VRIJE
UNIVERSITEIT
BRUSSEL

Thesis submitted in partial fulfilment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering:
Computer Science

INCREMENTAL THREAD-MODULAR STATIC ANALYSIS FOR CONCURRENT PROGRAMS WITH FUTURES AND ATOMS

Jens Van der Plas

2018-2019

Promotor: Prof. Dr. Coen De Roover
Advisor: Dr. Quentin Stiévenart

Sciences & Bio-Engineering Sciences

ABSTRACT

Building concurrent programs is hard as the use of multiple threads introduces nondeterminism in their executions. This nondeterminism leads to bugs that are often subtle to detect and difficult to resolve. Having tool support to detect such program defects may significantly reduce the effort required to resolve concurrency-related bugs.

In recent years, multiple programming paradigms and programming constructs have been developed to facilitate the development of concurrent programs. However, incorrect use of these constructs may cause bugs. Atomic variables, or atoms, are an example of such constructs and can be found in modern programming languages such as Clojure. Atoms provide concurrent but race-free updates to shared state. In this dissertation, we apply the *Abstracting Abstract Machines* (AAM) technique of Van Horn & Might, a well-known technique to build abstract interpreters, to a concurrent language containing futures and atoms. We formalise this language using a parallel CESK machine which is then systematically abstracted, obtaining the first AAM-based abstract interpreter able to analyse a concurrent language with atoms.

An important aspect of an abstract interpreter is its performance. To be sound, an abstract interpreter for concurrent languages must account for all possible thread interleavings in a concurrent program. Recent work has already introduced *thread-modular* analysis techniques to reduce the worst-case time complexity of such analyses. Recently, Stiévenart has introduced ModConc, a general approach to the design of thread-modular analyses. The approach results in an analysis that alternates between two phases: an intra-process analysis phase analyses each thread in isolation, while an intra-process analysis phase tracks the behaviour of all threads and invokes intra-process analyses where necessary. In this dissertation, we improve this algorithm by introducing incrementality. We construct the results of the intra-process analysis in an incremental way by allowing results from prior invocations to be reused. This way, we aim to reduce the analysis time needed by the abstract interpreter, making the analysis more scalable to large, real-world programs. Furthermore, we investigate additional optimisations that are possible.

We evaluate our incremental analysis algorithm on a set of benchmark programs and show that it reduces the average analysis time for most of our benchmark programs compared to the original non-incremental algorithm. For several benchmarks, we find significant reductions of the average analysis time, ranging from 40% up to 63%. On multiple occasions, the size of the resulting abstract state graph is reduced as well, leading to fewer spurious paths in the result graph. We find that IncAtom succeeds in reusing large parts of previously calculated results for some benchmarks.

In summary, in this dissertation, we introduce an AAM-based incremental thread-modular analysis and apply this to a concurrent language with futures and atoms. By doing so, we not only extend the application domain of AAM-based analyses but also introduce a new algorithm that improves the performance of thread-modular analyses, thereby improving their scalability and making them applicable to increasingly large programs.

ACKNOWLEDGEMENTS

First of all, I would like to thank my promotor Prof. Dr. Coen De Roover, whose enthusiasm and confidence in this work helped me to successfully complete this thesis. Second, I would like to thank my advisor, Dr. Quentin Stiévenart, for guiding me through this year. His expertise, useful remarks and insights, as well as his continuous guidance, were of utmost importance for the success of this work. He also provided me with useful pointers and ideas and always freed up some of his time whenever I burst into his office, for which I am most grateful. I would also like to thank all members of the Software Languages Lab for their interesting questions and remarks during the intermediate thesis presentations, which allowed me to reflect on the work I was doing.

I would also like to thank my family for their continuous support during this year, and always. Their permanent encouragement and positive energy enabled me to focus on this work under the best of circumstances. I would also like to thank them for giving me some useful hints and for helping me read the unreadable.

Finally, I want to express my gratitude to all people who have read this dissertation in advance and gave useful remarks to further improve this writing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

In this dissertation, we present two contributions to the state of the art in static analysis of concurrent programs. Our first contribution is the application of the *Abstracting Abstract Machines* (AAM) technique of Van Horn & Might (2010) to a concurrent higher-order language containing futures and atomic variables (atoms). By applying this technique to a language with these programming constructs, we enable the analysis of concurrent higher-order programs that make use of these programming constructs. Second, to reduce the worst-case time complexity of the analysis, we follow Cousot & Cousot (1977)'s strategy of *modular analysis*, obtaining a thread-modular analysis. ModConc, a design method for thread-modular AAM-based analyses, has recently been presented by Stiévenart (2018). We find however that ModConc analyses may duplicate work and we present an incremental variant that avoids this duplication and hence overcomes the corresponding performance overhead.

1.1. Motivation and Research Context

Modern-day society has come to rely on software, which has significantly impacted our quality of life through the automation of numerous processes. To support these applications, computer manufacturers continuously invest effort to increase the performance of their systems. However, it has become increasingly difficult for CPU manufacturers to further increase the performance of individual processor cores (Etiemble, 2018). Instead, nowadays, multiple processor cores are placed together on a single chip, further increasing the computing power of contemporary processors.

Multicore processors have become increasingly common and many programming languages provide support for writing concurrent, multithreaded programs. However, the use of multiple threads introduces nondeterminism in the execution of a program: the interleaving of instructions from different threads may differ from one execution to another. This behaviour has introduced new types of bugs, such as deadlocks and data races. Due to the nondeterministic nature of concurrent programs, a concurrency-related bug may only cause problems during some executions of a program but remain harmless during others. Hence, concurrency bugs typically are subtle and hard to find. Due to the importance of software in modern-day society, such bugs can cause major damage when left unsolved. A prominent example is the Northeast blackout of 2003, in which a concurrency-related software bug caused a power outage in a vast part of the United States and Canada, affecting approximately 50 million people and resulting in instabilities of the power grid that lasted for weeks (Chadwick, 2013; Poulsen, 2004). Clearly, tool support for detecting program defects and for ensuring quality is indispensable.

Static analysis techniques are used to infer program properties at compile time. They can give strong guarantees about program behaviour and hence provide a strong means to detect program defects. One technique to perform static analysis is abstract interpretation. An abstract interpreter works similarly to a concrete interpreter but uses an approximation of a program's semantics. This way, a program's behaviour can be approximated and program properties can be derived. Van Horn & Might (2010) have presented Abstracting Abstract Machines (AAM), a systematic approach to convert a concrete interpreter to an abstract interpreter, to facilitate the construction of abstract interpreters.

Recent work has applied the AAM technique to concurrent higher-order programs with dynamic thread creation (Might & Van Horn, 2011; Stiévenart et al., 2015; Stiévenart, 2018). However, although the AAM technique has already been applied to several concurrency constructs, the technique has not yet been applied to atoms, which provide a way to update shared state without the risk of race conditions. Hence, current static analysers built using the AAM technique are unable to analyse programs containing atoms.

Building scalable and well-performing static analysers for concurrent programs is difficult. To make analyses of concurrent higher-order programs scale, often, thread-modular designs have been proposed. These analysis designs analyse the different threads of a program in isolation, resulting in lower analysis times. Stiévenart (2018) presents ModConc, a general a general approach to the design of thread-modular analyses. The approach results in an algorithm based on two alternating phases. An intra-process analysis phase analyses each thread in isolation and an inter-process analysis phase decides on the threads for which an intra-process analysis must be run. Both analysis phases perform a fixed-point computation and therefore, it is possible that an intra-process analysis is run multiple times for the same thread. When this is the case, the algorithm discards any result that was obtained from a prior invocation of the intra-process analysis phase for the given thread.

1.2. Objectives and Contributions

The goal of this dissertation is twofold. Our first objective is to apply the AAM technique of Van Horn & Might (2010) to a concurrent language with futures and atoms. This will enable our static analyser to analyse concurrent programs making use of these two programming constructs. Our second objective is to alter the thread-modular analysis algorithm of Stiévenart (2018) to make it incremental with regard to the computation of the analysis result for a single thread, that is, to make it possible for results from a prior invocation of the intra-process analysis of a thread to be reused upon reinvocation. By reusing previously computed results, we aim at decreasing the analysis time to obtain analyses that scale better to large, real-world programs. To achieve these objectives, we make the following contributions:

- We present an abstract interpreter for a concurrent language with futures and atoms that is able to handle dynamic thread creation. We first formalise a concrete interpreter for this language using a parallel CESK machine and abstract it using the AAM technique of Van Horn & Might (2010). This results in the first AAM-based analysis able to analyse programs containing atoms. We implement our abstract interpreter in the Scala-AM framework, which is designed to facilitate the construction of abstract interpreters. The Scala-AM framework has a modular design and consists out of multiple components that can easily be reused.

- We formalise thread interference for the parallel language by means of *effects* and present ModAtom, an adaptation of the thread-modular analysis algorithm of Stiévenart (2018) to make it applicable to our concurrent language with futures and atoms. The effects generated during the intra-process analysis of a thread are used by the inter-process analysis to decide on the threads that need to be reanalysed.
- We present IncAtom, a new thread-modular analysis that computes the results for individual threads incrementally. To this end, we redesign ModAtom and introduce a fine-grained effect tracking mechanism that tracks the behaviour of individual threads and allows the intra-process analysis of a thread to be restarted exactly at the point where it may be influenced by another thread. We present two optimisations that may further reduce the analysis time of IncAtom.
- We implement our analysis algorithms in the Scala-AM framework. Due to the modular design of the framework, the comparative evaluation of different abstract interpreters becomes more trustworthy: since parts of the interpreter can be reused, a change in performance is solely due to the changes in the parts that have been modified. Hence, this increases the reliability of comparative studies performed using the framework.
- We present a thorough evaluation of our contributions. We empirically demonstrate the soundness of IncAtom and compare the behaviour of the analysis to ModAtom using the analysis time and size of the resulting state graph as metrics; we also evaluate the extent to which IncAtom is able to reuse previously computed results and asses the proposed optimisations using several metrics. For our evaluation, we use a set of 28 concurrent, higher-order benchmark programs. We evaluate the behaviour of IncAtom and compare its performance to its non-incremental counterpart, ModAtom.

1.3. Overview of the Dissertation

This dissertation is structured as follows. In Chapter 2, we present an introduction to static analysis. In Section 2.1, first, a general overview of static analysis is given. Thereafter, we discuss abstract interpretation, the analysis technique used throughout this dissertation, in Section 2.2. We conclude the chapter by discussing the application of static analyses to concurrent languages in Section 2.3.

In Chapter 3, we iteratively construct an abstract interpreter for λ_α, a concurrent language with atoms. We start from a simple sequential base language, λ_0, which is then successively extended with support for futures and atoms, resulting in λ_ϕ and λ_α respectively. We formalise these languages using an (abstract) parallel CESK machine and provide, in each step, both concrete and abstract semantics.

After having presented a non-modular abstract interpreter for λ_α, we present algorithms to perform a thread-modular analysis in Chapter 4. To do so, we formalise thread interference in λ_α in Section 4.1, obtaining λ_ϵ. Based on this formalisation, ModAtom, an algorithm for a thread-modular analysis of λ_α, is presented in Section 4.2. In Section 4.3, we present IncAtom, an incrementalised version of ModAtom. Afterwards, we discuss some further possible optimisations in Section 4.4 and present some general remarks in Section 4.5. We conclude by using the different algorithms to analyse a small example program in Section 4.6, illustrating the particularities of every algorithm.

The implementation of the abstract interpreter for λ_α, as well as the algorithms for the thread-modular analysis, are described in Chapter 5. Our implementation is incorporated into the

Scala-AM framework, which is presented in Section 5.1. Finally, the implementation of the semantics of λ_α and of the modular analysis algorithms are explained in Sections 5.2 and 5.3 respectively.

Our implementation is thoroughly evaluated in Chapter 6. In Section 6.1, we first evaluate soundness of IncAtom, after which we study the algorithm's behaviour with respect to several metrics in Section 6.2.

In Chapter 7, we present related work in the different research domains related to this dissertation, such as static analysis for concurrent programs and incremental static program analysis.

Finally, in Chapter 8, we present an overall conclusion and set out possible research directions for future work.

2 INTRODUCTION TO STATIC ANALYSIS

This chapter introduces static analysis. First, a general overview of static analysis is presented in Section 2.1. Thereafter, in Section 2.2, Abstract Interpretation, the static analysis technique used throughout this dissertation, is presented. In Section 2.3, it is explained how abstract interpretation can be applied to concurrent languages as well as how this can be made scalable using modular analysis techniques.

2.1. Fundamentals of Static Analysis

In this section, the fundamentals of static analysis are introduced. Static analyses are used to infer behavioural properties of programs without needing to run those programs. Hence, static analysers are typically used at compile time. An example of a program property that can be established by means of static analyses is the absence of certain types of defects (bugs). Unlike testing, which can only prove the presence of certain defects, static analyses can also provide guarantees about their absence, resulting in strong guarantees about the program's behaviour. In general, a static analyser tries to establish whether a program exhibits a certain property. If this is the case, the analyser will answer *yes*; otherwise, the analyser will answer *no*.

Unfortunately, static analysers are limited by a general limitation of Turing machines: any non-trivial program property is undecidable (Rice, 1953). As a consequence, it is impossible to write a static analyser that, for any program, can decide whether the program exhibits a certain property. Therefore, it is important for every static analysis technique to avoid this decidability problem that arises when analysing a given program for any property it may exhibit.

To resolve this issue regarding decidability, static analysis techniques *approximate* the actual solution, that is, the techniques sacrifice some *precision* to avoid undecidability. As a result of this loss in precision, a static analyser may encounter situations in which it cannot answer *yes* or *no* anymore as it does not have sufficient information to make a decision. In these cases, the analysis is inconclusive, that is, the static analyser will have to answer *maybe*. Clearly, this last option is often undesirable. However, such cases are unavoidable since the static analyser is required to run in finite time.

Despite the possibility to answer *maybe*, often, static analysers are designed to classify programs for which the analysis is inconclusive either as *yes* or *no*. The former option results in an analysis that may return *false positives*, i.e., programs that do not exhibit the property looked for may be categorised as *yes*. In this case, the set of programs categorised as *yes* is an *over-approximation* of the set of programs that actually exhibit the selected property: it will contain all programs that do exhibit the property, but may also contain some other programs that do not. On the other hand, classifying programs resulting in *maybe* as *no* causes the set of programs categorised as *yes* to be an *under-approximation* of the set of programs that actually exhibit the selected property because the set may not contain all programs exhibiting the property. In this case, the analysis results may contain *false negatives* as programs that do exhibit the property looked for may be categorised as *no*.

A static analysis technique may be judged according to several characteristics. Four important characteristics that can be thought of are *soundness*, *completeness*, *precision* and *analysis time*:

- A static analysis that is **sound** classifies all programs exhibiting a property correctly, that is, the analysis will never answer *no* for a program that *does* exhibit the property. Soundness is often important in analyses that are looking for program defects but may also result in false positives.
- A static analysis that is **complete** classifies all programs *not* exhibiting a property correctly, that is, the analysis will never answer *yes* for a program that *does not* exhibit the property. A complete analysis may result in false negatives, however.
- The **precision** of an analysis depends on the number of programs it classifies correctly; a static analyser generating a lot of false positives or false negatives is less precise than an analyser that misclassifies few programs. Clearly, the results of an imprecise analysis convey less useful information to developers, reducing the usability of the analysis. However, the precision of an analysis often is hard to quantify.
- The **time** a static analyser needs to analyse a program not only depends on the program itself, but also on the analysis technique used by the analyser. Clearly, fast analyses are preferred. However, the use of some analysis techniques results in a positive correlation between analysis time and precision, meaning that to shorten the analysis time, precision may need to be sacrificed.

The above parameters depend on the way a static analysis technique approximates the classification of a program. In the remainder of this dissertation, we will focus mostly on analysis time and precision and require our techniques to be sound, implying the analyses presented in this dissertation will over-approximate the set of programs exhibiting a property, thus resulting in the possibility of false positives. A kind of static analyser that can be expected to use such an over-approximation is a type checker, as illustrated in Example 2.1. It is important to see that a static analyser can never be sound and complete at the same time since this would imply that it classifies all programs correctly, a problem which is undecidable.

Example 2.1 Type Checker

A typical example of a static analyser is a type checker, which verifies whether a program contains type errors (*yes*) or whether is does not (*no*).

Since a type checker is a static analyser, an approximation needs to be used to avoid undecidability. For this reason, the type checker cannot classify all programs correctly. A sound type checker will never declare an unsafe program safe, i.e., answer *no* when the program contains a type error. Therefore, it over-approximates the set of programs containing type errors, i.e., it will answer *yes* for all programs containing a type error, but also for some programs that do not. Conversely, a complete type checker never declares a safe program unsafe, i.e., it never answers *yes* when the program is type safe. Consequently, it under-approximates the set of programs containing a type error and hence may answer *no* when the program does contain a type error.

It is clear that normally, a sound type checker, i.e., one that finds all errors, is preferred to a complete type checker; otherwise, some errors may remain undetected. As a result, when getting feedback from such a type checker, some programs may be rejected needlessly. For example, some type checkers may reject perfectly valid code such as `(+ 1 (if true 2 false))` due to the inequality of the types of both branches of the `if` statement.

2.2. Concrete and Abstract Interpretation

The static analysis technique used in this dissertation is *Abstract Interpretation* (Cousot & Cousot, 1977). Abstract interpretation is related closely to conventional (concrete) interpretation of computer programs. The technique allows to tune the precision of the analysis, although a higher precision may lead to increased analysis times and vice versa. Because an abstract interpreter closely resembles a concrete interpreter, programs to be analysed do not need any modification and no user interaction is required; it is, for example, not needed to instrument the program with some kind of annotations. In other words, abstract interpretation is a fully automated technique. Finally, abstract interpretation has already been applied to several important programming constructs, such as higher-order functions and concurrency (Might & Van Horn, 2011; Stiévenart et al., 2015; Stiévenart, 2018).

In the remainder of this section, we will first discuss concrete interpretation of a program, whereafter abstract interpretation is introduced. Next, we present some mathematical notions relating to abstract interpretation and use them to formalise the concept of an *abstraction*.

2.2.1. Concrete Interpretation

Consider the execution of a program e by a regular interpreter. During the interpretation of e, the interpreter will *step through* the instructions of the program until it reaches a final result, which is then returned. The steps taken by the interpreter are prescribed by the program's semantics. However, the interpreter is not guaranteed to reach a final result; in this case, the program (and hence the interpreter) will never terminate. In the remainder of this dissertation, this type of interpretation will be referred to as *concrete interpretation*.

Concrete interpretation of a program e can formally be described as follows: first, the program e is *injected* into an initial state, s_0, by means of an *injection function*. For every non-final state, a successor state can be determined based on the small-step operational semantics of the program. Hence, the *transition function* applies this semantics to a state s_i to obtain its successor state s_{i+1}, denoted $s_i \hookrightarrow s_{i+1}$. By applying this function iteratively starting from the program's initial state s_0, a *trace* of the program's execution is obtained. As the program is not guaranteed to end, the trace may be infinite. An example of such a trace is depicted in Figure 2.1.

The exact trace generated by the concrete interpreter may depend on user input, for example. As a result, multiple executions of a single program may lead to different traces being generated. The

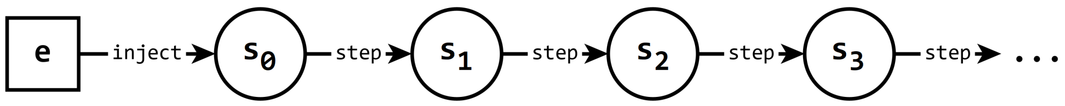


Figure 2.1.: Example of a trace resulting from concrete interpretation.

set of all possible traces of the concrete interpreter for a given program is called the program's *collecting semantics*. This set corresponds to all possible program executions and hence may be infinite, which is, for example, the case when a program depends on user input.

2.2.2. Abstract Interpretation

The result of a concrete interpretation is a potentially infinite trace of states generated by the transition function. The trace represents the successive states of the interpreter during the execution of the program. However, since the trace is not guaranteed to be finite, it is not suitable for analysis. Also, the states generated by the transition function may depend on (user) input, which does not suite static analysis well either.

To obtain a decidable analysis, the goal of abstract interpretation is to generate a *finite approximation* of a program's collecting semantics, that is, to compute an approximation of all traces that may be generated by the concrete interpreter for a specific program. To do so, an abstract interpreter makes use of an approximation of the program's semantics, which is obtained by *abstracting* the semantics. The abstractions used throughout this dissertation will render all infinite parts of the program's state space finite. As a result, the abstract interpreter's state space is finite and it can only generate a finite amount of states. Hence, a decidable analysis is obtained, although some precision is lost. The larger the size of the resulting state space, the closer the abstract interpretation will be to its concrete counterpart and the higher the precision of the analysis. However, as more states may need to be explored, the analysis time will also increase. For clarity throughout this dissertation, the following notational convention is used: abstracted elements, such as functions, sets and states, are denoted with a hat ($\widehat{\ldots}$). For example, if the injection function is denoted as *inject*, the abstract injection function will be denoted as $\widehat{inject}$.

An abstract interpreter and a concrete interpreter work quite alike. Formally, the abstract interpretation of a program e can be described as follows: first, the program e is injected into an *abstract* initial state, $\widehat{s}_0$, by means of an *abstract injection function*. Thereafter, the *abstract transition function* uses the *abstract* semantics of the program to step through *abstract* states, where every abstract state is an approximation of one or more concrete states. However, unlike concrete interpretation, every non-final state can now have *one or more* successor states due to the loss of precision.

Rather than ending with a linear, possible infinite, sequence of states, the result of an abstract interpreter is a graph that is always finite. This finiteness is a result of the approximation of the semantics that rendered the state space finite. The fact that a graph is obtained also stems from this approximation as this may introduce uncertainty in the decision making of the abstract interpreter and since it may also cause back edges to appear. Consider for example the abstract interpretation of an `if` statement. If, due to the loss of precision, the abstract interpreter cannot determine whether the predicate is true or false, it will have to explore *both* branches. An example of the resulting graph is depicted in Figure 2.2.

The resulting state graph is called the *abstract collecting semantics* of the program. When the analysis is sound, the abstract collecting semantics is an over-approximation of the program's

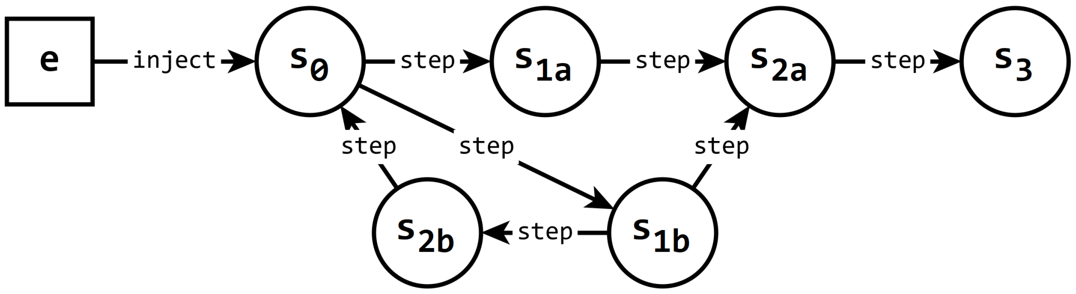


Figure 2.2.: Example of a graph resulting from abstract interpretation. Hats indicating abstraction have been omitted.

concrete collecting semantics, that is, it represents all paths the concrete interpreter may ever follow but it may also contain paths that are never followed by the concrete interpreter. Based on this graph, properties of the analysed program can be determined. For example, a *dead code analysis* may look for pieces of source code that were never executed by the abstract interpreter and hence do not appear in the result graph. Since the graph is an over-approximation of the states that may be reached during the execution of the program, code that does not appear in the graph can never be executed by a concrete interpreter and hence is deemed dead code.

2.2.3. Mathematical Preliminaries

In the previous section, the concept of approximation by means of abstraction was discussed. In this section, the mathematical concepts needed to formally define an abstraction are presented.

Definition 1. *A* ***partially ordered set*** *is a set S on which a binary relation $(\sqsubseteq) \subset S \times S$ is defined that is*

- *reflexive: $\forall x \in S : x \sqsubseteq x$,*
- *anti-symmetric: $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$, and*
- *transitive: $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.*

The relation $(\sqsubseteq)$ is called a partial order on S. We denote the partially ordered set as $(S, \sqsubseteq)$.

Definition 2. *An element $u \in S$ is an* ***upper bound*** *of $X \subseteq S$ if $\forall x \in X : x \sqsubseteq u$. An upper bound u of X is the* ***least upper bound*** *or* ***supremum*** *of X, denoted $\bigsqcup X$, if for every upper bound v of $X : u \sqsubseteq v$. We define a corresponding binary operator so that $\bigsqcup\{x, y\}$ can be denoted as $x \sqcup y$, where $\sqcup$ is called the* ***join*** *operator.*

Definition 3. *An element $l \in S$ is a* ***lower bound*** *of $X \subseteq S$ if $\forall x \in X : l \sqsubseteq x$. A lower bound l of X is the* ***greatest lower bound*** *or* ***infimum*** *of X, denoted $\bigsqcap X$, if for every lower bound k of $X : k \sqsubseteq l$. We define a corresponding binary operator so that $\bigsqcap\{x, y\}$ can be denoted as $x \sqcap y$, where $\sqcap$ is called the* ***meet*** *operator.*

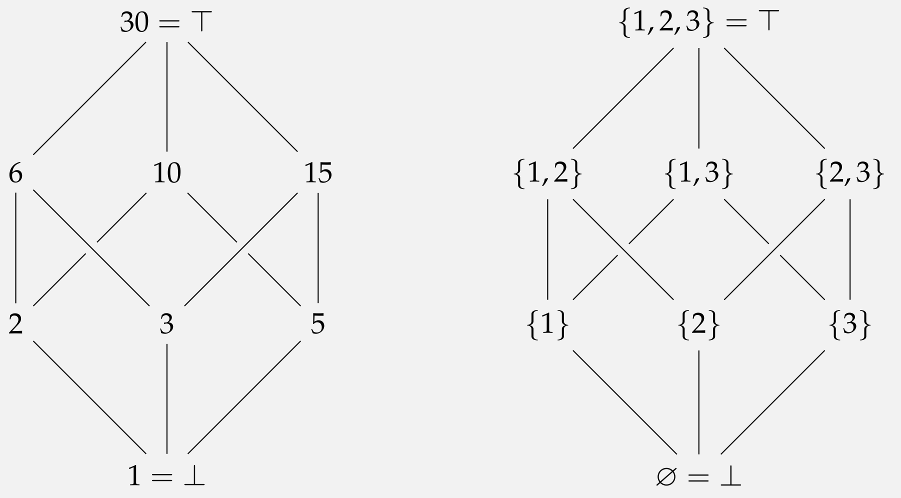
Definition 4. *A* ***lattice*** *is a partially ordered set L in which every subset of two elements has a supremum and an infimum. A* ***complete lattice*** *is a partially ordered set L in which every subset has a supremum and an infimum. Every complete lattice has a* ***bottom*** *element, $\bot$, so that $\bot = \bigsqcap L$ and a* ***top*** *element, $\top$, so that $\top = \bigsqcup L$.*

Definition 5. *A* ***Galois connection*** *between two partially ordered sets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ is a pair of functions $(\alpha : A \to B, \gamma : B \to A)$ so that $\forall a \in A, b \in B : \alpha(a) \sqsubseteq_B b \Leftrightarrow a \sqsubseteq_A \gamma(b)$ where α is called the* ***lower adjoint*** *and γ is called the* ***upper adjoint****.*

A partially ordered set $(S, \sqsubseteq)$, can be visualised by means of a *Hasse diagram* in which nodes represent elements of S and a vertex between $s_1, s_2 \in S$ means that $s_1 \sqsubseteq s_2$ if s_1 is depicted lower than s_2 and $s_2 \sqsubseteq s_1$ otherwise. To clarify these concepts, Example 2.2 shows two complete lattices by means of their Hasse diagrams and defines corresponding join and meet operators.

Example 2.2 Complete Lattice

Figure 2.3a shows the Hasse diagram of a partially ordered set of numbers, using the partial order *divides*, i.e., $a \sqsubseteq b$ if a divides b. The partially ordered set in this figure also is a complete lattice, where 1 corresponds to the $\perp$ element, since 1 can only be divided by itself but divides all other elements in the set, and 30 corresponds to the $\top$ element, since 30 is divisible by all elements in the set but it only divides itself. For this lattice, the join operator is defined as the *least common multiple* and the meet operator is defined as the *greatest common divisor*, that is, $x \sqcup y = lcm(x, y)$ and $x \sqcap y = gcd(x, y)$.



(a) Hasse diagram of the partially ordered set $(\{1, 2, 3, 5, 6, 10, 15, 30\}, \textit{divides})$.

(b) Hasse diagram of the partially ordered set $(\mathcal{P}(\{1, 2, 3\}), \subseteq)$.

Figure 2.3.: Hasse diagrams of different partially ordered sets.

Figure 2.3b depicts the Hasse diagram of the power set $\mathcal{P}(\{1, 2, 3\})$, whose elements form a partially ordered set of sets with set inclusion $\subseteq$ as the partial order. This partially ordered set also is a complete lattice, where the empty set $\varnothing$ corresponds to the $\perp$ element, since it is a subset of all other sets, and $\{1, 2, 3\}$ corresponds to the $\top$ element, since it is a superset of all other sets. The join operator for this lattice is set union $\cup$ and the meet operator is set intersection $\cap$. Henceforth, we will refer to this kind of lattice as a *set lattice*.

2.2.4. Formalising Abstractions

The relation between concrete and abstract interpretation can be seen as follows: concrete interpretation works on a domain of values whereas abstract interpretation works on a domain of abstract values. Whereas the domain of concrete values typically is infinite, the domain of abstract values typically is finite.

Definition 6. *An* ***abstraction*** *relating a possibly infinite set of values V and a finite set of abstract* *values $\widehat{V}$ is a Galois connection between the two complete lattices, $(\mathcal{P}(V), \subseteq)$ and $(\widehat{V}, \sqsubseteq)$, where the lower adjoint $\alpha : \mathcal{P}(V) \rightarrow \widehat{V}$ is called the* ***abstraction function*** *or the* ***abstraction map*** *and the upper adjoint $\gamma : \widehat{V} \rightarrow \mathcal{P}(V)$ is called the* ***concretisation function****.*

Intuitively, the above definition means that both the abstraction and concretisation function preserve the partial ordering of elements. In some cases where the concretisation function is of no use, the concretisation function can be omitted and the abstraction is solely defined by means of the abstraction function.

Since an abstract interpreter works on abstract values, the operations it applies to these values also need to be abstracted. Furthermore, other components of the state space need may to be abstracted as well. Examples of such abstracted components that we have previously mentioned are the abstract injection function, the abstract transition function and abstract states.

Example 2.3 shows a possible abstraction for the set of integers, $\mathbb{Z}$, and illustrates how information can be lost by performing operations on abstract numbers. In general, the lower an element sits in the partial order of the lattice, the more information it conveys to the abstract interpreter, that is, the bottom element conveys the most information and the top element conveys the least information. Normally, bottom represents a kind of inconsistency and any abstract operation performed on bottom will return bottom. For example, in the sign lattice presented in Example 2.3, bottom represents *not a number* whereas top represents *any* number. Hence, the choice of the abstraction map directly influences the precision of the analysis.

Example 2.3 Sign Abstraction

An example abstraction is the *sign abstraction* by which the infinite set of integers, $\mathbb{Z}$, is abstracted into a finite set of signs. More specifically, the abstraction is a Galois connection between the two complete lattices $(\mathcal{P}(\mathbb{Z}), \subseteq)$ and $(\{+, -, \widehat{0}, \top, \bot\}, \sqsubseteq_{sign})$, where $\sqsubseteq_{sign}$ is depicted in Figure 2.4. Henceforth, the latter will be referred to as the *sign lattice*.

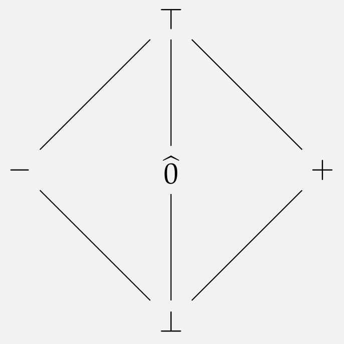


Figure 2.4.: Hasse diagram of the sign lattice.

Intuitively, the sign abstraction can be thought of as follows: an integer can be considered as a combination of a sign and an absolute value. The sign abstraction approximates this integer by abstracting away the absolute value, only keeping the sign of the number. Hence, the abstraction and concretisation functions can be defined as follows:

$$\alpha : \mathcal{P}(\mathbb{Z}) \to \{+, -, \widehat{0}, \top, \bot\} \qquad \gamma : \{+, -, \widehat{0}, \top, \bot\} \to \mathcal{P}(\mathbb{Z})$$

$$\alpha(Z) = \begin{cases} + & \text{if } Z \subseteq \mathbb{Z}_0^+ \\ - & \text{if } Z \subseteq \mathbb{Z}_0^- \\ \widehat{0} & \text{if } Z = \{0\} \\ \bot & \text{if } Z = \varnothing \\ \top & \text{otherwise} \end{cases} \qquad \gamma(S) = \begin{cases} \mathbb{Z}^+ & \text{if } S = + \\ \mathbb{Z}^- & \text{if } S = - \\ \{0\} & \text{if } S = \widehat{0} \\ \varnothing & \text{if } S = \bot \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

It is clear that regular functions cannot work on the abstract values $\{+, -, \widehat{0}, \top, \bot\}$. It is therefore required to abstract the operations on integers too, making them operate on abstract values. The abstract increment operator $\widehat{inc}$ can be defined as follows:

$$\widehat{inc}(S) = \begin{cases} + & \text{if } S = + \lor S = \widehat{0} \\ \bot & \text{if } S = \bot \\ \top & \text{otherwise} \end{cases}$$

This definition illustrates how precision can be lost by performing operations on abstract numbers. For example, $\widehat{inc}(-)$ returns $\top$, representing a number that can have any sign. The reason for this is that $-$ does not convey enough information to decide whether $\widehat{0}$ or $-$ should be returned.

Key to the development of an abstract interpreter is to construct the abstraction in such a way that the operations on the abstract domain return some information with regard to the operations on the concrete domain, that is, the abstraction needs to be sound: the abstract value obtained in the abstract domain should always be an over-approximation of the concrete value that would be obtained in the corresponding concrete domain, i.e.,

$$\alpha(f(v)) \sqsubseteq \widehat{f}(\widehat{v})$$

where $\widehat{f}$ is the abstract counterpart of f and $\widehat{v} = \alpha(v)$. If we now take the function f to be the transition function, we obtain the definition of a sound analysis (Van Horn & Might, 2010):

Definition 7. *Let ς denote a state reachable by the concrete interpreter, $\hookrightarrow$ the concrete transition function and $\widehat{\Sigma}$ the set of abstract states generated by the abstract interpreter. An analysis is **sound** if from $\varsigma \hookrightarrow \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \widehat{\varsigma}$, it follows that $\exists \widehat{\varsigma}' \in \widehat{\Sigma}$ so that $\widehat{\varsigma} \mathrel{\widehat{\hookrightarrow}} \widehat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \widehat{\varsigma}'$.*

Intuitively, this definition states that an analysis is sound if the finite set of abstract states generated by the abstract interpreter is an over-approximation of the (possibly infinite) set of concrete states that may be generated by the concrete interpreter. The bigger the size of the abstract domain, the more precision can be retained due to the increased size of the abstract interpreter's state space.

2.3. Static Analysis for Concurrent Languages

In Section 2.2.2, we have already discussed how an abstract interpreter generates a state graph that over-approximates the behaviour of a program. Every state in the resulting graph over-approximates one or more concrete states the concrete interpreter may reach. So far, we only discussed abstract interpretation for sequential programs. However, abstract interpretation, and static analysis in general, can also be applied to concurrent programs. By abstracting the semantics of a concurrent language, an abstract interpreter supporting that concurrent language can also be built. In this section, we first discuss how an abstract interpreter may analyse a multithreaded program. Then, we present modular static program analysis as a method to improve the time complexity of analyses for concurrent languages (Stiévenart et al., 2015; Stiévenart et al., 2019).

2.3.1. Non-Modular Static Analysis for Concurrent Languages

The execution of a multithreaded program typically is nondeterministic. This nondeterminism stems from the fact that threads work concurrently. However, variables and objects may be shared among multiple threads. Moreover, these shared variables and objects may also be mutated by the concurrent threads. The order in which these objects are manipulated by the threads may not only influence the values of these shared objects, but also the control flow within the threads themselves. Example 2.4 depicts a program that illustrates this case.

Example 2.4 Multithreaded Programming

In Listing 2.1, a multithreaded program is shown. Two variables, `x` and `y`, are defined and are shared among two threads, `thread1` and `thread2`.

```
(define x 0) ;; x and y are shared variables.
(define y 0)
(let ((thread1 (spawn (set! x 1)      ; (1)
                      (sleep 30)
                      (set! x 0)))    ; (2)
      (thread2 (spawn (if (= x 1)     ; (3)
                          (set! y 2)
                          (set! y -1)))))
  (join thread1) ;; Blocking call.
  (join thread2)
  (display y))
```

Listing 2.1: Example of a multithreaded program with shared variables.

The value of `y` after the execution of the program depends on the execution order of the threads: only when the value of the variable `x` in (3) is read after the execution of (1) but prior to the execution of (2), the value of `y` will be `2`; all other thread interleavings result in `-1` being final value of `y`. Hence, this example illustrates how the scheduling of the threads may influence the control flow of `thread2`. Situations in which the order in which the threads perform the operations is important for the final result of the program are called *race conditions* and are generally unintended. However, race conditions may be very hard to detect since bugs may only arise during some executions of the program. Nevertheless, a static analyser for concurrent languages can be used to detect the presence of race conditions or to guarantee their absence.

A static analyser for concurrent languages has to take the states of the different threads into account. Hence, every state the concrete interpreter may be in is over-approximated by an abstract state in the state graph produced by the static analyser, where an abstract state takes the evaluation state of all threads into account. Moreover, since the analyser needs to produce a sound result, the state graph it produces must be an over-approximation of all the possible states the concrete interpreter executing the multithreaded program may be in. This means that the analyser must account for all possible thread interleavings in the concurrent program. Otherwise, it may fail to produce a sound result since the produced state graph may not be a correct over-approximation. It is important to note that, for abstract interpretation, threads are abstracted as well. This is needed to keep the set of states the abstract interpreter can generate finite. In this light, the set of thread identifiers should be finite as well, since programs may spawn an arbitrary, and hence possibly infinite, number of threads.

The fact that a static analyser needs to account for all possible thread interleavings is problematic since the abstract interpreter may need to generate a state over-approximating every possible state the concrete interpreter may be in, that is, for every possible thread interleaving. This results in the *state explosion problem*: the number of possible thread interleavings, and hence the number of states that may need to be generated by the abstract interpreter, is subject to a combinatorial explosion. Suppose for example a program consisting out of two threads, where one thread has to execute m instructions and the other has to execute n instructions. In this case, the number of possible interleavings equals $\binom{m+n}{m} = \frac{(m+n)!}{m! \cdot n!}$. An example of a state explosion in the analysis of a program with two threads is depicted in Figure 2.5. A new thread is created in state $s_{1.0}$, which is later again joined into the main thread in state $s_{4.3}$. Between these two states, a state explosion occurs: the order in which both threads take a step is non-deterministic, and hence every possibility gives rise to a different path in the graph. Since each thread has to take three steps, in total, there are $\binom{3+3}{3} = 20$ possible interleavings the static analyser must consider. In practice, often less states will need to be generated since different thread interleavings not always lead to different interpreter states. This is for example the case when there is no interference between the different threads, or when this interference does not lead to multiple different abstract states. For example, in Figure 2.5, all paths again converge into a single abstract state, $s_{4.3}$.

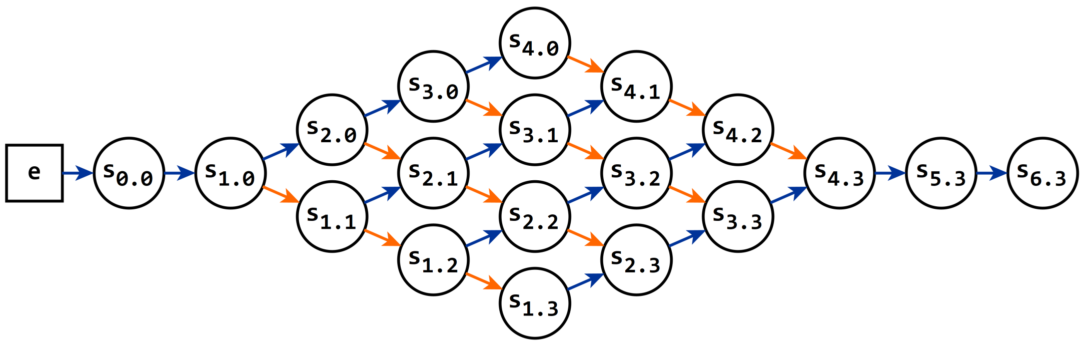


Figure 2.5.: Example analysis of a multithreaded program in which all interleavings must be accounted for. Abstract state $s_{m.n}$ indicates that the abstract blue thread has taken m steps and the abstract orange thread has taken n steps. Again, the hats indicating abstraction have been omitted.

The state explosion an abstract interpreter may suffer immediately impacts the analysis time needed by the analyser: due to the large number of states that may need to be generated, the abstract interpreter may need more time to complete the entire analysis. Also, the analysis may require a considerable amount of memory. Therefore, it is clear that analysing concurrent programs using this technique does not scale to large programs or to programs creating a large number of threads.

2.3.2. Thread-Modular Static Analysis for Concurrent Languages

In Section 2.2, we have presented abstract interpretation as a technique to perform static analyses. It is clear that the size of the program under analysis will have an impact on the time the analysis needs to complete, as well as on the static analyser's memory requirements. For large programs, the time and memory needed by an analyser may become obstructive. Often, the time needed by the analysis can be decreased by lowering its precision, but as a result, this may also lower the analysis' usability. This issue is especially pressing for the analysis of concurrent languages as a result of the state explosion problem discussed earlier. To remedy this problem, several solutions have been introduced to decrease the size of the abstract interpreter's state space:

- **Partial-order reduction techniques** are based on the observation that the order of some evaluation steps in a concurrent program is irrelevant, i.e., that multiple orderings yield the same result. In fact, this is often the case for concurrent programs, meaning that multiple interleavings yield the same result; these interleavings are said to be *equivalent*. Partial-order reduction techniques only explore one of the equivalent thread interleavings instead of all of them. Hence, given a property, partial-order methods only explore a reduced part of the state space that is sufficient for checking the given property (Godefroid, 1995).
- Static analyses using the technique of **macro stepping**, introduced by Agha et al. (1997), analyse each process until completion or to the point where a potentially interfering operation has been performed (Stiévenart, 2018). Since the analysis of a thread may take multiple steps at once, i.e., perform a macro step, only the interleavings of these macro steps must be explored by the analysis. As a result, less interleavings must be explored by the analysis.
- A **modular analysis** divides a program into components which are analysed in isolation. Because the components are small, they can be analysed using a high precision while maintaining low analysis times for the components. However, since the analyses of the different components may be interdependent, the analysis of one component may trigger the reanalysis of other components (Cousot & Cousot, 2002).

Partial-order reduction techniques and macro-stepping do not alter the worst case time complexity of the analysis. On the other hand, this complexity is reduced when using the technique of modular analysis, which is used throughout this dissertation.

Improving Scalability through Modularity

The general idea behind a modular analysis, as presented by Cousot & Cousot (2002), is based on their observation that the semantics of a program can be obtained compositionally from the semantics of its parts (components), such as functions, classes, methods, source files,. . . A modular analysis therefore analyses the different components of a program independently of one another, resulting in partial analysis results. Since the different parts are generally small, the analysis of the program parts can be done using a high precision and since the analyses are performed in separation, they can be performed simultaneously. The analysis result of the entire program can then be obtained by combining the results of its different parts.

The analysis just described is actually called a *compositional analysis*, since the analysis result for the entire program can just be obtained by composing the results of the analyses of the different parts of the program. In general however, it is difficult for a compositional analysis to produce results that are precise since the different components of a program may be interdependent, that is, the analysis of one component may depend on the analysis result of another. Consider for example the case where components are functions. The way a function is called may have an influence on the result it produces and hence on the result of its analysis, which therefore depends on how this function is called in the body of *other* functions. Hence, the result of the analysis of the function depends on results of the analyses of the calling functions. To account for these dependencies between the different parts of a program, a *modular analysis* tracks these dependencies. Whenever a new dependency is found or an existing dependency is updated, the modular analysis reanalyses the affected parts, possibly updating their results and triggering reanalysis of other program parts.

Thread-Modular Analyses for Concurrency

The concept of modular static analysis just presented provides a solution to the state explosion problem that was discussed in Section 2.3.1. Instead of letting the abstract interpreter explore all possible thread interleavings one by one, a *thread-modular static analysis* analyses the different threads in a program in isolation (Flanagan et al., 2002; Stiévenart, 2018). However, a thread may have *effects* on the environment it works in. For example, it may mutate shared variables, spawn new threads and read the return value of other threads. These effects need to be tracked as they create dependencies between the different threads. Consider again `thread2` in Example 2.4, which reads the value of variable `x`. Since `thread1` modifies `x`, the modular analysis will construct a dependency between the two threads. If this dependency would not be tracked, `thread2` would not be aware that the value of `x` is altered by another thread and the analysis would never consider the branch in which `y` is set to `2`. Hence, a modular analysis avoids having to explicitly take into account all possible thread interleavings by tracking the dependencies between threads and by reanalysing threads when necessary.

Instead of producing one graph for the entire program, the result of a thread-modular analysis is a collection of graphs, one for each thread. Figure 2.2 graphically depicts how the result of a modular analysis may look. In grey, the dependencies between the two threads are marked. In state s_{1a}, process A forks a new thread of which it later reads the return value in state s_{5a}. Both threads share a variable `x`, which is mutated by process A in state s_{2a} and read by process B in state s_{2b}. Hence, the change of the value of `x` may cause process B to be reanalysed. Conversely, the change of the return value of process B may cause process A to be reanalysed. The analysis continues until a fixed point is reached, which is guaranteed to happen due to the finite state space in which the abstract interpreter is operating.

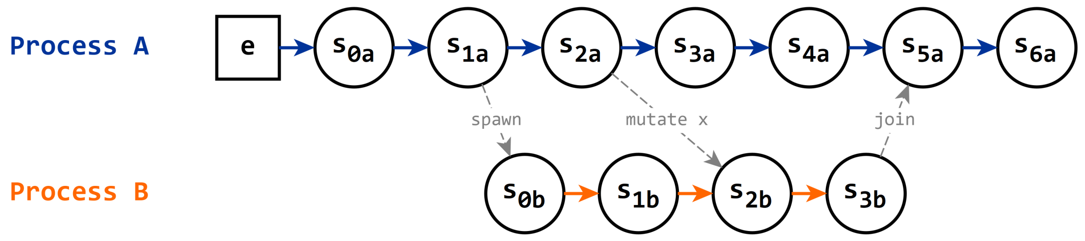


Figure 2.6.: Example of a modular analysis result. The grey arrows denote inter-thread dependencies.

2.4. Conclusion

In this chapter, we have presented static analysis, a technique to infer program properties at compile time. We have discussed important characteristics of a static analyser, such as soundness, precision and analysis time. We find that soundness is a crucial property and hence, we will require the static analyses presented in the remainder of this dissertation to be sound. The static analysis technique that is used throughout this dissertation is abstract interpretation, which works similarly to concrete interpretation but uses an approximation of a program's semantics. The abstract interpretation of a program results in an abstract state graph, a finite graph over-approximating the program's concrete collecting semantics; the approximation of program semantics relies on the mathematical concept of lattices, with set theory at its core.

When applying abstract interpretation to concurrent programs naively, the abstract interpreter falls subject to the state explosion problem as it must explicitly take all possible thread interleavings into account. This poses a burden on the scalability of the analysis. To avoid this issue, the concept of a thread-modular analysis, which is used throughout this dissertation, was introduced by Flanagan et al. (2002). A thread-modular analysis analyses the different threads of a program in isolation, thereby avoiding the need to explicitly account for all possible thread interleavings. Instead, the analysis must track the dependencies between the different threads to account for thread interference and reanalyse threads when necessary.

3

TOWARDS AN ABSTRACT INTERPRETER FOR λ_α, A CONCURRENT LANGUAGE WITH ATOMS

In this chapter, we first formalise λ_0, a simple sequential language, which is then extended two times successively. Each language is systematically introduced by the presentation of its syntax and semantics; our formalisation is based on the notational conventions of Stiévenart (2018). The semantics of the languages are presented using a (P)CESK machine, after which they are abstracted to obtain a formal description of an abstract interpreter. The first language, λ_0, is a simple sequential language based on the call-by-value lambda calculus. In a first extension, parallelism is enabled by the addition of futures, resulting in λ_ϕ. Finally, we present λ_α, a further extension of λ_ϕ obtained by the addition of atoms. Futures and atoms are already present several in modern-day languages. For this reason, the syntax and semantics presented in this chapter are based on the syntax and semantics of futures and atoms in Clojure.

3.1. λ_0, a Sequential Base Language

In this section, we present a simple sequential language, λ_0, that is based on the call-by-value lambda calculus. The language has been enriched with a `letrec` expression that can be used for variable binding. First, the syntax of λ_0 is presented, followed by its concrete semantics. Finally, we show how this semantics is abstracted, obtaining a formalisation of an abstract interpreter for λ_0.

3.1.1. Syntax

Figure 3.1 depicts the syntax of λ_0. An expression of the language is either an *atomic expression*, a function application or a `letrec` for variable binding. An atomic expression is either a variable reference or a lambda expression consisting out of a single parameter and a body. Finally, every program may contain a set of variable names, which is guaranteed to be finite due to the finite length of the program.

For simplicity, the presented formalisation assumes all functions are unary. This restriction does not limit the expressive power of the language in any way and the formalisation can be generalised easily to functions with an arbitrary number of arguments. For example, a nullary function can be written in λ_0 as a unary function that ignores its argument and a function of multiple arguments can be curried, that is, written as a sequence of nested unary functions. The same is true for `letrec`, where only one binding is assumed.

$\boxed{\lambda_0}$

$e \in \mathit{Exp}$ ::=			Expressions
		ae	*Atomic expression*
	|	`(`$e\ e$`)`	*Function application*
	|	`(letrec ((`$x\ e$`))` e`)`	*Bindings*
$ae \in \mathit{AExp}$ ::=			Atomic Expressions
		x	*Variable*
	|	*lam*	*Lambda*
$lam \in \mathit{Lam}$ ::=			Lambdas
		`(lambda (`x`)` e`)`	*Lambda*
$x \in \mathit{Var}$		A finite set of identifiers.	Variables

Figure 3.1.: Syntax of λ_0, a simple sequential language.

3.1.2. Concrete Semantics

The semantics of λ_0 is formalised by means of a CESK machine, an abstract machine similar to automata used to formalise concrete interpretation. A CESK machine can be described through the following four components:

- The **state space** of the machine describes the (possibly infinite) set of states the machine can generate;
- The **injection function** of the machine defines how an expression is converted to an initial state;
- The **transition function** defines how to step through the evaluation of an expression, that is, it defines the successor state(s) of a given state;
- The **evaluation function** defines the set of states that is reachable from an initial expression.

In the remainder of this section, a CESK machine for λ_0 is presented. This abstract machine formally describes a concrete interpreter for λ_0.

State Space

Figure 3.2 depicts the state space of the CESK machine for λ_0. A state ς contains all information needed to guide the evaluation of the program and consists out of two components:

- The **control** component c of a state contains the expression currently being evaluated by the machine coupled together with the environment in which this expression is to be evaluated, or it contains the value that was reached after an expression was fully evaluated. An **environment** ρ maps variable names to value addresses a.
- The **continuation address** k is the address of the current continuation in the continuation store Ξ, which maps continuation addresses to continuation frames. These continuation frames help the machine to keep track of its progress in the evaluation of a program.

Apart from a state, the transition function uses two auxiliary structures during evaluation:

- The **value store** σ maps value addresses a to values v. Henceforth, we may refer to the value store just as *store* and to value addresses just as *addresses*.
- The **continuation store** Ξ models the *stack* of the machine; it maps continuation addresses k to continuation frames ϕ, where each frame contains the continuation address of the next frame on the stack.

In the remaining part of this dissertation, we use the term *the stores* to refer to the ensemble of a value store and a continuation store.

λ_0

$\varsigma \in \Sigma$	$= \mathit{Control} \times \mathit{KAddr}$	States
$c \in \mathit{Control}$	$::=$	Control
	$\mathbf{ev}(e, \rho)$	*Expression evaluation*
	$\mid\ \mathbf{val}(v)$	*Reached value*
$v \in \mathit{Val}$	$::=$	Values
	$\mathbf{clo}(\mathit{lam}, \rho)$	*Closure*
$\rho \in \mathit{Env}$	$= \mathit{Var} \rightharpoonup \mathit{Addr}$	Environments
$\sigma \in \mathit{Store}$	$= \mathit{Addr} \rightharpoonup \mathit{Val}$	Value Stores
$\Xi \in \mathit{KStore}$	$= \mathit{KAddr} \rightharpoonup \mathit{Frame}$	Continuation Stores
$\phi \in \mathit{Frame}$	$::=$	Continuation frames
	$\mathbf{halt}$	*Halt continuation*
	$\mid\ \mathbf{fun}(e, \rho, k)$	*Operator continuation*
	$\mid\ \mathbf{arg}(v, k)$	*Operand continuation*
	$\mid\ \mathbf{bnd}(x, e, \rho, k)$	`letrec` *continuation*
$a \in \mathit{Addr}$	A set of addresses for values.	Value Addresses
$k \in \mathit{KAddr}$	A set of addresses for continuations.	Continuation Addresses

Figure 3.2.: State space for λ_0.

Note that our formalisation slightly diverges from the conventional CESK machine as introduced by Felleisen & Friedman (1987), but this will prove beneficial later on. An important difference is that the current continuation is not part of a state directly, but that a separate *continuation store* is used to map continuation addresses to continuation frames, as proposed by Johnson & Van Horn (2014). Although it is possible to use one unified store, mapping addresses to both values and continuation frames, the use of a separate continuation store will simplify the abstraction of the semantics. It is also possible to include a *timestamp* in the machine states, which can improve the precision of an analysis by introducing *context sensitivity* (Van Horn & Might, 2012). However, this does not fall into the scope of this dissertation, for which we will omit timestamps in the presented formalisations.

As a final remark, it is important to mention that a state does *not* contain a (continuation) store itself. The stores are passed to the transition function separately. This will prove useful later for the formalisation of concurrent languages.

Injection Function

To evaluate an expression using a CESK machine, the expression first needs to be *injected* into an initial machine state, to which the transition function can be applied. The injection function $inject : Exp \rightarrow \Sigma$ for λ_0 is defined as follows:

$$inject(e) = \langle \mathbf{ev}(e, []), k_0 \rangle$$

where k_0 is a special continuation address reserved for the **halt** continuation (see Figure 3.2) and [] denotes an empty environment. When the machine is given this state, it will evaluate the program e in an initial empty environment [], that is, in an environment in which no variables are bound. The **halt** continuation indicates *end of evaluation* to the CESK machine: when the machine reaches a state with a value in its control component and the **halt** continuation on the top of its stack, the evaluation has been completed and the value in the control component of the machine is the return value of the evaluation. In this case, no further transition rules are applicable and the machine halts.

Transition Function

In the syntax of λ_0, a distinction is made between *atomic* expressions and regular, *complex*, expressions. An expression is said to be atomic if it can be evaluated atomically, that is, if it can be evaluated in a finite amount of time, without needing to modify the value store or continuation store of the machine. An expression is said to be complex if it is not atomic. Before defining the transition function, first, a specific evaluation relation is presented, defining evaluation rules for atomic expressions.

Figure 3.3 defines the atomic evaluation function for λ_0. The atomic evaluation function is denoted as $\Downarrow$, where $\rho, \sigma \vdash ae \Downarrow v$ means that the atomic expression ae evaluates to the value v in environment ρ and using store σ. Variables are evaluated by looking up their value in the store using the address stored in the environment. The atomic evaluation of a lambda results in a closure, which couples the lambda together with its lexical environment.

$\boxed{\lambda_0}$

$$\frac{\sigma(\rho(x)) = v}{\rho, \sigma \vdash x \Downarrow v}\ \textsc{var} \qquad \frac{}{\rho, \sigma \vdash lam \Downarrow \mathbf{clo}(lam, \rho)}\ \textsc{lambda}$$

Figure 3.3.: Atomic evaluation rules for λ_0.

The transition function for λ_0 makes use of two auxiliary functions:

$$alloc : Var \times Store \rightarrow Addr$$
$$kalloc : Exp \times Env \times Store \times KStore \rightarrow KAddr$$

These two functions are used to allocate addresses in the value store and continuation store respectively. In the concrete case, the number of addresses may be infinite and as a result, (continuation) addresses can be represented by means of integers. In this case, a new address is generated by looking at the number of existing addresses, that is, the size of the domain of the respective store; this allocation scheme results in a fresh address for every allocation. Figure 3.4 shows the structure of (continuation) addresses in λ_0, as well as definitions for the concrete allocation functions.

$\boxed{\lambda_0}$

$Addr = \mathbb{N}$	Value Addresses
$KAddr = \mathbb{N}$	Continuation Addresses
$alloc(x, \sigma) = \mid dom(\sigma) \mid$	Address Allocation
$kalloc(e, \rho, \sigma, \Xi) = \mid dom(\Xi) \mid$	Continuation Address Allocation

Figure 3.4.: Addresses for λ_0.

Given the definition of the atomic evaluation function and the address allocation functions, the transition function $\hookrightarrow$: $\Sigma \times Store \times KStore \rightarrow \Sigma \times Store \times KStore$ for λ_0, depicted in Figure 3.5, can be described as follows:

- Rule ATOMIC-EVALUATION connects the atomic evaluation function to the transition function; it states that if the expression to evaluate is an atomic expression, the atomic evaluation rules apply.
- Rules APPL-OPERATOR, APPL-OPERAND and APPL-BODY stipulate how a function application is evaluated. To apply an operator e_f to an operand e_a in the call-by-value lambda calculus, three steps are required.
  1. First, rule APPL-OPERATOR applies, which causes the operator e_f to be evaluated. It pushes a new frame on the stack, containing the argument expression e_a to be evaluated later on. To push a frame on the stack, a new continuation address k' is generated using *kalloc*, which is then used to extend the continuation store Ξ.
  2. After the operator has been evaluated, rule APPL-OPERAND applies. This rule pops the topmost frame of the stack by looking up the current continuation address k in the continuation store. This frame contains the operand expression e_a which is to be evaluated next. A new frame ϕ containing the value of the operator v_f, resulting from prior evaluation, is pushed onto the stack.
  3. Finally, when the operand has been evaluated to a value v_a, rule APPL-BODY applies. It pops the topmost frame of the stack to get the value of the operator v_f and gets the value of the operand v_a as the result from prior evaluation. It then extends the environment ρ with a new binding for the formal argument x of the function, to a newly allocated value address a, which is generated by means of *alloc*. This address is then used to extend the value store with a new binding from the address to the value of the operand.
- Rules LETREC-BINDING and LETREC-BODY define how a `letrec` is evaluated. Evaluation of a `letrec` resembles function evaluation, where the evaluation of the binding can be related to the evaluation of the operand.

The premise $v_f = \mathbf{clo}(lam, \rho)$ of rule APPL-BODY avoids the rule being applied when the function operand does not evaluate to a function. In case no transition rule applies and the machine has not reached a value while having the **halt** continuation on top of its stack, the evaluation of the expression is said to be *stuck*. This indicates the expression under evaluation does not comply with the semantics of λ_0, that is, it contains an error.

$\boxed{\lambda_0}$

$$\frac{\rho,\sigma \vdash ae \Downarrow v}{\langle \mathbf{ev}(ae,\rho),k\rangle,\sigma,\Xi \hookrightarrow \langle \mathbf{val}(v),k\rangle,\sigma,\Xi}\ \textsc{atomic-evaluation}$$

$$\frac{\phi = \mathbf{fun}(e_a,\rho,k) \qquad k' = kalloc(e_f,\rho,\sigma,\Xi)}{\langle \mathbf{ev}(\texttt{(}e_f\ e_a\texttt{)},\rho),k\rangle,\sigma,\Xi \hookrightarrow \langle \mathbf{ev}(e_f,\rho),k'\rangle,\sigma,\Xi[k' \mapsto \phi]}\ \textsc{appl-operator}$$

$$\frac{\Xi(k) = \mathbf{fun}(e_a,\rho,k') \qquad \phi = \mathbf{arg}(v_f,k') \qquad k'' = kalloc(e_a,\rho,\sigma,\Xi)}{\langle \mathbf{val}(v_f),k\rangle,\sigma,\Xi \hookrightarrow \langle \mathbf{ev}(e_a,\rho),k''\rangle,\sigma,\Xi[k'' \mapsto \phi]}\ \textsc{appl-operand}$$

$$\frac{\Xi(k) = \mathbf{arg}(v_f,k') \qquad v_f = \mathbf{clo}(lam,\rho) \qquad lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ \ e\texttt{)} \qquad a = alloc(v_a,\sigma)}{\langle \mathbf{val}(v_a),k\rangle,\sigma,\Xi \hookrightarrow \langle \mathbf{ev}(e,\rho[x \mapsto a]),k'\rangle,\sigma[a \mapsto v_a],\Xi}\ \textsc{appl-body}$$

$$\frac{\phi = \mathbf{bnd}(x,e_b,\rho,k') \qquad k' = kalloc(e_x,\rho,\sigma,\Xi)}{\langle \mathbf{ev}(\texttt{(letrec ((}x\ e_x\texttt{))}\ e_b\texttt{)},\rho),k\rangle \hookrightarrow \langle \mathbf{ev}(e_x,\rho),k'\rangle,\sigma,\Xi[k' \mapsto \phi]}\ \textsc{letrec-binding}$$

$$\frac{\Xi(k) = \mathbf{bnd}(x,e_b,\rho,k') \qquad a = alloc(v_x,\sigma)}{\langle \mathbf{val}(v_x),k\rangle,\sigma,\Xi \hookrightarrow \langle \mathbf{ev}(e_b,\rho[x \mapsto a]),k'\rangle,\sigma[a \mapsto v_x],\Xi}\ \textsc{letrec-body}$$

Figure 3.5.: Transition rules for λ_0.

The use of the concrete transition rules for λ_0 for the evaluation of a program is illustrated in Example 3.1.

Example 3.1 Concrete evaluation in λ_0

After an expression is injected into an initial state, it is evaluated by successive applications of λ_0's concrete transition function. Figure 3.6 illustrates the evaluation of the expression `((lambda (x) (x x))(lambda (y) (y y)))`, starting from an initial state that is obtained by applying the injection function to the expression. However, the evaluation of this particular expression will never terminate, as is always possible when evaluatin ga program concretely (see Section 2.2.1).

$$\begin{aligned}
& \langle \mathbf{ev}(\texttt{((lambda (x) (x x))(lambda (y) (y y)))},[],k_0)\rangle, \\
& \quad [],[k_0 \mapsto \mathbf{halt}] \\
\xhookrightarrow{\textsc{appl-operator}}\ & \langle \mathbf{ev}(\texttt{(lambda (x) (x x))},[]),k_1\rangle, \\
& \quad [],[k_0 \mapsto \mathbf{halt}, k_1 \mapsto \mathbf{fun}(\texttt{(lambda (y) (y y))},[],k_0)] \\
\xhookrightarrow{\textsc{atomic-evaluation}}\ & \langle \mathbf{val}(\mathbf{clo}(\texttt{(lambda (x) (x x))},[])),k_1\rangle, \\
& \quad [],[k_0 \mapsto \mathbf{halt}, k_1 \mapsto \mathbf{fun}(\texttt{(lambda (y) (y y))},[],k_0)] \\
\xhookrightarrow{\textsc{appl-operand}}\ & \langle \mathbf{ev}(\texttt{(lambda (y) (y y))},[]),k_2\rangle, \\
& \quad [],[k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots, k_2 \mapsto \mathbf{arg}(\mathbf{clo}(\texttt{(lambda (x) (x x))}[]),k_0)] \\
\xhookrightarrow{\textsc{atomic-evaluation}}\ & \langle \mathbf{val}(\mathbf{clo}(\texttt{(lambda (y) (y y))},[])),k_2\rangle, \\
& \quad [],[k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots, k_2 \mapsto \mathbf{arg}(\mathbf{clo}(\texttt{(lambda (x) (x x))}[]),k_0)]
\end{aligned}$$

$$
\begin{aligned}
&\xhookrightarrow{\textsc{appl-body}} \langle \mathbf{ev}(\texttt{(x x)}, [\texttt{x} \mapsto a_1]), k_0\rangle,\\
&\qquad [a_1 \mapsto \mathbf{clo}(\texttt{(lambda (y) (y y))}, [])], [k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots, k_2 \mapsto \ldots]\\
&\xhookrightarrow{\textsc{appl-operator}} \langle \mathbf{ev}(\texttt{x}, [\texttt{x} \mapsto a_1]), k_3\rangle,\\
&\qquad [a_1 \mapsto \mathbf{clo}(\texttt{(lambda (y) (y y))}, [])], [k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots, k_2 \mapsto \ldots,\\
&\qquad k_3 \mapsto \mathbf{fun}(\texttt{x}, [\texttt{x} \mapsto a_1], k_0)]\\
&\xhookrightarrow{\textsc{atomic-evaluation}} \langle \mathbf{val}(\mathbf{clo}(\texttt{(lambda (y) (y y))}, [])), k_3\rangle,\\
&\qquad [a_1 \mapsto \mathbf{clo}(\texttt{(lambda (y) (y y))}, [])], [k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots, k_2 \mapsto \ldots,\\
&\qquad k_3 \mapsto \mathbf{fun}(\texttt{x}, [\texttt{x} \mapsto a_1], k_0)]\\
&\xhookrightarrow{\textsc{appl-operand}} \langle \mathbf{ev}(\texttt{x}, [\texttt{x} \mapsto a_1]), k_4\rangle,\\
&\qquad [a_1 \mapsto \mathbf{clo}(\texttt{(lambda (y) (y y))}, [])], [k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots, k_2 \mapsto \ldots,\\
&\qquad k_3 \mapsto \ldots, k_4 \mapsto \mathbf{arg}(\mathbf{clo}(\texttt{(lambda (y) (y y))}, []), k_0)]\\
&\xhookrightarrow{\textsc{atomic-evaluation}} \langle \mathbf{val}(\mathbf{clo}(\texttt{(lambda (y) (y y))}, [])), k_4\rangle,\\
&\qquad [a_1 \mapsto \mathbf{clo}(\texttt{(lambda (y) (y y))}, [])], [k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots, k_2 \mapsto \ldots,\\
&\qquad k_3 \mapsto \ldots, k_4 \mapsto \mathbf{arg}(\mathbf{clo}(\texttt{(lambda (y) (y y))}, []), k_0)]\\
&\xhookrightarrow{\textsc{appl-body}} \langle \mathbf{ev}(\texttt{(y y)}, [\texttt{y} \mapsto a_2]), k_0\rangle,\\
&\qquad [a_1 \mapsto \ldots, a_2 \mapsto \mathbf{clo}(\texttt{(lambda (y) (y y))}, [])], [k_0 \mapsto \mathbf{halt}, k_1 \mapsto \ldots,\\
&\qquad k_2 \mapsto \ldots, k_3 \mapsto \ldots, k_4 \mapsto \ldots]\\
&\xhookrightarrow{\textit{ad infinitum}} \ldots
\end{aligned}
$$

Figure 3.6.: Evaluation of a non-terminating program.

Evaluation Function

The evaluation function $eval : Exp \to \mathcal{P}(\Sigma \times Store \times KStore)$ defines the set of states the CESK machine can reach during the evaluation of an expression. It is defined as follows:

$$eval(e) = \{\langle \varsigma, \sigma, \Xi\rangle \mid inject(e), [], [k_0 \mapsto \mathbf{halt}] \hookrightarrow^* \varsigma, \sigma, \Xi\}$$

with [] denoting the empty store.

Formally, this definition defines the set $eval(e)$ as the set of states that can be reached by zero or more steps of the transition function when starting from an initial empty value store and a continuation store in which only the **halt** continuation is mapped to. This function is useful since static analysis is concerned with the *evaluative behaviour* of the program e rather than with its return value.

The set $eval(e)$ is called the *(concrete) collecting semantics* of e and may be infinite, which makes it unsuitable for static analysis since deciding set membership is undecidable. Therefore, the semantics of λ_0 is abstracted, transforming the formalisation of a concrete interpreter into the formalisation of an abstract interpreter that can be used to obtain a sound and decidable approximation of $eval(e)$.

3.1.3. Intermezzo: Administrative Normal Form (ANF)

In the language described so far, a distinction was made between *atomic* expressions and *complex* expressions; the first type of expressions was defined as expressions that can be evaluated to a value in finite time, without needing to modify the value or continuation store. On the other hand, evaluating the second type of expressions may require an infinite amount of time as multiple subexpressions must be evaluated. Examples of complex expressions are function applications, in which an operator, an operand and a body must be evaluated.

In Administrative Normal Form (ANF), all subexpressions of a function application, i.e., both the operator and the operand, must be atomic expressions. Only subexpressions that are in tail position are allowed to be non-atomic (Flanagan et al., 1993). ANF was introduced as an intermediate representation for compilers, being an alternative to *Continuation Passing Style* (CPS). ANF simplifies the definition of the semantics of the language, as it avoids the need of writing explicitly evaluation rules for the operator and operand, such as APPL-OPERATOR and APPL-OPERAND in Figure 3.5. Since every program can be converted to ANF without losing expressiveness, in future extensions to λ_0, the program is assumed to be written in or converted to ANF.

The conversion of a program to ANF requires the presence of a `let`-like construct for binding variables to the evaluation result of non-atomic expressions (in λ_0, this is the `letrec` construct). To convert an expression to ANF, all non-atomic subexpressions that are not in tail-position must become `let`-bound. Example 3.2 shows how an expression can be converted to ANF.

Example 3.2 Conversion to Administrative Normal Form

Consider the program in Listing 3.1. The function `sum` calculates the sum of all integers up to `n`. It keeps an accumulator `res`, which is returned when `n` reaches zero. Since λ_0 only supports functions with one argument, the function `sum` is curried, i.e., it is written as a sequence of nested unary functions. (In this example, it is assumed that λ_0 also has boolean values and an `if`-expression that behave as expected.)

The code in Listing 3.1 is not in ANF since, at several locations in the code, non-atomic expressions occur as a subexpression. For example, in the initial call to `sum`, `(sum 10)` is a non-atomic expression appearing at the operator position of the call `((sum 10) 0)`. Likewise, the operands in the recursive call of `sum` are non-atomic expressions.

```
(letrec ((sum (lambda (n)
                (lambda (res)
                  (if (= n 0)
                      res
                      ((sum (- n 1))
                       (+ res n)))))))
  ((sum 10) 0))
```

Listing 3.1: Example of program that is not in ANF.

To convert the code in Listing 3.1 to ANF, all non-atomic expressions not appearing in tail position must become bound by a `letrec`. In the code, these expressions are marked in red. Listing 3.2 shows the ANF conversion of the code. Note that although the calls to fn are not atomic expressions, they are in tail position, which is allowed in ANF (lines 9 and 11). Also, to convert (sum (- n 1)) to an atomic expression, two subsequent bindings are needed since (- n 1) is a complex expression as well, and hence is not allowed as a direct argument to sum.

```
1  (letrec ((sum (lambda (n)
2                  (lambda (res)
3                    (letrec ((bool (= n 0)))
4                      (if bool
5                          res
6                          (letrec ((m (- n 1)))
7                            (letrec ((fn (sum m)))
8                              (letrec ((res2 (+ res n)))
9                                (fn res2)))))))))
10   (letrec ((fn (sum 10)))
11     (fn 0)))
```

Listing 3.2: ANF conversion of the program in Listing 3.1.

3.1.4. Abstract Semantics

To obtain the abstract semantics of λ_0, we follow the approach presented by Van Horn & Might (2010, 2012), called *Abstracting Abstract Machines* (AAM). This is a systematic approach to abstracting CESK machines, thereby obtaining a formalisation of an abstract interpreter. The approach consists out of several refactorings followed by an abstraction step. Using this approach, a deterministic CESK machine with a possibly infinite state space is transformed into a nondeterministic abstract CESK machine, denoted as $\widehat{CESK}$, with a finite state space. The name of the approach, AAM, originates from the fact that the CESK machine, which is an abstract machine, is abstracted, resulting in an *abstract abstract machine*.

The concrete semantics of λ_0 presented so far already includes the refactorings described by Van Horn & Might (2012). Therefore, the CESK machine of Section 3.1.2 can readily be abstracted. Key to seeing how this machine can be abstracted is noticing that there are no (mutually) recursive structures in the state space of λ_0: although the state space contains two structures that normally are inherently recursive, environments and continuation frames, we have purposely broken their recursive structure. The recursive structure of environments has been broken by the introduction of the *value store* as this avoids environments and closures being mutually recursive. The recursive structure of continuation frames has been broken by introducing the *continuation store* so continuation frames contain a pointer to the next frame (k) rather than the next frame itself. Hence, the recursive structure of continuation frames is broken as well. Without the presence of recursive structures in the state space, it can now be made finite by bounding the number of value addresses and continuation addresses.

Since the number of abstract (continuation and value) addresses is finite, an address may need to be used multiple times, that is, the address allocation functions may allocate the same address multiple times. As a consequence, when storing a value at a certain address in the store, any other value that is stored at that address cannot simply be overwritten since this would not be sound, that is, the new value stored at that address would not be an over-approximation of the respective concrete values any more. For example, suppose that at a given address in the store resides a function that always returns an integer and that this function would be overwritten by a function returning a boolean. In this case, the new value stored at the specific address does not account for the behaviour of the function that was removed. When the address is then looked up in the store, the static analyser would only find a function returning a boolean and would then only study this function's behaviour. A similar problem applies to frames in the continuation store.

The solution to this soundness problem is the use of abstract values, that is, the stores do no longer map to a value or a continuation frame, but to *abstract values* and *abstract continuation frames* respectively, which are part of a lattice. Upon the allocation of a value or frame in the (continuation) store, the value or frame is joined with the lattice element that was already stored earlier. In the formalisation of the $\widehat{CESK}$ machine, a *set lattice* is used, which constructs abstract values by means of storing them in a set, using the partial order $\subseteq$ and set union $\cup$ as join operator. When reading a value or frame in one of the stores, all values or frames stored in the corresponding set must be considered by the abstract interpreter, resulting in nondeterminism and hence, loss of precision. As a result, the choice of lattice and the allocation of addresses have a significant influence on the precision of the analysis.

We now present the abstracted semantics of λ_0 by specifying the different components of the $\widehat{CESK}$ machine. Changes with regard to the concrete semantics of λ_0 are marked in grey.

Abstract State Space

Figure 3.7 depicts the abstract state space for λ_0. The state space is rendered finite by of bounding the set of (continuation) addresses. Hence, both $\widehat{Addr}$ and $\widehat{KAddr}$ are finite sets, which impacts the stores that now map addresses to sets of values and continuation addresses to sets of frames.

λ_0

$$
\begin{array}{llll}
\hat{\varsigma} \in \hat{\Sigma} = \widehat{Control} \times \widehat{KAddr} & & \hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}(\widehat{Val}) & \\
\hat{c} \in \widehat{Control} ::= & & \hat{\Xi} \in \widehat{KStore} = \widehat{KAddr} \to \mathcal{P}(\widehat{Frame}) & \\
& \mathbf{ev}(e, \hat{\rho}) & \hat{\phi} \in \widehat{Frame} ::= & \\
& \mid \ \mathbf{val}(\hat{v}) & & \mathbf{halt} \\
\hat{v} \in \widehat{Val} ::= & & & \mid \ \mathbf{fun}(e, \hat{\rho}, \hat{k}) \\
& \mathbf{clo}(lam, \hat{\rho}) & & \mid \ \mathbf{arg}(\hat{v}, \hat{k}) \\
\hat{\rho} \in \widehat{Env} = Var \to \widehat{Addr} & & & \mid \ \mathbf{bnd}(e, \hat{\rho}, \hat{k})
\end{array}
$$

$\hat{a} \in \widehat{Addr}$	A finite set of addresses for values.
$\hat{k} \in \widehat{KAddr}$	A finite set of addresses for continuations.

Figure 3.7.: Abstract state space for λ_0.

To see why the abstract state space is finite, consider the possible number of states $\hat{\Sigma} \times \widehat{Store} \times \widehat{KStore}$ it may contain when the sets of addresses are finite. Each state ς consists out of a control component $\hat{c}$ and a continuation address $\widehat{k}$. Since the number of continuation addresses is finite, the number of possible states is finite if the number of possible control components is finite. A control component is an evaluation component consisting out of an expression e and an environment $\hat{\rho}$ or a value v. Since every program has finite length, the number of expressions in the program is finite. The number of possible environments $\hat{\rho}$ is finite for the same reason, since a finite program can only contain a finite number of variables and can only map variables to a finite set of addresses. Hence, the number of closures that can be constructed is finite, meaning that the number of possible states is finite. Using similar reasoning, it can be shown that the number of possible value stores $\hat{\sigma}$, continuation stores $\widehat{\Xi}$ and frames $\hat{\phi}$ are finite. Hence, bounding the number of (continuation) addresses results in a finite state space, meaning that the number of states the abstract interpreter can generate is finite as well. This is a crucial requirement for an abstract interpreter, as it guarantees decidability.

Abstract Injection Function

The abstract injection function $\widehat{inject} : Exp \to \widehat{\Sigma}$ injects an expression into an *abstract* state. The function is defined as follows:

$$\widehat{inject}(e) = \langle \mathbf{ev}(e, []), \widehat{k}_0 \rangle$$

where $\widehat{k}_0$ is a special abstract continuation address reserved for the **halt** continuation, analogous to k_0 in the concrete semantics.

Abstract Transition Relation

Like the concrete transition function for λ_0, the abstract transition relation for λ_0 uses an abstract atomic evaluation relation to evaluate atomic expressions. This relation, depicted in Figure 3.8, is denoted as $\widehat{\Downarrow}$. The rule for evaluating lambdas does not need modification. On the other hand, in rule VAR, the new semantics of the store need to be taken into account: the abstract store relates addresses to sets of values instead of relating addresses to plain values.

$\boxed{\lambda_0}$

$$\frac{\hat{v} \in \hat{\sigma}(\hat{\rho}(x))}{\hat{\rho}, \hat{\sigma} \vdash x \mathrel{\widehat{\Downarrow}} \hat{v}}\ \textsc{var} \qquad \frac{}{\hat{\rho}, \hat{\sigma} \vdash lam \mathrel{\widehat{\Downarrow}} \mathbf{clo}(lam, \hat{\rho})}\ \textsc{lambda}$$

Figure 3.8.: Abstract atomic evaluation rules for λ_0.

Analogously to the concrete transition function, the abstract transition relation for λ_0 also makes use of two auxiliary functions for address allocation:

$$\widehat{alloc} : Var \times \widehat{Store} \to \widehat{Addr}$$
$$\widehat{kalloc} : Exp \times \widehat{Env} \times \widehat{Store} \times \widehat{KStore} \to \widehat{KAddr}$$

which now operate on the abstract state space. In the concrete semantics of λ_0, the exact definitions of *alloc* and *kalloc* are of low interest as long as they always generate a fresh address. In an abstract interpreter, however, they can be used to tune the precision of the analysis as their definitions also influence the structure of (continuation) addresses, as well as the number of addresses available. Hence, the definitions of these functions are crucial in the abstract CESK machine as they define when and how addresses are reused. These definitions also have an impact on the size of the state space of the abstract interpreter, since they directly interfere with the structure of (continuation) addresses. Hence, the definitions of $\widehat{alloc}$ and $\widehat{kalloc}$ immediately impact the precision of the analyses performed by the abstract interpreter. In Figure 3.9 an allocation strategy presented by Van Horn & Might (2012) is formalised, which uses variable names as addresses and expressions as continuation addresses. Note again that is crucial for the sets of abstract addresses and abstract continuation addresses, $\widehat{Addr}$ and $\widehat{KAddr}$, to be finite.

λ_0

$$\widehat{Addr} = Var \qquad\qquad \widehat{alloc}(x,\hat{\sigma}) = x$$

$$\widehat{KAddr} = Exp \qquad\qquad \widehat{kalloc}(e,\hat{\rho},\hat{\sigma},\widehat{\Xi}) = e$$

Figure 3.9.: Abstract addresses for λ_0.

The abstract transition relation $\hookrightarrow\ : \hat{\Sigma} \times \widehat{Store} \times \widehat{KStore} \to \hat{\Sigma} \times \widehat{Store} \times \widehat{KStore}$, using the abstract atomic evaluation rules and the definitions of the abstract address allocation functions, is depicted in Figure 3.10. The transition rules now take into account that values in the store and continuation store need to be joined upon updates and that multiple values may be present at the same address upon lookup.

λ_0

$$\frac{\hat{\rho},\hat{\sigma} \vdash ae \ \widehat{\Downarrow}\ \hat{v}}{\langle \mathbf{ev}(ae,\hat{\rho}),\hat{k}\rangle,\hat{\sigma},\widehat{\Xi} \hookrightarrow \langle \mathbf{val}(\hat{v}),\hat{k}\rangle,\hat{\sigma},\widehat{\Xi}}\ \text{ATOMIC-EVALUATION}$$

$$\frac{\hat{\phi} = \mathbf{fun}(e_a,\hat{\rho},\hat{k}) \qquad \hat{k}' = \widehat{kalloc}(e_f,\hat{\rho},\hat{\sigma},\widehat{\Xi})}{\langle \mathbf{ev}(\texttt{(}e_f\ e_a\texttt{)},\hat{\rho}),\hat{k}\rangle,\hat{\sigma},\widehat{\Xi} \hookrightarrow \langle \mathbf{ev}(e_f,\hat{\rho}),\hat{k}'\rangle,\hat{\sigma},\widehat{\Xi} \sqcup [\hat{k}' \mapsto \{\hat{\phi}\}]}\ \text{APPL-OPERATOR}$$

$$\frac{\mathbf{fun}(e_a,\hat{\rho},\hat{k}') \in \widehat{\Xi}(\hat{k}) \qquad \hat{\phi} = \mathbf{arg}(\hat{v}_f,\hat{k}') \qquad \hat{k}'' = \widehat{kalloc}(e_a,\hat{\rho},\hat{\sigma},\widehat{\Xi})}{\langle \mathbf{val}(\hat{v}_f),\hat{k}\rangle,\hat{\sigma},\widehat{\Xi} \hookrightarrow \langle \mathbf{ev}(e_a,\hat{\rho}),\hat{k}''\rangle,\hat{\sigma},\widehat{\Xi} \sqcup [\hat{k}'' \mapsto \{\hat{\phi}\}]}\ \text{APPL-OPERAND}$$

$$\frac{\mathbf{arg}(\hat{v}_f,\hat{k}') \in \widehat{\Xi}(\hat{k}) \qquad \hat{v}_f = \mathbf{clo}(lam,\hat{\rho}) \qquad lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \qquad \hat{a} = \widehat{alloc}(\hat{v}_a,\hat{\sigma})}{\langle \mathbf{val}(\hat{v}_a),\hat{k}\rangle,\hat{\sigma},\widehat{\Xi} \hookrightarrow \langle \mathbf{ev}(e,\hat{\rho}[x \mapsto \hat{a}]),\hat{k}'\rangle,\hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}_a\}],\widehat{\Xi}}\ \text{APPL-BODY}$$

$$\frac{\hat{\phi} = \mathbf{bnd}(x,e_b,\hat{\rho},\hat{k}') \qquad \hat{k}' = \widehat{kalloc}(e_x,\hat{\rho},\hat{\sigma},\widehat{\Xi})}{\langle \mathbf{ev}(\texttt{(letrec ((}x\ e_x\texttt{))}\ e_b\texttt{)},\hat{\rho}),\hat{k}\rangle \hookrightarrow \langle \mathbf{ev}(e_x,\hat{\rho}),k'\rangle,\hat{\sigma},\widehat{\Xi} \sqcup [\hat{k}' \mapsto \{\hat{\phi}\}]}\ \text{LETREC-BINDING}$$

$$\frac{\mathbf{bnd}(x,e_b,\hat{\rho},\hat{k}') \in \widehat{\Xi}(\hat{k}) \qquad \hat{a} = \widehat{alloc}(\hat{v}_x,\hat{\sigma})}{\langle \mathbf{val}(\hat{v}_x),\hat{k}\rangle,\hat{\sigma},\widehat{\Xi} \hookrightarrow \langle \mathbf{ev}(e_b,\hat{\rho}[x \mapsto \hat{a}]),\hat{k}'\rangle,\hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}_x\}],\widehat{\Xi}}\ \text{LETREC-BODY}$$

Figure 3.10.: Abstract transition rules for λ_0.

Due to the sets of values and continuation frames that may be stored at a particular address, nondeterminism is introduced in the analysis. After all, when an address is looked up in one of the stores, multiple values may be present. Hence, multiple transition rules may be applicable at the same time. As such, the abstract transition relation generates a graph of states, as shown in Figure 2.2.

Abstract Evaluation Function

The abstract evaluation function $\widehat{eval} : Exp \to \mathcal{P}(\hat{\Sigma} \times \widehat{Store} \times \widehat{KStore})$ defines the states the $\widehat{CESK}$ machine can reach during the abstract interpretation of an expression. It is defined as follows:

$$\widehat{eval}(e) = \{\langle \hat{\varsigma}, \hat{\sigma}, \hat{\Xi} \rangle \mid \widehat{inject}(e), [], [\hat{k}_0 \mapsto \mathbf{halt}] \hat{\hookrightarrow}{}^* \hat{\varsigma}, \hat{\sigma}, \hat{\Xi}\}.$$

Formally, the set $\widehat{eval}(e)$ is called the *abstract collecting semantics* of e and contains all states an abstract interpreter may reach in zero or more steps. By construction, this set is finite, making it suitable for performing static analyses. For a sound static analysis, the set $\widehat{eval}(e)$ must over-approximate the set $eval(e)$, that is, $\alpha(eval(e)) \sqsubseteq \widehat{eval}(e)$ under the given abstraction map α.

3.2. λ_ϕ, a Simple Concurrent Language

In this section, we extend λ_0 by adding *futures*, which can be used to build multithreaded programs. The language so obtained is called λ_ϕ. First, the syntax of λ_ϕ is presented. Afterwards, its concrete semantics is discussed, whereafter it is shown how this semantics can be abstracted. Again, our formalisation is based on the conventions of Stiévenart (2018).

3.2.1. Syntax

Figure 3.11 depicts the syntax of λ_ϕ. The set of expression types from λ_0 is *extended*, as is indicated by means of ellipsis. A future allows an expression to be evaluated in another thread and provides an interface to monitor that thread. Hence, an expression can be executed in another thread by means of the `future` primitive. For example `(future (fib 5))` creates a future that evaluates the expression `(fib 5)` in a new thread. Similarly, a nullary function can be called in another thread by means of `future-call`. To get the return value of a future, the function `deref` is used. A call to `deref` only succeeds if the future has already finished its computation; otherwise, it blocks until the future has finished its computation after which the return value of the future is returned, e.g., `(deref f)` returns the return value of the future `f` and blocks until it is present. Finally, it is also possible to abort the computation performed by a future using the `future-cancel` primitive, which stops the future and sets it to a special cancelled state.

The arguments to all functions except `future` are required to be atomic expressions since we assume the program is written in ANF (see Section 3.1.3). Only the argument to `future` may be any expression, as multithreaded evaluation is nonsensical for atomic expressions as they are evaluated in a single step (see the atomic evaluation rules for λ_0 in Figure 3.3). Also, only one expression can be passed to a future. This limitation is merely imposed to simplify our formalisation and does not impose functional restrictions on the language since multiple expressions can be combined together, for example using nested calls to `letrec`.

$\boxed{\lambda_\phi}$

$e \in \mathit{Exp} ::= \ldots$		Expressions
$\mid$	`(future` e`)`	*Future creation*
$\mid$	`(future-call` ae`)`	*Future creation*
$\mid$	`(future?` ae`)`	*Predicate*
$\mid$	`(deref` ae`)`	*State access*
$\mid$	`(future-done?` ae`)`	*Predicate*
$\mid$	`(future-cancel` ae`)`	*Future cancellation*
$\mid$	`(future-cancelled?` ae`)`	*Predicate*

Figure 3.11.: Syntactic extensions for λ_ϕ, a simple concurrent language.

Example 3.3 illustrates how futures may be used to create multithreaded applications.

Example 3.3 Parallel Computations using Futures

The function `sum-vector` in Listing 3.3 uses futures to create parallel computations in order to speed up a summation. The function `sum-vector` sums all the elements in a vector. Therefore, it uses a function `sum` which recursively divides the work among multiple futures: `sum` assigns half of the computation to a newly created future and takes care of the other half itself, except when the result is trivial and can be returned immediately. When both partial results have been calculated, they can be summed to produce the final result.

```scheme
(define (sum-vector vector)
  (define (sum left right)
    (if (>= left (- right 1))
        ;; Base case where the result is trivial.
        (vector-ref vector left)
        (let ((middle (round (+ left (/ (- right left) 2)))))
          ;; Create a new parallel computation.
          (let ((fut (future (sum left middle))))
            (let ((right-sum (sum middle right)))
              ;; Wait for the result of the future.
              (let ((left-sum (deref fut)))
                ;; Compute the final result.
                (+ left-sum right-sum)))))))
  (sum 0 (vector-length vector)))
```

Listing 3.3: Parallel vector summation.

Note that the algorithm in Listing 3.3 merely serves as an example of multithreaded computations using futures and that more performant algorithms exist.

3.2.2. Concrete Semantics

We formalise the semantics of λ_ϕ by presenting extensions to the CESK machine for λ_0. As a result, the abstract machine becomes a parallel CESK (PCESK) machine, due to the presence of multiple threads in the form of futures.

State Space

Figure 3.12 presents the extensions needed to the state space of λ_0 for λ_ϕ. Central to this extension is the addition of a thread map, Π, which provides a mapping from *thread identifiers* p to states ς, where a state ς represents the state of one thread. Since a thread is cancellable, a special thread state **cancelled** is added. Also, a new kind of value is added for futures. The letter p is used to denote thread identifiers to remain consistent with literature, which generally refers to threads and futures as *processes*.

$\boxed{\lambda_\phi}$

$\pi \in \Pi ::=$	$PID \to \Sigma$	Thread Map
$v \in Val ::=$	$\ldots$	Values
	$\mid$ $\mathbf{fut}(p)$	*Future*
$\varsigma \in \Sigma ::=$	$\ldots$	States
	$\mid$ **cancelled**	*Cancelled future*
$p \in PID$	A set of thread identifiers.	Thread Identifiers

Figure 3.12.: State space for λ_ϕ.

Injection Function

The injection function for λ_ϕ differs from the injection function for λ_0 due to the newly added thread map. The concurrent injection function $injc : Exp \to \Pi$ for λ_ϕ is defined as follows:

$$injc(e) = [p_0 \mapsto \langle \mathbf{ev}(e, []), k_0 \rangle]$$

where k_0 is a special continuation address reserved for the **halt** continuation and p_0 is a special thread identifier reserved for the initial (main) thread. When the machine reaches a state in which every thread has a value in its control component along with the **halt** continuation on top of its stack, the machine has completed evaluation and halts.

Transition Relation

Since λ_ϕ is an extension of λ_0, there is also a differentiation between atomic expressions and complex expressions. As no new atomic expressions have been introduced, the atomic evaluation function remains unaltered. Similarly, the address allocation functions used by the sequential CESK machine do not need modification. However, the transition function for λ_ϕ makes use of a third auxiliary function:

$$palloc : \Sigma \times \Pi \to PID$$

which is used to allocate new thread identifiers for futures. Like the address allocation functions, the definition of this function also impacts the precision of the analysis executed by an abstract interpreter. However, in the concrete case, the number of thread identifiers may be infinite. Hence, concrete thread identifier allocation can be defined analogously to concrete address allocation. Figure 3.13 shows the structure of thread identifiers in λ_ϕ, as well as the definition of *palloc*.

$\boxed{\lambda_\phi}$

$PID = \mathbb{N}$	Thread Identifiers
$palloc(\varsigma, \pi) = \mid dom(\pi) \mid$	Thread Identifier Allocation

Figure 3.13.: Thread identifiers for λ_ϕ.

The transition relation $\hookrightarrow: \Sigma \times Store \times KStore \rightarrow \Sigma \times Store \times KStore$ used for λ_0 is not suited to handle multiple threads. Therefore, we henceforth refer to it as the *sequential transition function*, since it defines how a single thread progresses during evaluation. For λ_ϕ, a *concurrent transition relation* $\rightsquigarrow: \Pi \times Store \times KStore \rightarrow \Pi \times Store \times KStore$ able to handle multiple threads is used. This most important rules of the concurrent transition relation are detailed in Figure 3.14. In each rule, the transition relation is annotated with the thread identifier p of the thread performing the transition ($\rightsquigarrow_p$):

- Rule SEQUENTIAL-STEP links the concurrent transition relation to the sequential transition relation: a thread being able to take a sequential step as described by the sequential transition relation can progress.
- Rules FUTURE and FUTURE-CALL stipulate how new futures are created. To create a new future, the thread map is extended with a binding from a newly allocated thread identifier to a new state containing the expression to be evaluated together with the current environment. In rule FUTURE-CALL, this expression is the body of the function referred to as well as its enclosing environment; the function argument is ignored as our formalisation only allows nullary functions, analogous to Clojure's semantics of `future-call`. The thread creating the future continues with a value representing a handle to the newly created future.
- Rule DEREF defines how the return value of a future can be read.

The remaining transition rules for λ_ϕ and their explanation are presented in Appendix A.

$\boxed{\lambda_\phi}$

$$\frac{\pi(p) = \varsigma \qquad \varsigma,\sigma,\Xi \hookrightarrow \varsigma',\sigma',\Xi'}{\pi,\sigma,\Xi \rightsquigarrow_p \pi[p \mapsto \varsigma'],\sigma',\Xi'}\ \textsc{sequential-step}$$

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(future } e\texttt{)},\rho),k\rangle \qquad \varsigma = \langle \mathbf{ev}(e,\rho),k_0\rangle \qquad p' = palloc(\varsigma,\pi)}{\pi,\sigma,\Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\mathbf{fut}(p')),k\rangle, p' \mapsto \varsigma],\sigma,\Xi}\ \textsc{future}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-call } ae\texttt{)},\rho),k\rangle \qquad \rho,\sigma \vdash ae \Downarrow v_f \qquad v_f = \mathbf{clo}(lam,\rho_f) \\ lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \qquad \varsigma = \langle \mathbf{ev}(e,\rho_f),k_0\rangle \qquad p' = palloc(\varsigma,\pi)\end{array}}{\pi,\sigma,\Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\mathbf{fut}(p')),k\rangle, p' \mapsto \varsigma],\sigma,\Xi}\ \textsc{future-call}$$

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(deref } ae\texttt{)},\rho),k\rangle \qquad \rho,\sigma \vdash ae \Downarrow \mathbf{fut}(p') \qquad \pi(p') = \langle \mathbf{val}(v),k_0\rangle}{\pi,\sigma,\Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(v),k\rangle],\sigma,\Xi}\ \textsc{deref}$$

Figure 3.14.: Main concurrent transition rules for λ_ϕ.

Since there is no rule to dereference a future that has not yet terminated, `deref` effectively is a blocking call, that is, a thread trying to dereference a non-terminated future will block until the future has terminated its computation.

The concurrent transition relation also shows how multithreaded evaluation is nondeterministic since it does not specify the order in which the threads should be transitioned, that is, any thread that can make a transition is allowed to do so. However, only one thread can make progress at any time due to the definition of the rules. Hence, the concurrent transition relation models all possible thread interleavings.

Evaluation Function

The evaluation function for λ_ϕ $evalc : Exp \to \mathcal{P}(\Pi \times Store \times KStore)$ defines the states reachable by the PCESK machine during the evaluation of an expression when starting with an empty store and a continuation store that only maps k_0 to the **halt** continuation. It is defined as follows:

$$evalc(e) = \{\langle \pi,\sigma,\Xi\rangle \mid injc(e), [], [k_0 \mapsto \mathbf{halt}] \rightsquigarrow^* \pi,\sigma,\Xi\}$$

where [] denotes the empty store.

3.2.3. Abstract Semantics

In this section, we present the abstract semantics of λ_ϕ and show how the PCESK machine just presented is transformed into an abstract PCESK machine, denoted as $\widehat{PCESK}$. Again, the goal is to obtain a machine with a finite state space. We now present the abstracted semantics of λ_ϕ by specifying the different components of the $\widehat{PCESK}$ machine; the differences with regard to the concrete semantics are marked in grey.

Abstract State Space

The abstract state space for λ_ϕ is depicted in Figure 3.15. By bounding the number of thread identifiers, the state space is made finite. This affects the abstract thread map $\widehat{\Pi}$, as multiple threads may now share the same thread identifier.

$\boxed{\lambda_\phi}$

$$\hat{\pi} \in \widehat{\Pi} ::= \widehat{PID} \to \mathcal{P}(\widehat{\Sigma})$$

$$\hat{v} \in \widehat{Val} ::= \dots \mid \mathbf{fut}(\hat{p})$$

$$\hat{\varsigma} \in \widehat{\Sigma} ::= \dots \mid \mathbf{cancelled}$$

$\hat{p} \in \widehat{PID}$ A finite set of thread identifiers.

Figure 3.15.: Abstract state space for λ_ϕ.

By bounding the number of thread identifiers, the state space is made finite (Might & Van Horn, 2011). To see why, consider the size of the state space of the abstract PCESK machine $(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore})$. In Section 3.1.4, we have already shown that the number of states $\hat{\varsigma}$ is finite, as well as the number of stores $\hat{\sigma}$ and continuation stores $\widehat{\Xi}$. To prove the entire state space is finite, we only need to show that the number of thread maps is finite, which is trivial since both the set of thread identifiers $\widehat{PID}$ and the set of states $\widehat{\Sigma}$ are finite.

Abstract Injection Function

The abstract injection function for λ_ϕ now injects an expression into an abstract state. The function is defined as follows:

$$\widehat{injc(e)} = [\widehat{p}_0 \mapsto \langle \mathbf{ev}(e, []), \widehat{k}_0 \rangle]$$

where $\widehat{k}_0$ is an abstract continuation address reserved for the **halt** continuation and $\widehat{p}_0$ is the thread identifier reserved for the initial thread.

Abstract Transition Relation

The transition relation for λ_ϕ is abstracted similarly to the sequential transition function for λ_0. Like the concrete concurrent transition relation, the abstract concurrent transition relation uses the following auxiliary function:

$$\widehat{palloc} : \widehat{\Sigma} \times \widehat{\Pi} \to \widehat{PID}$$

to allocate abstract thread identifiers. Like the abstract auxiliary functions for address allocation, its definition is very important to the precision of the $\widehat{PCESK}$ machine since it determines how thread identifiers are reused. After all, it is now required for $\widehat{PID}$ to be finite. Figure 3.16 depicts the structure of thread identifiers and defines the corresponding abstract allocation function.

λ_ϕ

$$\widehat{PID} = Exp \qquad\qquad \widehat{palloc}(\hat{\varsigma},\hat{\pi}) = e \text{ where } \hat{\varsigma} = \langle \mathbf{ev}(e,\_),\_\rangle$$

Figure 3.16.: Abstract thread identifiers for λ_ϕ.

This allocation strategy allocates the same abstract thread identifier for all threads that evaluate the same expression. Hence, the analysis cannot distinguish between such threads. We do not define $\widehat{palloc}$ when the control component of $\hat{\varsigma}$ is of the form $\mathbf{val}(\_)$ since such cases should never occur.

Of course, other definitions for $\widehat{palloc}$, as well as for $\widehat{alloc}$ and $\widehat{kalloc}$, are possible; we refer to Gilray et al. (2016) for an extensive study on allocation strategies. Importantly, they find that no allocation strategy produces an unsound analysis.

Given the above definition for $\widehat{palloc}$, it is easy to see that the set of abstract thread identifiers, $\widehat{PID}$, is finite. After all, we now have $|\widehat{PID}| = |\widehat{\Sigma}|$ and we have shown that the size of $\widehat{\Sigma}$ is finite in Section 3.1.4.

The most important rules of the abstract concurrent transition relation $\widehat{\rightsquigarrow} : \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore} \to \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}$ for λ_ϕ are detailed in Figure 3.17. In each rule, the transition relation is annotated with the abstract thread identifier $\hat{p}$ of the abstract thread performing the transition ($\widehat{\rightsquigarrow}_{\hat{p}}$). We refer again to Appendix A for the remaining abstract transition rules for λ_ϕ.

λ_ϕ

$$\frac{\hat{\varsigma} \in \hat{\pi}(\hat{p}) \qquad \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}{\hat{\pi},\hat{\sigma},\widehat{\Xi} \mathrel{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \hat{\varsigma}'],\hat{\sigma}',\widehat{\Xi}'} \text{ SEQUENTIAL-STEP}$$

$$\frac{\langle \mathbf{ev}(\texttt{(future } e\texttt{)},\hat{\rho}),\widehat{k}\rangle \in \hat{\pi}(\hat{p}) \qquad \hat{\varsigma} = \langle \mathbf{ev}(e,\hat{\rho}),\widehat{k}_0\rangle \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma},\hat{\pi})}{\hat{\pi},\hat{\sigma},\widehat{\Xi} \mathrel{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\mathbf{fut}(\hat{p}')),\widehat{k}\rangle, \hat{p}' \mapsto \hat{\varsigma}],\hat{\sigma},\widehat{\Xi}} \text{ FUTURE}$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-call } ae\texttt{)},\hat{\rho}),\widehat{k}\rangle \in \hat{\pi}(\hat{p}) \qquad \hat{\rho},\hat{\sigma} \vdash ae \Downarrow \hat{v}_f \qquad \hat{v}_f = \mathbf{clo}(lam,\hat{\rho}_f) \\ lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \qquad \hat{\varsigma} = \langle \mathbf{ev}(e,\hat{\rho}_f),\widehat{k}_0\rangle \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma},\hat{\pi})\end{array}}{\hat{\pi},\hat{\sigma},\widehat{\Xi} \mathrel{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\mathbf{fut}(\hat{p}')),\widehat{k}\rangle, \hat{p}' \mapsto \hat{\varsigma}],\hat{\sigma},\widehat{\Xi}} \text{ FUTURE-CALL}$$

$$\frac{\langle \mathbf{ev}(\texttt{(deref } ae\texttt{)},\hat{\rho}),\widehat{k}\rangle \in \hat{\pi}(\hat{p}) \qquad \hat{\rho},\hat{\sigma} \vdash ae \Downarrow \mathbf{fut}(\hat{p}') \qquad \langle \mathbf{val}(\hat{v}),\widehat{k}_0\rangle \in \hat{\pi}(\hat{p}')}{\hat{\pi},\hat{\sigma},\widehat{\Xi} \mathrel{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\hat{v}),\widehat{k}\rangle],\hat{\sigma},\widehat{\Xi}} \text{ DEREF}$$

Figure 3.17.: Main abstract concurrent transition rules for λ_ϕ.

The changes made to the concurrent transition relation to render it abstract all are related to the finiteness of the set of thread identifiers, which implies that now multiple thread states may be related to a single thread identifier. This causes more nondeterminism to arise in the analysis as multiple states may be present for any given thread identifier. As a result, multiple transition rules may be applicable at the same time and looking up the status of a thread may also become nondeterministic. Suppose for example a thread identifier p is related to two futures which both have finished their computation. When a thread now wants to get the result value of a future identified by p, rule DEREF is applicable twice, since multiple return values are related to p.

Abstract Evaluation Function

The abstract evaluation function for λ_ϕ $\widehat{evalc} : Exp \rightarrow \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore})$ defines the states the $\widehat{PCESK}$ machine can reach during the abstract interpretation of an expression. It is defined as follows:

$$\widehat{evalc}(e) = \{\langle \hat{\pi}, \hat{\sigma}, \widehat{\Xi} \rangle \mid \widehat{injc}(e), [], [\hat{k}_0 \mapsto \textbf{halt}] \rightsquigarrow^* \hat{\pi}, \hat{\sigma}, \widehat{\Xi}\}.$$

By construction, this set is finite and hence suitable for static analysis, contrary to the concrete evaluation function for λ_ϕ. This finiteness is an immediate result of the finite state space of the $\widehat{PCESK}$ machine.

3.3. λ_α, a Concurrent Language with Atoms

In this section, we present λ_α, a final extension to λ_0 and λ_ϕ. λ_α is a concurrent language with futures and *atoms*. Atoms provide race-free updates to shared state, that is, an atom provides functionalities to change the value it encapsulates while guaranteeing the absence of race conditions. It is typically required for the value stored within an atom to be immutable since direct mutations to the value stored within the atom are still subject to race conditions. Also, the values stored within multiple atoms cannot be updated in a coordinated manner.

We now introduce the new syntactic forms of λ_α and exemplify how atoms can be used in practice. Thereafter, the concrete and abstract semantics of λ_α are presented.

3.3.1. Syntax

The syntactic extensions for λ_α to λ_ϕ are outlined in Figure 3.18. Again, ANF is assumed. Five new expression types are presented. An atom can be created by means of the function `atom`, which takes an initial value to store within the atom as argument. For example, the expression `(atom 0)` creates an atom that stores the value zero. To update the value it stores, a function `reset!` is foreseen. The `compare-and-set!` function atomically compares the value contained in an atom to a given value and updates the value stored within the atom when the values are equal; `compare-and-set!` returns a boolean indicating whether the value within the atom was updated. The expression `(compare-and-set! atm 0 1)` stores the value `1` in atom `atm` if the atom currently stores `0`. The comparison of the value stored within the atom to `0` and the possible replacement of that value by `1` happen atomically. A third function to update the value within an atom is `swap!`, which can be used to change the value stored in an atom by applying a function to it, storing the function's result. However, if the value of the atom changes in the meantime, `swap!` starts again using the new value stored within the atom. For example, to increment the value stored within an atom `atm`, the following expression can be used: `(swap! atm inc)`. Finally, to read the value stored within an atom, the function `read` is provided.

λ_α

$e \in Exp ::= \ldots$		Expressions
$\mid$	`(atom` *ae*`)`	*Atom creation*
$\mid$	`(read` *ae*`)`	*State access*
$\mid$	`(reset!` *ae ae*`)`	*State change*
$\mid$	`(compare-and-set!` *ae ae ae*`)`	*Controlled state change*
$\mid$	`(swap!` *ae ae*`)`	*Controlled state change*

Figure 3.18.: Syntax of λ_α, a concurrent language with atoms.

The usage of the functions provided by atoms, as depicted in Figure 3.18, is illustrated in Example 3.4 which presents an implementation of the Producer-Consumer Problem. Next, Example 3.5 demonstrates how locks can be implemented using `compare-and-set!`.

Example 3.4 Producer-Consumer Problem

A work list shared by several threads is an example of shared state that can be managed using atoms. In the producer-consumer problem, some threads are *producers* who add items to the work list, and other threads are *consumers*, who take items from the work list. Several constraints apply, however. For example, the work list may never grow bigger than a specific size and every work item added to the list must (eventually) be consumed exactly once.

Listing 3.4 depicts a solution to this problem using atoms. The work list is encapsulated in an atom called `work`, which can safely be updated concurrently by using its dedicated functions. To consume an item, the work list is read. If it is empty, the consumer goes to sleep for some time and retries again later. Otherwise, the consumer tries to replace the list stored in the atom by a list of which the first element has been removed using `compare-and-set!` (line 10). This step only succeeds when the list stored in the atom was not manipulated since it was first read by the consumer. In this case, the consumer can consume the element it deleted from the work list. Otherwise, it has to try again. To produce an item, the producer first generates a new item which is to be added to the list. It then reads the list stored in the atom. If the list is full, the producer goes to sleep and tries again at a later point in time. Otherwise, it tries to replace the list by a list to which the newly created item is added (line 24). Upon success, the producer continues with the next item. Upon failure, the producer tries to add the same item again.

The solution presented in Listing 3.4 illustrates the use of atoms to manage shared memory.

```
1  (define work (atom '())) ;; Shared work list.
2  (define max-size 10) ;; Maximum size of the work list.
3
4  (define (consumer)
5    (let loop ()
6      (let ((worklist (read work))) ;; Read work list.
7        (if (null? worklist)
          (sleep 10) ;; Work list is empty, go to sleep.
          ;; Try to pop an item of the work list and process it.
          (begin (if (compare-and-set! work worklist (cdr worklist))
                     (consume-item (car worklist))))))
    (loop)))

(define (producer)
  (let loop ()
    ;; Create a work item and push it on the worklist.
    (let try-add ((workitem (produce-item)))
      (let ((worklist (read work)))
        (if (>= (length worklist) max-size)
            (begin
              (sleep 10) ;; Work list is full, go to sleep and try
               ↪  again.
              (try-add workitem))
            (begin
              (if (compare-and-set! work worklist (cons workitem
               ↪  worklist))
                  (loop) ;; Item has been added successfully, produce
                   ↪  next.
                  (try-add (read work)))))))) ;; Work list has been
                   ↪  changed, try again.

(define (consume-item item) ...)
(define (produce-item) ...)
```

Listing 3.4: Solution to the Producer-Consumer Problem using atoms.

Example 3.5 Lock Implementation

Concurrency primitives can often be used to build other known concurrency primitives. A concurrency primitive that can be built using the `compare-and-swap!` functionality provided by atoms are spinning locks.

A lock can be represented as an atom that stores a boolean which indicates whether the lock is held (*true*) or not (*false*). By storing the boolean in an atom, safe concurrent updates become possible. To acquire a lock, `compare-and-set!` is used to change the value stored in the atom to *true*, indicating the lock is held by a thread. Upon success, `compare-and-set!` returns true and the thread holds the lock. Otherwise, the thread loops until it succeeds to acquire the lock. To release a lock, it suffices to set the value encapsulated in the atom to *false* using `reset!`. There is no need to use `compare-and-set!` for this operation, since only the thread holding the lock can modify the value inside the atom. The complete implementation is shown in Listing 3.5.

```scheme
(define (new-lock) (atom #f))
(define (acquire lock)
  (let try () ;; Spin loop.
    (if (compare-and-set! lock #f #t)
        #t
        (try))))
(define (release lock) (reset! lock #f))
```

Listing 3.5: Implementation of spinning locks using atoms.

3.3.2. Concrete Semantics

We now present the concrete semantics of λ_α. The injection and evaluation functions of the PCESK machine will not be presented since these do not need modification. For their definition, we refer back to Section 3.2.2.

State Space

Figure 3.19 depicts the extensions needed to the state space of λ_ϕ for λ_α. Only minor extensions, a new value type for atoms and a new continuation frame for `swap!`, are necessary.

$\boxed{\lambda_\alpha}$

$$
\begin{array}{rcll}
v \in \mathit{Val} & ::= & \ldots & \text{Values} \\
& \mid & \mathbf{atom}(a) & \textit{An atom} \\
\phi \in \mathit{Frame} & ::= & \ldots & \text{Continuation frames} \\
& \mid & \mathbf{swp}(v, v, v, k) & \texttt{swap!}\ \textit{continuation}
\end{array}
$$

Figure 3.19.: State space for λ_α.

Transition Function

The concurrent transition relation $\rightsquigarrow$: $\Pi \times \mathit{Store} \times \mathit{KStore} \rightarrow \Pi \times \mathit{Store} \times \mathit{KStore}$ for λ_ϕ needs to be extended to support the newly introduced functions on atoms. Since these functions are executed locally to one thread, we formalise them by extending the *sequential* transition function $\hookrightarrow$: $\Sigma \times \mathit{Store} \times \mathit{KStore} \rightarrow \Sigma \times \mathit{Store} \times \mathit{KStore}$. By means of rule SEQUENTIAL-STEP, the transition rules for atoms also become part of the concurrent transition relation (see Figure 3.14).

Figure 3.20 depicts the rules that are added to the sequential transition relation to support atoms. Eight new rules are added:

- Rule ATOM defines the creation of an atom. Formally, an atom contains the address of the value it is said to store; the value itself is added to store.
- Rule READ stipulates how the value stored within an atom is read: to read the value stored within an atom, the value related to the address contained by the atom is looked up in the store.

- Rule RESET states that the value stored within an atom can be altered by updating the corresponding entry in the store.
- Rules CAS-T and CAS-F define `compare-and-set!`. To evaluate this function, the value stored within the atom is compared to its second argument. If the values are equal, the value stored within the atom is replaced and true is returned. Otherwise, nothing happens and false is returned. The premises indicating this condition are indicated in red. The arguments to `compare-and-set!` being atomic expressions guarantees that the function can be evaluated in a single step, thereby guaranteeing atomicity of the comparison and value update.
- Rules SWAP, SWAP-SUCCEED and SWAP-FAIL stipulate how `swap!` is executed. The argument to `swap!` is a unary function which is called with the value stored in the atom as its argument. If after the evaluation of the function call the value stored in the atom is unchanged, it is replaced with the result of the function call. Otherwise, the function is recalled with the new value that has been stored in the atom. As a result, the function that is given to `swap!` may be executed multiple times. The premises indicating the condition that the value must remain unaltered are indicated in red.

λ_α

$$\frac{a = alloc(ae, \sigma) \qquad \rho, \sigma \vdash ae \Downarrow v}{\langle \mathbf{ev}(\texttt{(atom}\ ae\texttt{)}, \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{val}(\mathbf{atom}(a)), k\rangle, \sigma[a \mapsto v], \Xi}\ \textsc{atom}$$

$$\frac{\rho, \sigma \vdash ae \Downarrow \mathbf{atom}(a) \qquad v = \sigma(a)}{\langle \mathbf{ev}(\texttt{(read}\ ae\texttt{)}, \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{val}(v), k\rangle, \sigma, \Xi}\ \textsc{read}$$

$$\frac{\rho, \sigma \vdash ae_a \Downarrow \mathbf{atom}(a) \qquad \rho, \sigma \vdash ae_v \Downarrow v}{\langle \mathbf{ev}(\texttt{(reset!}\ ae_a\ ae_v\texttt{)}, \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{val}(v), k\rangle, \sigma[a \mapsto v], \Xi}\ \textsc{reset}$$

$$\frac{\rho, \sigma \vdash ae_a \Downarrow \mathbf{atom}(a) \qquad \rho, \sigma \vdash ae_{old} \Downarrow v_{old} \qquad \rho, \sigma \vdash ae_{new} \Downarrow v_{new} \qquad \sigma(a) = v_{old}}{\langle \mathbf{ev}(\texttt{(compare-and-set!}\ ae_a\ ae_{old}\ ae_{new}\texttt{)}, \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{val}(\texttt{\#t}), k\rangle, \sigma[a \mapsto v_{new}], \Xi}\ \textsc{cas-t}$$

$$\frac{\rho, \sigma \vdash ae_a \Downarrow \mathbf{atom}(a) \qquad \rho, \sigma \vdash ae_{old} \Downarrow v_{old} \qquad \rho, \sigma \vdash ae_{new} \Downarrow v_{new} \qquad \sigma(a) \neq v_{old}}{\langle \mathbf{ev}(\texttt{(compare-and-set!}\ ae_a\ ae_{old}\ ae_{new}\texttt{)}, \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{val}(\texttt{\#f}), k\rangle, \sigma, \Xi}\ \textsc{cas-f}$$

$$\frac{\begin{array}{c} \rho, \sigma \vdash ae_a \Downarrow \mathbf{atom}(a) \\ \sigma(a) = v_{old} \qquad \rho, \sigma \vdash ae_f \Downarrow v_f \qquad v_f = \mathbf{clo}(lam, \rho_f) \qquad lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \\ \phi = \mathbf{swp}(\mathbf{atom}(a), v_{old}, k, v_f) \qquad a' = alloc(v_{old}, \sigma) \qquad k' = kalloc(e, \rho_f, \sigma, \Xi) \end{array}}{\langle \mathbf{ev}(\texttt{(swap!}\ ae_a\ ae_f\texttt{)}, \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{ev}(e, \rho_f[x \mapsto a']), k'\rangle, \sigma[a' \mapsto v_{old}], \Xi[k' \mapsto \phi]}\ \textsc{swap}$$

$$\frac{\Xi(k) = \mathbf{swp}(\mathbf{atom}(a), v_{old}, k', v_f) \qquad \sigma(a) = v_{old}}{\langle \mathbf{val}(v), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{val}(v), k'\rangle, \sigma[a \mapsto v], \Xi}\ \textsc{swap-succeed}$$

$$\frac{\begin{array}{c} \Xi(k) = \mathbf{swp}(\mathbf{atom}(a), v_{old}, k', v_f) \qquad \sigma(a) = v_{curr} \\ v_{curr} \neq v_{old} \qquad \phi = \mathbf{swp}(\mathbf{atom}(a), v_{curr}, k', v_f) \qquad v_f = \mathbf{clo}(lam, \rho_f) \\ lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \qquad a' = alloc(v_{curr}, \sigma) \qquad k'' = kalloc(e, \rho_f, \sigma, \Xi) \end{array}}{\langle \mathbf{val}(v), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{ev}(e, \rho_f[x \mapsto a']), k''\rangle, \sigma[a' \mapsto v_{curr}], \Xi[k'' \mapsto \phi]}\ \textsc{swap-fail}$$

Figure 3.20.: Sequential transition rules for λ_α.

3.3.3. Abstract Semantics

In this section, we present the abstract semantics of λ_α. The only part of the PCESK machine that needs abstraction is the transition function, of which the abstract version is shown in Figure 3.21. Changes with respect to the concrete evaluation rules are marked in grey.

$\boxed{\lambda_\alpha}$

$$\frac{\hat{a} = \widehat{alloc}(ae, \hat{\sigma}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathrel{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(\texttt{(atom } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{val}(\mathbf{atom}(\hat{a})), \widehat{k}\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}} \textsc{atom}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{atom}(\hat{a}) \qquad \hat{v} \in \hat{\sigma}(\hat{a})}{\langle \mathbf{ev}(\texttt{(read } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat{v}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{read}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae_a \mathrel{\widehat{\Downarrow}} \mathbf{atom}(\hat{a}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae_v \mathrel{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(\texttt{(reset! } ae_a\ ae_v\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat{v}), \widehat{k}\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}} \textsc{reset}$$

$$\frac{\begin{array}{c}\hat{\rho}, \hat{\sigma} \vdash ae_a \mathrel{\widehat{\Downarrow}} \mathbf{atom}(\hat{a}) \\ \hat{\rho}, \hat{\sigma} \vdash ae_{old} \mathrel{\widehat{\Downarrow}} \hat{v}_{old} \qquad \hat{\rho}, \hat{\sigma} \vdash ae_{new} \mathrel{\widehat{\Downarrow}} \hat{v}_{new} \qquad \hat{v}_{curr} \in \hat{\sigma}(\hat{a}) \qquad \hat{v}_{curr} = \hat{v}_{old}\end{array}}{\langle \mathbf{ev}(\texttt{(compare-and-set! } ae_a\ ae_{old}\ ae_{new}\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{val}(\texttt{\#t}), \widehat{k}\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}_{new}], \widehat{\Xi}} \textsc{cas-t}$$

$$\frac{\begin{array}{c}\hat{\rho}, \hat{\sigma} \vdash ae_a \mathrel{\widehat{\Downarrow}} \mathbf{atom}(\hat{a}) \\ \hat{\rho}, \hat{\sigma} \vdash ae_{old} \mathrel{\widehat{\Downarrow}} \hat{v}_{old} \qquad \hat{\rho}, \hat{\sigma} \vdash ae_{new} \mathrel{\widehat{\Downarrow}} \hat{v}_{new} \qquad \hat{v}_{curr} \in \hat{\sigma}(\hat{a}) \qquad \hat{v}_{curr} \neq \hat{v}_{old}\end{array}}{\langle \mathbf{ev}(\texttt{(compare-and-set! } ae_a\ ae_{old}\ ae_{new}\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{val}(\texttt{\#f}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{cas-f}$$

$$\frac{\begin{array}{c}\hat{\rho}, \hat{\sigma} \vdash ae_a \mathrel{\widehat{\Downarrow}} \mathbf{atom}(\hat{a}) \\ \hat{v}_{old} \in \hat{\sigma}(\hat{a}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae_f \mathrel{\widehat{\Downarrow}} \hat{v}_f \qquad \hat{v}_f = \mathbf{clo}(lam, \hat{\rho}_f) \qquad lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \\ \hat{\phi} = \mathbf{swp}(\mathbf{atom}(\hat{a}), \hat{v}_{old}, \widehat{k}, \hat{v}_f) \qquad \hat{a}' = \widehat{alloc}(\hat{v}_{old}, \hat{\sigma}) \qquad \widehat{k}' = \widehat{kalloc}(e, \hat{\rho}_f, \hat{\sigma}, \widehat{\Xi})\end{array}}{\langle \mathbf{ev}(\texttt{(swap! } ae_a\ ae_f\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{ev}(e, \hat{\rho}_f[x \mapsto \hat{a}']), \widehat{k}'\rangle, \hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{v}_{old}], \widehat{\Xi} \sqcup [\widehat{k}' \mapsto \hat{\phi}]} \textsc{swap}$$

$$\frac{\mathbf{swp}(\mathbf{atom}(\hat{a}), \hat{v}_{old}, \widehat{k}', \hat{v}_f) \in \widehat{\Xi}(\widehat{k}) \qquad \hat{v}_{curr} \in \hat{\sigma}(\hat{a}) \qquad \hat{v}_{curr} = \hat{v}_{old}}{\langle \mathbf{val}(\hat{v}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat{v}), \widehat{k}'\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}} \textsc{swap-succeed}$$

$$\frac{\begin{array}{c}\widehat{\Xi}(\widehat{k}) = \mathbf{swp}(\mathbf{atom}(\hat{a}), \hat{v}_{old}, \widehat{k}', \hat{v}_f) \qquad \hat{v}_{curr} \in \hat{\sigma}(\hat{a}) \\ \hat{v}_{curr} \neq \hat{v}_{old} \qquad \hat{\phi} = \mathbf{swp}(\mathbf{atom}(\hat{a}), \hat{v}_{curr}, \widehat{k}', \hat{v}_f) \qquad \hat{v}_f = \mathbf{clo}(lam, \hat{\rho}_f) \\ lam = \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \qquad \hat{a}' = \widehat{alloc}(\hat{v}_{curr}, \hat{\sigma}) \qquad \widehat{k}'' = \widehat{kalloc}(e, \hat{\rho}_f, \hat{\sigma}, \widehat{\Xi})\end{array}}{\langle \mathbf{val}(\hat{v}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{ev}(e, \hat{\rho}_f[x \mapsto \hat{a}']), \widehat{k}''\rangle, \hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{v}_{curr}], \widehat{\Xi} \sqcup [\widehat{k}'' \mapsto \hat{\phi}]} \textsc{swap-fail}$$

Figure 3.21.: Abstract sequential transition rules for λ_α.

3.4. Conclusion

In this chapter, we have gradually expanded the formalisation of a simple sequential language, λ_0, by successively adding futures (λ_ϕ) and atoms (λ_α). We also presented, for each language, their abstract semantics according to the AAM technique from Van Horn & Might (2012). By doing so, we have obtained the formalisation of a non-modular abstract interpreter for λ_α, a concurrent language with futures and atoms.

The abstraction of the interpreter is a source of nondeterminism and precision loss. As multiple items may be stored at the same location in the stores and thread map, all options must be considered by the abstract interpreter, that is, multiple transition rules may be applicable at a given point in the analysis of a program. In addition, the transition rules of the concurrent transition relation do not define in which order threads need to be evaluated and hence, all interleavings are possible. Additionally, the abstraction also causes the interpreter's state space to be finite. As a result, the obtained abstract interpreter is suitable for static analysis and its precision can be tuned by the specification of $\widehat{alloc}$, $\widehat{kalloc}$ and $\widehat{palloc}$, which influences the size of the state space and determines how addresses and thread identifiers are reused. A bigger state space may result in a higher precision but at the cost of a higher computation time; a smaller state space may result in a lower precision, but lowers the analysis' computation time.

4 An Incremental Thread-Modular Analysis for λ_α

In this chapter, we present an incremental thread-modular analysis for λ_α. In Section 4.1, we first present a formalisation of thread interference based on the concept of *effects*, analogously to what is common in recent literature. By doing so, we extend λ_α to λ_ϵ. Thereafter, in Section 4.2, we present an algorithm for a thread-modular analysis of λ_ϵ. This algorithm is built according to the ModConc design method of Stiévenart (2018) and is used as a basis for our own incremental algorithm, which is presented in Section 4.3. In Section 4.4, we discuss some possible optimisations our incremental algorithm may benefit from. In Section 4.5, we reflect on the presented algorithms by discussing some general considerations. Finally, in Section 4.6, we conclude this chapter by showing how the different analysis algorithms analyse a simple concurrent program and by comparing their behaviour and results.

4.1. λ_ϵ, a Formalisation of Thread Interference for λ_α

In Section 2.3.2, we already presented the concept of a thread-modular analysis for concurrent languages. In a thread-modular analysis, the different abstract threads in a program are analysed in isolation; in λ_α, threads are created by futures. Since threads may not be totally independent, the actions of one thread may cause other threads to be reanalysed; this reanalysis is needed to ensure soundness: if a thread would not be reanalysed, it cannot take into account the behaviour of other threads. This is unneeded in a non-modular analysis, since there, all thread interleavings are analysed explicitly.

In this section, we present a formalisation of thread interference, on which our thread-modular analysis will be based. To do so, we first identify the behavioural aspects of threads that cause them to interfere with one another. We then formalise these aspects and apply them to λ_α, resulting in λ_ϵ.

In a multithreaded program, the behaviour of one thread may influence the behaviour of one or more other threads. Such influences may be caused by the modifications of shared-memory, synchronisation and dynamic thread creation, for example. In general, we identify four standard behavioural aspects of λ_α-threads that cause them to interfere:

- A thread may **create** a new parallel computation (thread) that executes a given expression e in parallel to itself. In λ_α, parallel computations are created by calling the `future` and `future-call` procedures.

- A thread may **dereference** another thread to read its return value. In λ_α, reading the return value of a thread is done by means of a call to `deref`. This call may block, depending on the state of the dereferenced thread.
- A thread may **write** a new value to a certain address in shared memory. In λ_α, all memory is represented by means of the store σ and may be shared among threads.
- A thread may **read** a value from a certain address in memory. If the memory is shared, the thread will read the value that was last written by any of the threads that have access to that address in memory.
- A thread may **inspect** another thread's state. In λ_α, a thread can call `future-done?` to verify whether another thread has finished its execution and a thread may call `future-cancelled?` to verify whether another thread has been cancelled.
- A thread may **cancel** another thread's execution. In λ_α, calling `future-cancel` with the thread identifier of a future will cause that future to abort its computation.

In summary, threads may influence each other by the *effects* they have on their shared environment. For brevity, henceforth we omit the effects that are related to future cancellation and state inspection.

Now we have presented the effects threads may have on their environment, and therefore on each other, we present their formalisation. This formalisation requires two steps. First, we describe a formal notation for the effects themselves. Then, we annotate both the sequential and concurrent transition relations of λ_α with these effects, as well as the atomic evaluation relation. These annotations indicate, for each transition rule, the effects *generated* by the application of that rule. Since these annotations are identical for the concrete and abstract transition rules, we will only present the annotated abstract transition rules. Also, we only show the rules that need modification, omitting the rules that do not generate effects, and indicate the added effects in red. We will henceforth denote this extended formalisation of λ_α with effects as λ_ϵ. It is thus important to note that we do not extend the language itself, but merely its formalisation; the actual (abstract) transition rules do not change, but are only annotated with effects.

4.1.1. Effects

Figure 4.1 depicts the formalisation of the effects just presented. Upon the creation of a new thread, a *creation effect* $\mathbf{c}(p)$ is generated. This effect contains the thread identifier p of the newly created future. Similarly, when a future is dereferenced, a *dereferencing effect* $\mathbf{d}(p)$ is generated by the thread performing the dereferencing. Again, this effect contains the thread identifier p of the future that is dereferenced. When a thread reads a value at a particular address in the store, it generates a *read effect* $\mathbf{r}(a)$ containing the address a that was read. Likewise, when a thread writes a value at a particular address in the store, it generates a *write effect* $\mathbf{w}(a)$ containing the address a that was written to.

By incorporating thread identifiers and addresses in the effects, it becomes possible to track the behaviour of threads and to infer their influence on one another. It is not needed to include the thread identifier of the thread generating the effect since this can be inferred as the concurrent transition rule is annotated with this identifier. The exact use of these effects will be discussed in Section 4.2.

$\boxed{\lambda_\epsilon}$

$\mathit{eff} \in \mathit{Effect} ::=$		Effects
	$\mathbf{c}(p)$	*Future creation*
	$\mid\ \mathbf{d}(p)$	*Future dereferencing*
	$\mid\ \mathbf{r}(a)$	*Address read*
	$\mid\ \mathbf{w}(a)$	*Address write*

Figure 4.1.: Effects for λ_ϵ.

For brevity, we omit the definition of the abstract effects for λ_ϵ; their definition is completely analogous to the one of the concrete effects shown in Figure 4.1, but they include abstract thread identifiers and abstract addresses instead of concrete thread identifiers and concrete addresses respectively. We will also only annotate the stored thread identifiers and addresses with a hat, as this is sufficient to see that an effect is abstract.

4.1.2. Abstract Atomic Evaluation Relation

The abstract atomic evaluation relation for λ_ϵ is depicted in Figure 4.2. The generation of an abstract effect by the abstract atomic evaluation relation is denoted as $\underset{\widehat{\mathit{eff}}}{\widehat{\Downarrow}}$. Rule VAR now generates an abstract read effect. To see why, observe the abstract atomic evaluation of a variable x: to abstractly evaluate this variable, the abstract address $\widehat{a}$, related to x by the environment $\hat{\rho}$, is looked up in the store σ ($\hat{v} \in \hat{\sigma}(\hat{a})$), which results in the abstract value $\hat{v}$. In consequence, because the abstract address $\hat{a}$ is read, rule VAR needs to generate an abstract read effect. Rule LAMBDA remains unmodified as no effects need to be generated; for this reason, the rule is omitted.

$\boxed{\lambda_\epsilon}$

$$\frac{\hat{a} = \hat{\rho}(x) \qquad \hat{v} \in \hat{\sigma}(\hat{a})}{\hat{\rho}, \hat{\sigma} \vdash x \underset{\mathbf{r}(\hat{a})}{\widehat{\Downarrow}} \hat{v}}\ \textsc{var}$$

Figure 4.2.: Modified abstract atomic evaluation rules for λ_ϵ.

4.1.3. Abstract Sequential Transition Relation

The abstract sequential transition relation for λ_ϵ is depicted in Figure 4.3. For each rule, the generated effects are written underneath the transition arrow. Since this notation becomes slightly unpractical when multiple effects are generated by the execution of a transition rule, we sometimes use a shorthand notation, annotating the transition arrow with a set of effects. Again, rules that do not generate effects have been omitted.

$\boxed{\lambda_\epsilon}$

$$\frac{\hat\rho,\hat\sigma \vdash ae \underset{\widehat{eff}}{\widehat{\Downarrow}} \hat v}{\langle \mathbf{ev}(ae,\hat\rho),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{\widehat{eff}}{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat v),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi}}\ \textsc{Atomic-Evaluation}$$

$$\frac{\mathbf{arg}(\hat v_f,\widehat{k}') \in \widehat{\Xi}(\widehat{k}) \qquad \hat v_f = \mathbf{clo}(lam,\hat\rho) \qquad lam = (\lambda(x)\ \ e) \qquad \hat a = \widehat{alloc}(\hat v_a,\hat\sigma)}{\langle \mathbf{val}(\hat v_a),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{\mathbf{w}(\hat a)}{\widehat{\hookrightarrow}} \langle \mathbf{ev}(e,\hat\rho[x\mapsto \hat a]),\widehat{k}'\rangle,\hat\sigma \sqcup [\hat a \mapsto \{\hat v_a\}],\widehat{\Xi}}\ \textsc{Appl-Body}$$

$$\frac{\mathbf{bnd}(x,e_b,\hat\rho,\widehat{k}') \in \widehat{\Xi}(\widehat{k}) \qquad \hat a = \widehat{alloc}(\hat v_x,\hat\sigma)}{\langle \mathbf{val}(\hat v_x),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{\mathbf{w}(\hat a)}{\widehat{\hookrightarrow}} \langle \mathbf{ev}(e_b,\hat\rho[x\mapsto \hat a]),\widehat{k}'\rangle,\hat\sigma \sqcup [\hat a \mapsto \{\hat v_x\}],\widehat{\Xi}}\ \textsc{Letrec-Body}$$

$$\frac{\hat a = \widehat{alloc}(ae,\hat\sigma) \qquad \hat\rho,\hat\sigma \vdash ae \underset{\widehat{eff}}{\widehat{\Downarrow}} \hat v}{\langle \mathbf{ev}(\texttt{(atom}\ \ ae\texttt{)},\hat\rho),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{\substack{\widehat{eff}\\ \mathbf{w}(\hat a)}}{\widehat{\hookrightarrow}} \langle \mathbf{val}(\mathbf{atom}(\hat a)),\widehat{k}\rangle,\hat\sigma \sqcup [\hat a \mapsto \hat v],\widehat{\Xi}}\ \textsc{Atom}$$

$$\frac{\hat\rho, store \vdash ae \underset{\widehat{eff}}{\widehat{\Downarrow}} \mathbf{atom}(\hat a) \qquad \hat v \in \hat\sigma(\hat a)}{\langle \mathbf{ev}(\texttt{(read}\ \ ae\texttt{)},\hat\rho),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{\substack{\widehat{eff}\\ \mathbf{r}(\hat a)}}{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat v),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi}}\ \textsc{Read}$$

$$\frac{\hat\rho,\hat\sigma \vdash ae_a \underset{\widehat{eff}_a}{\widehat{\Downarrow}} \mathbf{atom}(\hat a) \qquad \hat\rho,\hat\sigma \vdash ae_v \underset{\widehat{eff}_v}{\widehat{\Downarrow}} \hat v \qquad E = \{\widehat{eff}_a, \widehat{eff}_v, \mathbf{w}(\hat a)\}}{\langle \mathbf{ev}(\texttt{(reset!}\ \ ae_a\ ae_v\texttt{)},\hat\rho),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{E}{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat v),\widehat{k}\rangle,\hat\sigma \sqcup [\hat a \mapsto \hat v],\widehat{\Xi}}\ \textsc{Reset}$$

$$\frac{\begin{array}{c}\hat\rho,\hat\sigma \vdash ae_a \underset{\widehat{eff}_a}{\widehat{\Downarrow}} \mathbf{atom}(\hat a) \qquad \hat\rho,\hat\sigma \vdash ae_{old} \underset{\widehat{eff}_{old}}{\widehat{\Downarrow}} \hat v_{old} \qquad \hat\rho,\hat\sigma \vdash ae_{new} \underset{\widehat{eff}_{new}}{\widehat{\Downarrow}} \hat v_{new}\\ \hat v_{curr} \in \hat\sigma(\hat a) \qquad \hat v_{curr} = \hat v_{old} \qquad E = \{\widehat{eff}_a, \widehat{eff}_{old}, \widehat{eff}_{new}, \mathbf{r}(\hat a), \mathbf{w}(\hat a)\}\end{array}}{\langle \mathbf{ev}(\texttt{(compare-and-set!}\ \ ae_a\ ae_{old}\ ae_{new}\texttt{)},\hat\rho),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{E}{\widehat{\hookrightarrow}} \langle \mathbf{val}(\texttt{\#t}),\widehat{k}\rangle,\hat\sigma \sqcup [\hat a \mapsto \hat v_{new}],\widehat{\Xi}}\ \textsc{CAS-T}$$

$$\frac{\begin{array}{c}\hat\rho,\hat\sigma \vdash ae_a \underset{\widehat{eff}_a}{\widehat{\Downarrow}} \mathbf{atom}(\hat a) \qquad \hat\rho,\hat\sigma \vdash ae_{old} \underset{\widehat{eff}_{old}}{\widehat{\Downarrow}} \hat v_{old} \qquad \hat\rho,\hat\sigma \vdash ae_{new} \underset{\widehat{eff}_{new}}{\widehat{\Downarrow}} \hat v_{new}\\ \hat v_{curr} \in \hat\sigma(\hat a) \qquad \hat v_{curr} \neq \hat v_{old} \qquad E = \{\widehat{eff}_a, \widehat{eff}_{old}, \widehat{eff}_{new}, \mathbf{r}(\hat a)\}\end{array}}{\langle \mathbf{ev}(\texttt{(compare-and-set!}\ \ ae_a\ ae_{old}\ ae_{new}\texttt{)},\hat\rho),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{E}{\widehat{\hookrightarrow}} \langle \mathbf{val}(\texttt{\#f}),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi}}\ \textsc{CAS-F}$$

$$\frac{\begin{array}{c}\hat\rho,\hat\sigma \vdash ae_a \underset{\widehat{eff}_a}{\widehat{\Downarrow}} \mathbf{atom}(\hat a) \qquad \hat v_{old} \in \hat\sigma(\hat a) \qquad \hat\rho,\hat\sigma \vdash ae_f \underset{\widehat{eff}_f}{\widehat{\Downarrow}} \hat v_f\\ \hat v_f = \mathbf{clo}(lam,\hat\rho_f) \qquad lam = (\lambda(x)\ \ e) \qquad \hat\phi = \mathbf{swp}(\mathbf{atom}(\hat a),\hat v_{old},\widehat{k},\hat v_f)\\ \hat a' = \widehat{alloc}(\hat v_{old},\hat\sigma) \qquad \widehat{k}' = \widehat{kalloc}(e,\hat\rho_f,\hat\sigma,\widehat{\Xi}) \qquad E = \{\widehat{eff}_a, \widehat{eff}_f, \mathbf{r}(\hat a), \mathbf{w}(\hat a')\}\end{array}}{\langle \mathbf{ev}(\texttt{(swap!}\ \ ae_a\ ae_f\texttt{)},\hat\rho),\widehat{k}\rangle,\hat\sigma,\widehat{\Xi} \underset{E}{\widehat{\hookrightarrow}} \langle \mathbf{ev}(e,\hat\rho_f[x\mapsto \hat a']),\widehat{k}'\rangle,\hat\sigma \sqcup [\hat a' \mapsto \hat v_{old}],\widehat{\Xi} \sqcup [\widehat{k}' \mapsto \hat\phi]}\ \textsc{Swap}$$

λ_ϵ

$$\frac{\mathbf{swp}(\mathbf{atom}(\hat{a}), \hat{v}_{old}, \widehat{k}', \hat{v}_f) \in \widehat{\Xi}(\widehat{k}) \qquad \hat{v}_{curr} \in \hat{\sigma}(\hat{a}) \qquad \hat{v}_{curr} = \hat{v}_{old}}{\langle \mathbf{val}(\hat{v}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \underset{\substack{\mathbf{r}(\hat{a})\\ \mathbf{w}(\hat{a})}}{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat{v}), \widehat{k}'\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}} \ \textsc{swap-succeed}$$

$$\frac{\begin{array}{c}\widehat{\Xi}(\widehat{k}) = \mathbf{swp}(\mathbf{atom}(\hat{a}), \hat{v}_{old}, \widehat{k}', \hat{v}_f) \qquad \hat{v}_{curr} \in \hat{\sigma}(\hat{a}) \\ \hat{v}_{curr} \neq \hat{v}_{old} \qquad \hat{\phi} = \mathbf{swp}(\mathbf{atom}(\hat{a}), \hat{v}_{curr}, \widehat{k}', \hat{v}_f) \qquad \hat{v}_f = \mathbf{clo}(lam, \hat{\rho}_f) \\ lam = (\lambda\,(x)\ \ e) \qquad \hat{a}' = \widehat{alloc}(\hat{v}_{curr}, \hat{\sigma}) \qquad \widehat{k}'' = \widehat{kalloc}(e, \hat{\rho}_f, \hat{\sigma}, \widehat{\Xi})\end{array}}{\langle \mathbf{val}(\hat{v}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \underset{\substack{\mathbf{r}(\hat{a})\\ \mathbf{w}(\hat{a}')}}{\widehat{\hookrightarrow}} \langle \mathbf{ev}(e, \hat{\rho}_f[x \mapsto \hat{a}']), \widehat{k}''\rangle, \hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{v}_{curr}], \widehat{\Xi} \sqcup [\widehat{k}'' \mapsto \hat{\phi}]} \ \textsc{swap-fail}$$

Figure 4.3.: Modified abstract sequential transition rules for λ_ϵ.

Rule ATOMIC-EVALUATION stipulates that the effects of the atomic evaluation relation are transferred to the sequential evaluation relation. Next, in rules APPL-BODY and LETREC-BODY, an address in the store is modified. Hence, write effects need to be generated. It may also be the case that multiple effects need to be generated by a single rule. For example, in rule ATOM, the effects resulting from the atomic evaluation as well as a write effect must be generated; we have chosen to explicitly denote all effects for completeness. Since the (abstract) sequential transition rules only operate on a single thread, no create or dereferencing effects are generated.

4.1.4. Abstract Concurrent Transition Relation

The abstract concurrent transition relation for λ_ϵ is depicted in Figure 4.3. Again, for each rule, the effects generated by the execution of that rule are written underneath the concurrent transition arrow and rules that do not generate effects have been elided.

λ_ϵ

$$\frac{\hat{\varsigma} \in \hat{\pi}(\hat{p}) \qquad \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \underset{\widehat{eff}}{\widehat{\hookrightarrow}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \underset{\widehat{eff}}{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \hat{\varsigma}'], \hat{\sigma}', \widehat{\Xi}'} \ \textsc{sequential-step}$$

$$\frac{\langle \mathbf{ev}(\texttt{(future } e\texttt{)}, \hat{\rho}), \widehat{k}\rangle \in \hat{\pi}(\hat{p}) \qquad \hat{\varsigma} = \langle \mathbf{ev}(e, \hat{\rho}), \widehat{k}_0\rangle \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma}, \hat{\pi})}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \underset{\mathbf{c}(\hat{p}')}{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\mathbf{fut}(\hat{p}')), \widehat{k}\rangle, \hat{p}' \mapsto \hat{\varsigma}], \hat{\sigma}, \widehat{\Xi}} \ \textsc{future}$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-call } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle \in \hat{\pi}(\hat{p}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae \underset{\widehat{eff}}{\widehat{\Downarrow}} \hat{v}_f \qquad \hat{v}_f = \mathbf{clo}(lam, \hat{\rho}_f) \\ lam = (\lambda\,(x)\ \ e) \qquad \hat{\varsigma} = \langle \mathbf{ev}(e, \hat{\rho}_f), \widehat{k}_0\rangle \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma}, \hat{\pi})\end{array}}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \underset{\substack{\widehat{eff}\\ \mathbf{c}(\hat{p}')}}{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\mathbf{fut}(\hat{p}')), \widehat{k}\rangle, \hat{p}' \mapsto \hat{\varsigma}], \hat{\sigma}, \widehat{\Xi}} \ \textsc{future-call}$$

$\boxed{\lambda_\epsilon}$

$$\frac{\langle \mathbf{ev}(\texttt{(deref } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle \in \hat{\pi}(\hat{p}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae \underset{\widehat{eff}_f}{\widehat{\Downarrow}} \mathbf{fut}(\hat{p}') \qquad \langle \mathbf{val}(\hat{v}), \widehat{k}_0\rangle \in \hat{\pi}(\hat{p}')}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \underset{\substack{\widehat{eff}_f \\ \mathbf{d}(\hat{p}')}}{\widehat{\rightsquigarrow}_{\hat{p}}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\hat{v}), \widehat{k}\rangle], \hat{\sigma}, \widehat{\Xi}} \textsc{deref}$$

Figure 4.4.: Modified abstract concurrent transition rules for λ_ϵ.

By means of rule SEQUENTIAL-STEP, the effects generated by the abstract sequential transition relation are carried over to the abstract concurrent transition relation. As opposed to the sequential transition relation of λ_ϵ, the annotated concurrent transition relation can also generate creation and dereferencing effects. The reason for this is obvious: creation and dereferencing effects are generated when futures are respectively created or dereferenced, and this is taken care of by the (abstract) concurrent transition relation.

4.2. A Non-Incremental Thread-Modular Analysis Algorithm for λ_α

Based on the formalisation of effects in Section 4.1, we can now describe an algorithm for the thread-modular abstract interpretation of λ_α (λ_ϵ). This algorithm is adapted from the one described by Stiévenart (2018, Chapter 6) and serves as the basis for our own incremental algorithm, described in Section 4.3.

The analysis algorithm, shown in Algorithm 1, consists out of two alternating phases, an **intra-process analysis phase** and an **inter-process analysis phase**. The former analyses a single abstract process in isolation and accumulates the effects generated by the transition function; the latter uses the effects inferred by the intra-process analysis phase to decide on the threads that need to be (re-)analysed next. Both analysis phases are fixed-point computations. We now discuss this algorithm in detail and prove termination. Note that in Algorithm 1, we have omitted the abstraction hats for brevity and readability. Also, since the algorithm is modular, there is a slight divergence between the algorithm and λ_ϵ's formalisation. After all, the concurrent transition function for λ_ϵ formalises a non-modular analysis. In a modular analysis, there is no need for a thread map that is continuously updated since threads are analysed in isolation. Therefore, π is only used to relate abstract thread identifiers to abstract initial states and is omitted as an argument to PALLOC. As a consequence, we can use the sequential injection function INJECT. We refer to Stiévenart (2018, Chapter 6) for the formalisation of the algorithm. Henceforth, we may refer to the modular analysis described in Algorithm 1 as ModAtom.

4.2.1. State Injection

To analyse an expression, the expression is first injected into an initial state (line 45, see also Section 3.1.4). Then, a thread identifier for the main thread is allocated and the initial state of the main thread is added to π, a data structure mapping thread identifiers to sets of initial states. (Note that for every thread identifier, multiple initial states may be present in π since every thread identifier may be allocated multiple times.) Then, the inter-process analysis is started with a work list containing the thread identifier just allocated.

Algorithm 1 Non-incremental thread-modular static program analysis

```
procedure ANALYSE($e$: $Exp$)
    $\pi \leftarrow []$                                    ▷ Maps thread identifiers to sets of initial states.
    $returnValues \leftarrow [\cdots \mapsto \bot]$        ▷ Maps thread identifiers to return values.
    $graphs \leftarrow []$                                 ▷ Holds the produced state graphs.
    $effects \leftarrow []$                                ▷ Collects the generated effects.

    procedure INTRA($p : PID, \sigma : Store, \Xi : KStore$)        ▷ Intra-process analysis.
        $graphs.set(p, [])$
        $work \leftarrow List[\pi.get(p)]$                 ▷ Start the analysis from the initial thread state(s).
        $visited \leftarrow []$
        $effects \leftarrow []$
        $result \leftarrow \bot$                           ▷ The abstract return value is part of a lattice.
        while $work$ $not$ $empty$ do
            $\varsigma \leftarrow work.serve()$
            if $visited.contains(\varsigma)$ then
                continue
            end if
            $(successors, effs, \sigma', \Xi', res) \leftarrow$ TRANSITION$(\varsigma, \sigma, \Xi)$   ▷ Apply the transition function.
            if $\sigma$ $equals$ $\sigma' \wedge \Xi$ $equals$ $\Xi'$ then
                $visited.add(\varsigma)$
            else
                $visited \leftarrow []$                    ▷ Upon a store change, the visited set must be emptied.
            end if
            $work.add(successors)$
            $graphs(p).removeEdges(\varsigma).addEdges(\varsigma, successors)$   ▷ Update outgoing edges of $\varsigma$.
            $effects.add(effs)$
            $result \leftarrow result \sqcup res$
            $(\sigma, \Xi) \leftarrow (\sigma', \Xi')$
        end while
        return $(effects, \sigma, \Xi, result)$     ▷ Returns generated effects, updated stores and return value.
    end procedure

    procedure INTER($work$: $List[PID]$, $\sigma$: $Store, \Xi : KStore$)   ▷ Inter-process analysis.
        while $work$ $not$ $empty$ do
            $p \leftarrow work.serve()$
            $(effs, \sigma', \Xi', ret) \leftarrow$ INTRA$(p, \sigma, \Xi)$   ▷ Perform an intra-process analysis for this thread.
            $todo \leftarrow$ PROCESSEFFECTS$(p, effects, effs, \sigma, \sigma', ret)$ ▷ Compute the threads to analyse next.
            $work.add(todo)$
            $effects.add(p, effs)$
            $returnValues.set(p, ret)$
            $(\sigma, \Xi) \leftarrow (\sigma', \Xi')$
        end while
    end procedure

    $initial \leftarrow$ INJECT$(e)$                      ▷ Inject the program into an initial state.
    $p \leftarrow$ PALLOC$(initial)$
    $\pi.add(p, initial)$
    INTER(List$[p]$, [], $[k_0 \mapsto$ halt$]$)           ▷ Start the inter-process analysis.
    return $graphs$
end procedure
```

4.2.2. Inter-Process Analysis Phase

The inter-process analysis contains in its work list (`work`) a list of abstract thread identifiers of the threads that need to be analysed. For every abstract thread identifier in the list, the intra-process analysis is run, which returns a set of generated effects, an updated store, an updated continuation store and the abstract return value of the analysed thread. Based on these effects and on the new store and return value, the procedure processEffects determines which threads need to be analysed next (line 37). Then, the thread identifiers collected in `todo` are added to the work list `work` (line 38), the generated effects `effs` are added to `effects` (line 39), the thread's return value is stored (line 40) and the stores are updated (line 41). When the generated effects `effs` are added to `effects` (line 39), they are tupled together with the thread identifier p; this allows to correlate effects to the thread that generated them, which is used in the function processEffects. Finally, if the work list is not empty, a new iteration starts; the work list will only become empty when a fixed point is reached and no threads need to be analysed anymore.

The procedure processEffects is outlined in Algorithm 2. Note that we assume that processEffects has access to the local scope of the procedure analysis of Algorithm 1. processEffects uses the generated effects as follows:

- processEffects first scans the newly generated effects `effs` for creation effects and adds the thread identifiers of the created threads to `todo`; the corresponding initial thread states are added to π. However, this only happens if an identical initial state was not yet present in π for the given thread identifier, since in the other case, the thread already exists (lines 3 to 6).
- processEffects checks the addresses written to by p and then verifies for which of these addresses the value stored in the updated store differs from the value in the original store since not every write leads to a changed abstract value stored at the respective address. For each of the addresses where the stored value is different, processEffects uses all read and write effects generated so far by other abstract threads to determine possible read-write and write-write conflicts. All abstract threads reading or writing an address the current thread has modified must be reanalysed and are added to `todo`. The procedure `getPID` is used, which retrieves the abstract thread identifier that was tupled with the effect; this thread identifier denotes the abstract thread that generated the effect, and hence indicates which abstract thread should be reanalysed (lines 7 to 14).
- When the return value `ret` of the thread p has changed, `processEffects` scans all effects for dereferencing effects with thread identifier `p` and adds all threads that have generated such an effect to `todo`; since these threads depend on that return value, they must be reanalysed (lines 15 to 21).

Finally, processEffects returns the abstract thread identifiers of the abstract threads that must be reanalysed, which is the union of `newThreads`, `conflictedThreads` and `returnConflicts` (line 22).

4.2.3. Intra-Process Analysis Phase

The intra-process analysis analyses the thread(s) related to an abstract thread identifier in isolation. While doing so, it builds the abstract state graph for the corresponding thread identifier. To this end, first, the initial states related to the thread identifier are retrieved and added to the work list. Like the inter-process analysis phase, the intra-process analysis phase is a fixed-point computation. To avoid duplication of work and ensure termination, a visited set `visited` is

Algorithm 2 Effect processing for the non-incremental program analysis algorithm

```
procedure PROCESSEFFECTS($p : PID, effs : Effects, newEffs : Effects, \sigma : Store, \sigma' : Store, ret : Val$)
    $allEffs \leftarrow effs + newEffs$
    $newThreads \leftarrow$                                      ▷ Thread identifiers of new threads.
        $newEffs.creationEffects$                              ▷ Get the creation effects...
        $.filter(exists, \pi)$                                 ▷ ... not corresponding to existing threads....
        $.map(getPID)$                                         ▷ ... and return the corresponding thread identifiers.
    $addressesWritten \leftarrow newEffs.writeEffects.map(getAddress)$
    $addressesChanged \leftarrow$                              ▷ Addresses whose values were modified.
        $addressesWritten.filter(a \rightarrow \sigma(a) \neq \sigma'(a))$
    $conflictedThreads \leftarrow$
        $(allEffs.readEffects + allEffs.writeEffects)$         ▷ Take all read and write effects...
        $.filter(e \rightarrow e.getPID \neq p)$               ▷ ... generated by other threads...
        $.filter(e \rightarrow e.getAddress \in addressesChanged)$   ▷ ... that cause conflicts...
        $.map(getPID)$                                         ▷ ... and return the affected threads.
    if $ret \neq returnValues(p)$ then
        $returnConflicts \leftarrow$
            $effs.dereferencingEffects.filter(e \rightarrow e.getPID = p)$
            $.map(getPID)$                                     ▷ Threads that read $p$'s return value.
    else
        $returnConflicts \leftarrow []$
    end if
    return $newThreads + conflictedThreads + returnConflicts$
end procedure
```

instantiated (line 10). Initially, this set is empty. The intra-process analysis also keeps track of the return value of the thread. Due to nondeterminism in the analysis, it is possible that multiple return values are found. Therefore, the accumulator for the return value of the thread under analysis is initialised to bottom ($\perp$, line 12) and every time a return value is encountered during the analysis, this value is joined together with the accumulator (line 27).

The fixed-point computation successively pops a state of the work list and verifies that it is not in the visited set; if it is, the state is ignored. Otherwise, the transition function is applied to the state, i.e., the state is stepped, using the current store σ and the current continuation store Ξ. The result of the transition function is a set of successor states (`successors`), a set of generated effects (`effs`), an updated store (σ') and an updated continuation store (Ξ', line 18). Also, a return value (`res`) may be generated when the state is a final state. If not, `res` is set to bottom, which is the neutral element for join.

Since the store σ and continuation store Ξ are not contained in the states themselves, the visited set needs to be cleared whenever one of the stores changes (lines 19 to 23). To see why, consider the transition function TRANSITION. This function steps a given state under a given store and continuation store. Hence, if one of the stores changes, this may have an influence on the result of TRANSITION. This is problematic since the visited set contains the states of which we ought to have computed the result already. Therefore, since the result may be different now, the visited set must be cleared. Hence, if a state that has been removed from the visited set is added to the work list again, it will be reanalysed.

Finally, the work list is updated, the effects are stored and the result and stores are updated. In addition, the state graph of the analysed thread is extended with edges from the transitioned state ς to its successor state(s) (line 25). However, before doing so, the edges previously going out ς are removed since they represent computations based on outdated information. Also, since in the intra-analysis phase, a thread is analysed from scratch, the thread's state graph is erased at the beginning of the intra-process analysis (line 8).

4.2.4. Termination

The algorithm for non-incremental modular analysis relies on two alternating phases that are expressed by means of a fixed-point computation. We will now show that our formulation of the computation, as expressed by Algorithm 1, terminates. We refer to Appendix B.1 for the complete proofs of following lemmas and of the following theorem. For clarity, the proofs in this appendix are numbered the same as the corresponding lemmas and theorems in this text.

Lemma 1. *Consider an abstract value store $\hat{\sigma}$. After a finite number of updates of $\hat{\sigma}$, a fixed-point is reached.*

Proof concept. We prove Lemma 1 by proving that updating a specific address $\hat{a}$ in the store $\hat{\sigma}$ only leads to a change in the stored value a finite number of times. □

Lemma 1 is related to an important concept in static analysis called *store monotonicity*, which is required for a sound analysis. The abstract store $\hat{\sigma}$ maps addresses to sets of abstract values. Upon an update to the store, only elements can be added to such sets. Consider now the Hasse diagram of the set lattice of $\mathcal{P}(\widehat{Val})$, then every update to the address $\hat{a}$ in $\hat{\sigma}$ will cause the set related to $\hat{a}$ in $\hat{\sigma}$ to be least as close to $\top$ as the set originally residing at that address in the store. An updated set will never be closer to $\bot$ than the set that was not updated. Therefore, it is said that updates to the store are monotonous. Intuitively, this means that information cannot get lost, which is important if we want to obtain a sound analysis.

Lemma 2. *Consider an abstract continuation store $\widehat{\Xi}$. After a finite number of updates of $\widehat{\Xi}$, a fixed-point is reached.*

Proof concept. Similar to the proof of Lemma 1. □

Lemma 3. *The intra-process analysis of a process in Algorithm 1 terminates.*

Proof concept. We show that the intra-process analysis terminates by showing that at a given moment, all states that can be generated must be in the visited set, which is the condition for the algorithm to terminate. □

Theorem 1. *The process-modular analysis of a program e in Algorithm 1 terminates.*

Proof concept. We show that the inter-process analysis terminates by proving that the intra-process analysis must only be called a finite number of times. □

4.3. Incrementalising the Thread-Modular Analysis Algorithm for λ_α

In the previous section, we presented an algorithm to perform a thread-modular analysis of λ_ϵ. This algorithm starts by performing an intra-process analysis of the main thread. Every time an intra-process analysis is run, the algorithm collects the effects generated by that analysis to decide on the threads for which the intra-process analysis must be (re-)run. When a fixed-point is reached, the alternation of these analysis phases stops and the graph representing the abstract collecting semantics has been computed.

The presented algorithm may not be entirely efficient however. Every time a thread interference is detected, the affected thread is entirely reanalysed from its initial state. Yet, not all states in the computed state graph may be affected by this interference and performing an entire reanalysis of that thread may result in a loss of precision. To understand why not all states may be affected, consider the program in Listing 4.1. This program computes the 6th Fibonacci number using a parallel computation. ModAtom, the modular analysis described in Algorithm 1, analyses this program according to following rough outline:

1. The main thread is analysed by the intra-process analysis. First, the binding of the **letrec** is analysed, resulting in a closure that is bound, via the store, to the variable `fibonacci`. Then, the call `(fibonacci 6)` is analysed. When the body of the function is analysed, the transition function generates a creation effect due to the call to `future` in line 4. Later, the return value of this future is read in line 7, which causes a dereferencing effect to be generated by the transition function. Since the newly created thread has not yet been analysed, the return value found for that thread is $\perp$ and the analysis of the main thread continues with this value.
2. The inter-process analysis processes the effect generated by the intra-process analysis and finds a creation effect. This causes the intra-process analysis to be called for the analysis of the new child thread in which the call `(fibonacci (- n 2))` is analysed.
3. When the intra-process analysis is finished with the evaluation of the child thread, it finds that its return value no longer is bottom and hence has changed. It scans the effects generated so far and finds that the main thread reads the thread's return value. Since this value has changed, the main thread must be reanalysed since it depends on the new value.

In summary, the main thread is analysed twice and the child thread is analysed once. Note that there is only one child thread that needs to be analysed, which is a consequence of the definition of $\widehat{palloc}$ (see Section 3.2.3): the analysis cannot differentiate between the different child threads and will consider them to be one and the same. Also, a loss of precision may arise due to the fact that ModAtom does not keep track of the order in which the effects are generated by the abstract threads: the collected effects are passed to processEffects in an unordered set.

```
1  (letrec ((fibonacci (lambda (n)
2                         (if (<= n 1)
3                             n
4                             (letrec ((fut (future (fibonacci (- n 2)))))
5                               (letrec ((v1 (fibonacci (- n 1))))
6                                 (+ v1
7                                    (deref fut))))))))
8    (fibonacci 6))
```

Listing 4.1: Parallel computation of the n^{th} Fibonacci number.

Due to the change in return value of the child thread, in step 3, the main thread is reanalysed. However, when a thread is reanalysed, it is reanalysed in its entirety. Yet, the change in return value does not influence the entire computation of the main thread; only when the return value of the child thread is read (line 7), the remaining part of the analysis may be influenced. Hence, up to the point where the return value of the child thread is read, there is no influence on the analysis result. The same principle is true for read-write and write-write conflicts.

In the remaining part of this section, we first introduce a new algorithm that incrementalises the computation of the intra-process analysis so that, upon reanalysis of a thread, previous analysis results can be reused, that is, we aim to reuse as much as possible of the thread's previously computed abstract state graph. This way, we avoid the needless repetition of work and aim at decreasing the time needed by the analysis. After having introduced this new algorithm, we present some optimisations that may further reduce the time needed by the analysis.

4.3.1. General Approach

The goal of this incrementalisation is to alter the intra-process and the inter-process analyses so that the intra-process analysis need not start from a thread's initial state upon reanalysis anymore. Instead, the intra-process analysis should restart from the states whose evaluation is influenced by another abstract thread. Since the intra-process analysis must not start its analysis from the abstract thread's initial state, it avoids the recomputation of non-affected states. However, the intra-process analysis may still need to recompute some of these states when there is a back edge present in the abstract state graph. The reason for this is that the analysis performs a fixed-point computation and that a thread's reanalysis is started with an empty visited set.

To see how ModAtom can be altered, we first look at how the algorithm decides which threads to reanalyse. After the execution of an intra-process analysis, the effects generated by that analysis are processed by processEffects and added to `effects`, the collection of all generated effects (Algorithm 1, line 39). However, `effects` does not only keep track of the generated effects but also relates these abstract effects to the abstract thread identifier of the abstract thread that generated them, that is, it stores effects $\widehat{eff}$ as tuples $\langle\widehat{eff}, \hat{p}\rangle$, where $\hat{p}$ is the abstract thread identifier of the abstract thread that generated the abstract effect. This way, whenever an effect of interest is found, it can be related to the thread that generated the effect and hence, it is known which thread is to be reanalysed. For example, if the return value of an abstract thread $\hat{p}$ changes, processEffects will look for dereferencing effects containing $\hat{p}$, and the abstract thread identifier tupled with this effect denotes the abstract thread to be reanalysed.

To incrementalise the intra-process analysis, the reanalysis of a thread must start from the state whose evaluation was influenced. To do so, we alter this tracking behaviour so that effects are not just related to thread identifiers anymore, but also to abstract states, that is, it now stores effects as triples $\langle\widehat{eff}, \hat{p}, \hat{\varsigma}\rangle$, where $\hat{p}$ is the thread identifier of the thread that generated the effect and $\hat{\varsigma}$ is the state whose evaluation is affected. As a result, whenever processEffects scans the set of generated effects and finds an effect of interest, it can now determine from which state the analysis of a given thread must be restarted.

Having presented our strategy to incrementalise the intra-process analysis of ModAtom, we can now discuss our incremental algorithm, shown in Algorithm 3; we henceforth may refer to this analysis as IncAtom. The algorithm is structured analogously to Algorithm 1; the most important changes are indicated in red. We now discuss IncAtom, focussing on the aspects of the analysis algorithm that are specific to the incremental algorithm.

Algorithm 3 Incremental thread-modular static program analysis

```
    procedure ANALYSE($e$: $Exp$)
        $\pi \leftarrow []$
        $returnValues \leftarrow [\cdots \mapsto \bot]$
        $graphs \leftarrow []$
5:      $effects \leftarrow []$

        procedure INTRA($\varsigma : \Sigma$, $p : PID$, $\sigma : Store$, $\Xi : KStore$)        ▷ Intra-process analysis.
            $graphs.set(p, [])$
            $work \leftarrow List[\varsigma]$        ▷ Start the analysis from the given state.
10:         $visited \leftarrow []$
            $effects \leftarrow []$
            $result \leftarrow \bot$
            while $work$ not empty do
                $\varsigma \leftarrow work.serve()$
15:             if $visited.contains(\varsigma)$ then
                    continue
                end if
                $(successors, effs, \sigma', \Xi', res) \leftarrow$ TRANSITION$(\varsigma, \sigma, \Xi)$
                if $\sigma$ equals $\sigma' \wedge \Xi$ equals $\Xi'$ then
20:                 $visited.add(\varsigma)$
                else
                    $visited \leftarrow []$
                end if
                $work.add(successors)$
25:             $graphs(p).removeEdges(\varsigma).addEdges(\varsigma, successors)$
                $effects.add(p, \varsigma, effs)$        ▷ Effects are now related to states.
                $result \leftarrow result \sqcup res$
                $(\sigma, \Xi) \leftarrow (\sigma', \Xi')$
            end while
30:         return ($effects$, $\sigma$, $\Xi$, $result$)
        end procedure

        procedure INTER($work$: $List[\langle PID, \Sigma \rangle]$, $\sigma$: $Store$, $\Xi : KStore$)        ▷ Inter-process analysis.
            while $work$ not empty do
35:             $\langle p, \varsigma \rangle \leftarrow work.serve()$
                $(effs, \sigma', \Xi', ret) \leftarrow$ INTRA$(p, \varsigma, \sigma, \Xi)$
                $todo \leftarrow$ PROCESSEFFECTS$(p, effects, effs, \sigma, \sigma', ret)$
                $work.add(todo)$
                $effects.add(effs)$
40:             $returnValues.set(p, ret \sqcup returnValues.get(p))$        ▷ Join the new and old return values.
                $(\sigma, \Xi) \leftarrow (\sigma', \Xi')$
            end while
        end procedure

45:     $initial \leftarrow$ INJECT$(e)$        ▷ Inject the program into an initial state.
        $p \leftarrow$ PALLOC$(initial)$
        $\pi.add(p, initial)$
        INTER(List[$\langle p, initial \rangle$], [], [$k_0 \mapsto$ halt])        ▷ Start the inter-process analysis.
        return $graphs$
50: end procedure
```

4.3.2. Inter-Process Analysis Phase

The work list of the inter-process analysis now contains tuples of abstract thread identifiers and abstract states. The state indicates where the intra-process analysis of the given process should start; the corresponding thread identifier identifies the thread the state belongs to. For every tuple in the list, the intra-process analysis is run; this returns a set of generated effects, an updated store, an updated continuation store and the abstract return value of the analysed thread.

Since the intra-process analysis may only have reanalysed part of the thread's state graph, it may not have reanalysed all of the thread's final states. For this reason, the new and old return values need to be joined, since the value returned by the intra-process analysis only over-approximates the return values that were found during reanalysis. Otherwise the result may not be sound (line 40). Another solution to this problem would be to set the initial result value of a thread in the intra-process analysis to the value stored in `returnValues`, that is, to modify line 12. Since a thread can logically never rely on its own return value, this is a mere implementation issue which does not influence the result of the analysis. Hence, both implementations will produce the same result.

In the non-incremental algorithm, the inter-process analysis related every effect returned by the intra-process analysis together with its thread identifier (line 39). However, in the incremental algorithm, every effect must be related to an abstract thread identifier and an abstract state. Therefore, this now happens in the intra-process analysis; the effects returned by the intra-process analysis are already tupled together with an abstract thread identifier and an abstract state. However, the inter-process analysis still has to determine for which states the intra-process analysis must be started by means of PROCESSEFFECTS; for brevity, we do not present an updated version of PROCESSEFFECTS since only minor modifications are required.

4.3.3. Intra-Process Analysis Phase

The intra-process analysis does not require a lot of change. The analysis now takes an extra parameter as argument, which is the state the analysis has to be started from. MODATOM always starts from the thread's initial state, but this is what we now try to avoid.

Except for the start of the analysis, it works analogously to its non-incremental counterpart. However, the intra-process analysis must now also relate the generate effects to the state which transition caused the effect. After all, such an effect indicates a dependency related to the evaluation of that state.

4.3.4. Termination

Now we have obtained an incremental version of Algorithm 1, we show that this new algorithm still terminates. Lemmas 1 and 2 do not require modification, as they are only related to the behaviour of the abstract value and continuation stores. However, Lemma 3 and Theorem 1 are specific to the non-incremental analysis. Therefore, we must again prove the termination of the intra-process and inter-process analysis phases.

Lemma 4. *The intra-process analysis of an abstract thread in Algorithm 3 terminates.*

Proof. Analogous to the proof of Lemma 3, but the intra-process analysis now starts from a given abstract state $\hat{\varsigma}$, instead of from the abstract thread's initial state. However, this does not impact the termination argument given in the proof of Lemma 3. □

Theorem 2. *The incremental process-modular analysis of a program e in Algorithm 3 terminates.*

Proof. The proof of this theorem is analogous to the proof of Theorem 1; the fact that the intra-process analysis now is started from a given abstract state does not invalidate the termination argument. □

4.3.5. Soundness

In Section 2.1, we imposed soundness as a requirement for the analyses presented in this dissertation. Stiévenart (2018) has already shown soundness for ModAtom, the analysis on which IncAtom is based. We do not formally proof soundness of IncAtom, as we did for its termination, but refer to Chapter 6 for an empirical evaluation of the soundness of IncAtom instead.

4.4. Optimisations

IncAtom, described in Algorithm 3, incrementalises the intra-process analysis phase of the process-modular analysis algorithm. In this section, we will discuss some changes to this algorithm that may further improve its performance, that is, lower the analysis time. In addition, we discuss the applicability of these optimisations to ModAtom.

4.4.1. Visited Set Caching

Whenever an intra-process analysis of an abstract thread $\hat{p}$ is started, this is done so with an empty visited set, both in ModAtom (Algorithm 1) as well as in IncAtom (Algorithm 3). For ModAtom, the visited set is always discarded because the entire thread must be reanalysed. On the contrary, for Algorithm 3, discarding the visited set may not be the best option. The reason for this is that, during reanalysis of $\hat{p}$, states may be generated that were already computed during the previous execution of the intra-process analysis of $\hat{p}$. Hence, when the reanalysis is started with an empty visited set, previously computed parts of $\hat{p}$'s abstract state graph may need to be computed again, resulting in a duplication of work that may needlessly slow down the static analyser.

To remedy this issue and to avoid duplication of work by IncAtom, it is possible to cache the visited set, which fits our goal of reusing as much of the previously calculated result as possible. Hence, whenever the intra-analysis of an abstract thread is restarted, the cached visited set can be restored. This optimisation is only applicable to IncAtom; it cannot be applied to ModAtom as this algorithm always reanalyses a thread in full and hence restarts the analysis of a thread with an empty visited set.

There are several possible points in the execution of the intra-process analysis where the visited set may be cached: the visited set may be cached whenever an effect is generated by $\hat{p}$ (it becomes then related to that effect), or that the end of the intra-process analysis, when all effects have been generated (it becomes then related to all effects). We think however that both strategies will result in a similar performance, since we have identified one important restriction to caching the visited set: whenever the store or continuation store is modified, the visited set must be invalidated or ignored, that is, a visited set may only be reused when there has been no store change between the moment it was saved and the moment it was restored. The reason for this was already explained in Section 4.2.3, where we discussed the clearance of the visited set: since a change in the store may impact the result of TRANSITION, we can no longer be sure that we obtained the correct successor states of the states in the visited set.

The condition that the store must remain unchanged may limit the number of times a cached visited set can actually be restored and hence also the gains that can be realised by the visited set caching technique. After all, all reanalyses triggered by a write effect cannot benefit from the cached visited set. Hence, the performance gain obtained by the caching of the set may be small; we refer to Chapter 6 for an evaluation of this optimisation.

4.4.2. Intra-Process Analysis Abortion

In the beginning of Section 4.3, we explained how the non-incremental thread-modular analysis algorithm analyses a simple parallel program that computes the n^{th} Fibonacci number. In a first step, the abstract main thread is analysed. During the abstract interpretation of this thread, the return value of the abstract child thread that was spawned is read. However, since that thread had not been analysed yet, $\perp$ is used as a placeholder for the child thread's return value in the remainder of the analysis of the main thread.

The use of bottom as the placeholder of an abstract thread's abstract return value allows the analysis of other abstract threads to continue even when they reference an unanalysed abstract thread. When an abstract thread has been analysed, its return value is updated and will differ from bottom; only when an abstract thread has not been analysed yet, its return value will equal bottom. At the point where an abstract return value is read, a dereferencing effect is generated; this effect indicates a dependency between the two abstract threads and causes the dereferencing thread to be reanalysed when the return value of the dereferenced thread changes. In our incremental algorithm, the thread will be reanalysed from the point where the modified return value is read.

The key to seeing how this analysis scheme can be optimised is to note that there is no point in continuing the analysis of an abstract thread when it has read $\perp$ as the abstract return value of another thread. After all, this implies that the dereferenced thread has not been analysed yet. As a result, the inter-process analysis will analyse the dereferenced thread and then restart the analysis of the dereferencing thread from the point where it reads the new abstract return value. Hence, since the abstract thread will have to be reanalysed from that point on in any case, it makes no sense to complete the first analysis using $\perp$ as the placeholder for the missing abstract return value. Therefore, when bottom is read as the return value of an abstract thread, the intra-process analysis can be aborted. However, it is important for the dereferencing effect to be generated, since otherwise, the dereferencing thread will not be reanalysed anymore when the return value of the dereferenced thread has been computed.

As both MODATOM and INCATOM have a similar structure, both algorithms may encounter situations where they read $\perp$ as the return value of an abstract thread, as was explained above. Therefore, this optimisation is applicable to both algorithms.

4.5. General Considerations

For completeness, we now briefly discuss some general remarks regarding the two presented algorithms.

4.5.1. Filtering the Abstract State Graph

During the intra-process analysis, an abstract state graph is built for each abstract thread. This state graph is the thread's abstract collecting semantics (see Section 2.2.2).

During the fixed-point computation performed by the intra-process analysis, an abstract state may need to be analysed multiple times, to ensure soundness. Hence, every time the state is analysed, the transition function produces a set of successor states of that state. In Algorithm 1, this happens on line 18:

$(\mathit{successors}, \mathit{effs}, \sigma', \Xi', \mathit{res}) \leftarrow \textsc{transition}(\varsigma, \sigma, \Xi)$

In the abstract state graph, the abstract state that was analysed must now become connected to its successor states. However, the graph may already contain outgoing edges for that abstract state, resulting from a prior analysis. Since these edges may represent outdated information, they are first removed before edges connecting the abstract state to its successor states are added. In Algorithm 1, this happens on line 25 as follows:

$\mathit{graphs}(p).\mathit{removeEdges}(\varsigma).\mathit{addEdges}(\varsigma, \mathit{successors})$

However, removing edges going out of a state may result in states that are present in the abstract state graph of an abstract thread, but that are not reachable from that abstract thread's initial state. This is caused by the fact that, upon reanalysis, different successor states may be generated when more information is available or precision is lost; the set of new successor states over-approximates the set of successor states generated earlier.

Consider for example the abstract state graph of an abstract thread p_A in Figure 4.5 (note that we have omitted the abstraction hats in the figure) and suppose the abstract thread p_A is analysed by our incremental thread-modular analysis. In state s_{1a}, the thread reads the abstract return value of abstract thread, p_B. Since this thread has not been evaluated yet, bottom is read and the analysis of p_A is continued along the blue path. After the analysis of the abstract thread p_B, the incremental algorithm restarts the analysis of p_A from state s_{1a}, but now reads `true` as the p_B's return value; the analysis now continues along the orange path. However, before an edge is added from state s_{1a} to state s_{2b}, first all previously outgoing edges from state s_{1a} are cut. In this case, states s_{2a} and s_{3a} become unreachable from state s_0. Unfortunately, it is non-trivial to just remove these states from the graph, since, in general, there may still be another path from s_0 to any of these states.

To avoid the analysis returning graphs in which such unreachable states occur, a filtering pass is needed after the inter-process analysis terminates. For clarity and conciseness, we have omitted this pass in our definition of Algorithm 1 and Algorithm 3. For each abstract thread, the filtering pass traverses the thread's abstract state graph and stores and copies all states and edges encountered during the traversal. As a result, states that are not reachable from the abstract thread's initial state are removed from the thread's abstract state graph.

Note that although we have given an example using the incremental thread-modular algorithm,

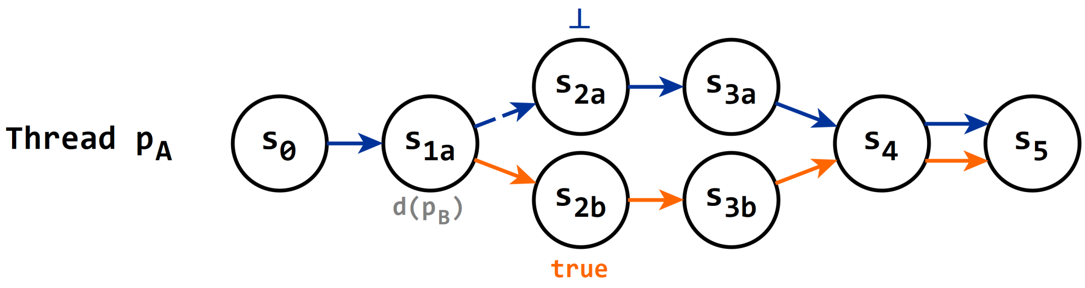


Figure 4.5.: Example of unreachable states in an abstract thread's abstract state graph.

the problem also arises for the non-incremental thread-modular algorithm, for example when the graph contains a back-edge. After all, all edges going out from a state that is recomputed must be invalidated since they represent computations based on outdated information.

4.5.2. Thread-Local Continuation Stores

For simplicity and conciseness, we have formalised λ_α (λ_ϵ), as well as our algorithms for the (non-)incremental thread-modular analysis of this language, using a value store and continuation store that were shared among all threads, that is, the value store and continuation store are not part of the states ς (for λ_0), nor are they part of the thread maps π (for λ_ϕ and λ_α). Not incorporating the stores into the states themselves, but keeping them separate, is called *global-store widening* and is used to reduce the worst-case time complexity of the analysis at the cost of a lower precision (Shivers, 1991; Van Horn & Might, 2012; Stiévenart, 2018). The loss in precision arises from the fact that a store is no longer correlated to a specific state, but over-approximates all information stored so far in the analysis.

A continuation store shared among all abstract threads may have a negative impact on precision as there is no way to distinguish continuation frames stored by one abstract thread from the ones that were stored by another. Hence, when such an abstract thread looks up an abstract continuation address in the continuation stores, it may also retrieve continuation frames that it did not store itself. One solution to this issue would be to incorporate the continuation stores into the states themselves, that is, to only apply global-store widening to the value store so that $\Sigma = \mathit{Control} \times \mathit{KAddr} \times \mathit{KStore}$. This would also benefit our algorithms for the thread-modular analysis: since the continuation store now is incorporated into the states themselves, there is no need anymore to clear the visited set in the intra-process analysis phase when such a continuation store is modified.

However, due to the increase in precision, the use of thread-local continuation stores may have a profoundly negative impact on the analysis time of an abstract interpreter. For this reason, we have not used thread-local continuation stores in this dissertation.

4.6. Example: Analysis of a Simple Concurrent Program

In the previous sections, we have discussed ModAtom and IncAtom, two algorithms for a thread-modular analysis. In this section, we apply these analyses to a simple concurrent program and compare their output and behaviour. For completeness, we also analyse this program with our non-modular analysis algorithm.

Consider the extremely simple λ_α program in Listing 4.2. On line 1, a future is created with a body containing the value `#t` (true). On line 2, this future is dereferenced by the main thread, creating a dependency between it and the future `f`.

```
1  (define f (future #t))
2  (deref f)
```

Listing 4.2: A simple concurrent program written in λ_α.

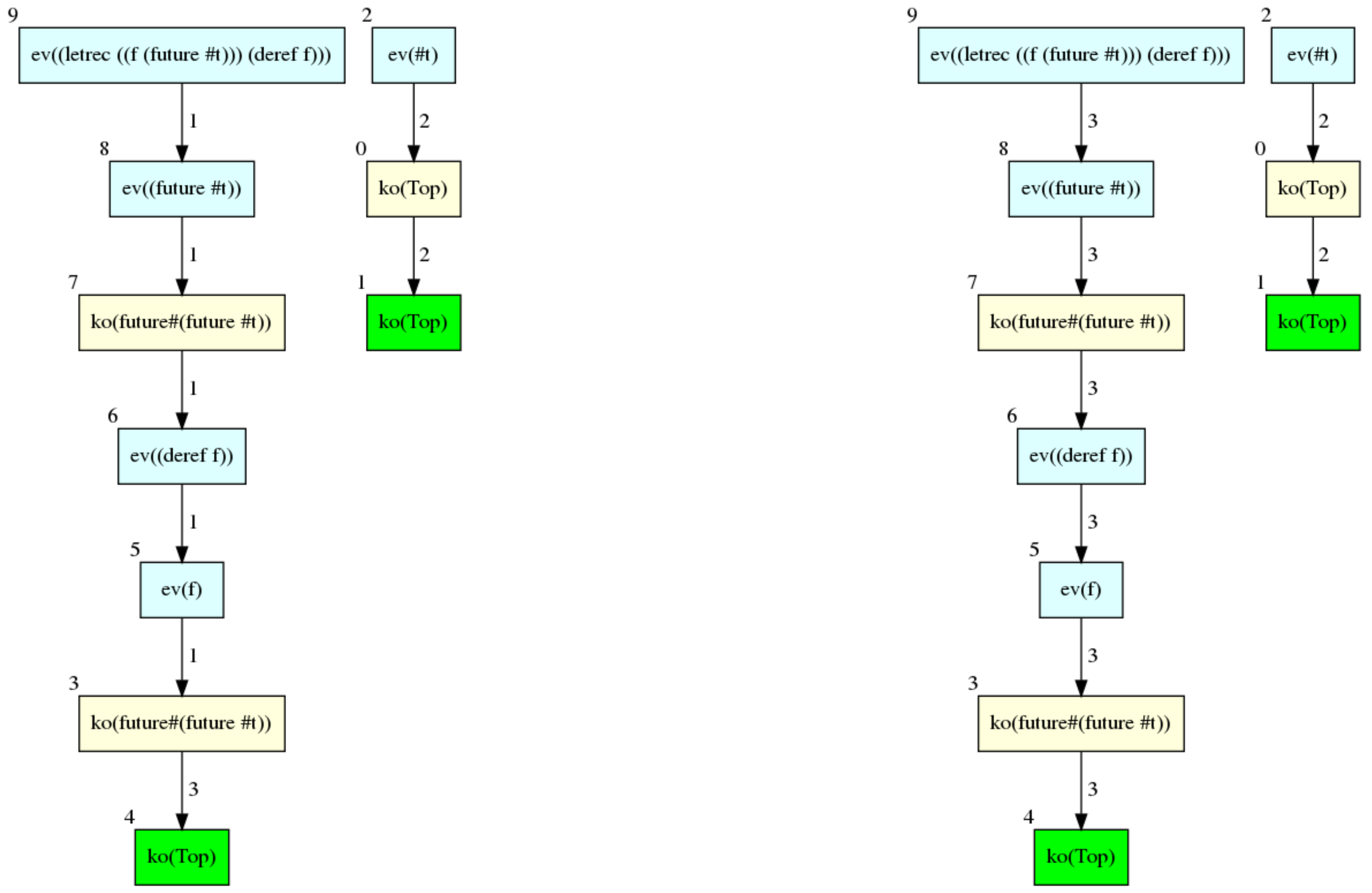
Figure 4.6a shows the output of the analysis of the program by IncAtom. IncAtom analyses this program as follows:

1. The main thread is analysed in isolation and a dereferencing dependency is generated.
2. The thread corresponding to the future `f` is analysed in isolation. Its new return value, top, is stored.
3. The new return value obtained in step 2 causes the main thread to be reanalysed. However, this reanalysis starts from the point where the return value of `f` is read by the main thread.

In Figure 4.6a, the transition arrows indicate the step in which a state was generated. The left graph component corresponds to the analysis of the main thread and is mostly generated in step 1 of the analysis. The updated return value of future `f` causes state 3 to be reanalysed in step 3, but reanalysis is avoided for states 5 to 9. The right graph component corresponds to the analysis of future `f`. As can be seen in the figure, this thread is analysed in step 2 of the analysis.

Figure 4.6b shows the output of the analysis of the program by ModAtom. ModAtom analyses this program in a similar way to IncAtom. However, in step 3 of the analysis, the main thread is reanalysed in its entirety, starting from its initial state (state 9). Hence, states 5 to 9 are reanalysed although they were not affected by the updated return value of future `f`.

Both ModAtom and IncAtom produce an abstract state graph containing 10 abstract states. Their modular nature avoids the need for the analysis to explicitly consider all thread interleavings. On the contrary, this is exactly how our non-modular analysis algorithm works. The result of the non-modular analysis of the program in Listing 4.2 is depicted in Figure 4.7. Now, a single graph containing 20 abstract states is obtained. Hence, even for a very simple program, the benefits of a modular analysis are clear.



(a) Analysis result of IncAtom.

(b) Analysis result of ModAtom.

Figure 4.6.: Result of analysing the program in Listing 4.2 using the modular analysis algorithms.

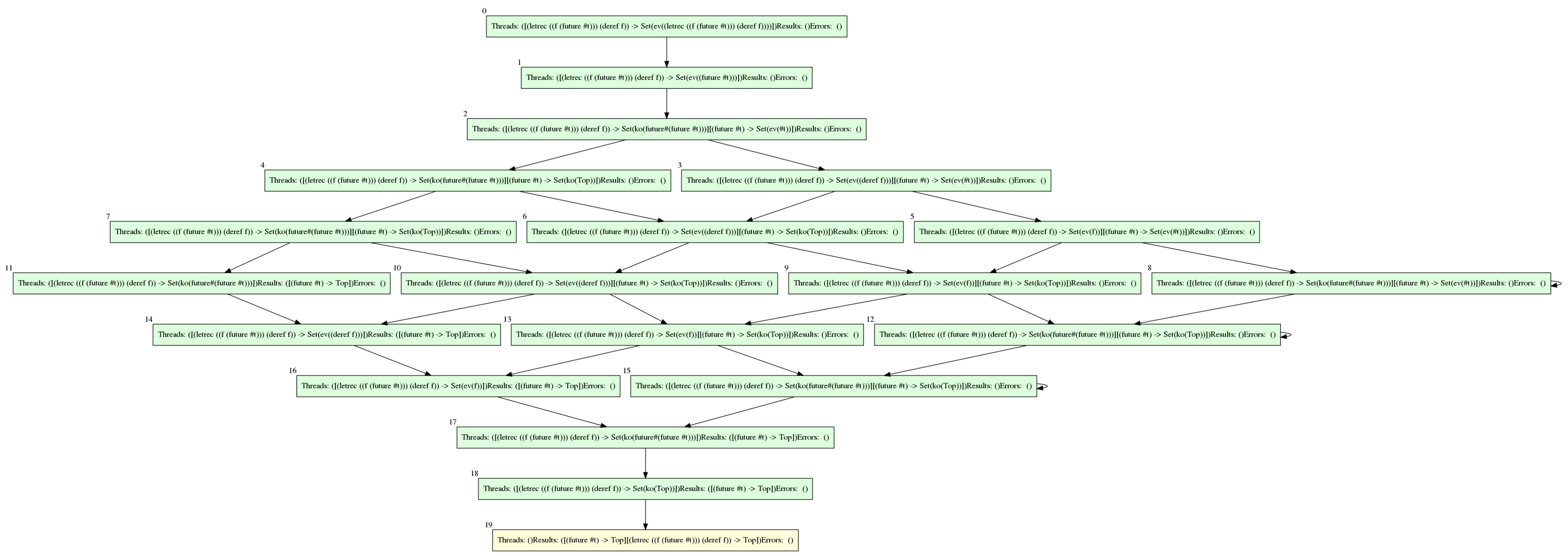

0
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev((letrec ((f (future #t))) (deref f))))])Results: ()Errors: ()
1
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev((future #t)))])Results: ()Errors: ()
2
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))][(future #t) -> Set(ev(#t))])Results: ()Errors: ()
4
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
3
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev((deref f)))][(future #t) -> Set(ev(#t))])Results: ()Errors: ()
7
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
6
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev((deref f)))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
5
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev(f))][(future #t) -> Set(ev(#t))])Results: ()Errors: ()
11
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))])Results: ([(future #t) -> Top])Errors: ()
10
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev((deref f)))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
9
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev(f))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
8
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))][(future #t) -> Set(ev(#t))])Results: ()Errors: ()
14
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev((deref f)))])Results: ([(future #t) -> Top])Errors: ()
13
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev(f))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
12
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
16
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ev(f))])Results: ([(future #t) -> Top])Errors: ()
15
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))][(future #t) -> Set(ko(Top))])Results: ()Errors: ()
17
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(future#(future #t)))])Results: ([(future #t) -> Top])Errors: ()
18
Threads: ([(letrec ((f (future #t))) (deref f)) -> Set(ko(Top))])Results: ([(future #t) -> Top])Errors: ()
19
Threads: ()Results: ([(future #t) -> Top][(letrec ((f (future #t))) (deref f)) -> Top])Errors: ()

Figure 4.7.: Result of analysing the program in Listing 4.2 using the non-modular analysis algorithm.

4.7. Conclusion

In this chapter, we first discussed effects as a means to formalise thread interference. We distinguished four types of effects and added them to the formalisation of λ_α, resulting in λ_ϵ. This addition enables λ_α to be analysed using a thread-modular static analysis.

We presented an algorithm for a non-incremental thread-modular analysis for λ_α, based on an existing algorithm introduced by Stiévenart (2018), in Section 4.2 and showed that this algorithm terminates. We found however that when the algorithm reanalyses an abstract thread, it entirely reanalyses the thread even though this may be unnecessary. Therefore, we proposed to render the intra-process analysis phase of the algorithm incremental. This allows an abstract thread to be reanalysed starting from the point where another thread interfered, therefore avoiding redundant work and lowering the analysis' runtime.

After having introduced our new incremental thread-modular analysis, we proposed two optimisations that may decrease the time needed by the analysis even further. First, we propose visited set caching to reduce the size of the abstract state space that may need to be traversed, but we also impose restrictions on the use of this technique to ensure soundness. Second, we present intra-process analysis abortion, which aborts the analysis of an abstract thread when it is known that a later reanalysis is needed. Hence, this technique aims to avoid duplication of work. We refer to Chapter 6 for an evaluation of these optimisations.

In Section 4.5, we discussed some general remarks regarding both the incremental and non-incremental algorithms for the thread-modular analysis of λ_α. Finally, we concluded by exemplifying how the different analysis algorithms analyse a simple concurrent program. This example illustrates the benefit of using modular analysis techniques, as well as how the incrementality of the intra-process analysis of IncAtom results in a reduction of the work performed by the static analysis.

5

IMPLEMENTATION

In Chapter 3, we presented λ_α, a concurrent language with atoms of which we formalised both the concrete and abstract semantics. Thereafter, in Chapter 4, we presented two algorithms to perform a thread-modular analysis of λ_α. In this chapter, we discuss how the semantics of λ_α and the presented algorithms are implemented.

The implementation of the work in this dissertation has been incorporated in Scala-AM, a modular framework written in Scala used to experiment with AAM-based abstract interpreters (Stiévenart, Vandercammen, et al., 2016; Stiévenart, Nicolay, et al., 2016). We first briefly introduce the framework in Section 5.1. Next, we discuss our implementation of futures and atoms in Section 5.2. Last, we discuss the implementation of the non-incremental and incremental thread-modular analysis algorithms for λ_α in Section 5.3. In our descriptions, we will try always to stay high-level and to omit detailed information to ease comprehension. Our entire implementation is publicly available in an online source code repository[1].

5.1. Background on Scala-AM

Scala-AM is a framework designed to facilitate the implementation of static analyses which are based on the AAM design method. The key property of the framework distinguishing it from other comparable frameworks is its functional and modular design: the implementation of Scala-AM is partitioned in different components that each are responsible for a different aspect of an abstract interpreter. By combining implementations for the different components, an abstract interpreter is obtained and by exchanging the implementation of one component for another implementation, the abstract interpreter can perform a different analysis, use a different precision, analyse another language,. . . without the need to modify the implementation of the other framework components.

The Scala-AM framework consists out of five main components (Stiévenart, Vandercammen, et al., 2016):

- The **semantics** component of Scala-AM prescribes how the expressions of a language must be evaluated. By changing the implementation of this component, the framework can be configured to analyse a different language.

[1]See branch `thesis` on `https://github.com/jevdplas/scala-am`.

- The value domain of the analysed language is implemented in Scala-AM's **lattice** component. This component represents the (abstract) values used by the framework's semantics.
- The **timestamp** component allows to write context-sensitive analyses (see our corresponding remark in Section 3.1.2).
- The algorithm used to compute the analysis result is implemented in the **machine** component of Scala-AM.
- The **address** component of Scala-AM implements the (continuation) addresses the framework uses.

For each component, a specific interface is defined by which the different components interact, allowing to interchange the implementations of components easily. For example, the framework can be used both as an abstract interpreter as well as a concrete interpreter by swapping in a concrete implementation for the lattice, timestamp and address components, but without having to alter the semantics and machine components. As a result, the framework's modular design simplifies the study of different analyses.

In addition to these five main components, there are also components that are used to facilitate the cooperation and decoupling of the different components. For example, to evaluate an expression under a given environment and store, the machine component calls the `stepEval` function of the semantics; to continue with a given value and continuation frame, the machine calls the `stepKont` function of the semantics. The results of such calls are *actions*, which are used by the semantics to instruct the machine component. By means of actions, the implementations of the semantics and machine components are strictly decoupled. In Scala-AM, `stepEval` and `stepKont` have the following type signatures:

```
def stepEval(e: Exp, env: Env, store: Store, t: Time): Set[Action]
def stepKont(v: Val, frame: Frame, store: Store, t: Time): Set[Action]
```

The arguments given to `stepEval` are the expression to evaluate, the environment in which the expression is to be evaluated, the current store and a timestamp. `stepKont` gets the value resulting from evaluation, the topmost stack frame, the current store and a timestamp.

The implementations of the different components used directly impact the precision of the analysis performed by the framework. Scala-AM already contains a relatively complete implementation of the semantics of the R5RS Scheme language, as well as some implementations for lattices, timestamps and addresses. Furthermore, the framework contains machine implementations for an AAM analysis of single-threaded programs, as well as for a concrete interpretation of such programs.

The contributions we make require modifications to the different components of Scala-AM. In the remainder of this chapter, we present our implementation of futures and atoms, as well as our implementation of the modular analyses presented in Chapter 4.

5.2. Implementation of the Semantics of λ_α

We now discuss the implementation of the semantics of λ_α, a concurrent language with atoms. We base our work on the implementation of the Scheme semantics that is already present in the framework and consider this as an extended implementation of λ_0. This results in a Scheme-like language that is enriched with futures and atoms. We now first discuss how futures are implemented and how the machine component of Scala-AM is modified to support this concurrent language. Thereafter, we discuss how atoms are implemented.

5.2.1. Implementation of Futures and a Non-Modular Concurrent Analysis

To extend the basic Scheme implementation with futures, we must modify three components of the framework:

- The semantics component that defines how expressions are evaluated must be extended to support futures.
- The machine component, which defines how the fixed-point computation is performed, must be extended to support multiple threads.
- The lattice component must be extended since thread identifiers are added as a new value type.

We now briefly explain the extension of these three components.

The Scheme semantics distinguishes *regular functions* from *special forms*. When a regular function is called, a standard evaluation procedure is followed: first the operator and operands are evaluated, whereafter the body of the function is executed in an extended environment (see rules APPL-OPERATOR, APPL-OPERAND and APPL-BODY in Figure 3.5). On the contrary, when a special form is used, a custom evaluation procedure is followed. Most built-in primitives and all user-defined functions are regular functions, such as, for example, `+`, `list` and `equal?`. Special forms are, for example, `if` and `letrec` (see rules LETREC-BINDING and LETREC-BODY in Figure 3.5). Note that in our formalisation, we used an atomic evaluation function that allowed us to shorten the formalisation of the transition rules. Although it is possible to implement such a function, we refrain from doing so in our implementation. Instead, in our implementation, we do not distinguish atomic expressions from complex expressions but handle every expression in a uniform manner. This does not impact soundness in any way, but the abstract interpreter may need to perform more steps to analyse a given expression.

We classify the new primitives for futures, introduced in Figure 3.11, as special forms since they require a specialised treatment by the abstract interpreter. SCALA-AM distinguishes between regular Scheme functions and Scheme special forms in the Scheme parser, which is where source code is compiled to expressions $e \in Exp$. As a result, to add new special forms, it is key to first extend the parser of the language.

When the `stepEval` function of the semantics component encounters an expression `(future e)`, several steps need to be performed. First, a new abstract thread identifier is allocated, which is then transformed to an abstract lattice value that can be handled by the different components of the framework; this transformation is needed because thread identifiers are first-class citizens in λ_α. By means of the new action `NewFuture`, the machine is then instructed to extend its thread map with a new abstract thread linked to the newly generated abstract thread identifier whose control component c is set to evaluate the expression e. To evaluate a call `(deref e)`, we first evaluate `e`. Thereafter, when the semantics' `stepKont` function is called with the thread identifier `v` that `e` evaluated to, the machine is instructed to look up the corresponding return values. This is shown in Listing 5.1.

```scala
override def stepKont(v: Val, frame: Frame, store: Store, t: Time): Actions =
  ↪  frame match {
  <...>
  case FrameDeref() =>
    val futures = getFutures(v)
    if (futures.isEmpty) {
      Set(Err(TypeError("Cannot dereference non-future values.", v)))
    } else {
      futures.map(tid => Action.DerefFuture(tid, store))
    }
  <...>
}
```

Listing 5.1: Implementation of the `deref` special form (part).

It is the responsibility of the machine component to drive the analysis and to handle the different abstract threads; the machine component is also responsible for storing the abstract return values of the abstract threads. Therefore, the framework is extended with an implementation of its machine component that can handle multiple threads and that can understand actions related thereto, such as `NewFuture`. Our implementation of the machine component keeps track of all information the abstract concurrent transition needs, i.e., it keeps track of the thread map and current value store. In each step of the analysis, the machine applies the transition function to every non-halted thread in the thread map while ensuring that all possible thread interleavings are explored. This way, a non-modular analysis for concurrency is obtained. Like the thread-modular algorithms presented earlier, this non-modular algorithm computes a fixed-point and uses a visited set to ensure termination.

Hence, the implementation of futures follows our formalisation of Section 3.2.3, although there are minor differences between the operational formulation and our formalisation. Due to the separation of concerns applied in Scala-AM, the implementation of futures requires the modification of multiple components of the framework. Also, in our implementation, we lift several restrictions that were imposed earlier to facilitate the formalisation of λ_0. For example, our implementation allows `future` to be called with a body containing an arbitrarily long list of expressions to be successively executed by the new future. We do however disallow an empty body.

5.2.2. Implementation of Atoms

To implement atoms in Scala-AM, we modify two components of the framework. First, since atoms are values, the lattice component of the framework is extended. Second, the semantics component of the framework is extended with implementations for the new functions related to atoms; these primitives were presented in Section 3.18.

The extensions of the lattice component of the frameworks with atoms is straightforward. An atom is represented as a pointer to an address in the value store. At the address pointed to by the atom, the value encapsulated by the atom is stored inside a wrapper, indicating that the value is stored within an atom. This implementation is similar to the existing implementation of `cons` cells that was already present in the implementation of the semantics component but slightly differs from our formalisation.

Unlike the primitives that were added for futures, the primitives for atoms need not be implemented as special forms. Listing 5.2 shows the implementation of the `atom` primitive. In Scala-AM, every primitive has a name and must implement a function `call`, which is called upon a call to the primitive. When the primitive is called, first the number of arguments is checked (line 5) and an error is returned when the number of arguments differs from one (line 9). Otherwise, a new store address is generated (line 7), the store is extended with the value stored inside the atom and a pointer to this address is returned.

```
1  /** Implementation of the "atom" primitive. */
2  object Atom extends Primitive {
3    val name = "atom"
4
5    def call(fexp: Exp, args: List[(Exp, Val)], store: Store, t: Time):
    ↪  MayFail[(Val, Store, Effects), Error] = args match {
6      case (_, v) :: Nil =>
7        val addr = allocator.pointer(fexp, t)
8        MayFail.success((pointer(addr), store.extend(addr, atom(v)),
          ↪  Effects.wAddr(addr)))
9      case _ => MayFail.failure(PrimitiveArityError(name, 1, args.size))
10   }
11 }
```

Listing 5.2: Implementation of the `atom` primitive.

The implementation of the `atom` primitive in Listing 5.2 shows a new additional component of Scala-AM: the `MayFail` monad. The reason for the existence of this component lies in the non-determinism that may arise in an abstract interpreter, which can cause a primitive to succeed and return an error at the same time. By use of the `MayFail` monad, such instances can easily be handled.

Scala-AM's Scheme semantics performs evaluation differently for regular primitives (that are not special forms) and user-defined functions. The evaluation of the body of all predefined regular functions happens *atomically* from the viewpoint of the different threads, whereas the execution of user-defined functions does not; the evaluation of special forms may not be atomic. The fact that predefined regular functions are executed atomically facilitates the implementation of the `compare-and-set!` primitive since atomicity is hence guaranteed by the design of the framework itself. Nevertheless, the implementation of this primitive is relatively complex which is why we omit it here.

Unfortunately, the design of the framework hampers the implementation of `swap!`, for which it is needed to manually construct function calls based on the value stored inside the atom. Therefore, we implement `swap!` on top of `compare-and-set!`, as shown in Listing 5.3. We store this definition in a *prelude* and load this prelude before the analysis of a λ_α program.

```
(define (swap! at fun)
  (let ((value (read at)))
    (if (not (compare-and-set! at value (fun value)))
      (swap! at fun))))
```

Listing 5.3: Implementation of `swap!` on top of `compare-and-set!`.

5.3. Implementation of the Modular Analyses for λ_α

In this section, we discuss the implementation of the modular analysis algorithms for λ_α (Algorithms 1 and 3). Since we have already presented and discussed the algorithms in detail in Chapter 4, we will remain brief and only highlight the key aspects of the implementation.

To implement an analysis algorithm in Scala-AM, it suffices to provide a new implementation for the machine component of the framework. However, our implementation of the semantics of λ_α, presented in Section 5.2, does not contain the generation of effects. To add effects, we also must extend the semantics component of the framework.

5.3.1. Addition of Effects

To extend the implementation of the semantics with effects, two steps are performed. First, we define an extra additional component to represent effects in Scala-AM. The definition of effects is completely similar to their formalisation and we foresee an implementation of the four types of effects that were identified in Section 4.1. We however adhere to Scala-AM's modular semantics and make sure new effect types can be added easily. Therefore, we implement effects as classes that extend a specific interface; new types of effects can be added by implementing a new class implementing the interface. We build extra class hierarchies to allow differentiation between different types of effects: we distinguish between effects that are related to concurrency and those that are not, as well as between effects that were generated by reading a value (dereferencing and read effects) and effects that were generated by the modification of the state of the abstract interpreter (creation and write effects). The four types of effects are defined as follows:

```
case class WriteAddrEff(target: Addr) extends Effect
case class  ReadAddrEff(target: Addr) extends Effect
case class     SpawnEff(target:  PID) extends Effect
case class      JoinEff(target:  PID) extends Effect
```

Every effect is represented by means of a Scala *case class* which enables the use of pattern matching on its instances. Since the framework is modular, effects may be used with different implementations for addresses and thread identifiers. Hence, a bounded type parameter is used.

The second step is to parameterise the entire implementation of the semantics and machine components using the newly defined effects. After all, the effects are generated by the transition function, which is implemented in the semantics and machine components of Scala-AM. It is important to make sure that all effects are generated correctly, since otherwise, the analysis may not be sound.

In the implementation, there is a difference between the points where creation and dereferencing effects are generated and the points where read and write effects are generated. Since the machine component of the framework is responsible for handling the different abstract threads, it is also responsible for generating creation and dereferencing effects. Moreover, the addition of such effects can happen in a single place, at the point where the machine handles the instructions of the semantics, such as `NewFuture`. On the contrary, read and write effects must be generated every time the store is accessed, which happens in a significant percentage of the built-in Scheme functions. Therefore, the generation of read and write effects cross-cuts the entire implementation of the semantics component. An example of this is given in Listing 5.4, which shows the implementation of Scheme's `list` primitive. A call to this primitive may cause changes to the store. Hence, for every address that is written to, a write effect is generated (line 11). The most important changes we have made to this primitive are marked in red.

```
1  object ListPrim extends StoreOperation("list", None) {
2    override def call(fexp: Exp, args: List[(Exp, Val)], store: Store[Addr, Val],
     ↪  t: Time): MayFail[(Val, Store[Addr, Val], Effects), Error] =
3      args match {
4        case Nil => MayFail.success((nil, store, Effects.noEff()))
5        case (exp, v) :: rest =>
6          for {
7            (restv, store2, effs) <- call(fexp, rest, store, t)
8            consv  = cons(v, restv)
9            consa  = allocator.pointer(exp, t)
10           store3 = store2.extend(consa, consv)
11         } yield (pointer(consa), store3, effs ++ Effects.wAddr(consa))
12     }
13 }
```

Listing 5.4: Modifications to the implementation of the `list` primitive.

5.3.2. Implementation of the Thread-Modular Analyses

The implementation of the (incremental) thread-modular analyses requires a change to the fixed-point algorithm used so far. As a result, to implement the (incremental) thread-modular analyses, it suffices to modify the machine component of the framework, given of course that effects are present.

In Chapter 4, we have already presented and discussed the algorithms to perform an (incremental) thread-modular analysis; the presented pseudocode was based on our actual implementation. Therefore, our discussion of the implementation can remain brief.

The implementations of ModAtom (Algorithm 1) and IncAtom (Algorithm 3) are fairly straightforward and follow the outline of the algorithms. However, the Scala-AM framework is written in a functional style. As a result, the implementations of the algorithms are functional as well. This functional style is particularly practical in the modular framework, as it ensures that exchanging the implementation of one of Scala-AM' components cannot influence the behaviour of the other components.

Both implementations of the thread-modular analysis make use of the technique of intra-process analysis abortion, which was presented in Section 4.4.2, as it is clear that the use of this technique can lead to reduced analysis time. It is however less clear whether the same is true for visited set caching, as this technique may have limited applicability and may require a significant amount of memory. Therefore, we implement two versions of the incremental thread-modular analysis, one that performs visited set caching and one that does not.

5.4. Schematic Overview of the Implementation

In Section 5.2, we have presented the implementation of λ_α, a concurrent language with futures and atoms that was introduced in Chapter 3. Thereafter, in Section 5.3, we have discussed the implementation of the modular analysis algorithms for this language; these analysis algorithms were presented in Chapter 4. For each contribution, several of the components of the Scala-AM framework have been modified. In Figure 5.1, the components of the framework are laid out schematically in accordance with the analysis of a program. Each component that has been modified is annotated with the number of the chapter describing the contribution for which the component was modified.

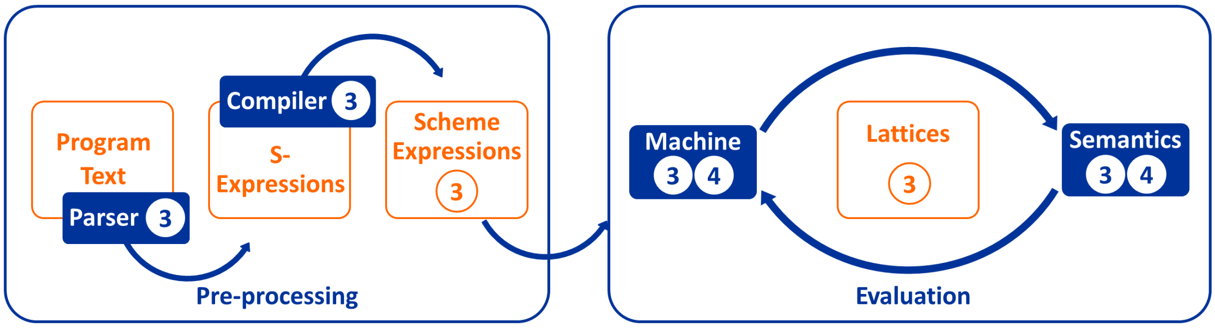


Figure 5.1.: Schematic overview of the components Scala-AM. The numbers indicate the chapters of this dissertation for which a component is modified.

5.5. Conclusion

In this chapter, we have presented an overview of the implementation of the contributions discussed in this dissertation. Our work has been incorporated in Scala-AM, a modular framework written in Scala to facilitate the construction of AAM-based abstract interpreters. The framework's implementation is modular to achieve a high separation of concerns; every component within the framework must implement a specified interface to allow interaction by other components.

We have implemented λ_α as an extension of the existing implementation of the Scheme programming language. To this end, we extended the semantics and lattice component of the Scala-AM framework. We also provided a new implementation of the framework's machine component to perform a non-modular analysis of λ_α.

To support modular analyses, a new additional component was added to the framework to represent the effects generated by the analysis. Afterwards, the actual algorithms to perform the modular analysis were implemented in the framework's machine component.

6 EVALUATION

In this chapter, we evaluate the contributions presented in the previous chapters. In this dissertation, we have made two contributions. First, we have applied a thread-modular static analysis to futures and atoms. Second, we have designed a new incremental algorithm to perform thread-modular analyses. We note again that this algorithm is not incremental with respect to source code changes, but with respect to the computation of the analysis result for a single abstract thread.

It is difficult to evaluate our first contribution as this requires an extensive quantitative investigation into the use of futures and atoms in real-life source code. In this chapter, we will therefore focus on the evaluation of our second contribution. In Section 6.1, we first empirically evaluate soundness of IncAtom since this is a crucial property: we require our analysis to be sound. Thereafter, we study the IncAtom in more detail in Section 6.2 by evaluating its behaviour according to several metrics. Finally, we present a conclusion in Section 6.3.

6.1. Soundness Testing

A first important aspect of IncAtom that is to be evaluated is soundness. To this end, we have empirically evaluated soundness of the algorithm. In Section 6.1.1, we first discuss the methodology used during the soundness evaluation, as well as our experimental setup. Next, in Section 6.1.3, we present our results.

6.1.1. Methodology and Experimental Setup

To evaluate the soundness of IncAtom, we perform an empirical evaluation wherein the results produced by IncAtom are compared to those produced by ModAtom. Our evaluation consists out of comparing the results of both algorithms on a set of 28 benchmark programs. We refer to the next section for a more detailed overview of these benchmark programs.

To evaluate soundness, we prepare two abstract abstract interpreters, one performing a ModAtom analysis and one performing an IncAtom analysis. For each of the 28 benchmark programs, we analyse the respective program using the two abstract interpreters, resulting in two sets of abstract state graphs. In addition, we also take the final stores resulting from the analyses into account; we will refer to the store resulting from the analysis performed by ModAtom as σ_M and to the store resulting from the analysis performed by IncAtom as σ_I. We call the analysis of a given program using a given algorithm *an experiment*.

The comparison of the results from ModAtom and IncAtom is done as follows: in each set of abstract state graphs, we look at the set of identifiers that are evaluated. For each identifier in this set, we look up the abstract value related to it by the corresponding final store, that is, in σ_M for the result of ModAtom and in σ_I for the result of IncAtom. For each identifier, we then verify that the abstract value stored in σ_I is an over-approximation of the abstract value stored in σ_M. If this is the case, IncAtom is sound given that ModAtom is sound. The reason for this is that in this case, it is guaranteed that the fixed-point computation performed by IncAtom considered at least the same abstract values than ModAtom for all identifiers. However, if this is not the case, we do not gain information on the soundness of IncAtom. The reason for this is that we now only know that σ_I is not an over-approximation of σ_M, which may be the case when IncAtom is unsound or when it is sound but more precise than ModAtom. Therefore, no conclusions can be drawn when σ_I does not over-approximate σ_M. The soundness of IncAtom is conditional on the soundness of ModAtom since the results of the latter are used to verify the results of the former. We refer to Stiévenart (2018) for a proof of the soundness of ModAtom.

Using the evaluation technique just described, which consists of comparing σ_I to σ_M, it is also possible to get information about the relative precision of IncAtom with respect to ModAtom. If, for every identifier, the corresponding abstract value in σ_I is identical to the abstract value in σ_M, IncAtom is at least as precise as ModAtom. If however, the abstract values related to some identifiers in σ_I are strict over-approximations of the values related to these identifiers by σ_M, the incremental analysis is less precise than the non-incremental analysis.

Now we have explained our methodology, we can discuss our experimental setup. As explained before, two abstract interpreters will be constructed; one abstract interpreter will use ModAtom whereas the other will use IncAtom. Hence, the two abstract interpreters use a different implementation of the machine component of Scala-AM. To allow a meaningful comparison of the results, the implementations of the other components of the framework are fixed.

All experiments are executed on a single machine whose specifications are listed in Table 6.1; henceforth, we will refer to this machine as *Bertha*. Note that although Bertha possesses multiple cores, our static analyser is a sequential program and hence only uses a single core. Also, we limit the amount of RAM used by the Scala runtime to 16 GB, an amount that is found in modern-day computers. Scala-AM is configured to use thread identifiers consisting of an expression and a timestamp, name addresses, no context sensitivity and the type lattice. Using name addresses signifies that the address of a variable equals its name (see the definition of $\widehat{alloc}$ in Figure 3.9). Having no context sensitivity means that

Hardware	
Model	**Dell PowerEdge R730 (2015)**
CPU	**2 × Intel Xeon 2637 v3**
Physical cores	2 × 4
Hyperthreading	Disabled
Base frequency	3.50 GHz
Max. turbo frequency	3.70 GHz
L1 data cache	2 × 4 × 32 kB
L1 instruction cache	2 × 4 × 32 kB
L2 unified cache	2 × 4 × 256 kB
L3 unified cache	2 × 15 MB
RAM	**256 GB**
OS	Ubuntu 18.04.2 LTS
Hostname	**bertha.vub.ac.be**
Software	
Java	1.8.0_212
Sbt	0.13.9
Scala	2.12.7

Table 6.1.: Specifications of the machine on which the experiments were executed.

timestamps are ignored. Since timestamps are ignored, abstract threads evaluating the same expression share the same thread identifier. Finally, the use of the type lattice implies that abstract values are represented by their type. For example, `1` is represented by `Int` and `"foo"` is represented by `String`. During soundness testing, we use a modified version of the semantics of λ_α that suppresses output but is otherwise identical to the regular semantics of λ_α.

Since the static analyser may possibly need a prohibitively long time to perform a single experiment, we impose a time out of 20 minutes. After this, we consider the experiment unusable and we do not collect results. Hence, for some experiments, we will not be able to present results.

6.1.2. Benchmark Programs

In this section, we give a detailed overview of the benchmark suite used for the evaluation of our work. In total, 28 benchmark have been used; these are listed in Table 6.2. For each benchmark, the name of the benchmark, the number of physical lines of code (PLOC) and a short description are given.

Originally, the benchmark programs made use of threads, locks and references. The programs were trivially adapted to use futures and atoms as follows: threads were converted to futures and references were converted to atoms. Additionally, to support the benchmarks using locks, we have used an extended preamble to which the code given in Example 3.5 is added. Finally, some minor modifications have been applied to some of the benchmarks where needed. The fact that the benchmarks initially did not use futures and atoms does not impact the validity of our results.

Table 6.2 contains a short description of every benchmark program. In each program, some code has been added to actually run the program, no such code was present yet. As an example, the `mcarlo` benchmark program is shown in Listing 6.1.

Name	PLOC	Description
abp	85	Simple client-server implementation.
actors	121	Thread-based implementation of actors.
atoms	67	Function memoisation using atoms, applied to the Fibonacci function.
bchain	111	Simple blockchain implementation.
count	51	Thread-safe counter.
crypt	221	Parallel cryptanalysis algorithm.
dekker	53	Dekker's algorithm for the critical section problem.
fact	69	Parallel factorial function.
life	169	Parallel implementation of Conway's game of life.
matmul	119	Benchmark that compares multiple matrix multiplication algorithms for correctness.
mcarlo	35	Parallel Monte Carlo simulation for the approximation of pi.
mceval	82	Meta-circular Scheme interpreter supporting threads.
minimax	129	Implementation of a simple game.
msort	45	Parallel merge sort.
nbody	163	Parallel mathematical simulation of the solar system.
pc	47	Producer-consumer problem.
phil	48	Dining philosophers problem.
phild	62	Alternative implementation of the dining philosophers problem.
pp	48	Parallel threads mutating shared memory.
pps	104	Implementation of the parallel prefix sum algorithm.
qsort	81	Parallel quicksort.
ringbuf	86	Ring-buffer benchmark.
rng	26	Parallel random number generator.
sieve	67	Parallel implementation of Eratosthenes' sieve.
stm	158	Thread-based implementation of software transactional memory.
sudoku	92	Parallel sudoku checker.
trapr	74	Parallel integration using trapezoids.
tsp	149	Travelling salesman problem.

Table 6.2.: Overview of the benchmarks.

```scheme
;; Monte-carlo simulation using futures.

(define MAXSIZE 10000)

(define (inside-circle? radius x y)
  (< (+ (* x x) (* y y)) (* radius radius)))

(define (monte-carlo-seq size n)
  (define (monte-carlo-helper i amount)
    (if (= i 0)
        amount
        (let ((px (random size))
              (py (random size)))
          (if (inside-circle? size px py)
              (monte-carlo-helper (- i 1) (+ amount 1))
              (monte-carlo-helper (- i 1) amount)))))
  (monte-carlo-helper n 0))

(define (monte-carlo-conc size n)
  (if (< n MAXSIZE)
      (monte-carlo-seq size n)
      (let ((t1 (future (monte-carlo-conc size (quotient n 2))))
            (t2 (future (monte-carlo-conc size (quotient n 2)))))
        (+ (deref t1) (deref t2)))))

(define (approximate-pi size iterations)
  (/ (* 4. (monte-carlo-conc size iterations)) iterations))

;; Run the program.
(define radius 1000000000)
(define pi (approximate-pi radius 100000000))
(display pi)
(if (< (abs (- 3.14 pi)) 0.01)
    (display "Looks like pi!")
    (display "Not really good."))
```

Listing 6.1: `mcarlo` benchmark program.

6.1.3. Results

In the previous sections, we have presented an experimental evaluation strategy to verify the soundness of IncAtom. A result was obtained for 24 out of the 28 benchmarks; the other benchmarks timed out or caused an error, due to a shortage of memory, for example. The results of the evaluation for these 24 benchmarks are depicted in Table 6.3.

Our experimental evaluation reveals that IncAtom is guaranteed to be sound on 21 out of the 24 benchmarks. We find that, on these benchmarks, IncAtom is as precise as ModAtom, that is, the incrementalisation of ModAtom has not lead to a loss of precision. This result was expected since the difference between IncAtom and ModAtom is limited to IncAtom using a finer granularity of effect tracking. Although this is a fundamental change to the algorithm, we did indeed not expect this to influence soundness or to have a negative impact on precision as we did not make changes that are expected to negatively influence the algorithm's soundness.

However, for the other 3 benchmarks, we find that the store returned by of IncAtom, σ_I is not an over-approximation of the store returned from ModAtom, σ_M. Importantly, this does not imply that IncAtom is unsound on these benchmarks but only means that soundness cannot be established by comparing σ_I to σ_M. Therefore, for these 3 benchmarks, we manually analyse the differences between the results of ModAtom and IncAtom. We find that the observed differences are related to precision improvements, that is, we find that on these 3 benchmarks, IncAtom is more precise than ModAtom. A more thorough discussion on why IncAtom may have a higher precision than ModAtom is given in section 6.2.2.

6.2. Metrics for IncAtom

After having verified soundness of IncAtom in the previous section, we now empirically evaluate the algorithm's behaviour according to several metrics, thereby comparing it to its non-incremental counterpart, ModAtom. First, in Section 6.2.1, we discuss our methodology and experimental setup. Next, in Section 6.2.2, we present the obtained results.

6.2.1. Methodology and Experimental Setup

To evaluate the behaviour of our incremental algorithm, we identify two metrics upon which our evaluation can be based:

- The **number of states** explored by the static analyser gives an indication of the amount of work that is needed to analyse a program. Under certain circumstances, the number of states is also an indicator for the precision of the analysis: when a fixed lattice is used, a lower number of states may indicate a higher precision.
- The **analysis time** is the time needed by the static analyser to analyse a program. Lower analysis times indicate that the analysis scales better to large programs.

In this section, we discuss how the metrics just described are evaluated and introduce our

Benchmark	Soundness Test	Precision	Benchmark	Soundness Test	Precision
abp	✓	Same	pc	✓	Same
actors	✓	Same	phil	✓	Same
atoms	✓	Same	phild	✓	Same
bchain	✓	Same	pp	✓	Same
count	✓	Same	pps	✓	Same
dekker	✓	Same	qsort	✓	Same
fact	**?**	–	ringbuf	✓	Same
mcarlo	✓	Same	rng	✓	Same
mceval	✓	Same	sieve	✓	Same
minimax	**?**	–	sudoku	✓	Same
msort	**?**	–	trapr	✓	Same
nbody	✓	Same	tsp	✓	Same

Table 6.3.: Results of the experimental soundness evaluation of IncAtom.

experimental setup.

To evaluate the number of states and the analysis time, we will compare three different analysis algorithms with respect to the above metrics:

- ModAtom (Algorithm 1 of Section 4.2);
- IncAtom (Algorithm 3 of Section 4.3);
- IncAtom with visited set caching.

We do not explicitly compare our thread-modular algorithms to a non-modular algorithm but refer to Stiévenart (2018) for a detailed comparison of a non-modular analysis and a ModConc analysis; our analyses are based on the latter.

For IncAtom, we foresee an extra metric: the *Average Computation Reuse Ratio* (ACRR). We define this metric as follows. For each thread that is analysed multiple times, the CRR is the percentage of edges in the thread's result graph that were not generated during the last intra-process analysis of the thread. The ACRR is then the average of the CRR for all threads that were analysed multiple times. For IncAtom with visited set caching, we also foresee an extra metric, the *Visited Set Reuse Ratio*, to determine how often a cached visited set actually is reused. We define this metric as the percentage of reanalyses where a cached visited set is used.

We compare the three algorithms mentioned above as follows. We prepare an abstract abstract interpreter by fixing the implementation of all of Scala-AM's components, except for the implementation of the machine component. For each of the algorithms above, we swap in the corresponding machine component and then use the framework to analyse a given program. Fixing the implementations of the other components is needed to obtain a meaningful comparison of the different machines, as the use of another implementation for one of the other components, such as for the lattice component, may lead to different results. Again, a set of 28 benchmark programs is used for the evaluation of the metrics and we refer again to Section 6.1.2 for a more detailed overview of these programs. We call the analysis of a given program using a given algorithm *an experiment*.

Our experimental setup is identical to the one used for soundness testing, which is described in Section 6.1.1. Again, all experiments are executed on Bertha, whose specifications are listed in Table 6.1. Scala-AM is configured to use thread identifiers consisting of an expression and a timestamp, name addresses, no context sensitivity and the type lattice. As before, the amount of RAM used by the Scala runtime is limited to 16 GB and a time out of 20 minutes is imposed. During each experiment, we use a modified version of the semantics of λ_α that suppresses output but is otherwise identical to the regular semantics of λ_α.

To measure the number of states that are explored by the static analyser, it suffices to execute every experiment only once, that is, to analyse every benchmark program once using every algorithm. The same is true for measuring the average computation reuse ratio and the visited set reuse ratio. However, every experiment must be repeated multiple times in order to measure its analysis time. The reason for this is that the exact time to run the analysis may differ from one execution to the next due to nondeterminism in the CPU of Bertha and the use of a JIT compiler by the JVM, for example. We foresee three warm-up runs followed by 20 timed iterations to measure the analysis time needed for every experiment. We then take the average of these measurements as the result for the given experiment. Every iteration of every experiment is preceded by a manual invocation of the garbage collector to reduce possible variances of the analysis time caused by memory management performed by the Scala runtime.

When measuring the analysis time for an experiment, we require every repetition of the experiment, including all warm-up runs, to run within the required time frame. Otherwise, the experiment is considered unusable and no results can be collected.

6.2.2. Results

In this section, we describe the results obtained from the execution of the experiments described in the previous section. We separately discuss the results obtained for each metric. We first present the results for the analysis time, followed by the results for the number of states generated by the analyses. Finally, we discuss the results for the average computation reuse ratio. Measurements of the analysis time are always rounded to three decimal places and percentages are always rounded to two decimal places.

Analysis Time

Table 6.4 depicts the average time needed by the different analysis algorithms to analyse the different benchmarks, as well as the sizes of the corresponding 95% confidence intervals which have been calculated using a Student's t-distribution with 19 degrees of freedom. Not all benchmarks could always be analysed successfully. We require all 3 warm-up runs and 20 timed repetitions of an experiment to have completed successfully in order for the result to be usable. When at least one repetition of an experiment timed out, this is indicated with ∞. Benchmarks which timed out on all analysers have been omitted. Also, for some experiments, the analysis resulted in an error. We found that these errors all are memory related. In the remainder of this chapter, we will only focus on the 23 benchmarks that were successfully analysed by at least one machine; the other 5 benchmarks have been omitted. The fact that all iterations must finish in time may explain why a soundness result was obtained for the `tsp` benchmark, whereas no timing results were obtained for the benchmark.

Modular analyses
An important contribution of this thesis is the incrementalisation of ModAtom. In Chapter 4, we claimed that ModAtom performs redundant work which could cause an unnecessary increase in analysis time. Therefore, a detailed comparison of the average analysis times of ModAtom and IncAtom is given in Table 6.5. We find that the analysis times of IncAtom are lower than those of ModAtom for 21 out of 23 benchmarks and higher for 2 out of 23 benchmarks. It is, however, important to point out that the absolute analysis time of some benchmarks, such as `dekker` and `pp`, is rather low, which means that the measurements for these benchmarks may contain a relatively large measurement error. Hence, interpreting such results may be misleading. Therefore, it is important to take the size of the confidence intervals, as depicted in Table 6.4, into account.

When looking at the last column of Table 6.5, we find that IncAtom only is slower than ModAtom on 2 out of 23 benchmarks, `actors` and `pps`. However, for `actors`, the confidence intervals overlap and hence we cannot conclude that there is indeed a performance difference for the benchmark. On `pps` however, the difference in performance is more substantial: IncAtom is around 10.5% slower than ModAtom and the confidence intervals do not overlap. The reason for this slowdown is unclear however and may need further investigation. However, in absolute time, the difference is negligible.

Looking again at the benchmarks where IncAtom performs better than ModAtom shows that the reduction of the analysis time is relatively small on some benchmarks but significant on others. The highest reduction of the analysis time is seen for `fact`, which is analysed almost

Benchmark	**ModAtom**	**IncAtom**	**IncAtom with visited set caching**
abp	48.937 ± 0.869	45.558 ± 0.657	1357.130 ± 8.732
actors	870.514 ± 15.136	874.208 ± 14.165	70996.681 ± 160.543
atoms	40.059 ± 0.632	35.817 ± 0.559	989.321 ± 2.871
bchain	87.447 ± 1.916	80.550 ± 1.177	4234.056 ± 11.297
count	21.249 ± 0.553	19.337 ± 0.492	230.963 ± 2.327
dekker	7.092 ± 0.579	6.580 ± 0.565	27.951 ± 0.634
fact	507.727 ± 8.456	185.647 ± 1.747	17439.612 ± 72.188
mcarlo	76.920 ± 0.430	32.663 ± 0.214	1115.855 ± 0.259
mceval	160546.763 ± 926.474	127847.833 ± 66.213	∞
minimax	7805.516 ± 55.231	7564.548 ± 71.653	739694.057 ± 620.763
msort	245.178 ± 4.982	144.588 ± 2.360	16320.332 ± 20.776
nbody	476.183 ± 6.723	472.187 ± 7.076	55966.308 ± 101.272
pc	18.553 ± 0.310	11.190 ± 0.293	153.333 ± 2.346
phil	15.958 ± 0.684	13.808 ± 0.229	127.453 ± 0.958
phild	28.071 ± 0.448	25.777 ± 0.296	413.474 ± 4.145
pp	14.141 ± 0.342	12.723 ± 0.269	117.358 ± 0.618
pps	291.446 ± 5.252	322.272 ± 3.426	15700.026 ± 32.355
qsort	182.635 ± 3.074	99.366 ± 1.699	3092.401 ± 21.297
ringbuf	31.365 ± 0.329	29.155 ± 0.398	857.569 ± 10.007
rng	14.398 ± 0.184	12.464 ± 0.113	340.833 ± 3.821
sieve	26.625 ± 0.370	23.684 ± 0.616	438.408 ± 3.808
sudoku	1301.171 ± 23.045	1285.861 ± 10.124	11451.039 ± 47.353
trapr	17.105 ± 0.908	13.647 ± 0.139	170.188 ± 1.835

Table 6.4.: Average time needed by the different static analysers to analyse the given benchmarks in milliseconds together with the size of the 95% confidence interval. ∞ indicates a time out. Benchmarks for which no result was obtained are omitted.

63.5% faster by IncAtom compared to ModAtom. Significant speed-ups are also seen for other benchmarks such as `mcarlo` and `qsort`. The biggest difference in absolute analysis time is seen for `mceval`, which is analysed around 30 seconds faster by IncAtom than by ModAtom.

However, the reductions in analysis time strongly depend on the actual program that is analysed. There may be different reasons for this. For example, thread interference, which may cause a thread to be reanalysed, can happen in the beginning of its body or it can happen in the end of the body. It is clear that in the first case, more of the thread's body needs to be reanalysed than in the second case. The same is true for reanalyses that are triggered due to read-write or write-write conflicts. Additionally, not every thread may be reanalysed and the time needed to analyse different abstract threads may also differ strongly. Hence, the performance gains that can be achieved using an incremental analysis are very program dependent.

In summary, our results show that, in general, IncAtom performs better than ModAtom. We found that only one benchmark was analysed significantly slower by IncAtom than by ModAtom but that the absolute difference in analysis time is negligible. However, IncAtom outperforms ModAtom on a vast majority of the benchmarks. For the `fact` benchmark, a reduction of the average analysis time of 63.44% is seen. The most significant reduction of the absolute analysis time is seen for `mceval`, which is analysed around 30 seconds faster. Hence, significant speed-ups are found, both in absolute as well as in relative numbers. However, the gains in analysis time are very program dependent.

Benchmarks	ModAtom	IncAtom	Difference
abp	48.937	45.558	−6.90%
actors	870.514	874.208	+0.42%
atoms	40.059	35.817	−10.59%
bchain	87.447	80.550	−7.89%
count	21.249	19.337	−9.00%
dekker	7.092	6.580	−7.22%
fact	507.727	185.647	−63.44%
mcarlo	76.920	32.663	−57.54%
mceval	160546.763	127847.833	−20.67%
minimax	7805.516	7564.548	−3.09%
msort	245.178	144.588	−41.03%
nbody	476.183	472.187	−0.84%
pc	18.553	11.190	−39.69%
phil	15.958	13.808	−13.47%
phild	28.071	25.777	−8.17%
pp	14.141	12.723	−10.03%
pps	291.446	322.272	+10.58%
qsort	182.635	99.366	−45.59%
ringbuf	31.365	29.155	−7.05%
rng	14.398	12.464	−13.43%
sieve	26.625	23.684	−11.05%
sudoku	1301.171	1285.861	−1.19%
trapr	17.105	13.647	−20.22%

Table 6.5.: Detailed comparison of the average time needed by ModAtom and IncAtom for the analysis of the different benchmarks. Times are denoted in milliseconds.

Visited Set Caching

In Section 4.4, two optimisations for IncAtom were introduced. Intra-process analysis abortion has been integrated both in ModAtom as well as in IncAtom. However, due to its possibly limited applicability, the technique of visited set caching was not readily incorporated into IncAtom. To evaluate whether visited set caching is a good technique to further reduce the analysis time, we compare two variants of IncAtom: one variant wherein visited set caching is not included and one variant wherein the optimisation is included. The results of these two algorithms are depicted in the two last columns of Table 6.4 respectively. We see that the version of IncAtom with visited set caching performs significantly worse than IncAtom on all benchmarks. For example, on the `actors` benchmark, we find that IncAtom with visited set caching is more than 81 times slower than IncAtom without visited set caching and on the `fact` benchmark, the version with visited set caching is even around 94 times slower. In fact, we find that the visited set reuse ratio is zero for all benchmarks.

The reason why the visited set reuse ratio is zero for all benchmarks may be caused due to the fact that there is a high probability that the store is changed between the point where a visited set is cached and the point where it could be reused. After all, a cached visited set is only reused if a thread is reanalysed, which may be the case when the store changes or when another abstract thread's return value changes. In the first case, the store is changed and it becomes impossible to use the cached visited set. Although the store is not necessarily changed in the second case, there is a high probability that the store has changed indeed. Hence, we conclude that visited set caching is not a usable optimisation.

It remains unclear however why exactly the version of IncAtom with visited set caching results in significantly higher analysis times compared to the version of IncAtom without this *optimisation*; the analysis using visited set caching even times out on the `mceval` benchmark. Reasons for this behaviour may be the increased memory overhead: we have not implemented invalidation of cached visited sets and hence they can never be garbage collected. As a result, there may be a higher garbage collection overhead. However, this is most likely not the only cause of this significant difference in performance, as otherwise, we would expect an exception to be thrown by the Java runtime environment. Hence, we assume that there might be a hidden overhead related to the caching of the visited set in the implementation. Unfortunately, the exact reason for such a performance overhead may be very difficult to pinpoint.

Benchmark	ModAtom	IncAtom	IncAtom with visited set caching	Difference ModAtom−IncAtom
abp	636	636	636	
actors	2128	2128	2128	
atoms	406	406	406	
bchain	795	795	795	
count	504	504	504	
dekker	313	313	313	
fact	3097	2641	2605	−14.72%
mcarlo	998	918	918	−8.02%
mceval	15197	14897	∞	−1.97%
minimax	3957	3912	3912	−1.13%
msort	1924	1856	1856	−3.53%
nbody	1700	1700	1700	
pc	361	361	361	
phil	372	372	372	
phild	450	450	450	
pp	396	396	396	
pps	972	972	972	
qsort	2234	2202	2202	−1.43%
ringbuf	486	486	486	
rng	265	265	265	
sieve	442	442	442	
sudoku	22005	22005	22005	
trapr	459	459	459	

Table 6.6.: Number of states in the abstract state graphs generated by the different algorithms for the different benchmarks. ∞ indicates a time out and hence the absence of measurements.

Number of States

Table 6.6 depicts the number of states generated by the different analysis algorithms for the different benchmarks. Note that the numbers in this table denote the number of states present in the states graph outputted by the analysis algorithms after filtering (see Section 4.5.1). The graphs outputted by ModAtom and IncAtom do not always contain the same number of states. On the contrary, the incorporation of visited set caching in IncAtom does not change the number of states. Only for the `fact` benchmark, the version with visited set caching results in slightly less states but the reason for this is still to be investigated.

The last column of Table 6.6 shows a detailed comparison of the sizes of the graphs outputted by ModAtom and IncAtom. Although the number of states is equal on 17 out of 23 benchmarks, IncAtom produces fewer states in 6 out of 23 cases, with a reduction of 14.72% for the `fact` benchmark. The fact that IncAtom produces smaller state graphs than ModAtom indicates that it can analyse the benchmarks with a higher precision. After all, since both analyses are sound, this means that the graphs produced by ModAtom contain spurious paths that are not generated by IncAtom.

The fact that IncAtom may generate fewer states than ModAtom stems from the algorithm's incremental nature: the algorithm only reanalyses an abstract thread from the point where it may be affected by another abstract thread, as well as from an assumption that is made in the Scala-AM framework but that is unsound in general. The assumption made within the framework is that no accessor function is ever applied to an empty list or to an empty vector; the goal of this assumption is to suppress the generation of many false error states and it does not cause significant soundness issues. Concretely, when an accessor is applied to an empty list or to an empty vector, $\perp$ is returned instead of an error. Under certain conditions, this may also cause no dereferencing effect to be generated by the analysis when a call to `deref` is evaluated.

The fact that no dereferencing effect is generated may seem problematic but in fact allows IncAtom to avoid the generation of spurious paths in the abstract state graph and hence causes the result of IncAtom to be more precise than the result of ModAtom under these specific circumstances. To explain this, consider the Scheme program in Listing 6.2 and a specific trace in the graph of the abstract main thread resulting from the analysis of the program by IncAtom in Figure 6.1.

```
(define (iter n)
  (if (> n 0)
    (cons
      (future #t)
      (iter (- n 1)))))

```

```
(let loop ((f (iter 2)))
  (if (null? f)
    #t
    (begin
      (deref (car f))
      (loop (cdr f)))))
```

Listing 6.2: Scheme program for which IncAtom generates fewer states than ModAtom.

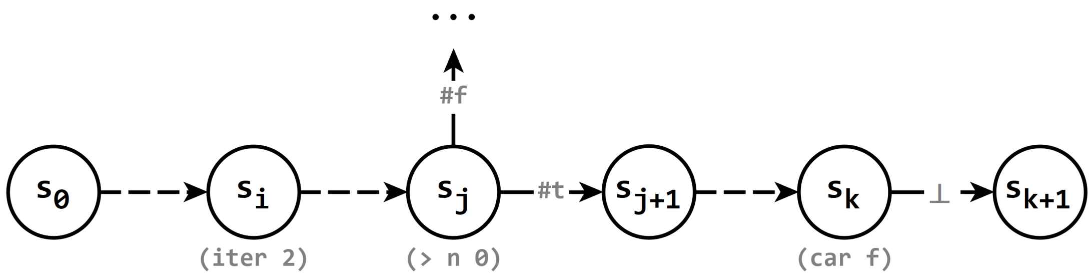


Figure 6.1.: Specific trace in the graph of the main thread of the program in Listing 6.2.

The abstract analysis algorithm starts by analysing the main thread. At a given moment, `(iter 2)` is analysed in a state s_i. When the abstract interpreter continues, it finds the predicate `(> n 0)` and analyses it (state s_j). In our experiments, the abstract interpreter does not have sufficient precision to know which branch the program actually takes and hence has to explore both branches. When continuing in the branch where the predicate corresponds to true, `iter` returns and the if-statement in `loop` is evaluated. Again, the abstract interpreter does not have sufficient knowledge to decide on the branch to evaluate and hence has to evaluate both branches. When it evaluates the else-branch, it analyses the call `(car f)`. Since, in this path, `f` is empty, $\perp$ is returned. Since $\perp$ does not represent a thread identifier, no dereferencing effect is generated by the call `(deref (car f))` and this path in the graph ends with an error since a non-future value is dereferenced.

If the graph of the program is explored in this exact order, no values are stored in the store at the address where the list is stored. Hence, no thread identifier is found and no dereferencing effect is generated. If this branch of the program is reanalysed later, a thread identifier may be found since the store may have changed during the analysis of another abstract thread, for example. And, since addresses may be reused, a thread identifier may be stored at the given address. Hence, whereas the initial analysis stopped at s_{k+1}, the reanalysis of this branch will continue. However, when this branch of the abstract state graph is not analysed again, no extra states will be generated. Therefore, it is possible that this branch is not extended by IncAtom whereas it is extended by ModAtom. Note that the graph generated by IncAtom still correctly over-approximates the behaviour of the given program.

Average Computation Reuse Ratio

Table 6.7 depicts the average computation reuse ration for the different benchmarks. A high ACRR indicates that the analysis is able to reuse a significant amount of the previously generated result upon reanalysis of an abstract thread. As can be seen in the table, we were unable to compute the ACRR for the `actors` benchmark. The reason for this is that none of the abstract threads is reanalysed during the analysis of the benchmark, which is possible since no future is dereferenced in the benchmark program. When looking at the other benchmarks, we see that the average computation result ratio of ranges up to 96.31%. On 13 out of 23 benchmarks, the ACRR exceeds 50% and on 7 out of 23 benchmarks, the ACRR exceeds 65%. Hence, overall, IncAtom is indeed able to successfully reuse previously computed results.

However, the ACRR only takes threads that are reanalysed into account, which means a high ACRR does not necessarily correspond to a high reduction of the analysis time and vice versa. For example, suppose a program consisting out of two threads of which one of the two threads is reanalysed. If the abstract state graph of the thread that is reanalysed is very small compared to the abstract state graph of the thread that is not reanalysed, being able to reuse a large part of the previously calculated result of the reanalysed thread may only have a minor influence on the total runtime of the analysis. Indeed, when relating the ACRR of the benchmarks to the difference in analysis time, depicted in Table 6.5, we see that it is not possible to correlate the ACRR and the analysis time.

Benchmark	ACRR	Benchmark	ACRR	Benchmark	ACRR
abp	96.03%	mceval	0.00%	pps	14.86%
actors	–	minimax	0.01%	qsort	96.06%
atoms	49.07%	msort	45.41%	ringbuf	56.38%
bchain	28.65%	nbody	28.30%	rng	55.21%
count	30.48%	pc	59.09%	sieve	30.40%
dekker	68.52%	phil	71.30%	sudoku	51.91%
fact	58.42%	phild	73.50%	trapr	56.80%
mcarlo	96.31%	pp	65.75%		

Table 6.7.: ACRR for the different benchmarks. The ACRR could not be computed for the `actors` benchmark since no abstract thread is reanalysed during the analysis of the benchmark.

6.3. Conclusion

In this chapter, we have empirically evaluated our incremental thread-modular analysis algorithm, that was presented in Chapter 4. First, in Section 6.1, we have experimentally shown soundness of IncAtom by comparing the store resulting from a program's analysis to the store returned by ModAtom. Thereafter, in Section 6.2, we have evaluated compared IncAtom to ModAtom using several metrics; we compared the analysis time and the number of states generated by the analyses using a benchmark suite containing 28 concurrent higher-order programs.

Additionally, we have evaluated visited set caching, which we proposed as a possible optimisation to further reduce the analysis time of IncAtom in Section 4.4, but found that it has a profoundly negative impact on the analysis time. Furthermore, we found the visited set reuse ratio to be zero for all benchmarks.

When comparing IncAtom to ModAtom, we found that, on a vast majority of benchmark programs, IncAtom outperforms ModAtom. IncAtom is only significantly slower on 1 out of 23 benchmarks but even then, the difference in absolute analysis time is negligible. Furthermore, the number of states generated by the incremental analysis is lower than the number of states generated by the non-incremental analysis for 6 out of 23 benchmarks. Lastly, we find that the ACRR is above 50% for 13 out of 23 benchmarks but see that there is no immediate correlation between the ACRR and the analysis time.

7 RELATED WORK

In this chapter, we present recent contributions related to the work in this dissertation, which is situated on the intersection of multiple domains within the field of static analysis and incremental computing. Our work is, among others, related to following domains: modular static analysis, static analysis for concurrent languages, abstract interpretation, abstract abstract machines and incremental computing. We now discuss related work in some of these domains. In Section 7.1, we present related work on modular analyses of concurrent programs, whereafter recent work on incremental static analysis is discussed in Section 7.2. Thereafter, in Section 7.3, we present related work on the design of abstractions. In Section 7.4, we present novel algorithmic approaches to compute the abstract collecting semantics of a program. Finally, in Section 7.5, we present recent work on the development of language semantics for combinations of concurrency constructs. We refer to Cousot & Cousot (2014) for a general overview of abstract interpretation and for an extensive overview of related literature, including work on abstraction design, optimisations and applications.

7.1. Analysis of Concurrent Programs

In this dissertation, we have presented an abstract interpreter for λ_α, a concurrent language with atoms. The analysis presented in Chapter 4 is a thread-modular analysis, an analysis that analyses the abstract threads present within a program in isolation. In general, we can classify the analyses that are developed for the analysis of concurrent languages in two categories: non-modular analyses and modular analyses, which we discuss in Sections 7.1.1 and 7.1.2 respectively. In each of these sections, we first introduce the related work and then discuss its relation with our own.

7.1.1. Non-Modular Analysis of Concurrent Programs

Martel & Gengler (2000) present a control-flow analysis for higher-order concurrent languages with synchronous inter-process communication and dynamic process creation. The presented analysis is based on the inter-process communication topology of the program, which the analysis creates by means of automata theory. Each process is represented by a finite automaton that encodes the possible orderings of executions of the synchronisation points, where synchronisation points are points in the execution of a process where it synchronises with other processes.

By computing the product automaton of the obtained automata, Martel & Gengler obtain an approximation of the way in which processes in the program synchronise. By using the order of synchronisation points available in the automata, the size of the product automaton can be reduced by the elimination of impossible synchronisation sequences, improving the precision of the analysis.

D'Osualdo et al. (2013) present a static analysis for Erlang-like concurrent programs based on abstract interpretation. Programs written in Erlang use the actor model for concurrency and each actor has a mailbox with a *First-In-First-Firable-Out* behaviour, as it is called by the authors. The analysis is focussed on safety properties, such as the maximum number of messages that may reside in an actor's mailbox. D'Osualdo et al. formalise the core language of Erlang in a language called λ_{Actor} and use this formalisation to obtain an abstract interpreter using the AAM technique of Van Horn & Might (2010). The abstract semantics so obtained is used to bootstrap the construction of an *Actor Communication System* (ACS). This ACS is an infinite-state model that allows to analyse several aspects of actors without the need to abstract actor mailboxes, that have an infinite capacity, and the number of actors that may be spawned, which also may be infinite. Like the abstract interpreter, the ACS uses *communication side effects* to track actor behaviour.

To improve the performance of control-flow analyses for higher-order languages performed by abstract interpreters, Might & Shivers (2006) present *abstract garbage collection* as a technique that lowers analysis running times and increases analysis precision. Abstract garbage collection works similar to regular garbage collection; it establishes a set of addresses in the store that are unreachable and upon reallocation, the old value is overwritten by the new value instead of joined with the new value (Might & Shivers, 2006). Stiévenart et al. (2015) apply this technique of abstract garbage collection to static analysers for concurrent languages, but find that this adaptation does not lead to significant improvements in analysis time or precision. In addition, they find that an analysis may benefit from implementing the semantics of concurrency primitives directly in the abstract interpreter rather than on top of other concurrency primitives by showing that implementing locks directly in the static analyser improves the analysis time and allows to formulate client analyses more easily than when locks are implemented on top of the `compare-and-swap` primitive, confirming a similar prior result from Stiévenart (2014).

Since the work presented by Martel & Gengler (2000), D'Osualdo et al. (2013) and Might & Shivers (2006) concerns non-modular analyses, their analyses are subject to the state explosion problem we described in Section 2.3.1. Like our work, the analysis presented by Martel & Gengler supports dynamic process creation and is applicable to higher-order languages. However, their analysis is specialised for processes with synchronous inter-process communication, whereas our analysis is specialised to applications of atoms. Yet another type of concurrency is focussed on by D'Osualdo et al., who focus on programs using the actor model with dynamic process creation. Similar to λ_ϵ, their analysis tracks the interference between threads using effects, which they call *communication side effects*. However, the exact types of effects used differ from the ones used for λ_ϵ since the actor model for concurrency has no shared memory, for example. The technique of abstract garbage collection, introduced by Might & Shivers (2006), is also applicable to the analysis of concurrent languages such as the one developed in this dissertation, as is done by Stiévenart (2014) for example.

7.1.2. Modular Analysis of Concurrent Programs

Holík et al. (2017) propose *effect summaries* for thread-modular analyses to compute the interference among threads in linear time. This is achieved by summarising the influences the threads have on shared, lock-free data structures, where an effect summary is defined as a stateless program that over-approximates the effects a thread may have on a shared heap. Using these summaries, the interference between threads can be computed. However, the proposed technique is unsound in general, for which a mechanism to test the correctness of generated summaries is provided.

Miné (2014) presents an abstract interpretation thread-modular analysis. To increase the precision of their analysis, and hence decrease the number of false positives returned by the analysis, they infer relational and history-sensitive properties of thread interference. The interference of these properties is based on a reinterpretation of rely-guarantee reasoning, an extension to Hoare logic introduced by Jones (1981). Miné interprets this reasoning technique, which is thread-modular and therefore well-suited for the analysis of multithreaded programs, as a fixpoint semantics, which is then abstracted. The resulting analysis significantly reduces the number of false positives, at the cost of an increased analysis time.

Stiévenart (2018) applies the AAM method to two different concurrent programming paradigms: actors and shared-memory threads and compare the properties of this analysis to a list of desirable properties an analysis for concurrent languages must have. It is found that the AAM method does comply with this list on all points except scalability. As an improvement of the AAM technique, Stiévenart adapts two existing optimisations to analyses of concurrent programs and again tests their behaviour with regard to the list of desirable properties by applying them to the same two concurrent programming paradigms. A first analysis design technique, called MacroConc, uses Agha et al. (1997)'s technique of *macro-stepping*, which means each process in analysed until completion or to the point where a potentially interfering operation has been performed by the process (Stiévenart, 2018). The idea behind such macro-steps is the fact that not all process interleavings need to be accounted for explicitly by the analysis, since some of them are deemed to produce the same result, e.g., when processes are not mutating shared state, it is of no importance which process steps first. Stiévenart finds that although MacroConc results in a lower runtime on most benchmarks, its worst-case time complexity remains exponential. The second analysis technique, called ModConc, applies the concept of modular static program analysis to analyses for concurrent languages. ModConc consists out of an inter-process analysis and an intra-process analysis and has all desirable properties identified by Stiévenart as the analysis has a time-complexity that is now linear in the number of communication effects generated and hence the analysis scales well to programs with a large number of processes.

The three analyses just presented all are thread-modular analyses, like the analysis presented in this dissertation. The effect summaries of Holík et al. (2017) seem to be directly applicable to our own analysis since their analysis is applicable to lock-free data structures and atoms are such data structures. In fact, Holík et al. illustrate their technique using `compare-and-set!`, a function we have formalised as part of λ_α. Although Miné (2014) also uses abstract interpretation as we do, he used a different approach to obtain the abstract semantics of a program, based on rely-guarantee reasoning. The work of Stiévenart (2018) is very closely related to the work in this dissertation since ModAtom, our non-incremental thread-modular analysis, is built according to the ModConc design method and therefore consists out of an intra-process analysis and an intra-process analysis (see sections 4.2 and 4.3).

7.2. Incremental Static Analysis

Recent work on incremental static analysis is performed by Van Es (2017) and Van Es et al. (2017), who study the incrementalisation of AAM-based abstract interpreters with regard to changes in the input program. Van Es (2017) proposes *state invalidation* as a naive technique to incrementalise AAM-based abstract interpreters. State invalidation tracks the dependencies between the input program's abstract syntax tree (AST) and the state graph outputted by the analysis. Upon a change in the program AST, the state transitions (edges) going out of affected states (vertices) in the state graph impacted by this change are invalidated and the analysis is restarted from the affected states onwards. The technique allows reuse of existing states in the graph, that is, the computation may reach states that were already present in the state graph and these reached states, as well as their transitive successors, then become part of the updated result. Unfortunately, Van Es (2017) finds that the location of a change in the source code has a significant influence on the gains made by the use of state invalidation and that this technique results in limited reuse of existing states, since often there are small differences between already existing states and newly computed states, hampering state reuse. To further improve the incremental technique, *state adaptation* is presented. State adaptation results in a significantly lower number of states that need to be recomputed as it not only allows reuse of states that are identical but also of states that are *similar* in the sense that the difference between the states has no impact on the computed successor states. To this end, specific rules defining similarity are presented.

Van Es (2017) also presents a categorisation of analysis incrementalisation techniques. On one hand, *manual* incrementalisation techniques perform the incrementalisation of the analysis ad hoc, allowing to adapt the incrementalisation algorithm to the specific needs of the analysis. On the other hand, *automatic* incrementalisation techniques require the analysis to be implemented in a specific language of on top of a framework. In this case, the language or framework in which the analysis is implemented takes care of the incrementalisation of the analysis.

Tripp et al. (2013) present *Andromeda*, a framework that uses abstract interpretation to perform a taint analysis, an analysis that verifies whether a program handles untrusted input or confidential data in a secure way. Andromeda works in a demand-driven way, that is, it only computes specific parts of information that are requested. The analysis performed by Andromeda is incremental and falls in the category of manual incrementalisation techniques. The incremental analysis performed by Andromeda is built on a change-impact analysis that computes the part of the analysis result that needs to be updated as precisely as possible. Tripp et al. (2013) find that their incremental approach results in fast updated results, producing updated results more than 100 times faster compared to a complete reanalysis on some benchmarks.

An example of a framework for automatic incremental program analysis is *IncA*, a recent framework presented by Szabó et al. (2016). The IncA framework provides a domain-specific language for developers to write their analysis in, as well as for the specification of the lattices to be used. The framework represents the program under analysis as a graph pattern that is built on top of its abstract syntax tree. To perform an analysis, the analysis developer writes an analysis in the provided domain-specific language, which is then translated to a graph pattern by IncA. To perform the analysis, IncA uses an incremental engine for graph pattern matching. Upon a change in the program's abstract syntax tree, the graph patterns representing the program change and the incremental engine for graph pattern matching computes the updated result. The pattern matching engine has been adapted by the introduction of $DRED_L$, a new algorithm to incrementally solve Datalog rules that use recursive aggregation over lattices, also presented by Szabó et al..

Nichols et al. (2019) present an incremental static analysis for JavaScript. They find that existing work is limited by two assumptions – the presence of a readily available control-flow graph and an easily computable syntax mapping between different versions of the program – making them inapplicable to dynamic languages such as JavaScript. The presented incremental analysis is based on a technique called *fixpoint reuse*, which does not rely on these assumptions and hence is applicable to JavaScript. The technique is based on abstract interpretation and consists of three steps. First, the old and new version of the program are syntactically compared and a mapping between corresponding program points is generated. Second, this mapping is used to transfer abstract states from the prior result to the new result. Third, the analysis is restarted using the transferred states. During this analysis, every state must be visited to ensure soundness. The first step in the process is important since it influences the precision of the updated result, as well as the time needed to generate the incremental result. Nichols et al. identify a trade-off between precision and analysis speed resulting from this first step. On average across the benchmarks used by Nichols et al., the incremental computation using the fixpoint reuse technique takes less than half the time needed to fully compute the new result from scratch. However, the transfer of abstract states from the old solution to the new solution in the second step of the algorithm may have a small negative impact on precision.

Liu et al. (2019) present *IPA*, an algorithm for parallel incremental points-to analyses that scales well to large codebases without losing precision and claim it is the first such algorithm that is both incremental and parallel. IPA can handle source-code changes that result in a modified control-flow graph and it is based on two new observations that allow the avoidance of unnecessary computations. First, Liu et al. find that to handle a code deletion, no global reachability analysis needs to be performed. Second, they find that their novel incremental analysis has a *change idempotency property* that allows the incremental analysis to be parallelised easily. Liu et al. find that IPA is two orders of magnitude faster than some other incremental analyses and up to five orders of magnitude faster than performing an entire reanalysis from scratch on average across the DaCapo-9.12 benchmarks. However, IPA may require a significant amount of memory (up to 140 GB).

In this section, we have presented related work on incremental static analysis. There is however a key difference between the incrementalisation performed by the analysis frameworks discussed in this section and the incrementalisation approach discussed in this dissertation. The incremental analyses presented here are incremental with regard to changes in the program that is analysed. On the contrary, our incremental analysis incrementally computes partial results but is not incremental with regard to changes in the program under analysis. Hence, we focus on a different aspect of incrementality than prevalent literature. We are unaware of any other work presenting incremental static analyses that incrementally compute results irrespective of changes to the program, like the technique presented in this dissertation.

7.3. Improved Abstractions

The abstract machines obtained by the use of the AAM technique are finite-state machines. Johnson & Van Horn (2014) state that this approach has a negative impact on the precision of control-flow analyses for languages with dynamic features, such as first-class continuations and closures, since a static analyser that is a finite state machine is imprecise when it has to reason about the return stack that is built during the execution of a program; it cannot determine exactly where a function call returns. As a solution, Johnson & Van Horn extend the AAM technique, resulting in a general technique, called *Abstracting Abstract Control* (AAC), for the creation of abstract interpreters that are based on pushdown automata rather than finite-state machines. Such pushdown automata have an infinite stack through which calls and returns can be matched exactly, whereas this is not possible using finite-state machines. The AAC technique differs from the AAM technique in that continuation frames are no longer allocated in the value store, but in a separate continuation store indexed by continuation addresses consisting out of the expression to be evaluated, the environment, the value store and a timestamp. These components make up the calling context of the given expression. Hence, if two continuation frames are stored at the same address, this means the function is called twice in exactly the same context. The function will still return to both call sites due to nondeterminism, but this is correct and hence, the analysis correctly predicts the control flow of the application.

As part of their AAC technique, Johnson & Van Horn (2014) introduce the use of a separate continuation store Ξ. Inspired by this concept, our formalisation of λ_α, presented in Chapter 3, also differentiates between a value store σ and continuation store Ξ. The use of two separate stores allows to further fine-tune the precision of the analysis and, for example, to enables exact matching of calls and returns. For this reason, we have defined the (abstract) allocation functions $alloc$ and $palloc$ to allocate value addresses and continuation addresses respectively (see Section 3.1.2).

7.4. Algorithmic Optimisations

In this section, we present some recent contributions presenting techniques to speed up the static analysis of programs by altering the design of the fixed-point computation. The incremental algorithm presented in Section 4.3 of this dissertation also is such a technique: to speed up the computation of the fixed-point, the intra-process analysis in Algorithm 3 is made incremental. We now first introduce the related work, before discussing its relation with our own.

Wei et al. (2018) propose to optimise abstract interpreters by using multi-stage programming. They propose *staged abstract interpreters*, abstract interpreters that are specialised for a given program and show that such staged abstract interpreters can result in significantly reduced analysis times. Wei et al. also find that applying this technique to an open program, that is, a program containing unbound variables, results in a sound modular analysis when the free program variables are treated as dynamic inputs, resulting in a partial analysis result that can later be reused and composed with other partial analysis results.

Algorithms to perform a static analysis often are work list based and require to maintain a set of visited states to avoid duplication of work and to ensure termination. Nicolay et al. (2019) state that the use of such a visited set incurs a significant overhead since set membership needs to be tested for every state that is removed from the work list. To resolve this overhead Nicolay et al. propose *MODF*, an abstract-interpreter based analysis that avoids the use of a visited set and that is modular by design on the granularity of function calls. Each function execution is analysed in isolation and until completion while the abstract interpreter generates the required effects, that is, while tracking the function's behaviour by identifying which variables are read from and written to and which other functions are called. The analysis also caches the return values of the function executions. Based on the generated effects, the analysis infers which functions to analyse next, i.e., the analysis is driven by the effects it generates. For example, when the return value of a function execution changes, function executions relying on that value need reanalysis. Because function executions are analysed as a whole and in isolation, MODF avoids the need for maintaining a set of visited states; every function execution that is in the work list needs to be analysed. The analysis is applicable to programs featuring higher-order functions and mutable state (Nicolay et al., 2019).

Germane et al. (2019) present a 0CFA *demand control-flow analysis*. Regular, exhaustive, control-flow analyses calculate information for every variable in a program. However, this may be unnecessary as often only particular program variables are of interest. To alleviate this issue, the presented demand control-flow analysis allows for selective computation of information, that is, to only compute control-flow information for specific variables. The presented analysis works by combining two *modes* of operation. On the one hand, the *evaluation mode*, which is standardly used by regular (abstract) interpreters, establishes the value of an expression by evaluating it (abstractly). On the other hand, *tracing mode* starts from a value (stored within a variable) and looks for the set of expressions that evaluate to that value. As such, evaluation and tracing mode can be considered as counterparts as they work in the opposite direction. The interplay of these two modes allows to calculate the required information only for specific program variables, instead of having to calculate this information for every variable in the program. Germane et al. find that their demand 0CFA control-flow analysis is equally precise to an exhaustive 0CFA control flow analysis while having a good performance.

The modular analysis presented by Wei et al. (2018) differs from the thread-modular analysis used throughout this dissertation in that the analysis is modular with regard to static source code components since modularity is obtained by considering unbound variables as dynamic inputs. A thread-modular analysis, such as used throughout this dissertations, is modular in the processes to be analysed and allows for complex features such as dynamic thread creation. Hence, the division between modules is dynamic since, for example, the number of different abstract threads may be unknown in advance. On the other hand, the division used by Wei et al. is static and based on the program's source code. The same remark can be made for the analysis presented by Nicolay et al. (2019) as their module division is made based on functions, whereas the one presented in this dissertation is thread-based. However, the algorithms presented in this dissertation also make use of a visited set. The MODF technique seems not directly transferable to thread-based modularity however: since MODF is used to analyse sequential programs, it does not have to take into account thread interleavings. As a result, the technique may need serious modifications to be usable for the analysis of concurrent programs. The same remark may be true for the analysis developed by Germane et al. (2019), whose demand control-flow analysis can be used to compute information for specific variables. In multithreaded programs, however, the values stored in variables may be affected by the different threads in the program. For this reason, it seems that the technique of Germane et al. is not suited for the analysis of concurrent programs.

7.5. Semantics for Combined Concurrent Programming Constructs

Static analyses for concurrent programs are often used to detect concurrency bugs, such as race conditions. To prevent concurrency bugs, programming languages offer concurrency constructs such as atoms, futures and software-transactional memory (STM), which provide guarantees on program behaviour by providing functions to handle access to shared resources and imposing restrictions on the use of these resources, for example. The guarantees provided by such concurrency constructs are expressed by their semantics. For example, the formal semantics of λ_α expresses the guarantees made by atoms. Based on this semantics, we have built an abstract interpreter for λ_α. Hence, knowing how concurrency constructs behave not only allows using them while programming but also makes program analysis of concurrent languages possible.

Swalens et al. (2014) find that although multiple concurrency constructs are often combined, such a combination can lead to a violation of the guarantees made by the individual constructs. Using these findings, Swalens studies how different constructs can be combined and presents *Chocola*, a language unifying actors, futures and transactions. New in this language is that Chocola also provides guarantees on the behaviour of these constructs when used in combination: apart from the definition of the semantics of every concurrency construct in separation, Chocola also defines the semantics of their combinations, such as *transactional actors* and *transactional futures*.

The explicit semantics for combinations of concurrency constructs defined in the work of Swalens (2018) allows the construction of abstract interpreters that can handle these combinations while knowing exactly how the combined constructs behave. This is of course of crucial importance during the development of sound abstract interpreters for concurrent languages.

7.6. Conclusion

In this chapter, we have discussed work related to the work in this dissertation. Our work is situated on the intersection of multiple domains within the field of static analysis. Recent work on concurrent modular analyses coincides with our work by supporting dynamic thread creation and by using effects to formalise thread interference. Also, the related work on thread-modular analyses that was discussed is closely related. The incrementalisation technique introduced in this thesis differs from other work on incremental static analysis as our technique is not incremental with regard to source code changes in the program under analysis. On the contrary, we incrementally compute the analysis result for a given version of the program. Hence, the challenges discussed in this dissertation are not directly linkable to other work concerning incremental static program analysis. We concluded the chapter by presenting related work introducing improved abstractions and algorithms but found that the work on the latter category may not immediately be applicable to the analysis of concurrent programs.

8 CONCLUSION

In this dissertation, we have presented two main contributions. Our first contribution is the application of a thread-modular analysis to a concurrent higher-order language with futures, atoms and dynamic thread creation. To this end, we have built an abstract interpreter using the AAM technique of Van Horn & Might and formalised this interpreter by means of an abstract PCESK machine. Thereafter, we modified the algorithm of Stiévenart to make our abstract interpreter thread-modular according to the principles laid out by Cousot & Cousot. This way, ModAtom is obtained.

ModAtom discards all results for a given abstract thread upon reanalysis of the thread, that is, the algorithm starts the reanalysis of every abstract thread from scratch. We find that this may not be an optimal solution since only part of the analysis results for that abstract thread may need to be recomputed, i.e., the ModAtom algorithm may needlessly duplicate work, resulting in increased analysis times. To avoid this unnecessary duplication of work, we propose to incrementally compute the analysis results of the different abstract threads in a program, thereby only recomputing the parts of the analysis results that are required to be updated and reusing parts of the analysis results that do not need to be updated. To incrementalise ModAtom, we have developed a more fine-grained tracking mechanism for effects, allowing the intra-process analysis phase to be restarted from a specific abstract state. This results in IncAtom, a new incremental thread-modular analysis algorithm. In addition, we have presented intra-process analysis abortion and visited set caching as possible optimisations to further decrease the analysis time, respectively of IncAtom and of ModAtom and IncAtom. We have proven that both ModAtom and IncAtom are guaranteed to terminate, which is a critical requirement for a static analyser.

In our evaluation, we have experimentally shown that IncAtom is sound by comparing the abstract store resulting from the analysis of a program to the one produced by ModAtom. Furthermore, our experiments have shown that, on 21 out of 23 benchmarks, the use of our incremental algorithm results in a significantly lower average analysis time, with reductions of the average analysis time up to 63%. Also, there was a reduction of the number of states in the resulting abstract state graph for 6 out of 23 benchmarks, meaning that our analysis is more precise than ModAtom on these benchmarks as the resulting abstract state graph contains fewer spurious paths. Using the ACRR, we are able to measure how much of a previously computed result IncAtom reuses. We find that the value of the ACRR ranges up to 96% and is above 50% for 13 out of 23 benchmark programs. However, there is no immediate correlation between the ACRR and the analysis time. Lastly, we found that the incorporation of visited set caching into

IncAtom does not reduce the analysis time on any benchmark. On the contrary, we find the visited set reuse ratio to be zero for all benchmarks and note a significant increase in analysis time of up to a factor 94.

8.1. Future Work

In this final section of this dissertation, we present four possible directions for future research. A first improvement to our current research is to further incrementalise IncAtom with regard to changes in the source code of the program under analysis, as discussed in Section 8.1.1. A second improvement, discussed in Section 8.1.2, concerns the filtering pass that needs to be applied after the analysis of a program. Third, we find that it may be possible to introduce effect summaries into our work, as is discussed in Section 8.1.3. Finally, we think it is recommended to perform a more extensive evaluation of our incremental algorithm, which we discuss in Section 8.1.4.

8.1.1. Handling Source Code Changes

In this dissertation, we have presented an algorithm to incrementally compute the analysis results for a given program. Upon reanalysis of an abstract thread, our algorithm is able to reuse as much as possible of previously computed results. However, our algorithm is not incremental with regard to changes in the program under analysis. Upon a change in the program, no matter how small, a new analysis must be started from scratch. Therefore, a further improvement to our work would be to further incrementalise IncAtom with regard to changes to the source program under analysis. As a result, both the intra-process analysis phase as well as the inter-process analysis phases will be rendered incremental.

A change to the program under analysis may also cause previously generated effects to become invalid. For example, a programmer may remove a line of code wherein a future is dereferenced. When these effects are preserved, they may cause a loss in precision and imprecision may be built-up upon several successive reanalyses after a program change. To remedy this, a change impact analysis will have to be introduced that is able to infer which effects have become invalid. These effects can then be removed to avoid building-up of imprecision.

8.1.2. State Filtering

At the end of every analysis, both ModAtom and IncAtom require a filtering pass to remove states that are not reachable from the initial state of any abstract thread. It is clear that such filtering operation introduces an overhead to the analysis. In future work, we can explore techniques to avoid this filtering pass. A possible technique could be to immediately remove all states that become disconnected during the analysis. This will, however, require a mechanism to track which states are still reachable. Also, it must be investigated whether this immediate removal of states leads to an increased or decreased overhead.

8.1.3. Introducing Effect Summaries

To compute thread interference, both ModAtom and IncAtom continuously track the effects that are generated by the intra-process analysis of a thread. This is needed to ensure soundness of the analysis, which must account for all possible thread interleavings. In Section 7.1.2, we discussed effect summaries, developed by Holík et al. (2017) to compute thread interference in linear time and we stated that this technique seems to be directly applicable to our own analysis. Therefore, it would be interesting to see how effect summaries can be used in combination with our incremental analysis algorithm. This way, we may be able to avoid the need to track effects continuously, as is currently done by IncAtom.

8.1.4. Extended Evaluation

In Chapter 6, we have presented an evaluation of IncAtom. In this section, we discuss how this evaluation may further be improved upon.

First, in Section 6.1, we have verified soundness of the analysis by comparing it to its non-incremental counterpart, ModAtom. However, it may be more interesting to compare the results of IncAtom immediately to the results of a concrete interpreter as this may produce more detailed soundness results. During such a soundness analysis, it must, however, be taken into account that the result of a concrete interpreter may be nondeterministic due to the parallelism in λ_α programs.

Second, in Section 6.2, we have studied the behaviour of IncAtom according to several metrics. However, the reason why some results were obtained remains unclear. For example, we were unable to pinpoint the exact reason for the dramatic increase in analysis time when the visited set caching optimisation was enabled. Further analysis steps may need to be taken to fully clarify our results.

Finally, throughout our evaluation, we have configured Scala-AM to use thread identifiers consisting of an expression and a timestamp, name addresses, no context sensitivity and the type lattice. It may, however, be interesting to see how our incremental algorithm behaves when using a different configuration of the framework. For example, it may be interesting to investigate the effect of enabling context sensitivity or to study the impact of alternative definitions for $\widehat{alloc}$, $\widehat{kalloc}$ and $\widehat{palloc}$.

8.2. Concluding Remarks

In this dissertation, we have presented two additions to the state of the art in static analysis of concurrent languages. Our first contribution is the extension of the application domain of AAM-based analyses to concurrent languages with atoms. Thereafter, we have presented IncAtom, an incremental thread-modular algorithm. IncAtom incrementally computes the analysis results for the different abstract threads and results in lower analysis times and smaller abstract state graphs, indicating that the analysis has a better scalability and a higher precision. This allows IncAtom to analyse more substantial programs while producing more accurate results.

A

ADDITIONAL TRANSITION RULES FOR λ_ϕ

In this appendix, we present the concrete and abstract transition rules for the predicates and cancellation functions of λ_ϕ that have been omitted in Chapter 3. The concrete transition rules for these functions are depicted in Figure A.1 and the corresponding abstract transition rules are shown in Figure A.2. Again, each rule of the transition relation is annotated with the thread identifier p of the thread performing the transition ($\leadsto_p$):

- Rules FUTURE-CANCEL-T, FUTURE-CANCEL-F1 and FUTURE-CANCEL-F2 specify how future cancellation works. A future can only be cancelled when it has not been cancelled before and when it has not reached the end of its computation already. In this case, the thread map is updated and the future's state is replaced by **cancelled**. The premises checking these conditions are indicated in red.
- Rules ISFUTURE-T, ISFUTURE-F, FUTURE-DONE-T, FUTURE-DONE-F, FUTURE-CANCELLED-T and FUTURE-CANCELLED-F describe evaluation rules for the predicates `future?`, `future-done?` and `future-cancelled?` respectively. Two rules are needed for every predicate, since each of them may evaluate to true or false. The differences within every pair of corresponding rules are indicated in red.

λ_ϕ

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(future? } ae\texttt{)}, \rho), k\rangle \qquad \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p')}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#t}), k\rangle], \sigma, \Xi} \text{ ISFUTURE-T}$$

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(future? } ae\texttt{)}, \rho), k\rangle \qquad \rho, \sigma \vdash ae \Downarrow v \qquad v \neq \mathbf{fut}(p')}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#f}), k\rangle], \sigma, \Xi} \text{ ISFUTURE-F}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-done? } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p') \qquad \pi(p') = \langle \mathbf{val}(v), k_0\rangle\end{array}}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#t}), k\rangle], \sigma, \Xi} \text{ FUTURE-DONE-T}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-done? } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p') \qquad \pi(p') \neq \langle \mathbf{val}(v), k_0\rangle\end{array}}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#f}), k\rangle], \sigma, \Xi} \text{ FUTURE-DONE-F}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-cancel } ae\texttt{)}, \rho), k\rangle \qquad \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p') \\ \pi(p') = \varsigma \qquad \varsigma \neq \langle \mathbf{val}(v), k_0\rangle \qquad \varsigma \neq \mathbf{cancelled}\end{array}}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#t}), k\rangle, p' \mapsto \mathbf{cancelled}], \sigma, \Xi} \text{ FUTURE-CANCEL-T}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-cancel } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p') \qquad \pi(p') = \langle \mathbf{val}(v), k_0\rangle\end{array}}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#f}), k\rangle], \sigma, \Xi} \text{ FUTURE-CANCEL-F1}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-cancel } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p') \qquad \pi(p') = \mathbf{cancelled}\end{array}}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#f}), k\rangle], \sigma, \Xi} \text{ FUTURE-CANCEL-F2}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-cancelled? } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p') \qquad \pi(p') = \mathbf{cancelled}\end{array}}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#t}), k\rangle], \sigma, \Xi} \text{ FUTURE-CANCELLED-T}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(future-cancelled? } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{fut}(p') \qquad \pi(p') \neq \mathbf{cancelled}\end{array}}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\texttt{\#f}), k\rangle], \sigma, \Xi} \text{ FUTURE-CANCELLED-F}$$

Figure A.1.: Concurrent transition rules for λ_ϕ (continued).

λ_ϕ

$$\frac{\langle \mathbf{ev}(\texttt{(future? }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \qquad \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p')}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#t}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{isfuture-t}$$

$$\frac{\langle \mathbf{ev}(\texttt{(future? }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \qquad \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \hat v \qquad \hat v \neq \mathbf{fut}(\hat p')}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#f}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{isfuture-f}$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-done? }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \\ \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p') \qquad \langle \mathbf{val}(\hat v),\widehat{k}_0\rangle \in \hat\pi(\hat p')\end{array}}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#t}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{future-done-t}$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-done? }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \\ \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p') \qquad \hat\varsigma \in \hat\pi(\hat p') \qquad \hat\varsigma \neq \langle \mathbf{val}(\hat v),\widehat{k}_0\rangle\end{array}}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#f}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{future-done-f}$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-cancel }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \\ \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p') \qquad \hat\varsigma \in \hat\pi(\hat p') \qquad \hat\varsigma \neq \langle \mathbf{val}(\hat v),\widehat{k}_0\rangle \hat\varsigma \neq \mathbf{cancelled}\end{array}}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#t}),\widehat{k}\rangle, \hat p' \mapsto \mathbf{cancelled}],\hat\sigma,\widehat{\Xi}}\ \textsc{future-cancel-t}$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-cancel }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \\ \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p') \qquad \langle \mathbf{val}(\hat v),\widehat{k}_0\rangle \in \hat\pi(\hat p')\end{array}}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#f}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{future-cancel-f}1$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-cancel }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \\ \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p') \qquad \mathbf{cancelled} \in \hat\pi(\hat p')\end{array}}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#f}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{future-cancel-f}2$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-cancelled? }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \\ \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p') \qquad \mathbf{cancelled} \in \hat\pi(\hat p')\end{array}}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#t}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{future-cancelled-t}$$

$$\frac{\begin{array}{c}\langle \mathbf{ev}(\texttt{(future-cancelled? }\ ae\texttt{)},\hat\rho),\widehat{k}\rangle \in \hat\pi(\hat p) \\ \hat\rho,\hat\sigma \vdash ae \mathrel{\widehat{\Downarrow}} \mathbf{fut}(\hat p') \qquad \hat\varsigma \in \hat\pi(\hat p') \qquad \hat\varsigma \neq \mathbf{cancelled}\end{array}}{\hat\pi,\hat\sigma,\widehat{\Xi} \mathrel{\widehat{\leadsto}_{\hat p}} \hat\pi \sqcup [\hat p \mapsto \langle \mathbf{val}(\texttt{\#f}),\widehat{k}\rangle],\hat\sigma,\widehat{\Xi}}\ \textsc{future-cancelled-f}$$

Figure A.2.: Abstract concurrent transition rules for λ_ϕ (continued).

B

PROOFS

In this appendix, we present the full proofs of the lemmas and theorems presented throughout this dissertation.

B.1. Termination of the Non-Incremental Modular Analysis Algorithm

In this section, we show that our formulation of the non-incremental modular analysis (Algorithm 1), which relies on two alternating phases, terminates.

Lemma 1. *Consider an abstract value store $\hat{\sigma}$. After a finite number of updates of $\hat{\sigma}$, a fixed-point is reached.*

Proof. We prove Lemma 1 by proving the statement for a single abstract address $\hat{a}$ in $\hat{\sigma}$. If the statement holds for a single abstract address $\hat{a}$, it is clear that it also holds for the entire abstract store $\hat{\sigma}$.

By definition, an abstract value store $\hat{\sigma}$ maps abstract addresses to sets of abstract values (see Section 3.1.4). The sets of abstract values form a set lattice of which the join operator $\sqcup$ is defined as set union $\cup$.

Whenever an address $\hat{a}$ in the store is updated, according to the abstract semantics of λ_α, the set of abstract values residing at that address in the store is joined together with a singleton set containing the new abstract value. As a result, the set of values related to $\hat{a}$ may grow, or, if the value was already included, remains the same size. By construction, the set of abstract values is finite. Assume $n = |\widehat{Val}|$, then only n updates to the abstract address $\hat{a}$ can result in a modification of $\hat{\sigma}$. $\square$

Lemma 2. *Consider an abstract continuation store $\widehat{\Xi}$. After a finite number of updates of $\widehat{\Xi}$, a fixed-point is reached.*

Proof. Similar to the proof of Lemma 1 and based on the given that, by construction, the set of abstract continuation frames is finite. $\square$

Lemma 3. *The intra-process analysis of an abstract thread in Algorithm 1 terminates.*

Proof. The intra-process analysis of an abstract thread $\hat{p}$ starts from the abstract initial states related to $\hat{p}$ by $\hat{\pi}$. The transition function iteratively generates abstract successor states, starting from the given abstract initial states. By construction, the number of abstract states is finite and the visited set prevents states from being stepped multiple times. Therefore, only a finite number of states can be generated by the transition function.

Upon a change in the (continuation) store, the visited set is emptied and previously stepped states may be stepped again by the transition function. By Lemmas 1 and 2, the visited set can only be emptied a finite number of times.

Since the number of abstract states is finite and the visited set can only be emptied a finite number of times, the analysis must reach a point in which the visited set contains all generated states. At this point, the intra-process analysis reaches a fixed-point and terminates. □

Theorem 1. *The process-modular analysis of a program e in Algorithm 1 terminates.*

Proof. To prove that the analysis of the program e terminates, it suffices to prove that the inter-process analysis of Algorithm 1 terminates.

The inter-process analysis starts with the thread identifier of the main thread in its work list. To show the inter-process analysis terminates, we have to prove that the intra-process analysis need only be called a finite number of times.

The intra-process analysis of a thread is called when it is a newly created thread, the thread reads a modified return value of another thread or there is a read-write or write-write conflict with another thread.

Since, by construction, the number of abstract thread identifiers and the number of states are finite, the number of abstract threads that can be created is finite. By design, the return value of such an abstract thread is part of a lattice and can only be updated a finite number of times. Also, the number of reanalyses caused by read-write and write-write conflicts is finite, since the addresses in the value store may only be updated a finite number of times (Lemma 1). Hence, the number of times the intra-process analysis is executed is finite. Lemma 3 gives us that each intra-process analysis terminates. As a result, the inter-process analysis must terminate as well. □

BIBLIOGRAPHY

Agha, G., Mason, I., Smith, S., & L. Talcott, C. (1997). A Foundation for Actor Computation. *Journal of Functional Programming*, *7*(1), 1–72. doi: 10.1017/S095679689700261X

Chadwick, J. E. (2013). How a Smarter Grid Could Have Prevented the 2003 U.S. Cascading Blackout. In *2013 IEEE Power and Energy Conference at Illinois, PECI 2013, Champaign, IL, USA, February 22-23, 2013* (pp. 65–71). New York, NY, USA: IEEE. doi: 10.1109/PECI.2013.6506036

Cousot, P., & Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In R. M. Graham, M. A. Harrison, & R. Sethi (Eds.), *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1977, Los Angeles, CA, USA, January 17-19, 1977* (pp. 238–252). New York, NY, USA: ACM. doi: 10.1145/512950.512973

Cousot, P., & Cousot, R. (2002). Modular Static Program Analysis. In R. N. Horspool (Ed.), *Proceedings of the 11th International Conference on Compiler Construction, CC 2002, Grenoble, France, April 8-12, 2002* (pp. 159–178). Berlin, Heidelberg, Germany: Springer. doi: 10.1007/3-540-45937-5_13

Cousot, P., & Cousot, R. (2014). Abstract Interpretation: Past, Present and Future. In T. A. Henzinger & D. Miller (Eds.), *Proceedings of the Joint Meeting of the 23th EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014, Vienna, Austria, July 14-18, 2014* (pp. 1–10). New York, NY, USA: ACM Press. doi: 10.1145/2603088.2603165

D'Osualdo, E., Kochems, J., & Ong, C.-H. L. (2013). Automatic Verification of Erlang-Style Concurrency. In F. Logozzo & M. Fähndrich (Eds.), *Proceedings of the 20th International Symposium on Static Analysis, SAS 2013, Seattle, WA, USA, June 20-22, 2013* (pp. 454–476). Berlin, Heidelberg, Germany: Springer. doi: 10.1007/978-3-642-38856-9_24

Etiemble, D. (2018). 45-year CPU evolution: One law and two equations. *Computing Research Repository*, *abs/1803.00254*. Retrieved from `http://arxiv.org/abs/1803.00254`

Felleisen, M., & Friedman, D. P. (1987). A Calculus for Assignments in Higher-Order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1987, Munich, West Germany, January 21-23, 1987* (pp. 314–325). New York, NY, USA: ACM. doi: 10.1145/41625.41654

Flanagan, C., Freund, S. N., & Qadeer, S. (2002). Thread-Modular Verification for Shared-Memory Programs. In D. L. Métayer (Ed.), *Proceedings of the 11th European Symposium on Programming Programming Languages and Systems, Grenoble, France, April 8-12, 2002* (pp. 262–277). Berlin, Heidelberg, Germany: Springer. doi: 10.1007/3-540-45927-8_19

Flanagan, C., Sabry, A., Duba, B. F., & Felleisen, M. (1993). The Essence of Compiling with Continuations. In R. Cartwright (Ed.), *Proceedings of the 14th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1993, Albuquerque, NM, USA, June 23-25, 1993* (pp. 237–247). New York, NY, USA: ACM. doi: 10.1145/155090.155113

Germane, K., McCarthy, J., Adams, M. D., & Might, M. (2019). Demand Control-Flow Analysis. In C. Enea & R. Piskac (Eds.), *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2019, Cascais, Portugal, January 13-15, 2019* (pp. 226–246). Cham, Switzerland: Springer International Publishing. doi: 10.1007/978-3-030-11245-5_11

Gilray, T., Adams, M. D., & Might, M. (2016). Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-Flow Analysis. In J. Garrigue, G. Keller, & E. Sumii (Eds.), *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016* (pp. 407–420). New York, NY, USA: ACM. doi: 10.1145/2951913.2951936

Godefroid, P. (1995). *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem* (Doctoral dissertation). Université de Liège, Liège, Belgium.

Holík, L., Meyer, R., Vojnar, T., & Wolff, S. (2017). Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic. In F. Ranzato (Ed.), *Proceedings of the 24th International Symposium on Static Analysis, SAS 2017, New York, NY, USA, August 30 - September 1, 2017* (pp. 169–191). Cham, Switzerland: Springer International Publishing. doi: 10.1007/978-3-319-66706-5_9

Johnson, J. I., & Van Horn, D. (2014). Abstracting Abstract Control. *ACM SIGPLAN Notices*, *50*(2), 11–22. doi: 10.1145/2661088.2661098

Jones, C. B. (1981). *Development Methods for Computer Programs including a Notion of Interference* (Doctoral dissertation). Oxford University, Oxford, England, UK. (Printed as: Programming Research Group, Technical Monograph 25)

Liu, B., Huang, J., & Rauchwerger, L. (2019). Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems*, *41*(1), 6:1–6:31.

Martel, M., & Gengler, M. (2000). Communication Topology Analysis for Concurrent Programs. In K. Havelund, J. Penix, & W. Visser (Eds.), *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification, Stanford, CA, USA, August 30 - September 1, 2000* (pp. 265–286). Berlin, Heidelberg, Germany: Springer. doi: 10.1007/10722468

Might, M., & Shivers, O. (2006). Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting. In J. H. Reppy & J. L. Lawall (Eds.), *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, OR, USA, September 16-21, 2006* (pp. 13–25). New York, NY, USA: ACM. doi: 10.1145/1159803.1159807

Might, M., & Van Horn, D. (2011). A family of abstract interpretations for static analysis of concurrent higher-order programs. In E. Yahav (Ed.), *Proceedings of the 18th international static analysis symposium, SAS 2011, Venice, Italy, September 14-16, 2011* (pp. 180–197). Berlin, Heidelberg, Germany: Springer. doi: 10.1007/978-3-642-23702-7_16

Miné, A. (2014). Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In K. L. McMillan & X. Rival (Eds.), *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2014, San Diego, CA, USA, January 19-21,* *2014* (pp. 39–58). Berlin, Heidelberg, Germany: Springer. doi: 10.1007/978-3-642-54013-4_3

Nichols, L., Emre, M., & Hardekopf, B. (2019). *Fixpoint Reuse for Incremental JavaScript Analysis.* Retrieved 2019-05-09, from `https://www.cs.ucsb.edu/research/tech-reports/2019-02` (Contidionally accepted for the 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2019, Phoenix, AZ, USA, June 22-26, 2019.)

Nicolay, J., Stiévenart, Q., De Meuter, W., & De Roover, C. (2019). Effect-Driven Flow Analysis. In C. Enea & R. Piskac (Eds.), *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2019, Cascais, Portugal, January 13-15, 2019* (pp. 247–274). Cham, Switzerland: Springer International Publishing. doi: 10.1007/978-3-030-11245-5_12

Poulsen, K. (2004). *Tracking the Blackout Bug.* Retrieved 2019-05-27, from `https://www.securityfocus.com/news/8412`

Rice, H. G. (1953). Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, *74*(2), 358–366. doi: 10.2307/1990888

Shivers, O. (1991). *Control-Flow Analysis of Higher-Order Languages* (Doctoral dissertation). Carnegie Mellon University, Pittsburgh, PA, USA.

Stiévenart, Q., Nicolay, J., De Meuter, W., & De Roover, C. (2019). A General Method for Rendering Static Analyses for Diverse Concurrency Models Modular. *Journal of Systems and Software*, *147*, 17–45. doi: 10.1016/j.jss.2018.10.001

Stiévenart, Q. (2014). *Static Analysis of Concurrency Constructs in Higher-Order Programs* (Master's thesis). Université libre de Bruxelles, Brussels, Belgium.

Stiévenart, Q. (2018). *Scalable Designs for Abstract Interpretation of Concurrent Programs: Application to Actors and Shared-Memory Multi-Threading* (Doctoral dissertation). Vrije Universiteit Brussel, Brussels, Belgium.

Stiévenart, Q., Nicolay, J., De Meuter, W., & De Roover, C. (2015). Detecting Concurrency Bugs in Higher-Order Programs through Abstract Interpretation. In M. Falaschi & E. Albert (Eds.), *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming - PPDP 2015, Siena, Italy, June 14-16, 2015* (pp. 232–243). New York, NY, USA: ACM Press. doi: 10.1145/2790449.2790530

Stiévenart, Q., Nicolay, J., De Meuter, W., & De Roover, C. (2016). Building a Modular Static Analysis Framework in Scala (Tool Paper). In A. Biboudis, M. Jonnalagedda, S. Stucki, & V. Ureche (Eds.), *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016* (pp. 105–109). New York, NY, USA: ACM Press. doi: 10.1145/2998392.3001579

Stiévenart, Q., Vandercammen, M., Meuter, W. D., & De Roover, C. (2016). Scala-AM: A Modular Static Analysis Framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016* (pp. 85–90). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/SCAM.2016.14

Swalens, J. (2018). *A Multi-Paradigm Concurrent Programming Model* (Doctoral dissertation). Vrije Universiteit Brussel, Brussels, Belgium.

Swalens, J., Marr, S., De Koster, J., & Van Cutsem, T. (2014). Towards Composable Concurrency

Abstractions. In A. F. Donaldson & V. T. Vasconcelos (Eds.), *Proceedings of the 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, April 12, 2014* (pp. 54–60). doi: 10.4204/EPTCS.155.8

Szabó, T., Erdweg, S., & Voelter, M. (2016). IncA: A DSL for the Definition of Incremental Program Analyses. In D. Lo, S. Apel, & S. Khurshid (Eds.), *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016* (pp. 320–331). New York, NY, USA: ACM. doi: 10.1145/2970276.2970298

Tripp, O., Pistoia, M., Cousot, P., Cousot, R., & Guarnieri, S. (2013). Andromeda: Accurate and Scalable Security Analysis of Web Applications. In V. Cortellessa & D. Varró (Eds.), *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013, Rome, Italy, March 16-24, 2013* (pp. 210–225). Berlin, Heidelberg, Germany: Springer. doi: 10.1007/978-3-642-37057-1_15

Van Es, N. (2017). *Incrementalizing Abstract Interpretation* (Master's thesis). Vrije Universiteit Brussel, Brussels, Belgium.

Van Es, N., Vandercammen, M., & De Roover, C. (2017). Incrementalizing Abstract Interpretation. In S. Demeyer, A. Parsai, G. Laghari, & B. van Bladel (Eds.), *Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium, BENEVOL 2017, Antwerp, Belgium, December 4-5, 2017* (pp. 31–35). Aachen, Germany: CEUR Workshop Proceedings.

Van Horn, D., & Might, M. (2010). Abstracting Abstract Machines. In P. Hudak & S. Weirich (Eds.), *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, MD, USA, September 27-29, 2010* (pp. 51–62). New York, NY, USA: ACM. doi: 10.1145/1863543.1863553

Van Horn, D., & Might, M. (2012). Systematic Abstraction of Abstract Machines. *Journal of Functional Programming*, *22*(4-5), 705–746. doi: 10.1017/S0956796812000238

Wei, G., Chen, Y., & Rompf, T. (2018). *Staged Abstract Interpreters - Fast and Modular Whole-Program Analysis via Meta-Programming.* Retrieved 2018-12-17, from `https://www.cs.purdue.edu/homes/rompf/papers/wei-preprint201811.pdf` (Preprint November 2018.)