}(H, FS, GS)\end{array}}{\mathrm{isolation}(H, P)} \quad \text{(ISO-P}\textsc{rocs}\text{)}$$

Figure 5.15: Definition of isolation

$$\frac{a = ocap \implies \mathrm{ocr}(FS, H)}{H; a \vdash FS\ \ \mathbf{ocr}} \quad \text{(OCR-FS)}$$

$$\frac{\forall FS|^{\iota}_{a} \in P.\ H; a \vdash FS\ \ \mathbf{ocr}}{H \vdash P\ \ \mathbf{ocr}} \quad \text{(OCR-P)}$$

Figure 5.16: Definition of OCAP reachability

Isolation of threads is defined in Figure 5.15. Rule ISO-FS states that for two threads to be isolated with respect to heap H, all accessible roots of the two threads must have class object separation, i.e., all objects that are reachable from both threads must be of type Cell. Rule ISO-PROCS states that we have isolation for a thread set P under heap H if there is isolation between all pairs of threads such that at least one is $ocap$.

5.4.4 OCAP Reachability

In order to prove preservation of isolation we also need to ensure that all objects reachable from an $ocap$-annotated thread can be typed as an $ocap$ type. The next definition together with Figure 5.16 defines this property formally.

Definition 5.21 (OCAP Reachability)**.** For a heap H and frame stack FS we have $\mathrm{ocr}(FS, H)$ iff

$$\begin{aligned}&\forall o \in \mathrm{accRoots}(FS, H), o' \in \mathrm{dom}(H).\\ &\qquad \mathrm{reach}(H, o, o') \implies ocap(\mathrm{typeOf}(o', H))\end{aligned} \tag{5.6}$$

$$\frac{\forall FS|_a^\iota \in P.\, a = ocap \implies \forall o \in \mathrm{accRoots}(FS, H).\, \mathrm{csep}(H, o, o_g)}{H \vdash P \ \mathbf{gsep}}$$

(GSEP-THREADS)

Figure 5.17: Definition of global object separation

5.4.5 Global Object Separation

Another thing needed to prove preservation of isolation is global object separation. Simply stated, this means that no $ocap$ thread can reach the global object o_g through the heap object reference graph. Global object separation is formally defined in Figure 5.17.

5.4.6 No Thread Spawning

The next definition is needed to state and prove progress. Intuitively $\mathrm{noSpawn}(H)$ means there are no threads waiting to spawn (compare with rule E-SPAWN).

Definition 5.22 (No Spawn)**.** For any heap H we have $\mathrm{noSpawn}(H)$ if and only if

$$\begin{aligned}&\forall o \in \mathrm{dom}(H).\\ &\quad H(o) = \langle \mathsf{Cell}, DEP, l\rangle \implies \forall (l', cb)^\iota \in DEP.\, \neg(l' \sqsubseteq l).\end{aligned} \tag{5.7}$$

5.4.7 Unique Main Thread

In order to prove determinism, we must be sure that there is at most one non-$ocap$ thread running. Otherwise these could interfere since there are no constraints on whether these can share data. Therefore, we define the following property.

Definition 5.23 (Unique Main Thread)**.** For an OCAP status a, let

$$\chi_\epsilon(a) = \begin{cases} 1 & \text{if } a = \epsilon \\ 0 & \text{o.w.}\end{cases}$$

For a thread set P let

$$\chi(P) = \sum_{FS|_a^\iota \in P} \chi_\epsilon(a).$$

This is the number of non-OCAP-protected threads in P. Finally we define

$$\text{uniqMain}(P) \iff \chi(P) \leq 1$$

5.4.8 Well-Typed States

Finally we can define the notion of a well-typed state. This combines many of the properties already defined above into one.

Definition 5.24 (Well-Typed States)**.** For a state S we say that it is *well-typed* with regards to a well-typed program p, written

$$\vdash S \ \textbf{ok}_p$$

if $S = \textbf{error}$ or $S = H, P$ and

$$\begin{array}{ccc} \vdash H & H \vdash P & H \vdash P \ \textbf{ocr} \\ \text{isolation}(H, P) & H \vdash P \ \textbf{gsep} & \text{uniqMain}(P) \end{array} \tag{5.8}$$

Remark. We will henceforth refer to this only as being well-typed. The well-typed program p will therefore in many cases be implicit, as it already is in all judgements of equation (5.8).

Chapter 6

Properties of RACL

We are now finally ready to state and prove the progress and preservation properties of RACL. These are then used to prove quasi-determinism.

6.1 Preservation & Progress

The preservation theorem states that if we have a well-typed state that can step to another state, this will also be well-typed.

Theorem 1 (Preservation). *Let S, S' be states such that $\vdash S \quad \mathbf{ok}_p$ and $S \Rrightarrow S'$. Then $\vdash S' \ \mathbf{ok}_p$.*

The proof of preservation in its full can be found in Appendix A. The part concerning preservation of isolation is probably the most interesting since properties like well-typed heap and threads are standard. The preservation proof of isolation is based on the fact that, by OCAP reachability, all *ocap* threads can only reach *ocap* class objects. Combined with global object separation, this means that an *ocap* thread can never access the global object or create non-*ocap* objects through a `new` ...-expression. This implies that the thread follows the OCAP model as described in Section 2.5. Thus isolated threads will stay isolated.

Theorem 2 states that a well-typed program never gets stuck unless we encounter an error or finish execution properly.

Theorem 2 (Progress). *Let S be a state such that $\vdash S \ \mathbf{ok}_p$. Then either*

1. *$\exists S'$ s.t. $S \Rrightarrow S'$,*

2. $S = H, \emptyset$ *for some heap* H *s.t.* $\mathrm{noSpawn}(H)$ *or*

3. $S =$ **error**.

The proof of progress is by contradiction. The full proof can also be found in Appendix A.

6.2 Quasi-Determinism

Having proved soundness of the type system, the next thing to prove is quasi-determinism. In order to state this more formally, we need some definitions.

We first define a relation on states and show that it is an equivalence relation.

Definition 6.1. Let $\simeq$ be a binary relation on the set of states $\mathcal{S}$. We let $S \simeq S'$ if

$$S = S' = \textbf{error}$$

or if

$$S = H, P \qquad S' = H', P' \tag{6.1}$$

and there exists bijections $g \in \mathcal{O} \hookrightarrow \mathcal{O}, h \in \mathcal{D} \hookrightarrow \mathcal{D}$ such that

$$H' = \pi(H, g) \qquad P' = \rho(P, g, h) \tag{6.2}$$

The functions π and ρ replace all object and thread identifiers occuring in H or P according to the bijections g and h. They are explicitly defined in Definition B.3, Appendix B.

Proposition 6.2. $\simeq$ *is an equivalence relation.*

A proof sketch of this can be found in Appendix B. Similarly we can prove the following.

Proposition 6.3. *For any* $S, S' \in \mathcal{S}$ *such that* $S \simeq S'$

$$\vdash S \ \textbf{ok}_p \iff \vdash S' \ \textbf{ok}_p \tag{6.3}$$

A proof sketch can also be found in Appendix B. We can also make the following observation

Claim. *A transition between two states can be uniquely identified by:*

- *A start state S*
- *A base rule name R, e.g., $R =$* E-NEW *or $R =$* E-SPAWN. *We say base rule because in many cases the step involves more than one rule in its derivation tree of a complete state step. However the use of these extra rules are implied by which rule is used as the "base" of the tree. For example, using* E-CALL *to reduce a thread implies use of* E-FSPROP *to advance the state, and using* E-NEW *to advance one thread frame implies use of* E-FPROP *and* E-FSPROP *in order to advance the state.*
- *A thread identifier $\iota \in \mathcal{D}$. In the cases where we advance or terminate a thread, e.g.,* E-NULL *or* E-CALL, *this is the thread identifier of that thread, as in Definition 5.5. In the case where we spawn a new thread, this is the unique thread identifier as in Definition 5.3.*
- *A fresh object or thread identifier β which could be of value:*
  - *Object identifier $o' \in \mathcal{O}$ in the cases where the rule references a fresh object identifier, i.e.* E-NEW *and* E-NEWCELL.
  - *Thread id $\iota' \in \mathcal{D}$ for the case where a fresh thread id is referenced, i.e.* E-WHEN.
  - *☺, a default value for all other cases.*

Because of this claim we can make the following definition

Definition 6.4. We write a *transition identifier* as $R^{\iota,\beta}$. The *application* of $R^{\iota,\beta}$ to state S (if possible) is the use of a rule specified by R, ι and β as described above, to step from S to some state S'. We write this as

$$S \Rrightarrow^{R^{\iota,\beta}} S'$$

Remark. The rule name R in the above definition is actually redundant since this is completely decided by the state S and the thread identifier ι.

To simplify notation we make the following definition.

Definition 6.5. A *transition sequence* $\bar{R}$ is a finite length sequence of transition identifiers,

$$\bar{R} = R_1^{\iota_1,\beta_1}, R_2^{\iota_2,\beta_2}, \ldots, R_n^{\iota_n,\beta_n}.$$

The application of this sequence to a state S is the consecutive application of

$$R_i^{\iota_i,\beta_i} \qquad i = 1, \ldots, n$$

beginning with state S. I.e.

$$S \Rrightarrow^{R_1^{\iota_1,\beta_1}} S_1 \Rrightarrow^{R^{\iota_2,\beta_2}} S_2 \ \ldots\ S_{n-1} \Rrightarrow^{R_n^{\iota_n,\beta_n}} S_n$$

We shorten this to

$$S \Rrightarrow^{\bar{R}} S_n.$$

We call n the length of $\bar{R}$.

We state the following proposition without proof, just for reference. This follows from the observations made in the claim above.

Proposition 6.6. *Let $H, P \Rrightarrow H', P'$. Then there is a unique transition identifier $R^{\alpha,\beta}$ such that $H, P \Rrightarrow^{R^{\alpha,\beta}} H', P'$.*

We are now ready to state the last main theorem.

Theorem 3. *Let S, S', T, T' be well-typed states not equal to* **error** *such that*

$$S \simeq S'$$
$$S \Rrightarrow^{\bar{R}} T \qquad S' \Rrightarrow^{\bar{R}'} T',$$

Let n and n' be the lengths of $\bar{R}$ and $\bar{R}'$ respectively. Furthermore we assume that neither T or T' can make a step. Then

$$n = n' \quad \textit{and} \quad T \simeq T'.$$

The proof of Theorem 3 is based on induction on the length of the two transition sequences. It can be seen as incrementally transforming $\bar{R}'$ to $\bar{R}$ by moving transition identifiers to positions specified by $\bar{R}$. Each intermediate sequence is shown to halt in the same state. The full proof can be found in Appendix B.

Finally, quasi-determinism follows from Theorem 3 and proposition 6.6:

Corollary 6.7 (Quasi-Determinism)**.** *Let S, S', T, T' be well-typed states not equal to* **error** *such that*

$$S \simeq S'$$
$$S \Rrightarrow^{*} T \qquad S' \Rrightarrow^{*} T',$$

and neither T or T' can make a step. Then

$$T \simeq T'.$$

6.3 Summary

We have proved soundness for the formal model RACL. This means that, given that a program is well typed under our type system, it will never reach an error state unless it encounters a null-pointer exception. Furthermore, the program will never halt unless it has reached an error state, or no threads are running and no threads are waiting to spawn.

Furthermore we have proved that RACL posesses the property of quasi-determinism. This means that all non-erroneous halting states of a well-typed program are equivalent.

Chapter 7

Discussion & Conclusion

In this chapter, RACL, the formal system of this thesis is discussed from two angles: extensions and implementation. Finally, the report is summarized with conclusions.

7.1 A Remark About Determinism

In this report, quasi-determinism of RACL is proved. It should, however, be possible to prove complete determinism, meaning that you get the same halting result (also considering **error** as a result) in all executions which start in the same state. This is due to the fact that, from the point of view of a thread, all return values from cell operations are agnostic to the actual cell value. This implies that callback threads, with the current semantics, are in practice completely isolated and cannot share any information. This in turn means that the execution of a thread is deterministic. By this, all callback threads spawned in one execution will spawn in all possible executions, meaning that if there is an error spawning thread in one execution path, there will be one in all other executions as well.

7.2 Possible Extensions

Currently, RACL does not have any way to actually use the result of a computation: there is no way to access the final cell values. The freeze operation of LVish [13] is flawed and allows us to write non-deterministic programs, as shown at the end of Section 4.3. However, as hinted at the end of the same section, there should be a way to freeze values and access them without breaking determinism. This uses a form of quiescence, the concept introduced by Kuper et al. [13]. The difference is that this new form of freezing needs to be done globally, and at a time where changes to cell values are no longer going to take place. Quiescence allows us to do this. For example, if no threads are running or waiting to spawn, we can be sure that the cell values will not change. Therefore, with this definition of quiescence, introducing a program statement that blocks until quiescence and then freezes all cells should allow us to access the result without risking non-determinism. In order to ensure this does not cause deadlocks, this *on-quiescence-freeze* operation should only be allowed in the sole main (non-$ocap$) thread of RACL.

As mentioned in Section 3.2, LVars has an additional get operation, not modeled in RACL. Adopting the semantics of LVars, it should be possible to extend RACL with this operation as well.

Apart from cell objects, it should also be possible to share immutable data between threads. This would require objects being deeply immutable: apart from requiring the references passed to be immutable, an immutable object should only be able to contain immutable references. This property must also be transitive, requiring that all references held by referenced objects be immutable. A similar way to define immutability for Scala was examined by Haller and Axelsson [7].

7.3 Implementation

In order to implement RACL effectively, some changes have to be made in the way callbacks are registered and spawned. This is mainly due to the fact that the dependency set semantics of our formal system are hard to implement effectively, since it requires scanning all the dependency sets for threshold values that have been passed. The time required for this operation is linear in the combined size of the dependency sets.

The LVish implementation solved this inefficiency problem by optimizing their library for so called atomic lattices, i.e., lattices where all values can be written as a join of so called *atoms*. An atom a is an element of the lattice $\mathscr{L}$ where $l \sqsubseteq a \implies l \in \{\bot, a\}$. For example, the lattice of integer sets $(2^{\mathbb{Z}}, \subseteq)$ has the atoms

$$\{i\} \qquad i \in \mathbb{Z}.$$

Working with atomic lattices allows for expressing an LVar update by a delta value, i.e., a difference between the old and new value. Instead of using the threshold set semantics, LVish registers callback generators, which for each update to an LVar generates callbacks to be run. Leveraging the delta representation, these generators can be made efficient in many cases [13]. In a RACL implementation, these generators would of course also have to be isolated similar to regular callbacks, in order to guarantee determinism.

7.4 Machine-checked Proofs

The proofs presented in this thesis, and in works similar to it, naturally get quite long. As humans we are prone to make mistakes, which can be hard to find in such situations. This is probably what happened in the case of the erroneous proof of Kuper et al. [13] described earlier. One way to solve this problem is to generate a machine-checked proof, using e.g. the Coq Proof Assistant [4]. For example, an effort to formalize LaCasa in Coq was made during the work of Haller and Loiko [9]. However, doing this for RACL was outside the scope of this thesis.

7.5 Sustainability & Ethical Aspects

Because of the theoretical nature of this thesis, ethical complications are hard to anticipate. The theory concerning deterministic-by-design concurrent systems is explored to help developers reason about and construct parallelizable applications. What these applications then do is out of our control.

With regards to sustainability there are a few aspects that could be considered. The question of developer effort is one. Creating tools for developers to quicken development of software that more efficiently utilizes modern hardware is reasonable to strive for. Furthermore, a tool that ensures determinism in a concurrent setting makes reasoning about computation a lot easier, since we can be sure that wrong results do not depend on the execution order, something very much hidden from the programmer's view. Simplifying reasoning in turn simplifies understanding. Thus, bugs can be found easier, an important gain in a time when software safety and security are important concerns.

Another aspect of sustainability is energy usage. A deterministic-by-design system might have computational overheads, reducing efficiency in operation. However, in the work on LVish and Reactive Async there have been results indicating more efficient use of hardware, reducing time for computation [13, 8]. This shows that on the contrary, use of such systems can increase efficiency and thus reduce energy usage. Examining similar properties for RACL was outside the scope of this thesis, since we at the moment do not have a working implementation of our formalized model.

7.6 Conclusion

With this thesis, the aim was to construct a similar formal system to LVish, but with object-oriented characteristics, mostly leveraging the work of Haller and Loiko [9]. Furthermore, the aim was to prove quasi-determinism similar to LVish. This goal has been met. However, the discovery of an error in the proof of Kuper et al. [14] led to the elimination of the freeze operation in its current form. This indicates two things: firstly, when formal systems get larger, proofs get longer and great care has to be taken in order to ensure correctness. Secondly, the goal of determinisic-by-design concurrent systems is difficult to understand, and seemingly safe operations such as freezing could have unforseen consequences.

Bibliography

[1] *Akka framework*. URL: `https://akka.io/` (2018-08-21).

[2] Robert L. Bocchino. *An effect system and language for deterministic-by-default parallel programming*. en. Apr. 2013. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.220.395;%20http://www.cs.cmu.edu/%7Erbocchin/Publications_files/Bocchino-PhD-Thesis.pdf`.

[3] Robert Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. "Parallel Programming Must Be Deterministic by Default". In: *Proc. HotPar '09 (1st USENIX Workship on Hot Topics in Parallelism), USB Data Stick*. UIUC. Berkeley, CA: Usenix Assoc., Mar. 2009.

[4] *Coq Proof Assistant*. URL: `https://coq.inria.fr/` (2018-11-25).

[5] J. B. Dennis and E. C. VanHorn. "Programming Semantics for Multiprogrammed Computations". In: *CACM* 9.3 (Mar. 1966), pp. 143–155.

[6] *Deterministic Parallel Java project*. URL: `https://docs.scala-lang.org/tour/basics.html` (2018-08-21).

[7] Philipp Haller and Ludvig Axelsson. "Quantifying and Explaining Immutability in Scala". In: *Proceedings Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017.* 2017, pp. 21–27.

[8] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. "Reactive Async: expressive deterministic concurrency". In: *SCALA@SPLASH*. Ed. by Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche. ACM, 2016, pp. 11–20. ISBN: 978-1-4503-4648-1. URL: `http://doi.acm.org/10.1145/2998392`.

[9] Philipp Haller and Alexander Loiko. "LaCasa: lightweight affinity and object capabilities in Scala". In: *OOPSLA*. Ed. by Eelco Visser and Yannis Smaragdakis. ACM, 2016, pp. 272–291. ISBN: 978-1-4503-4444-9. URL: `http://doi.acm.org/10.1145/2983990`.

[10] C. Hewitt. "A universal, modular Actor formalism for artificial intelligence". In: *Proc.International Joint Conference on Artificial Intelligence*. 1973.

[11] G. Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Proc. IFIP 74*. Amsterdam: North-Holland, Aug. 1974, pp. 471–475.

[12] Lindsey Kuper and Ryan R Newton. "LVars: lattice-based data structures for deterministic parallelism". In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. ACM. 2013, pp. 71–84.

[13] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. "Freeze after writing: quasi-deterministic parallel programming with LVars". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014, pp. 257–270. DOI: `10.1145/2535838.2535842`. URL: `https://doi.org/10.1145/2535838.2535842`.

[14] Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. *Freeze after writing: Quasi-deterministic parallel programming with LVars, Technical report TR710*. Tech. rep. URL `https://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR710`. Indiana University, Nov. 2013.

[15] Simon Marlow, Ryan Newton, and Simon L. Peyton Jones. "A monad for deterministic parallelism". In: *Haskell*. Ed. by Koen Claessen. ACM, 2011, pp. 71–82. ISBN: 978-1-4503-0860-1. URL: `http://doi.acm.org/10.1145/2034675.2034685`.

[16] Heather Miller, Philipp Haller, and Martin Odersky. "Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution". In: *ECOOP*. Ed. by Richard E. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 308–333. ISBN: 978-3-662-44201-2.

[17] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Doctoral Dissertation. Baltimore, MD: Johns Hopkins Univ., 2006.

[18] Ryan Newton, Chih-Ping Chen, and Simon Marlow. *Intel Concurrent Collections for Haskell*. Mar. 2011. URL: `http://hdl.handle.net/1721.1/61759`.

[19] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: an appetizer*. Springer Science & Business Media, 2007.

[20] Rishiyur S Nikhil, Keshav K Pingali, et al. "I-structures: Data structures for parallel computing". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11.4 (1989), pp. 598–632.

[21] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[22] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. "FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction". In: *LCPC*. Ed. by Hironori Kasahara and Keiji Kimura. Vol. 7760. Lecture Notes in Computer Science. Springer, 2012, pp. 158–173. ISBN: 978-3-642-37657-3; 978-3-642-37658-0.

[23] *Scala basics*. URL: `https://docs.scala-lang.org/tour/basics.html` (2018-08-21).

[24] *Spark framework*. URL: `https://spark.apache.org/` (2018-08-21).

Appendix A

Proof of Preservation and Progress

A.1 Preliminaries

Definition A.1 (Heap Induced Graph)**.** The induced graph of heap H, written $\mathscr{G}(H)$ is the directed graph (V, E) such that $V = \mathrm{dom}(H)$ and

$$E = \left\{(o, o')_f \in \mathrm{dom}(H)^2 \times \mathscr{F} \mid H(o) = \langle C, FM\rangle \text{ and } FM(f) = o'\right\} \tag{A.1}$$

Definition A.2 (Graph Reachability)**.** We say an object o' is reachable from o in graph $G = (V, E)$, written $\mathrm{reach}(G, o, o')$ if there is a finite sequence $o_0, \ldots, o_n \in V$ such that $o = o_0, o_n = o'$ and

$$\forall i = 1, \ldots, n-1.\ \exists f_i \in \mathscr{F} \text{ s.t. } (o_i, o_{i+1})_{f_i} \in E. \tag{A.2}$$

The sequence $o_0, \ldots, o_n$ is called a *path*.

Definition A.3. Given a graph $G = (V, E)$ and a set $O \subseteq V$, we define

$$\mathrm{reachable}(O, G) = \{q \in V : \exists o \in O.\ \mathrm{reach}(G, o, q)\}$$

Proposition A.4 (Reachability equivalence)**.** *For a heap H*

$$\mathrm{reach}(H, o, o') \iff \mathrm{reach}(\mathscr{G}(H), o, o') \tag{A.3}$$

Proof. Both directions of the implication are simple to prove.

Case $\Longrightarrow$: By induction on the shape of derivation tree.

Case $\Longleftarrow$: By induction on the path length in graph $\mathscr{G}(H)$.

□

Proposition A.5. *For any heap H and frame stack FS we have*

$\mathrm{ocr}(FS, H)$
$\iff$
$\forall q \in \mathrm{reachable}(\mathrm{accRoots}(FS, H), \mathscr{G}(H)).$
$\quad ocap(\mathrm{typeOf}(q, H))$

Proof. Follows from definition of ocr, accRoots and reachable. □

Proposition A.6. *For any heap H and object references o, o' we have*

$\mathrm{csep}(H, o, o') \iff$
$\forall q \in \mathrm{reachable}(\{o\}, \mathscr{G}(H)) \cap \mathrm{reachable}(\{o'\}, \mathscr{G}(H)).$
$\qquad \mathrm{typeOf}(q, H) <: \mathsf{Cell}$

Proof. First of all we let $R = \mathrm{reachable}(\{o\}, \mathscr{G}(H)), R' = \mathrm{reachable}(\{o'\}, \mathscr{G}(H))$. We prove each direction of the implication separately.

Case $\implies$: Take any $q \in R \cap R'$. By definition of reachable, reachability equivalence (prop. A.4) and definition of $\mathrm{csep}(H, o, o')$ we have $\mathrm{typeOf}(q, H) <: \mathsf{Cell}$.

Case $\impliedby$: Take any $q, q' \in \mathrm{dom}(H)$ such that $\mathrm{reach}(H, o, q)$ and $\mathrm{reach}(H, o', q')$. If $q \neq q$ we are done. Otherwise $q = q'$ and by definition of reachable, $q \in R \cap R'$. By assumption $\mathrm{typeOf}(q, H) <: \mathsf{Cell}$.

□

Proposition A.7. *For any heap H and frame stacks FS, HS we have*

$\mathrm{isolated}(H, FS, HS) \iff$
$\forall q \in \mathrm{reachable}(\mathrm{accRoots}(FS, H), \mathscr{G}(H)) \cap$
$\qquad \mathrm{reachable}(\mathrm{accRoots}(HS, H), \mathscr{G}(H)).$
$\qquad \mathrm{typeOf}(q, H) <: \mathsf{Cell}$

Proof. We let

$$R = \mathrm{reachable}(\mathrm{accRoots}(FS, H), \mathscr{G}(H))$$
$$R' = \mathrm{reachable}(\mathrm{accRoots}(HS, H), \mathscr{G}(H))$$

We prove each direction of the implication separately.

Case $\implies$: Take $q \in R \cap R'$. Then

$$\exists o \in \text{accRoots}(FS, H), o' \in \text{accRoots}(HS, H)$$

such that

$$q \in \text{reachable}(\{o\}, \mathscr{G}(H)) \cap \text{reachable}(\{o'\}, \mathscr{G}(H)).$$

Thus by $\text{isolated}(H, FS, HS)$ we have $\text{csep}(H, o, o')$. Proposition A.6 yields $\text{typeOf}(q, H) <: \mathsf{Cell}$.

Case $\Longleftarrow$: Take any o, o' such that

$$o \in \text{accRoots}(FS, H) \qquad o' \in \text{accRoots}(HS, H).$$

By definition

$$\begin{gathered}\text{reachable}(\{o\}, \mathscr{G}(H)) \subseteq R \\ \text{reachable}(\{o'\}, \mathscr{G}(H)) \subseteq R'.\end{gathered}$$

Thus by assumption

$$\begin{aligned}&\forall q \in \text{reachable}(\{o\}, \mathscr{G}(H)) \cap \text{reachable}(\{o'\}, \mathscr{G}(H)).\\ &\quad \text{typeOf}(q, H) <: \mathsf{Cell}.\end{aligned}$$

Finally by Proposition A.6, we have $\text{csep}(H, o, o')$. Since o, o' were arbitrary, we are done.

$\square$

Proposition A.8. *Let H, H' be heaps and P, P' thread sets such that*

1. $P = Q \cup_D \{FS|_a^\iota\}$, $\text{isolation}(H, P)$, $H \vdash P\,\mathbf{ocr}$ *and* $P' = Q \cup_D \{FS'|_a^\iota\}$
2. $\mathscr{G}(H) = \mathscr{G}(H')$
3. $\forall HS|_b^{\iota'} \in Q.$ $\text{reachable}(\text{accRoots}(HS, H'), \mathscr{G}(H')) \subseteq$ $\text{reachable}(\text{accRoots}(HS, H), \mathscr{G}(H))$
4. $\text{reachable}(\text{accRoots}(FS', H'), \mathscr{G}(H')) \subseteq$ $\text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H))$
5. $\forall o \in \text{dom}(H).\ \text{typeOf}(o, H) = \text{typeOf}(o, H')$

Then $\text{isolation}(H', P')$ *and* $H' \vdash P'$ **ocr**.

Remark. Note that $\mathscr{G}(H) = \mathscr{G}(H')$ implies $\text{dom}(H) = \text{dom}(H')$. Otherwise many of the preconditions above would not make sense.

Proof. We prove that

$\forall$ distinct $HS|_b^{\iota'}, GS|_c^{\iota''} \in P'.\ b = ocap \vee c = ocap \implies \text{isolated}(H', HS, GS)$.

This implies $\text{isolation}(H', P')$ by rule ISO-PROCS. Thus take any distinct $HS|_b^{\iota'}, GS|_c^{\iota''} \in P'$. Then by assumption 3, 4 and basic set properties

$$\begin{gathered}\text{reachable}(\text{accRoots}(HS, H'), \mathscr{G}(H')) \cap \text{reachable}(\text{accRoots}(GS, H'), \mathscr{G}(H')) \\ \subseteq \\ \text{reachable}(\text{accRoots}(HS, H), \mathscr{G}(H)) \cap \text{reachable}(\text{accRoots}(GS, H), \mathscr{G}(H)).\end{gathered} \tag{A.4}$$

By this, Proposition A.7, $\text{isolation}(H, HS, GS)$ and assumption 5, we have $\text{isolation}(H', HS, GS)$. By (A.4) and assumption 3,4 and 5 we have $H' \vdash P'$ **ocr**. □

Corollary A.9. *Let* H, H' *be heaps and* P, P' *thread sets such that*

1. $P = Q \cup_D \{FS|_a^\iota\}$, $\text{isolation}(H, P)$, $H \vdash P$ **ocr** *and* $P' = Q \cup_D \{FS'|_a^\iota\}$
2. $\mathscr{G}(H) = \mathscr{G}(H')$
3. $\forall HS|_b^{\iota'} \in Q.\ \text{accRoots}(HS, H') \subseteq \text{accRoots}(HS, H)$
4. $\text{accRoots}(FS', H') \subseteq \text{accRoots}(FS, H)$
5. $\forall o \in \text{dom}(H).\ \text{typeOf}(o, H) = \text{typeOf}(o, H')$

Then $\text{isolation}(H', P')$ *and* $H' \vdash P'$ **ocr**.

Proof. Assumption 2,3 and 4 implies assumption 2,3 and 4 of Proposition A.8 □

Definition A.10. Let FS_g be any frame stack where

$$\text{accRoots}(FS_g, H) = \{o_g\}$$

Furthermore let $H; \epsilon \vdash FS_g$. FS_g thus is any well-typed frame stack that only has the global object as an accessible root. Now, for any thread set P, let $\tilde{P}$ be the thread set

$$\tilde{P} = P \cup \{FS_g|_\epsilon^\iota\}$$

for some fresh ι.

Proposition A.11. *For any H, P we have*

$$H \vdash P \textbf{ ocr} \iff H \vdash \tilde{P} \textbf{ ocr}$$

and

$$H \vdash P \iff H \vdash \tilde{P}.$$

Proof. Follows almost immediately from Proposition A.18 and the definitions of OCAP reachability and $\tilde{P}$. □

Proposition A.12. *For any heap H and thread set P*

$$\text{isolation}(H, \tilde{P}) \iff \text{isolation}(H, P) \textit{ and } H \vdash P \textbf{ gsep}$$

Proof. This follows almost immediately by the definitions of isolation and global object separation, and the fact that $\text{accRoots}(FS_g) = \{o_g\}$ □

We state the following without proof.

Proposition A.13. *Let $H, P \Rrightarrow^{R^{\iota,\beta}} H', P'$. Then $H, \tilde{P} \Rrightarrow^{R^{\iota,\beta}} H', \tilde{P'}$*

Proposition A.14. *For any H, H', P, P' such that $\vdash H, H \vdash P$, $H \vdash P$* **ocr** *and $H, P \Rrightarrow^{R^{\iota,\beta}} H', P'$, if*

$$\text{isolation}(H, P) \implies \text{isolation}(H', P')$$

then

$$\text{isolation}(H, \tilde{P}) \implies \text{isolation}(H', \tilde{P'})$$

Proof. By Proposition A.13, $H, \tilde{P} \Rrightarrow^{R^{\iota,\beta}} H', \tilde{P'}$. It is easy to see that $\vdash H$, $H \vdash \tilde{P}$ and $H \vdash \tilde{P}$ **ocr**. Since the assumption of the proposition refers to any H, H', P, P' for which these three properties hold the statement also holds for $H, H', \tilde{P}, \tilde{P'}$. □

Remark. This proposition may seem a bit circular, but what it does is allow us to get the global separation property for free in any case of the proof of Theorem 1 where we do not rely on the global separation property in order to prove preservation of isolation. This ends up to be a substantial number of cases. We state this in the following corollary and it can be proven by combining Proposition A.14 with Proposition A.12.

Corollary A.15. *For any H, H', P, P' such that $\vdash H, H \vdash P$, $H \vdash P$* **ocr** *and $H, P \Rrightarrow^{R^{\alpha,\beta}} H', P'$, if*

$$\text{isolation}(H, P) \implies \text{isolation}(H', P')$$

then

$$\text{isolation}(H, P) \textit{ and } H \vdash P \textbf{ gsep}$$
$$\implies$$
$$\text{isolation}(H', P') \textit{ and } H \vdash P' \textbf{ gsep}$$

Remark. In particular we can use this corollary in all cases where we can also use Proposition A.8 or its corollary A.9.

Proposition A.16. *Let H, H' be heaps. If $\text{dom}(H) = \text{dom}(H')$ and*

$$\forall o \in \text{dom}(H).\ \text{typeOf}(o, H) = \text{typeOf}(o, H')$$

then

$$\forall k \in \mathscr{V}.\ \text{typeOf}(k, H) = \text{typeOf}(k, H')$$

Proof. It is easy to see since the only values in $\mathscr{V}$ where its type depends on H is the object references o. □

Proposition A.17. *If*

$$\forall o \in \text{dom}(H) \subseteq \text{dom}(H').\ \text{typeOf}(o, H) = \text{typeOf}(o, H')$$

and $H \vdash \Gamma; L$, then $H' \vdash \Gamma; L$.

Proof. It is obvious from the structure of the rules WF-EnvVar and WF-Env and the assumptions of the proposition. □

Proposition A.18. *For any H, P we have*

$$H \vdash P \iff \forall HS|_b^{\iota'} \in P.\ H; b \vdash HS \tag{A.5}$$

Proof. (Sketch) Done in each direction separately.

Case $\implies$: By induction on the shape of the derivation tree.

Case $\impliedby$: By induction on size of P.

□

Proposition A.19. *Let* H, H' *be heaps such that* $\mathrm{dom}(H) \subseteq \mathrm{dom}(H')$ *and*

$$\forall o \in \mathrm{dom}(H).\ \mathrm{typeOf}(o, H) = \mathrm{typeOf}(o, H').$$

Then for any frame stack FS

$$H; a \vdash FS \implies H'; a \vdash FS \tag{A.6}$$

and

$$H; a \vdash^{x:\sigma} FS \implies H'; a \vdash^{x:\sigma} FS \tag{A.7}$$

Proof. We first prove (A.7) by induction on the lenght n of FS. To prove the implication we assume that $H; a \vdash^{x:\tau} FS$.

If $n = 0$ we have $FS = \varepsilon$. By T-FSEMPTY2 we are done.

If $n = i + 1$ we have $FS = F \circ GS$ where length of GS is i. By $H; a \vdash^{x:\sigma} FS$ and T-FS2 we have

$$F = \langle L, t\rangle^{y} \qquad H \vdash \Gamma; L \qquad \Gamma, x : \sigma; a \vdash t : \tau \qquad H; a \vdash^{y:\tau} GS.$$

By induction hypothesis we then have $H'; a \vdash^{y:\tau} GS$. We get $H' \vdash \Gamma; L$ from Proposition A.17. Applying rule T-FS2 yields $H'; a \vdash^{x:\sigma} FS$.

Now we prove (A.6). Similarly assuming $H; a \vdash FS$ we have two cases. If $FS = \varepsilon$ we are done by T-FSEMPTY1. If $FS = F \circ GS$ we by T-FS1 have

$$F = \langle L, t\rangle^{x} \qquad H \vdash \Gamma; L \qquad \Gamma; a \vdash t : \tau \qquad H; a \vdash^{x:\sigma} GS.$$

By (A.7), $H'; a \vdash^{x:\sigma} GS$. Proposition A.17 then yields $H' \vdash \Gamma; L$. Applying rule T-FS1 gives us $H'; a \vdash FS$. □

Proposition A.20. *For any* H, HS *we have*

$$\begin{gathered}\mathrm{ocr}(HS, H)\\ \iff\\ \forall q \in \mathrm{reachable}(\mathrm{accRoots}(HS, H), \mathscr{G}(H)).\\ \quad \mathit{ocap}(\mathrm{typeOf}(q, H))\end{gathered}$$

Proof. Follows immediately from reachability equivalence and the definition of ocr. □

A.2 Proof of preservation

Theorem (Preservation). *Let S, S' be states such that $\vdash S$ $\mathbf{ok}_p$ and $S \Rrightarrow S'$. Then $\vdash S'$ $\mathbf{ok}_p$.*

Proof. First of all we have that $S \neq \mathbf{error}$ since no step can be made from this state. Thus $S = H, P$ and

$$\begin{array}{ccc} \vdash H & H \vdash P & H \vdash P \ \mathbf{ocr} \\ \text{isolated}(H, P) & H \vdash P \ \mathbf{gsep} & \text{uniqMain}(P). \end{array}$$

We also assume we have $S' = H', P'$ since otherwise $S' = \mathbf{error}$ and the theorem holds trivially. What remains is to prove

$$\begin{array}{ccc} \vdash H' & H' \vdash P' & H' \vdash P' \ \mathbf{ocr} \\ \text{isolated}(H', P') & H' \vdash P' \ \mathbf{gsep} & \text{uniqMain}(P). \end{array}$$

We easily see that $\text{uniqMain}(P')$ must hold since no reduction rule changes the OCAP status of threads, and the only rule which spawns new threads is E-SPAWN which spawns an OCAP thread.

To prove everything else we proceed by cases.

Case E-FSPROP**:** The E-FSPROP rule definition says $P = Q \cup_D \{FS|^{\iota}_{a}\}$ and $P' = Q \cup \{FS'|^{\iota}_{a}\}$. We proceed by cases.

Case E-FPROP**:** By E-FPROP rule definition, $FS = F \circ GS$ and $FS' = F' \circ GS$. Furthermore we can assume that $F = \langle L, t\rangle^s$ and $F' = \langle L', t'\rangle^s$. We proceed by cases.

Case E-NULL**:** By this rule we have $t = \texttt{let}\ x = \texttt{null}\ \texttt{in}\ t'$, $H = H'$ and $L' = L[x \mapsto \texttt{null}]$. Immediatelly we have $\vdash H'$ by $H = H'$.

By T-PROCS and Proposition A.18 we have

$$H \vdash Q \qquad H; a \vdash FS \tag{A.8}$$

Trivially $H' \vdash Q$. By (A.8) and rule T-FS1 we have

$$\begin{array}{cc} H \vdash \Gamma; L & \Gamma; a \vdash t : \sigma' \\ \sigma' <: \sigma & H; a \vdash^{s:\sigma} GS. \end{array} \tag{A.9}$$

Let $\Gamma' = \Gamma, x : \mathsf{Null}$. By T-LET, T-NULL and (A.9) we have

$$\Gamma'; a \vdash t' : \sigma'. \tag{A.10}$$

Since $\mathrm{typeOf}(L(x), H') <: \mathsf{Null}$, by inspection of rules WF-ENVVAR and WF-ENV we can see that

$$H' \vdash \Gamma'; L' \tag{A.11}$$

By T-FS1, (A.9), (A.10) and (A.11)

$$H'; a \vdash FS' \tag{A.12}$$

By (A.12), T-PROCS and $H = H'$ we get

$$H' \vdash P' \tag{A.13}$$

$H' \vdash P'$ **ocr** and $\mathrm{isolation}(H', P')$ follows from the Corollary A.9. Then $H' \vdash P'$ **gsep** follows from Corollary A.15 (see remarks for Proposition A.14 and Corollary A.15).

Case E-LVAL**:** Similarly.

Case E-VAR**:** First, by rule E-VAR we have

$$\begin{array}{ll} F = \langle L, t\rangle^s & t = \mathtt{let}\ x = y\ \mathtt{in}\ t' \\ F' = \langle L', t'\rangle^s & L' = L[x \mapsto L(y)]. \end{array} \tag{A.14}$$

Also

$$H = H' \tag{A.15}$$

so $\vdash H'$. By $H \vdash P$, T-FS1 and Proposition A.18

$$\begin{array}{ll} H \vdash \Gamma; L & \Gamma; a \vdash t : \sigma' \\ \sigma' <: \sigma & H; a \vdash^{s:\sigma} GS \end{array} \tag{A.16}$$

By T-LET, T-VAR and $\Gamma; a \vdash t : \sigma'$

$$\Gamma, x : \gamma; a \vdash t' : \sigma' \qquad \Gamma(y) = \gamma \tag{A.17}$$

Let $\Gamma' = \Gamma, x : \gamma$. Then by (A.17)

$$\Gamma'; a \vdash t' : \sigma'. \tag{A.18}$$

By $H \vdash \Gamma; L$, (A.15), (A.18), and rules WF-ENVVAR, WF-ENV

$$\mathrm{typeOf}(L'(x), H') = \mathrm{typeOf}(L(y), H) <: \Gamma(y) = \Gamma'(x). \tag{A.19}$$

From definitions of Γ', L' and $H \vdash \Gamma; L$ we also have

$$\begin{aligned}&\forall z \in \mathrm{dom}(\Gamma') \text{ s.t. } z \neq x.\\ &\qquad \mathrm{typeOf}(L'(z), H') \leq \Gamma'(z).\end{aligned} \tag{A.20}$$

Thus by WF-EnvVar, WF-Env, (A.19) and (A.20)

$$H' \vdash \Gamma'; L' \tag{A.21}$$

By T-FS1, (A.16), (A.18) and (A.21) we have

$$H'; a \vdash FS'. \tag{A.22}$$

By Proposition A.18 and $H \vdash P$ we get

$$\forall HS|_b^{\iota'} \in Q.\ H; b \vdash HS.$$

By this, (A.15) and Proposition A.18

$$H' \vdash Q$$

Combining this with (A.22) and T-Procs finally yields

$$H' \vdash P'$$

We now move on to OCAP reachability, isolation and global object separation. We note that all preconditions of Corollary A.9 holds. Thus we immediately have

$$H' \vdash P' \ \textbf{ocr} \qquad \mathrm{isolation}(H', P').$$

$H' \vdash P'$ **gsep** immediately follows by Corollary A.15.

Case E-Select**:** By rule E-Select

$$\begin{gathered} H = H' \qquad F = \langle L, t\rangle^s \qquad t = \texttt{let}\ x\ \texttt{=}\ y\texttt{.}f\ \texttt{in}\ t' \\ L(y) = o \qquad H(o) = \langle C, FM\rangle \qquad f \in \mathrm{dom}(FM) \\ F' = \langle L', t'\rangle^s \qquad L' = L[x \mapsto FM(f)] \end{gathered} \tag{A.23}$$

We immediately see $\vdash H'$. By $\vdash H$ we have

$$\mathrm{typeOf}(FM(f), H) <: \mathrm{ftype}(f, C). \tag{A.24}$$

$H \vdash P$ and propositions A.18 and A.19 yields $H \vdash Q$ and thus $H' \vdash Q$. $H \vdash P$ and Proposition A.18 gives

$$H; a \vdash FS \tag{A.25}$$

which under inspection of rule T-FS1 immediately gives

$$\begin{array}{cc} H \vdash \Gamma; L & \Gamma; a \vdash t : \sigma' \\ \sigma' <: \sigma & H; a \vdash^{s:\sigma} GS. \end{array} \tag{A.26}$$

$\Gamma; a \vdash t : \sigma'$ together with rule T-LET gives

$$\Gamma; a \vdash y\,.\,f : \gamma \qquad \Gamma, x : \gamma; a \vdash t' : \sigma'. \tag{A.27}$$

which combined with T-SELECT and T-VAR gives us

$$(y : C) \in \Gamma \qquad \mathrm{ftype}(f, C) = \gamma \tag{A.28}$$

Let $\Gamma' = \Gamma, x : \gamma$. Similarly to case E-VAR we see that

$$\forall z \in L \text{ s.t. } z \neq x.\ \mathrm{typeOf}(L(z), H') <: \Gamma'(z). \tag{A.29}$$

Furthermore, because of (A.24) and $H = H'$

$$\begin{aligned} \mathrm{typeOf}(L'(x), H') &= \mathrm{typeOf}(FM(f), H') \\ &<: \mathrm{ftype}(f, C) \\ &= \Gamma'(x) \end{aligned} \tag{A.30}$$

Using WF-ENV, WF-ENVVAR, (A.29) and (A.30) we have

$$H' \vdash \Gamma'; L'. \tag{A.31}$$

Using T-FS1 with $H = H'$, (A.26), (A.27) and (A.31) we finally get

$$H'; a \vdash FS' \tag{A.32}$$

which with the help of T-PROCS and $H' \vdash Q$ gives us

$$H' \vdash P'.$$

Moving on to OCAP reachability, isolation and global object separation we can easily see that the preconditions of Proposition A.8 holds and thus we immediately get

$$H' \vdash P' \ \textbf{ocr} \qquad \mathrm{isolation}(H', P')$$

Again using Corollary A.15 similarly to previous cases, we get

$$H' \vdash P' \ \textbf{gsep}.$$

Case E-ASSIGN**:** The E-ASSIGN rule gives us that

$$\begin{gathered} F = \langle L, t\rangle^s \qquad t = \texttt{let}\ x \texttt{ = } y\,\texttt{.}\,f \texttt{ = } z\ \texttt{in}\ t' \\ F' = \langle L', t'\rangle^s \qquad L' = L[x \mapsto L(z)] \\ L(y) = o_y \qquad H(o) = \langle C, FM\rangle \qquad f \in \mathrm{dom}(FM) \\ FM' = FM[f \mapsto L(z)] \qquad H' = H[o \mapsto \langle C, FM'\rangle] \end{gathered} \tag{A.33}$$

We begin by proving $\vdash H'$. First we note that only change from H to H' is in what object o_y maps to. Clearly from (A.33) we have

$$\forall o \in \mathrm{dom}(H) = \mathrm{dom}(H').\ \mathrm{typeOf}(o, H) = \mathrm{typeOf}(o, H') \tag{A.34}$$

Thus by Proposition A.16

$$\forall k \in \mathscr{V}.\ \mathrm{typeOf}(k, H) = \mathrm{typeOf}(k, H') \tag{A.35}$$

Remark. This means that immediately it is clear that the conditions for $\vdash H'$ (see equations (5.3) and (5.4)) holds for all $o' \in \mathrm{dom}(H')$ s.t. $o' \neq o_y$.

By $\vdash H$, definition of H' and (A.35)

$$\begin{aligned} &\forall f' \in \mathrm{fields}(C), f' \neq f. \\ &\quad \mathrm{typeOf}(FM'(f'), H') <: \mathrm{ftype}(f', C) \end{aligned} \tag{A.36}$$

By $H \vdash P$ and Proposition A.18

$$H; a \vdash FS. \tag{A.37}$$

This and T-FS1 yields

$$\begin{aligned} H \vdash \Gamma; L \qquad & \Gamma; a \vdash t : \sigma' \\ \sigma' <: \sigma \qquad & H; a \vdash^{s:\sigma} GS \end{aligned} \tag{A.38}$$

By $H; a \vdash^{s:\sigma} GS$, (A.34) and prop. A.19 we have

$$H'; a \vdash^{s:\sigma} GS \tag{A.39}$$

$\Gamma; a \vdash t : \sigma'$ and T-LET gives

$$\Gamma; a \vdash y\,\texttt{.}\,f \texttt{ = } z : \gamma \qquad \Gamma, x : \gamma; a \vdash t' : \sigma' \tag{A.40}$$

which whith T-ASSIGN gives us

$$\begin{gathered}\Gamma; a \vdash y : C' \qquad \text{ftype}(f, C') = \alpha \\ \Gamma; a \vdash z : \alpha' \qquad \alpha' <: \alpha\end{gathered} \tag{A.41}$$

By T-VAR and $\Gamma; a \vdash y : C'$

$$(y : C') \in \Gamma \tag{A.42}$$

or equivalently $\Gamma(y) = C'$. Similarly

$$(z : \alpha') \in \Gamma. \tag{A.43}$$

By $H \vdash \Gamma; L$ we have $H \vdash \Gamma; L; z$ and thus

$$\begin{aligned}\text{typeOf}(L(z), H) &= \alpha'' \\ &<: \alpha' \\ &<: \alpha \\ &= \text{ftype}(f, C') \\ &= \text{ftype}(f, C)\end{aligned} \tag{A.44}$$

where the last equality comes from the fact that the classes are well-formed. By (A.33) we have $FM'(f) = L(z)$ and combining this with (A.44) we get

$$\text{typeOf}(FM'(f), H) <: \text{ftype}(f, C). \tag{A.45}$$

Combined with (A.35), (A.36) and the remark above we get

$$\vdash H'$$

$H' \vdash P'$ follows similarly to E-VAR case.

We now move on to OCAP reachability, isolation and global object separation. Note that the only insteresting case is if $L(z)$ is an object refence, since otherwise the preconditions to Corollary A.9 holds and we are done similarly to previous cases. Therefore we assume $L(z) \in \text{dom}(H)$. To help our efforts we state the following lemma.

Lemma A.21. *We let everything be as in case* E-ASSIGN *above. Furthermore we let $L(z) \in \text{dom}(H)$. Then*

$$\begin{gathered}\text{reachable}(\text{accRoots}(FS', H'), \mathscr{G}(H')) \subseteq \\ \text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H))\end{gathered} \tag{A.46}$$

If $HS|_b^{\iota'} \in Q$

$$\begin{aligned}&\text{reachable}(\text{accRoots}(HS, H'), \mathscr{G}(H')) \subseteq\\&\text{reachable}(\text{accRoots}(HS, H), \mathscr{G}(H)) \cup\\&\text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H)).\end{aligned} \tag{A.47}$$

Furthermore, if $a = ocap$ *or* $b = ocap$ *then*

$$\begin{aligned}&\text{reachable}(\text{accRoots}(HS, H'), \mathscr{G}(H')) \subseteq\\&\text{reachable}(\text{accRoots}(HS, H), \mathscr{G}(H))\end{aligned} \tag{A.48}$$

Proof. Let $\mathscr{G}(H) = (V_H, E_H)$. Then clearly

$$\begin{aligned}\mathscr{G}(H') &= (V_H, E_{H'})\\E_{H'} &= (E_H \setminus \{(o_y, FM(f))_f\}) \cup \{(o_y, o_z)_f\}\end{aligned} \tag{A.49}$$

We prove each part separately. We begin with (A.46). Let

$$q \in \text{reachable}(\text{accRoots}(FS', H'), \mathscr{G}(H')).$$

Because of reachability equivalence (Proposition A.4) this means there is an $o \in \text{accRoots}(FS', H')$ such that

$$\exists \text{ path } o_0, \ldots, o_n \in \mathscr{G}(H') \text{ s.t. } o_0 = o, o_n = q \tag{A.50}$$

Let f_i be the associated field names from which the path is formed, i.e. $(o_i, o_{i+1})_{f_i} \in \mathscr{G}(H')$. By (A.33)

$$\text{accRoots}(FS', H') \subseteq \text{accRoots}(FS, H). \tag{A.51}$$

Clearly $o \in \text{accRoots}(FS, H)$.

There are two cases:

1. For some $i \in \{0, \ldots, n-1\}$ $o_i = o_y, o_{i+1} = o_z$ and $f_i = f$, i.e. the edge $(o_y, o_z)_f$ is part of the path.
2. There is no such i.

We prove each case separately. For case 1 let i be the last such index. Examine the path

$$o_{i+1}, \ldots, o_n \tag{A.52}$$

Clearly this path is in $\mathscr{G}(H)$ since we chose i to be the last index that fulfills case 1. Since $o_{i+1} = o_z$ and o_z obviously is in $\text{accRoots}(FS, H)$

$$q \in \text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H)). \tag{A.53}$$

For case 2 the new edge $(o_y, o_z)_f$ is not part of the path. Thus all edges $(o_i, o_{i+1})_{f_i}$ are in $\mathscr{G}(H)$ and and thus the path $o_0, \ldots, o_n$ is a path in $\mathscr{G}(H)$ aswell. This means $q \in \text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H))$ which concludes the proof of the first part of the lemma.

To prove (A.47) take any

$$q \in \text{reachable}(\text{accRoots}(HS, H'), \mathscr{G}(H')). \tag{A.54}$$

Similarly to before there is an $o \in \text{accRoots}(HS, H')$ such that

$$\exists \text{ path } o_0, \ldots, o_n \in \mathscr{G}(H') \text{ s.t. } o_0 = o, o_n = q \tag{A.55}$$

We let f_i be the corresponding field names as before. We have exactly the same cases. For case 1 we let i be the last such index. Then similarly to before we get that $o_{i+1} \in \text{accRoots}(FS, H)$ and $q \in \text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H))$.

Case 2 is analogous and reaches the conclusion that $q \in \text{reachable}(\text{accRoots}(HS, H), \mathscr{G}(H))$.

Lastly we prove that if $a = ocap$ or $b = ocap$ then (A.48) holds. To prove this implication we assume that either $a = ocap$ or $b = ocap$. As before we take any

$$q \in \text{reachable}(\text{accRoots}(HS, H'), \mathscr{G}(H')). \tag{A.56}$$

Similarly we get that there there must be an $o \in \text{accRoots}(HS, H')$ such that

$$\exists \text{ path } o_0, \ldots, o_n \in \mathscr{G}(H') \text{ s.t. } o_0 = o, o_n = q \tag{A.57}$$

with related field names f_i as before. We get the exact same cases as above. For case 1 we instead let i be the first index with the properties described. As noted this means we have the edge $(o_y, o_z)_f$ as part of our path. Specifically this means

$$o_y \in \text{reachable}(\text{accRoots}(HS, H), \mathscr{G}(H)) \tag{A.58}$$

since $o_0, \ldots, o_i$ is a path in $\mathscr{G}(H)$ from o to o_y, due to the way i was chosen. Moreover

$$o_y \in \text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H)) \tag{A.59}$$

since $(y \mapsto o_y) \in L$. From (A.33) it is clear that

$$\mathrm{typeOf}(o_y, H) \not<: \mathsf{Cell}. \tag{A.60}$$

Now (A.58), (A.59) and (A.60) contradicts $\mathrm{isolation}(H, P)$ since $b = ocap$. Since case 1 leads to a contradiction, it is impossible.

For case 2 it is clear that $o_0, \dots, o_n$ is also a path from o to q in $\mathscr{G}(H)$ which immediately implies

$$q \in \mathrm{reachable}(\mathrm{accRoots}(HS, H), \mathscr{G}(H)). \tag{A.61}$$

$\square$

Continuing with the proof of case E-ASSIGN, first off we use the lemma to prove OCAP reachability. We first take any thread $HS|_b^{\iota'} \in Q$. Studying the **ocr** rules it is clear that we can prove that $H'; b \vdash HS$ **ocr** holds trivially for any thread $HS|_b^{\iota'}$ for which $b = \epsilon$. Therefore we from now on assume that $b = ocap$. By $H \vdash P$ **ocr**, $H; b \vdash FS$ **ocr**. By Proposition A.20 and $b = ocap$,

$$\begin{aligned}&\forall o \in \mathrm{reachable}(\mathrm{accRoots}(HS, H), H).\\&\quad ocap(\mathrm{typeOf}(o, H)).\end{aligned} \tag{A.62}$$

Lemma A.21 (equation (A.48)) and equations (A.34) and (A.62) means

$$\begin{aligned}&\forall o \in \mathrm{reachable}(\mathrm{accRoots}(HS, H'), \mathscr{G}(H')).\\&\quad ocap(\mathrm{typeOf}(o, H')).\end{aligned} \tag{A.63}$$

But this is equivalent to $H'; b \vdash HS$ **ocr** by Proposition A.20.

In order to prove $H' \vdash P'$ **ocr** the only thing that remains is to show $H'; a \vdash FS'$ **ocr**. This is anologous to the above reasoning about $H'; b \vdash HS$ **ocr** using (A.46) instead. Thus we have

$$H' \vdash P' \ \textbf{ocr}.$$

$\mathrm{isolation}(H', P')$ follows immediately from $\mathrm{isolation}(H, P)$, Proposition A.7, Lemma A.21, equation (A.34) and applying rule ISO-PROCS. We also note that nowhere have we relied on the fact that $H \vdash P$ **gsep** so using Corollary A.15 we also have $H' \vdash P'$ **gsep**.

Case E-NEW**:** According to rule E-NEW

$$\begin{gathered} F = \langle L, t \rangle^s \qquad t = \texttt{let}\, x = \texttt{new}\, C\, \texttt{in}\, t' \\ F' = \langle L', t' \rangle^s \qquad L' = L[x \mapsto o_x] \\ o_x \text{ fresh object reference} \\ FM = [f \mapsto \text{default}(\sigma) \mid (\texttt{var}\, f : \sigma) \in \text{fdecls}(C)] \\ H' = H[o_x \mapsto \langle C, FM \rangle] \end{gathered} \tag{A.64}$$

If we let $\mathscr{G}(H) = (V_H, E_H)$ it is easy to see that

$$\mathscr{G}(H') = (V_H \cup \{o_x\}, E_H) \tag{A.65}$$

By definition of default

$$\begin{aligned} &\forall f \in \text{fields}(C). \\ &\quad f \in \text{dom}(FM) \wedge \text{typeOf}(FM(f), H') <: \text{ftype}(f, C). \end{aligned} \tag{A.66}$$

Also

$$\forall o \in \text{dom}(H).\, H(o) = H'(o), \tag{A.67}$$

and thus

$$\forall o \in \text{dom}(H).\, \text{typeOf}(o, H) = \text{typeOf}(o, H'). \tag{A.68}$$

Using the definition of a well-typed heap together with (A.66), (A.67) and (A.68) we can easily show that $\vdash H'$.

By Proposition A.18

$$H \vdash Q \qquad H; a \vdash FS. \tag{A.69}$$

Propositions A.18 and A.19 implies

$$H' \vdash Q. \tag{A.70}$$

$H; a \vdash FS$ and T-FS1 yields

$$\begin{gathered} H \vdash \Gamma; L \qquad \Gamma; a \vdash t : \sigma' \\ \sigma' <: \sigma \qquad H; a \vdash^{s:\sigma} GS. \end{gathered} \tag{A.71}$$

Applying T-LET and T-NEW we get

$$\Gamma, x : C; a \vdash t' : \sigma' \qquad a = ocap \implies ocap(C). \tag{A.72}$$

Letting $\Gamma' = \Gamma, x : C$ we have

$$\Gamma'; a \vdash t' : \sigma'. \tag{A.73}$$

We have that $H' \vdash \Gamma'; L'; x$ since $\text{typeOf}(L'(x), H') <: \Gamma'(x)$. Thus we have

$$H' \vdash \Gamma'; L'. \tag{A.74}$$

By Proposition A.19

$$H'; a \vdash^{s:\sigma} GS. \tag{A.75}$$

Using T-FS1 together with $\sigma' <: \sigma$, (A.73), (A.74) and (A.75) we get

$$H'; a \vdash FS', \tag{A.76}$$

and combined with (A.70) and T-PROCS we finally get

$$H' \vdash P'. \tag{A.77}$$

Moving on to OCAP reachability. For any $HS|_b^{\iota'} \in Q$, we see that

$$\begin{aligned}&\text{reachable}(\text{accRoots}(HS, H'), \mathscr{G}(H')) = \\ &\text{reachable}(\text{accRoots}(HS, H), \mathscr{G}(H))\end{aligned} \tag{A.78}$$

and furthermore that

$$\begin{aligned}&\text{reachable}(\text{accRoots}(FS', H'), \mathscr{G}(H')) = \\ &\text{reachable}(\text{accRoots}(FS, H), \mathscr{G}(H)) \cup \{o_x\}\,.\end{aligned} \tag{A.79}$$

Clearly $\text{typeOf}(o_x, H') = C$. Combining this with (A.68), (A.72), (A.78), (A.79), OCR-P, $H \vdash P$ **ocr** and Proposition A.20 we see that

$$H' \vdash P' \ \textbf{ocr}. \tag{A.80}$$

Similarly it is easy to show $\text{isolation}(H', P')$ using equations (A.68), (A.78) and (A.79) together with Proposition A.7 and ISO-PROCS. Since the isolation proof never relies on $H \vdash P$**gsep** we can apply Corollary A.15.

$$\text{isolation}(H', P') \qquad H' \vdash P' \ \textbf{gsep} \tag{A.81}$$

Case E-NEWCELL**:** We have

$$H' = H[o \mapsto \langle \mathsf{Cell}, DEP, \bot_{\mathscr{L}} \rangle] \qquad DEP = \emptyset \tag{A.82}$$

for some fresh object reference o. Since DEP is empty (5.4) holds vacuously and since all other heap elements are the same as in H we have

$$\vdash H'. \tag{A.83}$$

Proving the rest is similar to case E-NEW.

Case E-PUT**:** According to rule E-PUT

$$\begin{gathered} F = \langle L, t \rangle^s \qquad t = \texttt{let}\ x = y\ \texttt{put}\ z\ \texttt{in}\ t' \\ F' = \langle L', t' \rangle^s \qquad L' = L[x \mapsto L(y)] \\ L(y) = o_y \qquad H(o_y) = \langle \mathsf{Cell}, DEP, l \rangle \\ L(z) = l' \qquad H' = H[o_y \mapsto \langle \mathsf{Cell}, DEP, l \sqcup l' \rangle]. \end{gathered} \tag{A.84}$$

Since the only change to the heap is l being updated to $l \sqcup l'$, by inspecting the definition for a well-typed heap and using $\vdash H$ it is easy to conclude that

$$\vdash H'. \tag{A.85}$$

The proof of $H' \vdash P'$ follows similar to case E-VAR. The preconditions of Corollary A.9 clearly hold. Thus

$$H' \vdash P' \ \textbf{ocr} \qquad \text{isolation}(H', P'). \tag{A.86}$$

We apply Corollary A.15 to get

$$H' \vdash P' \ \textbf{gsep} \tag{A.87}$$

Case E-WHEN**:** From rule E-WHEN we know

$$\begin{gathered} F = \langle L, t \rangle^s \\ t = \texttt{let}\ x = \texttt{when}\ y\ \texttt{pass}\ z\ \texttt{then}\ (\overline{cap}, w \Rightarrow t'')\ \texttt{in}\ t' \\ F' = \langle L', t' \rangle^s \qquad L' = L[x \mapsto L(y)] \\ L(y) = o_y \qquad H(o_y) = \langle \mathsf{Cell}, DEP, l \rangle \qquad L(z) = l' \\ L_{\text{env}} = [u \mapsto L(u') \mid (u = u') \in \overline{cap}] \qquad cb = (L_{\text{env}}, w \Rightarrow t'') \\ \iota \text{ fresh thread id} \qquad DEP' = DEP \cup (l', cb)^{\iota} \\ H' = H[o_y \mapsto \langle \mathsf{Cell}, DEP', l \rangle]. \end{gathered} \tag{A.88}$$

We note that $\mathrm{dom}(H) = \mathrm{dom}(H')$ and that

$$\begin{aligned}&\forall o \in \mathrm{dom}(H).\\ &\quad \mathrm{typeOf}(o, H) = \mathrm{typeOf}(o, H'),\end{aligned} \tag{A.89}$$

and thus

$$\begin{aligned}&\forall k \in \mathscr{V}.\\ &\qquad \mathrm{typeOf}(k, H) = \mathrm{typeOf}(k, H').\end{aligned} \tag{A.90}$$

Combining this with the fact that the only object on heap changed is $H(o_y)$ we have shown $\vdash H'$ if we can show that

$$\begin{aligned}&\forall (l^{\mathrm{cb}}, (L^{\mathrm{cb}}_{\mathrm{env}}, z^{\mathrm{cb}} \Rightarrow t^{\mathrm{cb}}))^{\iota^{\mathrm{cb}}} \in \mathit{DEP}'.\\ &\quad \forall (x \mapsto k) \in L^{\mathrm{cb}}_{\mathrm{env}}.\ \mathrm{typeOf}(k, H') <: \mathsf{Cell} \wedge\\ &\quad \Gamma_{\mathsf{Cell}}(L^{\mathrm{cb}}_{\mathrm{env}}), z^{\mathrm{cb}} : \mathcal{L}; \mathit{ocap} \vdash t^{\mathrm{cb}} : \gamma.\end{aligned} \tag{A.91}$$

It is simple to see that, because of $\vdash H$ and that the only difference between DEP and DEP' is the addition of $(l', (L_{\mathrm{env}}, w \Rightarrow t''))$, we have proved $\vdash H'$ if we can prove

$$\forall (x \mapsto k) \in L_{\mathrm{env}}.\ \mathrm{typeOf}(k, H') <: \mathsf{Cell} \tag{A.92}$$

and

$$\Gamma_{\mathsf{Cell}}(L_{\mathrm{env}}), w : \mathcal{L}; \mathit{ocap} \vdash t'' : \gamma. \tag{A.93}$$

Similarly to earlier cases, using $H \vdash P$ together with typing rules such as T-LET and T-WHEN we can easily get

$$H \vdash \Gamma; L \qquad \forall (u = u') \in \overline{cap}.\ \Gamma; a \vdash u' : \mathsf{Cell} \tag{A.94}$$

and

$$\begin{aligned}\Gamma_{\mathrm{cells}} &= [u \mapsto \mathsf{Cell} \mid (u = u') \in \overline{cap}]\\ \Gamma_{\mathrm{cells}}, w &: \mathcal{L}; \mathit{ocap} \vdash t'' : \gamma'.\end{aligned} \tag{A.95}$$

(A.95) is just a different formulation of (A.93) since the actual value of γ does not matter. Moreover, using equations (A.89) and (A.94) it is simple to prove (A.92) with the help of rules WF-ENVVAR, WF-ENV and T-VAR. Thus we are done and have

$$\vdash H'. \tag{A.96}$$

Proof of $H' \vdash P'$ is similar to case E-VAR and similarly to other cases we can apply corollaries A.9 and A.15 to get

$$H' \vdash P' \ \textbf{ocr} \qquad \text{isolation}(H', P') \qquad H' \vdash P' \ \textbf{gsep}. \tag{A.97}$$

This concludes the case E-FPROP.

Case E-CALL**:** From rule E-CALL

$$\begin{gathered} FS = \langle L, t\rangle^s \circ GS \qquad t = \texttt{let}\ x \texttt{ = } y\texttt{.}m\texttt{(}z\texttt{)}\ \texttt{in}\ t'' \\ FS' = \langle L', t'\rangle^x \circ \langle L, t\rangle^s \circ GS \\ L(y) = o_y \qquad H(o_y) = \langle C, FM\rangle \\ \text{mbody}(m, C) = u \to t' \\ L_{\text{base}} = \begin{cases} \emptyset & \text{if } a = ocap \\ L_0 & \text{if } a = \epsilon \end{cases} \\ L' = L_{\text{base}}[\texttt{this} \mapsto L(y), u \mapsto L(z)] \\ H = H'. \end{gathered} \tag{A.98}$$

Since $H = H'$

$$\vdash H'.$$

By $H = H'$, $H \vdash P$ and propositions A.18, and A.19, we have

$$H' \vdash Q \qquad H; a \vdash FS. \tag{A.99}$$

Using the second of these and rule T-FS1 we have

$$\begin{gathered} H \vdash \Gamma; L \qquad \Gamma; a \vdash t : \sigma' \\ \sigma' <: \sigma \qquad H; a \vdash^{s:\sigma} GS. \end{gathered} \tag{A.100}$$

Since $H = H'$

$$H' \vdash \Gamma; L \qquad H'; a \vdash^{s:\sigma} GS. \tag{A.101}$$

Using $\Gamma; a \vdash t : \sigma$ with rules T-LET, T-CALL and T-VAR we have that

$$\begin{gathered} \Gamma; a \vdash y\texttt{.}m\texttt{(}z\texttt{)} : \tau \qquad \Gamma, x : \tau; a \vdash t'' : \sigma' \\ \Gamma(y) = C' \qquad \text{mtype}(m, C') = \gamma \mapsto \tau \\ \Gamma(z) = \gamma' \qquad \gamma' \leq \gamma. \end{gathered} \tag{A.102}$$

Using T-FS2 with (A.100), (A.101) and (A.102) we get

$$H'; a \vdash^{x:\tau} FS. \tag{A.103}$$

By $H' \vdash \Gamma; L$ we have

$$\text{typeOf}(L'(u), H') = \text{typeOf}(L(z), H') <: \gamma' <: \gamma, \tag{A.104}$$

and similarly

$$\text{typeOf}(L'(\texttt{this}), H') = \text{typeOf}(L(y), H') <: C <: C'. \tag{A.105}$$

Now, a can be either ϵ or $ocap$. We first assume that $a = \epsilon$. By our program and thus by extension our classes being well-formed, together with $C <: C'$, we must have a class C'' s.t. $C <: C'' <: C'$ such that the the class definition of C'' includes the method definition

$$\texttt{def}\ m\texttt{(}u\ \texttt{:}\ \gamma\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ t'.$$

By C'' being well-formed

$$C'' \vdash \texttt{def}\ m\texttt{(}u\ \texttt{:}\ \gamma\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ t'. \tag{A.106}$$

By WF-METHOD this means that

$$\Gamma_0, \texttt{this} : C'', u : \gamma; \epsilon \vdash t' : \tau' \qquad \tau' <: \tau. \tag{A.107}$$

We let

$$\Gamma'_\epsilon = \Gamma_0, \texttt{this} : C'', u : \gamma. \tag{A.108}$$

It is not hard to prove that

$$H' \vdash \Gamma'_\epsilon; L'. \tag{A.109}$$

By (A.107) we clearly have

$$\Gamma'_\epsilon; a \vdash t' : \tau' \qquad \tau' <: \tau. \tag{A.110}$$

Using (A.103), (A.109), (A.110) and rule T-FS1 we have

$$H'; a \vdash FS'. \tag{A.111}$$

Combining this with $H' \vdash Q$ and T-PROCS we get

$$H' \vdash P'. \tag{A.112}$$

Now assume that $a = ocap$. By $H \vdash P$ **ocr** we have

$$\mathrm{ocr}(FS, H) \tag{A.113}$$

and thus by Proposition A.20

$$\begin{aligned} &\forall q \in \mathrm{reachable}(\mathrm{accRoots}(FS, H), \mathscr{G}(H)) \\ &\quad ocap(\mathrm{typeOf}(q, H)). \end{aligned} \tag{A.114}$$

Of course this means that $ocap(\mathrm{typeOf}(L(y), H))$ or equivalently $ocap(C)$. Proceeding similarly to the case $a = \epsilon$ we additionally have $ocap(C'')$ due to rule OCAP-CLASS, which demands that parent classes to C should be ocap aswell. Again using OCAP-CLASS, we get that

$$C'' \vdash_{ocap} \texttt{def}\ m\texttt{(}u\ \texttt{:}\ \gamma\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ t'. \tag{A.115}$$

By OCAP-METHOD we get that

$$\texttt{this} : C'', u : \gamma; \epsilon \vdash t' : \tau' \qquad \tau' <: \tau. \tag{A.116}$$

The rest is similar to the case where $a = \epsilon$, replacing Γ'_ϵ with

$$\Gamma'_{ocap} = \texttt{this} : C'', u : \gamma. \tag{A.117}$$

We now prove isolation, OCAP reachability and global object separation. Instead of working with P directly we instead use $\tilde{P}$. It is clear from Propositions A.12 and A.11 that in order to show

$$H' \vdash P' \ \textbf{ocr} \qquad \mathrm{isolation}(H', P') \qquad H' \vdash P' \ \textbf{gsep} \tag{A.118}$$

we can instead show

$$H' \vdash \tilde{P'} \ \textbf{ocr} \qquad \mathrm{isolation}(H', \tilde{P'}). \tag{A.119}$$

To show this we start by using propositions A.12 and A.11 to get

$$H \vdash \tilde{P} \ \textbf{ocr} \qquad \mathrm{isolation}(H, \tilde{P}). \tag{A.120}$$

We also note that

$$\tilde{P} = \tilde{Q} \cup_D \{FS|^{\iota}_{a}\} \text{ and } \tilde{P'} = \tilde{Q} \cup_D \{FS'|^{\iota}_{a}\}. \tag{A.121}$$

We first assume that $a = ocap$. It is easy to see that

$$\mathrm{accRoots}(FS', H') \subseteq \mathrm{accRoots}(FS, H) \tag{A.122}$$

and that

$$\begin{aligned}&\forall HS|_b^{\iota'} \in \tilde{Q}.\\ &\qquad \mathrm{accRoots}(HS, H') = \mathrm{accRoots}(HS, H).\end{aligned} \tag{A.123}$$

Clearly then, the preconditions to Corollary A.9 holds and we have $\mathrm{isolation}(H', \tilde{P}')$ and $H' \vdash \tilde{P}'$ **ocr**. Thus we are done.

Now instead assume $a = \epsilon$. Since $\mathrm{isolation}(H, \tilde{P})$ we have $\mathrm{isolated}(H, HS_1, HS_2)$ for any two distinct $HS_1|_b^{\iota'}, HS_2|_c^{\iota''} \in \tilde{Q}$ such that $b = ocap$ or $c = ocap$. Therefore, since $H = H'$

$$\begin{aligned}&\forall \text{ distinct } HS_1|_b^{\iota'}, HS_2|_c^{\iota''} \in \tilde{Q}.\\ &b = ocap \lor c = ocap \implies \mathrm{isolated}(H', HS_1, HS_2).\end{aligned} \tag{A.124}$$

Thus all we need for $\mathrm{isolation}(H', \tilde{P}')$ is to show that

$$\begin{aligned}&\forall HS|_b^{\iota'} \in \tilde{Q}.\\ &\qquad b = ocap \implies \mathrm{isolated}(H', FS', HS)\end{aligned} \tag{A.125}$$

since $a = \epsilon$. To prove this implication we take any $HS|_b^{\iota'} \in \tilde{Q}$ such that $b = ocap$. Note that this implies $HS \neq FS_g$. Because of $a = \epsilon$ and the definition of L' in (A.98)

$$\mathrm{accRoots}(FS', H') = \mathrm{accRoots}(FS, H) \cup \{o_g\}. \tag{A.126}$$

By $\mathrm{isolation}(H, \tilde{P})$ and $b = ocap$,

$$\mathrm{isolated}(H, HS, FS_g) \tag{A.127}$$

and

$$\mathrm{isolated}(H, FS, HS). \tag{A.128}$$

By definition

$$\mathrm{accRoots}(FS_g, H) = \{o_g\}. \tag{A.129}$$

By inspecting (A.126), (A.127), (A.129) and (A.128) and considering Proposition A.7 it is fairly easy to show

$$\mathrm{isolated}(H, FS', HS). \tag{A.130}$$

Specifically, to show this, assume for a contradiction that we do not have $\mathrm{isolated}(H, FS', HS)$. This would entail that

$$\exists q \in \mathrm{dom}(H), o \in \mathrm{accRoots}(HS, H), o' \in \mathrm{accRoots}(FS', H). \\ \mathrm{typeOf}(q, H) \not<: \mathsf{Cell} \wedge \mathrm{reach}(H, o, q) \wedge \mathrm{reach}(H, o', q). \tag{A.131}$$

If $o' = o_g$ this immediately contradicts $\mathrm{isolated}(H, HS, FS_g)$ by Proposition A.7. If we instead have $o' \neq o_g$ then by (A.126) we must have

$$o' \in \mathrm{accRoots}(FS, H). \tag{A.132}$$

Similarly this would contradict (A.128). Since $HS|_b^{t'}$ was arbitrary and $H = H'$ we have

$$\mathrm{isolation}(H', \tilde{P'}). \tag{A.133}$$

By $H \vdash \tilde{P}$ **ocr**, $H = H'$ and Proposition A.11 we get

$$H \vdash \tilde{Q} \ \mathbf{ocr}. \tag{A.134}$$

Since $a = \epsilon$ we have that

$$H'; a \vdash FS' \ \mathbf{ocr} \tag{A.135}$$

follows vacuously. Using rule OCR-P together with (A.134) and (A.135) we get

$$H' \vdash \tilde{P'} \ \mathbf{ocr}. \tag{A.136}$$

Thus we are done with case $a = \epsilon$.

Case E-RET**:** Clearly from rule E-RET we have

$$\begin{gathered} FS = \langle L, z \rangle^x \circ \langle L', t' \rangle^s \circ GS \\ FS' = \langle L'', t' \rangle^s \circ GS \\ L'' = L'[x \mapsto L(z)] \\ H = H'. \end{gathered} \tag{A.137}$$

By $H \vdash P$ and Proposition A.18 we have

$$H \vdash Q \qquad H; a \vdash FS. \tag{A.138}$$

$H = H'$ immediately yields

$$H' \vdash Q. \tag{A.139}$$

$H; a \vdash FS$ together with rule T-FS1 gives us that

$$\begin{gathered} H \vdash \Gamma; L \qquad \Gamma; a \vdash z : \tau' \\ \tau' <: \tau \qquad H; a \vdash^{x:\tau} \langle L', t' \rangle^s \circ GS, \end{gathered} \tag{A.140}$$

the last of which together with T-FS2 gives us

$$\begin{gathered} H \vdash \Gamma'; L' \qquad \Gamma', x : \tau; a \vdash t' : \sigma' \\ \sigma' <: \sigma \qquad H; a \vdash^{s:\sigma} GS \end{gathered} \tag{A.141}$$

We let $\Gamma'' = \Gamma', x : \tau$. By $H = H'$, $H \vdash \Gamma; L$ and $\tau' <: \tau$, it is obvious that

$$\text{typeOf}(L''(x), H') = \text{typeOf}(L(z), H') <: \tau. \tag{A.142}$$

Using $H \vdash \Gamma'; L'$, (A.142) and rules WF-EnvVar, WF-Env we get

$$H' \vdash \Gamma'', L''. \tag{A.143}$$

By definition of Γ'', (A.141), (A.143) and rule T-FS1 we have that

$$H'; a \vdash FS' \tag{A.144}$$

Applying T-Procs together with (A.138) and (A.144) we finally get

$$H' \vdash P' \tag{A.145}$$

Similarly to many previous cases we can apply corollarys A.9 and A.15 to get

$$H' \vdash P' \ \textbf{ocr} \qquad \text{isolation}(H', P') \qquad H' \vdash P' \ \textbf{gsep}$$

This concludes the E-FSProp **case.**

Case E-Spawn**:** By rule E-Spawn we have

$$\begin{gathered} o \in \text{dom}(H) \qquad H(o) = \langle \mathsf{Cell}, DEP, l \rangle \\ l' \sqsubseteq l \qquad (l', cb)^\iota \in DEP \qquad cb = (L_{\text{env}}, z \Rightarrow t') \\ L' = L_{\text{env}}[z \mapsto l'] \\ H' = H[o \mapsto \langle \mathsf{Cell}, DEP - (l', cb)^\iota, l \rangle] \\ P' = P \cup \left\{ FS'|^\iota_{ocap} \right\} \\ FS' = \langle L', t' \rangle^- \circ \varepsilon. \end{gathered} \tag{A.146}$$

By definition of H', $\vdash H$ and equation (5.4) from the definition of the well-typed heap it should be fairly obvious that

$$\vdash H' \tag{A.147}$$

since the only heap change is a removal of an element from DEP.

We note that

$$\begin{aligned}\forall o \in \mathrm{dom}(H) = \mathrm{dom}(H').\\ \mathrm{typeOf}(o, H) = \mathrm{typeOf}(o, H').\end{aligned} \tag{A.148}$$

By propositions A.18 and A.19 we have

$$H' \vdash P \tag{A.149}$$

From $\vdash H$ we can see that

$$\begin{aligned}\forall (x \mapsto k) \in L_{\mathrm{env}}.\\ \mathrm{typeOf}(k, H) <: \mathsf{Cell}\end{aligned} \tag{A.150}$$

and

$$\Gamma_{\mathsf{Cell}}(L_{\mathrm{env}}), z : \mathcal{L}; \mathit{ocap} \vdash t' : \gamma. \tag{A.151}$$

Letting

$$\Gamma' = \Gamma_{\mathsf{Cell}}(L_{\mathrm{env}}), z : \mathcal{L} \tag{A.152}$$

we can use (A.148),(A.150) and rules WF-ENVVAR, WF-ENV to get

$$H' \vdash \Gamma', L' \tag{A.153}$$

By rules T-FSEMPTY2, T-FS1 and (A.151), (A.153) we get

$$H'; \mathit{ocap} \vdash \mathit{FS}' \tag{A.154}$$

which combined with (A.149) and T-PROCS yields

$$H' \vdash P'. \tag{A.155}$$

Next we prove OCAP reachability. It is easy to see that

$$\mathscr{G}(H') = \mathscr{G}(H). \tag{A.156}$$

Because of this, $H \vdash P$ **ocr**, (A.148) and Proposition A.20,

$$H' \vdash P \ \textbf{ocr}. \tag{A.157}$$

This leaves only to prove

$$H'; ocap \vdash FS' \ \ \mathbf{ocr}. \tag{A.158}$$

in order to get $H' \vdash P' \ \ \mathbf{ocr}$. Inspecting OCR-FS we note that we are done if we can prove

$$\mathrm{ocr}(FS', H'). \tag{A.159}$$

But this is easy. By definition of FS'

$$\forall o \in \mathrm{accRoots}(FS', H').\ \mathrm{typeOf}(o, H') <: \mathsf{Cell} \tag{A.160}$$

which implies

$$\begin{aligned}&\forall o \in \mathrm{reachable}(\mathrm{accRoots}(FS', H'), \mathscr{G}(H')).\\&\quad \mathrm{typeOf}(o, H') <: \mathsf{Cell}\end{aligned} \tag{A.161}$$

The only objects of type Cell on the heap are cell objects whose exact type are Cell, which is *ocap* by OCAP-CELL. Combined with (A.161) this yields

$$\begin{aligned}&\forall o \in \mathrm{reachable}(\mathrm{accRoots}(FS', H'), \mathscr{G}(H')).\\&\quad ocap(\mathrm{typeOf}(o, H')).\end{aligned} \tag{A.162}$$

By Proposition A.20 we have shown (A.159).

Continuing by showing isolation we note that by (A.148), (A.156) and A.7 we have that

$$\mathrm{isolation}(H', P). \tag{A.163}$$

Thus by Proposition A.7 we are done if we can prove

$$HS|_b^{\iota'} \in P.\ \mathrm{isolated}(H', FS', HS). \tag{A.164}$$

Again using Proposition A.7 we are done if we show

$$\begin{aligned}\forall q \in\ &\mathrm{reachable}(\mathrm{accRoots}(FS', H'), \mathscr{G}(H')) \cap\\&\quad \mathrm{reachable}(\mathrm{accRoots}(HS, H'), \mathscr{G}(H')).\\&\quad \mathrm{typeOf}(q, H') <: \mathsf{Cell}.\end{aligned} \tag{A.165}$$

But this follows immediately from (A.161) and by the above we have

$$\mathrm{isolation}(H', P'). \tag{A.166}$$

Furthermore we never relied on the fact that $H \vdash P\,\textbf{gsep}$ to prove this and thus get

$$H' \vdash P' \ \textbf{gsep} \tag{A.167}$$

for free using Corollary A.15.

Case E-TERM**:** Trivial.

□

A.3 Proof of Progress

Theorem (Progress)**.** *Let S be a state such that $\vdash S \ \textbf{ok}_p$. Then either*

1. *$\exists S'$ s.t. $S \Rrightarrow S'$,*
2. *$S = H, \emptyset$ for some heap H s.t. $\mathrm{noSpawn}(H)$ or*
3. *$S = \textbf{error}$.*

Proof. Assume for a contradiction that $\vdash S \ \textbf{ok}_p$ but that neither 1, 2 or 3 from the theorem holds. We must then have that the following three statements hold

1. $\nexists S'$ s.t. $S \Rrightarrow S'$.
2. $S = H, \emptyset$ but $\mathrm{noSpawn}(H)$ does not hold, or $S = H, P$ where $P \neq \emptyset$.
3. $S \neq \textbf{error}$.

This gives us two cases: Either $S = H, \emptyset$ and $\mathrm{noSpawn}(H)$ does not hold, or $S = H, P$ and $P \neq \emptyset$.

We begin with the case where $S = H, \emptyset$ and $\mathrm{noSpawn}(H)$ does not hold. Looking at the definition of noSpawn this means that there is at least one $o \in \mathrm{dom}(H)$ such that $H(o) = \langle \mathsf{Cell}, DEP, l \rangle$ and

$$\exists (l', cb)^{\iota} \in DEP.\ l' \sqsubseteq l.$$

This clearly satisfies the preconditions of execution rule E-SPAWN and thus there is another state $S' = H', P'$ such that $S \Rrightarrow S'$. But this contradicts statement 1 above.

We proceed with the case where $S = H, P$ and $P \neq \emptyset$. This means that

$$P = Q \cup_D \{FS|^{\iota}_{a}\} \qquad FS = \langle L, t\rangle^{s} \circ GS$$

By $\vdash S$ $\mathbf{ok}_p$ we must have

$$H \vdash P.$$

By Proposition A.18 this means that

$$H; a \vdash FS.$$

Inspecting rule T-FS1 we see that

$$\begin{array}{ll} H \vdash \Gamma; L & \Gamma; a \vdash t : \sigma' \\ \sigma' <: \sigma & H; a \vdash^{s:\sigma} GS. \end{array} \tag{A.168}$$

We proceed by cases on the form of t.

Note. Note that we only state which *base rule* is used for each step found below. This means that if, e.g., the rule used is a frame reduction rule (one of the rules defined in Figure 5.2) the application of E-FPROP and E-FSPROP is implied. We will use the symbol $\lightning$to indicate a contradiction.

Case $t = x$**:** If $GS = \varepsilon$ then we can make a step to $S' = H, Q$ by applying rule E-TERM. $\lightning$

Otherwise we have that $GS = \langle L', t'\rangle^{s'} \circ HS$. This means we can step to $S' = H, P'$ where

$$P' = Q \cup_D \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L'', t'\rangle^{s'} \circ HS$$
$$L'' = L'[s \mapsto L(x)],$$

by base rule E-RET. $L(x)$ is defined by $H \vdash \Gamma; L$. $\lightning$

Case $t =$ `let` x `=` e `in` t'**:** We proceed on cases of e.

Case $e =$ `null`**:** We can step to $S' = H, P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t'\rangle^{s}$$
$$L' = L[x \mapsto \texttt{null}]$$

by base rule E-NULL. $\lightning$

Case $e = l$: We can step to $S' = H, P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s}$$
$$L' = L[x \mapsto l]$$

by rule E-LVal. ↯

Case $e = y$: We can step to $S' = H, P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s}$$
$$L' = L[x \mapsto L(y)]$$

by rule E-Var. $L(y)$ is defined by $H \vdash \Gamma; L$. ↯

Case $e = y\,.\,f$: By (A.168), T-Let, T-Select and T-Var

$$\Gamma(y) = C \qquad \text{ftype}(f, C) = \tau.$$

By $H \vdash \Gamma; L$

$$\text{typeOf}(L(y), H) <: C$$

which means that either $\text{typeOf}(L(y), H) = C' <: C$ or $\text{typeOf}(L(y), H) = \mathsf{Null}$.

If $\text{typeOf}(L(y), H) = \mathsf{Null}$ by definition of typeOf

$$L(y) = \texttt{null}$$

Then we can step to $S' =$ **error** by E-NullSelect. ↯

If $\text{typeOf}(L(y), H) = C'$ then

$$L(y) = o_y \qquad H(o_y) = \langle C', FM \rangle$$

By $C' <: C$, $\text{ftype}(f, C) = \tau$ and well-formedness of classes,

$$f \in \text{fields}(C') \qquad \text{ftype}(f, C') = \tau.$$

Then $\vdash H$ implies

$$f \in \text{dom}(FM).$$

But then by rule E-Select we can step to $S' = H, P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s}$$
$$L' = L[x \mapsto FM(f)].$$

↯

Case $e = y\,.\,f = z$: By (A.168), T-LET, T-ASSIGN and T-VAR

$$\Gamma(y) = C \qquad \text{ftype}(f, C) = \tau$$
$$\Gamma(z) = \tau' \qquad \tau' <: \tau.$$

By $H \vdash \Gamma; L$

$$\text{typeOf}(L(y), H) <: C,$$

which means that either $\text{typeOf}(L(y), H) = C' <: C$ or $\text{typeOf}(L(y), H) = \mathsf{Null}$.

If $\text{typeOf}(L(y), H) = \mathsf{Null}$ we get

$$L(y) = \texttt{null}.$$

We can then step to **error** by rule E-NULLASSIGN. ↯

If $\text{typeOf}(L(y), H) = C'$ then

$$L(y) = o_y \qquad H(o_y) = \langle C', FM \rangle.$$

By $C' <: C$, $\text{ftype}(f, C) = \tau$ and well-formedness of classes

$$f \in \text{fields}(C') \qquad \text{ftype}(f, C') = \tau.$$

Then by $\vdash H$ we have

$$f \in \text{dom}(FM).$$

We can then by E-ASSIGN step to $S' = H', P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s}$$
$$L' = L[x \mapsto L(z)] \qquad FM' = FM[f \mapsto L(z)]$$
$$H' = H[o_y \mapsto \langle C', FM' \rangle]$$

$L(z)$ is defined by $H \vdash \Gamma; L$. ↯

Case $e = \texttt{new}\, C$: By (A.168), T-LET and T-NEW

$$\Gamma; a \vdash \texttt{new}\, C : C.$$

Since the program is well-formed the class C exists.

$$FM = [f \mapsto \text{default}(\tau) : (\texttt{var}\ f\ :\ \tau) \in \text{fdecls}(C)]$$

is therefore well defined. By E-NEW we can therefore step to $S' = H', P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s}$$
$$L' = L[x \mapsto o] \qquad o \text{ fresh object reference}$$
$$H' = H[o \mapsto \langle C, FM \rangle].$$

$\lightning$

Case $e =$ `new Cell`**:** We can immediately step to $S' = H', P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s}$$
$$L' = L[x \mapsto o] \qquad o \text{ fresh object reference}$$
$$H' = H[o \mapsto \langle \mathsf{Cell}, \emptyset, \bot_{\mathscr{L}} \rangle]$$

by rule E-NEWCELL. $\lightning$

Case $e = y\texttt{.}m\texttt{(}z\texttt{)}$**:** By (A.168), T-LET, T-CALL and T-VAR

$$\Gamma(y) = C \qquad \text{mtype}(m, C) = \gamma \rightarrow \tau$$
$$\Gamma(z) = \gamma' \qquad \gamma' <: \gamma.$$

By $H \vdash \Gamma; L$

$$\text{typeOf}(L(y), H) <: C.$$

Then either $\text{typeOf}(L(y), H) = C' <: C$ or $\text{typeOf}(L(y)) =$ Null.

If $\text{typeOf}(L(y), H) =$ Null then

$$L(y) = \texttt{null}$$

and we can apply rule E-NULLCALL to step to **error**. $\lightning$

If $\text{typeOf}(L(y), H) = C'$

$$L(y) = o_y \qquad H(o_y) = \langle C', FM \rangle.$$

$C' <: C$ and classes being well-formed means that if $\text{mtype}(m, C)$ is defined then $\text{mbody}(m, C')$ is defined aswell. Thus let

$$\text{mbody}(m, C) = w \rightarrow t''.$$

By rule E-CALL we can then step to $S' = H, P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L'', t'' \rangle^{x} \circ \langle L, t' \rangle^{s}$$
$$L_{\text{base}} = \begin{cases} \emptyset & \text{if } a = ocap \\ L_0 & \text{if } a = \epsilon \end{cases}$$
$$L'' = L_{\text{base}}[\texttt{this} \mapsto L(y), w \mapsto L(z)].$$

$L(z)$ is defined by $H \vdash \Gamma; L$. $\lightning$

Case $e = y$ put z: By (A.168), T-LET, T-PUT and T-VAR

$$\Gamma(y) = \mathsf{Cell} \qquad \Gamma(z) = \mathcal{L}.$$

Thus by $H \vdash \Gamma; L$

$$\text{typeOf}(L(y), H) <: \mathsf{Cell} \qquad \text{typeOf}(L(z), H) <: \mathcal{L}.$$

By definition of typeOf and the structure of the type lattice this means that $\text{typeOf}(L(z), H) = \mathcal{L}$ and that either $\text{typeOf}(L(y), H) = \mathsf{Cell}$ or $\text{typeOf}(L(y), H) = \mathsf{Null}$.
$\text{typeOf}(L(z), H) = \mathcal{L}$ implies

$$L(z) = l' \qquad l' \in \mathscr{L}.$$

If $\text{typeOf}(L(y), H) = \mathsf{Null}$

$$L(y) = \texttt{null}$$

and we can apply E-NULLPUT to step to **error**. ↯

If $\text{typeOf}(L(y), H) = \mathsf{Cell}$ then

$$L(y) = o_y \qquad H(o_y) = \langle \mathsf{Cell}, \mathit{DEP}, l \rangle$$

Then we can use rule E-PUT to step to $S' = H', P'$ where

$$P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s}$$
$$L' = L[x \mapsto L(y)] \qquad c' = \langle \mathsf{Cell}, \mathit{DEP}, l \sqcup l' \rangle$$
$$H' = H[o_y \mapsto c']$$

↯

Case $e =$ when y pass z then $(\overline{cap}, w \Rightarrow t'')$: From (A.168), T-LET, T-WHEN and T-VAR

$$\Gamma(y) = \mathsf{Cell} \qquad \Gamma(z) = \mathcal{L}$$
$$\forall (u = u') \in \overline{cap}.\ \Gamma(u') = \mathsf{Cell}$$
$$\Gamma_{\text{cells}} = [(u : \mathsf{Cell}) \mid (u = u') \in \overline{cap}]$$
$$\Gamma_{\text{cells}}, w : \mathcal{L}; \mathit{ocap} \vdash t : \gamma.$$

Combinig the first two with $H \vdash \Gamma; L$

$$\text{typeOf}(L(y), H) <: \mathsf{Cell}$$
$$\text{typeOf}(L(z), H) <: \mathcal{L}.$$

Similarly to the previous case we have

$$L(z) = l' \qquad l' \in \mathscr{L}.$$

Also, either

$$\text{typeOf}(L(y), H) = \mathsf{Cell}$$

or

$$\text{typeOf}(L(z), H) = \mathsf{Null}.$$

If $\text{typeOf}(L(y), H) = \mathsf{Null}$ then

$$L(y) = \texttt{null}.$$

By E-NullWhen we can step to **error**. $\lightning$

If $\text{typeOf}(L(y), H) = \mathsf{Cell}$

$$L(y) = o_y \qquad H(o_y) = \langle \mathsf{Cell}, \mathit{DEP}, l \rangle$$

By E-When we can step to $S' = H', P'$ where

$$\begin{gathered} P' = Q \cup \{FS'|^{\iota}_{a}\} \qquad FS' = \langle L', t' \rangle^{s} \\ L' = L[x \mapsto L(y)] \qquad L_{\text{env}} = [u \mapsto L(u') \mid (u = u') \in \overline{cap}] \\ cb = (L_{\text{env}}, w \Rightarrow t'') \qquad \mathit{DEP}' = \mathit{DEP} \cup (l', cb)^{\iota} \\ \iota \text{ fresh thread id} \qquad H' = H[o_y \mapsto \langle \mathsf{Cell}, \mathit{DEP}', l \rangle]. \end{gathered}$$

$L(u')$ is defined for all u's by $H \vdash \Gamma; L$. $\lightning$

This concludes the case $t = \texttt{let}\ x = e\ \texttt{in}\ t'$.

Clearly, all cases for t leads to a contradiction and therefore we are done. □

Appendix B

Proof of Quasi-Determinism

B.1 Preliminaries

Definition B.1. Let $M : A \rightharpoonup B, g : B \hookrightarrow B$. Then $\delta(M, g) : A \rightharpoonup B$ and is defined as follows for any element $a \in A$.

$$\delta(M, g)(a) = g(M(a))$$

Definition B.2. Let g, h be as in definition B.3.

$$\eta(DEP, g, h) = \left\{(l, (\delta(L_{\text{env}}, g), z \Rightarrow t))^{h(\iota)} \mid (l, (L_{\text{env}}, z \Rightarrow t))^{\iota} \in DEP\right\}$$

I.e. we replace all object identifiers occuring in callback environments according to the replacement map g, and change the corresponding thread identifier ι to $h(\iota)$.

Definition B.3. Let $S = H, P$ and let

$$g \in \mathcal{O} \hookrightarrow \mathcal{O} \qquad h \in \mathcal{D} \hookrightarrow \mathcal{D}.$$

Then

$$\pi(H, g, h) = H^* \in \mathscr{H} \tag{B.1}$$

where

$$H^*(o) = \begin{cases} \langle C, FM^* \rangle & \text{if } H(g^{-1}(o)) = \langle C, FM \rangle \\ \langle \mathsf{Cell}, DEP^*, l \rangle & \text{if } H(g^{-1}(o)) = \langle \mathsf{Cell}, DEP, l \rangle \end{cases} \tag{B.2}$$
$$FM^* = \delta(FM, g) \qquad DEP^* = \eta(DEP, g, h).$$

Furthermore writing

$$P = \left\{FS_i|^{\iota_i}_{a_i}\right\}_{i=1}^{n} \tag{B.3}$$

we let

$$\rho(P, g, h) = P^*$$

where

$$FS_i = \langle L_{m_i}, t_{m_i} \rangle^{s_{m_i}} \circ \cdots \circ \langle L_1, t_1 \rangle^{s_1} \circ \varepsilon$$
$$FS_i^* = \langle \delta(L_{m_i}, g), t_{m_i} \rangle^{s_{m_i}} \circ \cdots \circ \langle \delta(L_1, g), t_1 \rangle^{s_1} \circ \varepsilon$$
$$P^* = \left\{ FS_i^* |_{a_i}^{h(\iota_i)} \right\}_{i=1}^{n}.$$

Remark. As noted earlier in Chapter 6, the π and ρ functions more or less just replace object and thread identifiers as specified by g and h.

We now restate and prove propositions 6.2 and 6.3.

Proposition. *$\simeq$ is an equivalence relation.*

Proof. (Sketch) To prove that $\simeq$ is an equivalence relation we need to prove reflexivity, symmetry and transitivity.

Reflexivity: Let S be a state. If $S =$ **error** then reflexivity follows trivially. Thus let $S = H, P$. We need to prove that there are $g \in \mathcal{O} \hookrightarrow \mathcal{O}, h \in \mathcal{D} \hookrightarrow \mathcal{D}$ such that $H = \pi(H, g, h)$ and $P = \rho(P, g, h)$. This is easy since the identity functions

$$g = \mathrm{Id}_{\mathcal{O}} \qquad h = \mathrm{Id}_{\mathcal{D}}$$

can be seen to fulfill this.

Symmetry: Let $S \simeq S'$. If $S =$ **error** then $S' =$ **error** and we have $S' \simeq S$. If $S = H, P$ and $S' = H', P'$, we must have

$$g \in \mathcal{O} \hookrightarrow \mathcal{O} \qquad h \in \mathcal{D} \hookrightarrow \mathcal{D}$$

such that

$$H' = \pi(H, g, h) \qquad P' = \rho(P, g, h).$$

It can be verified that g^{-1} and h^{-1} are bijections such that

$$H = \pi(H', g^{-1}, h^{-1}) \qquad P = \rho(P', g^{-1}, h^{-1}).$$

We thus get that $S' \simeq S$.

Transitivity: Let $S \simeq S'$ and $S' \simeq S''$. The case where $S =$ **error** is trivial. Thus let $S = H, P; S' = H', P'; S'' = H'', P''$. We must have

$$g \in \mathcal{O} \hookrightarrow \mathcal{O} \qquad h \in \mathcal{D} \hookrightarrow \mathcal{D}$$
$$g' \in \mathcal{O} \hookrightarrow \mathcal{O} \qquad h' \in \mathcal{D} \hookrightarrow \mathcal{D}$$

such that

$$\begin{aligned} H' &= \pi(H, g, h) \qquad & P' &= \rho(P, g, h) \\ H'' &= \pi(H', g', h') \qquad & P'' &= \rho(P', g', h'). \end{aligned}$$

It is easily shown that $g \circ g'$ and $h \circ h'$ are bijections such that

$$H'' = \pi(H', g \circ g', h \circ h') \qquad P'' = \rho(P', g \circ g', h \circ h').$$

Thus $S \simeq S''$.

□

Proposition. *For any two states S, S' such that $S \simeq S'$*

$$\vdash S \ \mathbf{ok}_p \iff \vdash S' \ \mathbf{ok}_p$$

Proof. (Sketch) Since $\simeq$ is an equivalence relation we only need to prove one direction. Thus assume

$$\vdash S \ \mathbf{ok}_p. \tag{B.4}$$

Letting $S = H, P$ and $S' = H', P'$, we need to show

$$\begin{gathered} \vdash H' \qquad H' \vdash P' \qquad H' \vdash P' \ \mathbf{ocr} \\ \text{isolation}(H', P') \qquad H' \vdash P' \ \mathbf{gsep} \qquad \text{uniqMain}(P'). \end{gathered} \tag{B.5}$$

Since the application of π to H amounts only to a renaming of the object and thread identifiers we have that $\mathscr{G}(H)$ is the same as $\mathscr{G}(H')$ up to a renaming of vertices. It does not change any types i.e.

$$\text{typeOf}(o, H) = \text{typeOf}(g(o), H'). \tag{B.6}$$

Thus it is not hard to prove $\vdash H'$.

The application of ρ similarly does the same kind of operation to P. Thus it is not hard proving $H' \vdash P'$ either. $H' \vdash P'$ **ocr** follows from the graph equivalence mentioned above and (B.6). Using the same properties we can show $\text{isolation}(H', P')$ and $H' \vdash P'$ **gsep**.

Finally, since ρ does not modify OCAP status of a thread, it is clear $\text{uniqMain}(P')$ holds. □

$$\begin{tikzcd}
S_2 \arrow[r, "R_1^{\iota_1,\beta_1}"] & S_{12} = S_{21} \\
S \arrow[u, "R_2^{\iota_2,\beta_2}"] \arrow[r, "R_1^{\iota_1,\beta_1}"'] & S_1 \arrow[u, "R_2^{\iota_2,\beta_2}"']
\end{tikzcd}$$

Figure B.1: Both sequences $R_1^{\iota_1,\beta_1}, R_2^{\iota_2,\beta_2}$ and $R_2^{\iota_2,\beta_2}, R_1^{\iota_1,\beta_1}$ lead to the same end state.

B.2 Proof of Quasi-Determinism

Lemma B.4. *Let S be a state s.t. $\vdash S$ $\mathbf{ok}_p$ where the two transitions*

$$R_1^{\iota_1,\beta_1} \neq R_2^{\iota_2,\beta_2}$$
$$\iota_1 \neq \iota_2 \qquad \beta_1 \neq \beta_2 \text{ or } \beta_1 = \beta_2 = \smiley$$

are applicable and will not lead to **error**. *Let*

$$\bar{R}_1 = R_1^{\iota_1,\beta_1}, R_2^{\iota_2,\beta_2} \qquad \bar{R}_2 = R_2^{\iota_2,\beta_2}, R_1^{\iota_1,\beta_1}$$

Then, both $\bar{R}_1, \bar{R}_2$ are applicable to S and

$$S \Rrightarrow^{\bar{R}_1} S_1' \qquad S \Rrightarrow^{\bar{R}_2} S_2'$$
$$S_1' = S_2'$$

Proof. The proof can be done by cases on R_1 and R_2. However, there are a big number of cases, most of which are easy to verify. Thus we are only going to do a few here to convince the reader. The others can be done similarly. One thing to note is that the proofs are symmetric in R_1 and R_2, meaning that if we prove case $R_1 = R, R_2 = R'$ we have also proven case $R_1 = R', R_2 = R$. An illustration of what we are proving for each case can be found in Figure B.1.

Case $R_1 =$ E-ASSIGN, $R_2 =$ E-ASSIGN**:** We note that this means that $\beta_1 = \beta_2 = \smiley$. Furthermore by the rules being applicable to S

we have

$$\begin{gathered}
S = H, P \qquad P = P_0 \cup_D \left\{ FS_1|^{\iota_1}_{a_1}, FS_2|^{\iota_2}_{a_2} \right\} \\
FS_1 = \langle L_1, \texttt{let}\ w_1 = x_1\,.\,f_1 = y_1\ \texttt{in}\ t'_1 \rangle^{s_1} \circ GS_1 \\
FS_1 = \langle L_2, \texttt{let}\ w_2 = x_2\,.\,f_2 = y_2\ \texttt{in}\ t'_2 \rangle^{s_2} \circ GS_2 \\
L_1(x_1) = o_1 \qquad L_2(x_2) = o_2 \\
H(o_1) = \langle C_1, FM_1 \rangle \qquad H(o_2) = \langle C_2, FM_2 \rangle \\
f_1 \in \mathrm{dom}(FM_1) \qquad f_2 \in \mathrm{dom}(FM_2) \\
S_1 = H_1, P_1 \qquad S \Rrightarrow^{R_1^{\iota_1}, \smile} S_1 \\
P_1 = P_0 \cup_D \left\{ FS'_1|^{\iota_1}_{a_1}, FS_2|^{\iota_2}_{a_2} \right\} \\
FS'_1 = \langle L'_1, t'_1 \rangle^{s_1} \circ GS_1 \qquad L'_1 = L_1[w_1 \mapsto L_1(y_1)] \\
FM'_1 = FM_1[f_1 \mapsto L_1(y_1)] \qquad H_1 = H[o_1 \mapsto \langle C_1, FM'_1 \rangle] \\
S_2 = H_2, P_2 \qquad S \Rrightarrow^{R_2^{\iota_2}, \smile} S_2 \\
P_2 = P_0 \cup_D \left\{ FS_1|^{\iota_1}_{a_1}, FS'_2|^{\iota_2}_{a_2} \right\} \\
FS'_2 = \langle L'_2, t'_2 \rangle^{s_2} \circ GS_2 \qquad L'_2 = L_2[w_2 \mapsto L_2(y_2)] \\
FM'_2 = FM_2[f_2 \mapsto L_2(y_2)] \qquad H_2 = H[o_2 \mapsto \langle C_2, FM'_2 \rangle]
\end{gathered} \tag{B.7}$$

By $\vdash S\ \mathbf{ok}_p$ we have

$$\mathrm{uniqMain}(P) \qquad \mathrm{isolation}(H, P). \tag{B.8}$$

These two imply that

$$o_1 \neq o_2. \tag{B.9}$$

Using this and inspecting $R_2^{\iota_2, \smile}$ it is not hard to see that

$$\begin{gathered}
S_{12} = H_{12}, P_{12} \qquad S_1 \Rrightarrow^{R_2^{\iota_2}, \smile} S_{12} \\
P_{12} = P_0 \cup_D \left\{ FS'_1|^{\iota_1}_{a_1}, FS'_2|^{\iota_2}_{a_2} \right\} \\
H_{12} = H[o_1 \mapsto \langle C_1, FM'_1 \rangle, o_2 \mapsto \langle C_2, FM'_2 \rangle]
\end{gathered} \tag{B.10}$$

and similarly

$$\begin{gathered}
S_{21} = H_{21}, P_{21} \qquad S_2 \Rrightarrow^{R_1^{\iota_1}, \smile} S_{21} \\
P_{21} = P_0 \cup_D \left\{ FS'_1|^{\iota_1}_{a_1}, FS'_2|^{\iota_2}_{a_2} \right\} \\
H_{21} = H[o_1 \mapsto \langle C_1, FM'_1 \rangle, o_2 \mapsto \langle C_2, FM'_2 \rangle]
\end{gathered} \tag{B.11}$$

Clearly then $S_{12} = S_{21}$ and we are done.

Remark. The main point made here is that the two transition identifiers operate on different parts of the heap because of isolation of threads. Thus all cases where we have this property, e.g., $R_1 = \text{E-S}\textsc{elect}, R_2 = \text{E-A}\textsc{ssign}$ follow similarly.

Case $R_1 = \text{E-V}\textsc{ar}, R_2 = \text{E-A}\textsc{ssign}$**:** We first note that

$$\beta_1 = \beta_2 = ☺.$$

Since the transition identifiers are applicable to S we have

$$\begin{gathered}
S = H, P \qquad P = P_0 \cup_D \left\{ FS_1|_{a_1}^{\iota_1}, FS_2|_{a_2}^{\iota_2} \right\} \\
FS_1 = \langle L_1, \texttt{let}\ x_1 \texttt{ = } y_1\ \texttt{in}\ t_1' \rangle^{s_1} \circ GS_1 \\
FS_1 = \langle L_2, \texttt{let}\ x_2 \texttt{ = } y_2 \texttt{.} f_2 \texttt{ = } z_2\ \texttt{in}\ t_2' \rangle^{s_2} \circ GS_2 \\
L_2(y_2) = o_2 \qquad H(o_2) = \langle C_2, FM_2 \rangle \\
f_2 \in \text{dom}(FM_2)
\end{gathered}$$

$$\begin{gathered}
S_1 = H_1, P_1 \qquad S \Rrightarrow^{R_1^{\iota_1}, ☺} S_1 \\
P_1 = P_0 \cup_D \left\{ FS_1'|_{a_1}^{\iota_1}, FS_2|_{a_2}^{\iota_2} \right\} \\
FS_1' = \langle L_1', t_1' \rangle^{s_1} \circ GS_1 \qquad L_1' = L_1[x_1 \mapsto L_1(y_1)] \\
H_1 = H
\end{gathered} \tag{B.12}$$

$$\begin{gathered}
S_2 = H_2, P_2 \qquad S \Rrightarrow^{R_2^{\iota_2}, ☺} S_2 \\
P_2 = P_0 \cup_D \left\{ FS_1|_{a_1}^{\iota_1}, FS_2'|_{a_2}^{\iota_2} \right\} \\
FS_2' = \langle L_2', t_2' \rangle^{s_2} \circ GS_2 \qquad L_2' = L_2[x_2 \mapsto L_2(z_2)] \\
FM_2' = FM_2[f_2 \mapsto L_2(y_2)] \qquad H_2 = H[o_2 \mapsto \langle C_2, FM_2' \rangle]
\end{gathered}$$

Furthermore it is not hard to see that

$$\begin{gathered}
S_{12} = H_{12}, P_{12} \qquad S_1 \Rrightarrow^{R_2^{\iota_2}, ☺} S_{12} \\
P_{12} = P_0 \cup_D \left\{ FS_1'|_{a_1}^{\iota_1}, FS_2'|_{a_2}^{\iota_2} \right\} \\
H_{12} = H[o_2 \mapsto \langle C_2, FM_2' \rangle]
\end{gathered} \tag{B.13}$$

and

$$\begin{gathered}
S_{21} = H_{21}, P_{21} \qquad S_2 \Rrightarrow^{R_1^{\iota_1}, ☺} S_{21} \\
P_{21} = P_0 \cup_D \left\{ FS_1'|_{a_1}^{\iota_1}, FS_2'|_{a_2}^{\iota_2} \right\} \\
H_{21} = H[o_2 \mapsto \langle C_2, FM_2' \rangle]
\end{gathered} \tag{B.14}$$

Clearly $S_{12} = S_{21}$ and we are done.

Remark. This case is very simple since one of the transition identifiers operate solely on the local state of a thread. All cases with this property follow similarly.

Case $R_1 = \text{E-P}\textsc{ut}, R_2 = \text{E-P}\textsc{ut}$**:** Clearly

$$\beta_1 = \beta_2 = ☺.$$

Similarly to previous cases we have

$$\begin{gathered} S = H, P \qquad P = P_0 \cup_D \left\{ FS_1|_{a_1}^{\iota_1}, FS_2|_{a_2}^{\iota_2} \right\} \\ FS_1 = \langle L_1, \texttt{let}\ x_1 = y_1\ \texttt{put}\ z_1\ \texttt{in}\ t'_1 \rangle^{s_1} \circ GS_1 \\ FS_1 = \langle L_2, \texttt{let}\ x_2 = y_2\ \texttt{put}\ z_2\ \texttt{in}\ t'_2 \rangle^{s_2} \circ GS_2 \\ L_1(y_1) = o_1 \qquad L_2(y_2) = o_2 \\ L_1(z_1) = l_1 \qquad L_2(z_2) = l_2 \end{gathered} \tag{B.15}$$

Now we have two cases since isolation does not prevent sharing of references to Cell objects. If $o_1 \neq o_2$ we are done similarly to case $R_1 = \text{E-A}\textsc{ssign}, R_2 = \text{E-A}\textsc{ssign}$. If $o_1 = o_2 = o$ we proceed as follows.

First of all,

$$H(o) = \langle \mathsf{Cell}, DEP, l \rangle. \tag{B.16}$$

Then

$$\begin{gathered} S_1 = H_1, P_1 \qquad S \Rrightarrow^{R_1^{\iota_1}, ☺} S_1 \\ P_1 = P_0 \cup_D \left\{ FS'_1|_{a_1}^{\iota_1}, FS_2|_{a_2}^{\iota_2} \right\} \\ FS'_1 = \langle L'_1, t'_1 \rangle^{s_1} \circ GS_1 \qquad L'_1 = L_1[x_1 \mapsto L_1(y_1)] \\ H_1 = H[o \mapsto \langle \mathsf{Cell}, DEP, l \sqcup l_1 \rangle] \end{gathered} \tag{B.17}$$

and similarly

$$\begin{gathered} S_2 = H_2, P_2 \qquad S \Rrightarrow^{R_2^{\iota_2}, ☺} S_2 \\ P_1 = P_0 \cup_D \left\{ FS_1|_{a_1}^{\iota_1}, FS'_2|_{a_2}^{\iota_2} \right\} \\ FS'_2 = \langle L'_2, t'_2 \rangle^{s_2} \circ GS_2 \qquad L'_2 = L_2[x_2 \mapsto L_2(y_2)] \\ H_2 = H[o \mapsto \langle \mathsf{Cell}, DEP, l \sqcup l_2 \rangle] \end{gathered} \tag{B.18}$$

We also have

$$\begin{gathered} S_{12} = H_{12}, P_{12} \qquad S_1 \Rrightarrow^{R_2^{\iota_2}, ☺} S_{12} \\ P_{12} = P_0 \cup_D \left\{ FS'_1|_{a_1}^{\iota_1}, FS'_2|_{a_2}^{\iota_2} \right\} \\ H_{12} = H[o \mapsto \langle \mathsf{Cell}, DEP, (l \sqcup l_1) \sqcup l_2 \rangle] \end{gathered} \tag{B.19}$$

and

$$\begin{gathered} S_{21} = H_{21}, P_{21} \qquad S_2 \Rrightarrow^{R_1^{\iota_1}, ☺} S_{21} \\ P_{21} = P_0 \cup_D \left\{ FS'_1|^{\iota_1}_{a_1}, FS'_2|^{\iota_2}_{a_2} \right\} \\ H_{21} = H[o \mapsto \langle \mathsf{Cell}, DEP, (l \sqcup l_2) \sqcup l_1 \rangle] \end{gathered} \tag{B.20}$$

By commutativity and associativity of the least upper bound operation $\sqcup$ we have

$$(l \sqcup l_1) \sqcup l_2 = (l \sqcup l_2) \sqcup l_1 \tag{B.21}$$

and thus we have $S_{12} = S_{21}$.

Case $R_1 = \text{E-WHEN}, R_2 = \text{E-WHEN}$**:** We have that

$$\beta_1 = \iota'_1 \qquad \beta_2 = \iota'_2 \qquad \iota'_1 \neq \iota'_2. \tag{B.22}$$

Similarly to previous cases we have

$$\begin{gathered} S = H, P \qquad P = P_0 \cup_D \left\{ FS_1|^{\iota_1}_{a_1}, FS_2|^{\iota_2}_{a_2} \right\} \\ FS_1 = \langle L_1, \texttt{let } x_1 \texttt{ = when } y_1 \texttt{ pass } z_1 \texttt{ then } \ldots \texttt{ in } t'_1 \rangle^{s_1} \circ GS_1 \\ FS_1 = \langle L_2, \texttt{let } x_2 \texttt{ = when } y_2 \texttt{ pass } z_2 \texttt{ then } \ldots \texttt{ in } t'_2 \rangle^{s_2} \circ GS_2 \\ L_1(y_1) = o_1 \qquad L_2(y_2) = o_2 \\ L_1(z_1) = l_1 \qquad L_2(z_2) = l_2 \end{gathered} \tag{B.23}$$

Where we use ... for things that are not really vital.

If $o_1 \neq o_2$ we are done similar to previous case. If $o_1 = o_2 = o$ then we proceed as follows. We use the notation of $H_\#, P_\#$ similar to earlier cases. It is simple to see that

$$\begin{gathered} H_1 = H[o \mapsto \langle \mathsf{Cell}, DEP_1, l \rangle] \qquad DEP_1 = DEP \cup \{(l_1, \ldots)^{\iota'_1}\} \\ H_2 = H[o \mapsto \langle \mathsf{Cell}, DEP_2, l \rangle] \qquad DEP_2 = DEP \cup \{(l_2, \ldots)^{\iota'_2}\} \\ H_{12} = H[o \mapsto \langle \mathsf{Cell}, DEP_{12}, l \rangle] \\ DEP_{12} = DEP \cup \{(l_1, \ldots)^{\iota'_1}, (l_2, \ldots)^{\iota'_2}\} \\ H_{21} = H[o \mapsto \langle \mathsf{Cell}, DEP_{21}, l \rangle] \\ DEP_{21} = DEP \cup \{(l_1, \ldots)^{\iota'_1}, (l_2, \ldots)^{\iota'_2}\} \end{gathered} \tag{B.24}$$

Clearly $DEP_{12} = DEP_{21}$, which implies $H_{12} = H_{21}$. Similarly to earlier cases, $P_{12} = P_{21}$ and thus $S_{12} = S_{21}$.

Case $R_1 = \text{E-W}\textsc{hen}, R_2 = \text{E-S}\textsc{pawn}$**:** First

$$\beta_1 = \iota_1' \qquad \beta_2 = ☺. \tag{B.25}$$

We have

$$\begin{gathered}
S = H, P \qquad P = P_0 \cup_D \left\{ FS_1|_{a_1}^{\iota_1} \right\} \\
FS_1 = \langle L_1, \texttt{let}\ x_1\ \texttt{= when}\ y_1\ \texttt{pass}\ z_1\ \texttt{then} \ldots\ \texttt{in}\ t_1' \rangle^{s_1} \circ GS_1 \\
L_1(y_1) = o_1 \qquad L_1(z_1) = l_1 \\
o_2 \in \mathrm{dom}(H) \qquad H(o_2) = \langle \mathsf{Cell}, \mathit{DEP}, l \rangle \\
(l_2, (L_{\text{env}}, z_2 \Rightarrow t''))^{\iota_2} \in \mathit{DEP}
\end{gathered} \tag{B.26}$$

Similarly to previous cases we have the case where $o_1 \neq o_2$ which follows similarly to case $R_1 = \text{E-A}\textsc{ssign}, R_2 = \text{E-A}\textsc{ssign}$ and the case $o_1 = o_2 = o$ for which we proceed as follows.

First we note that

$$H(o) = \langle \mathsf{Cell}, \mathit{DEP}, l \rangle. \tag{B.27}$$

We have that

$$\begin{gathered}
S_1 = H_1, P_1 \qquad S \Rrightarrow^{R_1^{\iota_1, \beta_1}} S_1 \\
P_1 = P_0 \cup_D \left\{ FS_1'|_{a_1}^{\iota_1} \right\} \\
FS_1' = \langle L_1', t_1' \rangle^{s_1} \circ GS_1 \qquad L_1' = L_1[x_1 \mapsto L_1(y_1)] \\
H_1 = H[o \mapsto \langle \mathsf{Cell}, \mathit{DEP}_1, l \rangle] \qquad \mathit{DEP}_1 = \mathit{DEP} \cup \{(l_1, \ldots)^{\iota_1'}\}
\end{gathered} \tag{B.28}$$

Stepping according to $R_2^{\iota_2, ☺}$ from S gives

$$\begin{gathered}
S_2 = H_2, P_2 \qquad S \Rrightarrow^{R_2^{\iota_2, ☺}} S_2 \\
P_2 = P_0 \cup_D \left\{ FS_1|_{a_1}^{\iota_1}, FS_2|_{ocap}^{\iota_2} \right\} \\
FS_2 = \langle L_{\text{env}}[z \mapsto l_2], t'' \rangle^{-} \circ \varepsilon \\
H_2 = \langle \mathsf{Cell}, \mathit{DEP}_2, l \rangle \qquad \mathit{DEP}_2 = \mathit{DEP} \setminus \{(l_2, \ldots)^{\iota_2}\}
\end{gathered} \tag{B.29}$$

Given (B.28) and (B.29) it is not hard to verify that

$$\begin{gathered}
S_{12} = H_{12}, P_{12} \qquad S_1 \Rrightarrow^{R_2^{\iota_2, ☺}} S_{12} \\
P_{12} = P_0 \cup_D \left\{ FS_1'|_{a_1}^{\iota_1}, FS_2|_{ocap}^{\iota_2} \right\} \\
H_{12} = H[o \mapsto \langle \mathsf{Cell}, \mathit{DEP}_{12}, l \rangle] \\
\mathit{DEP}_{12} = \mathit{DEP} \cup \{(l_1, \ldots)^{\iota_1'}\} \setminus \{(l_2, \ldots)^{\iota_2}\}
\end{gathered} \tag{B.30}$$

$$\begin{gathered}S_{21} = H_{21}, P_{21} \qquad S_2 \Rrightarrow^{R_1^{\iota_1,\iota_1'}} S_{21}\\ P_{21} = P_0 \cup_D \left\{ FS_1'|_{a_1}^{\iota_1}, FS_2|_{ocap}^{\iota_2} \right\}\\ H_{21} = H[o \mapsto \langle \mathsf{Cell}, DEP_{21}, l \rangle]\\ DEP_{21} = DEP \setminus \{(l_2, \dots)^{\iota_2}\} \cup \{(l_1, \dots)^{\iota_1'}\}\end{gathered} \tag{B.31}$$

Since ι_1' is fresh it is clear that $DEP_{12} = DEP_{21}$ and therefore we have $S_{12} = S_{21}$.

Remark. The main point here is that ι_1' being a fresh thread identifier makes the removal and addition to the dependency set DEP commute.

It should not be hard to verify the other cases in a similar manner. □

Definition B.5. Let $R_1^{\iota_1,\beta_1}, R_2^{\iota_2,\beta_2}$ be two transition identifiers. We call β_1 and β_2 compatible if

$$\beta_1 \neq \beta_2 \text{ or } \beta_1 = \beta_2 = ☺.$$

Lemma B.6. *Let S be a well-typed state. Let*

$$\bar{R} = R_1^{\iota_1,\beta_1}, \dots, R_n^{\iota_n,\beta_n} \qquad S \Rrightarrow^{\bar{R}} S_{\bar{R}}$$

Let $R^{\iota,\beta}$ be a transition such that

$$S \Rrightarrow^{R^{\iota,\beta}} S_{R^{\iota,\beta}} \qquad S_{\bar{R}} \Rrightarrow^{R^{\iota,\beta}} S',$$

for some states $S_{R^{\iota,\beta}}$ and S', and

$$\forall i \in \{1, \dots, n\}\,. \quad \iota \neq \iota_i. \tag{B.32}$$

Furthermore let β be compatible with β_i for all $i = 1, \dots, n$. Then

$$S \Rrightarrow^{R^{\iota,\beta}} S_{R^{\iota,\beta}} \Rrightarrow^{\bar{R}} S'.$$

Proof. By induction on n, the length of $\bar{R}$.

Case $n = 1$**:** Follows immediately from Lemma B.4.

Case $n = i + 1, i \geq 1$**:** This case can be visualized with Figure B.2. Let

$$\bar{R}_{\geq 2} = R_2^{\iota_2,\beta_2}, \dots, R_n^{\iota_n,\beta_n}.$$

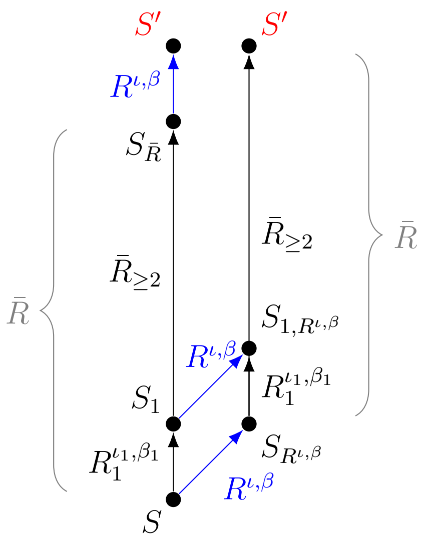

Figure B.2: $R^{\iota,\beta}$ is applicable at S_1. By induction hypothesis $S_1 \Rrightarrow^{R^{\iota,\beta}} S_{1,R^{\iota,\beta}} \Rrightarrow^{\bar{R}_{\geq 2}} S'$. By Lemma B.4 $S \Rrightarrow^{R^{\iota,\beta},R_1^{\iota_1,\beta_1}} S_{1,R^{\iota,\beta}}$. Thus $S \Rrightarrow^{R^{\iota,\beta},\bar{R}} S'$.

Since $\bar{R}$ is applicable to S we have

$$S \Rrightarrow^{R_1^{\iota_1,\beta_1}} S_1, \tag{B.33}$$

for some state S_1. Clearly

$$S_1 \Rrightarrow^{\bar{R}_{\geq 2}} S_{\bar{R}}. \tag{B.34}$$

By (B.32), thread ι never executes in the sequence specified by $\bar{R}$. Combining this with $R^{\iota,\beta}$ being applicable at S and the compatibility of β, it is not hard to see that $R^{\iota,\beta}$ is applicable at S_1, i.e.,

$$S_1 \Rrightarrow^{R^{\iota,\beta}} S_{1,R^{\iota,\beta}} \tag{B.35}$$

for some state $S_{1,R^{\iota,\beta}}$. Then, by induction hypothesis

$$S_1 \Rrightarrow^{R^{\iota,\beta}} S_{1,R^{\iota,\beta}} \Rrightarrow^{\bar{R}_{\geq 2}} S'. \tag{B.36}$$

Furthermore, $R_1^{\iota_1,\beta_1}$ and $R^{\iota,\beta}$ fulfills preconditions of Lemma B.4. Using this lemma we get

$$S \Rrightarrow^{R^{\iota,\beta}} S_{R^{\iota,\beta}} \Rrightarrow^{R_1^{\iota_1,\beta_1}} S_{1,R^{\iota,\beta}}. \tag{B.37}$$

But then combining (B.36) and (B.37) we have

$$S \Rrightarrow^{R^{\iota,\beta}} S_{R^{\iota,\beta}} \Rrightarrow^{\bar{R}} S'. \tag{B.38}$$

□

Proposition B.7. *Let $S_1 = H_1, P_1 \simeq H_2, P_2 = S_2$ be well-typed states and let g, h be the bijections such that*

$$H_2 = \pi(H_1, g, h) \qquad P_2 = \rho(P_1, g, h). \tag{B.39}$$

Let $o_1, o_2 \in \mathcal{O}$ be fresh at S_1, S_2 respectively. Let $\iota_1, \iota_2 \in \mathcal{D}$ be fresh at S_1, S_2 respectively. Then there are bijections

$$g' \in \mathcal{O} \hookrightarrow \mathcal{O} \qquad h' \in \mathcal{D} \hookrightarrow \mathcal{D}$$

such that

$$g'(o_1) = o_2 \qquad h'(\iota_1) = \iota_2 \tag{B.40}$$

and

$$H_2 = \pi(H_1, g', h') \qquad P_2 = \rho(P_1, g', h'). \tag{B.41}$$

Proof. The role of g and h is merely to define a renaming of identifiers in $\mathcal{O}(S_1)$ and $\mathcal{D}(S_1)$ to identifiers in $\mathcal{O}(S_2)$ and $\mathcal{D}(S_2)$. Thus it should not be hard to see that

$$\begin{aligned} \forall o \in \mathcal{O}(S_1).\quad & g(o) \in \mathcal{O}(S_2) \\ \forall \iota \in \mathcal{D}(S_1).\quad & h(\iota) \in \mathcal{D}(S_2) \end{aligned}$$

Since g and h are bijections,

$$\begin{aligned} g(o_1) \notin \mathcal{O}(S_2) \qquad & h(\iota_1) \notin \mathcal{D}(S_2) \\ g^{-1}(o_2) \notin \mathcal{O}(S_1) \qquad & h^{-1}(\iota_2) \notin \mathcal{D}(S_1). \end{aligned}$$

Because of the above, it should be clear that the following functions are well defined, are bijections and satisfy both (B.40) and (B.41).

$$g'(o) = \begin{cases} o_2 & \text{if } o = o_1 \\ g(o_1) & \text{if } g(o) = o_2 \\ g(o) & \text{otherwise.} \end{cases}$$

$$h'(\iota) = \begin{cases} \iota_2 & \text{if } \iota = \iota_1 \\ h(\iota_1) & \text{if } h(\iota) = \iota_2 \\ h(\iota) & \text{otherwise.} \end{cases}$$

□

Because of this proposition, the following is well defined.

Definition B.8. Let $S_1 = H_1, P_1 \simeq H_2, P_2 = S_2$ be well-typed states and let g, h be the bijections such that

$$H_2 = \pi(H_1, g, h) \qquad P_2 = \rho(P_1, g, h). \tag{B.42}$$

Let $o_1, o_2 \in \mathcal{O}$ be fresh at S_1, S_2 respectively. Let $\iota_1, \iota_2 \in \mathcal{D}$ be fresh at S_1, S_2 respectively. Then define

$$g[o_1 \mapsto o_2] \in \mathcal{O} \hookrightarrow \mathcal{O} \qquad h[\iota_1 \mapsto \iota_2] \in \mathcal{D} \hookrightarrow \mathcal{D} \tag{B.43}$$

as any bijections satisfying (B.40) and (B.41) with $g' = g[o_1 \mapsto o_2], h' = h[\iota_1 \mapsto \iota_2]$.

Lemma B.9. *Let S, S', T be well-typed states and $\bar{R}$ a transition sequence such that*

$$S \Rrightarrow^{\bar{R}} T \qquad S \simeq S'.$$

Then there is a state T' and a transition sequence $\bar{R}'$ of the same length as $\bar{R}$ such that

$$S' \Rrightarrow^{\bar{R}'} T' \qquad T \simeq T'.$$

Proof. By induction on the length n of $\bar{R}$.

Case $n = 0$**:** $\bar{R}$ is the empty sequence and thus $S = T$. Take $T' = S'$ and $\bar{R}' = \bar{R}$ and we are done.

Case $n = i + 1, i \geq 0$**:** Let

$$\bar{R} = R_1^{\iota_1,\beta_1}, \ldots, R_{i+1}^{\iota_{i+1},\beta_{i+1}}$$
$$\bar{R}_{\leq i} = R_1^{\iota_1,\beta_1}, \ldots, R_i^{\iota_i,\beta_i}$$

By assumption

$$S \Rrightarrow^{\bar{R}_{\leq i}} S_i \Rrightarrow^{R_{i+1}^{\iota_{i+1},\beta_{i+1}}} T$$

for some state S_i. By induction hypothesis there is $\bar{R}'_{\leq i}, S'_i$ such that

$$S' \Rrightarrow^{\bar{R}'_{\leq i}} S'_i \qquad S_i \simeq S'_i. \tag{B.44}$$

Since S_i can take another step

$$S_i = H, P \qquad S'_i = H', P'.$$

Now consider $R_{i+1}^{\iota_{i+1},\beta_{i+1}}$. Since $S_i \simeq S'_i$ there are

$$g \in \mathcal{O} \hookrightarrow \mathcal{O} \qquad h \in \mathcal{D} \hookrightarrow \mathcal{D}$$

such that

$$H' = \pi(H, g, h) \qquad P' = \eta(P, g, h).$$

Now let

$$R = R_{i+1} \qquad \iota = h(\iota_{i+1})$$

$$\beta = \begin{cases} ☺ & \text{if } \beta_{i+1} = ☺ \\ o'_{\text{new}} & \text{if } \beta_{i+1} = o_{\text{new}} \in \mathcal{O} \\ & \quad o_{\text{new}} \text{ is fresh at } S_i \\ & \quad o'_{\text{new}} \text{ is fresh at } S'_i \\ \iota'_{\text{new}} & \text{if } \beta_{i+1} = \iota_{\text{new}} \in \mathcal{D} \\ & \quad \iota_{\text{new}} \text{ is fresh at } S_i \\ & \quad \iota'_{\text{new}} \text{ is fresh at } S'_i \end{cases}$$

$$g_{\text{new}} = \begin{cases} g & \text{if } \beta_{i+1} = ☺ \text{ or } \beta_{i+1} = \iota_{\text{new}} \\ g[o_{\text{new}} \mapsto o'_{\text{new}}] & \text{if } \beta_{i+1} = o_{\text{new}} \end{cases}$$

$$h_{\text{new}} = \begin{cases} h & \text{if } \beta_{i+1} = ☺ \text{ or } \beta_{i+1} = o_{\text{new}} \\ h[\iota_{\text{new}} \mapsto \iota'_{\text{new}}] & \text{if } \beta_{i+1} = \iota_{\text{new}} \end{cases}$$

Since

$$S_i \Rrightarrow^{R_{i+1}^{\iota_{i+1},\beta_{i+1}}} T, \tag{B.45}$$

by rule inspection it is easy to verify that

$$S'_i \Rrightarrow^{R^{\iota,\beta}} T' \tag{B.46}$$

for some T'. If $T =$ **error** then by rule inspection we must have $T' =$ **error** and trivially that $T \simeq T'$. Furthermore, if $T = H_T, P_T$ then by rule inspection $T' = H_{T'}, P_{T'}$. By considering definition B.8 it is not hard to verify that we have

$$H_{T'} = \pi(H_T, g_{\text{new}}, h_{\text{new}}) \qquad P_{T'} = \eta(P_T, g_{\text{new}}, h_{\text{new}}).$$

Thus $T \simeq T'$.

□

Lemma B.10. *Let*

$$\bar{R} = R_1^{\iota_1,\beta_1}, \ldots, R_n^{\iota_n,\beta_n}$$

be a transition sequence and S, T *be well-typed states such that.*

$$S \Rrightarrow^{\bar{R}} T.$$

Then there is a state T' *and a series of mutually compatible*

$$\beta_i', i = 1, \ldots, n$$

such that

$$\beta_i' \notin \mathcal{O}(S), \beta_i' \notin \mathcal{D}(S) \qquad i = 1, \ldots, n,$$
$$\bar{R}' = R_1^{\iota_1,\beta_1'}, \ldots, R_n^{\iota_1,\beta_n'}$$
$$S \Rrightarrow^{\bar{R}'} T' \qquad T \simeq T'.$$

Proof. (Sketch) This is obvious from these two facts:

- There are infinitely many elements in both $\mathcal{O}$ and $\mathcal{D}$ and only finitely many in $\mathcal{O}(S)$ and $\mathcal{D}(S)$. This follows from the definition of $\mathcal{O}(S)$ and $\mathcal{D}(S)$.
- All choices of fresh identifiers results in equivalent states. I.e. for any state S, and transition identifiers $R^{\iota,\beta}, R^{\iota,\beta'}$ such that

  $$S \Rrightarrow^{R^{\iota,\beta}} S_1 \qquad S \Rrightarrow^{R^{\iota,\beta'}} S_1',$$

  we have

  $$S_1 \simeq S_1'.$$

  This is not hard to prove.

□

Finally we restate and prove Theorem 3.

Theorem. *Let* S, S', T, T' *be well-typed states not equal to* **error** *such that*

$$S \simeq S'$$
$$S \Rrightarrow^{\bar{R}} T \qquad S' \Rrightarrow^{\bar{R}'} T'.$$

Let n *and* n' *be the lengths of* $\bar{R}$ *and* $\bar{R}'$ *respectively. Furthermore we assume that neither* T *or* T' *can make a step. Then*

$$n = n' \quad \textit{and} \quad T \simeq T'.$$

Proof. If $S \neq S'$ by Lemma B.9 there is a state T'' and a transition sequence $\bar{R}''$ such that

$$S \Rrightarrow^{\bar{R}''} T'' \qquad T' \simeq T''. \tag{B.47}$$

Because of this and transistiveness of $\simeq$ WLOG we can assume that $S = S'$.

We proceed by induction on n and n'.

Case $n \leq 0, n' \leq 0$**:** Clearly, $n = n' = 0$ and we are done since $T = S = S' = T'$.

Case $n \leq i+1, n' \leq i+1$ **and** $i \geq 0$**:** First of all, the case where $n = 0$ ($n' = 0$) and $n' > 0$ ($n > 0$) is impossible since this immediately contradicts that T (T') cannot take another step. Thus we have that either $n = n' = 0$ (which falls under the case above) or $n \geq 1$ and $n' \geq 1$. Thus we assume the latter. In the continuation, this case can be visualized as in Figure B.3. We let

$$\bar{R} = R_1^{\iota_1,\beta_1}, \ldots, R_n^{\iota_n,\beta_n}$$
$$\bar{R}' = R_1'^{\iota_1',\beta_1'}, \ldots, R_{n'}'^{\iota_{n'}',\beta_{n'}'}.$$

Furthermore let

$$\bar{R}_{\geq 2} = R_2^{\iota_2,\beta_2}, \ldots, R_n^{\iota_n,\beta_n}$$

Consider the first transition identifier of $\bar{R}$: $R^{\iota,\beta} = R_1^{\iota_1,\beta_1}$. We let k be such that $R_k'^{\iota_k',\beta_k'}$ is the first transition identifier in $\bar{R}'$ for which

$$\iota_k' = \iota. \tag{B.48}$$

This k must exist since otherwise T' can take another step. This is due to that thread ι can execute at S and if it never executes before T', it will still be able to execute, thus generating another step. By the remark following definition 6.4 we have

$$R_k' = R. \tag{B.49}$$

We let

$$\bar{R}'_{<k} = R_1'^{\iota_1',\beta_1'}, \ldots, R_{k-1}'^{\iota_{k-1}',\beta_{k-1}'}$$
$$\bar{R}'_{>k} = R_{k+1}'^{\iota_{k+1}',\beta_{k+1}'}, \ldots, R_{n'}'^{\iota_{n'}',\beta_{n'}'}.$$

We can thus write

$$S \Rrightarrow^{\bar{R}'_{<k}} S_{k-1}' \Rrightarrow^{R^{\iota,\beta_k'}} S_k' \Rrightarrow^{\bar{R}'_{>k}} T'. \tag{B.50}$$

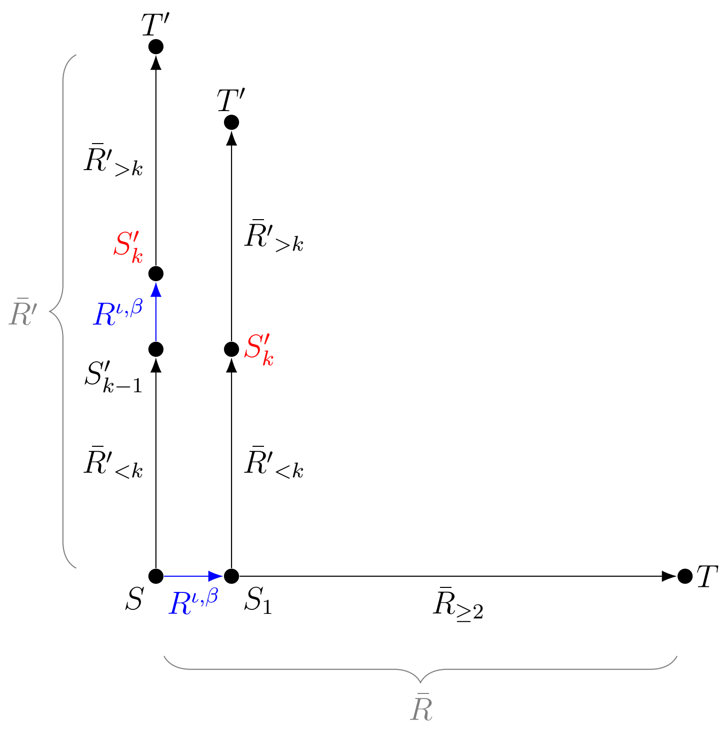

Figure B.3: $R^{\iota,\beta}$ is the first transition identifier of $\bar{R}$. It must also occur somewhere in the transition sequence $\bar{R}'$. We transform $\bar{R}'$ by moving $R^{\iota,\beta}$ to the beginning of the sequence. The halting state is the same because of Lemma B.6. The new sequence $R^{\iota,\beta}, \bar{R}'_{<k}, \bar{R}'_{>k}$ begins with the same step as $\bar{R}$ and therefore steps to S_1 in the first step. Thus we can examine the two shorter sequences $\bar{R}'_{<k}, \bar{R}'_{>k}$ and $\bar{R}_{\geq 2}$ and conclude that $T \simeq T'$ by the induction hypothesis.

WLOG we can assume that

$$\beta'_k = \beta \tag{B.51}$$

and that β is compatible with β'_i for all $i = 1, \ldots, k-1$. Otherwise we could just reassign $\beta_1, \ldots, \beta_n, \beta'_1, \ldots, \beta'_n$, in accordance with Lemma B.10.

By the choice of k,

$$\forall i \text{ s.t. } 1 \leq i < k. \quad \iota \neq \iota'_i.$$

Then by (B.50) and Lemma B.6 we have

$$S \Rrightarrow^{R^{\iota,\beta}} S_1 \Rrightarrow^{\bar{R}'_{<k}} S'_k,$$

which combined with (B.50) yields

$$S \Rrightarrow^{R^{\iota,\beta}} S_1 \Rrightarrow^{\bar{R}'_{<k}} S'_k \Rrightarrow^{\bar{R}'_{>k}} T'.$$

Furthermore, by $\bar{R}$ being applicable at S we have

$$S \Rrightarrow^{R^{\iota,\beta}} S_1 \Rrightarrow^{\bar{R}_{\geq 2}} T.$$

The transition sequences $\bar{R}_{\geq 2}$ and $\bar{R}'_{<k}, \bar{R}'_{>k}$ are of lengths $n-1$ and $n'-1$ respectively, both less than or equal to i. Thus, by the induction hypothesis $T \simeq T'$ and $n - 1 = n' - 1$. Thus,

$$n = n' \qquad T \simeq T'.$$

$\square$

TRITA -EECS-EX