Permission-Based Separation Logic for *Scala*

`Master's Thesis`

Author: Charl de Leur

Graduation Comittee:
prof. dr. M. Huisman
dr. S.C.C. Blom
dr. J. Kuper

Version of: August 25, 2015

Abstract

Scala is a versatile multi-paradigm general purpose programming language on the *Java Virtual Machine*, which offers full compatibility with existing *Java* libraries and code, while providing a multitude of advanced features compared to *Java* itself.

Permission-based separation logic has proven to be a powerful formalism in reasoning about memory and concurrency in object-oriented programs – specifically in *Java*, but there are still challenges in reasoning about more advanced languages such as *Scala*.

Of the features *Scala* provides beyond *Java*, this thesis focusses on first class functions and lexical closures. A formal model subset of *Scala* is defined, around these features. Using this foundation we present an extension to permission-based separation logic to specify and reason about functions and closures. Furtermore we provide a demonstration and an argument for its use by performing a case study on the Scala actors library.

Contents

1 Introduction

1.1 Motivation

The recent years have seen the growing popularity of language features, being borrowed from different paradigms such as functional programming, added to imperative and object-oriented languages to create hybrid paradigm languages. Popular examples include the inclusion of first-class functions in languages such as *C#*, with version 3.0, and *Java*, with version 8.0. There are, however, languages which take this approach even further, and do not just add features to an existing paradigm, but mix entire paradigms to allow for many-faceted approaches to programming challenges. One of the premier languages in this regard is *Scala*.

Meanwhile the recent years have also seen the rise of multi-threading and concurrency as a means to quench the ever-increasing thirst for computing power. With this new focus on concurrency, also came a response from the proponents of formal methods in computer science, with model checking and concurrent program verification techniques allowing for a more reliable creation of concurrent programs. This is important, as writing these concurrent programs by hand, without formal techniques, proved error-prone.

In this thesis we examine concurrent program verification techniques – especially the use of separation logic – in how they manage with a multi-paradigm language such as *Scala*. We do so by first examing the current state of the art in program verification using separation logic and examining the *Scala* language itself. We will then start a formalization process in which we develop a formal semantics for an interesting subset of *Scala* with an accompanying separation logic to prove its correctness. Finally we will provide a case study in which we use our logic to provide a specification of the *Scala* actor concurrency library.

1.2 Contribution

Our contribution is a means to specify *Scala* programs using permission-based separation logic, with a focus on a concise and correct method to specify first-class functions and lexical closures and a case study, which demonstrates our approach and its viability. We will provide a formalization of a subset of *Scala* including lexical closures and exceptions. Using this formalized subset, we will establish type-safety and a separation logic to establish memory safety and race freedom.

1.3 Document Outline

Background (Section 2)

In the *Background* section, we start with an introduction to, and background of, formal program verification using, first, Hoare logic and following that, Separation Logic. We show the defining features of separation logic and their advantages and uses in the verification of shared memory languages and concurrency. Secondly, we have a look at formal semantics, their uses and their role in program verification, by giving a short overview of the three most common forms of formalizing semantics, being *axiomatic*, *operational* and *denotational* formalization.

The *Scala* Language (Section 3)

In this chapter we describe the *Scala* programming language and its distinctive features. We provide examples with listings where relevant, to facilitate a basic understanding of the language, as required for this thesis.

A Formalized Model Language for *Scala* (Section 4)

An essential part of program verification, is the formalization of the semantics of the language being verified. As *Scala* is a multi-paradigm language with too many features to cover in this work, we define a number of subsets: We start with a basic expression language, which we first expand with first-class functions and closures, then with exceptions, classes and finally multi-threading. These subsets of *Scala* are then consecutively formalized using a program-context approach, resulting in what we call *Scala Core*.

Separation Logic for *Scala Core* (Section 5)

The primary goal in this section is to provide a variant of separation logic which can be used to verify programs written in our model language, secondly we provide typing rules, to assure type-safety. We start with the basic expression language with functions, as this is one of the most interesting cases for verification and continue towards exceptions and multi-threading.

A Case Study (Section 6)

With our formalization complete, it is interesting to see how it would function when used to specify a real-world larger piece of software. In this section we do so by writing a specification for the *Scala* actor library, which is as an alternative to shared-memory concurrency.

Conclusions and Future Work (Section 7)

Finally this chapter concludes our work by summarizing it, comparing it to similar approaches and describing what work remains to be done.

2 Background Information & Previous Work

For the work presented in this thesis, we build on previous work in the verification of (concurrent) programs, specifically via the use of separation logic and on the work in formal semantics. In this section we shall expand on the existing work in these areas, with a focus on the *JVM* and on *Scala* in particular. We shall start with a general introduction to static contract analysis in Section 2.1 and program logics in Section 2.2 and proceed with an introduction to separation logic in Section 2.3.1, which we shall expand to *Concurrent Separation Logic* in Section 2.3. Finally we shall give an introduction to formal semantics in Section 2.4

2.1 Static Contract Analysis

Of all the recent formal methods for program analysis, such as software model checking [54], static analysis [42, 41] and interactive theorem proving [33], our focus will be on program verification using logic assertions in program code, as first suggested by Hoare [29]. These assertions form contracts [39] between computational modules in software from which proof obligations can be derived and solved. This analysis can be done without executing the application, making it static in nature. Well-known tools based on this formalism include the more academic tools *Esc/Java* [24] and *Spec#* [8] and the more commercially used *Code Contracts* [23]. These tools are unfortunately restricted in the sense that they, and the formalisms backing them, break in concurrent situations. This is especially jarring as many applications, including all with a GUI, written in languages such as *C#*and *Java*, are concurrent in nature.

2.2 Verification using Classic Program Logic

2.2.1 Properties of Code

```
sort(a : Int[]) : Int[]
{
    ...
}
```

Listing 1: Simple Sort

We shall illustrate properties of code, using Listing 1, which shows a simple method, that sorts a given array. We can, independently of the implementation, state that this method indeed sorts an array, in first-order logic – e.g. as $\forall i, j.0 \leq i < j < a.length \Rightarrow a[i] \leq a[j]$. Unfortunately, there is no practical means to tell where this property holds: It could just as well have been a requirement for the method to execute, instead of a condition on the result. The solution, as pioneered by Hoare, involves making location explicit, by dividing the properties in so-called preconditions and postconditions [29]:

- **Precondition**: A property that should hold at method entry, specifying what is required to deliver correct output.
- **Postcondition**: A property that should hold at method exit, specifying what the method guarantees to the caller.

With these, we can now speak of what are commonly referred to as *Hoare triples*: Triples in the form of $\{P\}C\{Q\}$, where P is a precondition, C is a statement and Q is a postcondition. the triple is defined as having the following interpretation:

- **Definition:** Given that P is satisfied in a state s, and C terminates in state s', then Q is satisfied in state s'.
- **Alternatively:** The statement C *requires* the precondition P, to *ensure* the postcondition Q.

We can now specify a Hoare triple for the example in Listing 1 – in Listing 2:

```
// {true}
  sort(a : Array[Int]) : Array[Int]
  {
      /* [...] */
  }
// {∀ i, j. 0 ≤ i < j < length(a) ⇒ a[i] ≤ a[j]}
```

Listing 2: Simple Sort with Hoare Triple

2.2.2 Reasoning about Properties

Once we have a Hoare triple for a method – also called a proof outline – the next step is to reason about them and establish their validity. This is done by applying the axioms and logic rules of *Hoare logic* to determine a truth value [29]. A simple example of an axiom being the empty statement axiom, which states that any preconditions holding before the `skip`-statement will hold after – essentially saying that `skip` has no impact on the program state – is shown in Fig. 1:

$$Skip\ \frac{}{\{P\}skip\{P\}}$$

Figure 1: The Skip Axiom

An example of a rule is the sequential composition rule – shown in Fig. 2 – which specifies the conditions that should hold for sequential statements:

$$Seq\ \frac{\{P\}S_1\{Q\}\ \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

Figure 2: The Seq Rule

Given these examples, proving correctness of a program would seem like a lot of work, and indeed, these correctness proofs tend to take up sheets and sheets with rules being applied over and over until finally axiomatic statements are reached. Fortunately, this part can be largely (but not entirely) automated [24], resulting in the proof outline being sufficient to automatically determine correctness. This provides the basis for the practical use of formal verification via program logics. One essentially provides proof outlines and lets the automatic reasoning tool determine whether they hold; this type of verification is often referred to as static checking.

2.2.3 Relation to Programming by Contract

The pre- and postconditions mentioned in Section 2.2.1 may sound familiar to anyone familiar with the concept of design by contract (DBC); and indeed, the specifications are similar. In practice, in DBC, the specifications are often defined as a part of the programming language itself and thus executable [38]. The contracts are compiled to executable code along with the program code and thus violations of the contract are prohibited at runtime. This also means that DBC by itself makes no guarantees, without executing the program (of which the specification is now part). Because of this, it is often referred to as runtime checking, as opposed to the static checking mentioned in Section 2.2.2.

However, design by contract and static checking are not incompatible: Static checkers are often used in conjunction with DBC, as the specifications are largely similar. Examples would be *ESC/Java*, and the *JML*-tooling, that both use

the *JML* language as specifications to provide static checking and runtime checking respectively, and *Code Contracts*, which provides runtime as well as static checking on the same specifications.

2.2.4 Limitations of the Classic Approach

So far it seems that classic program logics are quite powerful in reasoning about programs, to the point that static checking of advanced specification languages such as *JML* can be built on top of them. However, there are limitations in the classical approach, mostly concerned with reasoning about pointers and memory:

$$\frac{}{\{P[E/x]\}x := E\{P\}}\ \textbf{Assignment Axiom}$$

$$\frac{}{\{y + 7 > 42\}x := y + 7\{x > 42\}}\ \text{Assignment with values}$$

$$\frac{}{\{y.a = 42\}x.a := 7\{y.a = 42\}}\ \text{Assignment with pointers}$$

Figure 3: The Issue with Pointers

To illustrate the pointer issue, Fig. 3 shows an example of the assignment axiom in use, which states that if P holds and all occurences of the assigned expression E in P are replaced by the variable x, P should still hold. For the example with values this clearly holds and for the example with pointers it seems to hold, but this turns out to be unsound when x aliases y, thus invalidating the axiom for use with pointers.

$$[y := -y;\ x := x + 1;\ y := -y] \quad (1)$$

$$[x := x + 1] \quad (2)$$

Figure 4: The Issue with Concurrency

Figure 4 shows two cases which in sequential execution satisfy the same input and output conditions, but in concurrent execution act differently due to interference with other statements of the first case. This means classic Hoare logic provides no guarantees of race-freedom. The result of this is that classic program logics are unsuitable to reason about concurrent programs, as any assertion established in a thread, can possibly be invalidated by another thread, at any time during the execution. While there exist logics which can take into account these multi-threaded scenarios, they tend to be either – in the case of *Owicki-Gries* [46] and *Rely-guarantee* [36] – too general to be of practical use, or – in the case of *concurrent Hoare logics* [30], too simplistic to handle the complex situations, involving heaps, in modern programming languages.

To allow for reasoning with concurrency and pointers, we must then look at a logic which can properly reason about memory; enter Separation logic (SL).

2.3 Separation Logic

2.3.1 Sequential Separation Logic

Separation logic is a recent generalization of Hoare logic, developed by O'Hearn, Ishtiaq, Reynolds, and Yang [45, 35, 52] based on the work by Burstall [16]. It allows for specification of pointer manipulation, transfer of ownership and modular reasoning between concurrent modules. Furthermore, it works on the principal of *local reasoning*, where only the portion of memory modified or used by a component is specified, instead of the entire program. It allows us to reason about memory by adding two concepts to the logical assertions, namely the *store* and the *heap*. The store is a function that maps local (stack) variables to values and the heap is a partial function that maps memory

locations to values (representing objects or otherwise dynamically allocated memory). These additions allow us to make judgements of the form $s, h \vDash P$, where s is a store, p a heap and P an assertion about them.

To allow for concise definitions of assertions over the heap and store, classical separation logic extends first-order logic with four assertions:

$\langle assert \rangle ::= \cdots$	
$\mid$ **emp**	empty heap
$\mid \langle exp \rangle \mapsto \langle exp \rangle$	singleton heap
$\mid \langle assert \rangle * \langle assert \rangle$	separating conjunction
$\mid \langle assert \rangle \mathrel{-\!\!*} \langle assert \rangle$	separating implication

Figure 5: Extensions to First-Order Logic

- The *emp* predicate denotes the empty heap and acts as the unit element for separation logic operations.
- The points-to predicate $e \mapsto e'$ means that the location e maps to value e'.
- The resource (or separating) conjunction $\phi * \psi$ means that the heap h can be split up in 2 disjoint parts $h_1 \perp h_2$ where $s, h_1 \vDash \phi$, and $s, h_2 \vDash \psi$.
- $\phi \mathrel{-\!\!*} \psi$ asserts that, if the current heap is extended with a disjoint part in which ϕ holds, then ψ will hold in the extended heap.

However, we will mostly be dealing with al alternate variant, called intuitionistic separation logic[21, 47], which, instead of extending classic first-order logic, extends intuitionistic logic. This is as classical separation logic is based on reasoning about the entire heap, which presents issues with garbage collected language like *Scala*, where the heap is in a state of flux and cannot generally be completely specified. The intuitionistic variant therefore admits weakening, that is to say $P * Q \Rightarrow Q$. Normally this would allow for memory leaks, but the garbage collection takes care of this. Instead of using *emp* as the unit element, intuitinionistic separation logic drops this predicate and uses *true*.

$$e \mapsto e_1, \ldots, e_n \stackrel{def}{=} e \mapsto e_1 * \ldots * e + (n-1) \mapsto e_n$$

$$p = x \mapsto 3 \qquad r = x \mapsto 3, y$$

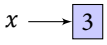
$$q = y \mapsto 3 \qquad s = y \mapsto 3, x$$

$$p \mathrel{-\!\!*} q$$

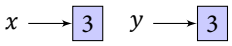
Figure 6: Example Assertions in Separation Logic

Given this syntax, we will now visualize the examples given in Fig. 6:

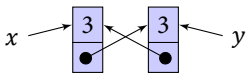
- p asserts that x maps to a cell containing 3.



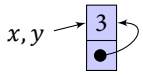
- $p * q$ asserts that p and q hold in disjoint parts of the heap.



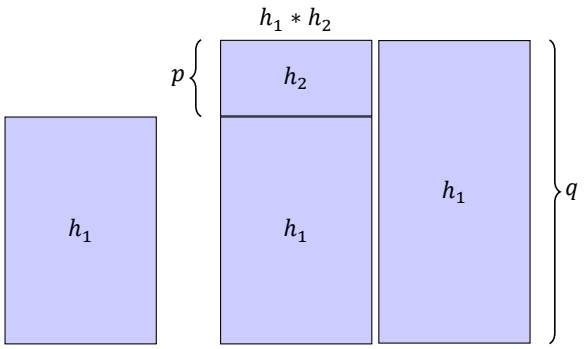
- $r * s$ asserts that two adjacent pairs hold in disjoint heaps.



- $r \wedge s$ asserts that two adjacent pairs hold in the same heap.



- $p \twoheadrightarrow q$ asserts that, if the current heap is extended with a disjoint part in which p holds, then q will hold in the extended heap.



Now let us look at an example specification of a simple *Scala*-method, with **`PointsTo`** being the ASCII representation of ↦:

```
class Simple
{
    var x = 0
    var y = 1
    var z = 3

    /*@
        requires PointsTo(x, _)
        ensures PointsTo(x, \result)
    */
    def inc() : Int
    {
        x = x+1
        x
    }
}
```

Listing 3: A Simple Specification in Separation Logic

The exact meaning of the specification in Listing 3 is not yet relevant, but an important detail to note is that the specification only mentions the part of the heap relevant to the method, in this case x. This is called local reasoning, made possible by the frame rule – shown in Fig. 7 – which prevents us from repeatedly having to specify the entire heap.

$$Frame\ \frac{\{P\}S\{Q\}}{\{P * R\}S\{Q * R\}}\ \text{None of the variables modified in } S \text{ occur free in } R$$

Figure 7: The Frame Rule

The frame rule states that if a program can execute in a small (local) state satisfying P, it can execute in an expanded

state, satisfying $P * R$, and that its execution will not alter the expanded part of the state i.e. the heap – outside of what is locally relevant – does not need to be considered when writing specifications.

2.3.2 Abstract Predicates

A practical extension to separation logic, especially for verification of data structures, is *abstract predicates* [48]. Similarly to abstract data types in programming languages, abstract predicates add abstraction to the logical framework.

Abstract predicates consist of a name and a formula and are *scoped*:

- Verified code inside the scope can use both the predicate's name and body.
- Verified code outside the scope must treat the predicate atomically.
- Free variables in the body should be contained in the arguments to the predicate.

To illustrate the use of abstract predicates for data abstraction, Fig. 8 shows the abstraction for singly-linked lists which is defined by induction on the length of the sequence α. The **list** predicate takes a sequence and a pointer to the first element as its arguments; An empty list consists of an empty sequence and a pointer to **nil** and longer lists are inductively defined by a pointer to a first element and the assertion that the following sequence is once again a list.

$$\mathbf{list}\,\epsilon\,i \stackrel{def}{=} \mathbf{emp} \wedge i \mapsto \mathbf{nil}$$
$$\mathbf{list}\,a : \alpha\,i \stackrel{def}{=} \exists j.i \mapsto a, j * \mathbf{list}\,\alpha j$$

Figure 8: A List Abstraction using Abstract Predicates

Figure 9 shows another example of abstract predicates, but one where a predicate functions more like a access ticket.

$\{Ticket\langle x\rangle \stackrel{def}{=} \mathbf{true}\}$

$\{\mathbf{true}\}$ `getTicket() : Int { }` $\{Ticket\langle ret\rangle\}$	$\{Ticket\langle x\rangle\}$ `useTicket(x : Int) { }` $\{\mathbf{true}\}$

Figure 9: An Example use of Abstract Predicates

2.3.3 Fractional Permissions

One of the useful extensions for verification of concurrent programs, is *permissions* [12]. In the logic described in Section 2.3.1 the points-to predicate is used to describe access to the heap. Another way to look at this is that the points-to predicate requests permission to access a certain part of the heap. The use of fractional permissions in separation logic makes this notion explicit by extending the points-to predicate with an additional fractional value in $(0, 1]$, where any value $0 < v < 1$ requests permission to read and a value of $v = 1$ to write. When proving the soundness of the verification rules, a global invariant is maintained, requiring the sum of all permissions for each variable to be less or equal to 1. This invariant, combined with the extended predicate, makes sure that a variable is either written by one, read by one or more, or untouched, guaranteeing mutual exclusion.

We shall illustrate this with an example in Listing 4:

```
class Simple
{
    var x = 0

    /*@
        requires PointsTo(x, 1, _)
        ensures PointsTo(x, \result)
    */
    def inc() : Int
    {
        x = x+1
        x
    }

    /*@
        requires PointsTo(x, 1/2,  _)
        ensures PointsTo(x, 1/2, _)
    */
    def read() : Int
    {
        x
    }
}
```

Listing 4: A Simple Example of Permissions

In Listing 4 `inc()` requires a permission of 1, and `read()` one of $\frac{1}{2}$, so at most two threads are allowed to read `x` using `read()`, but only one is ever allowed to increment, at a time.

When considering resources and permissions, the magic wand gains another use in the trading of permissions: Given a heap in which $p \mathrel{-\!\!*} q$ holds, the resource p can be consumed, yielding the resource q. This use is visualized in Fig. 10.

- $r = (x \overset{0.3}{\mapsto} 9) \mathrel{-\!\!*} (x \overset{1}{\mapsto} 9)$ holds in h_1.

  $h_1 = x \xrightarrow{0.7} \boxed{9}$

- Heap extension happens as before, but permissions are combined.

  $x \xrightarrow{0.3 + 0.7} \boxed{9}$

- Given the resource needed, we can trade: $((x \overset{0.3}{\mapsto} 9) * r) \Rightarrow (x \overset{1}{\mapsto} 9)$

  $h_1 = x \xrightarrow{1} \boxed{9}$

- $(x \overset{0.3}{\mapsto} 9)$ is consumed in the trade.

Figure 10: Trading Permissions using Separating Implication

2.3.4 Locks

Another key addition we require, is a means to reason about locks and reentrancy. Fortunately, a solution [28, 5] exists:

First we define **inv**, the so-called resource invariant, describing the resources a lock protects e.g. in Listing 5 the resource invariant protects x.

```
class Simple
{
    /*@ inv = PointsTo(x, 1 _)  */

    var x = 0
    /*@ commit */

    /*@
        requires Lockset(S) * (S contains this -* inv) * initialized
        ensures Lockset(S) * (S contains this -* inv)
    */
    def inc() : Int
    {
        synchronized
        {
            x+=1
            x
        }
    }
}
```

Listing 5: Specification of a Lock in Separation Logic

As **inv** may require initialization before becoming invariant, the invariant starts in a suspended state. Only when we commit the invariant, it is actually required to invariantly hold. A common place for such a commit to happen would be at the end of a constructor as in Listing 5.

Secondly we extend the logic with the following predicates:

- $Lockset(S)$: The multiset of locks held by the current thread. $Lockset(S)$ is empty on creation of a new thread.
- $e.fresh$: The resource invariant of e is not yet initialized en therefore doesn't hold.
- $e.initialized$: The resource invariant of e can be safely assumed to hold.
- $S\ contains\ e$: The multiset of locks of the current thread contains the lock e.

We will require an addition to the rule for object creation, regardless of the actual specifics of the existing rule, that specifies that all new objects (and therefore locks) are fresh. Furthermore, we require the additional rules given in Fig. 11:

- The **lock** rule applies for a lock that is acquired non-reentrantly. The precondition specifies this stating there is a lockset S for this thread, but the lock is not part of it. Furthermore the lock is required to have an initialized resource invariant. In the postcondition the lock must have been added to the lockset.
- The **relock** rule is a simple variant of **lock**, for the reentrant case.
- The **unlock** rule is the dual of **lock**, requiring the lock in the lockset in the precondition and having it removed in the postcondition. Once again a simple variant exists for the reentrant case.

- Finally we have the **commit** rule, which promotes a fresh lock to an initialized one.

In the rules given we assume, for easier illustration, a language with dedicated *lock* and *unlock* primitives, but this is extendable to constructions like *synchronized* and *wait/notify*.

$$\textbf{Lock}\ \frac{\{\text{Lockset(S)} * \neg(\text{S contains u}) * \text{u.initialized}\}\ \{\text{lock(u)}\}\ \{\text{Lockset(u.S)} * \text{u.inv}\}}{}$$

$$\textbf{Relock}\ \frac{\{\text{Lockset(u.S)}\}\ \{\text{lock(u)}\}\ \{\text{Lockset(u.u.S)}\}}{}$$

$$\textbf{Unlock}\ \frac{\{\text{Lockset(u.S)} * \text{u.inv}\}\ \{\text{unlock(u)}\}\ \{\text{Lockset(S)}\}}{}$$

$$\textbf{Commit}\ \frac{\{\text{Lockset(S)} * \text{u.fresh}\}\ \{\text{u.commit}\}\ \{\text{Lockset(S)} * \neg(\text{S contains u}) * \text{u.initialized}\}}{}$$

Figure 11: Extra Rules to Deal with Locks

2.4 Formal Semantics

As previously stated in Section 1, one of our goals is to provide a formal semantics for *Scala*; Here we shall give an introduction to formal semantics and its relevance to this thesis.

Programming languages are generally specified informally, using a language specification document, such as the ones for *Java* [27] and *Scala* [43], but to reason about languages using the previously covered program logics, this is insufficient; For them languages need to have a strict mathematical meaning, which can be linked to and used in the logical formulas. Such a mathematical description of language meaning is called a formal program semantics.

The idea of program semantics was introduced by Floyd [25] and formal semantics now exist in three major categories, namely axiomatic semantics, operational semantics and denotational semantics. For our purposes, we will mainly be interested in operational semantics, for the language itself, and, to a lesser extent, in axiomatic semantics and denotational semantics, to define the meaning of the assertion logic and its relation to the language semantics.

Operational semantics describe what is valid in a language as sequences of computational steps. They do so either – in the case of big-step semantics – by describing the overall result of an execution [34] or – in the case of small-step semantics – by describing the individual steps of the computation [50], using rules. Because our work is focused on concurrency, we will be dealing with the latter. We shall give a small example of a small-step semantics of a toy language:

$$e ::= m \mid e_0 + e_1$$

Figure 12: A Very Simple Language

Our demonstration language will consist of expressions, which can be either be a constant or an addition between two expressions. These expressions can be evaluated in four steps:

- **Constants**
  1. Constants remain, as they are already evaluated with themselves as the value.

- **Addition**
  1. In $e_0 + e_1$, e_0 is evaluated to a constant, say m_0.
  2. In $e_0 + e_1$, e_1 is evaluated to a constant, say m_1.
  3. In $e_0 + e_1$, $m_0 + m_1$ is evaluated to a constant, say m_2.

These steps can now be formalized as rules, which take the form $\frac{\text{premises}}{\text{conclusion}}$:

$$\frac{e_0 \to e_0'}{e_0 + e_1 \to e_0' + e_1}$$

$$\frac{e_1 \to e_1'}{m_0 + e_1 \to m_0 + e_1'}$$

$$\frac{}{m_0 + m_1 \to m_2} \text{ With } m_2 \text{ the sum of } m_0 \text{ and } m_1.$$

Figure 13: Reduction Rules for the Very Simple Language

These rules now give a strict formal meaning to our simple language.

Axiomatic semantics describe meaning using rules and axioms, of which we have seen examples in Section 2.3.

Denotational semantics describe meaning by providing a mapping to a domain with a known semantics, generally based in mathmatics.

3 The *Scala* Programming Language

3.1 Introduction

Scala[1] [22] is a purely object-oriented language with a unified type-system, blending in functional concepts, implemented as a statically typed language on the *JVM*[2], seamlessly interoperating with the existing Java libraries [44, pp. 49, 55–58]. Notable Features of *Scala* include:

- First-class functions and lexical closures.
- Traits.
- Unified Type System.
- Case Classes.
- Singleton Objects.
- Pattern Matching.
- Limited Type Inference.
- Properties.
- Abstract Types.

As the work presented in this document depends on an understanding of the *Scala* language, we will take some time to mention and clarify some of the features used. We will assume an understanding of the *Java* language and *JVM*, as well as a basic understanding of functional programming and languages. Furthermore, this is not meant to be a full tutorial on *Scala*, as better resources for that exist elsewhere [44]. We shall illustrate the features of the language using the sample program given in Listing 7.

3.2 A Guided Tour of *Scala*

The program described in Listing 7 represents and evaluates simple propositional logic formulas without variables. It encodes the logical formula in a tree of objects and visits them recursively to evaluate the truth-value. We shall begin our in-depth examination with the encoding of the basic logical formulas *true* and *false*:

```
case object True extends PropositionalFormula
case object False extends PropositionalFormula
```

Listing 6: True and False

Besides the fact that all formulas extend the class **`PropositionalFormula`**, we immediately encounter two *Scala*-specific features, being the **`case`**-keyword and the **`object`**-keyword. Let us first look at **`object`** :

In addition to the **`class`** -keyword as it would be used in *Java*, *Scala* also supports **`object`** . This defines a *singleton* [26] object. There are no different possible instantiations of **`True`** and **`False`** , so making them a global singleton makes sense.

[1] A portmanteau of *Sca*lable and *La*nguage
[2] There exist other implementations e.g. on the *.Net* runtime, but the *JVM* is the primary one.

```scala
trait EvaluatableToBoolean
{
  def boolValue : Boolean
}

trait EvaluatableToInt
{
  def intValue : Int
}

trait EvaluatableToString
{
  def stringValue : String
  def prefix : String
  def print = prefix + stringValue
}

object PropositionalFormula
{
  def evaluate(phi:PropositionalFormula) : Boolean =
    phi match
    {
      case True => true
      case False => false
      case Not(r) => !evaluate(r)
      case And(l, r) => evaluate(l) && evaluate(r)
      case Or(l, r) => evaluate(l) || evaluate(r)
      case Implies(l, r) => !evaluate(l) || evaluate(r)
      case Equivalent(l, r) => evaluate(Implies(l, r)) && evaluate(Implies(r, l))
    }
}

sealed abstract class PropositionalFormula extends EvaluatableToBoolean
{
  def value = boolValue
  override def boolValue = PropositionalFormula.evaluate(this)

  def >(right:PropositionalFormula) : PropositionalFormula = Implies(this, right)
  def <>(right:PropositionalFormula) : PropositionalFormula = Equivalent(this, right)
  def &(right:PropositionalFormula) : PropositionalFormula = And(this, right)
  def |(right:PropositionalFormula) : PropositionalFormula = Or(this, right)
  def unary_! : PropositionalFormula = Not(this)
}

case class Not(right:PropositionalFormula)
     extends PropositionalFormula
case class And(left:PropositionalFormula, right:PropositionalFormula)
     extends PropositionalFormula
case class Or(left:PropositionalFormula, right:PropositionalFormula)
     extends PropositionalFormula
case class Equivalent(left:PropositionalFormula, right:PropositionalFormula)
  extends PropositionalFormula
case class Implies(left:PropositionalFormula, right:PropositionalFormula)
     extends PropositionalFormula
     with EvaluatableToInt
     with EvaluatableToString
{
  override def prefix = "Value is : "
  override def stringValue = PropositionalFormula.evaluate(this).toString()
  override def intValue = if(PropositionalFormula.evaluate(this)) 1 else 0
}
case object True extends PropositionalFormula
case object False extends PropositionalFormula


object QuickLook extends App
{
  var f = (True | False)
  val v1 = f.value
  f = ((!(True<>False) & False) > True)
  val v2 = PropositionalFormula.evaluate(f)
  val v3 = f.asInstanceOf[Implies].print
  val v4 = f.asInstanceOf[Implies].intValue
  val list = List(v1, v2, v3, v4)
  Console.println(list map ((e) => e.toString))
}
```

Listing 7: A Sample *Scala* Program

Secondly there is **case**: Coupled with **object** , **case** has relatively little impact, as it provides only a default serialization implementation and a prettier `toString` [43, p. 69]. We use the **case**-keyword with **object** to keep the definitions in line, syntactially, with the *case classes*, where the impact is much stronger. So let us have a look at those, with the encodings of the logical operations:

```
  case class Not(right:PropositionalFormula)
       extends PropositionalFormula
  case class And(left:PropositionalFormula, right:PropositionalFormula)
       extends PropositionalFormula
  case class Or(left:PropositionalFormula, right:PropositionalFormula)
       extends PropositionalFormula
  case class Equivalent(left:PropositionalFormula, right:PropositionalFormula)
    extends PropositionalFormula
```

Listing 8: Logical Operators

We encode a single unary logical operation and a couple of binary ones, as *case classes*. Classes prefixed with **case** are, by default, immutable data-containing classes, relying on their constructor-arguments for initialization. They allow for a compact initialization syntax, without **new**, have predefined structural equality and hash codes and are serializable [44, pp. 312–313]. They are also given a *companion object* with implementations of the *extractor methods* `apply` and `unapply`, which, respectively, construct the object, given its fields and return the fields of the object [43, p. 67, 44, Chapter 15]. For the simple definition of say **And**, the compiler generates Listing 9 as a companion object where `apply` returns the left and right operands of **And** as a tuple and where `unapply` creates an instance of **And** given the left and right operands.

```
final private object And extends scala.runtime.AbstractFunction2
  with ScalaObject with Serializable {
  def this(): object this.And = {
    And.super.this();
    ()
  };
  final override def toString(): java.lang.String = "And";
  case def unapply(x$0: this.And):
    Option[(this.PropositionalFormula, this.PropositionalFormula)] =
      if (x$0.==(null))
        scala.this.None
      else
        scala.Some.apply[(this.PropositionalFormula, this.PropositionalFormula)]
          (scala.Tuple2.apply[this.PropositionalFormula, this.PropositionalFormula](x$0.left, x$0.right));
  case def apply(left: this.PropositionalFormula, right: this.PropositionalFormula):
    this.And = new $anon.this.And(left, right)
};
```

Listing 9: Compiler-generated **And** Companion Object

The presence of `apply` and `unapply` allow us to *pattern match* on the case classes [43, p. 116] as you would on *algebraic datatypes*(ADTs) in functional languages such as *Haskell*. Generally case classes and case objects are therefore used to mimic ADTs, but they do remain full-fledged classes, with their own implementation details. In our case we use this to add methods to case classes and extend superclasses.

Now let's have a look at the base class of all our formulas, PropositionalFormula:

```
sealed abstract class PropositionalFormula extends EvaluatableToBoolean
{
  def value = boolValue
  override def boolValue = PropositionalFormula.evaluate(this)

  def >(right:PropositionalFormula) : PropositionalFormula = Implies(this, right)
  def <>(right:PropositionalFormula) : PropositionalFormula = Equivalent(this, right)
  def &(right:PropositionalFormula) : PropositionalFormula = And(this, right)
  def |(right:PropositionalFormula) : PropositionalFormula = Or(this, right)
  def unary_! : PropositionalFormula = Not(this)
}
```

Listing 10: PropositionalFormula class

PropositionalFormula is **abstract**, as there will be no instantiations of it, and **sealed**, meaning it may not be directly inherited from, outside of this source file. We seal the class because when matching over formulas, we want the compiler to warn us if we happen to forget any cases, without adding a default catch-all case. The compiler can only do this if it knows the full range of possible cases. In general this would be impossible, as new case classes can be defined at any time and in arbitrary compilation units, but sealing the base class makes all the cases contained to a single source file and known at compile-time [44, pp. 326–328].

Besides making sure that pattern matching is exhaustive, **PropositionalFormula** defines operators used with formulas, so we can write them in familiar infix notation instead of prefix constructor notation, e.g. **True** `&` **False** instead of **And**`(`**True**`,` **False**`)`.

Binary operators are defined as any other method or function, using the keyword **def**, with the name of the operator being the method name, but for unary operators the special syntax `unary_` is used. As opposed to other languages, both binary and unary operators are to be chosen from a restricted set of symbols; this explains the seemingly odd choice for the implication and equivalence operators, as **=** is restricted. Type information is added in postfix notation following ':', as opposed to the prefix notation used in *Java*.

Furthermore the class defines the method `value`, which evaluates the formula, by passing it to `evaluate`, on the other **PropositionalFormula**. Let us have a look a that one:

```
object PropositionalFormula
{
  def evaluate(phi:PropositionalFormula) : Boolean =
    phi match
    {
      case True => true
      case False => false
      case Not(r) => !evaluate(r)
      case And(l, r) => evaluate(l) && evaluate(r)
      case Or(l, r) => evaluate(l) || evaluate(r)
      case Implies(l, r) => !evaluate(l) || evaluate(r)
      case Equivalent(l, r) => evaluate(Implies(l, r))
                            && evaluate(Implies(r, l))
    }
}
```

Listing 11: PropositionalFormula singleton

Singleton objects we have seen before in Listing 6. However, because this one shares the name with a class, it is of a special type called *companion objects* [44, p. 110]. As *Scala* has no notion of static members, class definitions are often coupled with *singleton objects*, whose members act as static members would in *Java* [44, p. 111]. When these

objects are given the same name as classes, they're called companion objects and can call private members, on the instantiations of the class, and vice versa [44, p. 110].

It was already mentioned, during our treatment of **case**, but here we finally see an instance of pattern matching, in the `evaluate` method. The `evaluate` method looks at a **PropositionalFormula** and recursively determines the Boolean valuation. If the **PropositionalFormula** is either **True** or **False**, this is simple, but in the other cases we extract the operands of the operator, using the extractor methods provided by case classes and recursively determine their valuation. Then the built-in language operators are used to determine the valuation of the formula containing the operator.

We shall have a slightly more in-depth look at the *case sequence* used in pattern matching, as this will be relevant to the case study in Section 6 First some definitions:

```
abstract class Function1[-a,+b] {
    def apply(x: a): b
}
abstract class PartialFunction[-a,+b] extends Function1[a,b] {
    def isDefinedAt(x: a): boolean
}
```

Listing 12: Definition of Function1 and PartialFunction

A function in *Scala* is an object with an `apply`-method. The unary function, **Function1**, is predefined with `apply` taking a single *contravariant* argument and returning a single *covariant* result. *Scala* has predefined syntax for these types of functions:

```
new Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
}
```

Listing 13: Use of Function1

```
(x: Int) => x + 1
```

Listing 14: Shorthand

A partial function is mathematically a function mapping only a subset of a domain X to a domain Y. Since this would make every *Scala* function a partial function, a slightly different approach is used. The trait **PartialFunction** is defined as a subclass of **Function1**, with an additional method `isDefinedAt(x)`, which determines whether the parameter x is an element of the subset of X, making the domain of the partial function explicit.

A common use[3] of **PartialFunction** in *Scala* is the case sequence, which features heavily in pattern matching:

```
{
    case p_1 => e_1;
    /*      ⋮      */
    case p_n => e_n
}
```

Listing 15: Case sequence

[3]Only in the general case, as the compiler may perform optimizations which turn it into e.g. a nested conditional.

The cases $p_1 \cdots p_n$ define the partial domain. `isDefinedAt(x)` returns true, if one of these cases match the argument x, and **`Apply`**`(x)` returns the value e_m for the first pattern p_m that matches. Listing 16 demonstrates a concrete case sequence with its **`PartialFunction`** representation.

```
{
    case 0:Int => false
    case 1:Int => true
}
```

```
new PartialFunction[Int, Boolean] {
    def apply(d: Int) = (d==1)
    def isDefinedAt(d: Int) = (d == 0) || (d==1)
}
```

Listing 16: Simple Case Sequence, as a **`PartialFunction`**

Back in the sample application, we have another major feature to look at, namely *traits*:

```
trait EvaluatableToBoolean
{
  def boolValue : Boolean
}

trait EvaluatableToInt
{
  def intValue : Int
}

trait EvaluatableToString
{
  def stringValue : String
  def prefix : String
  def print = prefix + stringValue
}
```

Listing 17: Traits

Traits are essentially constructorless abstract classes, or partially implemented interfaces. A class can inherit from multiple traits, allowing for mixin class composition [13]. In our sample we have 3 traits, namely **`EvaluatableToBoolean`**, **`EvaluatableToInt`** and **`EvaluatableToString`**. The first two act just like interfaces would in *Java*, but in the last one the `print`-method is actually implemented, concisely showing the difference.

In Listing 10 we see that **`PropositionalFormula`** extends **`EvaluatableToBoolean`** and implements the method it requires using the **`override`** keyword. And in Listing 18 we see that **`Implies`** implements the other two traits, showing multiple trait inheritance and once again showing that case classes are fully fledged classes.

```
case class Implies(left:PropositionalFormula, right:PropositionalFormula)
     extends PropositionalFormula
     with EvaluatableToInt
     with EvaluatableToString
{
  override def prefix = "Value is : "
  override def stringValue = PropositionalFormula.evaluate(this).toString()
  override def intValue = if(PropositionalFormula.evaluate(this)) 1 else 0
}
```

Listing 18: Multiple Trait Inheritance

Finally our example is tied together with an object that extends **App**, which signifies an entry point to the application [44, p. 112] :

```
object QuickLook extends App
{
  var f = (True | False)
  val v1 = f.value
  f = ((!(True<>False) & False) > True)
  val v2 = PropositionalFormula.evaluate(f)
  val v3 = f.asInstanceOf[Implies].print
  val v4 = f.asInstanceOf[Implies].intValue
  val list = List(v1, v2, v3, v4)
  Console.println(list map ((e) => e.toString))
}
```

Listing 19: Main Entry Point

Listing 19 shows the actual implementation of the primary constructor is placed in the class body [44, pp. 140-142]. Constructor parameters of the primary constructor would ordinarily follow the class name akin to method definition, but as there are none in this case, the `()` can be ommited. In the case of *singletons*, this constructor runs when then object is first accessed [44, p. 112].

Some other features are quickly demonstrated by Listing 19 as well, namely the **`var`**-keyword for *mutables* and the **`val`**-keyword for *immutables*, the syntax for list construction **`List`**`(..)` and for anonymous functions **`=>`** and the use of **`InstanceOf`**`[t]` which acts as a cast. The operators we have defined before allow us to write our logical formulas in a familiar fashion. The use of `map` shows that the familiar `.` and `()` of *Java* are optional in *Scala*; in *Java* it would look something like **`List`**`.map(..)`.

4 A Model Language Based on *Scala*

4.1 Introduction

To allow for a proper foundation of the logical assertions in our specifications of *Scala* programs, we first have to assign a mathematical meaning to the programs in the form of a formal semantics. We will do this for a modeling language called *Scala Core*, which is a subset of *Scala*.

Our semantics will be based around program contexts, which have been previously used to model non-local control flow in C [37]. This approach allows for a more natural way of handling the non-local aspects of closures and exceptions in a small-step operational semantics.

With the importance of program contexts to our semantics, we shall first have an in-depth look at those in Section 4.2. With the knowledge of how program contexts can model expression trees, we will give a semantics for a basic expression language in Section 4.3; this simple language will form the basis of *Scala Core*. After defining the basic expression language, we will subsequently expand it with functions in Section 4.4, exceptions in Section 4.5, classes and traits in Section 4.6 and finally multithreading in Section 4.7. After defining the semantics in this fashion, we will compare it to those of the actual *Scala* language and other relevant semantics in Section 4.8.

4.2 Program Contexts & Zippers

To fully understand program contexts, it is important to first take a look at their main source of inspiration: The *zipper* data structure – originally proposed by Huet [32].

The zipper data structure is, as originally described, a representation of a tree, together with a currently focused subtree. Generally this subtree is called the focus and this representation is called the context. It is called a context because a focussed subtree occupies a certain location within the tree as a whole: a context for the subtree. We shall illustrate this with the simple example of an expression tree for the expression $(a \times b) + (c \times d)$:

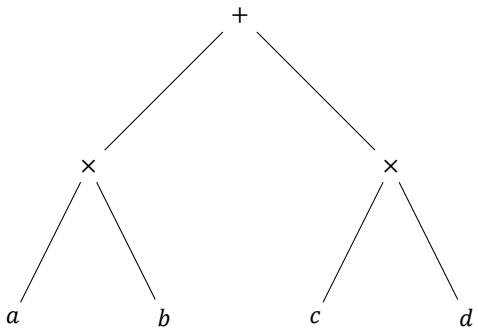


Figure 14: The Parse Tree of $(a \times b) + (c \times d)$

Figure 14 shows the parse tree of the sample expression, which we will encode in *Scala*, using the datatypes defined in Listing 20.

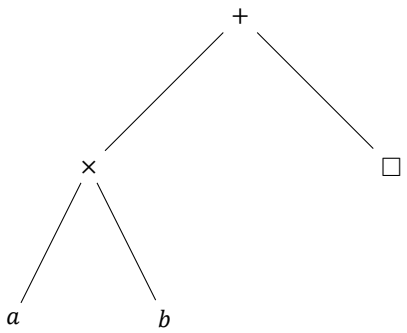
```
sealed abstract class BinaryTree
case class Leaf(item : String) extends BinaryTree
case class Split(left : BinaryTree, item : String,  right : BinaryTree) extends BinaryTree
```

Listing 20: Zipper Tree Data Type

A binary tree consists of either leafs with values or nodes with a left branch, a right branch and a value. Listing 21 shows the encoding of the expression parse tree shown in Fig. 14, using **`BinaryTree`**.

```
Split
(
    Split(Leaf("a"), "x", Leaf("b"))
    "+"
    Split(Leaf("c"), "x", Leaf("d"))
)
```

Listing 21: The Parse Tree of $(a \times b) + (c \times d)$ in *Scala*



Listing 22: The Context of the Right × in the Parse Tree of $(a \times b) + (c \times d)$

Now, if we look at the right × in Fig. 14, its context would be given by Listing 22: a tree with a hole where the focused subtree would fit. A way to concisely describe this context is by means of a path from the focused subtree to the root node. For instance, to reach the root from our subtree, we need to go up the right branch of the root. To demonstrate this notion in *Scala*, Listing 23 defines **`Context`**: A context consists of Top, for a root hole, or a left hole, with its parent context and its right sibling, or a right hole, with its left sibling and its parent context. Using this data structure, Listing 24 shows the encoding of the context of the right ×: It is a child of the root node and appears to the right of $a \times b$.

```
sealed abstract class context
case class Top() extends Context
case class L(parent : Context, sibling : Tree) extends Context
case class R(sibling : Tree, parent : Context) extends Context
```

Listing 23: Zipper Context Data Type

```
R(Split(Leaf("a"), "x", Leaf("b")), Top())
```

Listing 24: The Context of × in the Parse Tree of $(a \times b) + (c \times d)$ in *Scala*

Moving the focus to another part of the tree is easily expressed, as the movement itself is part of the path describing the context. `L(R(Split(Leaf("a"), "x", Leaf("b")), Top()), Leaf("d"))` for instance, would be the context, were we to move the focus to c, as it is the left sibling of d and its parent context is the one from Listing 24.

For more complicated expressions, the approachs remains the same: for $(a \times b) + (c \times (2 \times d)$ for instance, the context of c would be `L(R(Split(Leaf("a"), "x", Leaf("b")), Top()), Split(Leaf("2"), "x", Lead("d")))`, which is as before, but with the right sibling of c now being $2 \times d$.

Another way to express these contexts, is by means of a list of trees, annotated with the direction that was chosen. Listing 25 demonstrates this approach in with the focus on c in $(a \times b) + (c \times d)$.

```
sealed abstract class Direction
case object L : Direction
case object R : Direction

case class SingularContext(d : Direction, t : Tree)

List
(
    SingularContext(Split(Leaf("a"), "x", Leaf("b")), R),
    SingularContext(Leaf("d"), L)
)
```

Listing 25: List Representation of Contexts

Following Krebbers et al. [37], we can adapt and extend the zipper to model program execution: The focus shall be on (sub)expressions in our model language and the context will consist of the expression turned inside-out representing a path from the focused subexpression to the whole expression, i.e. the steps executed in the current scope to reach the focused subexpression. We will illustrate this with an example based on the *Scala Core* program given in Listing 26:

```
var x: Int;
x := 2*3+4*5
```

Listing 26: A Simple *Scala Core* Program

The example listing in Listing 26 is a *Scala Core* program, similar in meaning to the example expression we used before, but with concrete values for a, b, c, d and a variable assignment as the topmost expression. Figure 15 visualizes this program as a parse tree.

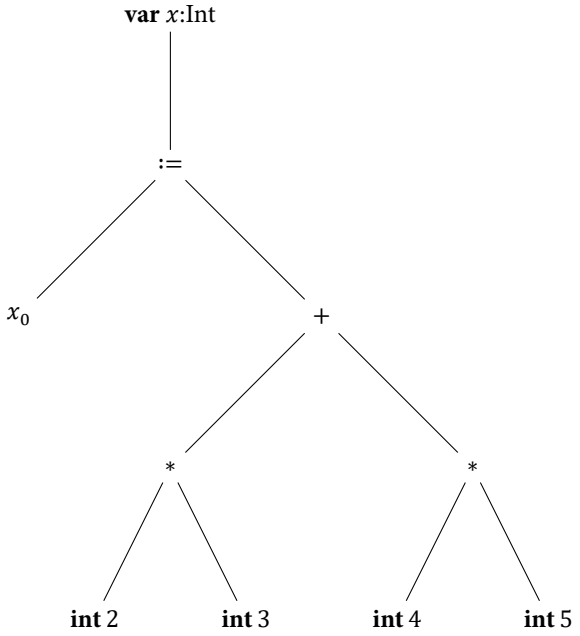


Figure 15: The Parse Tree of Listing 26

When we focus the rightmost multiplication, the context can be once again visualized – as shown in Fig. 16.

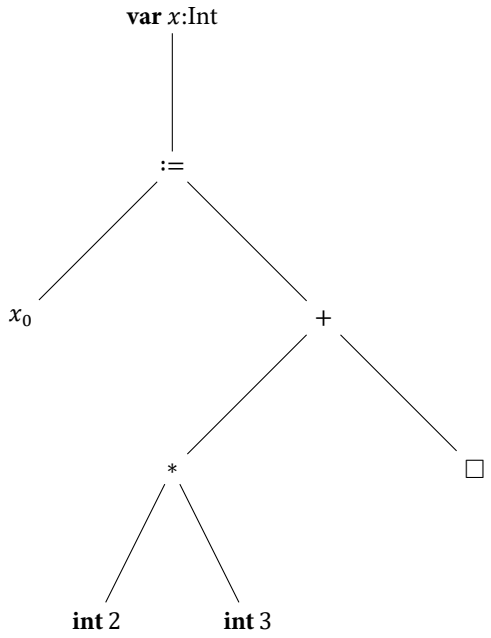


Figure 16: The Parse Tree of Listing 26

However, since we will be using this expression tree in a reduction semantics, which rewrites the tree along the way to propagate evaluated expressions, we will use a slightly different representation – as shown in Fig. 17 – where the expressions that have been evaluated, have been replaced by their values.

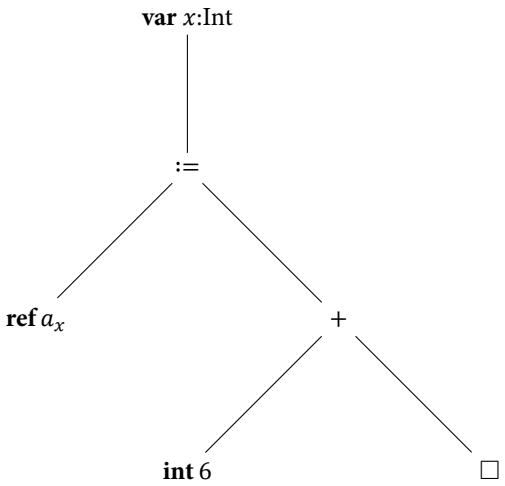


Figure 17: The Visualized Context of `4*5` in Listing 26 with evaluated expressions

Instead of using figures of trees, or *Scala* programs, we shall express the context of a focused expression in lists of what we shall call singular expression contexts. For the given example we shall define a number of singular expression contexts – as shown in 18:

$$\langle bop \rangle \ ::= ; \mid + \mid := \mid * \mid \ldots$$
$$K_S \ ::= \square \ \langle bop \rangle \ e_2 \mid v_1 \ \langle bop \rangle \ \square \mid \text{var}_a \ \square$$

Figure 18: Singular program contexts

The meaning of these singular contexts is as follows:

- The two binary expression contexts, $\square \ \langle bop \rangle \ e_2$ and $v_1 \ \langle bop \rangle \ \square$, mark the currently focused statement as either the left or right subexpression of a binary expression. if the binary expression context is of the second subexpression, the first subexpression has been evaluated to a value.
- The variable block context $\text{var}_a \ \square$ marks the current expression as the subexpression of a variable block, where the address of the variable is a.

We will add more singular expression contexts to the definition as they are required, e.g. when dealing with function calls.

Using these singular expression contexts, we can now succinctly express the context of `4*5` in the simple program – as shown in Fig. 19.

$$\mathbf{k} = \mathbf{int}\ 6 + \square$$
$$:: \mathbf{ref}\ a_x := \square$$
$$:: \text{var}_{a_x}$$
$$:: []$$

Figure 19: The Context of `4*5` in Listing 26

`4*5` is the right hand side of the addition where the left hand side has been evaluated to 6, so the direct context is of

the form $6 + \square$. The context of the addition is right hand side of the assignment where the left hand side is **ref** a_x, i.e. **ref** $a_x := \square$. Finally the context of the addition is the variable declaration block$_{a_x}$.

As before, we can traverse upwards and downwards along the zipper, in this case to subexpressions and superexpressions respectively. For instance, if we move downwards from `4*5`, the focus shall change to `4`, with the following context:

$$
\begin{aligned}
\mathbf{k} = {} & \square * \mathbf{int}\ 5 \\
:: {} & \mathbf{int}\ 6 + \square \\
:: {} & \mathbf{ref}\ a_x := \square \\
:: {} & \mathrm{var}_{a_x} \\
:: {} & []
\end{aligned}
$$

Figure 20: The Context of `4` in Listing 26

In the same fashion, if we move upwards from `4*5`, the focus changes to `2*3+4*5`, with the following context:

$$
\begin{aligned}
\mathbf{k} = {} & \mathbf{ref}\ a_x := \square \\
:: {} & \mathrm{var}_{a_x} \\
:: {} & []
\end{aligned}
$$

Figure 21: The Context of `2*3+4*5` in Listing 26

The practical effect of defining the data structure in this fashion, is that each focused statement comes with the execution history relevant to its scope, in the form of the context. It is this history which allows us to add non-local effects to the semantics. In the next section we will demonstrate the program contexts in action, as we shall describe the basic structure of program context semantics.

4.3 *Scala Core* with only Basic Expressions

For this first foray into program context semantics for a *Scala Core*, we will have a look at the simple expression language which will form the base of our model language. First we will define a syntax for this basic language, then we will provide and explain the runtime structures necessary for the semantics and provide a number of reduction rules and finally we will give a a sample reduction of a small program.

4.3.1 Syntax

The syntax of the basic expression language is defined as follows:

$$
\begin{aligned}
\mathrm{Type} &::= \mathbf{Ref}\ \mathrm{Type} \mid \mathbf{Int} \mid \mathbf{Unit} \\
\mathrm{Value} &::= \mathbf{ref}\ \mathrm{Address} \mid \mathbf{int}\ \mathrm{Integer} \mid \mathbf{unit} \\
\mathrm{Integer} &= \mathbb{N} \\
\mathrm{Address} &= \mathbb{N}
\end{aligned}
$$

Figure 22: Types & Values in the Basic Expression Language

There are only 3 types: *Pointer*, which is the type of addresses in memory, parametrized with the type of the element it points to, *Int*, the type of integer values and the *Unit* type. Corresponding to these types we have pointer values, integer values and the unit instance. The unit instance can not be directly used, but is used as the return type of expressions such as assignment.

In the semantics, we will generally refer to types as t_i, values as v_i and addresses as a_i, with $i \in \mathbb{N}$, dropping the subscript in the singular case.

We have the following expressions, which we will refer to using e_i:

$$
\begin{aligned}
\langle bop \rangle &::= ; \mid :=\\
\text{Expression} &::= \text{Expression } \langle bop \rangle \text{ Expression} \mid \textbf{load}\ \text{Expression} \mid \textbf{var}\ \text{Type}; \text{Expression} \mid x_i\\
&\quad \mid \text{Value}
\end{aligned}
$$

Figure 23: Expressions in the Basic Expression Language

- Assignment $e_1 := e_2$, which stores a value to a location on the heap. It will evaluate to **unit**.
- Sequential $e_1; e_2$, which evaluates two expressions sequentially. It will evaluate to the value of the second expression.
- Load **load** e, which loads a value from a given address in the heap.
- VarBlock **var** t; e, which declares a variable of type t. The variable has no name, as we refer to variables by their position on the stack instead of by name.
- Ident x_i, which provides the value at the i^{th} position of the stack.

4.3.2 Semantics: Runtime Structure

For our model language we shall give a small-step operational semantics which makes extensive use of the previously introduced program contexts.

For this we need to define the runtime state of our semantics:

$$
\begin{aligned}
\text{State} &= \text{Context} \times \text{Focus} \times \text{Heap}\\
\text{Context} &= \overrightarrow{K_S}\\
\text{Focus} &= \text{Direction} \times \text{Expression}\\
\text{Direction} &::= \searrow \mid \nearrow\\
\text{Heap} &= \text{Address} \rightharpoonup \text{Type} \times \text{HeapElement}\\
\text{HeapElement} &= \text{Value}
\end{aligned}
$$

Figure 24: Runtime Structure

States consist of a context, a focus and a heap. We will refer to states as S_i and use the syntactic shorthand $S_i = \mathbf{S}(k, \phi, h)$.

With only primitive types, a heap is a finite partial function from addresses to pairs of a type and a value; we shall refer to them using h_i. We shall use the shorthand $h[a \mapsto v]$ to store values in the heap and use $h[a \mapsto \bot]$ to allocate unassigned variables.

Contexts are lists of singular expression contexts, as described in Section 4.2. We generally refer to them using k_i. We shall define the singular expressions for the simple language in Fig. 25.

Foci consist of the currently focused expression, i.e. the actual focus of the zipper, and a direction. The direction indicates where the focus will be moved in the next state; up ↗, or down ↘ the parse tree of the expression. We will refer to foci using ϕ_i.

In this simple variant of our model language we have the following singular expression contexts:

$$\langle bop \rangle ::= ; \mid :=$$
$$K_S ::= \square\ \langle bop \rangle\ v_2 \mid e_1\ \langle bop \rangle\ \square \mid \text{var}_a\ \square \mid \text{load}\ \square$$

Figure 25: Basic Singular Expression Contexts

Most of the basic singular expression contexts are already familiar from Section 4.2, but we have added load □, which is used to provide the context for the subexpression e in **load** e expressions.

The stack ρ, which is implicitly contained in k, consists not of values, but of pointers to values on the heap, which will simplify capturing local variables by closures, as all captured variables will consist of references to heap-allocated variables. To prevent closures from referring to non-existing variables, variables are not explicitly deallocated when they go out of scope; a garbage collector is assumed to prevent heap pollution. We will refer to the stack using DeBruijn-indices [14] instead of names, meaning that instead of variable names we shall use x_i which will refer to the i^th variable-adress on the stack; this means we do not need special handling of overlapping variable names in different scopes.

The implicit availability of ρ in k is made possible by the fact that there are singular expression contexts marking every variable allocation. To illustrate this, we look at the example in listing Listing 27:

```
f(x: Int) : Unit
{
    var y: Int;
    y := 5
}
f(6)
```

Listing 27: Stack Example

The example program which we use to illustrate the stack implicitly contained in the context, shows a simple function definition and its call. This is one extension past the simple expression language, but it allows for a more effective illustration. The context of interest will be the one where `y:=5` is the focussed statement, in the execution of the function body; we provide this context in Listing 28.

$$\mathbf{k} = \text{var}_{a_y}\ \square$$
$$:: \text{funCall}\ cc\ \langle 6 \rangle$$
$$:: \text{var}_{a_f}\ \square$$
$$:: []$$

Listing 28: The Context of `y:=5` in Listing 27

To represent the context of a function call, we have introduced the additional singular expression context funCall $ck\ \overrightarrow{a}$,

which we shall examine in detail in Section 4.4; for now it is important to know that this context contains adresses of the evaluated parameters to the function in the form of $\overrightarrow{a}$.

The stack at the time of the assignment consists of the local variable y, the parameters to the function and the variable f, which is part of the lexical closure of the function. These values are contained in the program context, in the singular var_{a_y}, funCall $cc\ \langle 6\rangle$ and var_{a_f} contexts respectively. In the general case, the concatenation of the addresses, in singular variable contexts, and the parameters, in singular function call contexts, will provide the stack, in this fashion. We therefore define a $getStack$ function in Listing 29, which provides us the stack. We treat this function as an implicit conversion where needed and extend this function as more singular expression contexts are defined.

$$
\begin{aligned}
\text{getStack}(\square\ \langle bop\rangle\ e_2 :: k) &:= \text{getStack}\, k \\
\text{getStack}(v_1\ \langle bop\rangle\ \square :: k) &:= \text{getStack}\, k \\
\text{getStack}(\text{var}_a\ \square :: k) &:= a :: \text{getStack}\, k \\
\text{getStack}(\text{funCall}\ \overrightarrow{v}\ \square :: k) &:= \overrightarrow{v} +\!\!+ \text{getStack}\, k
\end{aligned}
$$

Listing 29: A Function to Derive the Stack from the Context.

4.3.3 Semantics: Reduction Rules

With the runtime structure defined in the previous section, we can now define the operational semantics for the simple expression language with the following reduction rules:

For the skip expression:

$$\mathbf{S}(k, (\searrow \mathbf{skip}), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow \mathbf{unit}), h) \qquad \textbf{red\_skip}$$

For subexpression reduction in binary expressions:

$$\mathbf{S}(k, (\searrow e_1\ \langle bop\rangle\ e_2), h) \twoheadrightarrow \mathbf{S}(\square\ \langle bop\rangle\ e_2 :: k, (\searrow e_1), h) \qquad \textbf{red\_binSubLeft}$$

$$\mathbf{S}(\square\ \langle bop\rangle\ e_2 :: k, (\nearrow v_1), h) \twoheadrightarrow \mathbf{S}(v_1\ \langle bop\rangle\ \square :: k, (\searrow e_2), h) \qquad \textbf{red\_binSubRight}$$

$$\mathbf{S}(k, (\searrow e_1\ \langle bop\rangle\ v_2), h) \twoheadrightarrow \mathbf{S}(\square\ \langle bop\rangle\ v_2 :: k, (\searrow e_1), h) \qquad \textbf{red\_binSubLeftVal}$$

$$\mathbf{S}(k, (\searrow v_1\ \langle bop\rangle\ e_2), h) \twoheadrightarrow \mathbf{S}(v_1\ \langle bop\rangle\ \square :: k, (\searrow e_2), h) \qquad \textbf{red\_binSubRightVal}$$

For binary expressions:

$$\mathbf{S}(\square; v_2 :: k, (\nearrow v_1), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v_2), h) \qquad \textbf{red\_seqLeft}$$

$$\mathbf{S}(v_1; \square :: k, (\nearrow v_2), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v_2), h) \qquad \textbf{red\_seqRight}$$

$$\mathbf{S}(\square := v :: k, (\nearrow \mathbf{ref}\, a), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow \mathbf{unit}), h[a \mapsto v]) \qquad \textbf{red\_assignLeft}$$

$$\mathbf{S}(\mathbf{ref}\, a := \square :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow \mathbf{unit}), h[a \mapsto v]) \qquad \textbf{red\_assignRight}$$

For variable declaration blocks:

$$\mathbf{S}(k, (\searrow (\mathbf{var}\, t;\ e)), h) \twoheadrightarrow \mathbf{S}(\text{var}_a\ \square :: k, (\searrow e), h[a \mapsto \bot]) \qquad \textbf{red\_varBlock}$$

for any a where $a \notin \text{dom}(h)$

$$\mathbf{S}(\text{var}_a\ \square :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v), h) \qquad \textbf{red\_varBlockUp}$$

$$\mathbf{S}(k, (\searrow \mathbf{var}\, t;\ v), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v), h) \qquad \textbf{red\_varBlockVal}$$

For variable lookup:

$$\mathbf{S}(k, (\searrow \mathbf{load}\, e), h) \twoheadrightarrow \mathbf{S}(\text{load}\ \square :: k, (\searrow e), h) \qquad \textbf{red\_loadSub}$$

$$\mathbf{S}(\text{load}\ \square :: k, (\nearrow \mathbf{ref}\, a), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v), h) \qquad \textbf{red\_load}$$

for any v where $\text{snd}\, h(a) = v$

Listing 30: Basic Reduction Rules

The basic reduction rules have the following meaning:

- **red_binSubLeft** evaluates a binary expression one step by marking the first subexpression for evaluation using ↘ and prepending a singular binary expression context containing the second subexpression and with a hole □ for the first.
- **red_binSubRight** marks the second subexpression of a binary expression for execution, when it encounters the resulting value of the first subexpression combined with a singular binary expression context. It swaps the encountered singular binary expression context with a hole for the first subexpression, with one for the second subexpression.
- **red_binSubLeftVal** marks the second subexpression of a binary expression for execution, in case the first one is already a value, skipping the evaluation of the first subexpression.
- **red_binSubRightVal** finishes up the subexpression reduction of a binary expression, when it encounters a value for the second subexpression and the resulting value of the first subexpression in a singular binary expression context, by removing the singular binary expression context and marking the binary expression for execution, now with values instead of expressions.
- **red_seqLeft** and **red_seqRight** evaluate a sequential expression, by discarding the value resulting from the first subexpression and propagating the value of the second subexpression.
- **red_assignLeft** and **red_assignRight** assign value v to address a in the heap. The value of an assignment expression is always **unit**.
- **red_varBlock** declares a local variable by prepending a singular var context, with a the address of the variable, which implicitely pushes a on the stack. The variable is uninitialized.
- **red_varBlockUp** removes a singular var context, consequently removing the variable from the stack. It does not deallocate the variable from the heap, as it could have been captured by a function closure. The value of a block is the value of its expression.
- **red_varBlockVal** removes a singular var context, consequently removing the variable from the stack. It does not deallocate the variable from the heap, as it could have been captured by a function closure. The value of a block is the value of its expression.
- **red_loadSub** evaluates the subexpression in a load expression.
- **red_load** evaluates a load expression from address a to the value stored in the heap at that address.

4.3.4 Reduction of a Simple Program

Since we have defined the semantics formally, let us run through the reduction of a simple program to see them in action:

```
var x : Int;
x := 5;
x := 3
```

Listing 31: Simple Program for Reduction Example

While we have technically defined a syntax without variable names, in the example listings we shall still write the familiar notation with variable names, as this is clearer to read.

$$\mathbf{k_1} = []$$
$$\boldsymbol{\phi_1} = \searrow \textbf{var}\ \text{Int}; x_0 := \textbf{int}\ 5; x_0 := \textbf{int}\ 3$$
$$\mathbf{h_1} = \{\}$$
$$\mathbf{S_1} = \mathbf{S}(k_1, \phi_1, h_1)$$

Figure 26: The Initial State in the Reduction of Listing 31

The initial state of the reduction of Listing 31 is given by Fig. 26: The context k_1 consists of an empty list, as we are at the top expression, and the heap h_1 starts empty as well. The focused expression ϕ_1 is a variable block, which will declare the local variable x for the expression $x_0 := \textbf{int}\ 5; x_0 := \textbf{int}\ 3$, which means the **red_varBlock** rule applies – resulting in the state of Fig. 27.

$$\mathbf{k_2} = \text{var}_{a_x}\ \square :: []$$
$$\boldsymbol{\phi_2} = \searrow x_0 := \textbf{int}\ 5; x_0 := \textbf{int}\ 3$$
$$\mathbf{h_2} = \{a_x \mapsto \bot\}$$
$$\mathbf{S_2} = \mathbf{S}(k_2, \phi_2, h_2)$$

Figure 27: The Second State in the Reduction of Listing 31

After applying **red_varBlock**, we have reached the second state of the reduction: A singular var context with the address of x has been appended to the context, making the address of x the zeroeth of the stack. The variable x has been defined and now points to an empty location in the heap. The focus is now on the sequence subexpression in the scope of the block, which has been marked for execution using $\searrow$. This means the **red_binSubLeft** rule applies – resulting in the state of Fig. 28.

$$\mathbf{k_3} = \square; x_0 := \textbf{int}\ 3 :: \text{var}_{a_x}\ \square :: []$$
$$\boldsymbol{\phi_3} = \searrow x_0 := \textbf{int}\ 5$$
$$\mathbf{h_3} = \{a_x \mapsto \bot\}$$
$$\mathbf{S_3} = \mathbf{S}(k_3, \phi_3, h_3)$$

Figure 28: The Third State in the Reduction of Listing 31

In the third state, **red_binExp** has been applied, which resulted in a singular sequence context being appended to the context, which signifies that we are executing the first subexpression of the sequence. The focused statement consists of this first subexpression, which is the assignment $x_0 := \textbf{int}\ 5$, marked for execution using $\searrow$. While the right hand side of the assignment expression is already a value, the left hand side still requires another reduction, so we apply **red_binSubLeftVal** – resulting in the state of Fig. 29.

$$
\begin{aligned}
\mathbf{k_4} &= \square := \mathbf{int}\,5 \\
&\quad :: \square; x_0 := \mathbf{int}\,3 \\
&\quad :: \mathrm{var}_{a_x}\ \square :: [] \\
\boldsymbol{\phi_4} &= \searrow x_0 \\
\mathbf{h_4} &= \{a_x \mapsto \bot\} \\
\mathbf{S_4} &= \mathbf{S}(k_4, \phi_4, h_4)
\end{aligned}
$$

Figure 29: The Fourth State in the Reduction of Listing 31

In the fourth state, **red_binSubLeftVal** has been applied and the variable corresponding to the first element of the stack evaluates to the address of the variable x. Since the first subexpression has been reduced and the second is already a value, we can now apply **red_assignLeft** – resulting in the state of Fig. 30.

$$
\begin{aligned}
\mathbf{k_5} &= \square; x_0 := \mathbf{int}\,3 \\
&\quad :: \mathrm{var}_{a_x}\ \square :: [] \\
\boldsymbol{\phi_5} &= \nearrow \mathbf{unit} \\
\mathbf{h_5} &= \{a_x \mapsto 5\} \\
\mathbf{S_5} &= \mathbf{S}(k_5, \phi_5, h_5)
\end{aligned}
$$

Figure 30: The Fifth State in the Reduction of Listing 31

In the fifth state, the application of **red_assignLeft** has resulted in the heap changing, to reflect the assignment of the integer value 5 to x, and the direction of the focused statement changing to $\nearrow$. This direction combined with the singular sequential context, with the $\square$ on the left side, at the head of the context, means the **red_binSubRight** rule now applies – resulting in the state of Fig. 31.

$$
\begin{aligned}
\mathbf{k_6} &= \mathbf{unit}; \square \\
&\quad :: \mathrm{var}_{a_x}\ \square :: [] \\
\boldsymbol{\phi_6} &= \searrow x_0 := \mathbf{int}\,3 \\
\mathbf{h_6} &= \{a_x \mapsto 5\} \\
\mathbf{S_6} &= \mathbf{S}(k_6, \phi_6, h_6)
\end{aligned}
$$

Figure 31: The Sixth State in the Reduction of Listing 31

In the sixth state, **red_binSubRight** has been applied, which resulted in the execution shifting to the second subexpression of the sequence. The previous singular sequential context has been replaced by a new one, which signifies the execution of the second subexpression. The focus is subsequently also on this subexpression, marked with $\searrow$. Since the subexpression is once again an assignment, we apply the same rules as before and fast forward to the $8^t h$ state in Fig. 32, where the assignment has finished.

$$\mathbf{k_8} = \mathbf{unit}; \square \\ :: \text{var}_{a_x}\ \square :: [] \\ \boldsymbol{\phi_8} = \nearrow \mathbf{unit} \\ \mathbf{h_8} = \{a_x \mapsto 3\} \\ \mathbf{S_8} = \mathbf{S}(k_8, \phi_8, h_8)$$

Figure 32: The 8^{th} State in the Reduction of Listing 31

In the 8^{th} state, **red_assignLeft** has been applied in the same fashion as in the fifth state, with similar results: The heap now reflects the integer value 3 for x and the value has been evaluated to **unit**. However, as this time we evaluated the second subexpression of a sequence, the **red_seqLeft** rule applies instead of **red_binSubRight** – resulting in the state of Fig. 33.

$$\mathbf{k_9} = \text{var}_{a_x}\ \square :: [] \\ \boldsymbol{\phi_9} = \nearrow \mathbf{unit} \\ \mathbf{h_9} = \{a_x \mapsto 3\} \\ \mathbf{S_9} = \mathbf{S}(k_9, \phi_9, h_9)$$

Figure 33: The 9^th State in the Reduction of Listing 31

In the 9^{th} state, after applying **red_seqLeft**, the singular sequential context has been removed from the context and the focused statement now consists the value of the sequential expression, **unit**, marked with ↗. This ↗, combined with the singular var context, means it is time to pop a_x of the stack by applying **red_varBlockUp** – resulting in the state of Fig. 34.

$$\mathbf{k_10} = [] \\ \boldsymbol{\phi_10} = \nearrow \mathbf{unit} \\ \mathbf{h_10} = \{a_x \mapsto 3\} \\ \mathbf{S_10} = \mathbf{S}(k_10, \phi_10, h_10)$$

Figure 34: The 10^{th} State in the Reduction of Listing 31

In the 10^{th} and final state, there are no more possible reductions and we have reduced the top expression to a value, meaning the program has successfully executed. As we are at the top expression, the context is once again empty.

4.4 Extending *Scala Core* with Functions

One of the more interesting extentions of our model language – and a primary focus – is the support for first-class functions and lexical closures. In this section we first give, and explain, the syntax, changes to the runtime structure and the necessary reduction rules, to incorporate functions in our semantics, and we provide an example reduction, using these rules.

4.4.1 Syntax

$$
\begin{aligned}
\text{Type} &::= \ldots \mid \text{FType} \\
\text{FType} &::= (\overrightarrow{\text{Type}}) : \text{Type} \\
\text{Expression} &::= \ldots \mid \text{FType} \rightarrow \{\text{Expression}\} \mid \textbf{call}\ \text{Expression}
\end{aligned}
$$

Figure 35: Syntactical Extensions for Functions

Extending our model language with functions requires an additional type for functions and, furthermore, two new expressions:

- Lambda $(\overrightarrow{t}) : t \rightarrow e$, which defines a function, with a list of parameters with types $\overrightarrow{t}$, return type t and function body e.
- Call **call** e, which calls a function.

There is no specific function value, as a function will consist of a pointer to a function store in the heap.

4.4.2 Semantics : Runtime Structure

$$
\begin{aligned}
\text{HeapElement} &= \text{Value} \mid \text{FunctionStore} \\
\text{FunctionStore} &= \text{Context} \times \text{Expression}
\end{aligned}
$$

Figure 36: Extended Runtime Structure for Functions

To allow functions to be stored on the heap, we have added the function store as a possible heap element. A function store consists of the defining context of a function, i.e. a list of singular expression contexts, and an expression, which is the body of the function.

Furthermore, we have added additional singular expression contexts for function expressions:

$$
K_S ::= \ldots \mid \text{call}\ \square\ \overrightarrow{e} \mid \text{call}\ v\ \square \mid \text{params}\ \overrightarrow{v}\ \overrightarrow{e}\ \square \mid \text{funCall}\ k\ \overrightarrow{a}\ \square
$$

Figure 37: The Singular Expression Contexts Expanded with Function Contexts

These additional contexts have the following meaning:

- The singular call contexts call $\square\ \overrightarrow{e}$ and call $e\ \square$ mark the currently focused subexpression as the called expression or the parameters of a call, respectively.
- The singular parameters context params $\overrightarrow{e}\ \overrightarrow{v}\square$ marks the currently focused subexpression as a member of a list of function parameters, with $\overrightarrow{v}$ the values of the evaluated parameters and $\overrightarrow{e}$ the remaining (unevaluated) subexpressions.
- The singular function call funCall $k\ \overrightarrow{a}$ context marks the currently focused subexpression as the body of an executing function, with k the calling context of the function and $\overrightarrow{a}$ the addresses of the parameters.

4.4.3 Semantics : Reduction Rules

To evaluate functions, we introduce the following reduction rules:

For function definition:

$\mathbf{S}(k, (\searrow (\overrightarrow{t}) : t \to e), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow \mathbf{ref}\, a), h[a \mapsto \langle k, e\rangle])$ **red_funDef**

for any a where $a \notin \mathrm{dom}(h)$

For subexpression reduction in function calls:

$\mathbf{S}(k, (\searrow \mathbf{call}\, e\ \overrightarrow{e}), h) \twoheadrightarrow \mathbf{S}(\mathrm{call}\ \square\ rev(\overrightarrow{e}) :: k, (\searrow e), h)$ **red_callLeft**

$\mathbf{S}(\mathrm{call}\ \square\ \overrightarrow{e} :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(\mathrm{call}\ v\ \square :: k, (\searrow \overrightarrow{e}), h)$ **red_callRight**

For function parameters:

$\mathbf{S}(\mathrm{call}\ v\ \square :: k, (\searrow []), h) \twoheadrightarrow \mathbf{S}(\mathrm{call}\ v\ \square :: k, (\nearrow []), h)$ **red_paramsEmpty**

$\mathbf{S}(\mathrm{call}\ v\ \square :: k, (\searrow e :: \overrightarrow{e}), h)$

$\twoheadrightarrow \mathbf{S}(\mathrm{params}[]\ \overrightarrow{e}\square :: \mathrm{call}\ v\ \square :: k, (\searrow e), h)$ **red_paramsHead**

$\mathbf{S}(\mathrm{call}\ v\ \square :: k, (\searrow v :: \overrightarrow{e}), h)$

$\twoheadrightarrow (\mathbf{S}(\mathrm{params}\ v :: []\ \overrightarrow{e}\square) :: \mathrm{call}\ v\ \square :: k, (\nearrow v), h)$ **red_paramsHeadVal**

$\mathbf{S}((\mathrm{params}\ \overrightarrow{v}\ e :: \overrightarrow{e}\ \square) :: k, (\nearrow v), h)$

$\twoheadrightarrow \mathbf{S}((\mathrm{params}\ v :: \overrightarrow{v}\ \overrightarrow{e}\ \square) :: k, (\searrow e), h)$ **red_paramsNext**

$\mathbf{S}((\mathrm{params}\ \overrightarrow{v}\ v_2 :: \overrightarrow{e}\ \square) :: k, (\nearrow v_1), h)$

$\twoheadrightarrow \mathbf{S}((\mathrm{params}\ v_1 :: \overrightarrow{v}\ \overrightarrow{e}\ \square) :: k, (\nearrow v_2), h)$ **red_paramsNextVal**

$\mathbf{S}((\mathrm{params}\ \overrightarrow{v}\ []\ \square) :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v :: \overrightarrow{v}), h)$ **red_paramsLast**

For function calls:

$\mathbf{S}(\mathrm{call}\ \mathbf{ref}\, a\square :: k_1, (\nearrow \overrightarrow{v}), h_1) \twoheadrightarrow \mathbf{S}(\mathrm{funCall}\ k_1\ \overrightarrow{a}\ \square :: k_2, (\searrow e), h_2)$ **red_call**

For any $\overrightarrow{a}, h_2, k_2, e$ where $\texttt{allocparams}\ h_1\ \overrightarrow{v}\ \overrightarrow{a}\ h_2$,

$\mathrm{snd}\ h_1(a) = \langle k_2, e\rangle$

For function returns:

$\mathbf{S}(\mathrm{funCall}\ k_2\ \overrightarrow{a}\ \square :: k_1, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k_2, (\nearrow v), h)$ **red_return**

Figure 38: Reduction rules for functions

These rules have the following meaning:

- **red_funDef** allocates the function store corresponding to the definition to the heap and returns the address.
- **red_callLeft** and **red_callRight** start the reduction of the subexpression e and the list of subexpressions $\overrightarrow{e}$ respectively, when a function call is encountered.
- The params rules reduce the subexpressions in the list of function parameters to values.
- **red_call** starts the evaluation, of the function with store $\langle k_2, e\rangle$ at address a and list of parameters $\overrightarrow{v}$, by prepending the singular funCall $k\ \overrightarrow{a}$ context, to the defining context k_2 of the function, and starting the execution of the body in that context. The function parameter values are allocated to the heap using `allocParams` – shown in Fig. 39.
- **red_return** Propagates the return value v to the calling context k_2, when a funCall $k_2\ \overrightarrow{a}$ context is encountered in combination with an evaluated value v.

$$\frac{}{\texttt{allocParams}\ h\ []\ []\ h}$$

$$\frac{\texttt{allocParams}\ h_1\ \vec{a}\ \vec{v}\ h_2 \qquad a \notin \mathrm{dom}(h_1)}{\texttt{allocParams}\ h_1\ (a :: \vec{a})\ (v :: \vec{v})\ h_2 \qquad h_2[a \mapsto v]}$$

Figure 39: The Inductively Defined `allocParams` Relation

4.4.4 Reduction of a Simple Program with Functions

With these reduction rules we can now evaluate the program in Listing 32. Compared to the previous reduction of the simple example, the margins now contain pointers to retain an intuitive idea of where the execution of the program is at, in the sea of reduction steps.

```
var f : (Int):Int;
f = (x : Int):Int ->
{
    x
};
f(5)
```

Listing 32: A Small Program with a Function

Declaration of f

$$\begin{aligned}
\mathbf{k_1} &= [] \\
\boldsymbol{\phi_1} &= \searrow \mathbf{var}(\mathrm{Int}) : Int; (x_0 := (Int) : Int \to \mathbf{load}\, x_1); \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\ 5 :: []) \\
\mathbf{h_1} &= \{\} \\
\mathbf{S_1} &= \mathbf{S}(k_1, \phi_1, h_1)
\end{aligned}$$

Figure 40: The Initial State in the Reduction of Listing 32

The initial state of the reduction of Listing 32 is given by Fig. 40: The context and the heap are empty, as we are at the top expression, which is a variable declaration. We apply the **red_varBlock** rule to proceed to the state in Fig. 41.

f declared

$$\begin{aligned}
\mathbf{k_2} &= \mathrm{var}_{a_f}\ \square \\
&\quad :: [] \\
\boldsymbol{\phi_2} &= \searrow (x_0 := (Int) : Int \to \mathbf{load}\, x_1); \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\ 5 :: []) \\
\mathbf{h_2} &= \{a_f \mapsto \bot\} \\
\mathbf{S_2} &= \mathbf{S}(k_2, \phi_2, h_2)
\end{aligned}$$

Figure 41: The Second State in the Reduction of Listing 32

In the second state, **red_varBlock** has been applied, which resulted in the function f being declared and a singular var context containing its address being prepended to the context. The subexpression of the function block is a sequential expression, so we apply **red_binSubLeft** to reduce the first subexpression – which results in de state shown in Fig. 42.

Definition of f

$$
\begin{aligned}
\mathbf{k_3} &= \square; \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
&\quad :: \mathrm{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_3} &= \searrow\, x_0 := ((Int) : Int \to \mathbf{load}\, x_1) \\
\mathbf{h_3} &= \{a_f \mapsto \bot\} \\
\mathbf{S_3} &= \mathbf{S}(k_3, \phi_3, h_3)
\end{aligned}
$$

Figure 42: The Third State in the Reduction of Listing 32

In the third state, **red_binSubLeft** has been applied. The first subexpression is an assignment with reducible subexpressions, so we once again apply **red_binSubLeft** – which results in the state shown in Fig. 43.

$$
\begin{aligned}
\mathbf{k_4} &= \square := ((Int) : Int \to \mathbf{load}\, x_1) \\
&\quad :: \square; \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
&\quad :: \mathrm{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_4} &= \nearrow\, \mathbf{ref}\, a_f \\
\mathbf{h_4} &= \{a_f \mapsto \bot\} \\
\mathbf{S_4} &= \mathbf{S}(k_4, \phi_4, h_4)
\end{aligned}
$$

Figure 43: The Fourth State in the Reduction of Listing 32

In the fourth state, **red_binSubLeft** has reduced the first subexpression of the assignment to a value, which means we can apply **red_binSubRight** to reduce the second – resulting in the state of Fig. 44.

$$
\begin{aligned}
\mathbf{k_5} &= \mathbf{ref}\, a_f := \square \\
&\quad :: \square; \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
&\quad :: \mathrm{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_5} &= \searrow\, (Int) : Int \to \mathbf{load}\, x_1 \\
\mathbf{h_5} &= \{a_f \mapsto \bot\} \\
\mathbf{S_5} &= \mathbf{S}(k_5, \phi_5, h_5)
\end{aligned}
$$

Figure 44: The Fifth State in the Reduction of Listing 32

In the fifth state, **red_binSubRight** has been applied and the focused expression is now a function definition expression. We apply **red_funDef** to reduce this expression to a value – resulting in the state of Fig. 45.

$$
\begin{aligned}
\mathbf{k_6} &= \mathbf{ref}\, a_f := \square \\
&\quad :: \square; \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
&\quad :: \mathrm{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_6} &= \nearrow \mathbf{ref}\, a_{f_1} \\
\mathbf{h_6} &= \{a_f \mapsto \bot, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle\} \\
\mathbf{S_6} &= \mathbf{S}(k_6, \phi_6, h_6)
\end{aligned}
$$

Figure 45: The Sixth State in the Reduction of Listing 32

In the sixth state, **red_funDef** has allocated the function at address a_{f_1} and returns the address. We now apply **red_assignRight** to finish the actual assignment – resulting in the state of Fig. 46.

f defined

$$
\begin{aligned}
\mathbf{k_7} &= \square; \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
&\quad :: \mathrm{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_7} &= \nearrow \mathbf{unit} \\
\mathbf{h_7} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle\} \\
\mathbf{S_7} &= \mathbf{S}(k_7, \phi_7, h_7)
\end{aligned}
$$

Figure 46: The Seventh State in the Reduction of Listing 32

In the seventh state, **red_assignRight** has been applied, resulting in the address a_f pointing to the proper value. The return value of assignments is always **unit**. With this the first subexpression of the sequential expression has been fully reduced – i.e. the function f has been defined – so we apply **red_binSubRight** to start on the second one – resulting in the state of Fig. 47.

Starting call

$$
\begin{aligned}
\mathbf{k_8} &= \mathbf{unit}; \square \\
&\quad :: \mathrm{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_8} &= \searrow \mathbf{call}(\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
\mathbf{h_8} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle\} \\
\mathbf{S_8} &= \mathbf{S}(k_8, \phi_8, h_8)
\end{aligned}
$$

Figure 47: The Eighth State in the Reduction of Listing 32

In the eighth state, **red_binSubRight** has marked the second subexpression for evaluation, which is a function call expression. As this call has yet unreduced subexpressions, we apply **red_callLeft** to reduce the first subexpression – resulting in the state of Fig. 48.

$$
\begin{aligned}
\mathbf{k_9} &= \text{call}\,\square(Int5 :: []) \\
&\quad :: \mathbf{unit};\square \\
&\quad :: \text{var}_{a_f}\,\square \\
&\quad :: [] \\
\boldsymbol{\phi_9} &= \searrow \mathbf{load}\, x_0 \\
\mathbf{h_9} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1\rangle\} \\
\mathbf{S_9} &= \mathbf{S}(k_9, \phi_9, h_9)
\end{aligned}
$$

Figure 48: The 9^{th} State in the Reduction of Listing 32

In the 9^{th} state, **red_callLeft** has been applied, resulting in a load expression being marked for execution. To load a value from the heap, we first reduce the expression e in the load expression to an address using **red_loadSub** – resulting in the state of Fig. 49.

$$
\begin{aligned}
\mathbf{k_1 0} &= \text{load}\,\square \\
&\quad :: \text{call}\,\square(Int5 :: []) \\
&\quad :: \mathbf{unit};\square \\
&\quad :: \text{var}_{a_f}\,\square \\
&\quad :: [] \\
\boldsymbol{\phi_1 0} &= \searrow x_0 \\
\mathbf{h_1 0} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1\rangle\} \\
\mathbf{S_1 0} &= \mathbf{S}(k_1 0, \phi_1 0, h_1 0)
\end{aligned}
$$

Figure 49: The 10^{th} State in the Reduction of Listing 32

In the 10^{th} state, applying **red_loadSub** resulted in an ident expression, which means we can now apply **red_load** to load the actual value – resulting in the state of Fig. 50.

Function identified as f

$$
\begin{aligned}
\mathbf{k_1 1} &= \text{call}\,\square(\mathbf{int}\,5 :: []) \\
&\quad :: \mathbf{unit};\square \\
&\quad :: \text{var}_{a_f}\,\square \\
&\quad :: [] \\
\boldsymbol{\phi_1 1} &= \nearrow \mathbf{ref}\, a_{f_1} \\
\mathbf{h_1 1} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1\rangle\} \\
\mathbf{S_1 1} &= \mathbf{S}(k_1 1, \phi_1 1, h_1 1)
\end{aligned}
$$

Figure 50: The 11^{th} State in the Reduction of Listing 32

In the 11^{th} state, **red_load** has been applied with the address a_f, to obtain the value $\mathbf{ref}\, a_{f_1}$. As this concludes the reduction of the first subexpression of the function call expression, we now apply **red_callRight** to evaluate the function parameters – resulting in the state of Fig. 51.

Evaluating parameters

$$
\begin{aligned}
\mathbf{k_1 2} &= \text{call } \mathbf{ref}\, a_{f_1}\ x_1 \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{var}_{a_f}\ \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 2} &= \searrow \mathbf{int}\, 5 :: [] \\
\mathbf{h_1 2} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle\} \\
\mathbf{S_1 2} &= \mathbf{S}(k_1 2, \phi_1 2, h_1 2)
\end{aligned}
$$

Figure 51: The 12^{th} State in the Reduction of Listing 32

In the 12^{th} state, after applying **red_callRight**, the focus is on a list of expressions, of which the first is already a value, in a function call context, which means the **red_paramsHeadVal** rule applies – resulting in the state of Fig. 52.

$$
\begin{aligned}
\mathbf{k_1 3} &= \text{params}(Int 5 :: []) \ [] \ \square \\
&\quad :: \text{call } \mathbf{ref}\, a_{f_1}\ x_1 \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{var}_{a_f}\ \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 3} &= \nearrow \mathbf{int}\, 5 \\
\mathbf{h_1 3} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle\} \\
\mathbf{S_1 3} &= \mathbf{S}(k_1 3, \phi_1 3, h_1 3)
\end{aligned}
$$

Figure 52: The 13^{th} State in the Reduction of Listing 32

In the 13^{th} state, after applying **red_paramsHeadVal**, the focused context is an evaluated value and the list of remaining expressions in the singular parameter context is empty, which means the **red_paramsLast** rule applies – resulting in the state of Fig. 53.

Parameters evaluated

$$
\begin{aligned}
\mathbf{k_1 3} &= \text{call } \mathbf{ref}\, a_{f_1}\ x_1 \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{var}_{a_f}\ \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 3} &= \nearrow (\mathbf{int}\, 5 :: []) \\
\mathbf{h_1 3} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle\} \\
\mathbf{S_1 3} &= \mathbf{S}(k_1 3, \phi_1 3, h_1 3)
\end{aligned}
$$

Figure 53: The 13^{th} State in the Reduction of Listing 32

In the 13^{th} state, **red_paramsLast** has changed the focus to a list of values, which means the function call expression is ready to be evaluated. We therefore apply **red_call** – resulting in the state of Fig. 54.

Evaluating call

$$
\begin{aligned}
\mathbf{k_1 4} &= \text{funCall}\, k_9\ (a_x :: []) \\
&\quad :: \mathbf{ref}\, a_f := \square \\
&\quad :: \square; (\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
&\quad :: \text{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 4} &= \searrow \mathbf{load}\, x_1 \\
\mathbf{h_1 4} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle, a_x \mapsto \mathbf{int}\, 5\} \\
\mathbf{S_1 4} &= \mathbf{S}(k_1 4, \phi_1 4, h_1 4)
\end{aligned}
$$

Figure 54: The 14^{th} State in the Reduction of Listing 32

In the 14^{th} state, after applying **red_call**, the context is now the defining context of the function, prepended with , which contains the calling context and a list containing the address of the function parameter. The focused expression is the function body, which in this case consists of a load expression. The load expression reduces as before, so we skip forward to the 17^{th} state – shown in Fig. 55 – where the function body has been reduced.

$$
\begin{aligned}
\mathbf{k_1 7} &= \text{funCall}\, k_9\ (a_x :: []) \\
&\quad :: \mathbf{ref}\, a_f := \square \\
&\quad :: \square; (\mathbf{load}\, x_0)(\mathbf{int}\, 5 :: []) \\
&\quad :: \text{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 7} &= \nearrow \mathbf{int}\, 5 \\
\mathbf{h_1 7} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle, a_x \mapsto \mathbf{int}\, 5\} \\
\mathbf{S_1 7} &= \mathbf{S}(k_1 7, \phi_1 7, h_1 7)
\end{aligned}
$$

Figure 55: The 17^{th} State in the Reduction of Listing 32

In the 17^{th} state, after applying **red_load**, the function has finished executing and we now apply **red_return** to propagate the return value back to the calling context – which results in the state of Fig. 56.

Call returned

$$
\begin{aligned}
\mathbf{k_1 8} &= \mathbf{unit}; \square \\
&\quad :: \text{var}_{a_f}\, \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 8} &= \nearrow \mathbf{int}\, 5 \\
\mathbf{h_1 8} &= \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle, a_x \mapsto \mathbf{int}\, 5\} \\
\mathbf{S_1 8} &= \mathbf{S}(k_1 8, \phi_1 8, h_1 8)
\end{aligned}
$$

Figure 56: The 18^{th} State in the Reduction of Listing 32

In the 18^{th} state, **red_return** has been applied and the value returned has now taken the place of the function call, back in the original calling context. What remains is applying **red_seq** to finish up the sequential expression and then **red_funBlockUp** to clear the stack – which results in the final state of Fig. 57.

$$\mathbf{k_1 9} = []$$
$$\boldsymbol{\phi_1 9} = \nearrow \mathbf{int}\, 5$$
$$\mathbf{h_1 9} = \{a_f \mapsto a_{f_1}, a_{f_1} \mapsto \langle k_7, \mathbf{load}\, x_1 \rangle, a_x \mapsto \mathbf{int}\, 5\}$$
$$\mathbf{S_1 9} = \mathbf{S}(k_1 9, \phi_1 9, h_1 9)$$

Figure 57: The 23^{rd} State in the Reduction of Listing 32

4.5 Extending *Scala Core* with Exceptions

Another feature of our model language, is the support for exceptions. In this section we first give, and explain, the necessary reduction rules, to incorporate exceptions in our semantics, and, we provide an example reduction, using these rules.

4.5.1 Syntax

For exceptions we extend the syntax with two expressions:

Expression ::= **try** {Expression} **catch**(Type) {Expression} | **throw** Expression

Figure 58: Syntactical Extensions for Exceptions

- Try/catch **try** $\{e_1\}$ **catch**(t) $\{e_2\}$, which evaluates e_1 either succesfully, or until an exception of type t is encountered, in which case it evaluates e_2.
- Throw **throw** e, which evaluates e and then propagates the resulting value up the expression tree until either a matching try/catch is found, or the topmost expression has been reached and the program terminates abnormally.

4.5.2 Semantics: Runtime Structure

To express the semantics of exceptions, we have added a traversal direction – shown in Fig. 59 – and a number of additional singular expression contexts – shown in Fig. 60 – and we have defined the neccesary reduction rules – shown in Fig. 61.

$Direction$ ::= $\nearrow | \searrow | \lightning_t$

Figure 59: An Additional Direction

The additional $\lightning_t$ direction is used to propagate exceptions upwards in the expression, moving the focus up one step at a time, until a singular trycatch t' e $\square$ context is encountered where $t = t'$.

K_S ::= ... | trycatch t e $\square$ | throw $\square$

Figure 60: The Singular Expression Contexts Expanded with Function Contexts

These additional contexts have the following meaning:

- The singular trycatch context trycatch $t\ e\ \square$ marks the currently focused subexpression as the try-expression in a try/catch expression, with e the expression to execute on catch and t the type of expression to catch.
- The singular throw context throw $\square$ marks the currently focused subexpression as the eventual value to be thrown in a throw expression.

4.5.3 Semantics: Reduction Rules

For try/catch expressions:

$\mathbf{S}(k, (\searrow \textbf{try}\ e_1\ \textbf{catch}\ t\ e_2), h) \twoheadrightarrow \mathbf{S}(\text{trycatch}\ t\ e_2\ \square :: k, (\searrow e_1), h)$ **red_try**

$\mathbf{S}(\text{trycatch}\ t\ e\ \square :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v), h)$ **red_tryNoEx**

For subexpression reduction in throw expressions:

$\mathbf{S}(k, (\searrow \textbf{throw}\ e), h) \twoheadrightarrow \mathbf{S}(\text{throw}\ \square :: k, (\searrow e), h)$ **red_throwSub**

For any t where $\texttt{typeOf}(e, t)$

For throwing exceptions:

$\mathbf{S}(\text{throw}\ \square :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t v), h)$ **red_throw**

For catching exceptions:

$\mathbf{S}(\text{trycatch}\ t\ e\ \square :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(\text{val}_a\ \square :: k, (\searrow e), h[a := v])$ **red_catch**

For any a where $h\ a = \bot$

For exception propagation:

$\mathbf{S}(\square\ \langle bop \rangle\ e_2 :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exBinSubLeft**

$\mathbf{S}(v_1\ \langle bop \rangle\ \square :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exBinSubRight**

$\mathbf{S}(\text{var}_a\ \square :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exBlock**

$\mathbf{S}(\text{load}\ \square :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exLoad**

$\mathbf{S}(\text{call}\ \square\ \overrightarrow{e} :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exCallLeft**

$\mathbf{S}(\text{call}\ v\ \square :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exCallRight**

$\mathbf{S}(\text{params}\ \overrightarrow{v}\ \overrightarrow{e}\ \square :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exParams**

$\mathbf{S}(\text{funCall}\ ck\ \overrightarrow{a} :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(ck, (\lightning_t\ v), h)$ **red_exCall**

$\mathbf{S}(\text{throw}\ \square :: k, (\lightning_t\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_t\ v), h)$ **red_exThrow**

$\mathbf{S}(\text{trycatch}\ t_1\ e\ \square :: k, (\lightning_{t_2}\ v), h) \twoheadrightarrow \mathbf{S}(k, (\lightning_{t_2}\ v), h)$ **red_exTry**

Where $t_1 \neq t_2$

Figure 61: Reduction rules for Exceptions

These additional reduction rules have the following meaning:

- **red_try** takes a try/catch expression, with the expression to try e_1, the type to catch t and the expression to evaluate on a catch e_2, and reduces it to the execution of e_1 in a singular trycatch context containing e_2 and t.
- **red_tryNoEx** is applied when a try expression has evaluated without throwing. The value v resulting from the try expression is propagated as the result of the entire try/catch expression. The singular trycatch context is removed.
- **red_throw** is applied after **red_throwSub** has resulted in a value v, of type t, to throw. The direction is changed to $\lightning_t$, and the focused expression is the value v.

- **red_catch** is applied when, during exception propagation – signified by the direction $\lightning_t$ and a value v, we encounter a singular trycatch context with type t. The exception is now considered caught and the catch expression e_2 is evaluated in a singular variable block context containing the address to the thrown value, which has been allocated on the heap.
- The **red_ex** rules are generally straightforward, as they simply propagate the exception one step up the context. The outlier is **red_exCall**, which, instead of moving the exception up one step in the current context, propagates it to the calling context of an executing function.

4.5.4 Example Program with Exceptions

With these reduction rules we can now evaluate the program in Listing 33:

```
var x : Int;
x := 6;
try
{
    throw x;
}
catch(y : Int)
{
    x := 2
}
```

Listing 33: A Small Program with Exception Handling

$$
\begin{aligned}
\mathbf{k_7} &= \mathbf{unit}; \square \\
&\quad :: \mathrm{var}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi_7} &= \searrow \mathbf{try}(\mathbf{throw}\,\mathbf{load}\,x_0)\,\mathbf{catch}\,\mathrm{Int}(x_0 := \mathbf{int}\,2) \\
\mathbf{h_7} &= \{a_x \mapsto \mathbf{int}\,6\} \\
\mathbf{S_7} &= \mathbf{S}(k_7, \phi_7, h_7)
\end{aligned}
$$

Figure 62: The Seventh State in the Reduction of Listing 33

As variable declaration and assignment are by now quite familiar, we will begin this example in the seventh state, where the try/catch expression is marked for evaluation. In this state we apply **red_try** – resulting in the state of Fig. 63.

$\mathbf{k_8}$ = trycatch Int(x_0 := **int** 2) □
:: **unit**; □
:: var_{a_x}
:: []
$\boldsymbol{\phi_8}$ =↘ **throw load** x_0
$\mathbf{h_8}$ = $\{a_x \mapsto \textbf{int}\ 6\}$
$\mathbf{S_8}$ = $\mathbf{S}(k_8, \phi_8, h_8)$

Figure 63: The Eighth State in the Reduction of Listing 33

In the eighth state, **red_try** has been applied, resulting in a singular trycatch context being prepended to the context, with x_0 := **int** 2 the catch expression and Int for the type to catch. The focused expression is now the throw expression **throw load** x_0, with the direction ↘ marking it for execution. As the throw expression has a yet unreduced subexpression, we apply **red_throwSub** – resulting in the state of Fig. 64.

$\mathbf{k_9}$ = throw □
:: trycatch Int(x_0 := **int** 2) □
:: **unit**; □
:: var_{a_x}
:: []
$\boldsymbol{\phi_9}$ =↘ **load** x_0
$\mathbf{h_9}$ = $\{a_x \mapsto \textbf{int}\ 6\}$
$\mathbf{S_9}$ = $\mathbf{S}(k_9, \phi_9, h_9)$

Figure 64: The Ninth State in the Reduction of Listing 33

In the ninth state, after applying **red_throwSub**, the focused expression is the familiar load expression, so we fast forward to the 12^{th} state where it has been fully evaluated.

$\mathbf{k_{12}}$ = throw □
:: trycatch Int(x_0 := **int** 2) □
:: **unit**; □
:: var_{a_x}
:: []
$\boldsymbol{\phi_{12}}$ =↗ **int** 6
$\mathbf{h_{12}}$ = $\{a_x \mapsto \textbf{int}\ 6\}$
$\mathbf{S_{12}}$ = $\mathbf{S}(k_{12}, \phi_{12}, h_{12})$

Figure 65: The 12^{th} State in the Reduction of Listing 33

In the 12^{th} state, we encounter a value in a singular throw context, which means we can apply **red_throw**, to throw the value – resulting in the state of Fig. 66.

$$
\begin{aligned}
\mathbf{k_{13}} &= \text{trycatch Int}(x_0 := \textbf{int}\ 2)\ \square \\
&\quad :: \textbf{unit}; \square \\
&\quad :: \text{var}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi_{13}} &= \lightning_{\text{Int}}\ \textbf{int}\ 6 \\
\mathbf{h_{13}} &= \{a_x \mapsto \textbf{int}\ 6\} \\
\mathbf{S_{13}} &= \mathbf{S}(k_{13}, \phi_{13}, h_{13})
\end{aligned}
$$

Figure 66: The 13^{th} State in the Reduction of Listing 33

In the 13^{th} state, after applying **red_throw**, we encounter the value **int** 6 with direction $\lightning_{\text{Int}}$, in a singular trycatch context with matching type. This means we can apply **red_catch** to catch the value – resulting in the state of Fig. 67.

$$
\begin{aligned}
\mathbf{k_{14}} &= \text{var}_{a_y}\ \square \\
&\quad :: \textbf{unit}; \square \\
&\quad :: \text{var}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi_{14}} &= \searrow x_0 := \textbf{int}\ 2 \\
\mathbf{h_{14}} &= \{a_x \mapsto \textbf{int}\ 6, a_y \mapsto \textbf{int}\ 6\} \\
\mathbf{S_{14}} &= \mathbf{S}(k_{14}, \phi_{14}, h_{14})
\end{aligned}
$$

Figure 67: The 14^{th} State in the Reduction of Listing 33

In the 14^{th} state, **red_catch** has been applied and the assignment expression $x_0 := \textbf{int}\ 2$ is marked for execution. A singular var context has been prepended to the context, containing the address of the caught value. The assignment reduces as before and is not relevant to the treatment of exception handling, so we omit further reduction steps from this example.

4.6 Extending *Scala Core* with Classes & Traits

A large, but necessary, extension, is the addition of classes and traits, to make *Scala Core* into an object-oriented language, just as *Scala* itself. In this section we provide the necessary syntax and semantics for objects and demonstrate their use with an example reduction.

4.6.1 Syntax

To extend our model language with classes and traits, the syntax requires a substantial extension. First there is the syntax to define classes and traits:

$$
\begin{aligned}
\text{Class} &::= \textbf{class}\ \text{ClassIdentifier}\ \textbf{extends}\ \text{ClassIdentifier}\ \textbf{with}\ \overrightarrow{\text{TraitIdentifier}} \\
&\quad \{\overline{\text{Field}}\ \overline{\text{Method}}\} \\
\text{Trait} &::= \textbf{trait}\ \text{TraitIdentifier}\ \textbf{extends}\ \text{TraitIdentifier}\ \textbf{with}\ \overrightarrow{\text{TraitIdentifier}} \\
&\quad \{\overline{\text{Field}}\ \overline{\text{Method}}\} \\
\text{Method} &::= \text{MethodIdentifier}(\overrightarrow{\text{Type}}) : \text{Type}\ \{\text{Expression}\} \\
\text{Field} &::= \text{MutableField} \mid \text{ImmutableField} \\
\text{MutableField} &::= \textbf{var}\ \text{FieldIdentifier} : \text{Type} \\
\text{ImmutableField} &::= \textbf{val}\ \text{FieldIdentifier} : \text{Type}
\end{aligned}
$$

Figure 68: Syntactical Extensions for Class Definition

Classes consist of an identifier, a superclass (which will be *Any* if none is specified), a number of implemented traits, a set of fields and set of methods. The constructor is the method **init**, which may have any number of parameters, but must always return a pointer to the constructed object.

Traits are similar to classes. They consist of an identifier, a number of implemented traits and sets of fields and methods. They have no constructor and may not be instantiated.

Methods are comparable to functions, except that they are named.

Fields can be either mutable or immutable. Immutable fields are mutable until their initial assignment.

$$
\begin{aligned}
\text{Type} &::= \text{Pointer} \mid \text{Unit} \mid \text{ClassIdentifier} \mid \text{TraitIdentifier} \mid \text{FType} \\
\text{FType} &::= (\overrightarrow{\text{Type}}) : \text{Type}
\end{aligned}
$$

Figure 69: Syntactical Extensions for Classes

Secondly, types now consist of those defined by classes and traits, a pointer type and a function type.

There are no longer any primitive types, as those will be implemented using classes and syntactic sugar, for a more unified type system.

We will refer to class and trait identifiers using C_i and T_i respectively, to method identifiers using m_i and to field identifiers using f_i.

Finally there are three new expressions:

$$
\begin{aligned}
\text{Expression} ::= \ldots &\mid \textbf{call}\ \text{Expression}\,.\,\text{MethodIdentifier}(\overrightarrow{\text{Expression}}) \mid \text{Expression}\,.\,\text{FieldIdentifier} \\
&\mid \textbf{new}\ \text{ClassIdentifier}(\overrightarrow{\text{Expression}})
\end{aligned}
$$

Figure 70: Syntactical Extensions for Object Expressions

- Method call **call** $e.m(\vec{e})$, which calls a method on an object.
- Field dereference $e.f$, which points to a field of an object.
- Object allocation **new** $C(\vec{e})$, which allocates a new object of type C in memory and returns a pointer to it.

4.6.2 Semantics: Runtime Structure

To express the semantics of classes and traits, we have altered the runtime structure as follows:

$$\begin{aligned} \text{HeapElement} &= \text{Address} \mid \text{FunctionStore} \mid \text{ObjectStore} \\ \text{ObjectStore} &= \text{FieldIdentifier} \rightharpoonup \text{Address} \times \text{Mutability} \\ \text{Mutability} &::= \textbf{mutable} \mid \textbf{immutable} \end{aligned}$$

Figure 71: Extended Runtime Structure for Objects

To allow objects to be stored on the heap, we have added a new type of heap element in the form of the object store. An object store is a finite partial function from field identifiers to tuples of an address, which is the value of the field and a marker stating if a field is mutable or immutable. We define $h[a_1.f \mapsto a_2]$ as a shorthand for $[h[a_1 \mapsto \langle t, f \mapsto \langle a_2 \text{ mt}\rangle\rangle]]$.

The class and trait definitions, including the methods, are not part of the state, but exist as a given. We will use lookup functions when information contained in these definitions is required. The class and trait definitions define *class tables* : $ct \in \text{Class} \cup \text{Trait}$. We define $\preceq_{ct}$ as the partial order induced on type identifiers by class table ct, with the following restrictions :

1. If type t occurs in ct, then t is declared in ct.
2. ct does not contain duplicate declarations, or declarations of **Any** or **Unit**.

We can now inductively define subtyping:

$$\frac{}{t_1 :< \textbf{Any}}$$

$$\frac{}{t_1 :< t_1}$$

$$\frac{t_1 :< t_2 \; t_2 :< t_3}{t_1 :< t_3}$$

$$\frac{t_1 \textbf{ extends } t_2 \textbf{ with } \overrightarrow{t}}{\forall t_3 \in (t_2 :: \overrightarrow{t}) \mid t_1 :< t_3}$$

Figure 72: The Inductively Defined Subtyping Relation

Furthermore, we have added additional singular expression contexts for dereferenced expressions, method calls and constructor calls:

$$K_S ::= \ldots \mid \text{deref}\,\square.f \mid \text{deref}\,\square.m \mid \text{methodCall}\ \overrightarrow{a}\ \square \mid \text{ctorCall}\ \overrightarrow{a}\ \square$$

Figure 73: The Singular Expression Contexts Expanded with Method Contexts

These additional contexts have the following meaning:

- Deref Field $\text{deref}\,\square.f$ marks the currently focussed subexpression as an object expression that will be dereferenced by field f.
- Deref Method $\text{deref}\,\square.m$ marks the currently focussed subexpression as an object expression that will be dereferenced by method m.

- Method Call methodCall $\overrightarrow{a}$ $\square$ marks the currently focussed subexpression as the body of a method being called.
- Constructor Call ctorCall $\overrightarrow{a}$ $\square$ marks the currently focussed subexpression as the body of a constructor being called.

There are no additional contexts for parameter evaluation in methods, as those used for functions can be used for methods as well.

4.6.3 Semantics: Reduction Rules

With support for classes and traits comes a number of additional reduction rules, which we give in Fig. 74.

For dereferenced expressions:

$\mathbf{S}(k, (\searrow e.f), h) \rightarrow_{ct} \mathbf{S}(\text{deref}\,\square.f :: k, (\searrow e), h)$ **red_derefField**

$\mathbf{S}(k, (\searrow e.m), h) \rightarrow_{ct} \mathbf{S}(\text{deref}\,\square.m :: k, (\searrow e), h)$ **red_derefMethod**

$\mathbf{S}(\text{deref}\,\square.f :: k, (\nearrow \mathbf{ref}\, a), h) \rightarrow_{ct} \mathbf{S}(k, (\nearrow \mathbf{ref}\, a.f), h)$ **red_derefFieldUp**

$\mathbf{S}(\text{deref}\,\square.m :: k, (\nearrow \mathbf{ref}\, a), h) \rightarrow_{ct} \mathbf{S}(k, (\nearrow \mathbf{ref}\, a.m), h)$ **red_derefMethodUp**

For object allocation:

$\mathbf{S}(k, (\searrow \mathbf{new}\, C_1(\overrightarrow{e})), h)$

$\rightarrow_{ct} \mathbf{S}(\text{call}\ \mathbf{ref}\, a.\mathbf{init}\ \square :: k, (\searrow (\mathbf{ref}\, a :: \overrightarrow{e})), h[a \mapsto o])$ **red_new**

for any a, o where $\texttt{fld}(C_1, \text{flds}) \wedge a \notin dom(h)$

$\wedge\, o = \{\langle f, \bot\rangle \mid \langle t_2, f\rangle \in \text{flds}\}$

For method invocation:

$\mathbf{S}(\text{call}\ \mathbf{ref}\, a.m\ \square :: k, (\nearrow \overrightarrow{a}), h)$

$\rightarrow_{ct} \mathbf{S}(\text{methodCall}\ \overrightarrow{a}\ \square :: k, (\searrow e), h)$ **red_methodCall**

for any e where $\texttt{mbLookup}(\mathbf{fst}\, h(a)\ m\ e) \wedge \neg(m = \mathbf{init})$

$\mathbf{S}(\text{call}\ \mathbf{ref}\, a.\,\mathbf{init}\ \square :: k, (\nearrow \overrightarrow{a}), h)$

$\rightarrow_{ct} \mathbf{S}(\text{ctorCall}\ \overrightarrow{a}\ \square :: k, (\searrow e), h)$ **red_ctorCall**

for any e where $\texttt{mbLookup}(\mathbf{fst}\, h(a)\ m\ e)$

For method returns:

$\mathbf{S}(\text{methodCall}\ \overrightarrow{a}\ \square :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow v), h)$ **red_methodReturn**

$\mathbf{S}(\text{ctorCall}(a :: \overrightarrow{a})\ \square :: k, (\nearrow v), h) \twoheadrightarrow \mathbf{S}(k, (\nearrow \mathbf{ref}\, a), h)$ **red_ctorReturn**

For field lookup:

$\mathbf{S}(\text{load}\ \square :: k, (\nearrow \mathbf{ref}\, a_1.f), h) \rightarrow_{ct} \mathbf{S}(k, (\nearrow \mathbf{ref}\, a_2), h)$ **red_fLoad**

for any a_2 where $h[a_1.f \mapsto a_2]$

For field assignment:

$\mathbf{S}(\mathbf{ref}\, a_1.f := \square :: k, (\nearrow \mathbf{ref}\, a_2), h)$

$\rightarrow_{ct} \mathbf{S}(k, (\nearrow \mathbf{unit}), h[a_1.f \mapsto \langle a_2, mtb\rangle])$ **red_fStore**

where $((\text{snd}\, h(a_1))(f) = \bot \vee \text{snd}(\text{snd}\, h(a_1))(f) = \mathbf{mutable})$

$\wedge\, \text{fst}\, h(a_1) = t \wedge \texttt{lookupMtb}(t, mtb)$

Figure 74: Reduction rules for Classes & Traits

These reduction rules have the following meaning:

- **red_new** allocates a new instance of an object of type C_1 to the heap. The fields – both direct and inherited – are looked up using `fld` – which is defined in Fig. 76 – and added to the domain of the partial field function, with value $\bot$, marking them as uninitialized. The constructor is scheduled for execution with the given parameters, with the pointer to the newly allocated object store, i.e. the **this** pointer, as the first parameter.
- **red_methodCall** calls the method with name m on the object with address a, if the method is not a constructor. The method is looked up in the definition, of the type of the object and its supertypes, using `mbLookup` – which is defined in Fig. 75. The method body e is executed in a singular method call context.
- **red_ctorCall** is essentially identical to method calls, but only used in the case of constructors. The constructor body e is executed in a singular constructor call context, which will be used to return the pointer to the newly allocated object.
- **red_methodReturn** is functionally identical to the previously **red_return**, but for methods.
- **red_ctorReturn** instead of returning the value of the expression, constructors return the pointer to the object store of the newly allocated object.
- **red_fLoad** is similar to the previously discussed **red_load**, but for pointers to objects dereferenced with a field.
- **red_fStore** stores a value in an object field, that is either uninitialized or mutable. The newly stored value is annotated with the mutability of the field, which is looked up using `mtbLookup` – which is defined in Fig. 77.
- Parameter reduction for methods and constructors is done using the same rules we defined for functions.

$$\texttt{mbLookup Base}\ \frac{\textbf{class Any}\ \{\ldots m(\overrightarrow{t})\{e\}\ \ldots\}}{\texttt{mbLookup}(m, \textbf{Any}, e)}$$

$$\texttt{mbLookup Defining Class}\ \frac{\textbf{class}\ C\ \{\ldots m(\overrightarrow{t})\{e\}\ \ldots\}}{\texttt{mbLookup}(m, C, e)}$$

$$\texttt{mbLookup Defining Trait}\ \frac{\textbf{trait}\ C\ \{\ldots m(\overrightarrow{t})\{e\}\ \ldots\}}{\texttt{mbLookup}(m, C, e)}$$

$$\texttt{mbLookup Super Class}\ \frac{\begin{array}{c}\textbf{class}\ C_1\ \textbf{extends}\ C_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \\ m \notin \mathrm{dom}(\overline{m}) \\ \exists t \in (C_2 :: \overrightarrow{T})\ |\ \texttt{mblookup}(m, t, e)\end{array}}{\texttt{mbLookup}(m, C_1, e)}$$

$$\texttt{mbLookup Super Trait}\ \frac{\begin{array}{c}\textbf{trait}\ T_1\ \textbf{extends}\ T_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \\ m \notin \mathrm{dom}(\overline{m}) \\ \exists T \in (T_2 :: \overrightarrow{T})\ |\ \texttt{mblookup}(m, T, e)\end{array}}{\texttt{mbLookup}(m, C_1, e)}$$

Figure 75: The Inductively Defined `mbLookup` Relation

$$\texttt{fld}\ \text{Base}\ \frac{}{\texttt{fld}(\textbf{Any}, \emptyset)}$$

$$\texttt{fld}\ \text{Extended Class}\ \frac{\textbf{class}\ C_1\ \textbf{extends}\ C_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \quad flds = \overline{f} \cup \bigcup\{\overline{f'} \mid \texttt{fld}(t, \overline{f'}) \wedge t \in (C_2 :: \overrightarrow{T})\}}{\texttt{fld}(C_1, flds)}$$

$$\texttt{fld}\ \text{Extended Trait}\ \frac{\textbf{trait}\ T_1\ \textbf{extends}\ T_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \quad flds = \overline{f} \cup \bigcup\{\overline{f'} \mid \texttt{fld}(t, \overline{f'}) \wedge t \in (T_2 :: \overrightarrow{T})\}}{\texttt{fld}(T_1, flds)}$$

Figure 76: The Inductively Defined `fld` Relation

$$\texttt{mtbLookup}\ \text{Mutable}\ \frac{\texttt{flds}(t, \overline{f}) \quad \textbf{var}\ f\ \ldots \in \overline{f}}{\texttt{mtbLookup}(t, f, \textbf{mutable})}$$

$$\texttt{mtbLookup}\ \text{Immutable}\ \frac{\texttt{flds}(t, \overline{f}) \quad \textbf{val}\ f\ \ldots \in \overline{f}}{\texttt{mtbLookup}(t, f, \textbf{immutable})}$$

Figure 77: The Inductively Defined `mtbLookup` Relation

4.6.4 Example Program with Classes

With the new reduction rules we can now evaluate the program in Listing 32:

```
class Foo {}
class Bar
{
    var x : Foo;

    init()
    {
        x := new Foo();
    }
}
new Bar();
```

Listing 34: A Small Program with Classes

$$\begin{aligned}\mathbf{k_1} &= [] \\ \boldsymbol{\phi_1} &= \searrow \textbf{new}\ \text{Bar}([]) \\ \mathbf{h_1} &= \{\} \\ \mathbf{S_1} &= \mathbf{S}(k_1, \phi_1, h_1)\end{aligned}$$

Figure 78: The Initial State in the Reduction of Listing 34

The initial state in the reduction of Listing 34 is given by Fig. 78: The context and heap are empty and the focused top expression is an object allocation expression. To allocate an object of type **Bar**, we apply **red_new** – which results in the state of Fig. 79.

$$
\begin{aligned}
\mathbf{k_2} &= \text{call } \mathbf{ref}\, a_{o_1}.\mathbf{init}\ \square \\
&\quad :: [] \\
\boldsymbol{\phi_2} &= \searrow [] \\
\mathbf{h_2} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
\mathbf{S_2} &= \mathbf{S}(k_2, \phi_2, h_2)
\end{aligned}
$$

Figure 79: The Second State in the Reduction of Listing 34

In the second state, after **red_new** has been applied, the context now consists of a singular call context, to call the constructor method. The heap contains the newly allocated Bar object at address a_{o_1}. The focused expression is the list of parameter expressions, which is empty. Because it is empty, we apply **red_paramsEmpty** – which results in the state of Fig. 80.

$$
\begin{aligned}
\mathbf{k_3} &= \text{call } \mathbf{ref}\, a_{o_1}.\mathbf{init}\ \square \\
&\quad :: [] \\
\boldsymbol{\phi_3} &= \nearrow [] \\
\mathbf{h_3} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
\mathbf{S_3} &= \mathbf{S}(k_3, \phi_3, h_3)
\end{aligned}
$$

Figure 80: The Third State in the Reduction of Listing 34

In the third state, the empty list of expressions has been reduced to an empty list of values, which means the actual constructor call can proceed, by applying **red_ctorCall** – which results in the state of Fig. 81.

$$
\begin{aligned}
\mathbf{k_4} &= \text{ctorCall}(\mathbf{ref}\, a_{o_1} :: []\ \square \\
&\quad :: [] \\
\boldsymbol{\phi_4} &= \searrow x_0.x := \mathbf{new}\ \text{Foo}([]) \\
\mathbf{h_4} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
\mathbf{S_4} &= \mathbf{S}(k_4, \phi_4, h_4)
\end{aligned}
$$

Figure 81: The Fourth State in the Reduction of Listing 34

In the fourth state, after applying **red_ctorCall**, it is time to evaluate the expression which makes up the body of the constructor. The constructor expression is an assignment expression with unevaluated left and right hand expressions, so we start by evaluating the left hand side using **red_binRedLeft** – which results in the state of Fig. 82.

$$
\begin{aligned}
\mathbf{k_5} &= \square := \textbf{new}\,\text{Foo}([]) \\
&\quad :: \text{ctorCall}(\textbf{ref}\,a_{o_1} :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_5} &= \searrow x_0.x \\
\mathbf{h_5} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
\mathbf{S_5} &= \mathbf{S}(k_5, \phi_5, h_5)
\end{aligned}
$$

Figure 82: The Fifth State in the Reduction of Listing 34

In the fifth state, after applying **red_binRedLeft**, we encounter a dereferenced ident expression, so we apply **red_derefField** – which results in the state of Fig. 83.

$$
\begin{aligned}
\mathbf{k_6} &= \text{deref}\,\square.x \\
&\quad :: \square := \textbf{new}\,\text{Foo}([]) \\
&\quad :: \text{ctorCall}(\textbf{ref}\,a_{o_1} :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_6} &= \searrow x_0 \\
\mathbf{h_6} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
\mathbf{S_6} &= \mathbf{S}(k_6, \phi_6, h_6)
\end{aligned}
$$

Figure 83: The Sixth State in the Reduction of Listing 34

In the sixth state, after applying **red_derefField**, we can now apply **red_ident**, to reduce the ident expression to a value – which results in the state of Fig. 84.

$$
\begin{aligned}
\mathbf{k_7} &= \text{deref}\,\square.x \\
&\quad :: \square := \textbf{new}\,\text{Foo}([]) \\
&\quad :: \text{ctorCall}(\textbf{ref}\,a_{o_1} :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_7} &= \nearrow \textbf{ref}\,a_{o_1} \\
\mathbf{h_7} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
&\ 7
\end{aligned}
$$

Figure 84: The Seventh State in the Reduction of Listing 34

In the seventh state, after **red_ident** has been applied, we now have the address of the object, so we can apply **red_derefFieldUp** to combine it with the field name to a dereferenced field value – which results in the state of Fig. 85.

$$
\begin{aligned}
\mathbf{k_8} &= \square := \textbf{new}\,\text{Foo}([]) \\
&\quad :: \text{ctorCall}(\textbf{ref}\,a_{o_1} :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_8} &= \nearrow \textbf{ref}\,a_{o_1}.x \\
\mathbf{h_8} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
\mathbf{S_8} &= \mathbf{S}(k_8, \phi_8, h_8)
\end{aligned}
$$

Figure 85: The Eighth State in the Reduction of Listing 34

In the eighth state, after applying **red_derefFieldUp**, the left hand side of the assignment expression has been fully reduced, so we proceed to the right hand side by applying **red_binRedRight** – which results in the state of Fig. 86.

$$
\begin{aligned}
\mathbf{k_8} &= \textbf{ref}\,a_{o_1}.x := \square \\
&\quad :: \text{ctorCall}(\textbf{ref}\,a_{o_1} :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_8} &= \searrow \textbf{new}\,\text{Foo}([]) \\
\mathbf{h_8} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}\} \\
\mathbf{S_8} &= \mathbf{S}(k_8, \phi_8, h_8)
\end{aligned}
$$

Figure 86: The Ninth State in the Reduction of Listing 34

In the ninth state, after applying **red_binRedRight**, we encounter an object allocation expression, so we once again apply **red_new** to allocate it – which results in the state of Fig. 87.

$$
\begin{aligned}
\mathbf{k_8} &= \text{call}\,\textbf{ref}\,a_{o_2}.\textbf{init} \\
&\quad :: \textbf{ref}\,a_{o_1}.x := \square \\
&\quad :: \text{ctorCall}(\textbf{ref}\,a :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_8} &= \searrow [] \\
\mathbf{h_8} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}, a_{o_2} \mapsto \{\}\} \\
\mathbf{S_8} &= \mathbf{S}(k_8, \phi_8, h_8)
\end{aligned}
$$

Figure 87: The 8^{th} State in the Reduction of Listing 34

In the 8^{th} state, the object has been allocated to the heap as before and we proceed to the constructor call parameters using **red_paramsEmpty**, as they are once again empty – which results in the state of Fig. 88.

$\mathbf{k_9}$ = call **ref** a_{o_2}.**init**
:: **ref** $a_{o_1}.x := \square$
:: ctorCall(**ref** a_{o_1} :: []) $\square$
:: []
$\boldsymbol{\phi_9} = \nearrow$ []
$\mathbf{h_9} = \{a_{o_1} \mapsto \{x \mapsto \bot\}, a_{o_2} \mapsto \{\}\}$
$\mathbf{S_9} = \mathbf{S}(k_9, \phi_9, h_9)$

Figure 88: The 9^{th} State in the Reduction of Listing 34

In the 9^{th} state, the actual constructor call takes place, by applying **red_ctorCall** – which results in the state of Fig. 89.

$\mathbf{k_1 0}$ = ctorCall(**ref** a_{o_2} :: []) $\square$ :: **ref** $a_{o_1}.x := \square$
:: ctorCall(**ref** a_{o_1} :: []) $\square$
:: []
$\boldsymbol{\phi_1 0} = \searrow$ **skip**
$\mathbf{h_1 0} = \{a_{o_1} \mapsto \{x \mapsto \bot\}, a_{o_2} \mapsto \{\}\}$
$\mathbf{S_1 0} = \mathbf{S}(k_1 0, \phi_1 0, h_1 0)$

Figure 89: The 10^{th} State in the Reduction of Listing 34

In the 10^{th} state, we encounter the default constructor body, in the form of a skip expression. We apply **red_skip** to proceed to the state of Fig. 90.

$\mathbf{k_1 0}$ = ctorCall(**ref** a_{o_2} :: []) $\square$ :: **ref** $a_{o_1}.x := \square$
:: ctorCall(**ref** a_{o_1} :: []) $\square$
:: []
$\boldsymbol{\phi_1 0} = \nearrow$ **unit**
$\mathbf{h_1 0} = \{a_{o_1} \mapsto \{x \mapsto \bot\}, a_{o_2} \mapsto \{\}\}$
$\mathbf{S_1 0} = \mathbf{S}(k_1 0, \phi_1 0, h_1 0)$

Figure 90: The 10^{th} State in the Reduction of Listing 34

In the 10^{th} state, we encounter a returned value in a constructor call context, which means the body has finished executing. Instead of returning the value, as we would with a method or function call, we return the pointer to the parent object of this constructor, by applying **red_ctorReturn** – which results in the state of Fig. 91.

$$
\begin{aligned}
\mathbf{k_1 1} &= \mathbf{ref}\, a_{o_1}.x := \square \\
&\quad :: \mathsf{ctorCall}(\mathbf{ref}\, a_{o_1} :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 1} &= \nearrow \mathbf{ref}\, a_{o_2} \\
\mathbf{h_1 1} &= \{a_{o_1} \mapsto \{x \mapsto \bot\}, a_{o_2} \mapsto \{\}\} \\
\mathbf{S_1 1} &= \mathbf{S}(k_{14}, \phi_{14}, h_{14})
\end{aligned}
$$

Figure 91: The 11^{th} State in the Reduction of Listing 34

In the 11^{th} state, after the constructor has returned, we now encounter a returned value in a right binary assignment context, which means the subexpressions have been reduced and we can perform the actual assignment by applying **red_fStore** – which results in the state of Fig. 92.

$$
\begin{aligned}
\mathbf{k_1 1} &= \mathsf{ctorCall}(\mathbf{ref}\, a_{o_1} :: []) \square \\
&\quad :: [] \\
\boldsymbol{\phi_1 1} &= \nearrow \mathbf{unit} \\
\mathbf{h_1 1} &= \{a_{o_1} \mapsto \{x \mapsto \langle a_{o_2}, \mathbf{mutable}\rangle\}, a_{o_2} \mapsto \{\}\} \\
\mathbf{S_1 1} &= \mathbf{S}(k_1 1, \phi_1 1, h_1 1)
\end{aligned}
$$

Figure 92: The 11^{th} State in the Reduction of Listing 34

In the 11^{th} state, after applying **red_fStore**, the heap reflects the assignment and the body of the constructor has finished executing. We once again return the pointer to the parent object using **red_returnCall** – which results in the state of Fig. 93.

$$
\begin{aligned}
\mathbf{k_1 2} &= [] \\
\boldsymbol{\phi_1 2} &= \nearrow \mathbf{ref}\, a_{o_1} \\
\mathbf{h_1 2} &= \{a_{o_2} \mapsto \{x \mapsto \langle a_{o_2}, \mathbf{mutable}\rangle\}, a_{o_2} \mapsto \{\}\} \\
\mathbf{S_1 2} &= \mathbf{S}(k_1 2, \phi_1 2, h_1 2)
\end{aligned}
$$

Figure 93: The 12^{th} State in the Reduction of Listing 34

The 12^{th} state is the final state of this example, with the topmost expression being the evaluated value of the entire expression in an empty context.

4.7 Extending *Scala Core* with Threads & Locking

The final extension, which completes *Scala Core*, will be the addition of threads and locks, to allow for concurrency. In this section we will extend the runtime structure to deal with threads and locks, add a number of contexts, provide the rules to fork and join threads and to lock and unlock and finally give an example derivation of a multithreaded program.

4.7.1 Syntax

To add threads and locks in the *Java* and *Scala* sense, there is no need to add additional syntax, as the functionality will be stored in the methods **fork**(), **join**(), **lock**() and **unlock**() on **Any**, with the following meaning:

- Fork thread *e*.**fork**() creates a new thread and runs the **run**-method on the object in that thread; the pointer to the object serves as the thread-identifier. The expression *e* must always result in an instance of an object with a **run**() method, returning **unit**. **fork**() always returns **unit**.
- Join thread *e*.**join**() joins a finished thread; it always returns **unit**.
- Lock *e*.**lock**() assigns the lock with the pointer of the callee object as identifier to the current thread, or raises the reentrancy count by one, if it is already owned by this thread; it always returns **unit**.
- Unlock *e*.**unlock**() removes the lock with the pointer of the callee object as identifier, if it belongs to the current thread, or decreases the reentrancy count by one if it is higher than one; it always returns **unit**.

4.7.2 Semantics : Runtime Structure

Adding support for threading and locking requires a significant reorganization in the runtime structure of the semantics. First we have added the required structures to represent and store threads and locks:

$$\begin{aligned} \text{Threadpool} &= \text{Address} \rightharpoonup \mathit{Thread} \\ \text{Thread} &= \text{Context} \times \text{Focus} \\ \text{Locktable} &= \text{Address} \rightharpoonup \mathit{Address} \times \text{ReentrancyLevel} \\ \text{ReentrancyLevel} &= \mathbb{N} \end{aligned}$$

Figure 94: Runtime Structures for Threads and Locks

The threadpool, which is a finite partial function from a thread identifier, in the form of an object address, to threads, keeps track of the threads in a program and which object represents them. Threads themselves have their own focus and context and execute als programs previously would, but sharing the heap. Initially the threadpool will always consist of one thread, which executes the main program expression, which may spawn additional threads. In the semantics we will refer to threads as $\tau_i = \phi$ in k, to threadpools as $p_i = a_1$ is $\tau_1 \mid \ldots \mid a_n$ is τ_n and to lock tables as l_i.

The lock table, which is a finite partial function from addresses to pairs of an address and a reentrancy level, maps locks to pairs of the thread that owns the lock and the number of times they have been reentrantly acquired.

Secondly we have adapted the states in our semantics to multithreading:

$$\text{State} = \text{Heap} \times \text{Threadpool} \times \text{LockTable}$$

Figure 95: State for Threads and Locks

Since we now have multiple threads executing, the single focus and context per state will no longer suffice. Therefore we have redefined states as a tuple of a heap, a threadpool and a lock table. The semantics continue to sequentially execute small steps, but these steps will be interleaved by non-determinically choosing one of the currently enabled threads.

We will continue to refer to states with S_i and will use the syntactic shorthand of $S_i = \mathbf{S}(h, p, l)$.

4.7.3 Semantics : Reduction Rules

The reduction rules that allow for multithreading are the following:

For method invocation:

$$\mathbf{S}(h, (p \mid a \text{ is}(\nearrow \vec{a}) \text{ in}(\text{call } \mathbf{ref}\ a.m\ \square :: k)), l)$$
$$\rightarrow_{ct} \mathbf{S}(h, (p \mid a \text{ is}(\searrow e) \text{ in}(\text{methodCall } \vec{a}\ \square :: k)), l) \qquad \textbf{red\_methodCall}$$
$$\text{for any } e \mid \texttt{mbLookup}(\mathbf{fst}\ h(a)\ m\ e) \wedge m \notin \mathbf{restricted}$$

For forking & joining threads:

$$\mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow []) \text{ in}(\text{call } \mathbf{ref}\ a_2.\mathbf{fork}\ \square :: k)), l)$$
$$\rightarrow_{ct} \mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow \mathbf{unit}) \text{ in}(k) \mid a_2 \text{ is}(\searrow \mathbf{call}\ \mathbf{ref}\ a_2.\mathbf{run}()) \text{ in}([])), l) \qquad \textbf{red\_fork}$$
$$\mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow []) \text{ in}(\text{call } \mathbf{ref}\ a_2.\mathbf{join}\ \square :: k_1) \mid a_2 \text{ is}(\nearrow v) \text{ in}(k_2)), l)$$
$$\rightarrow_{ct} \mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow \mathbf{unit}) \text{ in}(k_1) \mid a_2 \text{ is}(\nearrow v) \text{ in}(k_2)), l) \qquad \textbf{red\_join}$$

For locking & unlocking:

$$\mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow []) \text{ in}(\text{call } \mathbf{ref}\ a_2.\mathbf{lock}\ \square :: k)), l_1)$$
$$\rightarrow_{ct} \mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow \mathbf{unit}) \text{ in}(k)), l_2) \qquad \textbf{red\_lock}$$
$$\text{for any } l_2 \mid (l_1(a_2) = \bot \wedge l_2 = l_1[a_2 \mapsto \langle a_1, 1\rangle])$$
$$\vee\ (l_1(a_2) = \langle l_1, i\rangle \wedge l_2 = l_1[a_2 \mapsto \langle a_1, i+1\rangle])$$
$$\mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow []) \text{ in}(\text{call } \mathbf{ref}\ a_2.\mathbf{unlock}\ \square :: k)), l_1)$$
$$\rightarrow_{ct} \mathbf{S}(h, (p \mid a_1 \text{ is}(\nearrow \mathbf{unit}) \text{ in}(k)), l_2) \qquad \textbf{red\_unlock}$$
$$\text{for any } l_2 \mid l_1(a_2) = \langle a_1, i\rangle \wedge ((i > 1 \wedge l_2 = l_1[a_2 \mapsto \langle a_1, i-1\rangle])$$
$$\vee\ (i = 1 \wedge l_2 = l_1[a_2 \mapsto \bot]))$$

Figure 96: Reduction rules for Multithreading

these additional rules have the following meaning:

- **red_methodCall** is adjusted to skip any method m mentioned in the set of restricted method names, currently defined as **restricted** $= \{$**init**, **fork**, **join**, **lock**, **unlock**$\}$. It also demonstrates how rules on our previous state definition can be rewritten to work with the new state.
- **red_fork** spawns a new thread which executes the **run** method on the callee object identified by a_2. The new thread is given a_2 as an identifier. The context in the new thread is empty, as the call to **run** will be the topmost expression.
- **red_join** joins a finished thread. The fact that the thread is finished is indicated by the focus being on a returned value, which means the expression of the thread has been fully evaluated.
- **red_lock** grants the current thread a free lock or reacquires a lock it already owns. If a lock is already owned, it is reentrantly acquired, by raising the reentrancy level by one. Newly acquired locks are given a reentrancy level of one.
- **red_unlock** decreases the reentrancy level by one for reentrantly acquired locks and releases the lock owned by the current thread for non reentrantly acquired locks.

4.7.4 Example Multithreaded Program

With these reduction rules we can now evaluate the program in Listing 35:

```
class Foo
{
    run()
    {
        this.lock();
        skip;
        this.unlock()
    }
}
val o : Foo;
o := new Foo();
o.fork();
o.join()
```

Listing 35: A Small Multithreaded Program

$$
\begin{aligned}
\mathbf{k}^1_{17} &= \text{call}(\textbf{ref}\, a_o).\textbf{fork} \\
&\quad :: \square; (\textbf{load}\, x_0).\textbf{join}() \\
&\quad :: \textbf{unit}; \square \\
&\quad :: \text{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi}^1_{17} &= \nearrow [] \\
\mathbf{p_{17}} &= \{\_ \mapsto \langle \phi^1_{17}, k^1_{17} \rangle\} \\
\mathbf{l_{17}} &= \{\} \\
\mathbf{h_{17}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}\} \\
\mathbf{S_{17}} &= \mathbf{S}(p_{17}, l_{17}, h_{17})
\end{aligned}
$$

Figure 97: The 17^th State in the Reduction of Listing 35

As declarations, object instantiation, assignment and loads are by now all familiar and essentially unchanged from single to multi-threaded programs, we will start this example in the 17^{th} state, where we ecounter a (fork) method on an object pointer, which allows us to apply **red_fork** – resulting in the state of Fig. 98.

$$
\begin{aligned}
\mathbf{k^1_{18}} &= \square;\ (\mathbf{load}\ x_0).\mathbf{join}() \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathrm{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{18}} &= \nearrow \mathbf{unit} \\
\mathbf{k^2_1} &= [] \\
\boldsymbol{\phi^2_1} &= \mathbf{call\ ref}\ a_o.\mathbf{run}() \\
\mathbf{p_{18}} &= \{\_ \mapsto \langle \phi^1_{18}, k^1_{18} \rangle, a_o \mapsto \langle \phi^2_1, k^2_1 \rangle\} \\
\mathbf{l_{18}} &= \{\} \\
\mathbf{h_{18}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}\} \\
\mathbf{S_{18}} &= \mathbf{S}(p_{18}, l_{18}, h_{18})
\end{aligned}
$$

Figure 98: The 18^th State in the Reduction of Listing 35

In the 18^{th} state, after applying **red_fork**, the threadpool now shows an additional thread, about to execute the run method on the thread object. Execution in either the main thread or the new thread may proceed. We, taking the role of the scheduler, decide to first execute 6 steps on the main thread, until the **join** method call is the focused expression, in state 24 – shown in Fig. 99.

$$
\begin{aligned}
\mathbf{k^1_{24}} &= \mathrm{call}\ \mathbf{ref}\ a_o.\mathbf{join} \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathrm{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{24}} &= \nearrow [] \\
\mathbf{k^2_1} &= [] \\
\boldsymbol{\phi^2_1} &= \mathbf{call\ ref}\ a_o.\mathbf{run}() \\
\mathbf{p_{24}} &= \{\_ \mapsto \langle \phi^1_{24}, k^1_{24} \rangle, a_o \mapsto \langle \phi^2_1, k^2_1 \rangle\} \\
\mathbf{l_{24}} &= \{\} \\
\mathbf{h_{24}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}\} \\
\mathbf{S_{24}} &= \mathbf{S}(p_{24}, l_{24}, h_{24})
\end{aligned}
$$

Figure 99: The 24^th State in the Reduction of Listing 35

In the 24^{th} state, the main thread can no longer proceed, as the thread it is trying to join has not yet finished execution. Therefore, we execute 6 steps in the second thread, which will bring us to the execution of a **lock** method, in state 30 – which is shown in Fig. 100.

$$
\begin{aligned}
\mathbf{k^1_{24}} &= \text{call } \mathbf{ref}\ a_o.\mathbf{join} \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{24}} &= \nearrow [] \\
\mathbf{k^2_7} &= \text{call } \mathbf{ref}\ a_o.\mathbf{lock} \\
&\quad :: \square; \mathbf{skip}; \mathbf{call\ load}\ x_0.\mathbf{run}() \\
&\quad :: \text{methodCall}(\mathbf{ref}\ a_p :: [])\square \\
\boldsymbol{\phi^2_7} &= \nearrow [] \\
\mathbf{p_{30}} &= \{\_ \mapsto \langle \phi^1_{24}, k^1_{24} \rangle, a_o \mapsto \langle \phi^2_7, k^2_7 \rangle\} \\
\mathbf{l_{30}} &= \{\} \\
\mathbf{h_{30}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}, a_p \mapsto a_o\} \\
\mathbf{S_{30}} &= \mathbf{S}(p_{30}, l_{30}, h_{30})
\end{aligned}
$$

Figure 100: The 30^th State in the Reduction of Listing 35

In the 30^{th} state, we encounter the **lock** method being called on an object pointer, which means we can apply **red_lock** – resulting in the state of Fig. 101.

$$
\begin{aligned}
\mathbf{k^1_{24}} &= \text{call } \mathbf{ref}\ a_o.\mathbf{join} \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{24}} &= \nearrow [] \\
\mathbf{k^2_8} &= \square; \mathbf{skip}; \mathbf{call\ load}\ x_0.\mathbf{run}() \\
&\quad :: \text{methodCall}(\mathbf{ref}\ a_p :: [])\square \\
\boldsymbol{\phi^2_8} &= \nearrow \mathbf{unit} \\
\mathbf{p_{31}} &= \{\_ \mapsto \langle \phi^1_{24}, k^1_{24} \rangle, a_o \mapsto \langle \phi^2_8, k^2_8 \rangle\} \\
\mathbf{l_{31}} &= \{a_o \mapsto \langle a_o, 1 \rangle\} \\
\mathbf{h_{31}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}, a_p \mapsto a_o\} \\
\mathbf{S_{31}} &= \mathbf{S}(p_{31}, l_{31}, h_{31})
\end{aligned}
$$

Figure 101: The $31^t h$ State in the Reduction of Listing 35

In the 31^{st} state, after applying **red_lock**, the lock table shows the current thread owning the lock with a reentrancy count of 1. The lock method always returns **unit**. This completes the first subexpression of a binary expression. The second subexpression is another sequential expression, with the first subexpression consisting of merely a **skip**, so we forward to state 38, where the **unlock** method is being called – in Fig. 102.

$$
\begin{aligned}
\mathbf{k^1_{24}} &= \text{call } \mathbf{ref}\ a_o.\mathbf{join} \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{24}} &= \nearrow [] \\
\mathbf{k^2_{15}} &= \text{call } \mathbf{ref}\ a_o.\mathbf{unlock} \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{methodCall}(\mathbf{ref}\ a_p :: [])\square \\
\boldsymbol{\phi^2_{15}} &= \nearrow [] \\
\mathbf{p_{38}} &= \{\_ \mapsto \langle \phi^1_{24}, k^1_{24} \rangle, a_o \mapsto \langle \phi^2_{15}, k^2_{15} \rangle\} \\
\mathbf{l_{38}} &= \{a_o \mapsto \langle a_o, 1 \rangle\} \\
\mathbf{h_{38}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}, a_p \mapsto a_o\} \\
\mathbf{S_{38}} &= \mathbf{S}(p_{38}, l_{38}, h_{38})
\end{aligned}
$$

Figure 102: The 38^th State in the Reduction of Listing 35

In the 38^{th} state, after some sequential expression reductions and the execution of the **skip**, we encounter the **unlock** method being called on an object pointer. We apply **red_unlock** – which results in the state of Fig. 103.

$$
\begin{aligned}
\mathbf{k^1_{24}} &= \text{call } \mathbf{ref}\ a_o.\mathbf{join} \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{24}} &= \nearrow [] \\
\mathbf{k^2_{16}} &= \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{methodCall}(\mathbf{ref}\ a_p :: [])\square \\
\boldsymbol{\phi^2_{16}} &= \nearrow \mathbf{unit} \\
\mathbf{p_{39}} &= \{\_ \mapsto \langle \phi^1_{24}, k^1_{24} \rangle, a_o \mapsto \langle \phi^2_{16}, k^2_{16} \rangle\} \\
\mathbf{l_{39}} &= \{a_o \mapsto \bot\} \\
\mathbf{h_{39}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}, a_p \mapsto a_o\} \\
\mathbf{S_{39}} &= \mathbf{S}(p_{39}, l_{39}, h_{39})
\end{aligned}
$$

Figure 103: The 39^th State in the Reduction of Listing 35

In the 39^{th} state, **red_unlock** has been applied, which resulted in the entry in the lock table belonging to this thread being cleared. **unlock** always returns **unit**. With the rightmost subexpression in the method body expression finished, the sequential expression can finished along with the method call itself – resulting in the state of Fig. 104.

$$
\begin{aligned}
\mathbf{k^1_{24}} &= \text{call } \mathbf{ref}\ a_o.\mathbf{join} \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{24}} &= \nearrow [] \\
\mathbf{k^2_{21}} &= [] \\
\boldsymbol{\phi^2_{21}} &= \nearrow \mathbf{unit} \\
\mathbf{p_{44}} &= \{\_ \mapsto \langle \phi^1_{24}, k^1_{24} \rangle, a_o \mapsto \langle \phi^2_{21}, k^2_{21} \rangle\} \\
\mathbf{l_{44}} &= \{a_o \mapsto \bot\} \\
\mathbf{h_{44}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}, a_p \mapsto a_o\} \\
\mathbf{S_{44}} &= \mathbf{S}(p_{44}, l_{44}, h_{44})
\end{aligned}
$$

Figure 104: The 44^th State in the Reduction of Listing 35

In the 44^{th} state, the second thread has finished executing, so the join in the main thread may now proceed. We apply **red_join** – which results in the state of Fig. 105.

$$
\begin{aligned}
\mathbf{k^1_{25}} &= \mathbf{unit}; \square \\
&\quad :: \mathbf{unit}; \square \\
&\quad :: \text{val}_{a_x} \\
&\quad :: [] \\
\boldsymbol{\phi^1_{25}} &= \nearrow \mathbf{unit} \\
\mathbf{k^2_{21}} &= [] \\
\boldsymbol{\phi^2_{21}} &= \nearrow \mathbf{unit} \\
\mathbf{p_{45}} &= \{\_ \mapsto \langle \phi^1_{25}, k^1_{25} \rangle, a_o \mapsto \langle \phi^2_{21}, k^2_{21} \rangle\} \\
\mathbf{l_{45}} &= \{a_o \mapsto \bot\} \\
\mathbf{h_{45}} &= \{a_x \mapsto a_o, a_o \mapsto \{\}, a_p \mapsto a_o\} \\
\mathbf{S_{45}} &= \mathbf{S}(p_{45}, l_{45}, h_{45})
\end{aligned}
$$

Figure 105: The 45^th State in the Reduction of Listing 35

In the 45^{th} state, the second thread has been joined and the rightmost subexpression in the sequential expression has finished. The program now finishes by applying the usual remaining reduction rules.

4.8 Comparisons

In this section, having defined our semantics and language, we evaluate how it stacks up against other approaches to *Scala* semantics, the informal semantics of *Scala* and the program context semantics for *C*.

4.8.1 Compared to *Scala*

Compared to *Scala*, *Scala Core* is definitely a subset, but it is not a strict one, as we approach some aspects of the language (slightly) differently or in an expanded fashion:

Firstly, functions in *Scala* are just syntactic sugar for classes and objects, whereas we have given them their own type and semantics. The benefit of the *Scala* approach is a completely unified type system and the ability to extend functions similarly to objects. The benefit of our approach is a first-class function type which we can use when defining permission-based separation logic for our model language.

Secondly, exceptions in *Scala* are only allowed to throw objects which are of specific types, whereas our semantics allow throwing any value. This is a minor difference, as constraints on the types throwable can be put in place in our semantics as well.

Thirdly, creating and starting threads in *Scala* is generally similarly to *Java* by extending the **`Thread`** class, instantiating the object and calling the `start` method. We have taken an approach similar to that used in the Vercors project by Amighi et al. [5] by providing a `fork` and `join` method on all objects. In the same vein, while locking is generally done using synchronized blocks, we have provided the `lock` and `unlock` methods on all objects, which combines the functionality of synchronized and explicitly created instances of the **`ReentrantLock`** class.

4.8.2 Compared to existing *Scala* semantics

Previous work in specifying a formal semantics for *Scala* consist primarily of the work done by Cremet [19] and of Cremet et al. [20] in their work on the decidability of *Scala* Type Checking.

The PhD Thesis approaches the semantics by providing a powerful object-calculus called *Scaletta* with a reduction semantics, to which a core functional subset of *Scala* could be translated. This object-calculus is far more capable in expressing the intricacies of the *Scala* type system than our approach, as type parameters and mixins can be fully expressed. However, our approach makes functions explicitly first-class, instead of mapping them to classes and objects, which should provide benefits in bringing CSL to *Scala* in Section 5 and it provides mechanisms for exceptions and multithreading, which the approach by Cremet lacks. More importantly, our approach deals with state, whereas the approach by Cremet doesn not, which is fundamental for our goals of verifying concurrent *Scala* programs with side-effects.

In the decidability paper, Cremet [19] define Featherweight Scala with a small-step semantics, similarly to Featherweight Java[34], which is a purely small functional subset of *Scala*. As before, this semantics is unsuited to our purpose, as it models a stateless language and it lacks the exceptions and multi-threading support we provide.

4.8.3 Compared to the Program Context Semantics for *C*

Compared to the *C* semantics by Krebbers and Wiedijk [37], on which we have based our program context approach, our approach is a significant departure. The major reason for this difference is that *Scala Core* is an expression language, whereas *C* has statements; this leads to far more expression contexts, in the *Scala Core* case, and different reduction rules even for similar language constructs. Another large difference is the fact that *Scala Core* is extended with objects, exceptions and first-class functions, none of which have counterparts in *C* and most of which require significant expansions to the runtime structure.

5 Adapting Permission-Based Separation Logic to *Scala*

5.1 Introduction

Previously the use of permission-based separation logic has been shown successful for languages such as *Java*, but *Scala* brings its own challenges in the form of being an expression-based language with side-effects and first-class functions and closures. In this section we will assess those challenges and determine the point at which the *Java* approach can be adapted and where different paths need to be taken.

To this end, we will – in Section 5.2 – assess the challenges that come with the specification of *Scala Core*. Following this, we shall present a type system in Section 5.3 and a separation logic in Section 5.4, which are capable of solving these challenges for a subset of *Scala Core*, with its accompanying Hoare rules in Section 5.4.3. This smaller logic will then see extensions in Section 5.5, Section 5.6 and Section 5.7, until it is capable of fully specifying *Scala Core* programs. Finally we shall compare our approach to that of others in Section 5.8.

5.2 Elements of our Separation Logic

For the specification of *Scala Core*, our logic will contain a number of elements not commonly found in other variants of separation logic. In this section we will intuitively approach these features before delving into the formal definitions in the next sections.

5.2.1 Expressions and Side-Effects

As opposed to languages with separate commands, which have side-effects, but no value, and expressions, which have values, but no side-effects, *Scala* – and subsequently *Scala Core* – features expressions which have both values and possible side-effects. For instance, for some heap-variable x, `x=6;x` is a valid expression with value 6 and the side-effect of mutating the heap. To illustrate the effect this has on the proof rules of our logic, we shall have a look at the assignment rule:

In a language where expressions do not have side-effects, we could define the $\mapsto$-operator to evaluate the expressions on the left and right hand side, similar to the approach of Birkedal et al. [10, Appendix D. long version]:

$$\frac{}{\Gamma; \Delta \vdash \{P * e_1 \mapsto \_\}\, e_1 := e_2\, \{Q * e_1 \mapsto e_2\}}$$

Figure 106: Sample Assignment Rule

Or we could introduce an operator to more explicitly evaluate expressions – as used by Krebbers and Wiedijk [37]:

$$\frac{}{\Gamma; \Delta \vdash \{P * e_1 \Downarrow v_1 * v_1 \mapsto \_\}\, e_1 := e_2\, \{Q * e_1 \Downarrow v_1 * e_2 \Downarrow v_2 * v_1 \mapsto v_2\}}$$

Figure 107: Sample Assignment Rule

However, in our case, neither of these approaches would suffice, for with side-effects, the evaluation of e_1 could influence the evaluation of e_2:

var Int;
$x_0 := 5; x_0 := x_0 + 1$

Figure 108: Sample Program

We see that, were we to use the same approach as with side-effect free expressions, the left hand side evaluates to x_0 as expected, but the right hand side will be invalid, as the side-effect of the left hand side expression has been ignored. To prevent these types of issues from occurring, we could disallow or restrict them, which is standard in the case of assignments – which can always be rewritten to a sequence of simple assignments without side-effects – but as other expressions such as inline function definitions also produce side-effects and can not be easily disallowed or restricted. Therefore we opt to separately evaluate the left and right hand expressions in the assignment rule and refer to the values resulting from those evaluations and proceed similary with other expressions:

$$\frac{\Gamma; \Delta \vdash \{P\}\, e_1\, \{P' * v_1 = \textbf{result} * v_1 \mapsto \_\} \quad \Gamma; \Delta \vdash \{P'\}\, e_2\, \{P'' * v_2 = \textbf{result}\}}{\Gamma; \Delta \vdash \{P * v_1 \mapsto \_\}\, e_1 := e_2\, \{Q * v_1 \mapsto v_2 * \textbf{result} = \textbf{unit}\}}$$

Figure 109: A Rule for Assignment

In this case the side-effect of the left hand side expression is captured in the state used to evaluate the right hand side expression. We use **result**, in postconditions, to refer to value of the expression in the triple.

5.2.2 Hoare Triples as Assertions

One of the major features of our logic will be the ability to specify the first-class functions with closures in *Scala Core*. To achieve this in our logic, it will feature abstract predicates – previously illustrated in Section 2.3 – and Hoare triples as assertions – similarly to the work of Schwinghammer et al. [53] – instead of being defined, in the different syntactic class of specifications, in the usual fashion. This means our assertions will be used for both predicates on states and on expressions, unlike in standard separation logic. Furthermore it means, because the triples are themselves assertions, they can appear in both the pre- and postconditions of other triples, forming *nested triples*. These nested triples are the key to reasoning about functions. To wit, let us consider the following example:

```
def counter(x:Int) =
{
    var count = 0
    def inc = () => {count += 1; count}
    count = x
    inc
}
val inc = counter(5)
inc() // 6
```

Listing 36: An Example *Scala* Program with Functions

```
val &(Int) : (Unit) : Int;
  x0 := (Int) : (Unit) : Int →
  {
    var Int;
      x0 := 0;
      val &(Unit) : Int;
        x0 := (Unit) : Int →
            {
              x2 := (load x2) + 1
              x2
            };
        x1 := x2;
        x0
  }
  val(Unit) : Int;
    x0 := call load x1(5);
    call x0()
```

```
def counter(x:Int)
=
{
    var count
              = 0
    def inc
             = () =>
             {
                count += 1;
                count
             }
    count = x
    inc
}
val inc
         = counter(5)
inc() // 6
```

Listing 37: The Same Program in *Scala Core*

The example program defines `counter`, which has the local variables `count` and `inc`, which are initialized to 0 and an incrementing function respectively. The counter function assigns its argument to `count` and returns the incrementing function. The counter function is called with the initial value 5 and the resulting function is stored and subsequently called. The resulting value will be 6, as the incrementing function definition closes over the local variable `count`, capturing it and incrementing it on each call. Given such a program, it makes sense to provide `inc` with the following specification:

```
⋮
        x0 := requires x2 ↦ V;
              ensures x2 ↦ V + 1;
              (Unit) : Int →
              {
                x2 := (load x2) + 1;
                x2
              };
⋮
```

Listing 38: Naive Specification of inc

However, this leads to issues, as, by the time the function is called, this specification refers to variables out of scope at the callsite: e.g. x_1 will refer to the local variable holding the reference to `inc`. The solution to this issue involves the use of abstract predicates to abstract from the local variable reference:

```
⋮
        x0 := requires State(V);
              ensures State(V + 1);
              (Unit) : Int →
              {
                x2 := (load x2) + 1;
                x2
              };
⋮
```

Listing 39: inc Specified using Predicates

In this case the predicate $state(V)$ serves as an access ticket to the incrementing function: as long as we can provide an initial instance of $state(V)$ the first time we call the function, we can keep trading subsequent instances of the predicate for renewed access to the function. To provide this initial instance, we define $state(V)$ as $state(V) \stackrel{def}{=} x_1 \mapsto V$ and provide the initial instance in the postcondition of counter:

```
    ⋮
      x0 := requires true;
            ensures State(load x0);
            (Int) : (Unit) : Int →
    ⋮
```

Listing 40: Initial Predicate Instance from counter

In the postcondition of counter the predicate will be proven using its definition, in all further instances the name of the predicate is used as the access ticket. In our example the initial ticket instance is $state(5)$ which will be traded to $state(6)$ on a call to `inc`.

Now what remains is to provide `counter` itself with the second part of its postcondition; which is where the nested triples come into play, as we need a means to provide the contract to the returned function, without divulging the body of `counter`:

```
state(V) =def x1 ↦ V
val &(Int) : (Unit) : Int;
  x0 := requires true;
        ensures State(load x1) * result ↦ {State(V)}_{State(V + 1)};
        (Int) : (Unit) : Int →
        {
          var Int;
            x0 := 0;
            val &(Unit) : Int;
              x0 := requires State(V);
                    ensures State(V + 1);
                    (Unit) : Int →
                    {
                      x2 := (load x2) + 1;
                      x2
                    };
              x1 := x2;
              x0
        }
  val(Unit) : Int;
    x0 := call load x1(5);
    call x0()
```

Listing 41: The Specified Program in *Scala Core*

Using the nested triple the postcondition of `counter` states that the value returned is a pointer to some function body for which the given triple holds.

5.2.3 Invariant Extension

The frame rule of separation logic is what enables local reasoning; for our language it would be given in the following form:

$$\frac{\Gamma; \Delta \vdash \{P\}\, e\, \{Q\}}{\Gamma; \Delta \vdash \{P * R\}\, e\, \{Q * R\}}$$

Figure 110: A Rule for Assignment

When we apply this rule to a triple containing nested triples, it is clear that the assertion R is added only to the assertions in the outermost one:

$$\frac{\Gamma; \Delta \vdash \{P * \{R\}\, e_1\, \{S\}\}\, e_2\, \{Q\}}{\Gamma; \Delta \vdash \{P * \{R\}\, e_1\, \{S\} * R'\}\, e_2\, \{Q * R'\}}$$

Figure 111: A Rule for Assignment

To enable local reasoning with functions – and in general for nested triples – we would like a rule which adds an assertion not only to the outermost triple, but to all this nested triples as well; something in the following shape:

$$\frac{\Gamma; \Delta \vdash P}{\Gamma; \Delta \vdash P \otimes Q}$$

Figure 112: A Rule for Assignment

This rule when applied to our example should have the following effect:

$$\frac{\Gamma; \Delta \vdash \{P * \{R\}\, e_1\, \{S\}\}\, e_2\, \{Q\}}{\Gamma; \Delta \vdash \{P * \{(R \otimes R') * R'\}\, e_1\, \{(S \otimes R') * R'\} * R'\}\, e_2\, \{Q * R'\}}$$

Figure 113: A Rule for Assignment

The $\otimes$-operation, called invariant extension, which allows for this definition of a higher order frame rule is adapted from the work of Schwinghammer et al. [53]; following the axioms given in Fig. 114, it performs exactly the required distribution of the assertion R through any nested triples.

$$\begin{aligned}
P \circ R &= (P \otimes R) * R \\
(\forall t\ \alpha \bullet P) \otimes R &\Leftrightarrow (\forall t\ \alpha \bullet (P \otimes R) \\
(\exists t\ \alpha \bullet P) \otimes R &\Leftrightarrow (\exists t\ \alpha \bullet (P \otimes R) \\
(P \oplus A) \otimes R &\Leftrightarrow (P \otimes R) \oplus (Q \otimes R) \qquad \oplus \in \{\Rightarrow, \wedge, \vee, *, \mathrel{-\!\!*}\} \\
P \otimes R &\Leftrightarrow P \qquad P \text{ is } \textbf{true}, \textbf{false}, e_1 = e_2 \text{ or } e_1 \mapsto e_2 \\
(P \otimes R) \otimes R' &\Leftrightarrow P \otimes (R \circ R') \\
\{P\}e\{Q\} \otimes R &\Leftrightarrow \{P \circ R\}e\{Q \circ R\}
\end{aligned}$$

Figure 114: Axioms for Invariant Extension

5.2.4 Recursive Predicates

As demonstrated in Section 2.3 with a predicate for lists, it benefits us to support recursive predicates. Often these predicates are defined inductively and the proof rules for the logic are equipped with an *open* and *close* rule.

Another approach to these predicates, without explicit open and close rules is to define them using the solution to the recursive equation described by the recursive definition. For instance, the recursively defined predicate in Fig. 115 gives rise to the equation in Fig. 116; what remains is to find the solution.

$$LeqTwo(2) \stackrel{def}{=} \textbf{true}$$
$$LeqTwo(x) \stackrel{def}{=} LeqTwo(x+1)$$

Figure 115: recursive Defined Predicate

$$LeqTwo = \lambda x.x = 2 \vee LeqTwo(x+1)$$

Figure 116: Recursive Equation Belonging to Fig. 115

In logics with a least fixed point operation μ, solutions to these equations are easily provided in the form $LeqTwo = \mu X(x).x = 2 \vee X(x+1)$. An example of such a logic is higher order separation logic where μ can be defined syntactically as follows[9]:

For an arbitrary predicate $q = \phi(q)$, with q only occurring positively in ϕ:

$$\mu q.\phi(q) = (\phi(q) \Rightarrow q) \Rightarrow q$$

Figure 117: Syntactic Definition of the Least Fixed Point

This defines the least fixed point as both $\phi(\mu q.\phi(q) \Rightarrow \mu q.\phi(q))$ and $\forall p.(\phi(p) \Rightarrow p) \Rightarrow (\mu p.\phi(q) \Rightarrow p)$ hold in the logic.

However, instead of defining the least fixed point syntactically, we will take an semantic approach, as used by Birkedal et al. [10], where the existence of least fixed points is given in the underlying semantic representation of the logic.

5.2.5 Permissions

As shown in Section 2.3, permissions are a powerful addition to the logic for use in concurrent scenarios. As we will want to specify the concurrent programs in *Scala Core*, our logic will be extended with permissions. As opposed to the common [48] approach of storing permissions in the heap, our permissions will be stored in an additional data structure called a permission table.

5.3 Typing *Scala Core* with Basic Expressions & Functions

A first step in proving the correctness of *Scala Core* programs, is to make sure they are well-typed. In this section we will present a Kripke-style semantics for types in *Scala Core* without classes, along with the necessary typing judgements. The approach used here to define the semantics of our types, will also serve as the introduction to a general approach to these Kripke-style semantics, as described by Birkedal et al. [10], which, will serve as the basis of our separation logic semantics in Section 5.4.

5.3.1 Semantics of Types

Semantically, types can be seen subsets of the closed values of a language defined by predicates on the set; for instance the type *Nat* of natural numbers can be defined as all the values that are elements of $\mathbb{N}$, i.e. the set $\{v \mid v \in \mathbb{V} \wedge v \in \mathbb{N}\}$. However, when a language contains reference types such as *Scala Core* does, this approach needs refinement, as the

predicate is dependent on existing types, e.g. the type **ref** Nat of natural numbers consists of all values which are references to values that are natural numbers, making it dependent on knowing which values are natural numbers. Following Birkedal et al. [10] we therefore define semantic types not as the obvious $T = Pred(V)$ but as follows:

$$W = \mathbb{N} \to_{mfin} T$$
$$T = W \to_{mon} Pred(Value)$$

Figure 118: Recursive Definition of Semantic Types

We define worlds W as partial functions from addresses to types. Intuitively, these worlds will provide additional information to the heap, by containing the type of the value at a given address. Our semantic types are still predicates on values, but now parameterized on worlds. Our example type Ref Nat(w) can now be defined as all the values, which are references, to some address, in worlds where that address has type Nat, i.e. the set $\{\text{Ref}\, a \mid w(a) = \text{Nat}\}$.

With this, our semantic model of types is given by a Kripke model over a recursively-defined set of worlds. Unfortunately, due to cardinality reasons, the solution to the above equations describing our model does not exist in the category of sets [2]. Commonly, to address this issue, methods are used which are based on step-indexing which solve approximate versions of the equations [2, 6], which – as demonstrated by Hobor, Dockins, and Appel [31] – are often sufficient for practical purposes. However, we will use the approach used by Birkedal et al. [10], which provides a recipe to transform the sets in the equations to objects in the Cartesian closed category $CBUlt_{ne}$ of complete, 1-bounded, non-empty, ultrametric spaces and non-expansive functions between them and solve the equations in that category:

Since we have defined the semantics of *Scala Core* as a small step operational semantics, the first step consists of defining V, which will act as the domain for semantic values, as the set of closed syntactic values from the operational semantics.

In the second step we define an object $Pred(V)$ in $CBUlt_{ne}$, which will represent predicates on values. Due to the operational nature of our semantics, we take $UPred(V)$ for $Pred(V)$, which consists of the set of predicates on step-approximated values [10], where the step numbers coincide with the reduction steps taken in the operational semantics:

$$UPred(V) = \{p \subseteq \mathbb{N} \times V \mid \forall (n, v) \in p. \forall m \leq n. (m, v) \in p\}$$

Figure 119: Definition of $UPred(V)$

This set can always be made into a well-defined object $(UPred(V), d)$ in $CBUlt_{ne}$, where the elements represent predicates on values, by providing the following distance function d:

$$d(p, q) = \begin{cases} 2^{-max\{m \mid p_{[m]} = q_{[m]}\}} & \text{if } p \neq q \\ 0 & \text{otherwise.} \end{cases}$$

where $p_{[n]} = \{(m, v) \in p \mid m < n\}$ represents the m^{th} approximation of p.

Figure 120: Definition of the Distance Function for $UPred(V)$

Intuitively, this distance function measures to which level the predicates agree, e.g. the predicate p – describing values approximated to the eighth step – and the predicate q – describing values approximated to the tenth step – agree to the eighth step, so their distance is 8.

It is interesting to note that the construction of $UPred(V)$ does not depend on the choice of V; we will make use of this when defining contexts.

In the third step we obtain the solution $\widehat{T}$ by solving the following equation in $CBUlt_{ne}$:

$$\widehat{T} \cong \tfrac{1}{2} \cdot ((\mathbb{N} \rightharpoonup_{mfin} \widehat{T}) \rightarrow_{mon} UPred(V))$$

Figure 121: Solving the Recursive Equation

And in the fourth and final step we now define W and T using $\widehat{T}$.

$$\begin{aligned} T &= \tfrac{1}{2} \cdot W \rightarrow_{mon} UPred(V) \\ W &= \mathbb{N} \rightharpoonup_{mfin} \widehat{T} \end{aligned}$$

Figure 122: Solving the Recursive Equation

A striking feature here is there inclusing of the $\frac{1}{2}$: This shrinking factor is a standard technique to make sure the functor is locally contractive [3], which is required for the existence of recursive solutions.

With the solution to our equations we can now give an interpretation of $t \in$ Type, in program context k and world w:

$$\begin{aligned} [\![t]\!]_k &: \tfrac{1}{2} \cdot W \rightarrow_{mon} UPred(V) \\ [\![\text{Unit}]\!]_k &= \lambda w.\mathbb{N} \times \{\textbf{unit}\} \\ [\![\text{Bool}]\!]_k &= \lambda w.\mathbb{N} \times \{\textbf{true}, \textbf{false}\} \\ [\![\text{Int}]\!]_k &= \lambda w.\mathbb{N} \times \mathbb{Z} \\ [\![\text{Ref}\, t]\!]_k &= \lambda w.\{(n,a) \mid a \in dom(w) \wedge [\![t]\!] \overset{m}{=} w(a)\} \\ [\![(t) : t']\!]_k &= \lambda w.\{(n,v) \mid \forall m \leq n. \forall v' \in V. \forall w' \sqsupseteq w. (m,v') \in [\![t]\!]_k(w) \\ &\qquad \Rightarrow (m, \textbf{call}\, v(v')) \in \mathcal{E}[\![t']\!]_k(w')\} \\ \mathcal{E}[\![t]\!]_k &= \lambda w.\{(n,e) \mid \forall m \leq n. \forall h. \forall v.h :_k w \wedge v = [\![e]\!]_k^h \\ &\qquad \Rightarrow \exists w' \sqsupseteq w.(n-m,v) \in [\![t]\!]_k(w')\} \end{aligned}$$

Figure 123: Interpretation of Types

Interpretations of primitive types are trivially given by their step-approximated values.

The interpretation of reference types however, makes use of the extra information available in the given world, consisting of all step-approximated reference values of which the type, of the referenced value, agrees.

Finally function types of $(t) : t'$ are interpreted as all step-approximated values, where the function call expression, of such a value, with an argument of v' type t, evaluates to a value of type t'.

Using different sets for V, namely the set of worlds and the set of stacks, we can define contexts Γ of typed references and Δ of typed variables:

$$
\begin{aligned}
[\![\Gamma]\!]_k &: \tfrac{1}{2} \cdot W \to_{mon} UPred(W) \\
[\![\Gamma]\!]_k &= \{(n, w) \mid \forall (e, t) \in \Gamma.(n, e) \in \mathcal{E}[\![\mathrm{Ref}\, t]\!]_k(w)\} \\
\\
[\![\Delta]\!]_k &: \tfrac{1}{2} \cdot W \to_{mon} UPred(Stack) \\
[\![\emptyset]\!]_k &= \lambda w.\mathbb{N} \times \emptyset \\
[\![\Delta, i : t]\!]_k &= \lambda w.\{(n, \rho[i \mapsto v]) \mid (n, v) \in [\![t]\!]_k(w) \wedge (n, \rho) \in [\![\Delta]\!]_k(w)\}
\end{aligned}
$$

Figure 124: Interpretation of Contexts

The context Γ – of typed references – is defined as all pairs of an Expression e and a type t where the value, resulting from the evaluation of e, corresponds to the type $\mathrm{Ref}\, t$, and interpreted as the step-approximation of w.

The context Δ – of typed variables – is inductively defined as all pairs of an index i and a type t, where the value v, at position i in the stack, is of type t, and is interpreted as the step-approximation of ρ. This context is the reason for the parameterization, of the type interpretations, by context k, as we require it to determine the stack ρ.

$$
\Gamma; \Delta \vdash e : t \overset{def}{\Leftrightarrow} \forall n \geq 0. \forall k. \forall w. (n, w) \in [\![\Gamma]\!]_k \wedge (n, \rho) \in [\![\Delta]\!]_k(w) \Rightarrow (n, e) \in \mathcal{E}[\![t]\!]_k(w)
$$

Figure 125: Definition of Well-Typed Expressions

An expression e in the language is now classified as well-typed with type t, when, for all approximations of worlds and contexts, the approximated world is contained in the interpretation of Γ, the approximated stack is contained in the interpretation Δ and the approximated expression is contained in the interpretation of $\mathcal{E}[\![t]\!]_k(w)$, i.e. the given contexts give rise to a world and program context where e evaluates to a value with type t.

5.3.2 Type Judgements

Using our previous definitions, we now provide the following type judgments:

$$\textbf{T-Var}\ \frac{\Gamma\uparrow, x_0 : t_2; \Delta\uparrow, 0 : \text{Ref}\, t_2 \vdash e : t_1}{\Gamma; \Delta \vdash \textbf{var}\, t_2; e : t_1} \qquad \textbf{T-Seq}\ \frac{\Gamma; \Delta \vdash e_1 : t_1 \quad \Gamma; \Delta \vdash e_2 : t_2}{\Gamma; \Delta \vdash e_1; e_2 : t_2} \qquad \textbf{T-Eq}\ \frac{\Gamma; \Delta \vdash e_1 : t \quad \Gamma; \Delta \vdash e_2 : t}{\Gamma; \Delta \vdash e_1 = e_2 : \text{Bool}}$$

$$\textbf{T-Assign}\ \frac{\Gamma; \Delta \vdash e_1 : \text{Ref}\, t \quad \Gamma; \Delta \vdash e_2 : t}{\Gamma; \Delta \vdash e_1 := e_2 : \text{Unit}} \qquad \textbf{T-Load}\ \frac{\Gamma; \Delta \vdash e : \text{Ref}\, t}{\Gamma; \Delta \vdash \textbf{load}\, e : t}$$

$$\textbf{T-If}\ \frac{\Gamma; \Delta \vdash e_1 : \text{Bool} \quad \Gamma; \Delta \vdash e_2 : t \quad \Gamma; \Delta \vdash e_3 : t}{\Gamma; \Delta \vdash \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 : t} \qquad \textbf{T-Fundef}\ \frac{\Gamma\uparrow, x_0 : t_1; \Delta\uparrow, 0 : \text{Ref}\, t_1 \vdash e : t_2}{\Gamma; \Delta \vdash (t_1) : t_2 \to \{e\} : \text{Ref}(t_1) : t_2}$$

$$\textbf{T-Funcall}\ \frac{\Gamma; \Delta, \text{E} \vdash e_1 : \text{Ref}(t_1) : t_2 \quad \Gamma; \Delta \vdash e_2 : t_1}{\Gamma; \Delta, \text{E} \vdash \textbf{call}(e_1) : e_2 : t_2} \qquad \textbf{T-True}\ \frac{}{\Gamma; \Delta \vdash \textbf{true} : \text{Bool}} \qquad \textbf{T-False}\ \frac{}{\Gamma; \Delta \vdash \textbf{false} : \text{Bool}}$$

$$\textbf{T-Unit}\ \frac{}{\Gamma; \Delta \vdash \textbf{unit} : \text{Unit}} \qquad \textbf{T-Int}\ \frac{}{\Gamma; \Delta \vdash \textbf{int}\, n : \text{Int}}$$

Figure 126: Type Judgements

The judgements have the following meaning:

- **T_Var**: A variable declaration expression adds a variable of type t to the heap and pushes a reference to it on the stack. This rule therefore, on encountering a variable declaration expression with a variable of type t_2 and a body of type t_1, adds the reference to Δ and the heap location and type to Γ. The type of a variable declaration expression is the type of its body.
- **T_Seq**: The type of a sequential expression is the type of the second subexpression, assuming both subexpressions are well-typed.
- **T_Eq**: The type of an equivalence expression is Bool, assuming both subexpressions have the same type.
- **T_Assign**: The type of an assignment is always Unit and the second subexpression must be assignable to the first, i.e. the first should be a reference of the type of the second.
- **T_Load**: The type of a load expression is t, assuming the subexpression is a reference of type t.
- **T_If**: The type of an if expression is t, assuming the first subexpression – the condition – is of type Bool and the other two are both of type t.
- **T_Fundef**: The type of a function definition is a reference type of the function being defined. The body of the function should have a type equal to the return type of the function. The function parameter is added to the contexts for evaluation of the body type. We split the stack-environment, as the function will always be executed at the defining site with a stack smaller or equal to that of the calling site.
- **T_Funcall**: The type of a function call is the return type of the function being called. The expression being called should be a function reference type and the parameter type should match the function signature.

5.4 *Separation Logic for Scala Core* with Basic Expressions & Functions

Now that expressions can be typed, we will focus our attention on proving the correctness of *Scala Core* programs using a separation logic. We will start with the syntax of assertions in our language, and then provide the Kripke-style semantics in a similar fashion as before.

5.4.1 Assertion Language Syntax

To define assertions and their relation to the program, we add the following syntactical constructs:

Assertion ::= **true** | **false** | Assertion ∨ Assertion | Assertion ∧ Assertion
| Expression = Expression | Expression ≤ Expression | Assertion ⇒ Assertion
| ∀ LVar: Type . Assertion | ∃ LVar: Type . Assertion | Expression ↦ Expression
| Assertion ∗ Assertion | Assertion −∗ Assertion
| {Assertion} Expression {Assertion} | Assertion ⊗ Assertion | Predicate
Expression ::= ... | **requires** Assertion; **ensures** Assertion; FType → {Expression} | ...

Figure 127: Assertions in *Scala Core*

The assertion syntax consists of the familiar operators from first-order logic and permission-based separation logic extended with the prevously discussed invariant extension operator $P \otimes Q$, (nested) Hoare triples of the form $\{P\}e\{Q\}$ and with recursive predicates of the form $(\mu\alpha.\lambda\beta.P)(e)$. We will use P, Q, R and S to identify assertions and α, β, γ to identify logical terms.

5.4.2 Meaning of Assertions

Where, before, semantic types were described as subsets of the closed values of a language, defined by predicates on the set, semantic assertions for a separation logic can be seen as subheaps, defined by predicates on the set of heaps.

Using the approach outlined in Section 5.3.1, we define Kripke-model over a recursively defined set of worlds:

$$W = \mathbb{N} \to_{mfin} A$$
$$A = W \to_{mon} Pred(Heap)$$

Figure 128: Recursive Definition of Semantic Assertions

Once again, these worlds will provide additional information to the heap, but in this case worlds $w \in W$ describe invariants – for instance, accompanying stored functions – which all possible computations should preserve. For classical, or Boolean, BI-algebra, the construction would be identitical to the one in Section 5.3.1 and we could simply substitute Heap for the set of closed values to obtain a solution. However, as we are interested in an intuistionistic, or Heyting, BI-algebra, we use a variation on this approach similar to the one by Birkedal et al. [10, section 3.2]:

Recall the set $UPred(V)$ of uniform predicates on values:

$$UPred(V) = \{p \subseteq \mathbb{N} \times V \mid \forall (n, v) \in p. \forall m \leq n. (m, v) \in p\}$$

For a poset $(A, \sqsubseteq)$, the set $UPred\uparrow(A)$ of upward closed uniform predicates is defined as:

$$UPred\uparrow(A) = \{p \subseteq \mathbb{N} \times V \mid \forall (n, a) \in p. \forall a \sqsubseteq b. (n, b) \in p\}$$

Figure 129: Definition of $UPred\uparrow(A)$

From this point on we follow the same approach as in Section 5.3.1, but with $UPred\uparrow$(Heap) instead of $UPred(V)$ and the following partial order on Heap:

$$h_1 \sqsubseteq h_2 \Leftrightarrow \mathrm{dom}(h_1) \supseteq \mathrm{dom}(h_2) \wedge \forall l \in \mathrm{dom}(h_2).h_1(l) = h_2(l)$$

Figure 130: Intuitionistic Ordering of Heaps

With *UPred*↑(Heap) made into an object in $CBUlt_{ne}$ by using the same distance function as before, we can now obtain a solution to our recursive equation in $CBUlt_{ne}$:

$$A = \frac{1}{2} \cdot W \rightarrow_{mon} \mathit{UPred}\uparrow(\mathrm{Heap})$$
$$W = \mathbb{N} \rightarrow_{mfin} \widehat{A}$$

Figure 131: Solution the Recursive Equation

Since worlds represent invariants which speak of functions and are thus themselves world-dependent, it is natural they must double as functions $W \rightarrow_{mon} \mathit{UPred}(\mathrm{Heap})$. Consequently, the solution can be simplified to the following:

$$W = \frac{1}{2}(W \rightarrow \mathit{UPred}\uparrow(\mathrm{Heap}))$$

Figure 132: Simplified Solution to the Recursive Equation

Following Birkedal et al. [10], using this solution we define *Pred* as $\mathit{Pred} = \frac{1}{2}(W \rightarrow \mathit{UPred}\uparrow(\mathrm{Heap}))$ with isomorphism $i : \mathit{Pred} \rightarrow W$. We will model assertions as elements of Pred. For the interpretation of the standard assertions in intuitionistic separation logic we will define a complete Heyting BI-algebra on *Pred*:

First we provide a pointwise ordering of the elements in Pred:

$$\forall p, q \in Pred.p \leq q \Leftrightarrow \forall w \in W.p(w) \subseteq q(w)$$

Figure 133: Pointwise Ordering of Elements in $Pred$

Then we define the unit element of the algebra, which coincides with the top element of the algebra, due to our choice of an intuitionistic logic:

$$\mathcal{I} = \mathbb{N} \times \mathrm{Heap} = \top$$

Figure 134: Unit

Finally we define the required operations:

$$p * q = \lambda w.\{(n,h) \mid \exists h_1, h_2.h_1 \cdot h_2 = h \wedge (n,h_1) \in p(w) \wedge (n,h_2) \in q(w)\}$$
$$p \mathrel{-\!\!*} q = \lambda w.\{(n,h) \mid \forall m \leq n.((m,h') \in p(w) \wedge h \# h') \Rightarrow (m, h \cdot h') \in q(w)\}$$
$$p \Rightarrow q = \lambda w.\{(n,h) \mid \forall m \leq n.\forall h' \sqsubseteq h.(m,h') \in p(w) \Rightarrow (m,h') \in q(w)\}$$

Where $h_1 \# h_2$ denotes two heaps having disjoint domains and $h_1 \cdot h_2$ denotes their union.

Figure 135: Separation Logic Operations on $Pred$

With the previous ordering, operations, unit and *meet* and *join* given by set-intersection and set-union respectively, *Pred* is a complete Heyting BI-algebra[9], which provides us a sound interpretation of the basic elements of the logic:

$$
\begin{aligned}
[\![\mathbf{false}]\!]_{\eta,k} &= \lambda w.\emptyset \\
[\![\mathbf{true}]\!]_{\eta,k} &= \lambda w.\mathbb{N} \times \text{Heap} \\
[\![P \wedge Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \cap [\![Q]\!]_{\eta,k} w \\
[\![P \vee Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \cup [\![Q]\!]_{\eta,k} w \\
[\![P \Rightarrow Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \Rightarrow [\![Q]\!]_{\eta,k} w \\
[\![P * Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w * [\![Q]\!]_{\eta,k} w \\
[\![P \mathrel{-\!\!*} Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \mathrel{-\!\!*} [\![Q]\!]_{\eta,k} w
\end{aligned}
$$

Where $\eta \in$ Environment – as defined in Fig. 141 – is an environment of ghost variables and predicates and $k \in$ Context is the program context.

Figure 136: Interpretation of Basic Connectives

First of we extend these with quantifiers, equality and the *pointsto* operation:

$$
\begin{aligned}
[\![e_1 \mapsto e_2]\!]_{\eta,k} &= \lambda w. \begin{cases} \{(n, h[a \mapsto [\![\eta(e_2)]\!]^h_k]) \mid n \in \mathbb{N}, h \in \text{Heap}\} \\ \qquad \text{if } [\![\eta(e_1)]\!]^h_k = \mathbf{ptr}\, a \\ \emptyset \quad \text{otherwise.} \end{cases} \\
[\![e_1 = e_2]\!]_{\eta,k} &= \lambda w. \begin{cases} \mathbb{N} \times \text{Heap} & \text{if } [\![e_1]\!]_{\eta,k} = [\![e_2]\!]_{\eta,k} \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![\forall \alpha{:}t.P]\!]_{\eta,k} &= \lambda w.\ \cap_{v \in \{v \mid v \in \text{Value} \wedge \mathit{typeof}(v) = t\}} [\![P]\!]_{\eta[\alpha \mapsto v],k}(w) \\
[\![\exists \alpha{:}t.P]\!]_{\eta,k} &= \lambda w.\ \cup_{v \in \{v \mid v \in \text{Value} \wedge \mathit{typeof}(v) = t\}} [\![P]\!]_{\eta[\alpha \mapsto v],k}(w)
\end{aligned}
$$

Where $\eta \in$ Environment – as defined in Fig. 141 – is an environment of ghost variables and predicates and $k \in$ Context is the program context.

Figure 137: Interpretation of Basic Connectives

Quantification is typed and makes use of an environment mapping logical variables to expressions. The interpretation of a quantifier is then given by the union or disjunction of the quantified set.

Equality is determined simply by expressions reducing to the same value, and in that case given the interpretation of **true**.

For the *pointsto* operation, we use capture-avoiding substitution of the logical – or ghost – variables with their values and associate the heap in which the resulting pointer points to a value as the interpretation.

What remains are the more interesting cases of the interpretation of Hoare triples, recursive predicates and invariant extension:

For invariant extension we need to define the operator $\otimes$ on the set of semantic predicates *Pred*. Following Birkedal et al. [10], we do so by providing the recursive equation it should satisfy:

There exists a unique function $\otimes : Pred \times W \to Pred$ in $CBUlt_{ne}$ satisfying:

$$p \otimes w = \lambda w'.p(w \circ w')$$

where $\circ : W \to W \times W$ is given by:

$$w_1 \circ w_2 = i((i^{-1}(w_1) \otimes w_2) * i^{-1}(w_2))$$

with the basic properties:

1. $(W, \circ, true)$ is a monoid.
2. The operator is a monoid action of W on *Pred*: for all $P \in Pred$ we have $P \otimes true = P$ and $(P \otimes w_1) \otimes w_2 = P \otimes (w_1 \circ w_2)$

where $true = i(\lambda w.\mathcal{I})$, i.e. the image of the BI-unit.

Figure 138: Recursive Equation for Invariant Extension

Proof of the existence of this operator can be obtained by application of Banach's fixed point theorem[53], which guarantees the existence and unicity of fixed points in certain self-maps of metric spaces, provided the mapping is contractive.

As triples can be nested, the interpretation of triples and assertions will be given simultaneously, using semantic triples:

To define semantic triples, we again follow Birkedal et al. [10] and define $Safe_m$ as the set of configurations which are safe for m steps and we write $\twoheadrightarrow^k$ as the k-step reduction relation, in the operational semantics. Now we can define $\vDash_n$:

$w, k \vDash_n (p, e, q)$ iff: For all $r \in Pred$, all $m < n$ and all $h \in$ Heap if $(m, h) \in p(w) * i^{-1}(w)(true) * r$ then:

1. $\mathbf{S}(k, (\searrow e), h) \in Safe_m$.
2. For all $j < m$, $v \in$ Value and all $h \in$ Heap if $\mathbf{S}(k, (e), h) \twoheadrightarrow^k_{ct} \mathbf{S}(k, (\nearrow v), h')$, then $(m - k, h') \in q[v/\mathbf{result}](w) * i^{-1}(w)(true) * r$

Figure 139: Semantic Hoare Triples

The semantic triples bake in the first-order frame property, by joining the invariant r. As this would otherwise result in endless recursion, we close the loop by applying the world w, on which the triple implicitly depends, to the unit $true$. We substitute the value of the expression for **result** in the postcondition. Step-indexing is used to determine to what extent pre- and postconditions should hold. Intuitively these semantic triples corresponds to the standard interpretation of a hoare triple: in a state where p holds ($(m, h) \in p(w) * i^{-1}(w)(true) * r$), the evaluation of the expression is possible ($\mathbf{S}(k, (\searrow e), h) \in Safe_m$) and results in a state where q should hold (if $\mathbf{S}(k, (e), h) \twoheadrightarrow^k_{ct} \mathbf{S}(k, (\nearrow v), h')$, then $(m - k, h') \in q[v/\mathbf{result}](w) * i^{-1}(w)(true) * r$).

Hoare triples can now be interpreted using the given semantic construct and the interpretation of an assertion $\Gamma; \Delta \vdash P$ is defined to be an element in $[\![P]\!] \in Pred$.

The one thing that remains is the interpretation of recursively defined predicates, which is, following [10, 53], defined using Banach's fixpoint theorem:

Let I be a set and suppose that, for every $i \in I$, $F_i : Pred^I \to Pred$ is a contractive function. There then exists a unique $\vec{p} = (p_i)_{i \in I} \in Pred$ such that $F_i(\vec{p}) = p_i$, forall $i \in I$.

Figure 140: Interpretation of Recursive Predicates

The interpretation of the connectives are now, in summary, the following:

$$
\begin{aligned}
\text{Environment} &= \text{Ghost} \mid \text{RecPred} \\
\text{Ghost} &= \text{LVar} \rightarrow_{mfin} \text{Value} \\
\text{RecPred} &= \text{LVar} \rightarrow_{mfin} (\text{Value} \rightarrow \text{Assertion})
\end{aligned}
$$

Figure 141: Environment

$$
\begin{aligned}
[\![\mathbf{false}]\!]_{\eta,k} &= \lambda w.\emptyset \\
[\![\mathbf{true}]\!]_{\eta,k} &= \lambda w.\mathbb{N} \times \text{Heap} \\
[\![P \wedge Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \cap [\![Q]\!]_{\eta,k} w \\
[\![P \vee Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \cup [\![Q]\!]_{\eta,k} w \\
[\![P \Rightarrow Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \Rightarrow [\![Q]\!]_{\eta,k} w \\
[\![P * Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w * [\![Q]\!]_{\eta,k} w \\
[\![P \mathrel{-\!\!*} Q]\!]_{\eta,k} &= \lambda w.[\![P]\!]_{\eta,k} w \mathrel{-\!\!*} [\![Q]\!]_{\eta,k} w \\
[\![P \otimes Q]\!]_{\eta,k} &= \lambda w.\left([\![P]\!]_{\eta,k} \otimes i([\![Q]\!]_{\eta,k})\right) w \\
[\![e_1 \mapsto e_2]\!]_{\eta,k} &= \lambda w.\begin{cases} \{(n, h[a \mapsto [\![\eta(e_2)]\!]^h_k]) \mid n \in \mathbb{N}, h \in \text{Heap}\} \\ \qquad \text{if } [\![\eta(e_1)]\!]^h_k = \mathbf{ptr}\, a \\ \emptyset \quad \text{otherwise.} \end{cases} \\
[\![e_1 = e_2]\!]_{\eta,k} &= \lambda w.\begin{cases} \mathbb{N} \times \text{Heap} & \text{if } [\![e_1]\!]_{\eta,k} = [\![e_2]\!]_{\eta,k} \\ \emptyset & \text{otherwise.} \end{cases} \\
[\![\{P\}\, e\, \{Q\}]\!]_{\eta,k} &= \lambda w.\{(n, h) \mid k, w \vDash_n ([\![P]\!]_{\eta,k}, e, [\![Q]\!]_{\eta,k})\} \\
[\![\forall \alpha{:}t.P]\!]_{\eta,k} &= \lambda w.\, \cap_{v \in \{v \mid v \in \text{Value} \wedge \mathit{typeof}(v) = t\}} [\![P]\!]_{\eta[\alpha \mapsto v],k}(w) \\
[\![\exists \alpha{:}t.P]\!]_{\eta,k} &= \lambda w.\, \cup_{v \in \{v \mid v \in \text{Value} \wedge \mathit{typeof}(v) = t\}} [\![P]\!]_{\eta[\alpha \mapsto v],k}(w) \\
[\![(\mu\alpha.\lambda\beta.P)(e)]\!]_{\eta,k} &= \mathit{fix}\left(\lambda P_1^{\text{Value} \to \text{Assertion}} \lambda v.[\![P]\!]_{\eta[\alpha \mapsto P_1, \beta \mapsto v],k}\right)([\![\eta(e_1)]\!]^h_k)
\end{aligned}
$$

Where $\eta \in$ Environment – as defined in Fig. 141 – is an environment of ghost variables and predicates and $k \in$ Context is the program context.

Figure 142: Interpretation of Connectives

Furthermore we define interpretations of the contexts Δ and Γ as follows:

$$
\begin{aligned}
\Gamma &= \lambda w.\mathbb{N} \times \text{LVar} \\
\Delta &= \lambda w.\mathbb{N} \times \text{Heap}
\end{aligned}
$$

Figure 143: Interpretation of Contexts

The context Γ contains the names of ghost variables and predicates, whereas Δ contains stack indices. These contexts will be used for book keeping which is required for the soundness argument.

5.4.3 Hoare Rules

Having defined the formulas in our logic, we will now provide the proof rules used to establish the validity of *Scala Core* programs:

$$\frac{\Gamma; \Delta \vdash P' \Rightarrow P \quad \Gamma; \Delta \vdash Q \Rightarrow Q'}{\Gamma; \Delta \vdash \{P\}\, e\, \{Q\} \Rightarrow \{P'\}\, e\, \{Q'\}}$$

Figure 144: Consequence Rule

To emphasize having Hoare triples as assertions, we give a slightly modified formulation of the standard consequence rule – for strengthening preconditions and weakening postconditions – which instead of putting $\{P\}\, e\, \{Q\}$ in the premises and $\{P'\}\, e\, \{Q'\}$ in the conclusion, has an implication between the triples in the conclusion.

$$\frac{}{\Gamma; \Delta \vdash \{P\}\, e\, \{Q\} \Rightarrow \{P * R\}\, e\, \{Q * R\}}$$

Figure 145: First-Order Frame Rule

The first-order frame rule is just the standard frame rule for separation logic – previously seen in Section 2.3.1 – but once again making use of the fact that our Hoare triples are first class assertions. This rule will not suffice for nested triples, as it adds the invariant to only the outermost triple. To deal with the nested case, we require the addition of a higher-order frame rule, as decribed by Schwinghammer et al. [53]:

$$\frac{\Gamma; \Delta \vdash P}{\Gamma; \Delta \vdash P \otimes Q}$$

Figure 146: Higher-Order Frame Rule

$$\begin{aligned}
P \circ R &= (P \otimes R) * R \\
(\forall t\ \alpha \bullet P) \otimes R &\Leftrightarrow (\forall t\ \alpha \bullet (P \otimes R) \\
(\exists t\ \alpha \bullet P) \otimes R &\Leftrightarrow (\exists t\ \alpha \bullet (P \otimes R) \\
(P \oplus A) \otimes R &\Leftrightarrow (P \otimes R) \oplus (Q \otimes R) \qquad \oplus \in \{\Rightarrow, \wedge, \vee, *, \mathrel{-\!\!*}\ \} \\
P \otimes R &\Leftrightarrow P \qquad P \text{ is } \textbf{true}, \textbf{false}, e_1 = e_2 \text{ or } e_1 \mapsto e_2 \\
(P \otimes R) \otimes R' &\Leftrightarrow P \otimes (R \circ R') \\
\{P\}e\{Q\} \otimes R &\Leftrightarrow \{P \circ R\}e\{Q \circ R\}
\end{aligned}$$

Figure 147: Axioms for Invariant Extension

With the higher-order frame rule, we allow the invariant R not only to be added to the outermost triple, but all nested triples inside. It is formulated using the invariant extension operator, which distributes the invariant R through the formula using the axioms previously shown in Fig. 114, which we repeat in Fig. 147 for the sake of clarity.

We define the following rules for the basic expressions in the language:

$$\frac{\Gamma; \Delta\uparrow, 0 \vdash \{x_0 \mapsto \_ * P\uparrow * \mathit{mutable}(x_0)\}\, e\, \{x_0 \mapsto \_ * Q\uparrow\}}{\Gamma; \Delta \vdash \{P\}\, \mathbf{var}\, t; e\, \{Q\}}$$

Figure 148: Variable Declaration

The first rule we provide is for the declaration of variables. This axiom makes use of a lifting operator ↑ over assertions, which increments the indices of the variables contains within the assertion by one i.e. it replaces all occurrences of x_i with x_{i+1}. A variable declaration expression then is correct, given the correctness of its body e in a state, where the original stack is lifted and a top element exists, i.e. where a variable has been pushed on the stack.

$$\frac{\Gamma; \Delta \vdash \{P\}\, e\, \{Q * v_1 = \mathbf{result} * v_1 \mapsto v_2\}}{\Gamma; \Delta \vdash \{P * v_1 \mapsto v_2\}\, \mathbf{load}\, e\, \{Q * v_2 = \mathbf{result} * v_1 \mapsto v_2\}}$$

Figure 149: Load

The next rule is for the load expression. In this case, a load expression is correct when the expression e evaluates to a valid pointer and the value, residing at the address of the pointer, is equal to the resulting value of the expression.

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash \{P\}\, e_1\, \{P' * v_1 = \mathbf{result} * v_1 \mapsto \_\} \\ \Gamma; \Delta \vdash \{P'\}\, e_2\, \{Q * \mathbf{result} = v_2\}\end{array}}{\Gamma; \Delta \vdash \{P * v_1 \mapsto \_\}\, e_1 := e_2\, \{Q * v_1 \mapsto v_2 * \mathbf{result} = \mathbf{unit}\}}$$

Figure 150: Assignment

Next there is the assignment expression, which is correct when the left hand expression evaluates to a pointer, the right hand expression evaluates to a value.

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash \{P\}\, e_1\, \{P'\} \\ \Gamma; \Delta \vdash \{P'\}\, e_2\, \{Q\}\end{array}}{\Gamma; \Delta \vdash \{P\}\, e_1; e_2\, \{Q\}}$$

Figure 151: Composition

Finally there is the straightforward rule for sequential expressions.

We now focus our attention on functions, starting with their definition:

$$\frac{}{\Gamma; \Delta \vdash \{P\}\, \mathbf{requires}\, R; \mathbf{ensures}\, S; t_1 : t_2 \to e\, \{Q * v_1 \mapsto \{R\} \_ \{S\} * v_1 = \mathbf{result}\}}$$

Figure 152: Function Definition

A function definition is correct when the returned value of the expression is a pointer to the allocated function body.

$$\frac{\Gamma;\Delta,\mathrm{E}\vdash\{P\}\,e_1\,\{P'*v_1=\textbf{result}*v_1\mapsto\{R\}\_\{S\}\}\quad\Gamma;\Delta,\mathrm{E}\vdash\{P'\}\,e_2\,\{P''*v_2=\textbf{result}\}\quad\forall v_2:t\bullet\Gamma;\Delta,0\vdash\{R\}\,e_3\,\{S\}\Rightarrow\{P''*x_0\mapsto v_2\}\,e_3\,\{Q*x_0\mapsto v_2\}}{\Gamma;\Delta,\mathrm{E}\vdash\{P*v_1\mapsto\{R\}\_\{S\}\}\,\textbf{call}\,e_1(e_2)\,\{Q\}}$$

Figure 153: Function Call

A function call is correct when e_1 is a valid function pointer, to an expression, with a contract. Secondly, the argument – we assume a single argument for the sake of legibility – evaluates to a value v_2. Finally the contract of the function, applied to the body, should imply the contract of the call.

5.5 Expanding *Separation Logic for Scala Core* to Exceptions

To support exceptions, as demonstrated with *Scala Core* in Section 4.5, we will take the usual approach and modify the notation ofour Hoare triples to take into account exceptional post-conditions – the definition for which is given in Fig. 156:

$$\frac{}{\Gamma;\Delta\vdash\{P\}\,e\,\{Q\mid\overrightarrow{ex_t}\Rightarrow\overrightarrow{R_t}\}\Rightarrow\{P*R\}\,e\,\{Q\mid\overrightarrow{ex_t}\Rightarrow\overrightarrow{R_t}*R\}}$$

Figure 154: First-Order Frame Rule with Exceptions

$$\frac{\Gamma;\Delta\vdash\{P\}\,e_1\,\{P'\mid\overrightarrow{ex_t}\Rightarrow\overrightarrow{R_t}\}\quad\Gamma;\Delta\vdash\{P'\}\,e_2\,\{Q\mid\overrightarrow{ex_t}\Rightarrow\overrightarrow{R_t}\}}{\Gamma;\Delta\vdash\{P\}\,e_1;e_2\,\{Q\mid\overrightarrow{ex_t}\Rightarrow\overrightarrow{R_t}\}}$$

Figure 155: Composition with Exceptions

As demonstrated with the previous examples in Fig. 154 and Fig. 155, most rules remain essentially the same, but now, if it terminates with an exception of type t, there is a matching exceptional postcondition that holds, instead of the non-exceptional one. We codify this in the semantics of the Hoare triple as follows:

$w,k\vDash_n(p,e,q,\overrightarrow{r_t})$ iff: For all $r\in Pred$, all $m<n$ and all $h\in$ Heap if $(m,h)\in p(w)*i^{-1}(w)(true)*r$ then:

1. $\mathbf{S}(k,(\searrow e),h)\in Safe_m$.
2. For all $j<m$, $v\in$ Value and all $h\in$ Heap if $\mathbf{S}(k,(e),h)\twoheadrightarrow^k_{ct}\mathbf{S}(k,(\nearrow v),h')$, then $(m-k,h')\in q[v/\textbf{result}](w)*i^{-1}(w)(true)*r$
3. For all $r_t\in\overrightarrow{r_t}$, $j<m$, $v\in$ Value and all $h\in$ Heap if $\mathbf{S}(k,(e),h)\twoheadrightarrow^k_{ct}\mathbf{S}(k,(\lightning_t v),h')$, then $(m-k,h')\in r_i[v/\textbf{result}](w)*i^{-1}(w)(true)*r$

$$[\![\{P\}\,e\,\{Q\mid\overrightarrow{ex_t}\Rightarrow\overrightarrow{R_t}\}]\!]_{\eta,k}=\lambda w.\{(n,h)\mid k,w\vDash_n([\![P]\!]_{\eta,k},e,[\![Q]\!]_{\eta,k},[\![\overrightarrow{R_t}]\!]_{\eta,k})\}$$

Figure 156: Semantic Hoare Triples for Exceptions

The definition remains largely similar to the one in Fig. 139, which the addition of point 3, which states that in case of

an exceptional end to an execution, the matching exceptional postcondition holds, instead of the non-exceptional one.

We also add a number of additional rules to deal with the exception expressions and modify the consequence rule to deal with weakening of exceptions:

$$\frac{\Gamma; \Delta \vdash P' \Rightarrow P \quad \Gamma; \Delta \vdash Q \Rightarrow Q' \quad \Gamma; \Delta \vdash \overrightarrow{R_t} \Rightarrow \overrightarrow{R'_t}}{\Gamma; \Delta \vdash \{P\}\, e\, \{Q \mid \overrightarrow{ex_t} \Rightarrow \overrightarrow{R_t}\} \Rightarrow \{P'\}\, e\, \{Q' \mid \overrightarrow{ex_t} \Rightarrow \overrightarrow{R'_t}\}}$$

Figure 157: Consequence Rule for Exceptions

$$\frac{\Gamma; \Delta \vdash \{P\}\, e_1\, \{Q \mid ex_t \Rightarrow R_t, \overrightarrow{ex_t} \setminus ex_t \Rightarrow \overrightarrow{R_t} \setminus R_t\} \quad \Gamma; \Delta \vdash \{R_t\}\, e_2\, \{Q \mid \overrightarrow{ex_t} \Rightarrow \overrightarrow{R_t}\}}{\Gamma; \Delta \vdash \{P\}\, \textbf{try}\, e_1\, \textbf{catch}\, t\, e_2\, \{Q \mid \overrightarrow{ex_t} \Rightarrow \overrightarrow{R_t}\}}$$

Figure 158: TryCatch Rule

$$\frac{}{\Gamma; \Delta \vdash \{P\}\, \textbf{throw}\, e\, \{\textbf{false} \mid ex_t \Rightarrow R_t\}}$$

Figure 159: Throw Rule

The support of exceptions will also require an extensive, but straightforward rework of the type system, as any expression can now also return an exception. As is this isn't relevant to the main goal of this section, we will not reproduce this work here.

5.6 Expanding *Separation Logic for Scala Core* to Classes & Traits

To deal with classes, we take a similar approach to Amighi et al. [5] in line with our previously defined operational semantics in Section 4. For use with our logic, methods in the class table are expanded with a pre- and postcondition, similarly to functions. The `mblookup` relation – for looking up method bodies, given their name and class type – is modified to reflect this:

$$\texttt{mbLookup Base}\ \frac{\textbf{class Any}\ \{...\ \textbf{Requires}\ P\ \textbf{Ensures}\ Q\, m(\overrightarrow{t}) : t\{e\}\ ...\}}{\texttt{mbLookup}(m, \textbf{Any}, \{P\}\, e\, \{Q\})}$$

$$\texttt{mbLookup Defining Class}\ \frac{\textbf{class}\ C\ \{...\ \textbf{Requires}\ P\ \textbf{Ensures}\ Q\, m(\overrightarrow{t}) : t\{e\}\ ...\}}{\texttt{mbLookup}(m, C, \{P\}\, e\, \{Q\})}$$

$$\texttt{mbLookup Defining Trait}\ \frac{\textbf{trait}\ C\ \{...\ \textbf{Requires}\ P\ \textbf{Ensures}\ Q\, m(\overrightarrow{t}) : t\{e\}\ ...\}}{\texttt{mbLookup}(m, C, \{P\}\, e\, \{Q\})}$$

$$\texttt{mbLookup Super Class}\ \frac{\begin{array}{c}\textbf{class}\ C_1\ \textbf{extends}\ C_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \\ m \notin \mathrm{dom}(\overline{m}) \\ \exists t \in (C_2 :: \overrightarrow{T}) \mid \texttt{mblookup}(m, t, \{P\}\, e\, \{Q\})\end{array}}{\texttt{mbLookup}(m, C_1, \{P\}\, e\, \{Q\})}$$

$$\texttt{mbLookup Super Trait}\ \frac{\begin{array}{c}\textbf{trait}\ T_1\ \textbf{extends}\ T_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \\ m \notin \mathrm{dom}(\overline{m}) \\ \exists T \in (T_2 :: \overrightarrow{T}) \mid \texttt{mblookup}(m, T, \{P\}\, e\, \{Q\})\end{array}}{\texttt{mbLookup}(m, C_1, \{P\}\, e\, \{Q\})}$$

Figure 160: The Modified `mblookup` Relation

Additionally the `mtlookup` relation – for looking up method types, given their name and class type – is added:

$$\texttt{mbLookup Base}\ \frac{\textbf{class Any}\ \{...\ \textbf{Requires}\ P\ \textbf{Ensures}\ Q\, m(\overrightarrow{t}) : t\{e\}\ ...\}}{\texttt{mbLookup}(m, \textbf{Any}, (\overrightarrow{t}) : t}$$

$$\texttt{mbLookup Defining Class}\ \frac{\textbf{class}\ C\ \{...\ \textbf{Requires}\ P\ \textbf{Ensures}\ Q\, m(\overrightarrow{t}) : t\{e\}\ ...\}}{\texttt{mbLookup}(m, C, (\overrightarrow{t}) : t}$$

$$\texttt{mbLookup Defining Trait}\ \frac{\textbf{trait}\ C\ \{...\ \textbf{Requires}\ P\ \textbf{Ensures}\ Q\, m(\overrightarrow{t}) : t\{e\}\ ...\}}{\texttt{mbLookup}(m, C, (\overrightarrow{t}) : t}$$

$$\texttt{mbLookup Super Class}\ \frac{\begin{array}{c}\textbf{class}\ C_1\ \textbf{extends}\ C_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \\ m \notin \mathrm{dom}(\overline{m}) \\ \exists t \in (C_2 :: \overrightarrow{T}) \mid \texttt{mblookup}(m, t, \{P\}\, e\, \{Q\})\end{array}}{\texttt{mbLookup}(m, C_1, (\overrightarrow{t}) : t}$$

$$\texttt{mbLookup Super Trait}\ \frac{\begin{array}{c}\textbf{trait}\ T_1\ \textbf{extends}\ T_2\ \textbf{with}\ \overrightarrow{T}\ \{\overline{f}\ \overline{m}\} \\ m \notin \mathrm{dom}(\overline{m}) \\ \exists T \in (T_2 :: \overrightarrow{T}) \mid \texttt{mblookup}(m, T, \{P\}\, e\, \{Q\})\end{array}}{\texttt{mbLookup}(m, C_1, (\overrightarrow{t}) : t}$$

Figure 161: The `mtlookup` Relation

To modify the type system to account for classes, primitive types are removed and replaced with classes with the appropriate syntactical sugar.

$$[\![t]\!]_k : \tfrac{1}{2} \cdot W \to_{mon} UPred(V)$$
$$\ldots$$
$$[\![\mathrm{C}]\!]_k = \lambda w.\mathbb{N} \times \{o\} \wedge o = \{\langle f, \_\rangle \mid \langle t, f\rangle \in flds \wedge fld(C, flds)\}$$
$$\ldots$$

Figure 162: Interpretation of Class Types

The interpretation of a class type, designated by its identifier, is given all step-approximated values, that match its fields.

Additional rules are added for creating classes, accessing their fields and calling methods:

$$\textbf{T-New}\ \frac{\Gamma;\Delta \vdash C \in \mathrm{ct} \quad \Gamma;\Delta \vdash e : t}{\Gamma;\Delta \vdash \textbf{new}\, C(e) : \mathrm{Ref}\, C} \qquad \textbf{T-Get}\ \frac{\mathtt{fld}(t, flds) \quad \langle t', f\rangle \in flds \quad \Gamma;\Delta \vdash e : \mathrm{Ref}\, t}{\Gamma;\Delta \vdash \textbf{load}\, e.f : t'}$$

$$\textbf{T-Set}\ \frac{\mathtt{fld}(t, flds) \quad \langle t', f\rangle \in flds \quad \Gamma;\Delta \vdash e_1 : \mathrm{Ref}\, t \quad \Gamma;\Delta \vdash e_1 : t'}{\Gamma;\Delta \vdash e_1.f := e_2 : Unit} \qquad \textbf{T-Methodcall}\ \frac{\mathtt{mtlookup}(m, t, (t') : t'') \quad \Gamma;\Delta \vdash e_1 : \mathrm{Ref}\, t \quad \Gamma;\Delta \vdash e_2 : t'}{\Gamma;\Delta \vdash \textbf{call}\, e_1.m(e_2) : t''}$$

Figure 163: Type Judgements for Classes

For the logic, we add 4 new Hoare rules:

$$\frac{\mathtt{typeof}(v_1, t) \quad \mathtt{fld}(t, flds) \quad f \in flds \quad \Gamma;\Delta \vdash \{P\}\, e\, \{P' * v_1 = \textbf{result}\}}{\Gamma;\Delta \vdash \{P * v_1.f \mapsto v_2\}\, \textbf{load}\, e.f\, \{Q * v_1 \mapsto v_2 * v_2 = \textbf{result}\}}$$

Figure 164: Get

Field access is correct when the field in question belongs to the class pointed to by the pointer to which e evaluates and the resulting value of the expression is the contents of the field.

$$\frac{\mathtt{typeof}(v_1, t_1) \quad \mathtt{typeof}(v_2, t_2) \quad t_2 :< t_1 \quad \mathtt{fld}(t_1, flds) \quad f \in flds \quad \Gamma;\Delta \vdash \{P\}\, e_1\, \{P' * v_1 = \textbf{result} * v_1 \mapsto \_\} \quad \Gamma;\Delta \vdash \{P'\}\, e_2\, \{P'' * v_2 = \textbf{result} * v_2 \mapsto \_\}}{\Gamma;\Delta \vdash \{P * v_1.f \mapsto \_\}\, e_1.f := e_2\, \{Q * v_1.f \mapsto v_2\}}$$

Figure 165: Set

When writing to fields, correctness is essentially expressed by a combination of the previous rules for reading fields and assignment: once again the field should belong to the referenced object and subexpression should evaluate to their proper values and types.

$$\frac{\texttt{typeof}(v_1, t_1) \quad \texttt{mblookup}(m, t_1, \{R\}e_3\{S\}) \quad \Gamma;\Delta \vdash \{P\}\, e_1\, \{P' * v_1 = \textbf{result}\} \quad \Gamma;\Delta \vdash \{P'\}\, e_2\, \{P'' * v_2 = \textbf{result}\} \quad \forall v_2 \bullet \Gamma;\Delta, 0 \vdash \{R\}\, e_3\, \{S\} \Rightarrow \{P'' * x_0 \mapsto v_1 * x_1 \mapsto v_2\}\, e_3\, \{Q * x_0 \mapsto v_1 * x_1 \mapsto v_2\}}{\Gamma;\Delta \vdash \{P * \{R\}\, e_3\, \{S\}\}\, \textbf{call}\, e_1.m(e_2)\, \{Q\}}$$

Figure 166: Method Call

Calling methods is similar to calling functions, with the addition of the correct evaluation to the receiver pointer and the fact that the first argument of the function is a pointer to the receiver.

$$\frac{C \in ct \quad \Gamma;\Delta \vdash \{v_1 \mapsto o\}\, \textbf{call ptr}\, v_1.init(e)\, \{Q\}}{\Gamma;\Delta \vdash \{true\}\, \textbf{new}\, C(e)\, \{Q * v_1 = \textbf{result} * v_1 \mapsto o * \Gamma v_1 = C * v_1.init\}}$$

Figure 167: New

The new expression is valid when the constructor evaluates correctly for the newly allocated object.

5.7 Expanding *Separation Logic for Scala Core* with Permissions

As with the model language in Section 4, our final expansion concerns multithreading. To assure race-freedom we use a permission-based approach as described in Section 2. To allow permissions in our logic, we take an approach similar to Amighi et al. [4] and embed them in resources:

$$\begin{aligned} \text{Resource} &= \text{Heap} \times \text{PermissionTable} \\ \text{PermissionTable} &= \text{Address} \times \text{FieldIdentifier} \cup \{\bot\} \rightharpoonup [0, 1] \end{aligned}$$

Figure 168: Resources & Permission Tables

Resources $\mathcal{R}$ range over Resource with the binary relation # and the binary operation $\cdot$ where # defines disjointness between resources and $\cdot$ denotes their union.

Permission tables are defined to carry the same $(\#, \cdot)$ structure as resources similary to heaps, by having # denote disjoint domains and having $\cdot$ denote the union by pointwise addition of permissions.

As both heaps and permission tables carry the same structure as resources, the operations on resources are defined by a pointwise lifting of its components. Similarly, the meaning of assertions can be ammended to resources, by simply substituting Resource for Heap in the definitions of Section 5.4.2.

With the embedding of permissions, an expansion to multithreading with fork/join and reentrant locks can be made in the same vein as the work by Amighi et al. [4].

5.8 Related Work

The work with the strongest relation to our own – and on which we have based a lot of our work – is that of Schwinghammer et al. [53] and Birkedal et al. [10]. The main differences reside in our attempt to match the work to *Scala* and the fact that instead of describing general code stored on the heap, our approach is restricted to functions

as defined in Section 4.4. Another approach would be to based our work on a higher order separation logic [9], which allows to express the higher order nature of our functions, but along with this expressiveness, brings even more difficulties with regards to the computational complexity of the eventual implementation in a tool.

While our approach, following Birkedal et al. [10] is complicated and highly abstract, earlier simpler approaches [18, 51] often came with downsides, such as complications with parameters of stored functions or deep nesting.

With regard to classes, a lot of our approach is based on the work of Amighi et al. [4], which makes use of a predefined class-table, which matches our program semantics in Section 4. An alternative, would be to integrate the type system in the formalization, as seen in the work of Cremet et al. [20]. This allows for a more thorough correctness and brings to fore the extensive type system of the *Scala* language.

For multi-threading and locks, we extended the logic with permissions, which can be used in an approach similar to the work of Amighi et al. [4]. However, this will require many additional axioms and bookkeeping to integrate it with our approach. An alternative would be the approach taken by Buisse, Birkedal, and Støvring [15], which elegantly integrates locks in the many-worlds approach we have taken here, but loses out by the lack of a sound separating implication operator.

While we have said hardly anything with regards to the soundness of our approach, much will be once again similar to the work of Birkedal et al. [10]. The main differences will lie in the different definition of the semantic Hoare triple and the integration of exceptions and multithreading.

6 Specification of *Scala* Actors: A Case Study

6.1 Introduction

In this section we evaluate the practical application of our work in previous sections, by using our extended separation logic to specify a representative piece of software, namely the *Scala* actors library. Since it is a part of the standard library, it is an example of idiomatic *Scala*. It also inherently deals with concurrency. We will start in Section 6.2 by describing this formalism and then give a detailed example of the library in use in Section 6.3. After this introduction, we shall examine the internal architecture of the library on a high level in Section 6.4 and then provide the detailed specified implementation in Section 6.5. Finally we shall evaluate the practicality of our extensions in use, along with further conclusions in Section 6.6.

6.2 On The Actor Model

As our case study deals with a library based on the actor model formalism, we shall first have a look at the model itself, with its formal definition in Section 6.2.1 and the practical use in Section 6.2.2.

6.2.1 Formally

The *Actor Model* is a general model of computation originating in AI-research [7]. Akin to the object-oriented model where everything is an object, the primitive is the *actor*. Actors have the following properties:

- They possess an identity, which acts as an address [7], as apposed to the anonymous processes in process algebra [40].
- On receiving a message, they can concurrently:
    1. Send a message to a finite number of actors.
    2. Create a finite number of new actors.
    3. Specify the behavior to handle the next incoming message.

As an inherently concurrent formalism, it is especially suited to reason about concurrent and distributed systems [1].

6.2.2 In Concurrent Programming

Another approach is to use the constructs formalized in the Actor Model as a basis for message-based concurrency in a programming language. The language would support Actors as a language construct and have them conform to the properties as specified by the Actor Model. More concretely, these languages generally allow you to instantiate Actors in a similar way to objects in object-oriented programming and have operators to asynchronously send messages between them. These languages suffer less of the problems commonly associated with shared-state concurrency involving locks, as the immutable message passing inherent to the model promotes share-nothing approach [44, pp. 724–725]. Common examples of languages which use the Actor Model in this way are *Erlang* and *Scala*.

- In *Erlang*, every object in the language, down to for instance an integer, is an actor. Each actor is backed by a user-mode thread, which are scheduled on kernel threads as required. Immutability of messages is guaranteed. This has resulted in *Erlang* being used for large scale commercial applications involving high levels of concurrency [55].
- The *Scala* language on the other hand, uses a library to facilitate actor model concurrency [44, pp. 724–725]. As it is an object-oriented language by nature, not everything is an actor, but only objects, specifically designated

as such, are. There is not a certain way to say which objects should be actors; it all depends on the way the application is modeled. Commonly however, actors either guard a resource or perform a (part of a) computation. As the overhead is generally low, it is not a problem to have many actors and it is best to split up computations in multiple actors to maximize parallelism. In *Scala* actors are described as: "a thread-like entity that has a mailbox to receive messages" [44, pp. 724–725]. They are implemented by subclassing the **`Actor`**-trait and implementing the `act`-method, something we shall look at in detail in Section 6.3.

6.2.3 Remaining Issues

While the share-nothing approach helps to prevent common problems that arise due to concurrent programming, there are some issues: Deadlocks are still possible, but become a matter of protocol design [17]. Starvation is an implementation-specific issue, depending on the fairness of the scheduler used. Non-determinism is actually likelier to occur due to the default of asynchronous message passing, but can be lessened by using petri-nets [49] or rendezvous communication. Actors tend to be composable, however, the messaging protocol often needs to be adjusted. Following general best practices [44, pp. 733-740] for such implementations lessens the issues named here and makes it a reliable alternative, or at least extension, to the shared-data approach [44, pp. 724–725].

6.3 Using *Scala* Actors

In this quick illustration of the *Actor model* as used in *Scala*, we shall take a common, well-defined, problem in concurrent computing, and see how it fits the *Actor Model* and implement it in *Scala*. Let us look at the *Bounded Buffer Problem*:

The *bounded buffer*, or *producer/consumer* problem is a commonly used illustration of concurrent processes, in which we have a process manufacturing items and a process consuming them – both as fast as they can – with the restriction that there's only a finite storage space for produced items. This finite storage space, or the bound of the so-called buffer, requires processes to wait on each other, when either the consumer is faster and the storage space is empty, or the producer is faster and the storage space is full.

In the example we will be describing here, the buffer or storage space will be an actor itself, allowing it to coordinate the producer and consumer via messages.

6.3.1 The Message Protocol

Because our actors will be communicating via messages, we will first have to define a protocol and the messages themselves. The protocol that we will use in our bounded buffer example, is shown in Section 6.3.1:

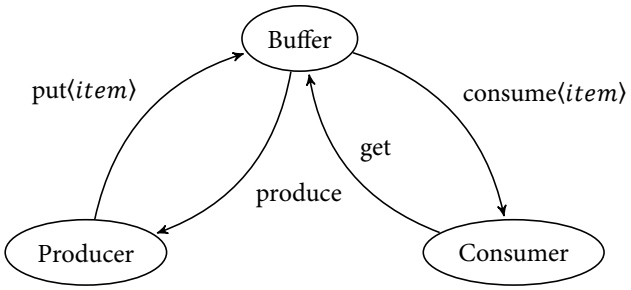


Figure 169: Visual Representation of the Bounded Buffer Protocol

The producer can **put** items in the buffer, the consumer **get**s them out, but only when the buffer allows them to by sending either **produce** or **consume**.

The messages are defined *case objects*:

```
class get (consumer : Actor)
class put (producer : Actor, item: Int)
class consume (item: Int)
object produce
```

Listing 42: Messages

Listing 42 shows that **get** and **put** are messages parametrized with an actor, essentially functioning as a return-address for the buffer to return messages to either the producer or the consumer.

6.3.2 Threaded Variant

Scala supports different variants of *Actor Model* concurrency. The first variant we'll be looking at is the one requiring a *JVM*-thread per actor (more on this when we will discuss the implementation in Section 6.5.5). We shall provide an example of its use:

```
class Buffer(contents : List[Int], capacity : Int) extends Actor
{
  var _capacity = capacity
  var _contents = contents

  def act()
  {
    while(true) receive
    {
      case put(producer, item) if _capacity > 0 =>
        Console.println("Buffering " + item)
        producer ! produce
        _capacity = _capacity - 1
        _contents = _contents ++ List(item)
      case get(consumer) if _contents.size != 0 =>
        Console.println("Unbuffering " + _contents.head)
        consumer ! consume(_contents.head)
        _capacity = _capacity + 1
        _contents = _contents.tail
    }
  }
}
```

Listing 43: Asynchronous Buffer

The lynchpin of the example is the class **`Buffer`** in Listing 43. Its internal state consists of an integer specifying the size of the buffer and a list representing the actual buffered items. It waits on producers and consumers either wanting to **get** or **put** an item. If a producer wants to **put** an item, we block until the internal state satisfies the guard `capacity > 0`. When the guard is satisfied, we asynchronously send **produce**, using the `!`-operator, to signal the producer to continue producing. Furthermore, we adjust the internal state as needed by decreasing the capacity and adding the newly produced item to the buffer. The case for the consumer is symmetrical.

Next up, let us look at the producer and consumer classes for our asynchronous example in Listing 44 and Listing 45 respectively:

```
class Producer(buffer : Actor) extends Actor
{
  var lastProduced = 0
  def act()
  {
    while(true)
    {
      lastProduced = lastProduced + 1
      Console.println("Producing" + lastProduced)
      buffer ! put(self, lastProduced)
      receive
      {
        case produce =>
      }
    }
  }
}
```

Listing 44: Asynchronous Producer

```
class Consumer(buffer : Actor) extends Actor
{
  def act()
  {
    while(true)
    {
      buffer ! get(self)
      receive
      {
        case consume(item) =>
          Console.println("Consuming " + item)
      }
    }
  }
}
```

Listing 45: Asynchronous Consumer

After the buffer, these two are nearly trivial. The producer continuously wants to produce and does so when it is allowed and the case is symmetric for the consumer, as `receive(f)` blocks when there are no matching messages. The producer has some internal state in the form of `lastProduced`, which serves only as an incrementing number to make for slightly clearer output.

Now the only thing remaining is to tie it all together into a functioning program:

```
object main extends App
{
  val buffer = new Buffer(List.empty, 10)
  val producer = new Producer(buffer)
  val consumer = new Consumer(buffer)
  buffer.start()
  producer.start()
  consumer.start()
}
```

Listing 46: Asynchronous Main

A seemingly trivial, but quite necessary part is constructing and starting the actors shown in Listing 46. The buffer is constructed first, to allow it's address to be passed on to the producer and consumer.

With the first complete example out of the way, it's time to look at some different methods of communication, namely synchronous messaging and the use of futures. First off, the synchronous variant:

Synchronous, or Rendezvous communication as it is sometimes called, works on the principle of blocking a send until you have received a reply. In *Scala* it is available with the `!?`-operator and `reply(message)`. Listing 47 shows the modifications to the buffer for synchronous communication.

```
case put(producer, item) if _capacity > 0 =>
  Console.println("Buffering " + item)
  reply(produce)
  _capacity = _capacity - 1
  _contents = _contents ++ List(item)
```

Listing 47: Synchronous Buffer

As seen, we can now simply `reply(message)` to the last message received. The rest of the buffer remains exactly the same.

On the consumer side, we have a slight variation in sending the message (the producer is practically symmetrical, thus will be omitted here):

```
while(true)
{
  buffer !? get(self) match
  {
    case consume(item) => Console.println("Consuming " + item)
  }
}
```

Listing 48: Synchronous Consumer

As we can see in Listing 48, we now use the `!?`-operator for sending, which blocks until an answer is given, so we no longer need a separate receive. `!?` returns the answer in message form, as we would normally encounter in it `receive`, so we use pattern matching to unpack the message and retrieve the item.

Our last example in this section deals with the use of futures, similar to synchronous communication in that it allows replies, but without immediately blocking the sending side to wait for a reply. Futures require the use of the `!!`-operator, `reply(message)` and `future()`.

```
while(true)
{
  lastProduced = lastProduced + 1
  Console.println("Producing" + lastProduced)
  val future = buffer !! put(self, lastProduced)
  future()
}
```

Listing 49: Synchronous Producer

The buffer is the same as in the previous example, so this time we start with the producer in Listing 49. The `!!`-operator returns immediately with an function called a **future**. The execution of function is a blocking operation returning the reply to the original message. This allows for intermediate actions, before deciding to wait on the reply.

6.3.3 Event-Driven Variant

Besides the thread-per-actor model, *Scala* also allows for actors to be event-based and to be scheduled on a pool of threads by an internal scheduler. As this requires some special attention due to specifics within the implementation, a different syntax is used. We shall once again look at the bounded buffer to see what this entails. The event-driven model allows for synchronous communication and events in the same fashion as the threaded model, so in this case we shall only demonstrate two variants of asynchronous bounded buffers. The first one is a direct equivalent of our original example:

```
def act()
{
  loop
  {
    react
    {
      case put(producer, item) if _capacity > 0 =>
        [..]
      case get(consumer) if _contents.size != 0 =>
        [..]
    }
  }
}
```

Listing 50: Event-Driven Asynchronous Buffer

Listing 50 shows the changes in buffer compared to the threaded variant. We change `while(true)` into `loop(f)` and `receive(f)` into `react(f)`. We do this because the fundamental issue with **react** is that it never returns. If we would use an ordinary loop, it would simply hang after the first iteration. The reason for this is that react does not preserve the call stack (more on this when we discuss the implementation). This attribute of `react(f)` does however allow for an interesting approach, namely the use of recursion in actors, as you would in *Erlang*, without risking the stack. As the rest of this example is the same as the original, with the two mentioned swaps, let us look at a recursive version instead:

```
def buffer(contents : List[Int], capacity : Int)
{
  react
  {
    case put(producer, item) if capacity > 0 =>
      Console.println("Buffering " + item)
      producer ! produce
      buffer(contents ++ List(item), capacity - 1)
    case get(consumer) if contents.size != 0 =>
      Console.println("Unbuffering " + contents.head)
      consumer ! consume(contents.head)
      buffer(contents.tail, capacity + 1)
  }
}
```

Listing 51: Event-Driven Recursive Asynchronous Buffer

As we can see in Listing 51, the buffer is now no longer an object, but simply a recursive function definition. There is, however, nothing making this function an actor quite yet, so this is something we'll have to keep in mind for later. The functionality remains exactly the same as before, however the variables we originally had for our internal state are removed in favour of passing values on to the recursive call to `buffer(contents, capacity)`.

```
def producer(buf : Actor, lastProduced : Int)
{
  Console.println("Producing" + lastProduced + 1)
  buf ! put(self, lastProduced + 1)
  react
  {
    case produce => producer(buf, lastProduced + 1)
  }
}
```

Listing 52: Event-Driven Recursive Asynchronous Producer

In the same fashion Listing 52 shows a recursive producer. Once again, we replace the state variable we had before with a value we pass on. A consumer in the same vein can be easily imagined.

What rests is to take the recursive functions and create actors out of them. The key to doing that lies in the function `react(f)` which takes a function and returns a started actor. We tie it together in listing Listing 53.

```
val buf = actor { buffer(List.empty, 10) }
actor { producer(buf, 0) }
actor { consumer(buf) }
```

Listing 53: Event-Driven Recursive "main"

6.3.4 Advanced Features

The actors library contains many more advanced features such as exception handling using actors, actor linking and automatic failover on actor termination [44, Chapter 32]. However, as this is not meant to be a comprehensive summary on the use of actors in *Scala*, we shall not go into any depth regarding those.

6.4 Architecture of *Scala* Actors

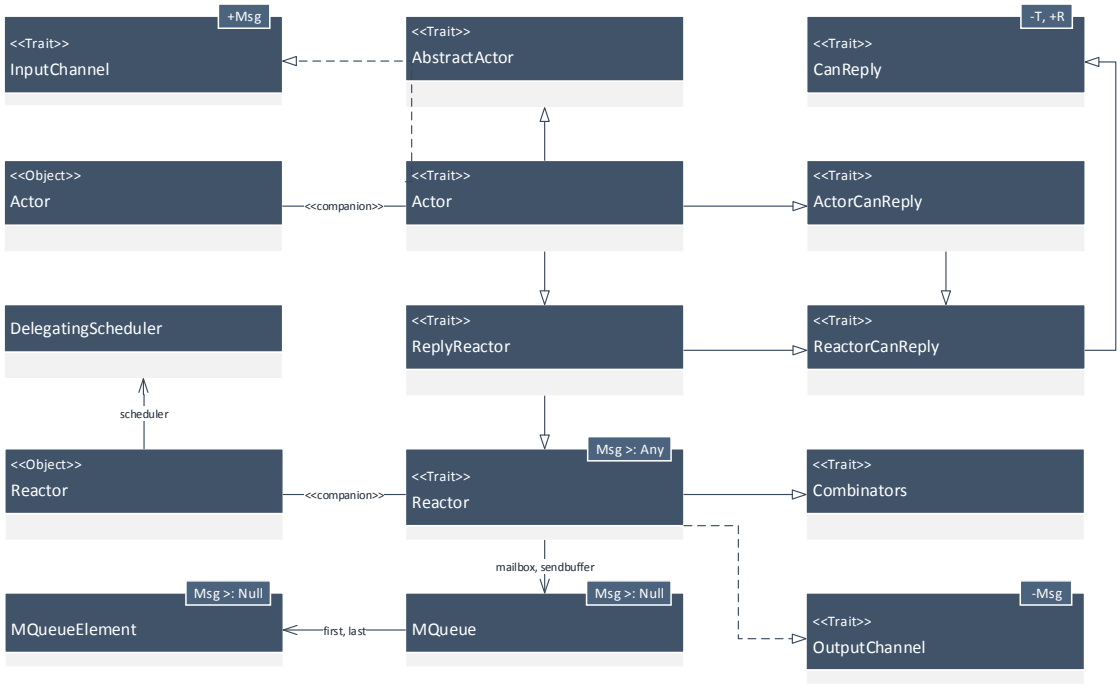


Figure 170: High Level Diagram of scala.actors

The `scala.actors` package is quite large and contains many classes and traits, of which the main ones are shown in Fig. 170. Central to all of these is the **`Reactor`** trait, which defines the basic actor model functionality: The mailbox in which messages are received.

Essentially, scheduling issues and filtering messages aside, the entire implementation can be summed up in two queues: one which temporarily holds newly received messages and the mailbox. All further elements of the implementation either have to do with scheduling, with determining which message from the mailbox to perform an action on, or with putting items into the queues. To put this in terms of the traits shown:

- **`Combinators`** defines helper methods for the composition of suspendable closures, used with react.
- **`InputChannel`** and **`OutputChannel`** are interfaces for receiving and sending messages respectively.
- **`AbstractActor`** is essentially an interface for methods specific to **`Actor`**, but with the addition of defining a concrete type for **`Future`**.
- **`CanReply`** is once again almost an interface, this time for sending messages which may have replies, and again additionally defines a concrete type for **`Future`**.
- **`ReactorCanReply`** and **`ActorCanReply`** are realizations of **`CanReply`**.
- **`Reactor`** defines the core functionality of actors, its companion object guards the scheduler.
- **`Actor`** unifies the traits and adds some specific functionality; this trait and its companion object are what are actually directly used when programming actors.

6.5 Implementation and Specification of *Scala* Actors

6.5.1 Regarding the Specification Language Used

For our specifications in text-format, we shall use a subset of JML, extended with separation logic constructs, with the following basic syntax:

R ::= b | **PointsTo**(**field**; **frac**; e)
| **Perm**(**field**, **frac**) | (**forall**$*$ T v; b; r)
| r_1 ** r_2 | r_1 −$*$ r_2 | b_1 ==> r_2 | r_1 f| => r_2
| $e.P(e_1, \ldots, e_n)$
R ::= Any pure expression
B ::= Any pure Boolean expression
| (**forall** T v; b_1; b_2)
| (**exists** T v; b_1; b_2)
Where T is an arbitrary type, v is a variable, P is a resource predicate and f is a function.

Figure 171: ASCII Specification Language

The ASCII-syntax given in Fig. 171 maps to the separation logic defined in Section 5, with the closure arrow being the representation of the first class Hoare triple, when used in pre- and postconditions of methods and functions. The syntax is a slight restriction of the separation logic in Section 5, but it sacrifices some expressivity for clarity and suffices for the task at hand. The specifications will be placed in comments, using indicators such as `ensures` for preconditions, **`requires`** for postconditions, `given` for ghost arguments, `yields` for ghost returns and `invariant` for class-invariants. To refer to the return value of a method we shall use `\result` and to refer to the return value of a function f in r_1 f| => r_2, we shall use **`\return`**. To refer to the pre-state of a formula we shall use `\old(p)`. We also assume the existance of a predicate $list$ to abstractly deal with data structures, with concatenation ++, length #, reverse and indexing [] operations predefined.

6.5.2 MQueue

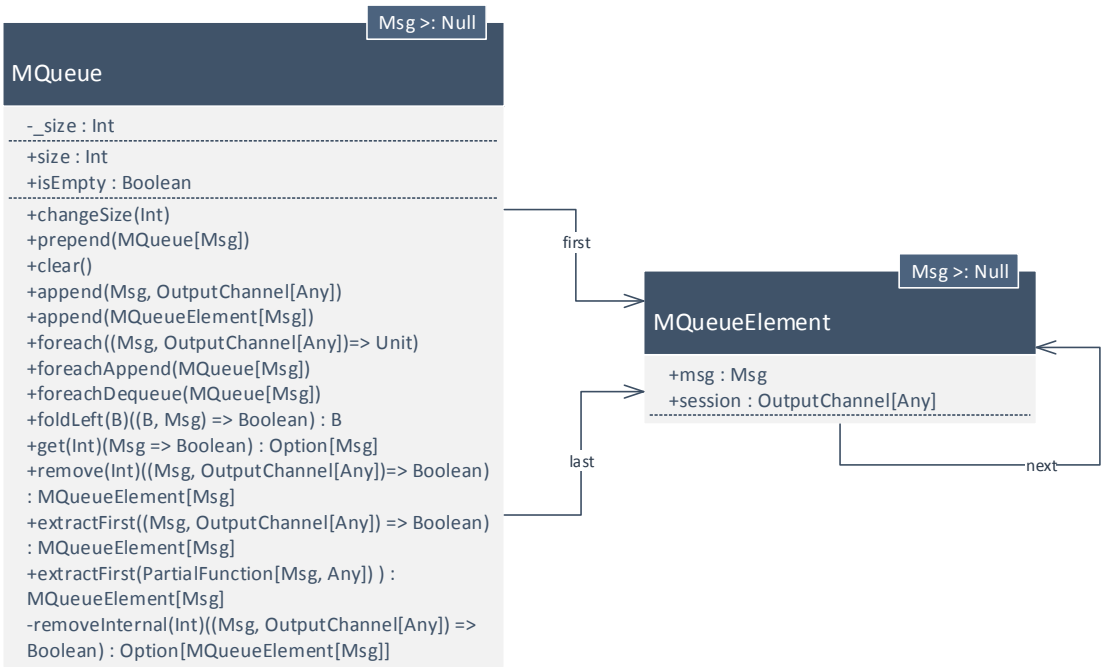


Figure 172: Class Diagram of MQueue

The first class to be specified is **`MQueue`**: It is mostly self-contained, used only in the actors framework, either under guard of a lock or in non-contentious situations, makes use of closures and is of fundamental importance to the framework, making it an ideal first choice.

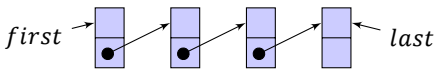


Figure 173: Visual Representation of the Internal List

`MQueue` models a queue with added actor-specific functionality. Internally it consists of a singly linked list of **`MQueueElement`** instances, visualized in Fig. 173. Specification of **`MQueueElement`** is trivial and therefore omitted.

```
private[actors] class MQueue[Msg >: Null](protected val label: String) {[..]}

protected /*@ spec_public */ var first: MQueueElement[Msg] = null
protected var last: MQueueElement[Msg] = null
private var _size = 0
```

Listing 54: Constructor & Fields of MQueue

Listing 54 shows the constructor and internal fields of the queue. The queue is constructed with a name and a generic

parameter specifying the type of message this queue contains. The fields consist of two pointers, one to the first element and one to the last element of the linked list, and an integer representing the length of the linked list and thus of the queue. The pointer `first` is annotated *spec_public* to allow its use in publicly visible specifications.

For specification of the queue we need a means to publicly state facts about the contents of the queue, without leaking implementation details. Furthermore it is likely that specifications of classes using an instance the queue will require a means to reason about the contents of the queue as well. Because this is fundamentally a case of abstraction, we once again turn to abstract predicates: We use the *list* predicate as defined in Section 6.5.1 and formulate some practical predicates, shown in Listing 55.

```
/*@
    public pred state<perm p> = PointsTo(_size, p, _) * PointsTo(first, p, _) * state<p, first.next>
    public pred state<perm p, MQueueElement[Msg] mqe> = PointsTo(mqe, p, _) * (mqe != null ==>
                  state<p, mqe.next>)

    public pred queue<out alpha, MQueueElement fst> = list alpha fst

    public pred length<out int l, MQueueElement fst> = queue<alpha, fst> ** #alpha == l

    public pred isLength<int l, MQueueElement fst> = length<l', fst> ** l == l'

    public pred empty<MQueueElement fst> = length<0, fst> ** PointsTo(first, \epsilon, null) **
                                           PointsTo(last, \epsilon, null) ** PointsTo(_size, 1, 0)

    public pred contains<MQueueElement mqe, MQueueElement fst> = mqe == fst || contains<a, alpha>

    public pred isFirst<MQueueElement mqe, MQueueElement fst> = peekFirst<m, fst> ** mqe == m

    public pred isLast<MQueueElement mqe, MQueueElement fst> = peekLast<m, fst> ** mqe == m

    public pred peekFirst<out MQueueElement mqe, MQueueElement fst> = queue<mqe::alpha, fst>
                ** mqe == fst

    public pred peekLast<out MQueueElement mqe, MQueueElement fst> = queue<alpha, fst> **
                  (\exists beta, (reverse alpha mqe::beta))
*/
```

Listing 55: Predicates Defined for Specification of MQueue

While the **list** predicate itself is sufficient for most of our purposes, we define variants using *out*-parameters to cut down on the visual clutter of constant existential quantifications in the specification. Of the predicates shown in Listing 55 the meaning is as follows:

- $state$ asserts a given permission over all elements of the list, which allows us to concisely specify access.
- $queue$ provides the list starting at the given element; The complete internal list α is then given by $queue\langle\alpha, first\rangle$.
- $length$ provides the length of the list starting at the given element, with the length l of the entire list being given by $length\langle l, first\rangle$.
- $isLength$ is the assertion that there exists a list starting at the given element with length l.
- $empty$ asserts the empty list
- $contains$ asserts there exists a list starting at the given element which contains the element mqe.
- $peekFirst$ and $peekLast$ respectively provide the first and last element of a list starting at the given element.
- $isFirst$ and $isLast$ assert the first or last element of a list being equal to the given element.

Using these predicates it is now possible to state a number of invariants – as shown in Listing 56 – and to abstractly specify all the public functions of the queue, starting with one of the most straightforward, the `size` method, in Listing 57.

```
/*@
    public invariant list first
    public invariant isFirst<first, first>
    public invariant empty<first> <==> isEmpty() <==> isLength<0, first>
    public invariant length<l, first> ** l == size

    private invariant isLast<last, first>
*/
```

Listing 56: Invariants of MQueue

The invariants state that the `first` pointer always points to a list and it is the first element of the list it points to. Furthermore, equivalence between `isEmpty`, the $empty\langle first\rangle$ predicate and $length\langle first, 0\rangle$ predicate is guaranteed, just als the equivalence of the result of the $length$ predicate and the `size` method. Finally there is an invariant that the `last` pointer is in fact the last element of the list.

```
/*@
    public normal_behaviour:
        pure
        requires state<\epsilon>
        ensures state<\epsilon> * length<\result, first>
*/
def size = _size
```

Listing 57: The size Method of MQueue

The `size` method is a trivial accessor method, but it directly demonstrates the use of the predicates: We can specify the fact that it requires a read access on the queue of some ϵ using $state$ and we can link the integer returned directly to the length of our abstract representation using $length$. In the same manner we can specify the other trivial method, `isEmpty`, using the $empty$ predicate; The full listing is omitted for brevity.

```
/*@
    protected normal_behaviour:
      requires state<1> ** length<l, first> ** (l + diff >= 0)
      ensures state<1> ** \old(length<l, first>) ** isLength<l+diff, first>
*/
protected def changeSize(diff: Int) {
    _size += diff
}
```

Listing 58: The changeSize Method of MQueue

Slightly more interesting is the `changeSize` method – shown in Listing 58 – as it is the first method to change the internal state of the queue. As this method modifies the internal state, we specify full write permissions on the state using $state\langle 1\rangle$. The functionality of the method is related to the abstract internal list using the $length$ and $isLength$ predicates.

With the helper and accessor methods out of the way, it is time to have a look at the functional aspects of the queue. We shall skip the straightforward `clear` method, which is easily specified using *state* and *empty* and start with `foreach` and `foldleft`.

Due to the specific internal use of **`MQueue`**, it does not implement any of the expected basic collection traits such as **`Traversable`** or **`Iterable`**, but instead it provides its own `foreach` and `foldLeft`. The functionality of these methods is straightforward and as expected. We shall specify the methods using the specification methods for closures as proposed in Section 5 and the by now somewhat familiar predicates.

```
/*@
    public normal_behaviour:
      given cs
      requires state<1> ** length<j, first> ** \old(queue<alpha, first>) ** cs
      requires (\forall int i; 0 <= i && i < j,
                 (cs f(alpha[i].msg, alpha[i].session)<cs> |=> cs))
      ensures state<1> ** length<j, first> ** \old(queue<alpha, first>) ** cs
      ensures (\forall int i; 0 <= i && i < j,
                (cs f(alpha[i].msg, alpha[i].session)<cs> |=> cs))
      yields cs
*/
def foreach(f: (Msg, OutputChannel[Any]) => Unit) {
    var curr = first
    /*@
        invariant queue<alpha, curr> ** queue<beta, first>
        invariant list<alpha, curr> -* list<beta, first>
        invariant cs
        invariant (\forall int i; 0 <= i && i < (#beta-#alpha),
                    (cs f(curr.msg, curr.session)<cs> |=> cs) ** cs)
        decreases #alpha
    */
    while (curr != null) {
        f(curr.msg, curr.session)
        curr = curr.next
    }
}
```

Listing 59: The foreach Method of MQueue

```
/*@ pred state<Int x> = Pointsto(count, 1, x) */

/*@
    given state<V>
    requires state<V>
    ensures (x%2) ==> state<V+1> ** !(x%2) ==> state<V>
    yields state
*/

def inc = (x : Int, session : OutputChannel[Any]) =>
    {count = if(x%2) count+1 else count}
```

Listing 60: A Sample Specified Function for use with foreach

The `foreach` method – shown in Listing 59 – is specified with write permissions, as `f` may have side-effects on the queue. Since the `foreach` method takes a general function as its argument, without many constraints, the closure specification takes a similar general approach. It states that the precondition, given by an instance of *cs*, of the function `f`, should hold for all elements in the queue and after executing `f` on such an element, its postcondition will once again be an instance of *cs*. This is made possible by the fact that unmentioned parameters of predicates are implicitly quantified. This would, for instance, make Listing 60 a valid function for this method, given **`Msg >: Int`**, while having a parameterized state predicate.

```
/*@
    public normal_behaviour:
      given cs<V, W>
      requires state<1> ** length<l, first> ** \old(queue<alpha>) ** cs<z, first>
      requires (\forall int i; 0 <= i && i < l, (\exists B acc,
                (cs<acc, alpha[i]> f (acc, alpha[i].msg) <cs<acc, alpha[i]> > |=>
                  cs<\return, alpha[i].next>)))
      ensures state<1> ** length<l, first> ** queue<alpha>
      ensures peekLast<lst, first> ** cs<\result, null>
      ensures (\forall int i; 0 <= i && i < l, (\exists B acc,
                (cs<acc, alpha[i]> f (acc, alpha[i].msg) <cs<acc, alpha[i]> > |=>
                  cs<\return, alpha[i].next>))
      yields cs<\result, null>
*/
def foldLeft[B](z: B)(f: (B, Msg) => B): B = {
    var acc = z
    var curr = first
    /*@
        invariant queue<alpha, curr> ** queue<beta, first>
        invariant list<curr> -* list<first>
        invariant cs
        (\forall int i; 0 <= i && i < (#beta-#alpha),
          (cs<acc, beta[i]> f (acc, beta[i].msg) <cs<acc, alpha[i]> > |=>
            cs<\return, alpha[i].next>))
        decreases #alpha
    */
    while (curr != null) {
        acc = f(acc, curr.msg)
        curr = curr.next
    }
    acc
}
```

Listing 61: The foldLeft Method of MQueue

```
/*@ pred state<Int x, MQueueElement[Int] e> = queue.state<1> */

/*@
    given state<V, W>
    requires state<V, W>
    ensures (m%2) ==> state<V+1, W.next> ** !(x%2) ==> state<V, W.next>
    yields state
*/

def inc = (z : Int, m : Int) : Int => {if(m%2) x+1 else x}
```

Listing 62: A Sample Specified Function for use with foldLeft

The `foldLeft` method – shown in Listing 61 – is similar to `foreach`, since it applies a function to all the elements in the queue, but with an additional accumulated value of type `B`. The specification is therefore similar as well, but it has the important distinction of dealing with a parameterized variant of the predicate. This parameterization allows us to specify meaningful connections between the function state and the initial and final accumulated values: Initially the predicate with the first element and the initial accumulator value `z`, should satisfy the precondition of the function. Then, for every element there should exist an instance of *cs* with an accumulated value and the current element which satisfies the precondition of the function and after execution there should exist an instance of *cs* with the result of `f` and the next message which satisfies the postcondition. We provide a sample specified function which would qualify for `f` in Listing 62.

Next we shall cover a group of closely related methods, namely `prepend`, `foreachAppend` and `foreachDequeue`.

```
/*@
    public normal_behaviour:
      requires !other.isEmpty() ==> other.state<1> ** state<1>
      ensures !other.isEmpty() ==> other.state<1> ** state<1> **
                                   \old(other.queue<gamma, other.first>) **
                                   queue<alpha, first> **
                                   \old(queue<beta, first>) **
                                   alpha == gamma ++ beta
    also protected normal_behaviour:
      ensures PointsTo(other.last.next, 1, first) ** PointsTo(first, 1, other.first)
      ensures Perm(other.first, \epsilon)
*/
def prepend(other: MQueue[Msg]) {
    if (!other.isEmpty) {
        other.last.next = first
        first = other.first
    }
}
```

Listing 63: The prepend Method of MQueue

The method `prepend` – seen in listing Listing 63 – modifies the current queue by prepending the target queue. Our predicates allow us to specify this by saying that the new queue will simply be the concatenation of the old queue with the target one, given write permission on both queues. We also provide a list invariant, which implies the postcondition, specifies permission transfer and shows termination in terms of the length of the remaining unprepended list.

```
/*@
    public normal_behaviour:
      requires state<1> ** target.state<1>
      ensures state<1> ** target.state<1> ** queue<alpha, first> ** target.queue<beta>
      ensures \old(target.queue<gamma, target.first>) ** (beta == (gamma ++ alpha))
*/
def foreachAppend(target: MQueue[Msg]) {
    var curr = first
    /*@
        invariant \old(target.length<l, target.first>) ** target.queue<gamma, target.first>
        invariant queue<alpha, curr> ** queue<beta, first>
        invariant list<alpha, curr> -* list<beta, first>
        invariant (\forall int j; 0 <= j && j < (#beta-#alpha), gamma[l+j] == beta[j])
        decreases #alpha
    */
    while (curr != null) {
        target.append(curr)
        curr = curr.next
    }
}
```

Listing 64: The foreachAppend Method of MQueue

Listing 64 shows the method `foreachAppend`, which is similar in functionality to `prepend`, with the difference that it instead appends the current queue to the target one. The specification is largely the same, but with the current and target queues swapped and a reverse order of concatenation.

The method `foreachDequeue` is almost identical to `foreachAppend` with the addition that it clears the source queue after the append, so we omit its listing for brevity.

Finally we will cover the single-element operations of the queue, namely `append` in Listing 65, `get` in Listing 67, `remove` in Listing 66 and `extractFirst`.

```
/*@
    public normal_behaviour:
      requires state<1>
      ensures peekLast<m, first> **  m.msg eq msg ** m.session eq session
      ensures (empty ==> (peekFirst<m'> ** m eq m'))
      ensures old(queue<beta, first> ** queue<alpha, first> ** alpha == beta ++ [m]
      ensures \old<length<i, first> > ** isLength<i+1, first>
    also protected normal_behaviour:
      ensures peekLast<m, first> **  m.msg eq msg ** m.session eq session
      ensures pointsto(last, 1, m) ** empty ==> pointsto(first, 1, m) **
      ensures !empty ==> pointsto(last.next, 1, m))
*/
def append(msg: Msg, session: OutputChannel[Any]) {
    changeSize(1) // size always increases by 1
    val el = new MQueueElement(msg, session)

    if (isEmpty) first = el
    else last.next = el
    last = el
}
```

Listing 65: The append Method of MQueue

The append-method appends a single element. The specification requires the usual access to the queue and it ensures the new element will be the new last element, with the rest of the queue remaining the same, and that it will be the first element, if it is appended to an empty queue.

The get-method returns the n^{th} element of the elements that match the predicate p, while leaving the queue intact. In the specification we require read permission only, as the method is pure and we ensure that, when there is a result, there are at least n items matching p in the queue and, using the $nMsgMatch$ predicate, that the result is indeed the n^{th} one matching the predicate p. We also provide a list invariant, which implies the postcondition, specifies permission transfer and shows termination in terms of the length of the remaining untested list.

```
/*@
    public pred nMsgSessionMatch<Msg m, int n, (Msg, OutputChannel[Any])->Boolean p, a::alpha> =
                (p(a.msg, a.session) ** a.msg eq m ** n == 1) || (m == p(a.msg, a.channel) **
                nMsgMatch<m, n-1, p, alpha>) || (nMsgMatch<m, n, p, alpha)

    public normal_behaviour:
      given cs
      requires state<1> ** length<l, first> ** cs
      requires (\forall int i; 0 <= i && i < l,
                  (<cs> p (alpha[i].msg, alpha[i].session) <cs> |=> cs))
      ensures state<1> ** \old<length<l, first> > ** \old(queue<alpha, first>)
      ensures queue<beta, first> ** cs
      ensures (\forall int i; 0 <= i && i < l, p (alpha[i].msg, alpha[i].session) <cs> |=> cs)
      ensures \result != None ==> nMsgSessionMatch<\result.get._1, n, p, alpha> **
                                  ((<cs> p (alpha[i].msg, alpha[i].session) <cs> |=> cs) ** \return) **
                                  isLength<l-1, first> ** !nMsgSessionMatch<\result.get._1, n, p, beta>
      ensures \result == None ==> !nMsgSessionMatch<\result.get._1, n, p, alpha>
      yields cs
*/
def remove(n: Int)(p: (Msg, OutputChannel[Any]) => Boolean): Option[(Msg, OutputChannel[Any])] =
    removeInternal(n)(p) map (x => (x.msg, x.session))
```

Listing 66: The remove Method of MQueue

The `remove`-method is essentially like `get` in function, with the addition that it removes the matching element from the queue.

The `extractFirst`-method and its straightforward overload which takes a **`PartialFunction`** are simply `remove` with an n of 0 and will therefore be omitted here. Methods `extractFirst` and `remove` both defer to the `removeInternal` method, but as the specification is identical to remove and the implementation is straightforward, we omit a listing in this case as well.

```
/*@
    public pred nMsgMatch<Msg m, int n, Msg->Boolean p, a::alpha> =
      (p(a.msg) ** a.msg eq m ** n == 1) ||
      (m == p(a.msg) ** nMsgMatch<m, n-1, p, alpha>) ||
      (nMsgMatch<m, n, p, alpha)

    public normal_behaviour:
      pure
      given cs
      requires state<\epsilon> ** cs ** (\forall int i; 0 <= i && i < j,
                                          (<cs> p alpha[i].msg <cs> |=> cs))
      ensures state<\epsilon> ** length<j> ** queue<alpha> ** cs
      ensures (\forall int i; 0 <= i && i < j, (cs p alpha[i].msg <cs> |=> cs))
      ensures \result != None ==> (nMsgMatch<\result.get, n, p, alpha> **
          ((cs p \result.get <cs> !=> cs) ** \return))
      ensures \result == None ==> !nMsgMatch<\result.get, n, p, alpha>
      yields cs
*/
def get(n: Int)(p: Msg => Boolean): Option[Msg] = {
    var pos = 0

    def test(msg: Msg): Boolean =
        p(msg) && (pos == n || { pos += 1; false })

    var curr = first
    /*@
        invariant queue<alpha, curr> ** queue<beta, first>
        invariant list<alpha, curr> -* list<beta, first>
        invariant cs ** (\forall int i; 0 <= i && i < (#beta-#alpha),
                           (cs p alpha[i].msg <cs> |=> cs) ** !test(alpha[i].msg))
        invariant test(curr.msg) ==> nMsgMatch<curr.msg, n, p, beta> **
                  ((<cs> p \result.get <cs> !=> cs) ** \return)
        decreases #alpha
    */
    while (curr != null)
    {
        if (test(curr.msg)) return Some(curr.msg) // early return
        else curr = curr.next
    }

    None
}
```

Listing 67: The get Method of MQueue

6.5.3 Reactor

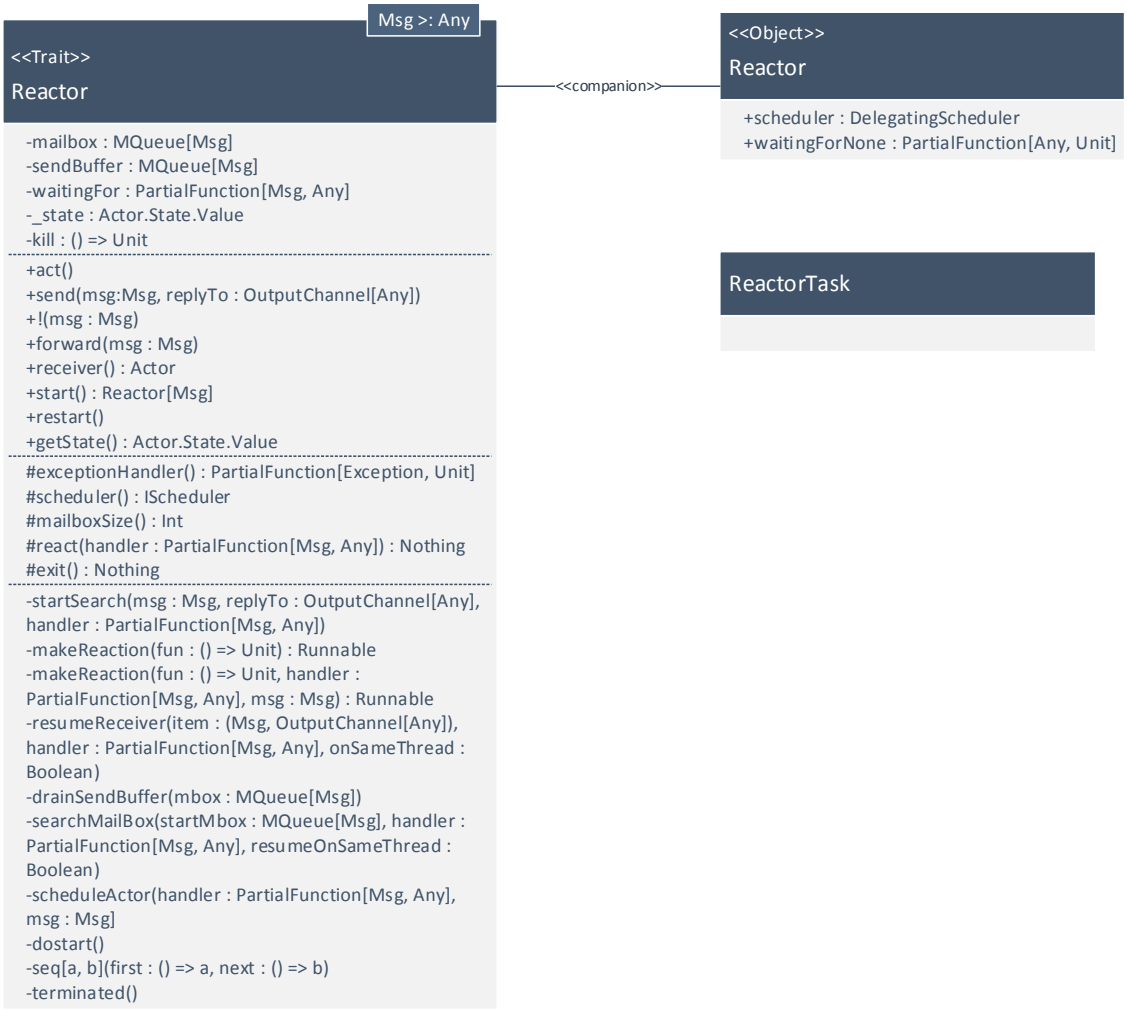


Figure 174: Reactor Class Diagram

The **Reactor**-trait forms the core of the `actors` package, providing much of the basic functionality. It makes use of the **MQueue** class which we specified in Section 6.5.2 and the **ReactorTask** class which we shall cover in Listing 80. **Reactor** is used in multithreaded scenarios, using locks to protect critical resources.

```
trait Reactor[Msg >: Null] extends OutputChannel[Msg] with Combinators {..}
```

Listing 68: The **Reactor** Trait Definition

As shown in Listing 68, **Reactor** extends the **OutputChannel** interface and the **Combinators** trait, which provides some basic constructs to deal with continuations. The generic parameter **Msg** is restricted to reference types. There is no constructor.

```
private[actors] val mailbox = new MQueue[Msg]("Reactor")

private[actors] val sendBuffer = new MQueue[Msg]("SendBuffer")

private[actors] var waitingFor: PartialFunction[Msg, Any] =
    Reactor.waitingForNone
```

Listing 69: The Fields of **Reactor**

Listing 69 shows three of the five fields in **Reactor**:

1. `mailbox`: Where received messages are stored until they are processed. the mailbox is only ever accessed by the main thread the actor is executing on, so it is not guarded by a lock.
2. `sendBuffer`: Where other threads put their messages via `send`. `sendBuffer` is guarded using the **Reactor** itself as a lock.
3. `waitingFor`: The stored continuation of a currently suspended actor, i.e. the **PartialFunction** given to either `receive` or `react`. When an actor is executing, i.e. not suspended, it is equal to **WaitingForNone**. This field is accessed by multiple threads and is guarded by the **Reactor**-lock.

```
/*@
    initially _state == Actor.State.New
    invariant _state == Actor.State.New ||
              _state == Actor.State.Runnable ||
              _state == Actor.State.Terminated

    constraint \old(_state) ==
        Actor.State.New =>
            _state == Actor.State.New ||
            _state == Actor.State.Runnable ||
            _state == Actor.State.Terminated
        && Actor.State.Runnable =>
            _state == Actor.State.Runnable ||
            _state == Actor.State.Terminated
        && Actor.State.Terminated =>
            _state == Actor.State.Runnable ||
            _state == Actor.State.Terminated
*/
private[actors] var _state: Actor.State.Value = Actor.State.New
```

Listing 70: The **_state** Field in **Reactor**

```
def getState: Actor.State.Value = synchronized {
    if (waitingFor ne Reactor.waitingForNone)
        Actor.State.Suspended
    else
        _state
}
```

Listing 71: How Other States Rise from the Main Three

Listing 70 shows **_state** – the fourth of the fields in **Reactor** – and its specification. This field, as its name implies, represents the current state of the actor. It is guarded by **Reactor** for concurrent access. Of the possible states of an actor shown in Listing 86, **_state** can only take the three values shown in the specification. The others are combinations of these three states and conditions that arise in the actor; an example of those is shown in Listing 71.

```
/*@
    inv = sendBuffer.state<1> ** Perm(_state, 1) ** Perm(waitingFor, 1)
*/
```

Listing 72: The Resource Invariant of **Reactor**

The resource invariant – shown in Listing 72 – mirrors the state guarded by the **Reactor**-lock, which consists of the fields `sendBuffer`, `waitingFor` and **_state**.

With the internal state out of the way, we shall now look at the methods of **Reactor**, beginning with the `start` in Listing 73.

```
/*@
    public normal_behaviour:
      requires fresh;
      ensures Lockset(S) ** (S contains this -* inv);
*/
def start(): Reactor[Msg] =
  /*@ assert Lockset(S) ** (S contains this -* inv) ** initialized */
  /*@ commit */
  synchronized {
    if (_state == Actor.State.New)
      dostart()
    this
  }
```

Listing 73: The `start` method in **Reactor**

```
/*@
    private normal_behaviour:
      requires
        Lockset(S) ** S contains this ** inv
        ** _state == Actor.State.New
      ensures
        Lockset(S) ** S contains this * inv
        ** _state == Actor.State.Runnable
*/
private[actors] def dostart() {
  _state = Actor.State.Runnable
  scheduler newActor this
  scheduler execute makeReaction(() => act(), null, null)
}
```

Listing 74: The `doStart` method in **Reactor**

The start method – shown in Listing 73 – starts the execution of the actor, assuming it is not already running. As there is no constructor and `start` is always the first method called on an actor, we commit the resource invariant at

this point. The now committed invariant is immediately put to use as the start method uses a `synchronized` block to access **`_state`**.

The `dostart`-method in Listing 74 is where the actual transition to the running state happens by changing **`_state`**, announcing the actor to the scheduler, and scheduling the act method, wrapped in a **`ReactorTask`** by `makeReaction`, for execution. It is only called from `start` and therefore always guarded by the lock, which allows us to specify the pre and post state of **`_state`**.

```
/*@
    public normal_behaviour:
      requires
        Lockset(S) ** (S contains this -* inv) ** initialized
        ** \rec_imm(msg)
      ensures Lockset(S) * (S contains this -* inv)
*/
def send(msg: Msg, /*@ nullable */ replyTo: OutputChannel[Any]) {
  val todo = synchronized {
    if (waitingFor ne Reactor.waitingForNone) {
      val savedWaitingFor = waitingFor
      waitingFor = Reactor.waitingForNone
      startSearch(msg, replyTo, savedWaitingFor)
    } else {
      sendBuffer.append(msg, replyTo)
      () => { /* do nothing */ }
    }
  }
  todo()
}
```

Listing 75: The `send`-method in **`Reactor`**

The `send`-method is one of the most important in **`Reactor`**, as sending and receiving messages are the defining aspects of actors. As the `send`-method is called concurrently by other actors, it uses a `synchronized` block to access `waitingFor` and `sendBuffer`. Therefore we specify the usual pre- and postconditions for a lock. Furthermore we demand the message to be provably immutable, to prevent in-flight modification.

Intuitively, all `send` would do is append a message to the mailbox, but the implementation shown in Listing 75 is more complicated: The first thing to explain is the use of `todo`: `todo` does not in fact execute the `synchronized` block, as this would have no useful effect, but it executes the function that results from the synchronized-block. In the case that `waitingFor` equals `waitingforNone` i.e. the actor is currently executing on some thread, the incoming message is appended to the `sendBuffer` and an empty function is returned, and executed outside of the block. In case `waitingFor` is not `waitingForNone` – meaning that the actor is suspended and the continuation is stored in `waitingFor` – the continuation is passed to `startSearch` and the function that is executed by calling `todo` is the function returned by `startSearch`. The practical result of this is that when an actor is currently executing, send adds to the buffer, which the executing actor checks for new messages, and when it is not executing, it starts the process of scheduling the actor for execution to handle the new message. The lock guarantees that even with multiple actors sending, whether the actor is or is not already executing, the `act` method is only ever executed on one thread.

```
def !(msg: Msg) {
  send(msg, null)
}

def forward(msg: Msg) {
  send(msg, null)
}
```

Listing 76: The `forward` and `!` Methods in **`Reactor`**

`Reactor` provides two shorthand methods using `send`, both sending the message without a provided return address; these are shown in Listing 76.

```
/*@
  private normal_behaviour:
    requires Lockset(S) ** S contains this ** inv ** \rec_imm(msg)
    requires handler != Reactor.WaitingForNone
    ensures Lockset(S) ** S contains this ** inv
    ensures cs \result () <cs> |=> cs
    yields cs
*/
private[actors] def startSearch(msg: Msg, replyTo: OutputChannel[Any],
                                handler: PartialFunction[Msg, Any]) =
  /*@
    pred cs = Perm(Reactor.scheduler, \epsilon)
  */
  /*@
    given cs; requires cs; ensures cs; yields cs
  */
  () => scheduler execute makeReaction(
    /*@
      pred csl = Reactor.Lockset(S) ** (S contains this -* Reactor.inv) **
                 Reactor.initialized
    */
    /*@
      given csl; requires csl; ensures csl; yields csl
    */
    () => {
        val startMbox = new MQueue[Msg]("Start")
        synchronized { startMbox.append(msg, replyTo) }
        searchMailbox(startMbox, handler, true)
      })
```

Listing 77: The `startSearch`-method in **`Reactor`**

Listing 77 shows the `startSearch`-method which is called from `send`. As mentioned before – while covering `send` – this method returns a function, which schedules an actor for execution, to handle the incoming message. Its specification shows that it is always executed under a lock, namely the one from the `send`-method from which it is called, the remaining immutability of the message and the fact there should be a valid continuation to schedule. Furthermore, it specifies and yields the required access predicate to call the returned function. The innermost function – which is the argument to `makeReaction` – is specified using a predicate which encapsulates the standard

pre- and postconditions for acquiring a lock, however, it is unclear what the use of the `synchronized` block is. This innermost function is what is finally scheduled for execution, after it has been wrapped into a **`ReactorTask`** object by `makeReaction`.

Listing 79 shows what gets executed on the thread, on which the **`ReactorTask`** is scheduled, via `startSearch`. As mentioned before, only one thread is scheduled to execute an actor at any time, so this method executes without contention on `mailbox`. The specification requires access to the queues involved and, assuming that `tmpMbox` does not initially contain a match, it requires locking. This lock guards the `sendBuffer`, where messages may have been added due to `send`, and `waitingFor`, to store the continuation when suspending. The method searches for a match in the incoming messages using `isDefinedAt` and resumes the actor if it finds one. If it does not, it stores the continuation and throws an exception to signal suspension.

```
/*@
  private normal_behaviour:
    signals SuspendActorControl
*/
private[actors] def resumeReceiver(item: (Msg, OutputChannel[Any]),
    handler: PartialFunction[Msg, Any], onSameThread: Boolean) {
  if (onSameThread)
    makeReaction(null, handler, item._1).run()
  else
    scheduleActor(handler, item._1)

  throw Actor.suspendException
}
```

Listing 78: The `resumeReceiver`-method in **`Reactor`**

The method resumeReceiver – shown in Listing 78 – is responsible for resuming the actor, either by running the continuation on the same thread, or by scheduling it on another. In case of being called from a **`ReactorTask`** created via `send`, it is always resumed on the same thread and vice versa when it is called via `react`. Eventually an exception is thrown to influence control flow in the **`ReactorTask`** from which this method is called.

```
/*@
  private normal_behaviour:
    requires
      startMBox.state<1> **
      mailbox.state<1> **
      !(\exists int i; i >= 0 && i < startMbox.size(),
          startMbox.queue<alpha, startMbox.first> **
          ((handler.isDefinedAt(alpha[i].msg)) => Lockset(S) **
          (S contains this -* inv) ** this.initialized)
    ensures
      startMBox.state<1> **
      mailbox.state<1> **
      !(\exists int i; i >= 0 && i < startMbox.size(),
          startMbox.queue<alpha, startMbox.first> **
          (handler.isDefinedAt(alpha[i].msg)) =>
            Lockset(S) ** (S contains this -* inv))
  also private exceptional_behaviour:
    ensures waitingFor == handler
    signals SuspendActorControl
*/
private[actors] def searchMailbox(startMbox: MQueue[Msg],
                                  handler: PartialFunction[Msg, Any],
                                  resumeOnSameThread: Boolean) {
  var tmpMbox = startMbox
  var done = false
  while (!done) {
    val qel = tmpMbox.extractFirst(handler)
    if (tmpMbox ne mailbox)
      tmpMbox.foreachAppend(mailbox)
    if (null eq qel) {
      synchronized {
        if (!sendBuffer.isEmpty) {
          tmpMbox = new MQueue[Msg]("Temp")
          drainSendBuffer(tmpMbox)
        } else {
          waitingFor = handler
          throw Actor.suspendException
        }
      }
    } else {
      resumeReceiver((qel.msg, qel.session), handler, resumeOnSameThread)
      done = true
    }
  }
}
```

Listing 79: The `searchMailbox`-method in **`Reactor`**

```
def run() {
  try {
    beginExecution()
    try {
      if (fun eq null)
        handler(msg)
      else
        fun()
    } catch {
      case _: KillActorControl =>
        // do nothing

      case e: Exception if reactor.exceptionHandler.isDefinedAt(e) =>
        reactor.exceptionHandler(e)
    }
    reactor.kill()
  }
  catch {
    case _: SuspendActorControl =>
      // do nothing (continuation is already saved)

    case e: Throwable =>
      terminateExecution(e)
      reactor.terminated()
      if (!e.isInstanceOf[Exception])
        throw e
  } finally {
    suspendExecution()
    this.reactor = null
    this.fun = null
    this.handler = null
    this.msg = null
  }
}
```

Listing 80: The `run`-method in **`ReactorTask`**

In the previous listings control flow exceptions are thrown which have to be caught somewhere. This happens in the `run`-method of the before mentioned **`ReactorTask`**, shown in Listing 80. The `run`-method of the **`ReactorTask`** runs the provided function, which is usually the `act` method via `dostart`, or handler in the case of an actor continuation, followed by terminating the actor, from which it was spawned. The **`SuspendActorControl`** exception is thrown to allow the function or handler in **`ReactorTask`** to finish and allow suspending the actor from which the **`ReactorTask`** is spawned, instead of terminating it.

```
/*@
  protected normal_behaviour:
    requires Lockset(S) ** (S contains this -* inv) ** this.initialized
    ensures Lockset(S) ** (S contains this -* inv)
    signals SuspendActorControl
*/
protected def react(handler: PartialFunction[Msg, Unit]): Nothing = {
  synchronized { drainSendBuffer(mailbox) }
  searchMailbox(mailbox, handler, false)
  throw Actor.suspendException
}
```

Listing 81: The `react`-method in **`Reactor`**

Besides allowing to send messages via `send`, **`Reactor`** allows to receive messages in an event-driven fashion, by using the `react`-method shown in Listing 81. The `react`-method is only ever called from within `act`, which, as mentioned before, executes on a single thread. It uses a `synchronized` block to take the messages from the guarded `sendbuffer` and to append them to the mailbox. Once the mailbox is up to date, the previously examined method `searchMailbox` is called to check if any message matches the patterns provided to `react` by the **`PartialFunction`** `handler`. The difference with the call via `send` is that `resumeOnSameThread` is passed as false, resulting in the continuation being scheduled on a new thread when a matching message is found.

`react` never returns, which is signalled by its return type of **`Nothing`**, as it always throws a **`SuspendActorControl`**. Therefore the rest of the computation performed by the actor should be contained in the continuation. The specification reflects the use of the lock and the execution ending in an exception.

Practically, `react` schedules a handler task when new messages are to be handled and then suspends the actor. cf. `receive` which we will discuss in depth in Listing 84, which always executes on the same thread, blocking when there are no matching messages. Because `react` never finishes, receiving messages continuously needs to manually use recursion or the `loop` construct, as shown in Section 6.3. We will look in depth at `loop` and the other combinators when examining the **`Actor`** trait in Section 6.5.5.

6.5.4 CanReply & ReactorCanReply & ReplyReactor

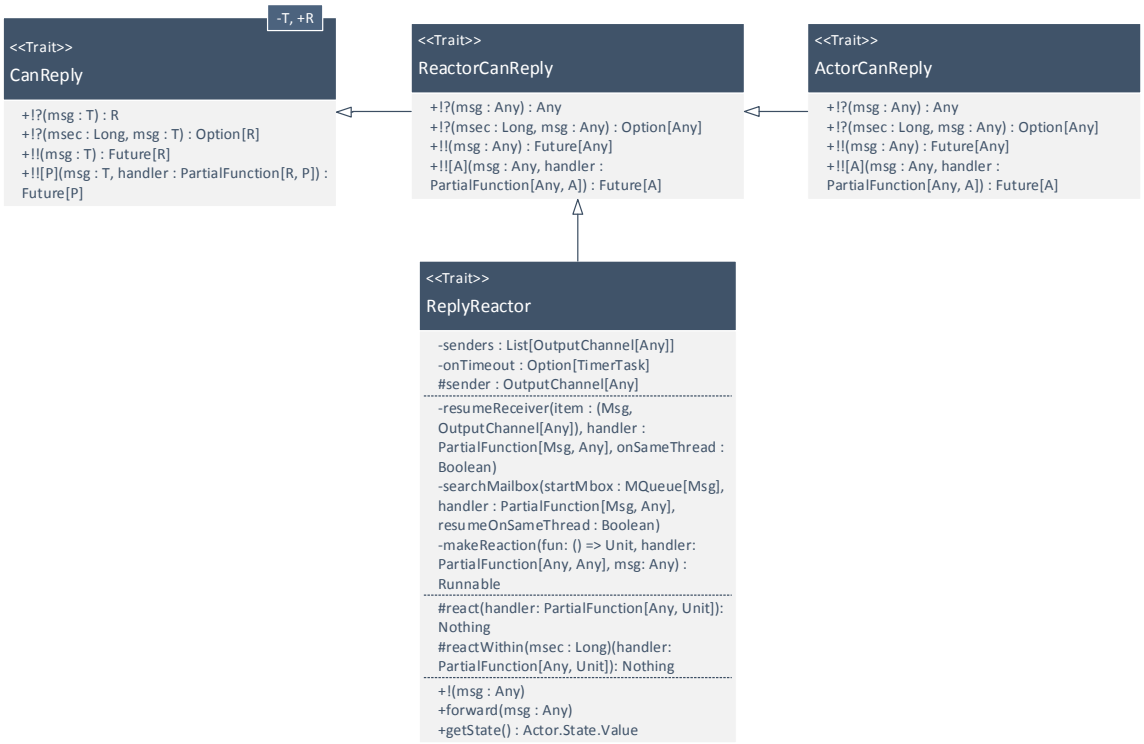


Figure 175: Class Diagram of CanReply, ReactorCanReply & ReplyReactor

The **CanReply** trait functions as an interface defining the `!!`- and `!?`-operators. These operators are then implemented in the **ReactorCanReply** and **ActorCanReply** traits, which we will be examining here, starting with the implementations of `!!` in Listing 82.

The basic specification of `!!` is relatively straightforward, as it merely specifies access to the **Actor** singleton and the need for messages to be immutable. It yields a predicate, abstracting permissions, as the method returns an anonymous implementation of **Future**[**T**], resulting in the same issues as with function closures. All the methods in the implementation of **Future**[**T**] are specified using this predicate.

The method functions by defining a **ReactChannel** and an anonymous implementation of an **OutputChannel** to itself. The **OutputChannel** is used as an argument to `send`, specifying a return address. The result of this is, that, when a reply to `msg` is sent using this channel, the **SyncVar** is set to this reply. The **ReactChannel** is a specialized **InputChannel** which proxies the given **Reactor**, but wraps all messages in a container with the channel. Using this convoluted method, it is possible to respond to a future returned by `!!`. In the returned **Future**[**T**], the `apply`- and the `isset`-methods are implemented using the same **SyncVar** as before. `respond` is implemented using the **ReactChannel**.

The specification of the `!?`-operator is not noteworthy, as it is trivially implemented using `!!`.

ActorCanReply extends /ReactorCanReply/ with timed variants of the operators, which we omit here.

```
/*@
  public normal_behaviour:
    requires Perm(\epsilon, Actor) ** \rec_imm(msg)
    ensures Perm(\epsilon, Actor)
    yields state
*/
def !![A](msg: Any, handler: PartialFunction[Any, A]): Future[A] = {
  val myself = Actor.rawSelf(this.scheduler)
  val ftch = new ReactChannel[A](myself)
  val res = new scala.concurrent.SyncVar[A]

  val out = new OutputChannel[Any] {
    def !(msg: Any) = {
      val msg1 = handler(msg)
      ftch ! msg1
      res set msg1
    }
    def send(msg: Any, replyTo: OutputChannel[Any]) = {
      val msg1 = handler(msg)
      ftch.send(msg1, replyTo)
      res set msg1
    }
    def forward(msg: Any) = {
      val msg1 = handler(msg)
      ftch forward msg1
      res set msg1
    }
    def receiver =
      myself.asInstanceOf[Actor]
  }

  this.send(msg, out)
  /*@ pred state = Perm(\epsilon, this) ** Perm(\epsilon, res) **
                   res.state<1> ** Perm(\epsilon, ftch) */
  new Future[A] {
    /*@ given state; requires state; ensures state; yields state */
    def apply() = {
      if (!isSet)
        fvalue = Some(res.get)

      fvalueTyped
    }
    /*@ given state; requires state; ensures state; yields state */
    def respond(k: A => Unit): Unit =
      if (isSet) k(fvalueTyped)
      else inputChannel.react {
        case any => fvalue = Some(any); k(fvalueTyped)
      }
    /*@ given state; requires state; ensures state; yields state */
    def isSet =
      !fvalue.isEmpty
    /*@ given state; requires state; ensures state; yields state */
    def inputChannel = ftch
  }
}
```

Listing 82: the !!-operator in ReactorCanReply

The methods implemented in **ReactorCanReply** depend on the ability to specify return addresses to send. This functionality is implemented partially in **Reactor**, but lacks the proper variant of searchMailBox to function. **ReplyReactor** implements this method – which is shown in Listing 83 along with the additional `senders`-field.

The implementation of `searchMailbox` is virtually identical to the one in **Reactor**, with the only difference being that it sets senders to the matching actor. In addition to in `searchMailbox`, `senders` is also set in `makeReaction`, but as there is no additional functionality besides that, we shall omit another listing. `senders` will be used primarily in the **Actor**-trait.

Besides the reply-functionality, **ReplyReactor** lays the groundwork for timed operations. As these add complication and little additional insight, we shall omit them from the case study.

```
  private[actors] var senders: List[OutputChannel[Any]] = List()


  /*@
    private normal_behaviour:
      requires
        startMBox.state<1> ** Perm(1, senders)
        mailbox.state<1> **
        (!(\exists int i; i >= 0 && i < startMbox.size(),
                startMbox.queue<alpha, startMbox.first> **
                  handler.isDefinedAt(alpha[i].msg)) =>
                          Lockset(S) ** (S contains this -* inv) **
                          this.initialized)
      ensures
        startMBox.state<1> **
        mailbox.state<1> **
        !(\exists int i; i >= 0 && i < startMbox.size(),
                startMbox.queue<alpha, startMbox.first> **
            handler.isDefinedAt(alpha[i].msg)) =>
               Lockset(S) * (S contains this -* inv)  **
        ((\exists int i; i >= 0 && i < startMbox.size(),
                startMbox.queue<alpha, startMbox.first> **
            handler.isDefinedAt(alpha[i].msg)) =>
                senders == alpha[i].session)
    also private exceptional_behaviour:
      ensures waitingFor == handler
      signals SuspendActorControl
  */
  private[actors] override def searchMailbox(startMbox: MQueue[Any],
                                             handler: PartialFunction[Any, Any],
                                             resumeOnSameThread: Boolean) {
    var tmpMbox = startMbox
    var done = false
    while (!done) {
      val qel = tmpMbox.extractFirst((msg: Any, replyTo: OutputChannel[Any]) => {
        senders = List(replyTo)
        handler.isDefinedAt(msg)
      })
      if (tmpMbox ne mailbox)
        tmpMbox.foreach((m, s) => mailbox.append(m, s))
      if (null eq qel) {
        synchronized {
          // in mean time new stuff might have arrived
          if (!sendBuffer.isEmpty) {
            tmpMbox = new MQueue[Any]("Temp")
            drainSendBuffer(tmpMbox)
            // keep going
          } else {
            waitingFor = handler
            // see Reactor.searchMailbox
            throw Actor.suspendException
          }
        }
      } else {
        resumeReceiver((qel.msg, qel.session), handler, resumeOnSameThread)
        done = true
      }
    }
  }
```

Listing 83: searchMailbox in ReplyReactor

6.5.5 Actor

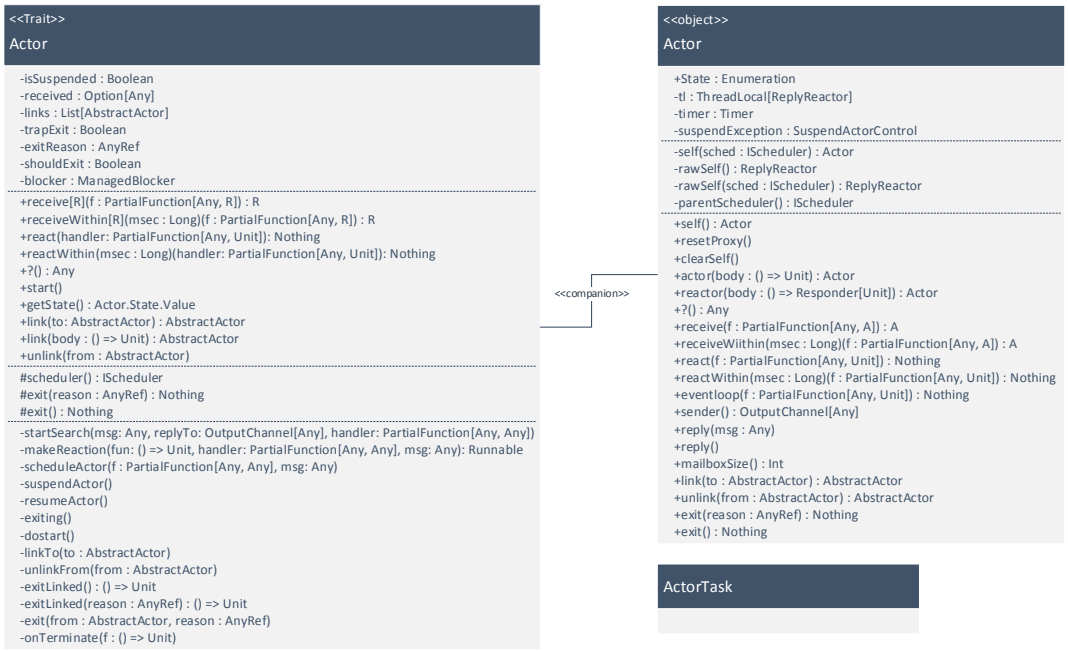


Figure 176: Actor Class Diagram

The **`Actor`**-trait resides at the bottom of the inheritance hierarchy and combines all the previously discussed functionality, along with the blocking `receive`-method and the linking functionality. The `receive`-method is shown in Listing 84.

```
/*@
  protected normal_behaviour:
    requires Lockset(S) ** (S contains this -* inv) ** this.initialized
    ensures Lockset(S) ** (S contains this -* inv)
*/
def receive[R](f: PartialFunction[Any, R]): R = {
  assert(Actor.self(scheduler) == this, "receive from channel belonging to other actor")

  synchronized {
    if (shouldExit) exit() // links
    drainSendBuffer(mailbox)
  }

  var done = false
  while (!done) {
    val qel = mailbox.extractFirst((m: Any, replyTo: OutputChannel[Any]) => {
      senders = replyTo :: senders
      val matches = f.isDefinedAt(m)
      senders = senders.tail
      matches
    })
    if (null eq qel) {
      synchronized {
        // in mean time new stuff might have arrived
        if (!sendBuffer.isEmpty) {
          drainSendBuffer(mailbox)
          // keep going
        } else {
          waitingFor = f
          isSuspended = true
          scheduler.managedBlock(blocker)
          drainSendBuffer(mailbox)
          // keep going
        }
      }
    } else {
      received = Some(qel.msg)
      senders = qel.session :: senders
      done = true
    }
  }
```

Listing 84: The receive method in Actor

The implementation and specification of receive is very similar to that of `react`, but when no matching message is found, the current thread is blocked, using the scheduler and a **`ManagedBlocker`**. The override of `startSearch` – shown in Listing 85 – unblocks the actor to handle incoming messages, using the `resumeActor`-method, which is a wrapper for `notify()`.

```
    /*@ inherited from Reactor.startSearch
        private normal_behaviour:
          requires Lockset(S) ** S contains this ** inv ** \rec_imm(msg)
          requires handler != Reactor.WaitingForNone
          ensures Lockset(S) ** S contains this ** inv
          ensures cs \result () <cs> |=> cs
          yields cs
*/
        private[actors] override def startSearch(msg: Any, replyTo: OutputChannel[Any], handler: PartialFunction[Any, Any]) =
          if (isSuspended) {
            () => synchronized {
              mailbox.append(msg, replyTo)
              resumeActor()
            }
          } else super.startSearch(msg, replyTo, handler)
```

Listing 85: The startsearch method in Actor

The **`Actor`** companion object houses all the syntax and wrapper methods which allow for the practical use of the actors library, as demonstrated in Section 6.3. Because of this it adds very little implementation, as it mostly proxies instances of **`Actor`**, and we shall cover relatively little of the methods defined. Something that is of importance is the definition of the State enumeration, which we show in Listing 86.

```
        object State extends Enumeration {
          val New,
              Runnable,
              Suspended,
              TimedSuspended,
              Blocked,
              TimedBlocked,
              Terminated = Value
        }
```

Listing 86: Actor States

The following states are defined for actors:

- **New**: The actor has not yet been started.
- **Runnable**: The actor is executing.
- **Suspended**: The actor is suspended inside a `react`.
- **TimedSuspended**: The actor is suspended inside a `reactWithin`.
- **Blocked**: The actor is suspended inside a `receive`.
- **TimedBlocked**: The actor is suspended inside a `receiveWithin`.
- **Terminated**: The actor has been terminated.

Linking functionality, as mentioned in Section 6.3, is a nonessential and advanced feature, which we shall currently omit from our case study.

6.5.6 Issues Encountered

A practical issue to consider when specifying multi-threaded applications with *Separation Logic*, is the notion of interleaving. Take the following example specification:

```
class Simple
{
    var x = 0
    /*@
        inv = PointsTo(x, 1, _)
    */

    /*@
        requires Lockset(S) ** (S contains this -* inv) ** initialized
        ensures Lockset(S) ** (S contains this -* inv) ** PointsTo(x, 1, \old(x)+1)
    */
    public def inc() : Int
    {
        synchronized
        {
            x = x+1
            x
        }
    }

    /*@
        requires Lockset(S) ** (S contains this -* inv) ** initialized
        ensures Lockset(S) ** (S contains this) -* inv ** PointsTo(x, 1, 1)
    */
    public def reset()
    {
        synchronized
        {
            x = 1
        }
    }
}
```

Listing 87: An Example Specification

In Listing 87 we see a method `inc` which increments `x` by 1 and returns it and a method `reset` which sets it to one, both within a `synchronized`-block such that these methods cannot interfere in a multi-threaded scenario. Furthermore we see the resource invariant protecting write access to `x` and the standard pre- and postconditions for situations involving locking. We also see the effect of the methods mentioned in the postcondition; for `inc`, `x` points to its previous value plus one, and for `reset`, `x` points to 1.

Unfortunately, the specification given in Listing 87 is incorrect, and this has to do with interleaving. Multithreaded programs are interleaved in atomic chunks, the base chunk being a single atomic operation and larger ones being formed by locks. In this case, the synchronized blocks are atomic, but the specifications are not part of it. This means that after the synchronized block has finished executing, other statements can be interleaved, for instance in this case modifying the value of x. We have visualized this in Fig. 177. Recently however, progress has been made in this area by Blom, Huisman, and Zaharieva-Stojanovski [11], where abstract program actions are logged in a so-called history;

using such a history we could in this case have at least specified in our postcondition that at one point in the history, the increment has occurred.

1. Execution of `inc()` starts.
2. Precondition of `inc()` holds.
3. Synchronized block executes, $x = 1$.
4. Execution of `reset()` is interleaved.
5. Precondition of `reset()` holds.
6. Synchronized block executes, $x = 0$.
7. Postcondition of `reset()` holds.
8. Execution returns to the postcondition of `inc()`.
9. Postcondition is violated.

Figure 177: Interleaving

The end result is that our postcondition cannot say anything regarding the value of `x`, restricting us to the resource invariant. While certain specifications can be moved into the locked region as assertions as in Listing 88, this still only has meaning within the synchronized block, making it hard to keep track of the state of the program.

```
class Simple
{
    var x = 0
    /*@
        inv = PointsTo(x, 1, _)
    */

    /*@
        requires Lockset(S) ** (S contains this -* inv) ** initialized
        ensures Lockset(S) ** (S contains this -* inv)
    */
    public def inc() : Int
    {
        synchronized
        {
            x = x+1
            /*@ assert PointsTo(x, 1, \old(x)+1) */
            x
        }
    }

    /*@
        requires Lockset(S) ** (S contains this -* inv) ** initialized
        ensures Lockset(S) ** (S contains this) -* inv
    */
    public def reset()
    {
        synchronized
        {
            x = 1
            /*@ assert PointsTo(x, 1, 1) */
        }
    }
}
```

Listing 88: The Example with Asserts

These types of issues regarding the specification of (partially) concurrent methods, means it requires a very detailed mental model, of the concurrent interactions inside a program, to specify it.

An issue regarding the use of separation logic, is that methods that have overrides in subclasses may have stricter preconditions in the overridden variant, because they might touch more state. This seems in conflict with the general principle that preconditions may only be weakened.

Another issue is the specification of interfaces, which is often preferred. As separation logic is implementation-specific by definition, interfaces can only be specified either without using the separation logic assertions, or by wrapping everything inside of predicates. It also raises the question whether locking can and should be specified on interfaces.

6.6 Conclusions

Using our specification language, we have managed to specify a significant portion of the actors library. While there were no unsurmountable issues, finding the proper level of abstraction remains an issue in these types of specifications and is complicated by the low-level nature of specifications using separation logic. Furthermore, while using separation logic will allow you to prove race-freedom, it requires very detailed knowledge of the thread-interactions in the program beforehand. In this case any proofs regarding race-freedom are complicated by the fact that much of the concurrency-related work is delegated to the scheduler, which resides in a core *Java* library which (currently) lacks these types of specifications. The library did provide a means to demonstrate our operator on functions with closures, but it also highlighted the difficulties in cases of nested function definitions and the fact that a similar approach is still lacking for anonymous classes and their methods and unsugared instances of function objects, such as instances of `PartialFunction`.

7 Conclusions & Future Work

7.1 Summary

In this work we developed a partial approach to the formal specification of *Scala* programs using separation logic. We started this in Section 2 and Section 3 by providing a compact overview of program verification using separation logic and the *Scala* programming language. While the explanations in Section 2 are limited and do not cover the exact approach we eventually decided to use in Section 5, it helps in understanding the basics, without immediately diving into complex logic and mathematics. Similarly, Section 3 only provides a very shallow look at the *Scala* programming language; Enough to develop a taste for it and to understand this thesis, but not nearly covering the entire scope of the language.

In Section 4 we set out to explain and use the program context approach [37] to create a small-step semantics for a subset of *Scala* we called *Scala Core*. This approach turned out to be well-suited to describe the semantics of the language, but was at times overly verbose, by having to manage the context in detail and thus requiring many additional reduction steps. In addition, with our language being an expression language, steps to reduce the verbosity, such as head reduction for pure expressions as used by Krebbers and Wiedijk [37] could not be used. Therefore, while this approach sufficed for our purposes and explicitly demonstrated all steps of the language semantics, a more abstract approach as used by Birkedal et al. [10] could have been more compact and more practical in preparation for the work on the separation logic in Section 5.

Additionally, while focusing on a particular subset of a language to reduce the scope is necessary, in our case we partially lost what makes *Scala*, *Scala*, in the creation of *Scala Core*, e.g. pattern matching and algebraic datatypes. This had a positive side in that our work is more generically applicable for any expression language with first class functions and reducing the clutter. However, it can be said that the clutter of language features is a defining aspect of the *Scala* programming language. At the same time, our model language is quite powerful and most missing elements of *Scala* can be translated to it. However, this also requires a translation from contracts established with regards to *Scala* to ones with the same meaning to the *Scala Core* translation. Finally, we also abstracted away from the intricate typing system used in *Scala*, once again helping our focus, but distancing us from what we set out to do, which is adapt separation logic specifically to *Scala*.

Then in Section 5 we set out to create a separation logic which is applicable to *Scala Core*. We used a highly abstract approach, developed by Birkedal et al. [10], to construct a semantics for a type system and a separation logic. While this is one of the few applicable approaches to deal with complex matters such as nested Hoare triples, which we require to deal with first class functions and closures, it creates an incredibly dense read. Additionally, the lack of small practical examples, makes it difficult to reach an intuitive understanding at first, or even second glance. Furthermore the aspect of concurrency is only briefly touched upon and hardly a mention is made of soundness.

Finally in Section 6 we provided a case study on the practical viability of *Scala* programs. We demonstrate the difficulty of specifying nested functions and the general issue of abstraction levels in writing specifications. Missing however, is a clear link from the specification language used to the separation logic developed in Section 5. Also, while demonstrating specification of an actual library is interesting, we ran into the issue of it simply being too big to serve as a practical overview and we would probably have benefited from multiple smaller examples.

All in all, we delivered a viable outline to the specification of expression languages with first class functions and closures, which can be applied to the *Scala* programming language. Much fleshing out, of both the model language and the separation logic, is still required to reach the full extent of *Scala*.

7.2 Contribution

In our work we have made the following contributions:

1. We have defined a model language with a formal semantics for a subset of *Scala* using a novel program context approach, which allows first-class functions, which are executed in their defining context, allowing full lexical

closures.

2. We have designed a separation logic for use with the model language with a Kripke-style many-worlds semantics, which uses nested Hoare triples to specify first-class functions, along with supporting classes, exceptions and the basic framework for permissions which allows for the future extension to multithreading.
3. We have done a case study on the practical applicability of specification with this logic on a real-world library.

7.3 Future Work

For future work there remains a comprehensive proof of the soundness of our approach, a proper extension of the separation logic to multithreading, as well as the more close integration of the *Scala* type system. It would also be interesting to see how our approach would deal with first class functions as introduced in *Java* 8.0, which is statement and not expression based and which only allows `final` fields to be referred to in functions, similarly to their abstract classes. Another interesting thing to look at is the approach to locks by Buisse, Birkedal, and Støvring [15], which more tightly integrates with our approach to the separation logic, while providing additional benefits at the cost of a sound magic wand operator. There remains also the potential work to be done on automated tooling support, which would take our approach from the theoretical realm to practical applications.

References

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[2] Amal Jamil Ahmed. "Semantics of Types for Mutable State". PhD thesis. 2004.

[3] Pierre America and Jan Rutten. "Solving reflexive domain equations in a category of complete metric spaces". In: *Journal of Computer and System Sciences* 39.3 (1989), pp. 343–375. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/0022-0000(89)90027-5. URL: http://www.sciencedirect.com/science/article/pii/0022000089900275.

[4] Afshin Amighi et al. "Permission-Based Separation Logic for Multithreaded Java Programs". In: *CoRR* abs/1411.0851 (2014). URL: http://arxiv.org/abs/1411.0851.

[5] Afshin Amighi et al. "The VerCors project: setting up basecamp". In: *Proceedings of the sixth workshop on Programming languages meets program verification*. PLPV '12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 71–82. ISBN: 978-1-4503-1125-0. DOI: 10.1145/2103776.2103785. URL: http://doi.acm.org/10.1145/2103776.2103785.

[6] Andrew W. Appel and David McAllester. "An Indexed Model of Recursive Types for Foundational Proof-carrying Code". In: *ACM Trans. Program. Lang. Syst.* 23.5 (Sept. 2001), pp. 657–683. ISSN: 0164-0925. DOI: 10.1145/504709.504712. URL: http://doi.acm.org/10.1145/504709.504712.

[7] Russell Atkinson and Carl Hewitt. "Synchronization in actor systems". In: *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. Los Angeles, California: ACM, 1977, pp. 267–280. DOI: http://doi.acm.org/10.1145/512950.512975.

[8] Mike Barnett et al. "Verification of Object-Oriented Programs with Invariants". In: *JOURNAL OF OBJECT TECHNOLOGY* 3 (2004), p. 2004. DOI: 10.1.1.10.4654.

[9] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. "BI-hyperdoctrines, Higher-order Separation Logic, and Abstraction". In: *ACM Trans. Program. Lang. Syst.* 29.5 (Aug. 2007). ISSN: 0164-0925. DOI: 10.1145/1275497.1275499. URL: http://doi.acm.org/10.1145/1275497.1275499.

[10] Lars Birkedal et al. "Step-indexed Kripke Models over Recursive Worlds". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 119–132. ISSN: 0362-1340. DOI: 10.1145/1925844.1926401. URL: http://doi.acm.org/10.1145/1925844.1926401.

[11] S. C. C. Blom, M. Huisman, and M. Zaharieva-Stojanovski. *History-based Verification of Functional Behaviour of Concurrent Programs*. Technical Report TR-CTIT-15-02. Enschede: Centre for Telematics and Information Technology, University of Twente, Mar. 2015.

[12] Richard Bornat et al. "Permission accounting in separation logic". In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 259–270. ISSN: 0362-1340. DOI: 10.1145/1047659.1040327. URL: http://doi.acm.org/10.1145/1047659.1040327.

[13] Gilad Bracha and William Cook. "Mixin-based Inheritance". In: ACM Press, 1990, pp. 303–311.

[14] N. G. De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem". In: *INDAG. MATH* 34 (1972), pp. 381–392.

[15] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. *A Step-Indexed Kripke Model of Separation Logic for Storable Locks*. 2011.

[16] R. Burstall. "Some techniques for proving correctness of programs which alter data structures". In: *Machine Intelligence 6*. 1971, pp. 23–50.

[17] Richard Carlsson and Hakan Millroth. *Towards a deadlock analysis for Erlang programs*. 1997.

[18] Nathaniel Charlton. "Hoare Logic for Higher Order Store Using Simple Semantics". English. In: *Logic, Language, Information and Computation*. Ed. by LevD. Beklemishev and Ruy de Queiroz. Vol. 6642. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 52–66. ISBN: 978-3-642-20919-2. DOI: 10.1007/978-3-642-20920-8_10. URL: http://dx.doi.org/10.1007/978-3-642-20920-8_10.

[19] Vincent Cremet. "Foundations for SCALA". PhD thesis. Lausanne: IC, 2006. DOI: 10.5075/epfl-thesis-3556.

[20] Vincent Cremet et al. "A Core Calculus for Scala Type Checking". In: *Proceedings of MFCS 2006*. Lecture Notes in Computer Science. Stará Lesná, Slovak Republic, 2006.

[21] Dino Distefano and Matthew J. Parkinson J. "jStar: towards practical verification for java". In: *SIGPLAN Not.* 43.10 (Oct. 2008), pp. 213–226. ISSN: 0362-1340. DOI: 10.1145/1449955.1449782. URL: http://doi.acm.org/10.1145/1449955.1449782.

[22] Switzerland École Polytechnique Fédérale de Lausanne (EPFL) Lausanne. *The Scala Programming Language*. URL: http://www.scala-lang.org.

[23] Manuel Fähndrich. "Static verification for code contracts". In: *Proceedings of the 17th international conference on Static analysis*. SAS'10. Perpignan, France: Springer-Verlag, 2010, pp. 2–5. ISBN: 3-642-15768-8, 978-3-642-15768-4. URL: http://dl.acm.org/citation.cfm?id=1882094.1882096.

[24] Cormac Flanagan et al. "Extended static checking for Java". In: *SIGPLAN Not.* 37.5 (May 2002), pp. 234–245. ISSN: 0362-1340. DOI: 10.1145/543552.512558. URL: http://doi.acm.org/10.1145/543552.512558.

[25] R. W. Floyd. "Assigning Meaning to Programs". In: *Proceedings of the Symposium on Applied Maths*. Vol. 19. AMS, 1967, pp. 19–32.

[26] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software.* ISBN-10: 0201633612 ISBN-13: 978-0201633610. Addison-Wesley, Mar. 1995, p. 128.

[27] James Gosling et al. *The Java Language Specification*. Oracle America, Inc., 2013. URL: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf.

[28] Alexey Gotsman et al. "Local Reasoning for Storable Locks and Threads". In: *APLAS*. 2007, pp. 19–37.

[29] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[30] C. A. R. Hoare. "The origin of concurrent programming". In: ed. by Per Brinch Hansen. New York, NY, USA: Springer-Verlag New York, Inc., 2002. Chap. Towards a theory of parallel programming, pp. 231–244. ISBN: 0-387-95401-5. URL: http://dl.acm.org/citation.cfm?id=762971.762978.

[31] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. "A Theory of Indirection via Approximation". In: *SIGPLAN Not.* 45.1 (Jan. 2010), pp. 171–184. ISSN: 0362-1340. DOI: 10.1145/1707801.1706322. URL: http://doi.acm.org/10.1145/1707801.1706322.

[32] Gérard Huet. "The Zipper". In: *J. Funct. Program.* 7.5 (Sept. 1997), pp. 549–554. ISSN: 0956-7968. DOI: 10.1017/S0956796897002864. URL: http://dx.doi.org/10.1017/S0956796897002864.

[33] M. Huisman. "Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle". PhD thesis. University of Nijmegen, 2001.

[34] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (May 2001), pp. 396–450.

[35] Samin Ishtiaq and Peter W. O'Hearn. *BI as an Assertion Language for Mutable Data Structures*. 2000.

[36] C. B. Jones. "Tentative steps toward a development method for interfering programs". In: *ACM Trans. Program. Lang. Syst.* 5.4 (Oct. 1983), pp. 596–619. ISSN: 0164-0925. DOI: 10.1145/69575.69577. URL: http://doi.acm.org/10.1145/69575.69577.

[37] Robbert Krebbers and Freek Wiedijk. "Separation Logic for Non-local Control Flow and Block Scope Variables." In: *FoSSaCS*. Vol. 7794. Lecture Notes in Computer Science. Springer, 2013, pp. 257–272. ISBN: 978-3-642-37074-8. URL: http://dblp.uni-trier.de/db/conf/fossacs/fossacs2013.html#KrebbersW13.

[38] Gary T. Leavens and Yoonsik Cheon. *Design by Contract with JML*. 2006.

[39] Bertrand Meyer. "Applying "Design by Contract"". In: *IEEE Computer* 25.10 (1992), pp. 40–51.

[40] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, I". In: *Inf. Comput.* 100.1 (1992), pp. 1–40. ISSN: 0890-5401. DOI: http://dx.doi.org/10.1016/0890-5401(92)90008-4.

[41] Mayur Naik, Alex Aiken, and John Whaley. "Effective static race detection for Java". In: *SIGPLAN Not.* 41.6 (June 2006), pp. 308–319. ISSN: 0362-1340. DOI: 10.1145/1133255.1134018. URL: http://doi.acm.org/10.1145/1133255.1134018.

[42] Mayur Naik et al. "Effective static deadlock detection". In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 386–396. ISBN: 978-1-4244-3453-4. DOI: 10.1109/ICSE.2009.5070538. URL: http://dx.doi.org/10.1109/ICSE.2009.5070538.

[43] Martin Odersky. *The Scala Language Reference Version 2.9 Draft*. Programming Methods Laboratory, EFPL, 2011. URL: http://www.scala-lang.org/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf.

[44] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. 2nd. USA: Artima Incorporation, 2011. ISBN: 0981531644, 9780981531649.

[45] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Proceedings of the 15th International Workshop on Computer Science Logic*. CSL '01. London,

UK, UK: Springer-Verlag, 2001, pp. 1–19. ISBN: 3-540-42554-3. URL: http://dl.acm.org/citation.cfm?id=647851.737404.

[46] Susan S. Owicki and David Gries. "An Axiomatic Proof Technique for Parallel Programs I". In: *Acta Inf.* 6 (1976), pp. 319–340.

[47] Matthew J. Parkinson. *Local reasoning for Java*. Tech. rep. UCAM-CL-TR-654. University of Cambridge, Computer Laboratory, Nov. 2005. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-654.pdf.

[48] Matthew J. Parkinson and Gavin M. Bierman. "Separation logic, abstraction and inheritance". In: *POPL*. 2008, pp. 75–86.

[49] James L. Peterson. "Petri Nets". In: *ACM Comput. Surv.* 9.3 (1977), pp. 223–252. ISSN: 0360-0300. DOI: http://doi.acm.org/10.1145/356698.356702.

[50] G. D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN-19. University of Aarhus, 1981. URL: http://citeseer.ist.psu.edu/plotkin81structural.html.

[51] Bernhard Reus and Thomas Streicher. "About Hoare Logics for Higher-Order Store." In: *ICALP*. Ed. by Luís Caires et al. Vol. 3580. Lecture Notes in Computer Science. Springer, Sept. 6, 2005, pp. 1337–1348. ISBN: 3-540-27580-0. URL: http://dblp.uni-trier.de/db/conf/icalp/icalp2005.html#ReusS05.

[52] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9. URL: http://dl.acm.org/citation.cfm?id=645683.664578.

[53] J. Schwinghammer et al. "Nested Hoare Triples and Frame Rules for Higher-order Store". In: *Proceedings of CSL 2009*. Apr. 2009.

[54] Willem Visser et al. "Model Checking Programs". In: *AUTOMATED SOFTWARE ENGINEERING JOURNAL*. 2000, pp. 3–12.

[55] Erlang Web. *Erlang – What is Erlang*. URL: http://www.erlang.org/faq/introduction.html#id49610.