ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Static Analysis of Concurrency Constructs in Higher-Order Programs

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil informaticien

Quentin Stiévenart

Promotor
Prof. Wolfgang De Meuter

Co-Promotor
Dr. Coen De Roover

Advisor
Jens Nicolay

Service
Software Languages Lab (VUB)

Academic year
2013 - 2014

Abstract

Detecting and debugging concurrency bugs is particularly hard. The complexity of concurrent programs is huge due to the exponential number of possible interleavings between the multiple units of execution of those programs. Having help from the computer to detect concurrency bugs — or prove their absence — can substantially reduce debugging time.

In this dissertation, we investigate the use of *abstract interpretation* to detect race conditions and deadlocks in concurrent programs. We use the PCESK machine, as described by Might and Van Horn, to compute an over-approximation of the set of states reachable by a concurrent program. We provide a number of benchmarks to analyze the influence of garbage collection and state subsumption on the performance of this machine in terms of state space size and analysis time.

We define several analyses that use the computed set of reachable states to deduce relevant properties of a concurrent program. Race condition analysis and deadlock analysis are the most notable analyses we develop. Although both these analyses are unsound and thus might miss some defects, they exhibit high precision and recall, allowing them to detect a good portion of bugs in relatively short programs.

This dissertation shows that it is possible to use the PCESK machine for building useful analyses to detect race conditions and deadlocks. The main limitation of this approach is the size of the state spaces involved in the analyses. We therefore investigate the PCESK_L machine, a variant of the PCESK machine that uses first-class locks instead of `cas` (compare-and-swap) as a synchronization primitive. We find that the PCESK_L machine leads to simplified analyses and reduced state space size, resulting in reduced analysis time. This machine is therefore more suited for analyzing longer and more complex programs.

Keywords: static analysis, concurrency, abstract interpretation, race conditions, deadlocks.

Résumé

Nom : Quentin Stiévenart
Master : Master Ingénieur Civil en Informatique
Année académique : 2013 – 2014
Titre : Static Analysis of Concurrency Constructs in Higher-Order Programs

Il est particulièrement difficile de détecter ainsi que de corriger des erreurs liées à la concurrence dans des programmes. À cause du nombre exponentiel de possibilités d'entrelacement entre plusieurs unités d'éxécution d'un programme concurrent, la complexité de ces programmes est grande. Il est possible de considérablement réduire le temps de débogage de ces programmes en utilisant des outils permettant de détecter automatiquement la présence de ces bogues — ou même de prouver leur absence.

Ce mémoire explore l'utilisation de l'*interprétation abstraite* dans le but de détecter des conditions de courses et des interblocages dans des programmes concurrents. Cela est fait en utilisant une machine de type PCESK, telle que décrite par Might et Van Horn, qui permet de sur-approximer l'ensemble des états accessibles par un programme. Est donné un ensemble de *benchmarks* qui analyse l'influence d'un ramasse-miettes (*garbage collector*) et d'une méthode de subsomption d'état sur les performances de la machine en terme de taille de l'ensemble d'états et du temps d'analyse.

Sont ensuite définies plusieurs analyses utilisant cet ensemble d'états dans le but de déduire certaines propriétés des programmes analysés. Les analyses les plus utiles développées dans ce mémoire sont une analyse des conditions de courses, ainsi qu'une analyse des interblocages. Bien que ces analyses ne soient pas exhaustives (*sound*), et risquent donc de ne pas détecter certains problèmes. leur valeurs de précision et de rappel (*recall*) sont suffisament grandes que pour détecter une proportion importante de bogues sur des programmes relativement courts.

Ce mémoire montre donc qu'il est possible d'utiliser une machine de type PCESK pour produire des analyses utiles qui détectent des conditions de courses et des interblocages. Cependant, la principale restriction de cette approche est la taille de l'ensemble d'états utilisé par les analyses. Est alors introduite la machine de type PCESK$_L$, une variante de la machine de type PCESK qui utilise des verrous en tant qu'objet de première classe comme primitive pour la synchronisation, plutôt que la primitive `cas` (*compare-and-swap*), utilisée par la machine de type PCESK. Cette variante permet de simplifier la définition des analyses ainsi que de réduire la taille de l'ensemble d'états, et donc le temps d'analyse. Il est donc aussi possible d'analyser des programmes plus longs et complexes avec cette variante.

Mots-clés : analyse statique, programmation concurrente, interprétation abstraite, condition de course, interblocage.

Acknowledgements

I am the most thankful toward my supervisors, Coen De Roover and Jens Nicolay, for their very valuable help and advices. They helped me keeping focus on the important parts of this work throughout this year. Producing this work would not have been possible without their numerous advices.

I also express my sincere gratitude towards my promotor, Wolfgang De Meuter, for giving me the opportunity to work on this thesis, having taken the time with Jens and Coen to find a subject that aroused my curiosity, as well as for his comments on the content of this thesis.

I would also like to thank the other members of the Software Language Lab, for their relevant reflexions and comments during the thesis presentations.

Finally, I would like to thank those who read parts of this work even though it was not their area of expertise. They gave me helpful remarks and comments. Thank you Mathieu Bivert, Rémy Delrue, Quentin Pradet and Philippe Stiévenart.

Contents

List of Figures

List of Tables

List of Examples

Chapter 1

Introduction

This dissertation investigates the use of abstract interpretation to detect concurrency issues in higher-order programs. Our thesis is that abstract interpretation can be used to develop useful analyses in order to find defects in concurrent programs. Moreover, defining the analyses with locks as a synchronization primitive instead of the traditional `cas` leads to simpler and more efficient analyses for race conditions and deadlocks.

Detecting bugs such as race conditions and deadlocks is particularly hard, and they tend to appear late in the development of programs. Being able to detect them automatically using static analysis tools is therefore important. In this dissertation, we start from Might and Van Horn's PCESK machine, from which we can build a graph of states reachable by a higher-order program. We design analyses that use this graph to detect race conditions and deadlocks. We then improve the PCESK machine to be able to analyze larger programs based on locks. The race condition and deadlock analyses are simplified by this improvement, and we are able to analyze more complex programs.

1.1 Background

At the time of writing, the improvement in processor frequency has now stalled for almost ten years. However, the number of transistors per processor keeps increasing, following Moore's law. The processing power of computers also keeps increasing, but it is now distributed over multiple cores. Indeed, multicore processors started to appear on personal computers at around the same time as the frequency stopped increasing. Previously, to get an increase in the speed of a program, one only had to wait the release of a new generation of processors with higher frequency, it was a *free lunch*. The consequence of the stall in processor frequency is that the free lunch has been over for almost a decade now [Sut05], and programs have to take multicore processors into account in order to take advantage of this gain in processing power.

Techniques that can take advantage of multicore processors have existed long before multicore processors themselves appeared. These techniques date back to the origins of modern operating systems, when time-sharing was invented to handle multiple processes running concurrently on the same computer. Later developments introduced the notion of *threads*, which were lighter than processes and made communicating between multiple execution units easier. Threads are now supported by most of the programming languages currently in use.

Programming a complex system that has multiple threads is far from easy. Two executions of the same program with the same input data may lead to two different interleavings of the program's instructions. It is therefore not possible to conclude that a program will always execute correctly with some input parameters by testing it on some executions, as those executions might not run threads in an interleaving that will exhibit a possible bug. Bugs due to concurrency in such programs tend to appear in some executions and disappear in others, making debugging a hassle. Also, some bugs might only appear in very specific cases (e.g. a high load on the machine) and are not detected until the program is deployed in production on multiple servers, when fixing the bug has become much more costly. Because defects found late in the development are costly, and concurrency bugs tend to appear late in the development, it is important to be able to detect them as soon as possible.

Concurrency bugs can be divided into three classes: race conditions, deadlocks, and livelocks.

1. A *race condition* happens when the outcome of a program (or the value of a variable) depends on the order in which the threads are executed, that is, when there is a *race* between multiple threads to read from or write to the same variable. Race conditions may lead to incorrect values computed by a program. For example, if a banking software has a race condition and a user performs two withdrawals concurrently on the same bank account, the resulting amount of money in the bank account could be the same as if he did only one withdrawal, while he got the money from both.

2. A *deadlock* happens when multiple threads block each other, for example when trying to access the same resources in a different order. When a deadlock happens, parts of the program get entirely blocked and the execution never terminates.

3. A *livelock* has the same result as a deadlock, but for a different reason. When a livelock happens, the threads are not blocked, but will execute the same instructions over and over again, each one for example acquiring a lock, trying to acquire another lock, failing, releasing the first lock, and restarting. As Nick Falkner puts it[1], "*a deadlock is a brick wall in the corridor, a livelock is the dance between you and a coworker as you try to sidestep each other in the same corridor until you both die of politeness.*"

1.2 Objectives and Contributions

As concurrency bugs can be particularly hard to find, any assistance from the computer to find them is welcome. Such help is provided by static or dynamic analysis tools. Our objective is to use one of the most important static analysis techniques, namely abstract interpretation, in order to find such defects. We base our work on the PCESK machine, a formalism described by Might and Van Horn [MVH11]. Our contributions are the following.

1. We provide a working implementation of the PCESK machine, along with results from our benchmarks. Might and Van Horn describe the PCESK machine, but did not provide an implementation. They identify the computational complexity of a flow analysis, but no further indication about the practical performance (in term of state space size and analysis time) is given. No information about the complexity nor practical performance of their May-Happen-in-Parallel (MHP) analysis is given either.

2. We present two improvements to the PCESK machine: abstract garbage collection, and state subsumption. Abstract garbage collection was already formally described for the CESK machine, but it requires modifications to be used in the setting of the PCESK machine. State subsumption is also an existing technique, but again has not been described in the context of the PCESK machine.

3. We adapt the MHP analysis of Might and Van Horn to avoid detection of some specific false positives.

[1] `http://nickfalkner.wordpress.com/2012/11/11/`

4. We define multiple new analyses based on the PCESK machine that are able to find concurrency bugs: a race condition analysis and a deadlock analysis. We validate those analyses on multiple examples.

5. We present the PCESK$_L$ machine, a variant of the PCESK machine that uses locks as a synchronization primitive. We adapt the race condition analysis and define a new deadlock analysis for the PCESK$_L$ machine. We validate those analyses on multiple examples, and provide benchmarks demonstrating that this machine leads to better performance than the original PCESK machine.

1.3 Overview of the Approach

This dissertation starts by describing the principles behind static analysis (Section 2.1), and then describes abstract interpretation, an important static analysis technique used in the approach of this dissertation (Section 2.2).

We also review existing static analysis tools that can analyze concurrency constructs in programs (Section 2.3), and we look at other techniques not yet included into released tools (Section 2.4).

Chapter 3 introduces material needed to describe the PCESK machine. In Section 3.1, we formally describe the CESK machine and existing refinements that improve the analyses, and demonstrate how to implement them. Section 3.2 describes Administrative Normal Form, a way of writing programs that simplifies the semantics and that will be useful when reasoning about concurrency in the PCESK machine.

The rest of the dissertation is devoted to describe our proposed solution to analyzing concurrency constructs in higher-order programs. This solution is based on the PCESK machine, a way of formalizing abstract semantics of concurrent, higher-order languages. The language we use is CScheme (for *Concurrent Scheme*), a simplified version of Scheme, with additional concurrency operators, and `cas` (compare-and-swap) as the only synchronization primitive (Section 4.1). The semantics of this language are abstracted in order to compute an over-approximation of the reachable states of any CScheme program in finite time (Section 4.3). With this graph of reachable states, it is possible to build multiple analyses that detect concurrency defects in CScheme programs. We develop a race condition analysis (Section 5.4) and a deadlock analysis (Section 5.5). Both have limitations and are unsound, but we demonstrate their usefulness on simple examples (Sections 6.4 and 6.5).

Chapter 7 delves into our implementation of the PCESK machine and its analyses, and gives the results of our benchmarks on this implementation.

In Chapter 8, we overcome some of the limitations of our earlier analyses. We design CScheme$_L$, a variant of CScheme, in which locks are used as concurrency primitives instead of `cas`. We adapt the semantics of the PCESK machine, obtaining the PCESK$_L$ machine. The use of locks simplifies the formalization of the race condition and deadlock analyses, and solves some of their precision problems. Additionally, the PCESK$_L$ machine performs better than the PCESK machine, which opens up the possibility of analyzing larger programs.

1.4 Notation

We define here some notations used throughout this document.

- *Domain and range*: given a function $f : X \to Y$ (or a partial function $f : X \rightharpoonup Y$), $dom(f)$ denotes the set X and $range(f)$ denotes the set Y.

- *Free variables*: given a language with a set of expression *Exp*, a set of variable identifiers *Var*, $\mathbf{fv} : Exp \to \mathcal{P}(Var)$ computes the set of free variables appearing in an expression. A standard definition for such a function can be found for example in [Tur08, pp. 247].

- *Map*: $X \rightharpoonup Y$ is a mapping from elements of set X to elements of set Y which is not defined for every element of X (it is a *partial mapping*). This can be seen as a *map* in traditional programming languages, where keys are elements of X and values are elements of Y.

- *Map restriction*: with $A = X \rightharpoonup Y$ and $B \subset X$, $A|B$ represents the map A restricted to the keys that are contained in B.

- *Powerset*: given a set X, $\mathcal{P}(X)$ is the set of all subsets of X.

- *Set difference*: given $A \subset X$ and $B \subset X$, $A \setminus B$ is the set containing all the elements of A that are not in B.

Chapter 2

State of the Art

This chapter describes the state of the art in static analysis of concurrent programs. First, Section 2.1 describes more precisely what static analysis is, and why it is interesting in our case. We then describe in more detail an important technique for performing static analysis, namely *abstract interpretation*. This is the technique we use throughout this dissertation, and we explain its mathematical foundations in Section 2.2. The main commercial and open-source tools that perform static analysis of concurrent programs are reviewed in Section 2.3, and other existing static analyses for detecting bugs in concurrent programs, which are not included in one of the reviewed tools, are described in Section 2.4.

2.1 Static Analysis

Static analysis consists of analyzing computer programs without needing to execute them, in order to find defects or to prove some properties about the program. Unfortunately, Rice's theorem [Ric53] tells us that designing a program that proves a non-trivial property of any program is similar to solving the halting problem, which is undecidable.

To stay on the decidable side, static analysis techniques need to simplify the problem. Instead of having an algorithm that can tell for any program whether it satisfies or not a property (as in Figure 2.1) and being limited by Rice's theorem (i.e. not being able to decide any non-trivial properties about a program), we can restrict it to some instances of this problem (answering *yes* or *no*), but let other instances unsolved (and answer *maybe*, see Figure 2.2).

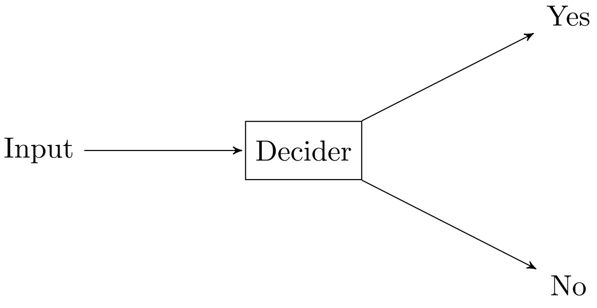

Figure 2.1: Possible output of a decider.

Obviously, it is possible to always answer *maybe*, but that would not be very useful: we would not have any *precision*. We want to solve a maximum number of instances, i.e. have enough precision so that it suits the problem. Since static analysis is decidable, it can be run in finite time at compilation time, and can provide strong guarantees about the execution of the program.

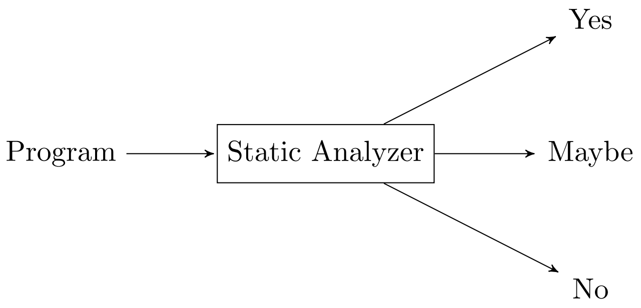

Figure 2.2: Possible output of a static analyzer.

For example, a programming language with a safe type system (a form of static analysis) is guaranteed not to have any type errors at runtime. However, to achieve this, the type checker has to reject some valid programs. For example, the program `(let ((x (if #t 0 "foo"))) (+ x 1))` will be rejected by most statically typed programming languages because both branches of the `if` do not have the same type (integer vs. string), even though this program will correctly return `1` and never result in a runtime type error. The set of rejected programs is an *over-approximation* of the set of invalid programs. Every program that has a possible runtime type error is included in this set, but some perfectly valid programs, without any runtime type error, are also included. This is the price of decidability.

Many different techniques of static analysis exists. We will use *abstract interpretation* because it mimics interpretation and thus allows to stay very close to the original language semantics and avoids to force the user to modify his program in order to analyze it. It is also more suited to analyze higher-order programs, that is, programs that use functions that can take functions as arguments and return functions. Abstract interpretation also provides a way to parameterize the precision, thus allowing to achieve the precision needed. Finally, the base of analyzing concurrent programs with abstract interpretation have already been formalized [MVH11], but no implementation seems to exist.

2.2 Abstract Interpretation

Abstract interpretation [CC77] is a technique used to reason about programs by interpreting an approximation of those programs through abstraction of the semantics. Using abstract interpretation, it is possible to perform sound static analyses with a precision that can be tuned to fit the needs. By various mechanisms, it is possible to increase the precision at the cost of analysis time.

Abstract interpretation works in a similar way as conventional interpretation (which we will call *concrete* interpretation). When using small-step semantics, concrete interpretation can be described as follows [Mig10]: *inject* the program e to interpret into an initial state s_0, then step from this state to the next state using a *transition function*, until a final state is reached. If no final state is reached, the execution will continue indefinitely. As a result, we get a (possibly infinite) *trace* of the execution of the program (represented in Figure 2.3 where the execution of the program goes through states s_0, s_1, $\dots$), that might depend on user input or other values that can change from one execution to the other. Therefore, it is impossible to perform static analysis using this trace, and we need to apply abstraction in order to compute something that is finite.

When doing abstract interpretation, the components of the state spaces that are infinite (primitive values and addresses) are *abstracted* to be finite. This introduces a loss of precision, but allows to have something computable in finite time. The program is also injected, but into an initial *abstract* state $\hat{s}_0$, and the steps are done using an *abstract*

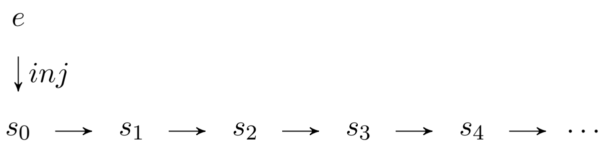

Figure 2.3: Concrete interpretation using small-step semantics.

transition function that can go from one state to multiple states because of this imprecision. For example, to compute `(if x 1 2)`, when `x` has been abstracted, the abstract transition function may need to go to both branches. As a result, we get a finite state *graph*. Such a state graph is represented in Figure 2.4, where we can see that the abstract state $\hat{s}_2$ goes to both abstract states $\hat{s}_3$ and $\hat{s}'_3$, and that the possibly infinite trace starting at $\hat{s}_4$ is now represented by a loop in the state graph. This state graph contains all the possible traces of the execution of the program. Using this graph, we can prove properties that hold on the analyzed program.

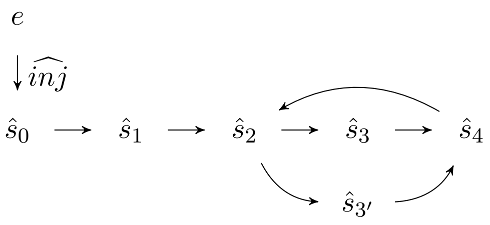

Figure 2.4: Abstract interpretation using small-step semantics.

2.2.1 Mathematical Foundations

This section introduces the mathematical concepts used in abstract interpretation. Those concepts will be used to formally define an *abstraction*, but are not primordial to understand the rest of this dissertation.

Definition 1. *A relation $\sqsubseteq$: $S \times S$ is a* ***partial order*** *if it is:*

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

Definition 2. *A* ***partially ordered set*** *$(S, \sqsubseteq)$ is a set associated with a partial order on that set.*

A partially ordered set can be defined graphically using a *Hasse diagram*, in which a vertex that connects two elements means that the element above the other is greater.

> **Example 1** (Hasse diagram)**.** The partially ordered set $(\mathcal{P}(\{1,2,3\}), \subseteq)$ is represented in the Hasse diagram of Figure 2.5.

Definition 3. *For a subset $X \subseteq S$, $u \in S$ is an* ***upper bound*** *of X if $\forall x \in X, x \sqsubseteq u$, and $u \in S$ is the* ***least upper bound*** *of X if, for every upper bound x of X, $x \sqsubseteq u$. The least upper bound of X will be denoted by $\bigsqcup X$, and $x \sqcup y$ denotes $\bigsqcup\{x, y\}$. $\sqcup$ is called the*

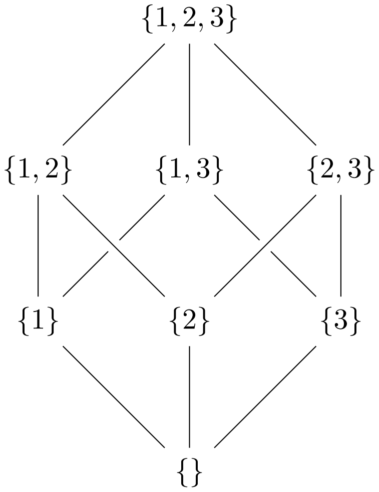

Figure 2.5: Hasse diagram of the partially ordered set $(\mathcal{P}(\{1,2,3\}), \subseteq)$.

join *operator. Similarly, $l \in S$ is an* ***lower bound*** *of X if $\forall x \in X, l \sqsubseteq x$, and $l \in S$ is the* ***greatest lower bound*** *of X if, for every lower bound x of X, $l \sqsubseteq u$. The greatest lower bound of X will be denoted by $\bigsqcap X$, and $x \sqcap y$ denotes $\bigsqcap\{x, y\}$ and is called the* ***meet*** *operator.*

Definition 4. *A* ***lattice*** *$(L, \sqsubseteq)$ is a partially ordered set where every subset of L composed of two elements has a least upper bound and a greatest lower bound.*

Definition 5. *A* ***complete lattice*** *$(L, \sqsubseteq)$ is a partially ordered set where every subset of L has a least upper bound and a greatest lower bound. Also, a complete lattice comprises two special elements: a* ***bottom*** *element $\bot = \bigsqcap L$ and a* ***top*** *element $\top = \bigsqcup L$.*

Definition 6. *A* ***Galois connection*** *between two partially ordered sets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ is a pair of functions $(\alpha : A \to B, \gamma : B \to A)$ such that $\forall a \in A, b \in B, \alpha(a) \sqsubseteq_B b \Leftrightarrow a \sqsubseteq_A \gamma(b)$.*

2.2.2 Abstraction

In the case of abstract interpretation, an *abstraction* $\hat{X}$ of a concrete set X consists of a Galois connection between $(\mathcal{P}(X), \subseteq)$ and $(\hat{X}, \sqsubseteq)$. Abstract values, sets and functions will generally be denoted by the same name as their concrete counterpart, but with a hat. In the context of abstract interpretation, the function α is called the *abstraction function* and γ is called the *concretization function*. Because values are abstracted, operations on values should also be abstracted.

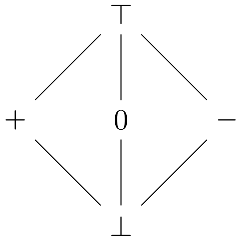

Figure 2.6: Partially ordered set of signs.

Example 2 (Sign abstraction)**.** A way to abstract the set of integers $\mathbb{Z}$ could be to map it into the set of signs $\widehat{Sign} = \{+, -, \hat{0}, \top, \bot\}$ which forms a complete lattice with the $\sqsubseteq$ ordering as shown in the Hasse diagram of Figure 2.6. The abstraction and concretization functions are defined as follows:

$$\begin{aligned}
\alpha &: \mathcal{P}(\mathbb{Z}) \to \widehat{Sign} \\
\alpha(N) &= \bot \text{ when } N = \varnothing \\
&= \hat{0} \;\text{ when } N = \{0\} \\
&= + \text{ when } \forall n \in N, n > 0 \\
&= - \text{ when } \forall n \in N, n < 0 \\
&= \top \text{ otherwise}
\end{aligned}$$

$$\begin{aligned}
\gamma &: \widehat{Sign} \to \mathcal{P}(\mathbb{Z}) \\
\gamma(P) &= \varnothing \text{ when } P = \bot \\
&= \{0\} \text{ when } P = \hat{0} \\
&= \mathbb{Z}^{+} \text{ when } P = + \\
&= \mathbb{Z}^{-} \text{ when } P = - \\
&= \mathbb{Z} \text{ otherwise}
\end{aligned}$$

The negation operation $f(N) = \{-n \mid n \in N\}$ can be abstracted as follows:

$$\begin{aligned}
\hat{f}(\bot) &= \bot \\
\hat{f}(0) &= 0 \\
\hat{f}(+) &= - \\
\hat{f}(-) &= + \\
\hat{f}(\top) &= \top
\end{aligned}$$

Figure 2.7 represents the two sets $\mathcal{P}(\mathbb{Z})$ and $\widehat{Sign}$ and graphically depicts the application of the abstraction and concretization function, and highlights the difference between applying f in the concrete state and applying $\hat{f}$ in the abstract state space. When we are in the abstract state space, we lose precision. If we apply the concretization function after having applied the abstract operation, we can see that the result is less precise than when we apply the concrete operation directly:

$$\{1\} = f(\{-1\}) \subseteq (\gamma \circ \hat{f} \circ \alpha)(\{-1\}) = \mathbb{Z}^{+}$$

However, $\mathbb{Z}^{+}$ is an over-approximation of $\{1\}$, and properties that hold for $\mathbb{Z}^{+}$ will also hold for $\{1\}$.

For performing abstract interpretation, all the infinite components of the concrete state space have to be abstracted, as well as the operations on those components. Among those components, the *state* component Σ represents the current state of the interpretation, and the *transition function* $(\to)$ steps from one state to the next possible states. This transition function also has to be abstracted.

Soundness An abstraction will be *sound* if it preserves the abstraction map [VHM10]. For a sound abstraction, if we have $\varsigma \to \varsigma'$, and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then $\exists \hat{\varsigma}' \in \widehat{\Sigma}$ such that $\hat{\varsigma} \mathrel{\widehat{\to}} \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$. Having soundness ensures that the states reachable by the concrete transition function are also reachable by the abstract transition function, so that we can use only the

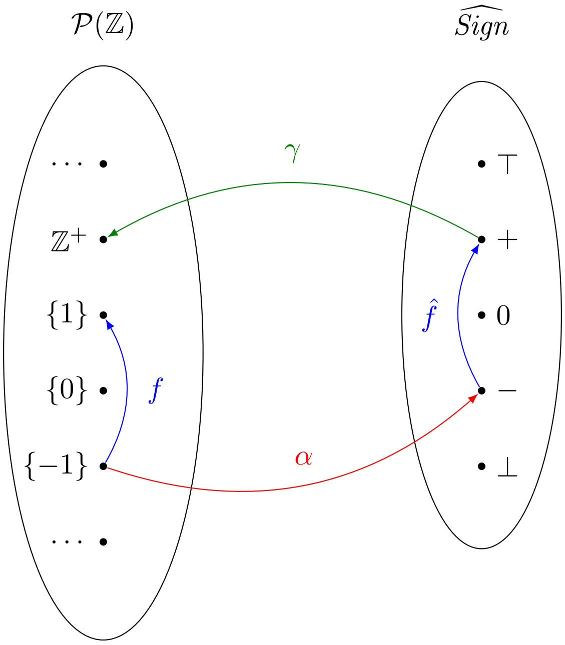

Figure 2.7: Galois connection between integers and the sign abstraction.

abstract transition function to prove properties about the concrete transition function.

From an abstraction, we can build static analyses that inspect the produced state graph to find certain properties about the program. A static analysis meant to find a defect will be *sound* if it finds at least every defect in the program (and maybe has false positives). To have a sound static analysis, it is necessary to use a sound abstraction, but not sufficient, as the defects found will depend on how the static analysis is formalized.

> **Example 3** (Soundness of the sign abstraction)**.** With the same sign abstraction as the one used in Example 2, we can see in Figure 2.7 that, if we use f as the transition function, $\forall \hat{n}$ such that $\alpha(\{1\}) \sqsubseteq \hat{n}$ (i.e. $\hat{n} \in \{\top, +\}$) we always have a $\hat{n}'$ such that $\hat{f}(\hat{n}) = \hat{n}'$ and $\alpha(\{-1\}) \sqsubseteq \hat{n}'$. Indeed, if $\hat{n} = \top$, $\hat{n}' = \hat{f}(\hat{n}) = \top$ and $\alpha(\{-1\}) = - \sqsubseteq \top$, and if $\hat{n} = +$, $\hat{n}' = \hat{f}(\hat{n}) = -$ and $\alpha(\{-1\}) = - \sqsubseteq -$. We can further verify that this property holds not only for $\{-1\}$ but for any $n \in \mathcal{P}(\mathbb{Z})$, meaning that this abstraction is sound.

Precision The closer an abstraction is to its concrete counterpart, the higher the precision. However, an abstraction with higher precision will lead to increased computing time. The abstraction used has to make a trade-off between computation time and precision, since we want a precision that is sufficient to analyze the properties we are interested in, but we also want the analysis to terminate in a reasonable amount of time.

> **Example 4** (Precision of the sign abstraction)**.** Suppose that we are interested to know whether an integer will always be positive during the execution of a program. The sign abstraction of Example 2 might be sufficient, depending on the operations that are performed on the numbers. For example, if we analyze a program that has a counter `i` that starts at 0 and that is continually increased by 1, this abstraction is sufficient to know that `i` will always be positive (0 or a positive number increased by a positive number is a positive number). However, if it is increased by 2 and then decreased by 1, the abstraction will not be precise enough to reason about the sign of `i` because it will observe that `i` is increased by *some positive number* and then increased by *some negative number*, without having any information about the value of those numbers, and it will conclude that the abstract value of `i` is $\top$, which contains non-positive numbers.

2.3 Existing Analysis Tools

Many tools that can detect bugs in concurrent programs already exist. These tools can either perform dynamic or static analysis. *Dynamic analysis* tools will only detect bugs when they happen at runtime. Some dynamic analysis tools might for example tweak the scheduling algorithm to expose more of those bugs, as IBM ConTest does [Ur08]. On the other hand, *static analysis* tools are executed right before or after compilation and are thus able to detect bugs without having to execute the program. This section will review the most common static analysis tools among those that can verify concurrent programs, and discuss some of their shortcomings.

Among static analysis tools there exists many *model checkers* that verify if a program (described by its *model*) satisfies a set of properties (that is, the *specification* of the program). Most of the model checkers that can verify concurrent programs (such as SPIN [Hol04]) require the user to describe the system to verify in a specific language (such as PROMELA, in the case of SPIN) and are thus not able to automatically analyze programs without requiring action of the user. Notable exceptions are Bandera, a model checker for concurrent programs written in Java, and McErlang, a model checker for Erlang. Those are described later in this section.

Other static analysis tools use various techniques. Tools such as FindBugs identify bugs that follow some pre-encoded patterns, which makes the analysis unsound as it is not possible to encode every pattern for a class of bug. CheckThread relies on annotations given by the user, to identify calls to non-thread-safe functions from thread-safe ones. Other tools such as CodeSonar and MayPar relies on symbolic execution. CodeSonar supports common programming languages while MayPar only supports an uncommon language. Finally, ThreadSafe is a commercial tool which can detect many concurrency bugs but we found no description of the techniques it uses.

2.3.1 Bandera

Bandera [CDH$^+$00] is a tool that eases the use of conventional model checking tools that accept specific input languages (e.g. SPIN with PROMELA) to verify programs written in Java. Typical model checking tools accept a language that describes the finite-state transition system corresponding to the program. Bandera does the automatic translation of a Java program to one of those languages, and has multiple translator to support different model checking tools. It explicitly supports Java's concurrency constructs [HD01] and is thus adapted to analyze concurrent programs.

Bandera resolves the barrier of language difference for using model checking tools. However, in order to verify a program with Bandera, the user still has to explicitly state which properties are to be verified.

2.3.2 McErlang

McErlang [FS07] is a model checker that implements an Erlang virtual machine that will explore multiple possible paths taken by a program in order to find states which results in errors.

The user can also specify linear temporal logic (LTL) formulas to verify during the execution of the program. Similarly to Bandera, this tool resolves the problem of the input language, but the user still has to specify the properties the tool should verify (even though in this case, fatal errors such as uncaught exceptions are found automatically).

2.3.3 FindBugs

FindBugs [HP04a] is a popular open-source static analysis tool that analyzes Java programs for bugs. It is able to detect many classes of bugs, including concurrency bugs [HP04b].

However, the analysis is carried out by looking at common *structural* bug patterns (as opposed to *behavioral* bug patterns). For example, it is a common error to call the `run()` method on a Java thread instead of the `start()` method. This will execute the code of the thread on the current thread and will not create a new thread. Such patterns are recognized by `findbugs` and reported to the user.

The analysis made by `findbugs` scales to production systems and is able to detect many bugs on real codebases [APM$^+$07]. However, since it is based on bug patterns it is only able to detect the patterns that have been encoded and will likely miss existing bugs on a given codebase. This implies that `findbugs` cannot be used to reason about the absence of certain bug classes. Also, many of the bug patterns depend on features of the Java language (e.g. calling `run()` instead of `start()`) and not on higher-level, language-independent constructs.

2.3.4 CodeSonar

CodeSonar[1] is a proprietary static analysis tool that can analyze C, C++ and Java programs to detect many different bugs. Because CodeSonar is based on symbolic execution, Grammatech argues that CodeSonar is able to detect more bugs than traditional static analysis tools that use bug patterns. One key feature of CodeSonar is that it puts great emphasis on detecting concurrency errors.

Among other bugs, CodeSonar is able to detect data races, deadlocks and incorrect uses of synchronization techniques. The analysis of those concurrency errors relies on the fact that the programming model uses locks. No information could be found about how CodeSonar scales with respect to the size of the codebase.

2.3.5 CheckThread

CheckThread[2] is an open-source static analysis tool aimed at finding concurrency bugs in Java bytecode. CheckThread requires to define a thread policy by adding Java annotations to chunks of code that are considered as thread-safe (`@ThreadSafe`) or not (`@NotThreadSafe`), or as thread-confined (`@ThreadConfined`). When a non-thread-safe method is called from a thread-safe part of the code, an error will be displayed by CheckThread at compile time. Also, inside a method declared as thread-safe CheckThread will ensure that no shared data will be subject to reads and writes without synchronization. A downside of this approach is that the user is required to annotate the entire program, which can take a non-negligible amount of time. If done incorrectly, the analysis might also fail to recognize some errors. This is for example the case if the user declares some parts of the code as non thread-safe, while they should be thread safe for the correctness of the application.

2.3.6 MayPar

MayPar [AFMG12] is a proprietary static analysis tool that targets a language based on *concurrent objects*, where objects are the concurrency unit and asynchronous methods are used for communication. It is able to find whether two expressions in a program written in this language may happen in parallel or not. This form of analysis is called a *may-happen-in-parallel* analysis (MHP analysis). However, it does not support bug detection.

2.3.7 ThreadSafe

ThreadSafe[3] is a proprietary static analysis Eclipse plugin aimed at finding concurrency bugs in Java programs. It can detect bugs such as race conditions, deadlocks, unpredictable

[1] `http://www.grammatech.com/codesonar`

[2] `http://checkthread.org/`

[3] `http://www.contemplateltd.com/threadsafe`

result, and can also advise the user on how to improve some parts of its code (e.g. in the presence of redundant synchronization). As it is a commercial product, no information was found as to how ThreadSafe works.

2.4 Static Analysis of Concurrent Programs

This section reviews various academic work related to static analysis of concurrent programs, which has not been integrated into one of the existing tools described previously.

Older work such as Jagannathan et al.'s aims at optimizing programs instead of detecting defects in them. Most of the recent work analyzes the use of locks to detect both race conditions and deadlocks. A notable exception is the race detection technique of Naik et al. which combines multiple passes of static analysis to detect the absence of such an error. The works of Flanagan et al. and Boyapati et al. take a different approach as it uses the type system to enforce the absence of concurrency bugs.

2.4.1 Jagannathan et al.

Jagannathan et al. [WJP94, JW94, Jag95] describe an abstract interpreter that is able to compute both intra- and inter-thread control-flow and dataflow information of programs written in a language inspired by Scheme and SML. This language supports dynamic process creation with `spawn` and uses shared memory through *shared locations*, created and manipulated with `mk-loc`, `read` and `write`. The `read` operation is a blocking read. It is shown how to implement `future` and `touch` with those features, and an efficient implementation of such an interpreter is described. An analysis useful for doing optimizations of concurrent programs is also described, but unlike other approaches presented here, no work seems to have been done on finding defects in such programs.

In fact, their abstract interpreter does not seem adapted to find defects such as race conditions. Their approach consists of assigning to each program location an abstract environment and an abstract store, indicating the possible variable values at each program location. However, to be able to find race conditions, one has to be able to reason about which expressions can be evaluated in parallel, and the information computed by this abstract interpreter does not allow such reasoning.

In this dissertation, we will build upon another abstract interpreter approach by Might and Van Horn [MVH11], which can build a state graph of a concurrent program's execution, thus allowing to reason about which expressions may be evaluated in parallel.

An interesting point however is the fact that the analysis is built for a language with a few concurrency primitives, and still are able to analyze other concurrency constructs such as futures.

2.4.2 RacerX

RacerX is a static analysis tool described in [EA03] but which does not seem to have been released. It is able to analyze huge C codebases such as the Linux and the FreeBSD kernels and to automatically detect both race conditions and deadlocks in those codebases. Both the deadlock checking algorithm and the race detection algorithm use a lockset analysis, and are thus dependent on the fact that the programs analyzed make heavy use of locks (as it is the case with most analyses presented in this section). Many different techniques that aim at minimizing the number of false positives or the analysis are described.

As a lockset analysis is highly dependent on the fact that the analyzed language uses locks, we will not use this technique in this work, as we will first analyze a language without locks, but with an atomic compare-and-swap.

2.4.3 Naik et al.

Naik describes an algorithm for detecting race conditions in [NAW06] and an algorithm to detect deadlocks in [NPSG09], both for Java programs. The two algorithms are similar in the fact that they both perform multiple static analyses to try to refute a condition for the absence of the defect. For the race detection algorithm, a pair of statement of a Java program is free of any race condition if one of the four following conditions holds.

1. The pair of statements never access the same memory location. This is checked by a *may-alias analysis*.
2. The memory location accessed by the statements is always thread-local. This is checked by a *thread-escape analysis*.
3. The statements are ordered by the thread structure of the program. This is checked by a *may-happen-in-parallel analysis*.
4. The statements are ordered by lock-based synchronization. This is checked by a *conditional-must-not-alias analysis* [NA07].

For the deadlock detection algorithm, there are six necessary conditions for a deadlock to happen between two threads that perform two locks operations (thread t^a does its lock operations at locations l_1^a and l_2^a, and similarly for thread t^b) with a pair of locks. However, nothing is said about deadlocks involving more than two locks.

1. Both threads should be able to reach l_2^i after having done the first lock operation at l_1^i, while still holding the lock (*reachability*). This is checked by a *call-graph analysis*.
2. The lock acquired at l_1^a can be the same as the lock acquired at l_2^b and similarly for l_1^b and l_2^a (*aliasing*). This is checked by a *may-alias analysis*.
3. A lock acquired by a thread can be accessible from more than one thread (*escaping*). This is checked by a *thread escape analysis*.
4. Both threads can reach l_2^a and l_2^b simultaneously (*parallel*). This is checked by a *may-happen-in-parallel* analysis.
5. Because locks are reentrant in Java, both threads should acquire different locks at l_1^i and l_2^i (*non-reentrent*). This is checked by a *may-alias analysis* while it would require a *must-alias analysis* to be sound (which is harder to check).
6. Both thread can respectively reach l_1^a and l_1^b without holding a common lock (*non-guarded*). This should also be checked by a *must-alias analysis* but is checked by a *may-alias analysis* at the cost of soundness.

Those two analyses are both unsound but seem to be able to find many race conditions and deadlocks. We will use the idea of combining multiple analyses to find race conditions when we describe our race condition analysis.

2.4.4 Lock-Order Graphs

Multiple analyses [AB01, vP04, WTE05] detect deadlocks in Java programs and libraries by building a lock-order graph that captures the locking information for the entire library. This lock-order graph contains cycles when there is a possibility of deadlock.

[AB01] extend the support of concurrency in Jlint (a general-purpose static analyzer for Java programs) in order to improve the detection of deadlocks. This approach is able to analyze huge Java programs by keeping the analysis simple, but it only detect specific cases of deadlocks (on 15 examples, only 6 are correctly analyzed).

[vP04] describes a generic unsound analysis to detect deadlocks, which seem to have very few false negatives.

[WTE05] aims at detecting deadlocks in Java libraries, and, similarly to RacerX, provides many approaches to reduce the number of false positives.

For the same reason that we will not use a lockset analysis inspired from RacerX, we will not use the lock-order graphs approach as it is dependent on the fact that locks are used as a synchronization primitive.

2.4.5 Flanagan et al.

A different approach to statically prevent race conditions and deadlocks is to use an appropriate type system. Flanagan et al. managed to express an analysis as an extension of Java's type system. This analysis checks whether the appropriate locks are held when shared fields are accessed [FA99, FF00], but it requires to annotate the Java code. However, a follow-up paper describes how to automatically infer those annotations [AFF06]. Flanagan and Qadeer also describe a type system that not only avoids deadlocks and race conditions, but that checks the *atomicity* of Java methods annotated with an `atomic` annotation [FQ03].

2.4.6 Boyapati et al.

Boyapati et al. [BR01] also describe a type system that ensures the absence of race conditions in Java programs. Later, the type system was extended to also avoid deadlocks [BLR02]. The type system is based on *ownership types* and *unique pointers* in order to enforce object encapsulation without being too constrained. However, this system requires the user to encode its locking strategy in the types of its program which can be a big overhead for existing codebases. The resulting language is formalized in details as SafeJava in [Boy04].

2.5 Conclusion

In this chapter we described the concepts behind static analysis, and looked in detail at one of the many possible approaches to do static analysis, namely abstract interpretation. We introduced the mathematical foundations of abstract interpretation, which comprises lattices, Galois connections, and abstractions.

We looked at both existing static analysis tools that can analyze concurrent programs, and existing academic work to analyze such programs. We have seen that many different approaches can be used (model checking, bug patterns, lockset analysis, type systems, . . .).

The ideal tool should have multiple properties. One important property is that the tool should be able to analyze a program without requiring the user to translate it into another language or adding annotations, as this might not be practicable for large programs. Another important property is that the user should not have to tell the tool what to look for — the tool should be able to detect defects in the program automatically. Indeed, finding properties that should hold in a program is far from easy and requires a good understanding of the program. Also, bugs tend to appear where they are not expected, and having an automatic analysis prevents the user from only checking parts where bugs are expected.

As having an analysis that is both sound (every bug of a certain kind is found, there are no false negatives) and complete (every bug found is indeed a potential bug, there are no false positives) is undecidable, we want an analysis that only has a “small” number of false negatives and false positives. A sound analysis will be able to prove the absence of certain classes of bugs in a program, while a complete analysis will just find some of the bugs contained in that program. A trivial complete analysis is the one that does not detect any bug. It is thus important to have an analysis that can detect a “good” portion of the defects it looks for, while only having a few false positives.

The related work described in this chapter tends to miss at least one of the desired properties, and most of the existing analyses depend on the fact that locks are used as a concurrency mechanism. Also, few of the existing concurrency analyses take higher-order programs into account. To investigate how we could combine our desired properties and have an analysis that is independent of locks, we will base our work on Might and Van Horn’s PCESK machine [MVH11], which is based on abstract interpretation. To describe this machine, we first need to introduce some concepts, which is done in the next chapter.

Chapter 3

Background Material

This chapter describes material that serves as the foundation for the rest of this dissertation. We start by introducing the CESK machine, a formalism used to describe an interpreter in its concrete as well as its abstract version. This machine computes a set of states that are reachable by the input program. Later, this will allow us to build static analyses that verify properties by inspecting this set of reachable states.

In Section 3.2, we present a brief description of atomic expressions and Administrative Normal Form (ANF). Distinguishing atomic expressions from non-atomic expressions is important in the context of concurrent programming, and ANF enforces atomicity where required.

3.1 The CESK Machine

In this Section we formalize the semantics of an untyped λ-calculus. We use a machine that allows us to abstract the semantics while staying close to the concrete one. We follow the approach of [VHM10] to build a sound abstract machine for a simple sequential language. First, the language interpreted by this machine is described in Section 3.1.1. Then, the *concrete* semantics are given in Section 3.1.2 in order to show how to interpret this language. Those concrete semantics are then *abstracted* in Section 3.1.3 to have a finite state space, thus allowing to have a decidable analysis. Finally, some existing refinements that improve either the running time of the machine or its precision (or both) are discussed in Section 3.1.4.

This machine will be then be used as a basis for the formalization of a parallel machine in Chapter 4.

3.1.1 Language

We define the CESK machine for an untyped λ-calculus whose syntax is given in Figure 3.1 and only contains abstraction and application.

$$\begin{aligned} v &\in \mathit{Var} \qquad \text{a set of identifiers} \\ e &\in \mathit{Exp} ::= v \mid (e\ e) \mid (\lambda v.e) \end{aligned}$$

Figure 3.1: Syntax of an untyped λ-calculus.

3.1.2 Concrete Semantics

The semantics of a machine such as the CESK machine can be described in four constructs:

1. the *state space* defines the structure of the states of the machine,

2. the *transition function* defines how to step from one machine state to the next,
3. the *injection function* defines how to inject the expression to evaluate into an initial machine state, and
4. the *evaluation function* defines the set of machine states that are reachable given an initial expression.

State Space The state space of this CESK machine is given in Figure 3.2. The states (Σ_{CESK}) are composed of five components:

1. the *control* component *Control* represented by the expression being currently evaluated or the value resulting from an evaluation,
2. the *environment* component *Env* which binds variable names to addresses,
3. the *store* component *Store* which binds addresses to values,
4. the *continuation* component *Addr* which indicates the address of the current continuation, and
5. the *time* component *Time* which will depend on the analysis one want to make.

The CESK machine initially takes its name from its components (**C**ontrol, **E**nvironment, **S**tore, **K**ontinuation), but we use a variant that replace the current continuation of a state by the *address* of this continuation, and that adds a timestamp to every state to simplify the allocation of new addresses.

$$\begin{aligned}
\varsigma_{\text{CESK}} \in \Sigma_{\text{CESK}} &= \mathit{Control} \times \mathit{Env} \times \mathit{Store} \times \mathit{Addr} \times \mathit{Time} \\
\mathit{Control} &= \mathit{Exp} + \mathit{Val} \\
\rho \in \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Val} \\
\mathit{val} \in \mathit{Val} &= \mathit{Clo} + \mathit{Kont} \\
\kappa \in \mathit{Kont} &::= \mathbf{halt} \mid \mathbf{ar}(e, \rho, a) \mid \mathbf{fn}(\mathit{clo}, a) \\
\mathit{clo} \in \mathit{Clo} &::= (\lambda v.e) \times \mathit{Env} \\
a, b \in \mathit{Addr} & \quad \text{an infinite set of addresses} \\
t, u \in \mathit{Time} & \quad \text{an infinite set of timestamps}
\end{aligned}$$

Figure 3.2: State space of the CESK machine.

Transition Function The transition function for this CESK machine is parameterized by two functions:

$$\begin{aligned}
\mathit{tick} &: \Sigma_{\text{CESK}} \to \mathit{Time} \\
\mathit{alloc} &: \Sigma_{\text{CESK}} \to \mathit{Addr}
\end{aligned}$$

The definition of those functions depend on the analysis one wants to do. For the concrete case, we have $\mathit{Time} = \mathit{Addr} = \mathbb{Z}$, and define $\mathit{tick}(\langle \_, \_, \_, \_, t\rangle) = t + 1$ and $\mathit{alloc}(\langle \_, \_, \_, \_, t\rangle) = t$.

The transition function is formalized by the relation $(\to) \subset \Sigma_{\text{CESK}} \times \Sigma_{\text{CESK}}$. We write $\varsigma_{\text{CESK}} \to \varsigma'_{\text{CESK}}$ to mean that $(\varsigma_{\text{CESK}}, \varsigma'_{\text{CESK}}) \in (\to)$. We also use the following: $\kappa = \sigma(a)$, $b = \mathit{alloc}(\varsigma_{\text{CESK}})$, $u = \mathit{tick}(\varsigma_{\text{CESK}})$. We can now give the three cases that compose the transition function.

1. To evaluate a variable v, look up in the store the value associated with the address of this variable:

$$\langle v, \rho, \sigma, a, t\rangle \to \langle \sigma(\rho(v)), \rho, \sigma, a, u\rangle.$$

2. To evaluate an abstraction, create a closure by coupling the abstraction with the current environment:

$$\langle \texttt{(}\lambda v.e\texttt{)}, \rho, \sigma, a, t\rangle \to \langle (\texttt{(}\lambda v.e\texttt{)}, \rho), \rho, \sigma, a, u\rangle.$$

3. To evaluate an application $\texttt{(}e_0\ e_1\texttt{)}$, first evaluate the operator e_0 and push a continuation to evaluate the argument e_1 afterwards. To push a continuation, update the store with the new continuation (which lives at $b = alloc(\varsigma_{\text{CESK}})$) and update the address component of the state to be b:

$$\langle \texttt{(}e_0\ e_1\texttt{)}, \rho, \sigma, a, t\rangle \to \langle e_0, \rho, \sigma[b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u\rangle.$$

4. When an expression has been reduced to a value, the current continuation (which is $\kappa = \sigma(a)$) has to be applied:

   - if it is an *argument evaluation* continuation (**ar**), evaluate the argument stored inside this continuation and push a *function application* continuation:

   $$\langle clo, \rho, \sigma, a, t\rangle \to \langle e, \rho', \sigma[b \mapsto \mathbf{fn}(clo, a')], b, u\rangle \quad \text{if } \kappa = \mathbf{ar}(e, \rho', a'),$$

   - if it is a *function application* (**fn**) continuation, bind the function's argument to the computed value, evaluate the function's body and pop a continuation:

   $$\langle val, \rho, \sigma, a, t\rangle \to \langle e, \rho'[v \mapsto b], \sigma[b \mapsto val], a', u\rangle \quad \text{if } \kappa = \mathbf{fn}((\texttt{(}\lambda v.e\texttt{)}, \rho'), a').$$

To evaluate a program, the machine will continually apply one of these rule until the current control component is a value and the current continuation is the **halt** continuation.

Injection Function To evaluate an expression, we first need to *inject* it into an initial machine state. This is done using the injection function $\mathcal{I}_{\text{CESK}} : Exp \to \Sigma_{\text{CESK}}$, defined as:

$$\mathcal{I}_{\text{CESK}}(e) = \langle e, \varnothing, [a_{\mathbf{halt}} \mapsto \mathbf{halt}], a_{\mathbf{halt}}, t_0\rangle$$

where $a_{\mathbf{halt}}$ and t_0 respectively corresponds to an initial address and an initial timestamp (e.g. $a_{\mathbf{halt}} = 0$ and $t_0 = 0$ if we have $Time = Addr = \mathbb{Z}$). When the machine has a value in its control component and has **halt** as its current continuation, the evaluation is completed and the machine halts (no more transition rule can apply).

Evaluation Function We can now define the evaluation function $eval : Exp \to \mathcal{P}(\Sigma_{\text{CESK}})$ which computes the set of reachable states for an expression (and not only the final value, which would not be useful to do static analysis):

$$eval(e) = \{\varsigma_{\text{CESK}} \mid \mathcal{I}_{\text{CESK}}(e) \to^* \varsigma_{\text{CESK}}\}.$$

Example 5 (Concrete CESK evaluation)**.** We show how to use this concrete machine to evaluate the set of states reachable by the folowing program:

$$\texttt{((}\lambda x.x\texttt{) (}\lambda y.\texttt{(}\lambda z.y\texttt{)))}$$

First, this program is injected into an initial state:

$$\mathcal{I}(((\lambda x.x)\ (\lambda y.(\lambda z.y)))) = \langle ((\lambda x.x)\ (\lambda y.(\lambda z.y))), \varnothing, \{a_{\mathbf{halt}} \mapsto \mathbf{halt}\}, a_{\mathbf{halt}}, 0\rangle$$

The concrete transition function is then continually applied, leading to the following states, where we avoid repeating store values that remain the same:

$$\begin{aligned}
&\langle ((\lambda x.x)\ (\lambda y.(\lambda z.y))), \varnothing, \{a_{\mathbf{halt}} \mapsto \mathbf{halt}\}, a_{\mathbf{halt}}, 0\rangle \\
\to\ &\langle (\lambda x.x), \varnothing, \{a_{\mathbf{halt}}, a_0 \mapsto \mathbf{ar}((\lambda y.(\lambda z.y)), \varnothing, a_{\mathbf{halt}})\}, a_0, 1\rangle \\
\to\ &\langle ((\lambda x.x), \varnothing), \varnothing, \{a_{\mathbf{halt}}, a_0\}, a_0, 2\rangle \\
\to\ &\langle (\lambda y.(\lambda z.y)), \varnothing, \{a_{\mathbf{halt}}, a_0, a_2 \mapsto \mathbf{fn}(((\lambda x.x), \varnothing), a_{\mathbf{halt}})\}, a_2, 3\rangle \\
\to\ &\langle ((\lambda y.(\lambda z.y)), \varnothing), \varnothing, \{a_{\mathbf{halt}}, a_0, a_2\}, a_2, 4\rangle \\
\to\ &\langle x, \{x \mapsto a_4\}, \{a_{\mathbf{halt}}, a_0, a_2, a_4 \mapsto ((\lambda y.(\lambda z.y)), \varnothing)\}, a_{\mathbf{halt}}, 5\rangle \\
\to\ &\langle ((\lambda y.(\lambda z.y)), \varnothing), \{x \mapsto a_4\}, \{a_{\mathbf{halt}}, a_0, a_2, a_4\}, a_{\mathbf{halt}}, 6\rangle
\end{aligned}$$

The result of $\mathit{eval}(((\lambda x.x)\ (\lambda y.(\lambda z.y))))$ would thus be the set containing all those states.

3.1.3 Abstract Semantics

Testing membership of a state inside the set $\mathit{eval}(e)$ for a given expression is undecidable because of the halting problem. To solve this, we need to *abstract* the machine in order to compute a *finite approximation* of the set $\mathit{eval}(e)$. To do so, we need to adapt the state space so that it becomes finite, and to adapt the transition function to take this change into account.

State Space In the state space defined in Figure 3.2, the only sources of infiniteness are the addresses and the timestamps. By making them finite, the resulting state space also becomes finite (Figure 3.3). Note that the store is now a mapping from addresses to *sets* of values, meaning that multiple values can be stored at the same address.

$$\begin{aligned}
\hat{\varsigma}_{\text{CESK}} \in \hat{\Sigma}_{\text{CESK}} &= \widehat{\mathit{Control}} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Addr}} \times \widehat{\mathit{Time}} \\
\hat{c} \in \widehat{\mathit{Control}} &= \mathit{Exp} + \widehat{\mathit{Val}} \\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathit{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \rightharpoonup \mathcal{P}(\widehat{\mathit{Val}}) \\
\widehat{\mathit{val}} \in \widehat{\mathit{Val}} &= \widehat{\mathit{Clo}} + \widehat{\mathit{Kont}} \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &::= \mathbf{halt} \mid \mathbf{ar}(e, \hat{\rho}, \hat{a}) \mid \mathbf{fn}(\widehat{\mathit{clo}}, \hat{a}) \\
\widehat{\mathit{val}} \in \widehat{\mathit{Val}} &::= (\lambda v.e) \times \widehat{\mathit{Env}} \\
\hat{a}, \hat{b} \in \widehat{\mathit{Addr}} & \quad \text{a finite set of addresses} \\
\hat{t}, \hat{u} \in \widehat{\mathit{Time}} & \quad \text{a finite set of timestamps}
\end{aligned}$$

Figure 3.3: State space of the abstract CESK machine.

Transition Function The transition function mainly stays the same, except that values in the store now have to be *joined* instead of updated, in order to preserve soundness. The functions *tick* and *alloc* are also adapted:

$$\widehat{tick} : \hat{\Sigma}_{\text{CESK}} \rightarrow \widehat{Time}$$
$$\widehat{alloc} : \hat{\Sigma}_{\text{CESK}} \rightarrow \widehat{Addr}$$

A typical implementation for these functions consists of remembering the last k call-sites, where k is called the *polyvariance* of the analysis. $\widehat{tick}$ will update this list of call-sites, while $\widehat{alloc}$ will use it to generate a new address. This kind of analysis is called *k-CFA* (k-Control-Flow-Analysis) [Shi91].

The abstract transition function $(\widehat{\rightarrow}) : \hat{\Sigma}_{\text{CESK}} \times \hat{\Sigma}_{\text{CESK}}$ is defined as follows, with $\hat{\kappa} \in \hat{\sigma}(\hat{a})$, $\hat{b} = \widehat{alloc}(\hat{\varsigma}_{\text{CESK}})$, $\hat{u} = \widehat{tick}(\hat{\varsigma}_{\text{CESK}})$.

1. Variable evaluation has to take into account the fact that multiple values may live at the same address, and thus the transition function may leads to multiple states (one for each value stored at the address of the variable).

$$\langle v, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle \mathrel{\widehat{\rightarrow}} \langle \widehat{clo}, \hat{\rho}', \hat{\sigma}, \hat{a}, \hat{u}\rangle$$
$$\text{where } \widehat{clo} \in \hat{\sigma}(\hat{\rho}(v)).$$

2. Application evaluation needs to perform a join on the store to push a new continuation.

$$\langle (e_0\ e_1), \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle \mathrel{\widehat{\rightarrow}} \langle e_0, \hat{\rho}, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \hat{\rho}, \hat{a})], \hat{b}, \hat{u}\rangle.$$

3. Abstraction evaluation stays the same and creates a closure by coupling the abstraction with the current environment.

$$\langle (\lambda v.e), \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle \rightarrow \langle ((\lambda v.e), \hat{\rho}), \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{u}\rangle$$

4. Continuation evaluation also needs to perform a join on the store.

   - For an argument continuation:

$$\langle \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle \mathrel{\widehat{\rightarrow}} \langle e, \hat{\rho}', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(\widehat{clo}, \hat{a}')], \hat{b}, \hat{u}\rangle \ \text{ if } \hat{\kappa} = \mathbf{ar}(e, \hat{\rho}', \hat{a}'),$$

   - For a function application continuation:

$$\langle val, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle \mathrel{\widehat{\rightarrow}} \langle e, \hat{\rho}'[v \mapsto \hat{b}], \hat{\sigma} \sqcup [\hat{b} \mapsto val], \hat{a}', \hat{u}\rangle \ \text{ if } \hat{\kappa} = \mathbf{fn}(((\lambda v.e), \hat{\rho}'), \hat{a}').$$

> **Example 6** (Abstract CESK evaluation)**.** When we use the abstract interpreter, the main difference with the concrete interpreter is that an address can be associated with more than one value. Therefore, an application of the transition function may lead to multiple reachable states. For example, the transition function leads to two different states when applied to the following state, as two values are associated to the address a_2:
>
> $$\langle x, \{x \mapsto a_2\}, \{a_2 \mapsto \{((\lambda x.x)), \varnothing), ((\lambda y.(\lambda z.y)), \varnothing)\}, \ldots\}, a_{\mathbf{halt}}, t\rangle$$
>
> The two resulting states are:
>
> $$\langle ((\lambda x.x)), \varnothing), \{x \mapsto a_2\}, \{a_2 \mapsto \{((\lambda x.x)), \varnothing), ((\lambda y.(\lambda z.y)), \varnothing)\}, \ldots\}, a_{\mathbf{halt}}, u\rangle$$
> $$\langle ((\lambda y.(\lambda z.y)), \varnothing), \{x \mapsto a_2\}, \{a_2 \mapsto \{((\lambda x.x)), \varnothing), ((\lambda y.(\lambda z.y)), \varnothing)\}, \ldots\}, a_{\mathbf{halt}}, u\rangle$$

Injection Function The injection function is adapted to the abstract state space:

$$\widehat{\mathcal{I}}(e) = \langle e, \varnothing, [\hat{a}_{\mathbf{halt}} \mapsto \mathbf{halt}], \hat{a}_{\mathbf{halt}}, \hat{t}_0 \rangle$$

where $\hat{a}_{\mathbf{halt}}$ and $\hat{t}_0$ are the abstract conterpart of $a_{\mathbf{halt}}$ and t_0 of Section 3.1.2.

Evaluation Function We can finally define the abstract evaluation function $\widehat{eval}(e) : Exp \to \mathcal{P}(\hat{\Sigma}_{\text{CESK}})$:

$$\widehat{eval}(e) = \{\hat{\varsigma}_{\text{CESK}} \mid \widehat{\mathcal{I}}(e) \mathrel{\widehat{\rightarrow}^*} \hat{\varsigma}_{\text{CESK}}\}.$$

Using this function, it is possible to build a *finite* set approximating all the possible states reachable during the execution of an expression, since this abstraction is *sound* (Theorem 2 of [VHM10]). By examining the set of reachable states, it is thus possible to prove certain properties about the program, such as the absence of certain classes of bugs.

In practice, it is sometimes more useful to reason about the *graph* of reachable states, which over-approximate every possible *path* that the execution of a program can take. The exploration of the state graph can be done by any graph exploration method. The initial node is the injected expression, and the successors of a node are the result of applying the transition function once to this node. We can thus build and explore the graph using a conventional method such as a breadth-first search (BFS) or a depth-first search (DFS).

Example 7 (CESK-based analysis of the `error` expression)**.** Suppose we add an `error` expression to the language of Figure 3.1, that leads to a runtime error when evaluated. Programs can contain references to this `error` expression without leading to errors, for example:

$$\texttt{(((}\lambda y\texttt{.(}\lambda z\texttt{.(}y\ z\texttt{))) (}\lambda x.x\texttt{)) (}\lambda x\texttt{.error))}$$

This program reduces to the following expression, after two application of the transition function.

$$\texttt{(}\lambda x\texttt{.error)}$$

This expression cannot be reduced anymore and the `error` expression has never been evaluated. To check whether such a program is exempt of errors generated by the `error` expression, we can build the set of reachable states using $\widehat{eval}$, and check that this set does not contain any state with `error` as its control component:

$$NoError(e) \Leftrightarrow \nexists \langle \texttt{error}, \_, \_, \_, \_ \rangle \in \widehat{eval}(e)$$

If the abstraction is precise enough, this property should hold when we evaluate this program with our abstract CESK machine. If this property holds and the abstraction is sound, we know that the program will never result in such an error.

However, if for example the functions $\texttt{(}\lambda x.x\texttt{)}$ and $\texttt{(}\lambda x\texttt{.error)}$ were stored at the same address in the store, one of the reachable state will have `error` as its control component, while this program will never reach this state when evaluated with the concrete interpreter. Such an abstraction would not allow us to detect that this program is exempt of such errors, because it is not precise enough. Thus, if the property does not hold, it does not mean that there will be such an error, just that this error *may* happen. Programs that are exempt of this error but for which *NoError* does not hold are called *false positives.* Since we want to perform a sound analysis, the approximation is *conservative* and is thus subject to false positives.

3.1.4 Existing Refinements

The set of reachable states computed by $\widehat{eval}$ will generally contain many non-needed states and thus slow down the analysis. Various techniques can be used to improve the size of this set, without losing precision. We describe here three existing techniques that either reduce the size of the reachable states set or even improve the precision of the analysis: abstract counting, abstract garbage collection and state subsumption checking.

Abstract Counting The use of a join instead of an update in the store of the abstract machine leads to a much bigger state space and a lot of unnecessary work. Abstract counting [MS06] consists of replacing most of those joins by updates when possible. This is done by counting the number of times an abstract address is allocated. When an address is allocated only once, we can replace every join by an update without losing soundness. When an address is allocated more than once, it means that this address corresponds to multiple concrete variables and cannot be safely updated, and the join is needed.

Abstract Garbage Collection Since the continuations and variable bindings live in the store and are never reclaimed, the store becomes filled with old continuations and other non-reachable variables. New values might be assigned the same address as one of those unused value, which will result in a loss of precision. Abstract garbage collection [MS06, VHM10] is the abstract counterpart of concrete garbage collection, and allows to reclaim those unused addresses, leading to an increase in both the precision and running time of the analysis. The garbage collector uses the concept of *live locations*, i.e. locations that are still accessible in the store and should not be reclaimed. Those live locations can be computed for the concrete machine with $\mathcal{LL}_\sigma$ and $\mathcal{LL}^\rho$ defined as:

$$\begin{aligned}
\mathcal{LL}_\sigma(e) &= \varnothing\\
\mathcal{LL}_\sigma(\mathit{clo}) &= \mathcal{LL}^\rho(e) \text{ where } (e,\rho) = \mathit{clo}\\
\mathcal{LL}_\sigma(\mathbf{halt}) &= \varnothing\\
\mathcal{LL}_\sigma(\mathbf{fn}(\mathit{clo},a)) &= \{a\} \cup \mathcal{LL}_\sigma(\mathit{clo}) \cup \mathcal{LL}_\sigma(\sigma(a))\\
\mathcal{LL}_\sigma(\mathbf{ar}(e,\rho,a)) &= \{a\} \cup \mathcal{LL}^\rho(e) \cup \mathcal{LL}_\sigma(\sigma(a))
\end{aligned}$$

$$\mathcal{LL}^\rho(e) = \mathit{range}(\rho)|\mathbf{fv}(e)$$

The live locations can also be computed for the abstract machine by replacing each occurrence of $\mathcal{LL}_\sigma(\sigma(a))$ by:

$$\bigcup_{\mathit{val}\in\sigma(a)} \mathcal{LL}_\sigma(\mathit{val})$$

With the ability to compute live locations, we can define a GC machine that computes reachable addresses, whose state is composed of a *grey* set $\mathcal{G}$ (reachable addresses not yet visited), a *black* set $\mathcal{B}$ (reachable addresses visited), and the store σ. This machine's states are thus in:

$$\Sigma_{GC} = \mathcal{P}(\mathit{Addr}) \times \mathcal{P}(\mathit{Addr}) \times \mathit{Store}$$

The transition function $(\rightarrow_{GC}) : \Sigma_{GC} \times \Sigma_{GC}$ takes an address a from $\mathcal{G}$, computes its live locations, add them to $\mathcal{G}$ and add a to $\mathcal{B}$:

$$\begin{aligned}
&\langle \mathcal{G}, \mathcal{B}, \sigma\rangle \rightarrow_{GC} \langle \mathcal{G}', \mathcal{B}', \sigma\rangle\\
&\text{where } a \in \mathcal{G}\\
&\qquad \mathcal{B}' = \mathcal{B} \cup \{a\}\\
&\qquad \mathcal{G}' = \mathcal{G} \cup \mathcal{LL}_\sigma(\sigma(a)) \setminus \mathcal{B}'
\end{aligned}$$

We also define a function $\mathcal{R} : \Sigma_{\text{CESK}} \to \mathcal{P}(Addr)$ that computes the set of reachable addresses for a machine state. This is not done in [VHM10], but it simplifes the adaptation of this abstract garbage collector for the PCESK machine, in Section 4.4.4. This function contains the main logic of the abstract garbage collection and is defined as:

$$\begin{aligned}\mathcal{R}(\langle e, \rho, \sigma, a, t\rangle) &= \mathcal{L}\\ \text{where } \langle \mathcal{LL}^{\rho}(e) \cup \mathcal{LL}_{\sigma}(\sigma(a)), \{a\}, \sigma\rangle &\to^{*}_{GC} \langle \varnothing, \mathcal{L}, \sigma\rangle\end{aligned}$$

Finally, we can define a new transition function $(\to') : \Sigma_{\text{CESK}} \times \Sigma_{\text{CESK}}$ for the CESK machine that will perform the garbage collection:

$$\begin{aligned}\langle e, \rho, \sigma, a, t\rangle &\to' \langle e, \rho, \sigma|\mathcal{L}, a, t\rangle\\ &\text{where } \mathcal{L} = \mathcal{R}(\langle e, \rho, \sigma, a, t\rangle)\end{aligned}$$

Note that this transition is defined for the concrete CESK machine, but it is directly applicable to the abstract machine by replacing the concrete elements by abstract ones. When running the CESK machine, this new transition $(\to')$ will be made after each CESK transition $(\to)$. Doing it after and not before is important because it will have more impact on reducing the size of the state space (the difference for the CESK machine is small, but becomes much bigger for the PCESK machine).

State Subsumption When exploring the abstract state graph, it can happen that at some point we visit an abstract state that is more specific than a previously visited state. Such a state can be skipped because all its successors have already been reached by the more general state. This technique is described for graph transition systems in [ZR12], but can be adapted for abstract interpretation.

First, we need to define the partial orders on environments and stores as follows.

$$\begin{aligned}&\hat{\rho}_1 \sqsubseteq \hat{\rho}_2 \Leftrightarrow \forall v \in dom(\hat{\rho}_1), \hat{\rho}_1(v) = \hat{a} \Rightarrow \hat{\rho}_2(v) = \hat{a}\\ &\hat{\sigma}_1 \sqsubseteq \hat{\sigma}_2 \Leftrightarrow \forall \hat{a} \in dom(\hat{\sigma}_1), \widehat{val}_1 \in \hat{\sigma}_1(\hat{a}) \Rightarrow \exists \widehat{val}_2 \in \hat{\sigma}_2(\hat{a}) \wedge \widehat{val}_1 \sqsubseteq \widehat{val}_2\end{aligned}$$

An abstract state $\hat{\varsigma}_{\text{CESK},1}$ is *subsumed* by another abstract state $\hat{\varsigma}_{\text{CESK},2}$, denoted by $\hat{\varsigma}_{\text{CESK},1} \sqsubseteq \hat{\varsigma}_{\text{CESK},2}$ if the following conditions hold:

- their control component is equal,
- the environment component $\hat{\rho}_1$ of $\hat{\varsigma}_{\text{CESK},1}$ is subsumed by the environment component $\hat{\rho}_2$ of $\hat{\varsigma}_{\text{CESK},2}$, i.e. $\hat{\rho}_1 \sqsubseteq \hat{\rho}_2$,
- the store component $\hat{\sigma}_1$ of $\hat{\varsigma}_{\text{CESK},1}$ is subsumed by the store component $\hat{\sigma}_2$ of $\hat{\varsigma}_{\text{CESK},2}$, i.e. $\hat{\sigma}_1 \sqsubseteq \hat{\sigma}_2$,
- their continuation component is equal, and
- their timestamp component is equal.

When we reach an abstract state $\hat{\varsigma}_{\text{CESK},1}$ during the graph exploration, and we have already visited a state $\hat{\varsigma}_{\text{CESK},2}$ such that $\hat{\varsigma}_{\text{CESK},1} \sqsubseteq \hat{\varsigma}_{\text{CESK},2}$, we can completely skip this state because all its successors have already been reached and either have been visited or are scheduled for being visited later.

Depending on the implementation of the state space exploration, we may first reach *larger* states that will subsume other states during the rest of the exploration. If we reach those larger states sooner, the resulting state space will be smaller because the exploration of those larger states prunes off the exploration of any state encountered later that they subsume. [ZR12] argues that a depth-first exploration tends to reach sooner larger states as the abstract state tends to be more and more abstracted as the transition function is applied to it. However, in our case it is the opposite, as we will show in the benchmarks of Section 7.2.

3.1.5 Implementation

We now describe how to implement a minimal CESK machine, without any refinements, in OCaml.

Concrete CESK Machine Implementation First, we have to describe the state space in terms of OCaml types, which requires to reorder the type definitions to only refer to previously defined types. Note that the store and component environment are implemented as parameterized maps[1] (instead of OCaml's default functorized maps) for the sake of brevity.

```
type var = string

type exp =
| Var of var
| App of exp * exp
| Abs of var * exp

type addr = int

type time = int

type env = (var, addr) BatMap.t

type clo = var * exp * env

type kont =
| Halt
| Arg of exp * env * addr
| Fun of clo * addr


type value =
| Clo of clo
| Kont of kont

type store = (addr, value) BatMap.t

type control =
| Exp of exp
| Val of value

type state = {
  control : control;
  env : env;
  store : store;
  addr : addr;
  time : time;
}
```

The manipulation of the environment and the store is abstracted from their implementation using the following functions. Even though they have the same implementation for the environments or for the store, we keep them with different names to ease further changes to the store.

```
let env_empty =
  BatMap.empty

let env_extend m k v =
  BatMap.add k v m

let env_lookup m k =
  BatMap.find k m


let store_empty =
  BatMap.empty

let store_set m k v =
  BatMap.add k v m

let store_lookup m k =
  BatMap.find k m
```

We will use the definition of *tick* and *alloc* given previously.

```
let tick state =
  state.time + 1

let alloc state =
  state.time
```

We can then implement the transition function as a `step` function, which pattern matches the control component of the state to find which transition rule to use. When reaching the final state, it returns the same state as it was given.

[1] All the OCaml modules starting with `Bat` are part of OCaml's extended standard library, *batteries* (`http://batteries.forge.ocamlcore.org/`), which is installable via OPAM (`http://opam.ocamlpro.com/`).

```
let step s =
  match s.control with
  | Exp e ->
    begin match e with
    | Var v ->
      { s with
        control = Val
            (store_lookup s.store
               (env_lookup s.env v));
        time = tick s }
    | Abs (v, e) ->
      { s with
        control = Val (Clo (v, e, s.env));
        time = tick s }
    | App (e0, e1) ->
      let b = alloc s in
      { s with
        control = Exp e0;
        store = store_set s.store b
            (Kont
               (Arg (e1, s.env, s.addr)));
        addr = b;
        time = tick s }
    end
  | Val value ->
    let b = alloc s
    and k = store_lookup s.store s.addr in
    begin match k with
    | Kont (Arg (e, env', a')) ->
      begin match value with
      | Clo clo ->
        { control = Exp e;
          env = env';
          store = store_set s.store b
              (Kont (Fun (clo, a')));
          addr = b;
          time = tick s }
      | _ -> failwith "Invalid state"
      end
    | Kont (Fun ((v, e, env'), a')) ->
      { control = Exp e;
        env = env_extend env' v b;
        store = store_set s.store b value;
        addr = a';
        time = tick s }
    | Kont Halt ->
      s
    | _ -> failwith "Invalid state"
    end
```

The injection function builds the initial state, and uses a fixed address for the **halt** continuation, set to −1 to ensure that it will not clash with other addresses (which are positive integers).

```
let a_halt = -1

let inject e =
  { control = Exp e;
    env = env_empty;
    store = store_set store_empty
        a_halt (Kont Halt);
    addr = a_halt;
    time = 0 }
```

For this concrete interpreter, the evaluation function will not build the set of reachable states but instead apply the transition function until it reaches a final state (i.e. a state which has a value as its control component and whose continuation is **halt**). The function `extract` will recognize this final state and extract its return value.

```
let eval e =
  let rec eval_aux s =
    match extract s with
    | Some v -> v
    | None -> eval_aux (step s)
  in
  eval_aux (inject e)
```

We can for example use this `eval` function to evaluate the program given in example 5, which gives the expected result.

```
# eval (App (Abs ("x", Var "x"), Abs ("y", Abs ("z", Var "y"))));;
  - : value = Clo ("y", Abs ("z", Var "y"), <abstr>)
```

Abstract CESK Machine Implementation The implementation of the abstract CESK machine is similar to the concrete one described previously. The main difference is that the set of addresses is finite, which implies that store can now have multiple values stored at the same address. We reflect this in the code by changing the domain of the store to a list of values.

```
type store = (addr, value list) BatMap.t
```

While the implementation of the environments stays the same, the store now use joins instead of updates, so `store_set` becomes `store_join`.

```
let store_join m k v =
  if BatMap.mem k m then
    BatMap.modify k (fun v' -> v::v') m
  else
    BatMap.add k [v] m
```

To demonstrate the effects of the loss of precision due to the finiteness of the abstract state space, we restrict the timestamps to integers between 0 and 3, which will have as a result to also restrict the addresses to integers in this range (and -1, for the **halt** continuation).

```
let tick state =
  (state.time + 1) mod 4
```

The change of structure of the store requires us to change the transition function, as it was described previously. Since the abstract transition function $\widehat{\rightarrow}$ is now non-deterministic, the `step` function does not return a single state anymore, but a list of states. Invalid states are now ignored because they might arise due to a loss of precision.

```
let step s =
  match s.control with
| Exp e ->
  begin match e with
  | Var v ->
    List.map
      (fun value ->
         { s with
             control = Val value;
             time = tick s })
      (store_lookup s.store
         (env_lookup s.env v))
  | Abs (v, e) ->
    [{ s with
        control = Val (Clo (v, e, s.env));
        time = tick s }]
  | App (e0, e1) ->
    let b = alloc s in
    [{ s with
        control = Exp e0;
        store = store_join s.store b
            (Kont (Arg (e1, s.env, s.addr)));
        addr = b;
        time = tick s }]
  end
| Val value ->
  let b = alloc s
  and ks = store_lookup s.store s.addr in
  List.concat @@
  List.map (function
      | Kont (Arg (e, env', a')) ->
        begin match value with
        | Clo clo ->
          [{ control = Exp e;
             env = env';
             store = store_join s.store b
                 (Kont (Fun (clo, a')));
             addr = b;
             time = tick s }]
        | _ -> []
        end
      | Kont (Fun ((v, e, env'), a'))
        ->
        [{ control = Exp e;
           env = env_extend env' v b;
           store = store_join s.store
               b value;
           addr = a';
           time = tick s }]
      | Kont Halt ->
        [s]
      | _ -> [])
    ks
```

The injection function has to use `store_join` instead of the old `store_set`.

```
let inject e =
  { control = Exp e;
    env = env_empty;
    store = store_join store_empty
        a_halt (Kont Halt);
    addr = a_halt;
    time = 0 }
```

Because the `step` function leads to multiple state, the state space is not a trace anymore as it was in the concrete case, but a graph (each state can lead to more than one state). We thus have to explore this entire graph to find the final states for the `eval` function. This is done using a breadth-first search using a queue[2].

[2] In fact, we use a Deque in which we only push one one side and pop from the other side, because OCaml does not have a functional queue implementation but batteries' `BatDeque` is functional.

```
let eval e =
  let rec eval_aux q r f =
    match BatDeque.front q with
    | Some (s, q') ->
      begin match extract s with
      | Some v ->
        eval_aux
          q'
          r
          (BatSet.add v f)
      | None ->
        let s' = step s in
        eval_aux
          (BatDeque.append_list q' s')
          (BatSet.union
             (BatSet.of_list s')
             r)
          f
      end
    | None -> f
  in
  BatSet.elements
    (eval_aux
       (BatDeque.cons (inject e)
          BatDeque.empty)
       BatSet.empty
       BatSet.empty)
```

We can finally test this implementation on the program of Example 5, and see that indeed because of the small number of possible addresses, we lost precision, as we now have two possible results for this example.

```
# eval (App (Abs ("x", Var "x"), Abs ("y", Abs ("z", Var "y"))));;
- : value list = [Clo ("y", Abs ("z", Var "y"), <abstr>);
                  Kont (Arg (Abs ("y", Abs ("z", Var "y")), <abstr>, -1))]
```

We can also tune the `eval` function to produce a state graph instead of just the final states. This is what is done our implementation of the CESK machine. Figure 3.4 represents such a graph for the following program (assuming we add support for `letrec`, numbers and strings to the CESK machine).

```
(letrec ((count (lambda (n) (if (= n 0) "done" (count (- n 1))))))
  (count 200))
```

Every state is represented by its control component (but contains all the components of a CESK state), and edges between the states are labeled with the identifier of the continuation that is pushed or popped (`+` means pushed, `-` means popped, `e` means that the continuation did not change). Green nodes are states whose control component is a value, while red nodes are states whose control component is an expression to evaluate. Note that the 200 iterations are abstracted into a cycle from which there is an exit path, but we cannot conclude from this graph how many iterations will be performed.

3.2 Administrative Normal Form

In the language defined in Figure 3.1, the evaluation of an application might require multiple steps in order to evaluate the argument. Consider for example the program `((`$\lambda x.x$`) ((`$\lambda y.y$`) (`$\lambda z.z$`)))` which requires first to evaluate the inner application `((`$\lambda y.y$`) (`$\lambda z.z$`))` before evaluating the outer application.

On the other hand, some other constructs such as an abstraction `(`$\lambda x.e$`)` can be evaluated *atomically*, without needing to remember which part of an expression we are currently evaluating. In the case of the CESK machine, an expression is *atomic* if it can be evaluated without needing to create a new continuation. It is then possible to define an *atomic evaluation function* that will evaluate such expressions without needing the transition function of the machine.

It is also possible to transform a program such as `((`$\lambda x.x$`) ((`$\lambda y.y$`) (`$\lambda z.z$`)))` into a program that requires its argument to be evaluated atomically, by introducing `let` statements. In this case, the previous program could be rewritten as `(let ((`a` ((`$\lambda y.y$`) (`$\lambda z.z$`)))) ((`$\lambda x.x$`) `a`))` This is known as Administrative Normal Form (ANF) [FSDF93] and it is an intermediate representation of programs used as an alternative to Continuation Passing Style (CPS). Any program can be expressed in ANF without losing expressiveness.

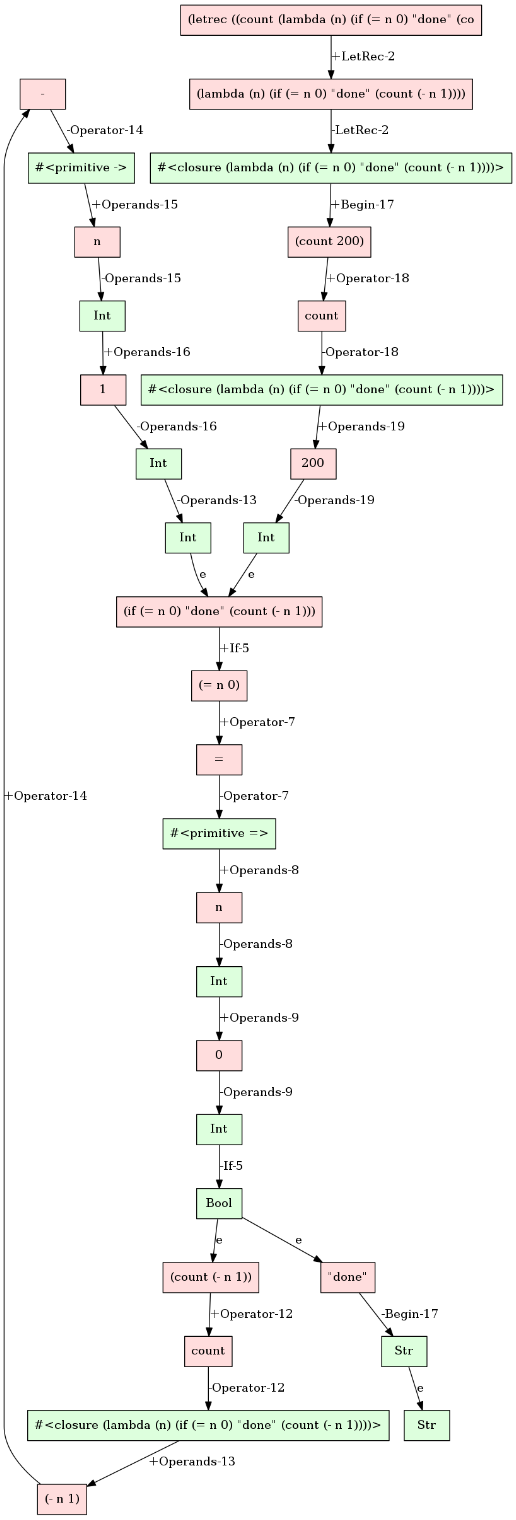

Figure 3.4: State graph of a program that contains a loop that terminates.

In the setting of concurrent programming, having a way to identify atomic expressions allows us to give guarantees about the atomicity of other expressions, as we will see in Chapter 4. If a special form requires all its arguments to be atomic, the evaluation of the special form itself might be done in only one step of the transition function, without creating new continuations for the evaluation of its arguments. It is thus possible to be sure that the evaluation of such a special form will not be dependent on the possible different interleavings between multiple threads. This will be useful when defining the semantics of `join` and `cas` in Section 4.2.

3.3 Conclusion

This chapter introduced material needed in the rest of this dissertation. The CESK machine allows us to statically build a graph of states reachable by a program. This state graph can be explored to perform static analysis. The PCESK machine introduced in the next chapter is based on this CESK machine. The PCESK machine adds support for multiple execution threads by capturing multiple CESK machines without their store component in a single state, together with one shared store per state.

We also described Administrative Normal Form, which will become useful when reasoning about the semantics of parallel operations in the PCESK machine. Requiring that the argument of a special form is atomic allows us to guarantee that this special form itself is atomic. Atomicity will be useful for `cas` (compare-and-swap), which is inherently atomic, but also to avoid coupling between the CESK and PCESK machines for other parallel special forms (specifically `join`).

Chapter 4

A Concurrent Scheme and its PCESK-based Semantics

This chapter starts by introducing the CScheme language (Section 4.1) we will be using in the rest of this work. Inspired by Might and Van Horn's work [Mig11, MVH11], we take a simplified version of Scheme on top of which we add three special forms to support concurrency through shared memory: `spawn`, `join`, and `cas`. The concrete semantics of this language are given in Section 4.2, and heavily rely on an underlying CESK machine that handles all the sequential constructs of CScheme. This sequential CESK machine is similar to the CESK machine defined in Section 3.1 but adds more constructs (`letrec`, `begin`, `if`, `set!`, multiple-arguments functions, and more data types). (The sequential CESK machine will not be described in detail here, as it is not the focus of this work.) We then abstract the PCESK machine in Section 4.3, and Section 4.4 presents refinements to improve analysis time and precision. Finally, Section 4.5 gives a brief summary of how the PCESK machine can be used to perform static analysis.

4.1 The Language: CScheme

Before defining the analyses, we first need a language that supports concurrency. We will use a simplified version of Scheme with special forms to handle concurrency through shared memory, which we call the CScheme language (for *Concurrent Scheme*). The language grammar is given in Figure 4.1.

This language supports four primitive values: numbers, booleans, functions, and thread identifiers. Functions are created using the `lambda` special form and can have multiple arguments (while it is not necessary, it simplifies the writing of programs). CScheme has two kinds of expressions: *atomic* expressions (*AExp*), and *compound* expressions (*CExp*).

Atomic expressions consist of the primitive values that can directly be constructed in the language, and of variable identifiers, which require a value lookup to evaluate.

Compound expressions contain multiple argument function calls, Scheme's classical `begin`, `letrec`, `if`, and `set!` special forms, and introduce three new special forms:

- `(spawn` e`)` creates a new thread to evaluate the expression e and immediately returns the corresponding thread id,
- `(join` $æ$`)` expects $æ$ to evaluate to a thread identifier t_1, blocks the execution of the current thread until t_1 finishes its execution, and returns the value of the last expression evaluated in t_1,
- `(cas` v $æ_{old}$ $æ_{new}$`)` compares the values of v and $æ_{old}$ and:
  - if they are equal, updates the value of v to $æ_{new}$, and returns `#t`,
  - else, returns `#f`.

$$
\begin{array}{rcl}
v \in \mathit{Var} & & \text{a set of identifiers} \\
n \in \mathbb{N} & & \text{a set of number literals} \\
b \in \mathbb{B} & ::= & \texttt{\#t} \mid \texttt{\#f} \\
e \in \mathit{Exp} & ::= & \textit{æ} \mid \mathit{cexp} \\
f, \textit{æ} \in \mathit{AExp} & ::= & \mathit{lam} \mid v \mid n \mid b \\
\mathit{lam} \in \mathit{Lam} & ::= & \texttt{(lambda (}v_1 \ldots v_n\texttt{)}\ e_1 \ldots e_n\texttt{)} \\
\mathit{cexp} \in \mathit{CExp} & ::= & \texttt{(}f\ e_1 \ldots e_n\texttt{)} \\
 & \mid & \texttt{(begin }e_1 \ldots e_n\texttt{)} \\
 & \mid & \texttt{(letrec ((}v_1\ e_1\texttt{)}\ \ldots\texttt{)}\ e_{body_1} \ldots e_{body_n}\texttt{)} \\
 & \mid & \texttt{(if }e_{cond}\ e_{cons}\ e_{alt}\texttt{)} \\
 & \mid & \texttt{(set! }v\ e\texttt{)} \\
 & \mid & \texttt{(spawn }e\texttt{)} \\
 & \mid & \texttt{(join }\textit{æ}\texttt{)} \\
 & \mid & \texttt{(cas }v\ \textit{æ}_{old}\ \textit{æ}_{new}\texttt{)}
\end{array}
$$

Figure 4.1: Grammar of CScheme.

When more than one thread is running, the context switches between threads happen nondeterministically. With those three special forms, we are able to implement multi-threaded programs that are safe, such as in Example 8. This program always increases the counter two times and returns `2`, no matter the order nor the frequency of context switches.

Example 8 (Parallel counter implemented with `cas`)**.** The following program creates a shared counter that gets incremented in two threads:

```
(letrec ((counter 0)
         (inc (lambda ()
                (letrec ((old counter)
                         (new (+ old 1)))
                  (if (cas counter old new)
                    #t
                    (inc)))))
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  counter)
```

This program can be written differently, using a *lock*. A lock is an object that can either be locked or not. Two operations can be done on a lock: it can be *acquired* by a thread, and it can be *released* by a thread that previously acquired it. When a lock is acquired by a thread, it becomes locked and cannot be acquired until it is released. Example 9 implements the simplest form of locks (a boolean variable that can be set to true or false, indicating whether the lock is locked or not) using `cas` as a building block.

Example 9 (Parallel counter implemented with locks)**.** The following program is a typical implementation of a shared counter using locks:

```
(letrec ((lock #f)
         (acquire (lambda ()
                    (if (cas lock #f #t)
                      nil
                      (acquire))))
         (release (lambda ()
                    (set! lock #f)))
         (counter 0)
         (inc (lambda ()
                (acquire)
                (set! counter (+ counter 1))
                (release)))
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  counter)
```

The style of programming concurrent programs with `cas` without using locks such as in Example 8 is known as *lock-free* programming, because one thread can never block another. If traditional *locks* were (incorrectly) used or simulated using `cas` it could however be the case.

However, many programs use locks, and it is thus necessary to be able to analyze programs that use locks. Fortunately, `cas` allows us to simulate locks as Example 9 showed.

The choice of `cas` as a synchronization primitive is thus justified by the fact that it is possible to implement many other popular synchronization primitives: locks as shown here, semaphores and condition variables [Bir04], as well as software transactional memory [Fra04].

4.2 Concrete Semantics: The PCESK Machine

We define the semantics of CScheme using a parallel CESK machine, PCESK, while staying consistent with the CESK machine previously defined. This PCESK machine relies heavily on the underlying sequential CESK machine. The only transition rules that have to be added are the ones related to the concurrency constructs.

4.2.1 Concrete State Space

The concrete state space of the PCESK machine is given in Figure 4.2.

This machine follows the shared memory model by having multiple threads and only one store. Each thread is identified by a thread identifier and is represented by a *context*, which is similar to a CESK machine without the store component. Note that thread identifiers are first-class values because they are returned by `spawn` and manipulated by `join`. Thread identifiers are also considered as addresses, as it will be explained in Section 4.2.4.

4.2.2 Atomic Evaluation Function

The concrete state space contains two kinds of expressions:

1. atomic expressions (*AExp*) are expressions that can be evaluated in finite time without using the stack, and

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathit{Threads} \times \mathit{Store} \\
T \in \mathit{Threads} &= \mathit{TID} \rightharpoonup \mathit{Context} \\
c \in \mathit{Context} &= \mathit{Control} \times \mathit{Env} \times \mathit{Addr} \times \mathit{Time} \\
\mathit{Control} &= \mathit{Exp} + \mathit{Val} \\
\rho \in \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Addr} \\
\kappa \in \mathit{Kont} ::&= \mathbf{rator}((e_0, \ldots, e_n), \rho, a) \\
&\mid \mathbf{rand}(\mathit{val}, (e_0, \ldots, e_n), (\mathit{val}_0, \ldots, \mathit{val}_n), \rho, a) \\
&\mid \mathbf{begin}((e_0, \ldots, e_n), \rho, a) \\
&\mid \mathbf{letrec}(a_{\mathit{binding}}, ((v_0, e_0), \ldots, (v_n, e_n)), \\
&\qquad\qquad (e_{\mathit{body},0}, \ldots, e_{\mathit{body},n}), \rho, a) \\
&\mid \mathbf{if}(e_{\mathit{cond}}, e_{\mathit{cons}}, e_{\mathit{alt}}, \rho, a) \\
&\mid \mathbf{set!}(v, \rho, a) \\
&\mid \mathbf{halt} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Val} \\
\mathit{val} \in \mathit{Val} &= \mathit{Clo} + \mathit{Bool} + \mathit{Num} + \mathit{Kont} + \mathit{TID} \\
\mathit{clo} \in \mathit{Clo} &= \mathit{Lam} \times \mathit{Env} \\
t \in \mathit{Time} & \quad \text{an infinite set of timestamps} \\
a \in \mathit{Addr} & \quad \text{an infinite set of addresses, includes } \mathit{TID} \\
\mathit{tid} \in \mathit{TID} & \quad \text{an infinite set of thread identifiers}
\end{aligned}
$$

Figure 4.2: Concrete state space of the PCESK machine.

2. compound expressions (*CExp*) are expressions that might require multiple steps to evaluate (e.g. a function call requires to evaluate the operator and all the operands before actually applying the function call).

We will need a function $\mathcal{E} : \mathit{AExp} \times \mathit{Env} \times \mathit{Store} \to \mathit{Val}$ that can evaluate atomic expressions in one step:

$$
\begin{aligned}
\mathcal{E}(n, \rho, \sigma) &= n \\
\mathcal{E}(b, \rho, \sigma) &= b \\
\mathcal{E}(v, \rho, \sigma) &= \sigma(\rho(v)) \\
\mathcal{E}(\mathit{lam}, \rho, \sigma) &= (\mathit{lam}, \rho)
\end{aligned}
$$

4.2.3 Conversion Functions

Due to the close relation between the PCESK's *Context* and CESK's Σ_{CESK}, we will define conversion functions between them. We also suppose to have a CESK machine which supports the whole set of values and continuations of Figure 4.2.

The two conversion functions are $\mathcal{S} : \mathit{Context} \times \mathit{Store} \to \Sigma_{\text{CESK}}$ which converts a parallel context and a store into a sequential state, and $\mathcal{C} : \Sigma_{\text{CESK}} \to \mathit{Context} \times \mathit{Store}$ which converts a sequential state into a parallel context and a store. They are defined as follows.

$$
\begin{aligned}
\mathcal{S}(\langle e, \rho, a, t \rangle, \sigma) &= \langle e, \rho, \sigma, a, t \rangle \\
\mathcal{C}(\langle e, \rho, \sigma, a, t \rangle) &= (\langle e, \rho, a, t \rangle, \sigma)
\end{aligned}
$$

4.2.4 Transition Function

Since we have a CESK machine that handles all the sequential transition through its transition function $(\rightarrow) \subset \Sigma_{\text{CESK}} \times \Sigma_{\text{CESK}}$, the PCESK transition function $(\Rightarrow) \subset \Sigma \times \Sigma$ needs to define how to use $(\rightarrow)$ for the sequential cases, and how to evaluate `spawn` and `join`. The `cas` operation is sequential and can be supported by $(\rightarrow)$ directly, by adding the following rule:

$$
\begin{aligned}
\overbrace{\langle \texttt{(cas}\ v\ \text{\ae}_{old}\ \text{\ae}_{new}\texttt{)}, \rho, \sigma, a, t\rangle}^{\varsigma} &\rightarrow \langle \texttt{\#t}, \rho, \sigma[\rho(v) \mapsto \mathcal{E}(\text{\ae}_{new}, \rho, \sigma)], a, u\rangle \text{ if } \sigma(\rho(v)) = \mathcal{E}(\text{\ae}_{old}, \rho, \sigma)\\
&\rightarrow \langle \texttt{\#f}, \rho, \sigma, a, u\rangle \text{ else}
\end{aligned}
$$

where $u = \mathit{tick}(\varsigma)$. Note that we ensure the atomicity of `cas` because its arguments should be atomic expressions. No new continuation will have to be created to evaluate those arguments, and `cas` can thus be evaluated in only one step and stay independent of the threads interleaving order.

We also need a function $\mathit{newtid} : \mathit{Context} \times \mathit{Threads} \rightarrow \mathit{TID}$ that allocates new thread identifiers, which is similar to the *alloc* function used in the CESK machine. For the concrete case, with $\mathit{TID} = \mathbb{Z}$, an adequate definition would be, for example:

$$\mathit{newtid}(c, T) = |\mathit{dom}(T)| + 1$$

We can then define the parallel transition function $(\Rightarrow)$.

- If one of the threads can do a sequential step, the PCESK machine can do a step.

$$
\begin{gathered}
\langle T[\mathit{tid} \mapsto c], \sigma\rangle \Rightarrow \langle T[\mathit{tid} \mapsto c'], \sigma'\rangle\\
\text{if } \mathcal{S}(c, \sigma) \rightarrow \varsigma_{\text{CESK}} \text{ and } \langle c', \sigma'\rangle = \mathcal{C}(\varsigma_{\text{CESK}})
\end{gathered}
$$

- To evaluate `spawn`, create a new thread to execute the given expression in the same environment.

$$
\begin{aligned}
\langle T[\mathit{tid}_1 \mapsto \overbrace{\langle \texttt{(spawn}\ e\texttt{)}, \rho, a, t\rangle}^{c}], \sigma\rangle \Rightarrow \langle T[\mathit{tid}_1 &\mapsto c_1, \mathit{tid}_2 \mapsto c_2], \sigma\rangle\\
\text{where } \mathit{tid}_2 &= \mathit{newtid}(c, T[\mathit{tid}_1 \mapsto c])\\
c_1 &= \langle \mathit{tid}_2, \rho, a, u\rangle\\
c_2 &= \langle e, \rho, a_{\mathbf{halt}}, t_0\rangle\\
u &= \mathit{tick}(\mathcal{S}(c, \sigma))
\end{aligned}
$$

- When a thread has finished its execution, its final value is written in the store at the address corresponding to its thread identifier (this is the reason why thread identifiers should also be considered as addresses).

$$\langle T[\mathit{tid} \mapsto \langle \mathit{val}, \rho, a_{\mathbf{halt}}, t\rangle], \sigma\rangle \Rightarrow \langle T, \sigma[\mathit{tid} \mapsto \mathit{val}]\rangle$$

- A `join` can only be evaluated when the thread we join on has finished (i.e. a value is stored at the address corresponding to its thread identifier). When this is the case, this value is returned.

$$
\begin{aligned}
\langle T[\mathit{tid} \mapsto \langle \texttt{(join}\ \text{\ae}\texttt{)}, \rho, a, t\rangle], \sigma\rangle &\Rightarrow \langle T[\mathit{tid} \mapsto \langle \mathit{val}, \rho, a, u\rangle], \sigma\rangle\\
&\text{if } \sigma(\mathcal{E}(\text{\ae}, \rho, \sigma)) = \mathit{val}\\
&\qquad \text{where } u = \mathit{tick}(\mathcal{S}(c, \sigma))
\end{aligned}
$$

  When this rule cannot be applied (that is, the thread we join on is still running), the context that does the join will block until this rule can be applied.

Note that `join` expects its argument to be atomic. If it was not the case, a continuation should be added to the CESK machine in order to apply the `join` after having evaluated its argument. It is possible to do this, but it would increase the coupling between the underlying CESK machine and the PCESK machine without adding any expressivity, as we can always replace `(join (spawn` e`))` by `(let ((t (spawn` e`))) (join t))`.

4.2.5 Injection Function

The injection function $\mathcal{I} : \mathit{Exp} \to \Sigma$ creates the initial machine state, containing only one thread that will evaluate the given expression. It uses the injection function of the CESK machine, $\mathcal{I}_{\text{CESK}} : \mathit{Exp} \to \Sigma_{\text{CESK}}$:

$$\begin{aligned}\mathcal{I}(e) &= \langle [\mathit{tid}_0 \mapsto c], \sigma\rangle \\ \text{where } \langle c, \sigma\rangle &= \mathcal{C}(\mathcal{I}_{\text{CESK}}(e))\end{aligned}$$

4.2.6 Evaluation Function

The evaluation function $\mathit{eval} : \mathit{Exp} \to \mathcal{P}(\Sigma)$ computes the set of reachable states and can be defined similarly as it was done for the CESK machine:

$$\mathit{eval}(e) = \{\varsigma \mid \mathcal{I}(e) \Rightarrow^* \varsigma\}$$

Example 10 (Concrete PCESK Evaluation)**.** We show how the concrete interpreter will evaluate the following program:

```
(letrec ((t (spawn (+ 1 2))))
  (join t))
```

First, the program (that we denote by e) is injected into an initial state:

$$\begin{aligned}\mathcal{I}(e) = \langle &[\mathit{tid}_0 \mapsto \langle e, \varnothing, a_{\mathbf{halt}}, 0_0\rangle], \\ &\{a_{\mathbf{halt}} \mapsto \mathbf{halt}\}\rangle\end{aligned}$$

Note that timestamps are subscripted with a number that identifies them to their corresponding thread. This is important to avoid conflicts between addresses if two different threads with the same timestamp need to store a value in the store. The first transition rule then applies (do a parallel step from a sequential one, i.e. evaluate the `letrec` with the underlying CESK machine), leading to the state:

$$\begin{aligned}\langle &[\mathit{tid}_0 \mapsto \langle \texttt{(spawn (+ 1 2))}, \varnothing, a_0, 1_0\rangle], \\ &\{a_{\mathbf{halt}}, a_{0_0} \mapsto \mathbf{letrec}(\ldots)\}\rangle\end{aligned}$$

Then, the `spawn` rule is applied. A new thread is created and its thread identifier is returned to the initial thread:

$$\begin{aligned}\langle[&\mathit{tid}_0 \mapsto \langle \mathit{tid}_1, \varnothing, a_0, 2_0\rangle, \\ &\mathit{tid}_1 \mapsto \langle \texttt{(+ 1 2)}, \varnothing, a_{\mathbf{halt}}, 0_1\rangle], \\ &\{a_{\mathbf{halt}}, a_{0_0}\}\rangle\end{aligned}$$

Then, either one of the threads can do a step. Suppose the first thread steps first:

$$\begin{aligned}\langle[&\mathit{tid}_0 \mapsto \langle \texttt{(join } t\texttt{)}, \{t \mapsto \mathit{tid}_1\}, a_{\mathbf{halt}}, 3_0\rangle, \\ &\mathit{tid}_1 \mapsto \langle \texttt{(+ 1 2)}, \varnothing, a_{\mathbf{halt}}, 0_1\rangle], \\ &\{a_{\mathbf{halt}}, a_{0_0}\}\rangle\end{aligned}$$

At this point, the first thread is blocked because it joins on the second thread which is still executing. The only thread that can step is thus the second one. We will not go over the details of the evaluation of `(+ 1 2)`. Just before the second thread finishes its execution, the state is:

$$\begin{aligned}\langle[&\mathit{tid}_0 \mapsto \langle\texttt{(join }t\texttt{)}, \{t \mapsto \mathit{tid}_1\}, a_{\mathbf{halt}}, 3_0\rangle,\\ &\mathit{tid}_1 \mapsto \langle 3, \varnothing, a_{\mathbf{halt}}, 5_1\rangle],\\ &\{a_{\mathbf{halt}}, a_{0_0}, \ldots\}\rangle\end{aligned}$$

We can now apply the transition rule that will store the result of this thread into the store.

$$\begin{aligned}\langle[&\mathit{tid}_0 \mapsto \langle\texttt{(join }t\texttt{)}, \{t \mapsto \mathit{tid}_1\}, a_{\mathbf{halt}}, 3_0\rangle,\\ &\mathit{tid}_1 \mapsto \langle 3, \varnothing, a_{\mathbf{halt}}, 5_1\rangle],\\ &\{a_{\mathbf{halt}}, a_{0_0}, \mathit{tid}_1 \mapsto 3, \ldots\}\rangle\end{aligned}$$

Finally, the first thread can step by applying the transition rule for `join`.

$$\begin{aligned}\langle[&\mathit{tid}_0 \mapsto \langle 3, \{t \mapsto \mathit{tid}_1\}, a_{\mathbf{halt}}, 3_0\rangle,\\ &\mathit{tid}_1 \mapsto \langle 3, \varnothing, a_{\mathbf{halt}}, 5_1\rangle],\\ &\{a_{\mathbf{halt}}, a_{0_0}, \mathit{tid}_1 \mapsto 3, \ldots\}\rangle\end{aligned}$$

All those states will be contained in $\mathit{eval}(e)$, but there will also be other states. When the second thread was created, instead of stepping the first one we could have stepped the second one, leading to different states.

4.3 Abstract Semantics: The Abstract PCESK Machine

As we want to have a finite state space in order to be able to compute the set of reachable states in finite time, we need to abstract the concrete PCESK machine defined in the previous section. Three components lead to an infinite state space: the infinite sets of addresses, timestamps and thread identifiers. We require that those three sets are finite. This change is reflected into other components of the PCESK machine. In this section, we adapt the state space to become finite, and then adapt the definition of the different functions used: the atomic evaluation function, the conversion functions, the transition function, the injection function and finally the evaluation function.

4.3.1 Abstract State Space

The concrete state space can be abstracted by replacing every component that is infinite by a finite approximation. For this machine, not only addresses and timestamps have to be abstracted, but also thread identifiers and primitive values. The resulting abstracted state space is given in Figure 4.3.

As for the CESK machine of Section 3.1, since there are now only a finite number of abstract addresses, the store becomes a mapping from abstract addresses to sets of abstract values. The same property applies to the thread map (*Threads*). Since there are only a finite number of abstract thread identifier, the thread map is now a mapping from abstract thread identifiers to a sets of abstract contexts. This will be a source of imprecision that will require a mechanism similar to abstract counting to solve the problem (see Section 4.4).

The lattice used for the abstraction $\widehat{\mathit{Num}}$ depends on the application. We can for example use $\widehat{\mathit{Num}} = \mathbb{Z} + \{\top, \bot\}$ where, as soon as two different elements of $\mathbb{Z}$ are joined together, the result is $\top$.

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \widehat{Threads} \times \widehat{Store} \\
\hat{T} \in \widehat{Threads} &= \widehat{TID} \rightharpoonup \mathcal{P}(\widehat{Context}) \\
\hat{c} \in \widehat{Context} &= \widehat{Control} \times \widehat{Env} \times \widehat{Addr} \times \widehat{Time} \\
\widehat{Control} &= Exp + \widehat{Val} \\
\hat{\rho} \in \widehat{Env} &= \widehat{Var} \rightharpoonup \widehat{Addr} \\
\hat{\kappa} \in \widehat{Kont} &::= \mathbf{rator}([e], \hat{\rho}, \hat{a}) \\
&\qquad \mid \mathbf{rand}(\widehat{val}, (e_0, \ldots, e_n), (\widehat{val}_0, \ldots \widehat{val}_n), \hat{\rho}, \hat{a}) \\
&\qquad \mid \mathbf{begin}((e_0, \ldots, e_n), \hat{\rho}, \hat{a}) \\
&\qquad \mid \mathbf{letrec}(\hat{a}_{binding}, ((v_0, e_0), \ldots, (v_n, e_n)), \\
&\qquad\qquad\qquad (e_{body,0}, \ldots, e_{body,n}), \hat{\rho}, \hat{a}) \\
&\qquad \mid \mathbf{if}(e_{cond}, e_{cons}, e_{alt}, \hat{\rho}, \hat{a}) \\
&\qquad \mid \mathbf{set!}(v, \hat{\rho}, \hat{a}) \\
&\qquad \mid \mathbf{halt} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Val}) \\
\widehat{val} \in \widehat{Val} &= \widehat{Clo} + Bool + \widehat{Num} + \widehat{Kont} + \widehat{TID} + \widehat{Addr} \\
\widehat{clo} \in \widehat{Clo} &= Lam \times \widehat{Env} \\
\hat{t} \in \widehat{Time} & \quad \text{a finite set of timestamps} \\
\hat{a} \in \widehat{Addr} & \quad \text{a finite set of addresses, includes } \widehat{TID} \\
\widehat{tid} \in \widehat{TID} & \quad \text{a finite set of thread identifiers}
\end{aligned}
$$

Figure 4.3: Abstract state space of the PCESK machine.

4.3.2 Atomic Evaluation Function

The abstract atomic evaluation function has to be adapted. It can now produce a set of values instead of a single value: $\widehat{\mathcal{E}} : AExp \times \widehat{Env} \times \widehat{Store} \to \mathcal{P}(\widehat{Val})$. Its definition is:

$$
\begin{aligned}
\widehat{\mathcal{E}}(n, \hat{\rho}, \hat{\sigma}) &= \{n\} \\
\widehat{\mathcal{E}}(b, \hat{\rho}, \hat{\sigma}) &= \{b\} \\
\widehat{\mathcal{E}}(v, \hat{\rho}, \hat{\sigma}) &= \sigma(\rho(v)) \\
\widehat{\mathcal{E}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\}
\end{aligned}
$$

4.3.3 Conversion Functions

The two conversion functions $\widehat{\mathcal{S}} : \widehat{Context} \times \widehat{Store} \to \hat{\Sigma}_{\text{CESK}}$ and $\widehat{\mathcal{C}} : \hat{\Sigma}_{\text{CESK}} \to \widehat{Context} \times \widehat{Store}$ are trivially adapted for the abstract state space:

$$
\begin{aligned}
\widehat{\mathcal{S}}(\langle e, \hat{\rho}, \hat{a}, \hat{t} \rangle, \hat{\sigma}) &= \langle e, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t} \rangle \\
\widehat{\mathcal{C}}(\langle e, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t} \rangle) &= \langle \langle e, \hat{\rho}, \hat{a}, \hat{t} \rangle, \hat{\sigma} \rangle
\end{aligned}
$$

4.3.4 Transition Function

The abstract transition function $(\widehat{\Rightarrow}) \subset \hat{\Sigma} \times \hat{\Sigma}$ will also be defined using the sequential transition function $(\widehat{\rightarrow}) \subset \hat{\Sigma}_{\text{CESK}} \times \hat{\Sigma}_{\text{CESK}}$.

The abstract sequential transition for `cas` is now non-deterministic, since we cannot always determine whether a variable is equal to a value or not. We can however know that a variable is not equal to a value if the meet between this variable's value and the value is the bottom element of our lattice. For example, with the lattice $\widehat{Num} = \mathbb{Z} + \{\top, \bot\}$, we have $3 \sqcap 5 = \bot$, so we know that 3 is never equal to 5. In the PCESK case, a lattice element is a set of value. Also, if the variable has only one possible concrete value which is equal to the only possible concrete value of $\ae_{old}$, we know that they are equal.

$$\begin{aligned}
\overbrace{\langle \texttt{(cas}\ v\ \ae_{old}\ \ae_{new}\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle}^{\hat{\varsigma}} \mathrel{\widehat{\rightarrow}}\ & \langle \widehat{\texttt{\#t}}, \hat{\rho}, \hat{\sigma} \cup [\hat{\rho}(v) \mapsto \widehat{\mathcal{E}}(\ae_{new}, \hat{\rho}, \hat{\sigma})], \hat{a}, \hat{u}\rangle \\
& \quad \text{unless } \hat{\sigma}(\hat{\rho}(v)) \sqcap \widehat{\mathcal{E}}(\ae_{old}, \hat{\rho}, \hat{\sigma}) = \bot \\
\mathrel{\widehat{\rightarrow}}\ & \langle \widehat{\texttt{\#f}}, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{u}\rangle \\
& \quad \text{unless } \hat{\sigma}(\hat{\rho}(v)) = \widehat{\mathcal{E}}(\ae_{old}, \hat{\rho}, \hat{\sigma}) = \{\widehat{val}\} \text{ and } \gamma(\widehat{val}) = \{val\}
\end{aligned}$$

where $\hat{u} = \widehat{tick}(\hat{\varsigma})$.

The function $\widehat{newtid} : \widehat{Context} \times \widehat{Threads} \mathrel{\widehat{\rightarrow}} \widehat{TID}$ has to be adapted to generate a finite number of thread identifiers.

The choice of the abstraction for $\widehat{newtid}$ will influence the precision of the analysis. One possible choice is to use the calling context as explained CESK's *alloc* in Section 3.1.3. However, as [MVH11] points out, this is not adapted to programs that create multiple threads from the same expressions (i.e. programs that uses a *thread-pool* pattern). When a bound on the number of threads created by the program is known, we can use an increasing bounded integer as thread identifier.

The abstract parallel transition function ($\widehat{\Rightarrow}$) can then be adapted to follow the changes made in the state space.

- When one abstract thread can do a sequential step, the PCESK machine also can. Note that the old context is kept in the thread map (and joined with the new one).

$$\begin{gathered}
\langle \hat{T}[\widehat{tid} \mapsto \{\hat{c}\} \cup \hat{C}], \hat{\sigma}\rangle \mathrel{\widehat{\Rightarrow}} \langle \hat{T} \sqcup [\widehat{tid} \mapsto \{\hat{c}'\}], \sigma'\rangle \\
\text{if } \widehat{\mathcal{S}}(\hat{c}, \hat{\sigma}) \mathrel{\widehat{\rightarrow}} \hat{\varsigma}_{\text{CESK}} \text{ and } \langle \hat{c}', \hat{\sigma}'\rangle = \widehat{\mathcal{C}}(\hat{\varsigma}_{\text{CESK}})
\end{gathered}$$

- For `spawn`, we also need to use a join instead of an update.

$$\begin{aligned}
\langle \hat{T}[\widehat{tid}_1 \mapsto \{\overbrace{\langle \texttt{(spawn}\ e\texttt{)}, \hat{\rho}, \hat{a}, \hat{t}\rangle}^{\hat{c}}\} \cup \hat{C}], \hat{\sigma}\rangle \mathrel{\widehat{\Rightarrow}}\ & \langle \hat{T} \sqcup [\widehat{tid}_1 \mapsto \{\hat{c}_1\}, \widehat{tid}_2 \mapsto \{\hat{c}_2\}], \hat{\sigma}\rangle \\
\text{where } \widehat{tid}_2 &= \widehat{newtid}(\hat{c}, T[\widehat{tid}_1 \mapsto \{\hat{c}\} \cup \hat{C}]) \\
\hat{c}_1 &= \langle \widehat{tid}_2, \hat{\rho}, \hat{a}, \hat{u}\rangle \\
\hat{c}_2 &= \langle e, \hat{\rho}, \hat{a}_{\mathbf{halt}}, \hat{t}_0\rangle \\
\hat{u} &= \widehat{tick}(\widehat{\mathcal{S}}(\hat{c}, \hat{\sigma}))
\end{aligned}$$

- When an abstract thread halts, its final value is saved in the store. Note that the old context remains in the thread map.

$$\begin{gathered}
\langle \hat{T}', \hat{\sigma}\rangle \mathrel{\widehat{\Rightarrow}} \langle \hat{T}', \hat{\sigma} \sqcup [\widehat{tid} \mapsto \{\widehat{val}\}]\rangle \\
\text{where } \hat{T}' = \hat{T} \sqcup [\widehat{tid} \mapsto \{\langle \widehat{val}, \hat{\rho}, \hat{a}_{\mathbf{halt}}, \hat{\sigma}\rangle\}]
\end{gathered}$$

- For `join`, the old context is also kept in the thread map.

$$\begin{aligned}
\langle \hat{T} \sqcup [\widehat{tid} \mapsto \{\overbrace{\langle \texttt{(join}\ \ae\texttt{)}, \hat{\rho}, \hat{a}, \hat{t}\rangle}^{\hat{c}}\}], \hat{\sigma}\rangle \mathrel{\widehat{\Rightarrow}}\ & \langle \hat{T} \sqcup [\widehat{tid} \mapsto \{\langle \widehat{val}, \hat{\rho}, \hat{a}, \hat{u}\rangle, \hat{c}\}], \hat{\sigma}\rangle \\
\text{if } \hat{\sigma}(\hat{a}_v) &= \widehat{val} \\
\text{where } \hat{a}_v &\in \widehat{\mathcal{E}}(\ae, \hat{\rho}, \hat{\sigma}) \\
\hat{u} &= \widehat{tick}(\widehat{\mathcal{S}}(\hat{c}, \hat{\sigma}))
\end{aligned}$$

4.3.5 Injection Function

The abstract injection function for the PCESK machine, $\hat{\mathcal{I}} : \mathit{Exp} \to \hat{\Sigma}$, makes use of the abstract injection function for the CESK machine, $\hat{\mathcal{I}}_{\text{CESK}} : \mathit{Exp} \to \hat{\Sigma}_{\text{CESK}}$:

$$\begin{aligned} \widehat{\mathcal{I}}(e) &= \langle [\widehat{tid}_0 \mapsto \{\hat{c}\}], \hat{\sigma} \rangle \\ \text{where } \langle \hat{c}, \hat{\sigma} \rangle &= \widehat{\mathcal{C}}(\widehat{\mathcal{I}}_{\text{CESK}}(e)) \end{aligned}$$

4.3.6 Evaluation Function

Finally, the abstract evaluation function $\widehat{eval} : \mathit{Exp} \to \mathcal{P}(\hat{\Sigma})$ can be defined:

$$\widehat{eval}(e) = \{\hat{\varsigma} \mid \widehat{\mathcal{I}}(e) \mathrel{\widehat{\Rightarrow}}^* \hat{\varsigma}\}$$

Example 11 (Loss of precision due to the finiteness of *TID*)**.** Suppose we use the abstract interpreter to evaluate the following program:

```
(letrec ((t1 (spawn 1))
         (t2 (spawn 2)))
  (join t1))
```

If the two new threads are given the same thread identifier, there will be a loss of precision. The variables `t1` and `t2` will point to the same thread identifier and it would be infeasible to distinguish the two threads. The `join` on `t1` will produce two possible results: 1 (as expected), and 2. $\widehat{eval}(e)$ will contain states that are unreachable by the concrete interpreter.

4.4 Refinements of the PCESK Machine

The PCESK machine described until here contains everything to perform an analysis, but due to the many sources of nondeterminism, especially the exponential number of possible thread interleavings, doing such an analysis is impractical on other than very small programs. This section presents multiple techniques we use to either decrease the complexity of the analysis or improve its precision.

4.4.1 Abstract Counting

Abstract counting has already been described for the CESK machine. There is nothing special to change to use this technique on the PCESK machine, since it only concerns the way the store does the joins and does not depend on other components of the machine.

4.4.2 Abstract Thread Counting

A major source of imprecision for an analysis in the abstract machine is the fact that the transition function has to join the new context with the old one when sequentially stepping a context, instead of replacing it. This results in having two different contexts with the same thread identifier. This is because a thread identifier might be associated with multiple threads, so a join is needed to avoid discarding possible interleavings. However, during the execution of the abstract machine, many joins could be replaced by updates, since it will generally be the case that only one thread is associated with a thread identifier (especially for programs with a bounded number of threads). This can be seen as an adaptation of abstract counting for the thread map instead of the store.

Those updates can be performed only when it is the case that only one thread is associated with the identifier being updated, so we need to keep track of the number of contexts associated with each thread identifier. This is done by introducing a component that counts the number of contexts per thread identifier, $\widehat{TCount}$. Note that we are only interested to know if there is no context (0), one context (1), or more than one context (∞) associated with a thread identifier.

$$\begin{aligned}\hat{\varsigma} \in \hat{\Sigma} &= \widehat{Threads} \times \widehat{Store} \times \widehat{TCount} \\ \hat{\mu} \in \widehat{TCount} &= \widehat{TID} \to \{0, 1, \infty\}\end{aligned}$$

The transition rules have to be adapted to exploit this information: when joining on a thread identifier $\widehat{tid}$, if $\hat{\mu}(\widehat{tid}) = 1$, we can do an update instead of a join. The transition rules become:

- in the first rule, replace $\hat{T} \sqcup [\widehat{tid} \mapsto \{\hat{c}'\}]$ by $\hat{T}[\widehat{tid} \mapsto \{\hat{c}\} \cup \hat{C}]$ if $\hat{\mu}(\widehat{tid}) = 1$ (else, keep the old rule),
- for `spawn`, we can do a strong update on $\widehat{tid}_1$ (if $\hat{\mu}(\widehat{tid}) = 1$), but not on $\widehat{tid}_2$, because we are creating a new context (thus, if a thread already exists with $\widehat{tid}_2$ as identifier, we must do a join),
- when an abstract thread halts, there is no joins so the rule does not change,
- for `join`, we can discard $\hat{c}$ in $\hat{T} \sqcup [\widehat{tid} \mapsto \{\langle \widehat{val}, \hat{\rho}, \hat{a}, \hat{u}\rangle, \hat{c}\}]$ which gives $\hat{T} \sqcup [\widehat{tid} \mapsto \{\langle \widehat{val}, \hat{\rho}, \hat{a}, \hat{u}\rangle\}]$.

From now on, $(\widehat{\Rightarrow})$ will refer to this updated transition function.

4.4.3 Removing Threads

The transition rule to handle a thread that halts keeps the thread in the thread map. If we are able to remove it, this will allows to free a thread identifier (if this is the only thread associated with this identifier), thus possibly improving precision if another thread is associated with this thread identifier later in the computation. This behavior requires a simple change in the transition rule. The new thread map becomes $\hat{T}$ instead of $\hat{T}'$:

$$\begin{gathered}\langle \hat{T}', \hat{\sigma}\rangle \mathrel{\widehat{\Rightarrow}} \langle \hat{T}, \hat{\sigma} \sqcup [\widehat{tid} \mapsto \{\widehat{val}\}]\rangle \\ \text{where } \hat{T}' = \hat{T} \sqcup [\widehat{tid} \mapsto \{\langle \widehat{val}, \hat{\rho}, \hat{a}_{\mathbf{halt}}, \hat{\sigma}\rangle\}]\end{gathered}$$

According to [MVH11], doing this change can be done safely while preserving soundness.

4.4.4 Abstract Garbage Collection

Abstract garbage collection has also already been described, but it cannot be used as-is with the PCESK machine. Abstract garbage collection starts collecting live location at the current expression of the CESK machine (the *control* component). In the PCESK machine however, there is not *one* current expression, but *one per context*. Abstract garbage collection has to be adapted to deal with those multiple expressions. The garbage collection reclaims addresses that are reachable by none of the PCESK contexts by joining the set of reachable addresses of each context and doing the store restriction on this set. The definitions of $\mathcal{LL}_\sigma$, $(\rightarrow)_{GC}$ and $\mathcal{R}$ given in Section 3.1.4 can stay the same, but the transition $(\rightarrow')$ becomes the parallel transition $(\Rightarrow') : \Sigma \times \Sigma$ and is defined as:

$$\begin{gathered}\langle T, \sigma\rangle \Rightarrow' \langle T, \sigma|\mathcal{L}\rangle \\ \text{where } \mathcal{L} = \bigcup_{c \in range(T)} \mathcal{R}(S(c, \sigma))\end{gathered}$$

4.4.5 State Subsumption

To adapt the state subsumption mechanism to the PCESK machine, we have to define when an abstract PCESK state subsumes another. An abstract state $\hat{\varsigma}_1$ (with store component $\hat{\sigma}_1$) is subsumed by $\hat{\varsigma}_2$ (with store component $\hat{\sigma}_2$) if:

- every context in the thread map $\hat{T}_1$ of $\hat{\varsigma}_1$ is subsumed by a context with the same thread identifier in the thread map $\hat{T}_2$ of $\hat{\varsigma}_2$, that is, $\forall \widehat{tid} \in dom(\hat{T}_1), \forall \hat{c}_1 \in \hat{T}_1(\widehat{tid}), \exists \hat{c}_2 \in \hat{T}_2(\widehat{tid})$ s.t. $\mathcal{S}(\hat{c}_1, \hat{\sigma}_1) \sqsubseteq \mathcal{S}(\hat{c}_2, \hat{\sigma}_2)$,
- $\hat{\sigma}_1$ is subsumed by $\hat{\sigma}_2$, i.e. $\hat{\sigma}_1 \sqsubseteq \hat{\sigma}_2$.

4.5 Output of the PCESK Machine

In this section we will summarize what the PCESK machine computes (the *how* has been described in details in the previous sections) as it will become useful in Chapter 5. If we want to analyze programs using the PCESK machine, it is not necessary to know the internals of this machine but rather what is its output.

To use the PCESK machine for analyzing a program, we first need to compute the set of reachable states of this program using the abstract evaluation function $\widehat{eval}$, which gives us a finite set of PCESK states (Σ) in output. As a reminder, each of those states contains a map of contexts associated with a thread identifier (each context has a control component, an environment, a continuation address and a timestamp), and a shared store. As explained before, this set of states is an over-approximation of all the states that are reachable by the program.

As described in Section 3.1.5, it is possible to adapt the evaluation function to compute a state graph instead of a state set. The initial state $\widehat{\mathcal{I}}(e)$ is the entry point of this graph and the successors of a state are found by applying one step of the transition function. The set of reachable states of the program is then equal to the set of vertices of this graph. The main difference with the graph produced by the CESK machine is that, in the case of the PCESK machine, each transition is labeled with the thread it operated on.

An example of such a state graph is given in Figure 4.4 and corresponds to the program below. We can see that as soon as a thread is spawned, there is more than one possible transition, and all the possible interleavings are taken into accounts. Once the thread `t` finished its execution, it is removed, and the transition of the `join` of the main thread can be performed.

```
(letrec ((t (spawn (+ 1 2))))
  (+ (join t) 3))
```

To analyze a program, one will generally compute this state graph (though the set of reachable states is sufficient to perform some analyses). With this state graph, it is possible to deduce properties of the program by analyzing components of some or every state of the graph. For example, we can see if an expression ever gets evaluated by looking if there exists a control component of a state that is equal to this expression. This is exactly what is done in Example 7 to find whether the expression `error` is evaluated in a program or not.

4.6 Conclusion

In this chapter, we described the language for which our analyses will be defined, and we have given its semantics based on a PCESK machine. The semantics have been abstracted to be able to compute the set of reachable states of a program in finite time. This will allow us to define decidable analyses in the next chapter.

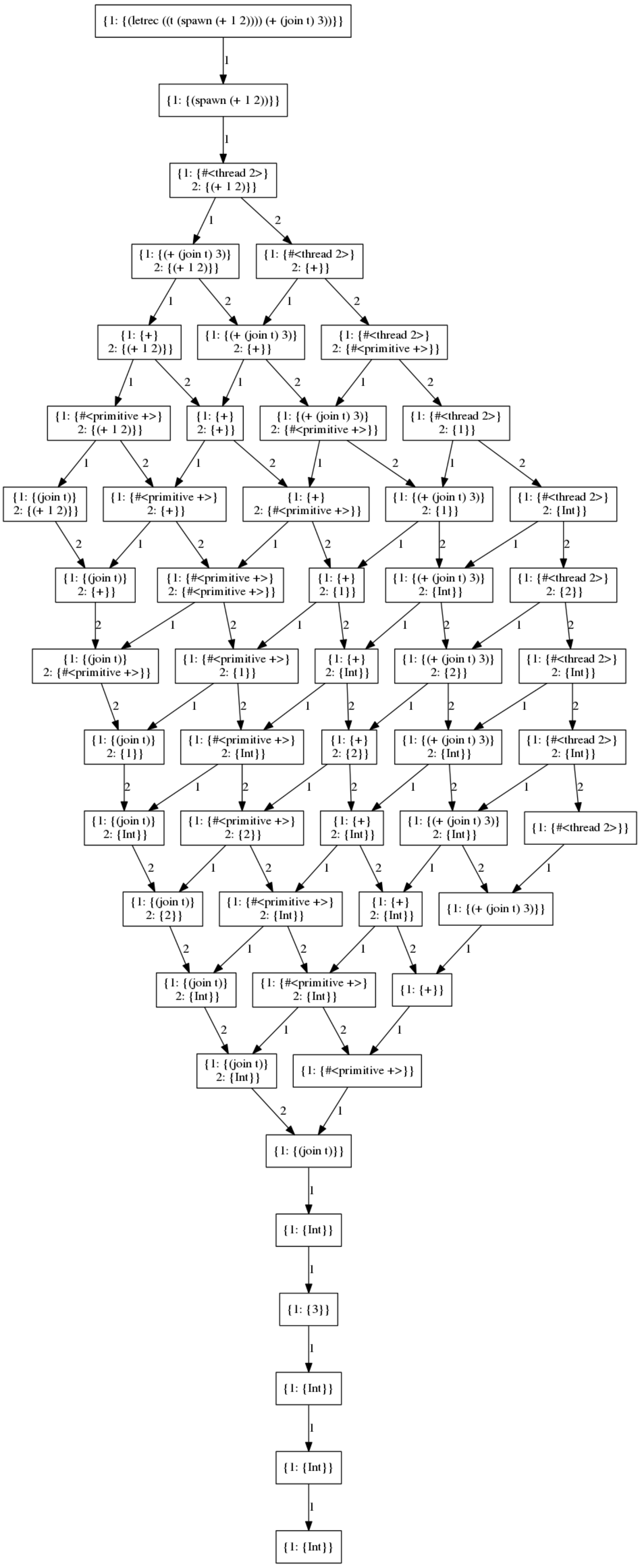
{1: {(letrec ((t (spawn (+ 1 2)))) (+ (join t) 3))}}
{1: {(spawn (+ 1 2))}}
{1: {#<thread 2>} 2: {(+ 1 2)}}
{1: {(+ (join t) 3)} 2: {(+ 1 2)}}
{1: {#<thread 2>} 2: {+}}
{1: {+} 2: {(+ 1 2)}}
{1: {(+ (join t) 3)} 2: {+}}
{1: {#<thread 2>} 2: {#<primitive +>}}
{1: {#<primitive +>} 2: {(+ 1 2)}}
{1: {+} 2: {+}}
{1: {(+ (join t) 3)} 2: {#<primitive +>}}
{1: {#<thread 2>} 2: {1}}
{1: {(join t)} 2: {(+ 1 2)}}
{1: {#<primitive +>} 2: {+}}
{1: {+} 2: {#<primitive +>}}
{1: {(+ (join t) 3)} 2: {1}}
{1: {#<thread 2>} 2: {Int}}
{1: {(join t)} 2: {+}}
{1: {#<primitive +>} 2: {#<primitive +>}}
{1: {+} 2: {1}}
{1: {(+ (join t) 3)} 2: {Int}}
{1: {#<thread 2>} 2: {2}}
{1: {(join t)} 2: {#<primitive +>}}
{1: {#<primitive +>} 2: {1}}
{1: {+} 2: {Int}}
{1: {(+ (join t) 3)} 2: {2}}
{1: {#<thread 2>} 2: {Int}}
{1: {(join t)} 2: {1}}
{1: {#<primitive +>} 2: {Int}}
{1: {+} 2: {2}}
{1: {(+ (join t) 3)} 2: {Int}}
{1: {#<thread 2>} 2: {Int}}
{1: {(join t)} 2: {Int}}
{1: {#<primitive +>} 2: {2}}
{1: {+} 2: {Int}}
{1: {(+ (join t) 3)} 2: {Int}}
{1: {#<thread 2>}}
{1: {(join t)} 2: {2}}
{1: {#<primitive +>} 2: {Int}}
{1: {+} 2: {Int}}
{1: {(+ (join t) 3)}}
{1: {(join t)} 2: {Int}}
{1: {#<primitive +>} 2: {Int}}
{1: {+}}
{1: {(join t)} 2: {Int}}
{1: {#<primitive +>}}
{1: {(join t)}}
{1: {Int}}
{1: {3}}
{1: {Int}}
{1: {Int}}
{1: {Int}}

Figure 4.4: State graph of a CScheme program.

Due to the potentially large size of this set of reachable states, multiple refinements are needed to obtain a set that is practically computable. We described many refinements that influence the size of the state space. Abstract counting can be taken as-is from the CESK machine, and can also be used on the thread map resulting in abstract thread counting. These refinements are existing refinements described by Might and Van Horn [MVH11]. We also introduced two new refinements to the PCESK machine: abstract garbage collection and state subsumption. Abstract garbage collection has already proven useful for the CESK machine, but required adaptation to be used in the PCESK machine. Indeed, the PCESK machine has multiple control components, and we have to be careful not to reclaim addresses that are not reachable in one thread but are still reachable in another. State subsumption is another new refinement that consists of avoiding exploration of parts of the state graph for which an over-approximation has already been explored.

Finally, this chapter summarized how the PCESK machine can be used to perform static analyses. The next chapter will use this reasoning to build multiple useful analyses.

Chapter 5

Applications of the PCESK Machine

In this chapter, we use the PCESK machine from the previous chapter to build analyses that can detect non-trivial concurrency issues. We start from Might and Van Horn's *may-happen-in-parallel* analysis [MVH11] and extend it into a *conflict analysis*. The may-happen-in-parallel analysis and the conflict analysis only look at individual states in the state graph built by the PCESK machine. We also describe an analysis that looks into portions of the state graph to detect uses of `cas` where a failed `cas` is not retried. We combine this analysis with the conflict analysis to get a more general *race condition analysis*. Finally, we present a *deadlock analysis* to detect deadlocks in programs using locks implemented with `cas`. The analyses built in this chapter will be tested and validated in the next chapter by applying them to various example programs.

5.1 May-Happen-in-Parallel Analysis

With the abstract PCESK machine, it is possible to deduce whether two expressions *may* or *will not* happen in parallel with what is called a *may-happen-in-parallel (MHP) analysis* [MVH11]. Since every state of the state graph computed by the abstract interpreter contains information about which expression each thread is currently evaluating, it is sufficient to look for an element of the state space that contains the two expressions as the context of two different threads.

We can define this as a relation $\mathit{MHP} \subset \mathit{Exp} \times \mathit{Exp}$. In the program e, the two expressions e_1 and e_2 may happen in parallel if $\mathit{MHP}(e_1, e_2)$. Might and Van Horn define this MHP relation as:

$$\mathit{MHP}(e_1, e_2) \Leftrightarrow$$
$$\exists \langle \hat{T}, \hat{\sigma} \rangle \in \widehat{\mathit{eval}}(e)\ \exists \widehat{\mathit{tid}}_1, \widehat{\mathit{tid}}_2 \in \mathit{dom}(\hat{T}),\ \langle e_1, \_, \_, \_ \rangle \in \hat{T}(\widehat{\mathit{tid}}_1) \wedge\ \langle e_2, \_, \_, \_ \rangle \in \hat{T}(\widehat{\mathit{tid}}_2).$$

However, this definition has a problem when $e_1 = e_2$. If the expression e is evaluated at some point by any thread in the program, $\mathit{MHP}(e, e)$ will be true, even if the program only contains one thread. Indeed, any context $\hat{c}$ evaluating e will satisfy $\exists \widehat{\mathit{tid}}_1, \widehat{\mathit{tid}}_2 \in \mathit{dom}(\hat{T}), \hat{c} \in T(\widehat{\mathit{tid}}_1) \wedge \hat{c} \in T(\widehat{\mathit{tid}}_2)$. Requiring that $\widehat{\mathit{tid}}_1 \neq \widehat{\mathit{tid}}_2$ is too restrictive, since we may have two different contexts that are associated with the same abstract thread identifier where both threads evaluate e.

We adapt this definition to ensure that if the two expressions are evaluated in the same thread, they are in distinct contexts. The definition now becomes:

$$MHP(e_1, e_2) \Leftrightarrow$$

$$\exists \langle \hat{T}, \hat{\sigma} \rangle \in \widehat{eval}(e)\ \exists \widehat{tid}_1, \widehat{tid}_2 \in dom(\hat{T}),\ \overbrace{\langle e_1, \_, \_, \_ \rangle}^{\hat{c}_1} \in \hat{T}(\widehat{tid}_1) \land \overbrace{\langle e_2, \_, \_, \_ \rangle}^{\hat{c}_2} \in \hat{T}(\widehat{tid}_2)$$
$$\land\ \hat{c}_1 \neq \hat{c}_2$$

It is important to remember that if this relation holds for two expressions, it does not necessarily mean that they *will* be evaluated in parallel. This will depend on the actual path taken during the execution of the program. Also, some states in the state graph might not be reachable as this state graph is an over-approximation of the program execution. However, when this relation does *not* hold for two expressions, those two expressions will *not* be evaluated in parallel (if the abstraction is sound). Indeed, if $MHP(e_1, e_2)$ does not hold, there is no state where e_1 and e_2 are evaluated in parallel, and thus there is no possible execution of the program leading to such a state.

This analysis alone does not allow us to automatically detect concurrency bugs in a program, as it only provides information about expressions whose evaluation may happen in parallel. By taking into account the way the store is used, we can extend it to obtain a more meaningful analysis, which is the subject of the next section.

5.2 Read/Write and Write/Write Conflicts Detection

By using the MHP analysis, we can detect conflicts between accesses to the store. A read is made on the store when a variable is evaluated, and a write is done by using either `set!` or `cas`. A program contains a read/write conflict if a read operation may happen in parallel with a write operation on the same store address. Similarly, a program contains write/write conflict if two write operations on the same address may happen in parallel.

Note that CScheme does not allow *pointer aliasing* (having more than one way to access a store location), although this is generally possible in other languages. For example, pointer aliasing is possible with *pointers* in C, with *references* in C++, by default with object arguments in Java, or in Scheme by using cons-cells and `set-car!` or `set-cdr!`.

In CScheme however, it is not possible to have more than one variable that points to the same address in the concrete state space. We could take advantage of this property to detect conflicts by checking whether two accesses (read/write or write/write) to the same *variable* may happen in parallel. Since no pointer aliasing is possible, such an analysis would be sound.

However, if we later want to extend CScheme to allow some form of pointer aliasing (e.g. by supporting mutable cons-cells), the analysis will become unsound and its usefulness will decrease.

We choose another solution to deal with aliasing while keeping soundness. Conflicts can be detected by checking whether two accesses to the same *address* may happen in parallel. This might result in a loss of precision since in the abstract state space, multiple variables may point to the same address because of the finiteness of the store. In such cases, a detected conflict that in reality will never happen is called a *false positive.*

> **Example 12** (Parallel counter with a `set!` conflict)**.** The following program implements a parallel counter similar to Example 8. However, instead of using the atomic operation `cas`, it directly uses a `set!`, introducing a race condition.

```
(letrec ((counter 0)
         (inc (lambda ()
                (set! counter (+ counter 1))))
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  counter)
```

Suppose thread `t1` first evaluate `(+ counter 1)`, which gives 1, then thread `t2` evaluates the whole `(set! counter (+ counter 1))`, storing 1 in `counter`. Thread `t1` will then do the `set!`, storing 1 in `counter`. The final value of `counter` will thus be 1, even though `inc` has been called twice. If those operations are ordered differently, `counter` might take the correct value of 2.

We want to be able to detect that there are two conflicts in this example: a read/write conflict between `counter` in `(+ counter 1)` and the `set!`, and a write/write conflict between the `set!` and itself, since it can be evaluated in two different threads at the same time.

Looking for concurrent read and writes successfully detects potentially harmful conflicts, but it will also detect some harmless conflicts that result from the use of `cas`. The general pattern that arise when we use `cas` to update some variable `x` is the following:

```
(letrec ((update-x (lambda ()
                     ;; Instead of (set! x (f x))
                     (let ((old x)
                           (new (f old)))
                       (if (cas x old new)
                         #t
                         (update-x))))))
  ...)
```

When `update-x` is called in more than one thread, it is possible that the `cas` is evaluated in parallel with either itself, or with the read on the variable `x`. While it is indeed a conflict, it is not a harmful one. In fact, `cas` can be seen as an atomic `set!` that does not perform the write operation when there is a conflict and notifies the programmer about this. It is then the programmer's responsibility to correctly handle this conflict by trying to update the variable later. As most correct uses of `cas` will lead to such harmless conflicts, it is useful to have an analysis that filters out this kind of conflict. As long as they are correctly handled by retrying the `cas` later in the execution in case of failure (we will see how to check if this is the case in Section 5.3), those conflicts are indeed harmless.

However, we have to be careful when filtering out conflicts involving only `cas` as a write operation, as shown in Example 13.

Example 13 (Parallel counter with a `cas` conflict)**.** Consider the following program, which is Example 8, extended with support for a decrease operation, but it contains a race condition due to an incorrect implementation of the function `dec`.

```
(letrec ((counter 0)
         (inc (lambda ()
                (letrec ((old counter)
                         (new (+ old 1)))
                  (if (cas counter old new)
                    #t
                    (inc)))))
         (dec (lambda ()
                (letrec ((old counter)
                         (new (- counter 1)))
                  (if (cas counter old new)
                    #t
                    (dec)))))
         (t1 (spawn (inc)))
         (t2 (spawn (dec)))
         (t3 (spawn (dec))))
  (join t1)
  (join t2)
  counter)
```

The computation of `new` is done by referring to `counter` instead of `old` (underlined in the code). While we expect this program to return −1 (an `inc` and two `dec`s), some interleavings might produce different results, such as 0 with the following interleaving:

- thread `t2` stores the value of `counter` (initially 0) in `old`,
- thread `t1` increases the value of `counter`, `counter` is now equal to 1,
- thread `t2` computes the value of `new` as `(- counter 1)`, which evaluates to 0,
- thread `t3` decreases the value of `counter`, `counter` is now equal to 0,
- thread `t2` performs its `cas`: `counter` is indeed equal to `old` (i.e. 0), so `counter` is set to the value of `new` (i.e. 0).

This program thus contains a harmful conflict involving only read operations and write operations through `cas`. This conflict should be detected by the analysis. We will describe how to filter out harmless conflicts but still detect conflicts such as this one in Section 5.2.2.

5.2.1 Extracting Reads and Writes

To formalize the conflict analysis, we first describe two relations to extract information about the addresses read from and written to by a context. This will allow us to have a more generic analysis that can be extended by only modifying the definition of those relations. This is necessary when we want to detect conflicts only involving writes from `set!` expressions, or conflicts also involving writes from `cas` expressions. Those relations are $\mathit{Read} \subset \widehat{\mathit{Context}} \times \widehat{\mathit{Addr}}$ and $\mathit{Write} \subset \widehat{\mathit{Context}} \times \widehat{\mathit{Addr}}$. If the context $\hat{c}$ contains a read (resp. write) operation on address $\hat{a}$, $\mathit{Read}(\hat{c}, \hat{a})$ (resp. $\mathit{Write}(\hat{c}, \hat{a})$) will hold. They are defined as follows.

$$\mathit{Read}(\langle v, \hat{\rho}, \_, \_\rangle, \hat{\rho}(v))$$
$$\mathit{Write}(\langle \texttt{(set!}\ v\ \_\texttt{)}, \hat{\rho}, \_, \_\rangle, \hat{\rho}(v))$$
$$\mathit{Write}(\langle \texttt{(cas}\ v\ \_\ \_\texttt{)}, \hat{\rho}, \_, \_\rangle, \hat{\rho}(v))$$

For the *Write* relation, we can drop the part of the definition related to `cas` if we are only interested in conflicts involving `set!`. Such a definition also allows to handle new read and write operations as they are added to the language, without modifying the analysis.

5.2.2 Conflict Analysis

We define three relations: one to detect read/write conflicts ($\mathit{RWConflict} \subset \mathit{Exp}$), one to detect write/write conflicts ($\mathit{WWConflict}(e) \subset \mathit{Exp}$), and one that combines those two to detect any conflict ($\mathit{Conflict} \subset \mathit{Exp}$). A read/write conflict is detected when both a read and a write on the same address may happen in parallel.

$$\begin{aligned}\mathit{RWConflict}(e) \Leftrightarrow\ & \exists\langle\hat{T},\hat{\sigma}\rangle \in \widehat{\mathit{eval}}(e), \exists\widehat{\mathit{tid}}_1, \widehat{\mathit{tid}}_2 \in \mathit{dom}(\hat{T}),\\ & \hat{c}_1 \in \hat{T}(\widehat{\mathit{tid}}_1) \wedge \mathit{Read}(\hat{c}_1, \hat{a}) \wedge\\ & \hat{c}_2 \in \hat{T}(\widehat{\mathit{tid}}_2) \wedge \mathit{Write}(\hat{c}_2, \hat{a})\end{aligned}$$

Write/write conflicts are handled similarly, except that we want to ensure that we are not detecting a non-existent conflict due to the fact that two write operations are executed by the same thread. This is the same problem we had with the *MHP* analysis, and it does not occur in the *RWConflict* analysis as reads and writes arise from different expressions. We have to be sure that there is indeed more than one concrete thread performing a write on address $\hat{a}$.

$$\begin{aligned}\mathit{WWConflict}(e) \Leftrightarrow\ & \exists\langle\hat{T},\hat{\sigma}\rangle \in \widehat{\mathit{eval}}(e), \exists\widehat{\mathit{tid}}_1, \widehat{\mathit{tid}}_2 \in \mathit{dom}(\hat{T}),\\ & \hat{c}_1 \in \hat{T}(\widehat{\mathit{tid}}_1) \wedge \mathit{Write}(\hat{c}_1, \hat{a}) \wedge\\ & \hat{c}_2 \in \hat{T}(\widehat{\mathit{tid}}_2) \wedge \mathit{Write}(\hat{c}_2, \hat{a}) \wedge\\ & \hat{c}_1 \neq \hat{c}_2\end{aligned}$$

Finally, we combine the two kinds of conflicts to get our conflict analysis.

$$\mathit{Conflict}(e) \Leftrightarrow \mathit{RWConflict}(e) \vee \mathit{WWConflict}(e)$$

From this formalization, it is possible to implement an analysis that also extracts addresses and expressions involved in a conflict. From this extracted information, we can then filter out some harmless conflicts (or at least, some conflicts that are considered harmless assuming that `cas` is correctly used). As explained before, the typical use of `cas` is that there is a read from a variable to get its current value, the new value is computed from this value, and the `cas` is tried. If the `cas` fails, we try again the whole procedure, reading again the content of the variable.

This will lead to a read/write and a write/write conflict between two threads performing this pattern, but this pattern can safely be filtered out. However, as soon as there is more conflicts than just a read/write and a write/write conflict, the program might be incorrect, even if `cas` are correctly retried, as Example 13 pointed out. This implies that we are not able to filter out non-problematic conflicts for programs where, for each address and pair of threads, there is more than a read/write and write/write conflict, such as in the program of Example 14.

> **Example 14** (Parallel counter with false positives)**.** This example is similar to Example 13, with the correct implementation of `dec`. The first thread calls `inc` followed by `dec` (underlined), which will create two read/write and two write/write conflicts with the second thread which calls `inc`. This kind of conflicts is not harmful, but is detected by our analysis as we have no way of differentiating them from harmless conflicts.

```
(letrec ((counter 0)
         (inc (lambda ()
                (letrec ((old counter)
                         (new (+ old 1)))
                  (if (cas counter old new)
                    #t
                    (inc)))))
         (dec (lambda ()
                (letrec ((old counter)
                         (new (- old 1)))
                  (if (cas counter old new)
                    #t
                    (dec)))))
         (t1 (spawn (begin (inc) (dec))))
         (t2 (spawn (inc)))
  (join t1)
  (join t2)
  counter)
```

The pattern that we filter out is the following. For each address $\hat{a}$ involved in a conflict, and for each pair of threads involved in the conflicts on $\hat{a}$, we can ignore conflicts involving $\hat{a}$ and the two threads if the only conflicts are:

1. a read/write conflict between the evaluation of a variable living at address $\hat{a}$ and a cas on the same address $\hat{a}$, and

2. a write/write conflict between two cas on the same address $\hat{a}$.

It seems that this can be extended to conflicts where either only condition 1 is met (i.e. there is only a read/write conflict), or both conditions are. Indeed, under the assumption that cas are correctly retried, if there is only a write/write conflict between two cas on the same address, one of the two cas will fail while the other will succeed, and the failing one will be retried later.

This way of filtering conflicts has to be done after detecting all possible conflicts, since we have to look at the number of conflicts involving the same address. This filtering introduces unsoundness in the analysis, as some potentially harmful existing conflicts will not be detected anymore. However, it seems that the undetected conflicts generally involve a cas that is not retried when it fails. The detection of this case is handled in the next section.

We conclude this section with Example 15 that discusses the importance of choosing appropriate addresses to avoid detecting some false positives with the conflict analysis.

Example 15 (False positive conflict)**.** Consider the following program.

```
(letrec ((x 1)
         (y 1)
         (f (lambda () (set! x (+ x 1))))
         (t1 (spawn (f)))
         (g (lambda () (set! y (+ y 1))))
         (t2 (spawn (g))))
  (join t1)
  (join t2)
  x)
```

Suppose that during the abstract evaluation of this program, variables `x` and `y` both point to the same store location. This will for example be the case if new addresses are generated from the current value of the timestamp and if timestamps are implemented as the k last call-sites. Since no call happens between the declaration of `x` and the declaration of `y`, they both will be stored at the same address.

The conflict detection will thus detect a write/write conflict between the body of `f` and the body of `g` because they both read and write from a variable that lives at the same address. This is a false positive since there is no possible conflict in this program.

To avoid this, we can rely on the fact that CScheme does not allow pointer aliasing to detect conflicts between accesses to variables instead of addresses, but as described previously this will not be transposable for other languages. A better solution is to improve the addresses to not only depend on the k last call-sites, but also on an identifier linked to the variable. This identifier is found by associating a unique *tag* for every node in the AST and using this tag as the identifier. Since `x` and `y` have a different tag, they will be stored at different addresses (the timestamp component of their address will be the same, but the tag component will be different).

5.3 Unretried `cas` Detection

When we use `cas` in a program, it is important to check its return value and to try it again in case it failed. A `cas` that is not retried may lead to race conditions, as shown by Example 16

Example 16 (Unretried `cas` leading to a race condition)**.** Consider the following program.

```
(letrec ((x 0)
         (t1 (spawn (cas x 0 1)))
         (t2 (spawn (cas x 0 2))))
  (join t1)
  (join t2)
  x)
```

Depending whether thread `t1` or thread `t2` gets executed first, the outcome of the program can either be 1 or 2.

5.3.1 Manipulating the State Graph

To detect whether a `cas` is retried or not when it fails, it is no longer sufficient to look at individual states in the state graph. Instead, we need to look into the structure of the state graph. Therefore, we need to manipulate this graph. We assume we have a function $\widehat{geval} : \mathit{Exp} \to \mathit{Vertices} \times \mathit{Edges}$, where $\mathit{Vertices} = \mathcal{P}(\hat{\Sigma})$ and $\mathit{Edges} = \mathcal{P}(\hat{\Sigma} \times \widehat{\mathit{TID}} \times \hat{\Sigma})$. This function is similar to $\widehat{eval}$, but gives us a state graph (defined by a set of vertices and a set of edges) instead of a state set. More details on how to implement it are given in Section 4.5. Note that edges are labeled by the thread identifier of the thread on which a transition rule is applied. The edge $(\hat{\varsigma}_1, \widehat{tid}, \hat{\varsigma}_2)$ means that we can reach state $\hat{\varsigma}_2$ by applying a transition rule on thread $\widehat{tid}$ from the state $\hat{\varsigma}_1$

We also need some relations that will make the formulas more clear.

- $\mathit{Successor} \subset \mathit{Edges} \times \hat{\Sigma} \times \hat{\Sigma}$ checks whether a state directly follows another state in the given set of edge.
- $\mathit{Path} \subset \mathit{Edges} \times \hat{\Sigma} \times \hat{\Sigma}$ checks whether a path exists between to states.

- $\mathit{TidExp} \subset \widehat{\mathit{Threads}} \times \widehat{\mathit{TID}} \times \mathit{Exp}$ checks whether a thread is currently evaluating a given expression.

- $\mathit{PathToTidExp} \subset \mathit{Edges} \times \hat{\Sigma} \times \widehat{\mathit{TID}} \times \mathit{Exp}$ checks whether there exists a path from a given state to a state that evaluates a given expression on some thread.

The definitions of these relations are the following.

$$\begin{aligned}
\mathit{Successor}(E, \hat{\varsigma}_1, \hat{\varsigma}_2) &\Leftrightarrow (\hat{\varsigma}_1, \_, \hat{\varsigma}_2) \in E \\
\mathit{Path}(E, \hat{\varsigma}_1, \hat{\varsigma}_2) &\Leftrightarrow \mathit{Successor}(E, \hat{\varsigma}_1, \hat{\varsigma}_2) \lor (\mathit{Successor}(E, \hat{\varsigma}_1, \hat{\varsigma}_{1'}) \land \mathit{Path}(E, \hat{\varsigma}_{1'}, \hat{\varsigma}_2)) \\
\mathit{TidExp}(\hat{T}, \widehat{\mathit{tid}}, e) &\Leftrightarrow \langle e, \_, \_, \_\rangle \in \hat{T}(\widehat{\mathit{tid}}) \\
\mathit{PathToTidExp}(E, \hat{\varsigma}, \widehat{\mathit{tid}}, e) &\Leftrightarrow \mathit{Path}(E, \hat{\varsigma}, \langle \hat{T}', \hat{\sigma}' \rangle) \ \land \ \mathit{TidExp}(\hat{T}', \widehat{\mathit{tid}}, e)
\end{aligned}$$

5.3.2 Unretried `cas` Analysis

A failed `cas` will be retried if the execution later reaches a state that evaluates the same `cas` on the same thread identifier. A program thus contains an unretried `cas` if there is a state such that:

- the state evaluates a `cas` on some thread,

- a successor state contains a `#f` control component in a context on the same thread, indicating that the `cas` failed in this successor,

- there is no path to a state that evaluates the same `cas` on the same thread.

Note that by *the same cas*, we mean the same expression in the AST node. It is necessary because requiring that we reach a `cas` on the same variable is not sufficient, as the value written to the variable might be different. Thus, whether the first `cas` succeeded or failed, we might have a different outcome. Also, requiring that we reach a `cas` with the same values for the two last arguments will cause some problems, as it is possible, and maybe frequent, depending on the abstraction used for values, that the values are abstracted and it is then impossible to know whether two values are equal.

This analysis is formalized as the $\mathit{UnretriedCas}(e) \subset \mathit{Exp}$ relation, defined as follows.

$$\begin{aligned}
\mathit{UnretriedCas}(e) \Leftrightarrow (V, E) = \widehat{\mathit{geval}}(e) \land \exists \langle \hat{T}_1, \hat{\sigma}_1 \rangle \in V, \exists \widehat{\mathit{tid}} \in \mathit{dom}(\hat{T}_1) \ \ \land& \\
\mathit{TidExp}(\hat{T}_1, \widehat{\mathit{tid}}, \overbrace{(\texttt{cas}\ \_\ \_\ \_)}^{e}) \land& \\
\mathit{Successor}(E, \langle \hat{T}_1, \hat{\sigma}_1 \rangle, \langle \hat{T}_2, \hat{\sigma}_2 \rangle) \land& \\
\mathit{TidExp}(\hat{T}_2, \widehat{\mathit{tid}}, \texttt{\#f}) \land& \\
\neg \mathit{PathToTidExp}(E, \langle \hat{T}_2, \hat{\sigma}_2 \rangle, \widehat{\mathit{tid}}, e)&
\end{aligned}$$

All the programs for which the *UnretriedCas* relation holds will possibly have a failed `cas` not retried. This analysis may find false positives, as we have no way of ensuring that the successor which evaluates to `#f` really corresponds to the same concrete thread as the one evaluating the `cas`. However, this only happens under specific circumstances. We should indeed have more than one context associated with $\widehat{\mathit{tid}}$ in $\hat{T}_2$, which can often be avoided by choosing an adapted abstract thread identifier allocation mechanism (the $\widehat{\mathit{newtid}}$ function, described in Section 4.3.4). Additionally, among those contexts associated to $\widehat{\mathit{tid}}$ there should be one related to another concrete thread evaluating to `#f`.

As for the conflict analysis, the unretried `cas` analysis is also unsound, because if we find a path to a state evaluating the same `cas` on the same abstract thread, we cannot know whether it corresponds to the same concrete thread (it might just be another concrete thread that has been associated with the same abstract thread identifier), or if this state is reachable. The choice of the thread identifier allocation mechanism is thus important for the analyses. In general, as soon as more than one concrete thread is associated with the same abstract thread identifier, most of the analyses will see their usefulness diminished, either because they will have a low precision, or because they might become unsound as it is the case here.

5.4 Race Condition Detection

It is possible to combine the conflict analysis with the unretried `cas` analysis to detect race conditions. In fact, any error found by one of these analyses may lead to a race condition. We can thus define the relation $\mathit{RaceCondition} \subset \mathit{Exp}$ as:

$$\mathit{RaceCondition}(e) \Leftrightarrow \mathit{Conflict}(e) \vee \mathit{UnretriedCas}(e)$$

As both the conflict analysis and the unretried `cas` analysis may lead to false positives, the race condition analysis may also have false positives. As we have shown that the unretried `cas` analysis is unsound in some specific cases, the race condition analysis itself is also unsound.

Also, this analysis does not check whether the race condition is harmful or not. A race condition can be *benign* if it does not influence the result of the program. It is arguable whether such races should be detected or not by a static analysis tool. They indeed are harmless for the program, but might become harmful when the code of the program evolves.

> **Example 17** (Benign race)**.** The following program shows an example of a benign race condition that is detected by this analysis. The variable `x` will always be set to 2 when this program terminates, no matter how the threads are interleaved.

```
(letrec ((x 1)
         (t1 (spawn (set! x 2)))
         (t2 (spawn (set! x 2))))
  (join t1)
  (join t2))
```

5.5 Deadlock Detection

As shown in Example 9, `cas` can be used as a building block for implementing locks. A naive implementation of locks can be the following. A lock is represented by a boolean variable that is true when the lock is locked. The lock can be locked with the `acquire` function and unlocked with the `release` function, implemented as follows:

```
(letrec ((lock #f)
         (acquire (lambda ()
                    (if (cas lock #f #t)
                        nil
                        (acquire))))
         (release (lambda ()
                    (set! lock #f))))
  ...)
```

As soon as we have the possibility of implementing locks, there is also a possibility of having *deadlocks* if a thread is trying to acquire a lock that is currently held by another thread that will not release this lock (e.g. if it is itself trying to acquire a lock held by the first thread). Example 18 shows what the state graph of a deadlock looks like.

Example 18 (Deadlock involving one thread)**.** We can simulate a deadlock with only one thread trying to acquire a lock that is initially already acquired, in order to have a relatively small state graph. The following program does exactly this.

```
(letrec ((lock #t) ; lock already locked
         (f (lambda ()
              (if (cas lock #f #t)
                nil
                (f)))))
  ;; Trying to acquire the lock, which is already locked
  (f))
```

The call to `cas` in `f` will always fail as the lock is already acquired. This can be seen in the portion of the state graph shown in Figure 5.1. We can indeed see that the call to `f` will never terminate as the execution will be blocked inside a cycle.

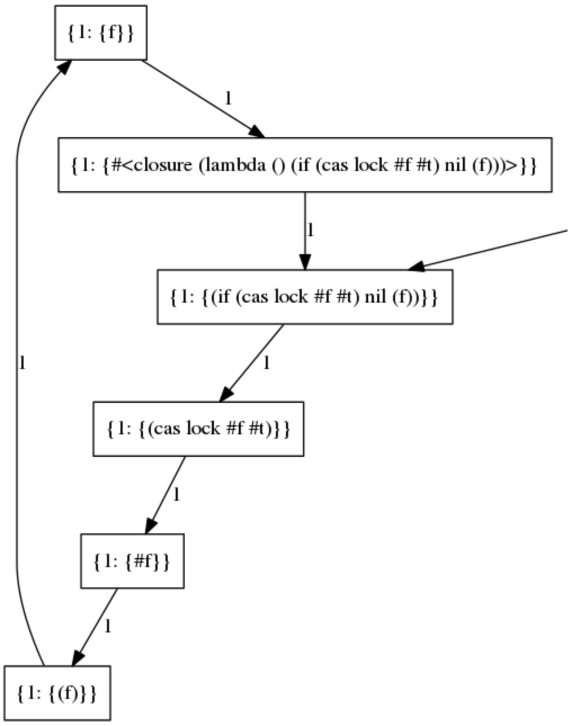
{1: {f}}
1
{1: {#<closure (lambda () (if (cas lock #f #t) nil (f)))>}}
1
{1: {(if (cas lock #f #t) nil (f))}}
1
{1: {(cas lock #f #t)}}
1
{1: {#f}}
1
{1: {(f)}}
1

Figure 5.1: Portion of the state graph related to a deadlock.

5.5.1 False positives

If we detect deadlocks from such patterns in the state graph, we will get many false positives. Indeed, when two threads want to acquire the same lock, and if thread 1 acquires it first, we can find a cycle where only thread 2 gets executed and continuously fail to acquire the lock, as thread 1 never gets executed along this cycle. While this might be a possibility, we do not want to detect such deadlocks as any real scheduler will eventually execute thread 1 again, allowing it to release its lock and avoiding the deadlock. We thus want to restrict such cycles in the graph to cycles where more than one thread has performed a transition.

There is one exception to this restriction. We still want to detect deadlocks involving a single thread such as in Figure 5.1. If the cycle starts at a state with only one `cas`, and every other thread is currently evaluating a join, then we detect this case as a deadlock. For it to be a real deadlock, the joins should all be blocked because of the thread doing the cas. For example, if thread 3 joins on thread 2 which joins on thread 1 which itself is continuously doing a cas, there is a deadlock. However, the case where there is no deadlock while having a cycle starting at a state with a single cas and multiple joins, seems quite rare. Requiring only cas and join in a state without further restriction does not introduce many false positives (we will investigate this in Section 6.5).

5.5.2 Precision

The main problem with our approach to detect deadlocks is that the value of the lock can be abstracted in two ways: the abstract value of the lock can be abstracted, or the abstract address where the value is stored can be merged with another address. As soon as the value of the lock gets abstracted (for example, if we know that the lock is a boolean but we do not know whether it is true or false), there might be no way of knowing whether the call to cas succeeded or not. This results in a state graph depicted in Figure 5.2, where we still have the cycle involving the cas, but there also exists an exit path of this cycle, due to the fact that we do not know whether the lock is locked or not. Such a pattern in a state graph cannot be distinguished from a correctly used cas, as both will have a cycle when the cas fails, and an exit branch when the cas succeeds.

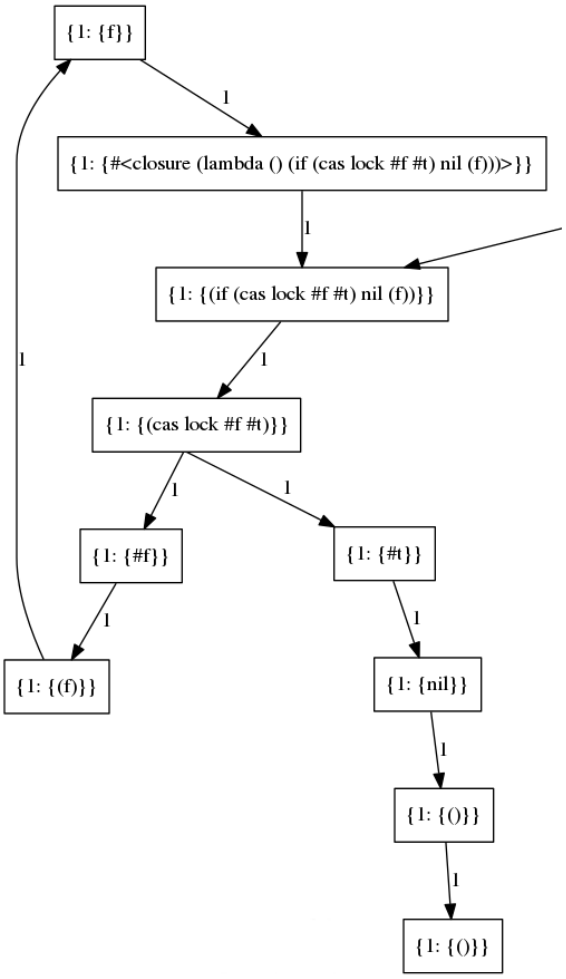

Figure 5.2: Part of the state graph related a deadlock where the value of the lock is too abstract.

5.5.3 Formalization

Since every correct use of cas will be indistinguishable from a deadlock, we give an unsound analysis that only detects a deadlock when, if the cycle is reached, it is sure that there will be a deadlock. This deadlock detection requires to look into more than one state, and, unlike the unretried `cas` detection of Section 5.3 where we looked for a failed `cas` with *no* path to itself when it failed, we are now looking for a path from a failed `cas` to itself. We also need to be sure that the `cas` will fail, i.e. it should not have a successor where it succeeded. Furthermore, we ensure that either one thread is involved in the deadlock and other threads are blocked by `join`s, or the deadlock involves more than one thread.

We need the following helper relations.

- $\mathit{Cycle} \subset \mathit{Edges} \times \hat{\Sigma}$ checks whether there exists a cycle from a state to itself in the state graph.

$$\mathit{Cycle}(E, \hat{\varsigma}) \Leftrightarrow \mathit{Path}(E, \hat{\varsigma}, \hat{\varsigma})$$

- $\mathit{FailingCas} \subset \mathit{Edges} \times \hat{\Sigma} \times \widehat{\mathit{TID}}$ checks whether there is a `cas` evaluated at the given node on the given thread, and ensures that this `cas` cannot succeed.

$$\begin{aligned}\mathit{FailingCas}(E, \langle \hat{T}, \hat{\sigma} \rangle, \widehat{tid}) \Leftrightarrow\, & \mathit{TidExp}(\hat{T}, \widehat{tid}, \texttt{(cas \_ \_ \_)}) \wedge \\ & \neg(\mathit{Successor}(E, \langle \hat{T}, \hat{\sigma} \rangle, \langle \hat{T}', \hat{\sigma}' \rangle) \wedge \\ & \quad \mathit{TidExp}(\hat{T}', \widehat{tid}, \texttt{\#t}))\end{aligned}$$

- $\mathit{NumberOfNotJoins} \subset \widehat{\mathit{Threads}} \times \widehat{\mathit{TID}} \times \mathbb{N}$ counts, for a given thread in a given state, the number of expression that do *not* evaluate a `join`.

$$\mathit{NumberOfNotJoins}(\hat{T}, \widehat{tid}, n) \Leftrightarrow |\{\hat{c} \in \hat{T}(\widehat{tid}) \text{ s.t. } \hat{c} \neq \langle \texttt{(join \_)}, \_, \_, \_ \rangle\}| = n$$

- $\mathit{TransitionsOnPath} \subset \mathit{Edges} \times \hat{\Sigma} \times \hat{\Sigma} \times \mathcal{P}(\widehat{\mathit{TID}})$ can be used to compute the set of threads for which a transition rule is used, along a path.

$$\begin{aligned}\mathit{TransitionsOnPath}(E, \hat{\varsigma}_1, \hat{\varsigma}_2, \mathit{Tr}) \Leftrightarrow & ((\hat{\varsigma}_1, \widehat{tid}, \hat{\varsigma}_2) \in E \;\wedge\; \mathit{Tr} = \{\widehat{tid}\}) \vee \\ & ((\hat{\varsigma}_1, \widehat{tid}, \hat{\varsigma}_{1'}) \in E \;\wedge\; \mathit{TransitionsOnPath}(E, \hat{\varsigma}_{1'}, \hat{\varsigma}_2, \mathit{Tr}') \wedge \\ & \; \mathit{Tr} = \mathit{Tr}' \cup \{\widehat{tid}\})\end{aligned}$$

- $\mathit{NumberOfTransitionsOnCycle} \subset \mathit{Edges} \times \hat{\Sigma} \times \mathbb{N}$ counts the number of different threads for which a transition rule is used, on a cycle from a given state to itself.

$$\mathit{NumberOfTransitionsOnCycle}(E, \hat{\varsigma}, n) \Leftrightarrow \mathit{TransitionsOnPath}(E, \hat{\varsigma}, \hat{\varsigma}, \mathit{Tr}) \wedge |\mathit{Tr}| = n$$

The deadlock analysis is formalized by the *Deadlock* $\subset$ *Exp* relation.

$$\begin{aligned}\mathit{Deadlock}(e) \Leftrightarrow\, & (V, E) = \widehat{\mathit{geval}}(e) \wedge \exists \langle \hat{T}, \hat{\sigma} \rangle \in V, \exists \widehat{tid} \in \mathit{dom}(\hat{T}) \wedge \\ & \mathit{FailingCas}(E, \langle \hat{T}, \hat{\sigma} \rangle, \widehat{tid}) \wedge \\ & \mathit{Cycle}(\langle \hat{T}, \hat{\sigma} \rangle) \wedge \\ & ((\forall \widehat{tid}' \neq \widehat{tid}, \mathit{NumberOfNotJoins}(\hat{T}, \widehat{tid}', 0) \wedge \\ & \quad \mathit{NumberOfNotJoins}(\hat{T}, \widehat{tid}, 1)) \vee \\ & \; (\exists n > 1, \mathit{NumberOfTransitionsOnCycle}(E, \langle \hat{T}, \hat{\sigma} \rangle, n))\end{aligned}$$

This analysis might not seem useful because in most programs values will eventually become abstracted and it will not be possible to distinguish whether a `cas` will succeed or fail. However, as our (naive) implementation of locks explicitly sets the value of the lock from `#f` to `#t` and from `#t` to `#f`, this loss of precision will generally not happen for such locks and we will always know whether the lock can be successfully acquired or not. The only case where the lock value will be abstracted is when its location in the store gets joined with another value because another variable is stored at the same address.

This analysis can thus be used to analyze programs that make use of such locks (or any other implementation of locks that preserves this property), even though it will generally not be able to detect deadlocks in a program that uses `cas` in a more general way (that is, by eventually setting a variable used in a cas to an abstracted value).

5.6 Conclusion

In this chapter, we presented several useful concurrency analyses. We first improved Might and Van Horn's MHP analysis by avoiding detection of false positives when checking whether an expression may be evaluated in parallel with itself. This analysis inspired our conflict analysis, where we look for concurrent accesses to the same store addresses. As this analysis needs to detect whether a write operation may happen in parallel with itself, it takes advantage of our improvement in the MHP analysis.

The other analyses required to look into the relation between multiple states in the state graph, and we introduced the necessary formalisms to do so. The unretried `cas` analysis detects when a failed `cas` is not retried, in which case it may lead to a race condition. By combining the conflict analysis with the unretried `cas` analysis, we were able to express a higher-level race condition analysis. Finally, we described a deadlock analysis. This analysis cannot distinguish some deadlocks from correct uses of `cas`, and therefore it is made unsound to avoid detecting too many false positives.

Any analysis that requires to look into more than one node of the state graph is unsound. This is the case for the unretried `cas` analysis and the deadlock analysis. The reason is that, as the abstract interpreter has a finite thread map, multiple threads might be assigned to the same thread identifier. Therefore, we cannot be sure that the evolution of an abstract thread from one state to the next corresponds to the evolution of a single concrete thread. Consequently, these analyses become unsound, meaning that potential defects might not be found. However, as soon as more than one thread is assigned to the same thread identifier, precision plummets and the usefulness of the analysis decreases anyway. It is acceptable to have an unsound analysis in this case as it will remain sound until the program becomes too complex to analyze.

The analyses described in this chapter will be validated in Chapter 6. The problem of the potentially large amount of false positives for the deadlock analysis will be addressed in Chapter 8 by introducing first-class locks into the language.

Chapter 6

Validation of the Analyses

In this chapter, we verify that the analyses described in the previous chapter actually detect the faults they were designed to detect, based on several examples. Here, we are only interested in whether each analysis works as expected and not in its running time. Most of the time taken to perform an analysis is spent on building the state graph, which is what is actually measured by the benchmarks of the PCESK machine, in Chapter 7. The analysis only needs to traverse this state graph, which often take a negligible amount of time compared to the time taken to compute this graph.

The results of each analysis are summarized in a table in each section, describing the complexity of the examples, the expected results, and the results obtained with the analysis. The complexity of an example depends on the number of AST nodes the source code contains, as well as the number of threads that are involved. For the deadlock analysis, the number of locks involved in each example is also given.

The results are given in terms of total number of defects found, how many among them are true positives (*tp*), and how many are false positives (*fp*). Each result also implicitly has information about the number of true negatives (*tn*), and false negatives (*fn*).

- True positives are *correctly* detected defects. The results of an analysis on an example might produce multiple true positives. For example, if a program contains two race conditions and both are detected, there are two true positives.

- True negatives correspond to the *correct* detection of the absence of defects. The result of an analysis on an example might produce either 0 or 1 true negatives. There is *no* true negative when the program doesn't contain any of the defects the analysis should find, but the analysis has false positives. That is, the analysis considers the program to have defects, while it doesn't. There is *one* true negative if the program doesn't have any of the defects the analysis should find, and the analysis correctly doesn't detect any defect.

- False positives are *incorrectly* detected defects. There can be 0 or more false positives on an example for an analysis. Ideally, an analysis should minimize the number of false positives.

- False negatives are *missed* defects. That is, an analysis has false negatives on a program that contains defects if the analysis doesn't detect all the defects it should find on the program. An analysis for a defect on an example has n false negatives if the example does contain n' occurrences of this defect, but the analysis only finds $n' - n$ occurrences among the true occurrences of the defect.

Minimizing the number of false negatives is more important than minimizing the number of false positives. Indeed, having false negatives implies that some programs will be labeled as correct even though they might contain bugs that the analysis should have detected. On the other hand, having false positives implies that correct programs might be labeled as incorrect, which is considered less critical than identifying an incorrect program as correct.

For each analysis, we compute the *precision*, the *recall* and the *accuracy* as follows:

$$\begin{aligned}\text{Precision} &= \frac{tp}{tp+fp}\\ \text{Recall} &= \frac{tp}{tp+fn}\\ \text{Accuracy} &= \frac{tp+tn}{tp+tn+fp+fn}\end{aligned}$$

The precision tells us how many defects founds were indeed real defects. A *complete* analysis has a precision of 100%. The recall tells us how many defects were found among all the existing defects. A *sound* analysis has a recall of 100%. The accuracy tells us how the analysis performed overall. An analysis without any error will have an accuracy of 100%.

Note that the polyvariance (k) of the abstract interpretation did not influence the results presented in this chapter. Having $k = 0$ is thus sufficient to use these analyses for the examples shown in this chapter.

6.1 May-Happen-in-Parallel Analysis

To check whether the MHP analysis works as expected, we use two parallel counter examples that use `set!`: the first example (Example 19) protects the critical section by a lock, while the second (Example 20) does not protect it. Those two examples are given below.

We are interested into two expressions: the read expression denoted e_r and the write expression denoted e_w. In the first example, as the critical section is protected by a lock, we should have $\neg \mathit{MHP}(e_r, e_w)$, as well as $\neg \mathit{MHP}(e_w, e_w)$. However, in the second example, as no locks are used to ensure that only one thread enters the critical section at a time, there exists execution paths were those expressions may be evaluated in parallel (both for e_w and e_r, and e_w and itself).

Example 19 (`pcounter-mutex.scm`)**.** This is the same example as Example 9, annotated to show e_r and e_w.

```
(letrec ((lock #f)
         (acquire (lambda ()
                    (if (cas lock #f #t)
                        nil
                        (acquire))))
         (release (lambda ()
                    (set! lock #f)))
         (counter 0)
         (inc (lambda ()
                (acquire)
```

$$\underbrace{\texttt{(set! counter (+ }\underbrace{\texttt{counter}}_{e_r}\texttt{ 1))}}_{e_w}$$

```
                (release)))
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  counter)
```

Example 20 (`pcounter-race.scm`)**.** This is the same example as Example 12, annotated to show e_r and e_w.

```
(letrec ((counter 0)
         (inc (lambda ()
```

$$\underbrace{\texttt{(set! counter (+ }\underbrace{\texttt{counter}}_{e_r}\texttt{ 1))}}_{e_w}\texttt{))}$$

```
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  counter)
```

The results found by our implementation and by Might and Van Horn's original MHP analysis (ran through our implementation) are given in Table 6.1.

Example	Length	Threads	Query	Expected	MVH's MHP	Our MHP
`pcounter-mutex.scm`	45	2	$MHP(e_r, e_w)$	no	no	no
			$MHP(e_w, e_w)$	no	yes	no
`pcounter-race.scm`	24	2	$MHP(e_r, e_w)$	yes	yes	yes
			$MHP(e_w, e_w)$	yes	yes	yes
				Precision	66%	100%
				Recall	100%	100%
				Accuracy	75%	100%

Table 6.1: Results of the MHP analysis.

6.2 Conflicts Detection

To verify the validity of the conflict detection analysis, we run it on various examples with or without conflicts. Our implementation contains multiple variations (called *targets*) of the conflict analysis:

- the `setconflicts` target finds conflict that only involves `set!`, and no `cas`,
- the `allconflicts` target finds conflict that can involve `set!` and/or `cas`,
- the `conflicts` target finds the same conflicts as the `allconflicts` target, but filters out those that are not considered are harmful, as explained in Section 5.2.2.

In general, it is sufficient to run the `conflicts` target as it will detect every `set!` conflicts as well as harmful `cas` conflicts, without producing too much false positives. However, we run the three analyses and show their result in Table 6.2. The examples used are the following.

- `pcounter.scm`: Example 8, page 32, a parallel counter correctly implemented with `cas`.
- `pcounter-mutex.scm`: Example 9, page 33, a parallel counter correctly implemented with locks, themselves implemented with `cas`.
- `pcounter-race.scm`: Example 12, page 46, a parallel counter incorrectly implemented and containing a race condition due to read/write and write/write conflicts.

- `pcounter-buggy.scm`: Example 21, page 61, a parallel counter containing a non-trivial implementation error, leading to a race condition.

- `false-pos.scm`: Example 15, page 50, containing a possible false positive if allocated addresses only consists of the last k call-sites.

- `benign.scm`: Example 17, page 53, containing a race condition not influencing on the result of the program.

- `race-cas.scm`: Example 16, page 51, containing a race condition due to an unretried `cas`.

- `race-set-cas.scm`: Example 22, page 62, also containing a race condition due to an unretried `cas`.

Example 21 (Subtle `cas` conflict)**.** The following program is a variant of Example 8 with a subtle error. When computing the new value of `counter` in the variable `new`, we refer to `counter` itself instead of `old`. Even though the outcome of this program is not affected by this conflict (if one thread modifies `counter` between the computation of `old` and `new`, the `cas` will fail as the counter value can only increase in this particular program), it might be problematic in other situations, where the value of `counter` might be changed when before computing `new` and restored before doing the `cas`. This problem is similar to what was explained in Example 13, which contains a real race condition but takes much more time to analyze, as it defines a decrease operation and involves three spawned threads.

```
(letrec ((counter 0)
         (f (lambda ()
              (letrec ((old counter)
                       ;; Bug introduced here
                       (new (+ counter 1)))
                (if (cas counter old new)
                  #t
                  (f)))))
         (t1 (spawn (f)))
         (t2 (spawn (f))))
  (join t1)
  (join t2)
  counter)
```

As noted previously the `setconflicts` target only detects conflicts involving `set!`, and we can see that it correctly detects the conflicts in the program containing such conflicts (`pcounter-race.scm`, `benign.scm`)

The `allconflicts` target detects every conflicts involving `set!` or `cas`, without filtering anything. We can see that, as we expected, it produces many false positives.

The false positive program is correctly handled by all the targets, as we use timestamps with a *tag* component and not only the last k call sites, as explained in Section 5.2.2.

As no effort has been made to avoid detecting benign race conditions (that is, race conditions that does not affect the outcome of the program), the three analyses detects a conflict in the `benign.scm` program, even though this conflict is benign.

Finally, the `race-cas.scm` program contains a race condition which is not detected by the `conflicts` target because it is filtered out, since it only involves a conflict between two `cas` on the same address, which could happen in a correct program. This example will however be detected as problematic by the unretried `cas` analysis.

Example	Length	Threads	Expected	`conflicts`			`setconflicts`			`allconflicts`		
				found	tp	fp	found	tp	fp	found	tp	fp
`pcounter.scm`	34	2	0	0	0	0	0	0	0	2	0	2
`pcounter-mutex.scm`	45	2	0	0	0	0	0	0	0	1	0	1
`pcounter-race.scm`	24	2	2	2	2	0	2	2	0	2	2	0
`pcounter-buggy.scm`	34	2	3	3	3	0	0	0	0	3	3	0
`false-pos.scm`	34	2	0	0	0	0	0	0	0	0	0	0
`benign.scm`	17	2	0	1	0	1	1	0	1	1	0	1
`race-cas.scm`	20	2	1	0	0	0	0	0	0	1	1	0
`race-set-cas.scm`	19	2	1	1	1	0	0	0	0	1	1	0
			Precision	86%			66%			64%		
			Recall	86%			29%			100%		
			Accuracy	81%			45%			62%		

Table 6.2: Results of the conflict analysis.

6.3 Unretried `cas` Detection

When a `cas` that fails is not retried, it may lead to a race condition, such as in Example 16. Note that this example also contains a conflict, which is not detected by the `conflict` target, as we filter out some patterns that are considered harmless. Such a conflict is however harmful, as the outcome of the program will depend on which thread gets executed first, but this is due to the fact that `cas` is not retried when it fails. By combining the `conflict` target and the `unretriedcas` target we can thus detect many potential errors.

A variation of this example is Example 22, which involves a conflict between a `set!` and a `cas`. The results of our unretried `cas` analysis are given in Table 6.3.

Example 22 (`race-set-cas.scm`)**.** The outcome of the following program depends on which thread gets executed first:

```
(letrec ((x 0)
         (t1 (spawn (set! x 2)))
         (t2 (spawn (cas x 2 1))))
  (join t1)
  (join t2)
  x)
```

Example	Length	Threads	Expected	`unretriedcas`		
				found	tp	fp
`pcounter.scm`	34	2	0	0	0	0
`pcounter-mutex.scm`	45	2	0	0	0	0
`pcounter-race.scm`	24	2	0	0	0	0
`pcounter-buggy.scm`	34	2	0	0	0	0
`false-pos.scm`	34	2	0	0	0	0
`benign.scm`	17	2	0	0	0	0
`race-cas.scm`	20	2	2	2	2	0
`race-set-cas.scm`	19	2	1	1	1	0
			Precision	100%		
			Recall	100%		
			Accuracy	100%		

Table 6.3: Results of the unretried `cas` analysis.

The unretried `cas` analysis doesn't detect any unretried `cas` when there is none, as expected, and correctly detects them on the two concerned examples. Note that since the `race-set-cas.scm` example contains not only an unretried `cas`, but also a write/write conflict between the `cas` and the `set!`, it is already detected as flawed by the conflict analysis.

6.4 Race Condition Detection

The race condition analysis consists of combining the conflict analysis with the unretried `cas` analysis. We have seen that many harmful conflicts are detected by the conflicts analysis, and those who were not contain unretried `cas` that were detected by the corresponding analysis. We were not able to find any harmful example that is not detected by one of the two analyses and thus believe that by combining these two analyses, it is possible to find a great number of race conditions in programs.

We can combine Tables 6.2 and 6.3 to obtain Table 6.4. The two unretried `cas` detected allows us to detect the missing race condition of the `race-cas.scm` example. As the race condition in the `race-set-cas.scm` example is already detected, the unretried `cas` does not bring more information. We can see an improvement in the recall and accuracy of the analysis, compared to races found by only using the conflict analysis.

Example	Length	Threads	Expected	`race`		
				found	tp	fp
`pcounter.scm`	34	2	0	0	0	0
`pcounter-mutex.scm`	45	2	0	0	0	0
`pcounter-race.scm`	24	2	2	2	2	0
`pcounter-buggy.scm`	34	2	3	3	3	0
`false-pos.scm`	34	2	0	0	0	0
`benign.scm`	17	2	0	1	0	1
`race-cas.scm`	20	2	1	1	1	0
`race-set-cas.scm`	19	2	1	1	1	0
			Precision	88%		
			Recall	100%		
			Accuracy	90%		

Table 6.4: Results of the race condition analysis.

6.5 Deadlock Detection

As explained in Section 5.5, the deadlock analysis does not work as soon as the value of the variable that is the source of the deadlock becomes too abstracted. We however state that it works as long as we want to detect deadlocks involving locks implemented on top of a `(cas lock #f #t)` and no loss of precision happens due to the finiteness of the store, as this `cas` will never introduce by itself a loss of precision in the value of the lock. The only possible loss of precision happens when two different variables get joined in the store, which we generally want to avoid as it is a major cause of imprecision in many analyses.

We verify this behavior by running this analysis on the following examples. The results are given in Table 6.5.

- `deadlock-simple.scm`: Example 18, page 54, containing a deadlock involving a single thread.
- `deadlock-abstract.scm`: a program similar to `deadlock-simple.scm`, but we force the initial lock value to be abstract, by setting it to the result of evaluating `(= 1 1)`, which is `AbsBoolean` in our implementation.

- `deadlock1.scm`: Example 23, page 64, containing a deadlock involving one lock shared among two threads, where both threads acquire the lock and none release it.
- `deadlock1-release.scm`: a program similar to `deadlock1.scm`, but we correctly release the lock in one of the two threads, removing one of the two possible deadlocks.
- `deadlock.scm`: Example 24, page 64, containing a deadlock involving two locks shared among two threads, with a circular dependency in the order of lock acquisition.
- `pcounter-mutex.scm`: Example 9, page 33, not containing a deadlock but using locks to ensure that the critical section is reached by only one thread at a time.

Example	Length	Threads	Locks	Expected	`deadlocks`		
					found	tp	fp
`deadlock-simple.scm`	15	0	1	1	1	1	0
`deadlock-abstract.scm`	18	0	1	1	0	0	0
`deadlock1.scm`	27	1	1	2	2	2	0
`deadlock1-release.scm`	29	1	1	1	1	1	0
`deadlock.scm`	65	2	2	2	4	2	2
`pcounter-mutex.scm`	45	2	1	0	0	0	0
				Precision	75%		
				Recall	85%		
				Accuracy	70%		

Table 6.5: Results of the deadlock analysis.

Example 23 (Deadlock involving one lock and two threads)**.** The following program acquires a lock in two different threads, forgetting to release it. There is two possibilities of deadlock: either the main thread acquires the lock first and thread `t1` can never acquire its lock, creating a deadlock as the main thread waits thread `t1`; or thread `t1` acquires the lock first and the main thread gets blocked trying to acquire the lock.

```
(letrec ((lock #f)
         (acquire (lambda ()
                    (if (cas lock #f #t)
                        nil
                        (acquire))))
         (release (lambda ()
                      (set! lock #f)))
         (t1 (spawn (begin
                      (acquire)
                      ; forgot to release the lock!
                      ))))
  (acquire)
  (join t1))
```

Example 24 (Deadlock involving two locks and two threads)**.** This programs is a typical case of cyclic dependencies between locks: thread `t1` wants to acquire lock `a` then lock `b`, while thread `t2` wants to acquire lock `b` then lock `a`, thus possibly leading to a deadlock (but not always).

```
(letrec ((lock-a #f)
         (lock-b #f)
         (acquire-a (lambda ()
                      (if (cas lock-a #f #t)
                          nil
                          (acquire-a))))
         (release-a (lambda ()
                      (set! lock-a #f)))
         (acquire-b (lambda ()
                      (if (cas lock-b #f #t)
                          nil
                          (acquire-b))))
         (release-b (lambda ()
                      (set! lock-b #f)))
         (t1 (spawn (begin
                      (acquire-a)
                      (acquire-b)
                      (release-b)
                      (release-a))))
         (t2 (spawn (begin
                      (acquire-b)
                      (acquire-a)
                      (release-a)
                      (release-b)))))
  (join t1)
  (join t2))
```

As expected, the deadlock detection correctly detects deadlocks in the cases where there is no loss of precision on the value of the lock. On example `deadlock-abstract.scm` however, as we force this value to be abstract, the deadlock is not found anymore, as a cycle exists but it cannot be distinguished from a correct use of `cas`.

The `deadlock1.scm` and `deadlock1-release.scm` programs shows that the analysis detects, as expected, one fewer possible deadlock when one of the threads correctly releases the lock.

Note that two false positives are detected on the `deadlock.scm` example. Due to the size of the state graph (around 6000 nodes), it has not been feasible to further investigate the source of those false positives.

On the `pcounter-mutex.scm` example, the analysis does *not* detect anything, as this program is deadlock-free.

One major downside of this analysis is the time it takes. As said previously, this time is mostly spent on building the state graph, and the problem is not a consequence of the analysis itself, but of the reason why this state graph takes time to build. For simple examples it stays relatively fast (around 4s on `deadlock-simple.scm`, 1min30s on `pcounter-mutex.scm`), but as soon as the program gets more complicated (especially if it uses many locks), the analysis time explodes. On `deadlock.scm`, the analysis takes 24 minutes to complete. This is due to the fact that, because locks are implemented as loops over calls to `cas`, the state space gets really big as the number of possible intervealings between the thread locks increases.

The deadlock analysis can thus detect deadlocks on simple programs involving up to two locks and two threads, but more complicated programs do not terminate in a reasonable amount of time or need too much memory.

6.6 Conclusion

In this chapter, we validated the analyses developed in the previous chapter. We showed that our adaptation of Might and Van Horn's MHP analysis is more precise than the original, as it does not detect false positives where an expression was considered to always be evaluated in parallel with itself as soon as it was evaluated once.

We compared the different approaches of detecting memory access conflicts: detecting only conflicts involving `set!`, detecting every conflict, or filtering some patterns of conflicts. As shown in Table 6.2, detecting all the conflicts seems sound but has low precision. By allowing unsoundness we can increase this precision, while the missed conflicts are detected by the unretried `cas` analysis.

The unretried `cas` analysis works as expected, and no example invalidating it was found. By combining this analysis with the conflict analysis that filters out some patterns, we obtain an analysis that can detect every race condition present in our example programs, with only one false positive.

Finally, even with the limitations we identified in the previous chapter when defining the deadlock analysis, we observed that this analysis is able to detect a certain number of deadlocks with good precision. As long as the locks involved in the programs are used through the user-defined `acquire` and `release` functions, the loss of precision is limited (it can only come from the finiteness of the store) and deadlocks are detected.

We did not mention the time taken for each analysis in this chapter. This is done in the benchmarks of the next chapter, which uses the same examples as the ones used here for the validation.

Chapter 7

Implementation and Benchmarks

The PCESK machine described in Chapter 4 has been implemented in OCaml and is publicly available[1]. In this chapter, we describe this implementation and the results of some benchmarks made to measure the impact of the refinements introduced in Sections 3.1.4 and 4.4 on the CESK and the PCESK machines.

7.1 Implementation of the CESK and PCESK Machines

This section describes the features of the implementation and how to use them. We then go into more details on the implementation and its relation to what has been described until now in this work. A more detailed description of how to use the implementation can be found in the `README` provided with the source code.

7.1.1 Features

This implementation supports the full CScheme language described in Section 4.1 with some additions:

- basic operations are supported on primitive values (addition, subtraction, comparison, negation, . . .),
- lists can be constructed and accessed with the conventional Scheme operators (`cons`, `car`, `cdr`, `empty?`),
- `call/cc` is supported.

All of the improvements discussed in Section 4.4 (abstract counting, abstract thread counting, thread removal, abstract garbage collection, and state subsumption) are implemented, and can be turned on or off by using arguments when launching the executable. Abstract counting, abstract thread counting and thread removal are turned on by default, but not garbage collection nor state subsumption, as they can have a negative impact on the analysis time. Polyvariance (the value of k, used to generate new addresses) can be specified. By default, support for parallel special forms is not turned on, and the CESK machine is used instead of the PCESK machine.

The input program is parsed, and a unique tag is assigned to every node of the AST, to be able to differentiate two nodes with the same expressions. Multiple operations (called *targets*) can then be run on this input program.

- By default, an abstract interpretation of the program is performed, the state graph is computed and can be written to a file in order to inspect it manually.

[1] `https://github.com/acieroid/pcesk`

- The parsed AST can be printed with tag annotations to be able to find the tag corresponding to an expression. Those tags are needed to specify expressions on which to perform the MHP analysis.

- The MHP analysis can be carried out if two expression tags are given. The abstract state graph will be built and then inspected in order to check whether the two expressions may happen in parallel in the given program, as described in Section 5.1.

- Multiple targets are available for detecting conflicts, and correspond to detecting only conflicts involving `set!`, conflicts that also involves `cas`, or all conflicts except those that match some patterns that are assumed correct.

- The unretried `cas` analysis of Section 5.3 can be performed.

- The deadlock detection analysis of Section 5.5 can be performed.

Example 25 (Detection of conflicts with the implementation)**.** To analyze the program of Example 12, which contains a race condition due to a conflict, we can run the implementation as follows.

```
$ ./main.byte -i input/pcounter-race.scm -target conflicts -p
```

The `-p` switch enables the use of the PCESK machine instead of the CESK machine, thus allowing the use of the parallel operators. The output is the following.

```
2 conflicts detected between the following pairs of expressions:
(set! counter (+@9 counter@10 1@11)@8)@6, (set! counter (+@9 counter@10 1@11)@8)@6
(set! counter (+@9 counter@10 1@11)@8)@6, counter@10
```

As can be seen in the output, this program contains two conflicts (read/write and write/write), and the corresponding expressions are printed along with their tag so that they can be located in the program.

7.1.2 Architecture

The architecture of the implementation puts a great emphasis on separating the different components, and the possibility to provide alternative implementations for each of those components. Also, there is a clear separation between the implementation of the sequential CESK machine and the implementation of the PCESK machine, although the PCESK machine uses the CESK machine for the sequential transitions, as described in 4.2.

The implementation is organized in a number of components, which are themselves composed of one or more modules, described in the remainder of this section. Note that a module with a name in all capitals is a module type, and can be seen as an interface that modules of this type should satisfy.

AST, Lexer and Parser

The `Ast` module defines the abstract syntax for CScheme. This AST is defined through the types `var` corresponding to *Var*, `exp` corresponding to *Exp*, and `node` associating an *Exp* with a unique tag.

The modules `Lexer` and `Parser` respectively define the following functions that can be plugged together to parse a program from an input channel:

```
Lexer.lex : in_channel -> Tokens.scheme_token Stream.t
Parser.parse : Tokens.scheme_token Stream.t -> Ast.node
```

Types

The `Types` module defines all the types needed by the CESK and PCESK machines:

- primitive values (*Val*),
- intermediate abstracted values (which do not form a lattice, but are used by the *Lattice* module to define $\widehat{Val}$),
- continuations (*Kont*),
- timestamps (*Time*),
- addresses (*Addr*),
- thread identifiers (*TID*)

This module also defines operations on the intermediate abstracted values, in particular to know whether two such values can be merged. For example, `AbsUnique (Integer 1)` can be merged with `AbsUnique (Integer 2)` and gives `AbsInteger`, but `AbsInteger` cannot be merged with `AbsBool` since no top element is defined. In the set lattice, joining these elements gives the lattice element {`AbsInteger`, `AbsBool`}.

Lattice and SetLattice

The `LATTICE` module type defines an interface that lattice implementations should satisfy, containing the usual operations on lattices: abstraction, join ($\sqcup$), meet ($\sqcap$), bottom element. It also provides a way to define unary and binary operations on the elements of a lattice.

The `SetLattice` module is an implementation of this interface that implements the conventional set lattice (where a lattice element is a set of values). When joining two lattice values, it uses the merge operation defined on intermediate abstracted values in the `Types` module to know if it is possible to merge values contained inside the lattice. When adding an intermediate value to the lattice, if a merge is possible with this value and a value contained inside the lattice, the values are merged, else the new value are added to the set of values.

Address

The `ADDRESS` module type defines an interface for addresses, so that other modules can be parameterized on such adresses. However, the actual definition of addresses is done in the `Types` module.

Environment

The `ENV` module type defines an interface for environments that requires environments to define the various operations that will be needed: lookup, extend, restrict, range, comparison, subsumption. A map-based implementation of such environments is given (module `Env`), where an environment is a map (OCaml's `Map.S`) that binds strings to addresses.

Store

The `STORE` module type defines the interface for stores, and also provides a map-based implementation (module `Store`). The store is implemented as an OCaml functor[2] that takes two parameters: an `ADDRESS` module and a `LATTICE` module.

A store should implement the conventional operations (lookup, update, join, narrow, comparison, subsumption).

[2] An OCaml functor is a higher-order module, i.e. a module that is parameterized by other modules.

Exploration

The `EXPLORATION` module type defines the strategy used to explore the state graph. It provides two implementation: one performing a depth-first search (`Dfs`), the other performing a breadth-first search (`Bfs`). By default, the breadth-first search module is used.

CESK

The `Cesk` module aggregates all the previous modules to define a CESK machine that supports the CScheme language, except for `spawn` and `join`. An implementation of a simplified version of the CESK machine has already been described in Section 3.1.5. The main differences are that we support a bigger language, and compute the state graph instead of only looking at the final state as we did in the described implementation.

This component therefore also defines a module `G` implementing the state graph, represented by the type `G.t`. The CESK module provides two functions:

- `step : state -> state list`: given a state (Σ_{CESK}), return all the possible results of applying ($\widehat{\rightarrow}$) to this state, and
- `eval : Ast.node -> G.t`: given an AST node, return the computed state graph. It is in fact the $\widehat{geval}$ of Section 5.3.1, which is based on $\widehat{eval}$, defined in Section 4.3.6.

The exploration method can be tuned by the user by overriding a runtime parameter used in the `eval` function. Other parameters can be tuned to activate other submodules such as garbage collection and state subsumption.

The `Garbage_collection` submodule provides the two following functions.

- `reachable : state -> AddressSet.t` computes the set of addresses that are reachable in a CESK state. This corresponds to the $\mathcal{R} : \Sigma_{\text{CESK}} \rightarrow \mathcal{P}(Addr)$ function defined in Section 3.1.4.
- `gc : state -> state` performs garbage collection on a state, freeing the addresses that are not reachable, and returns the collected state. This corresponds to the transition function $(\rightarrow') : \Sigma_{\text{CESK}} \rightarrow \Sigma_{\text{CESK}}$

The `gc` function is used in CESK's `step` when garbage collection is activated. However, as explained in Section 4.4.4, to perform garbage collection on the PCESK machine it is necessary to take all the threads into account, else one address might be collected by the CESK machine corresponding to a thread, while it is accessible in another thread. Therefore, the PCESK machine will make use of the `reachable` function to compute by itself what can safely be removed, instead of using `gc` directly.

PCESK

The `Pcesk` module has a very similar structure as the `Cesk` module except that it implements the full CScheme language, including `spawn` and `join`. Its definition is described in Section 4.2 and rely on the `Cesk` module to perform sequential transitions as well as other actions such as abstract garbage collection.

The interface is identical to the CESK component, except that the `step` functions uses `pstates` (Σ) and represents ($\widehat{\Rightarrow}$) instead of using `states` (Σ_{CESK}) and representing ($\widehat{\rightarrow}$). Also, the `eval` function returns a graph whose vertices are `pstate`s instead of `state`s.

Analysis

A module exists for each analysis described in Chapter 5.

The `Mhp` module implements the may-happen-in-parallel analysis by defining the function `mhp : G.t -> int -> int -> bool`. This function takes a state graph built by

`Pcesk.eval`, two tags corresponding to the two expressions that we are interested into, and returns a boolean indicating whether those two expressions may happen in parallel or not.

The `Conflict` module implements the different kinds of conflict analyses by defining the function `conflicts : ?handle_cas:bool -> ?ignore_unique_cas:bool -> G.t -> Ast.node -> (Ast.tag * Ast.tag * Types.addr) list`. This function looks for conflicts in the graph given in the first non-optional argument, which is built by `Pcesk.eval` from the expression given in the second non-optional argument. This function will return the list of conflicts as a triple containing the two expressions involved in the conflict as well as the address concerned by the conflict. The two optional arguments correspond to the cases described in Section 5.2: when `handle_cas` is true, `cas` is handled as a write operation, and when `ignore_unique_cas` is true, the pattern corresponding to a correct use of `cas` is not treated as a conflict (only a write/write conflict between two `cas` involving the same variable, or a read/write and a write/write conflict involving the same variable).

The `Unretried_cas` module implements the unretried `cas` detection by defining the function `unretried_cas : G.t -> Ast.tag list`, which takes a state graph as argument, and returns the tags corresponding to the expressions where an unretried `cas` has been detected.

The `Deadlock` module implements the deadlock detection by defining the function `deadlock : G.t -> (Types.tid * Ast.tag) list`, which takes a state graph as argument, and returns a list containing the thread identifier corresponding to a thread involved in the deadlock, as well as the expression from which the deadlock has been detected (that is, the `cas` from which the cycle in the graph is found).

Main

The `Main` module glues together the other modules in order to present a unified interface to the user. It will call either the CESK's or the PCESK's `eval` function to build a state graph from the input program, invoke the analysis requested by the user, and displays the result in a human-readable way.

7.2 CESK Benchmarks

A series of benchmarks have been performed on the implementation of the CESK machine in order to check whether it behaves as expected with the refinements described in Section 3.1.4. The results of those benchmarks are represented in Figures 7.1 and 7.2 (the raw numbers are in Table 7.1), for $k = 0$. The results for other values of k do not differ much and lead to the same conclusions.

Those graphs show respectively the size of the state graph and the time taken build this state graph, for multiple examples with different refinements turned on or off: `nothing` stands for the CESK machine without any refinement (except abstract counting, which is always enabled), `gc` for the CESK machine with abstract garbage collection, `sbfs` for the CESK machine with state subsumption when the exploration is done in a breadth-first way, `sdfs` is similar but with the exploration done in a depth-first way, `gc+sbfs` and `gc+sdfs` combine abstract garbage collection with state subsumption.

As expected, the combination of garbage collection and state subsumption generally leads to a great improvement in the size of the state space and the processing time. This is not the case for the few examples where there is no gain in the size of the state space, and there is thus a loss in processing time.

Unlike the results of [ZR12], the state space reduction done by the state subsumption mechanism alone seems to have more impact when exploring the state space in a breadth-first way rather than in a depth-first way. When state space subsumption is combined with garbage collection however, the exploration method used does not matter anymore.

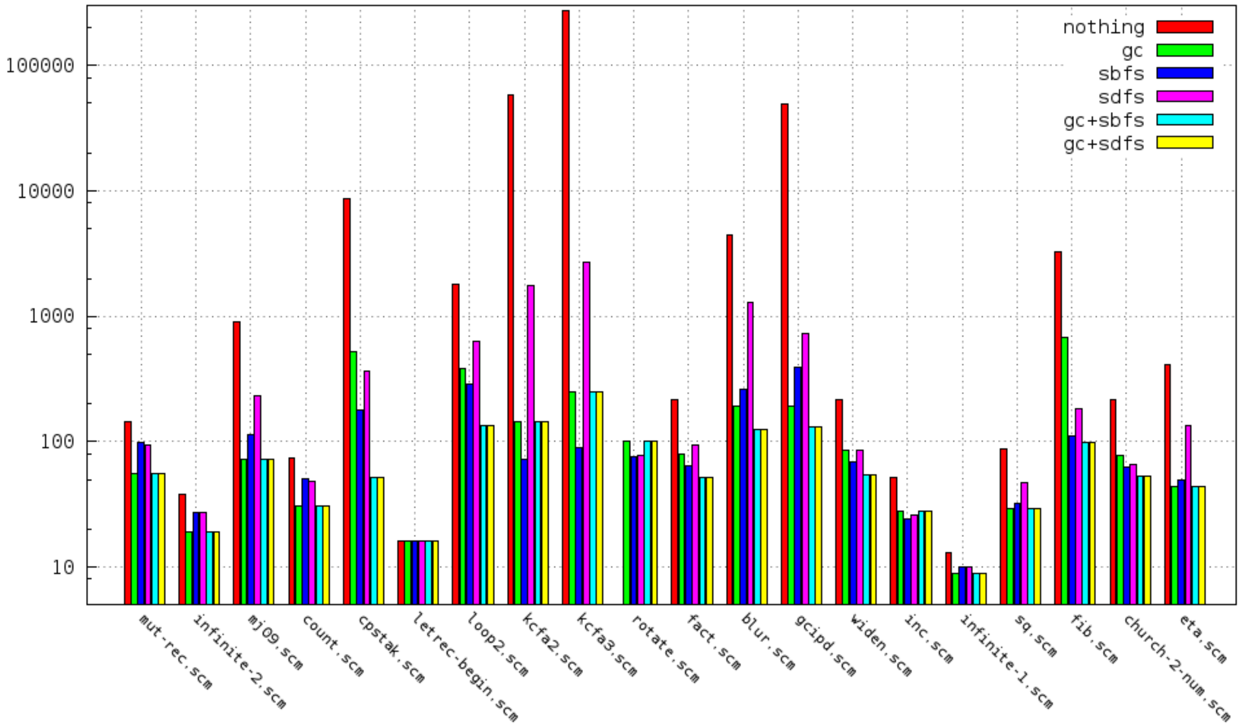

Figure 7.1: Number of states in the state graph computed by the CESK machine.

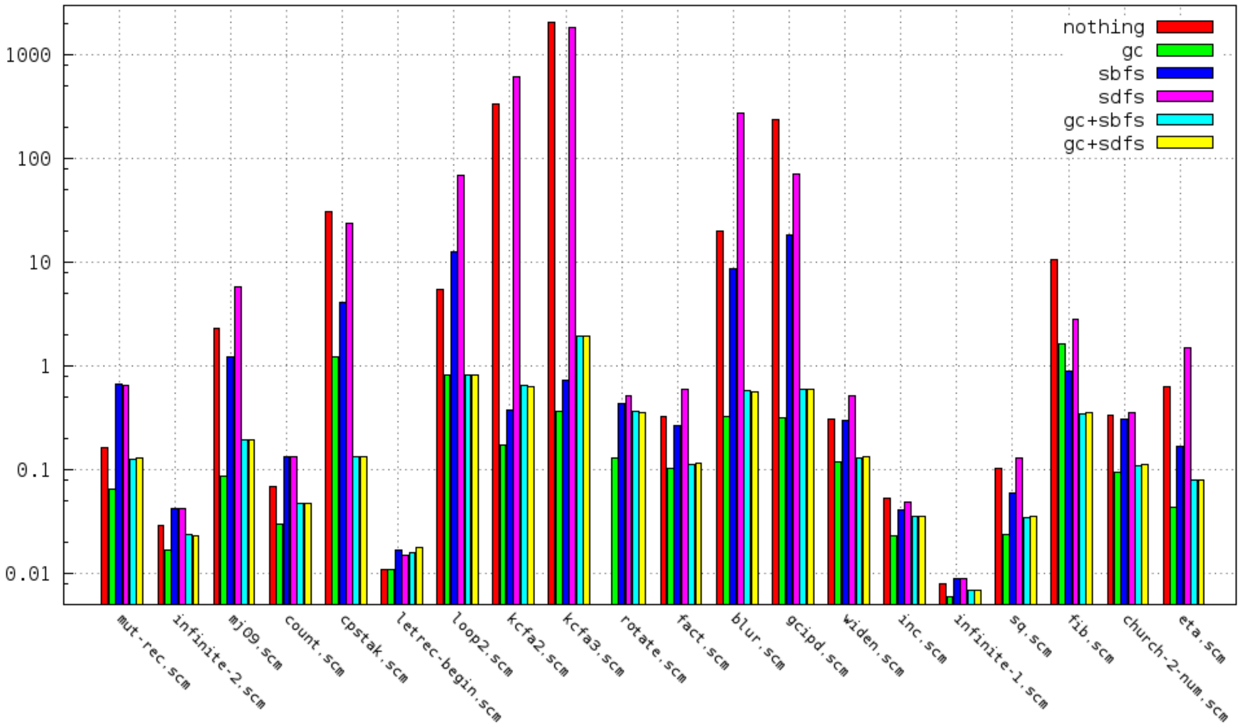

Figure 7.2: Time taken by the CESK machine to compute the state graph.

It seems to be reasonable to use both state subsumption and garbage collection by default for the CESK machine, as the state space tends to generally be smaller for a good improvement in processing time.

example	nothing		gc		sbfs		sdfs		gc+sbfs		gc+sdfs	
	#	t	#	t	#	t	#	t	#	t	#	t
mut-rec.scm	145	0.166	56	0.065	99	0.677	93	0.647	56	0.128	56	0.132
infinite-2.scm	38	0.029	19	0.017	27	0.043	27	0.043	19	0.024	19	0.023
mj09.scm	902	2.340	72	0.088	115	1.234	233	5.849	72	0.196	72	0.194
count.scm	74	0.070	31	0.030	51	0.135	48	0.133	31	0.048	31	0.048
cpstak.scm	8598	30.490	518	1.227	180	4.106	364	23.825	52	0.134	52	0.133
letrec-begin.scm	16	0.011	16	0.011	16	0.017	16	0.015	16	0.016	16	0.018
loop2.scm	1815	5.493	387	0.819	289	12.639	627	68.965	136	0.813	136	0.811
kcfa2.scm	58523	336.684	145	0.173	72	0.377	1767	615.895	145	0.658	145	0.633
kcfa3.scm	273717	2087.824	247	0.372	89	0.730	2671	1831.256	247	1.944	247	1.928
rotate.scm	—	—	101	0.130	75	0.438	77	0.513	101	0.367	101	0.362
fact.scm	218	0.328	79	0.104	64	0.267	94	0.599	52	0.113	52	0.117
blur.scm	4457	19.973	191	0.325	259	8.642	1277	273.542	124	0.577	124	0.573
gcipd.scm	48665	240.061	190	0.321	396	18.265	730	70.235	130	0.606	130	0.607
widen.scm	217	0.307	85	0.120	69	0.296	85	0.515	55	0.131	55	0.136
inc.scm	52	0.054	28	0.023	24	0.041	26	0.049	28	0.036	28	0.036
infinite-1.scm	13	0.008	9	0.006	10	0.009	10	0.009	9	0.007	9	0.007
sq.scm	87	0.103	29	0.024	32	0.060	47	0.130	29	0.035	29	0.036
fib.scm	3261	10.494	673	1.632	112	0.891	182	2.833	98	0.351	98	0.356
church-2-num.scm	215	0.341	78	0.094	62	0.310	65	0.354	53	0.111	53	0.112
eta.scm	410	0.625	44	0.044	49	0.167	136	1.511	44	0.081	44	0.081

Table 7.1: Number of states in the state graph computed (#), and time taken to compute it (t) by the CESK machine.

7.3 PCESK Benchmarks

Similarly as for the CESK machine, benchmarks have been performed on the PCESK machine to inspect the influence of the refinements described in Section 4.4 on the produced state space and the processing time. The results are given in Figures 7.3 and 7.4 (the raw numbers are in Table 7.2) for $k = 0$. Similarly as for the CESK machine, the results for other values of k lead to the same conclusions. Note that for the example `deadlock.scm`, enabling state subsumption alone lead to an analysis that did not terminate in a reasonable amount of time.

The `nothing` case corresponds to a PCESK machine with abstract counting (Section 4.4.1), abstract thread counting (Section 4.4.2), and threads removed when they finish their execution (Section 4.4.3).

example	nothing		gc		sbfs		sdfs		gc+sbfs		gc+sdfs	
	#	t	#	t	#	t	#	t	#	t	#	t
pcounter1.scm	305	1.110	123	0.421	102	0.792	184	3.856	81	0.425	101	0.597
pcounter.scm	8845	112.298	19047	342.835	2578	475.292	5753	2678.342	3703	795.219	5580	1612.348
pcounter5-seq.scm	1768	11.934	588	3.253	512	26.599	1060	113.555	378	6.844	478	10.647
pcounter-mutex.scm	2705	26.656	2497	28.747	1984	335.670	1860	406.975	1435	94.544	1441	99.699
pcounter-race.scm	457	3.347	669	5.191	457	11.323	457	11.175	457	9.543	457	10.581
pcounter-buggy.scm	8845	110.355	19047	293.326	2578	475.819	5753	2413.889	3703	773.126	5580	1622.439
producer-consumer-seq.scm	796	3.724	678	3.517	354	15.736	769	81.864	678	20.393	678	20.137
deadlock1.scm	1530	8.451	331	1.399	387	9.008	262	4.959	276	2.837	263	2.665
deadlock1-release.scm	3679	22.588	664	3.233	867	40.346	627	24.784	547	9.879	534	9.577
deadlock.scm	234962	4258.006	7413	104.741	—	—	—	—	6049	1520.637	5911	1440.423
race-cas.scm	81	0.198	81	0.205	81	0.345	81	0.347	81	0.324	81	0.331
race-set-cas.scm	95	0.265	95	0.292	95	0.506	95	0.533	95	0.472	95	0.490
false-pos.scm	461	3.605	461	3.536	461	15.507	461	15.237	461	9.193	461	9.453
benign.scm	75	0.258	75	0.270	75	0.414	75	0.426	75	0.384	75	0.392

Table 7.2: Number of states in the state graph computed (#), and time taken to compute it (t) by the PCESK machine.

First of all, we can see that abstract garbage collection sometimes has a negative impact on the size of the state space for the PCESK machine, while for the CESK machine

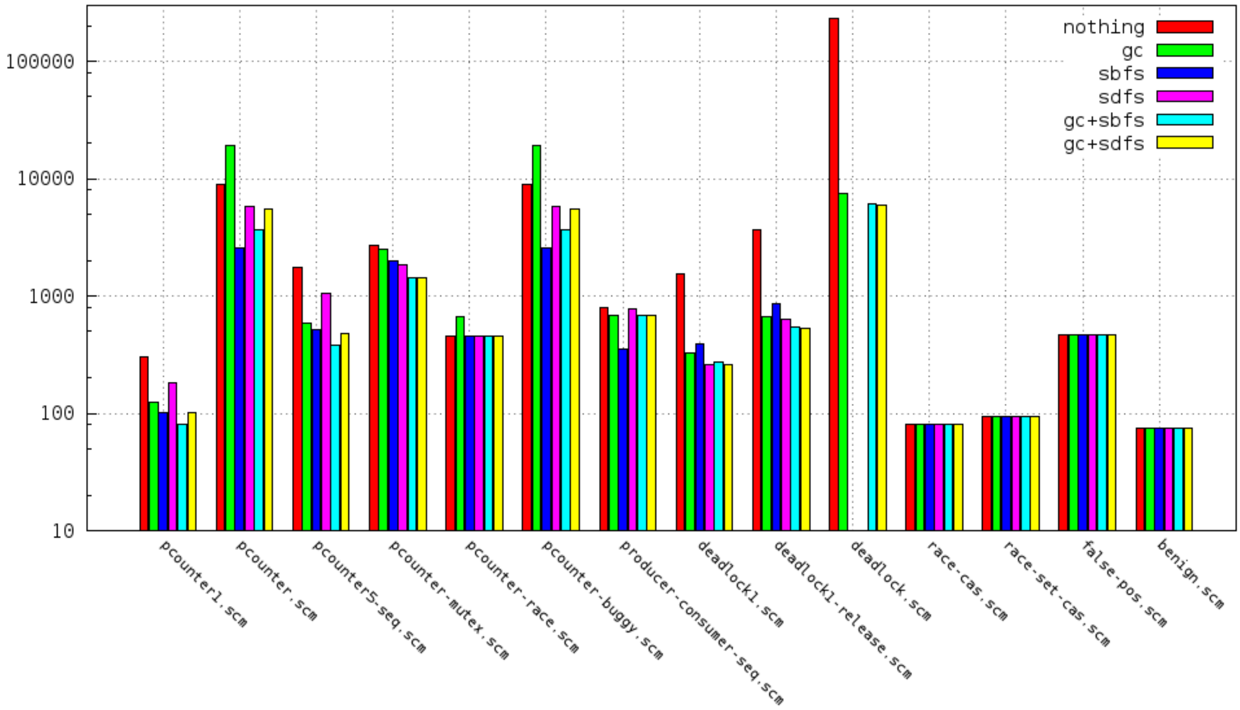

Figure 7.3: Number of states in the state graph computed by the PCESK machine.

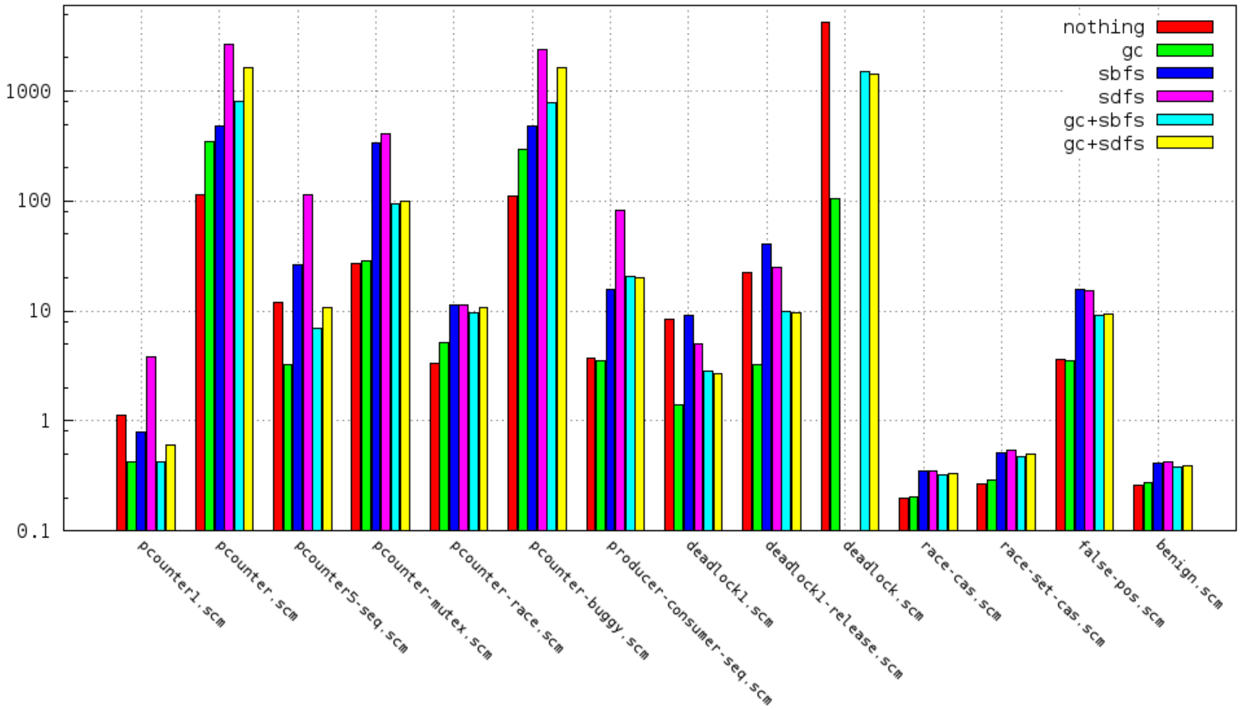

Figure 7.4: Time taken by the PCESK machine to compute the state graph.

it always either reduced its size or kept it the same. It seems that garbage collection sometimes prevent the possible merge of some states, who become different as garbage collection reclaims some addresses in the stores. Due to the high size of the state spaces involved, the reason for this higher size has not been further investigated.

We also see that, while it was a good idea to use both the garbage collection and the state subsumption (with either a DFS or a BFS) as default for the CESK machine, it is not the case anymore for the PCESK machine. The gain in the size of the state space might still be interesting in some cases (e.g. we go from 8887 states in `pcounter.scm` without any refinement, to 3709 by using abstract garbage collection and and state subsumption with a BFS), the price to pay in analysis time is high (in the same example, we go from 112s to 744s of analysis). In fact, the analysis time seems to generally be better when no refinement is used, and for the few cases where the time decreases with the refinement, this decrease is negligible (e.g. the `pcounter5-seq.scm` example goes from 11s to 3s when garbage collection is used).

Unlike the CESK machine, it seems to be better *not* to use garbage collection nor state subsumption on the PCESK machine. Should the analysis time be too high, garbage collection might be turned on as it may lead to big improvements in some cases, such as the deadlock examples in our benchmark.

7.4 Conclusion

In this chapter, we described our implementation of the PCESK machine and how it is related with the mathematical definition of this machine (described in Chapter 4). This fulfills the first objective of this thesis, which was to produce an implementation of the PCESK machine, as no other publicly available implementation currently exist.

We performed benchmarks on the CESK machine implementation to verify that it behaves as expected. Unsurprisingly, both the size of the state graph and the analysis time is greatly reduced by using garbage collection and state subsumption. Also, when combining state subsumption with garbage reduction, the exploration method (breadth-first or depth-first) does not matter. However, when used without garbage reduction, we found that state subsumption with a breadth-first exploration approach generally leads to smaller state graphs, which is the opposite result of [ZR12].

We also performed benchmarks on the PCESK machine implementation. We observed that it is no longer desirable to combine garbage collection and state subsumption by default, as it leads to a non-negligible increase in processing time for only a small decrease in the size of the state graph. A notable exception to this is the fact that examples involving deadlocks tend to gain from the use of garbage collection, both in size of the state graph and processing time.

Chapter 8

First-Class Locks: The PCESK$_L$ Machine

The ability to analyze programs using locks is important, because concurrent programs tend to use locks as synchronization mechanisms instead of lower-level constructs such as `cas`. In Chapter 6, we analyzed programs that use locks based on `cas`. Although using `cas` as the sole concurrency primitive in CScheme keeps the language small and simple, we found that the cost in terms of state space size and analysis time was high. As the `acquire` function is implemented as a loop over a `cas`, many states are added to the state graph. Eventually, with a few locks and a non-trivial use of `cas`-based locks, the state graph becomes practically incomputable.

In this chapter, we investigate an alternative approach. We use locks as the concurrency primitive instead of `cas`. We update the definition of the language, define the semantics of the lock-related expressions, update the race condition and deadlock analyses, and finally validate our approach. We conclude that using locks not only simplifies the analyses, but also reduces the size of the state space and the analysis time, allowing to analyze more complex programs that involve multiple locks.

8.1 The Language: CScheme$_L$

We update the language definition to use locks instead of `cas`. The updated language is called CScheme$_L$ (for *Concurrent Scheme with Locks*), and its grammar is given in Figure 8.1.

Two primitive values are added, corresponding to the possible states of a lock: `#locked` and `#unlocked`. We do not use booleans, as this might introduce imprecision because booleans support other operations that might lose precision information.

Two expressions are added to act on locks: `acquire` and `release`, which both take a lock as argument and either lock or unlock it.

Programs written with locks built on top of `cas` in CScheme are easily converted to CScheme$_L$ by initializing the locks to `#unlocked` instead of `#t`, and adapting the calls to the previously user-defined functions `acquire` and `release` to take a lock as argument.

> **Example 26** (Parallel counter implemented with first-class locks)**.** The program of Example 9, when expressed in the CScheme$_L$ language, becomes the following.

```
(letrec ((counter 0)
         (lock #unlocked)
         (inc (lambda ()
                (acquire lock)
                (set! counter (+ counter 1))
                (release lock)))
         (t1 (spawn (inc)))
         (t2 (spawn (inc))))
  (join t1)
  (join t2)
  counter)
```

$$
\begin{array}{rcl}
v \in \mathit{Var} & & \text{a set of identifiers} \\
n \in \mathbb{N} & & \text{a set of number literals} \\
b \in \mathbb{B} & ::= & \texttt{\#t} \mid \texttt{\#f} \\
l \in \mathbb{L} & ::= & \texttt{\#locked} \mid \texttt{\#unlocked} \\
e \in \mathit{Exp} & ::= & \textit{æ} \mid \mathit{cexp} \\
f, \textit{æ} \in \mathit{AExp} & ::= & \mathit{lam} \mid v \mid n \mid b \\
\mathit{lam} \in \mathit{Lam} & ::= & \texttt{(lambda (}v_1 \ldots v_n\texttt{)}\ e_1 \ldots e_n\texttt{)} \\
\mathit{cexp} \in \mathit{CExp} & ::= & \texttt{(}f\ e_1 \ldots e_n\texttt{)} \\
& \mid & \texttt{(begin }e_1 \ldots e_n\texttt{)} \\
& \mid & \texttt{(letrec ((}v_1\ e_1\texttt{)}\ \ldots\texttt{)}\ e_{body_1} \ldots e_{body_n}\texttt{)} \\
& \mid & \texttt{(if }e_{cond}\ e_{cons}\ e_{alt}\texttt{)} \\
& \mid & \texttt{(set! }v\ e\texttt{)} \\
& \mid & \texttt{(spawn }e\texttt{)} \\
& \mid & \texttt{(join }\textit{æ}\texttt{)} \\
& \mid & \texttt{(acquire }v\texttt{)} \\
& \mid & \texttt{(release }v\texttt{)}
\end{array}
$$

Figure 8.1: Grammar of CScheme$_L$.

8.2 Semantics of the PCESK$_L$ Machine

The semantics of the CScheme$_L$ (given by the PCESK$_L$ machine) are similar to those of the CScheme language (given by the PCESK machine). `spawn` and `join` are handled in exactly the same way in both machines. Similarly to `cas` in the the PCESK machine, the semantics of the `acquire` and `release` concurrency primitives are handled at the level of the underlying CESK machine. We only have to specify the sequential transition function for `acquire` and `release` to obtain the PCESK$_L$ machine.

8.2.1 Concrete Semantics

The transition rule for `acquire` can only make progress when the value of the lock is `#unlocked`. When this is the case, the value becomes `#locked` and the program can continue its execution. When this is not the case, other transition rules should be used to step other threads until the lock becomes unlocked.

$$\begin{aligned}\overbrace{\langle \texttt{(acquire } v\texttt{)}, \rho, \sigma, a, t\rangle}^{\varsigma} \to & \langle \texttt{\#t}, \rho, \sigma[\rho(v) \mapsto \texttt{\#locked}], a, u\rangle \\ & \text{if } \sigma(\rho(v)) = \texttt{\#unlocked} \\ & \text{where } u = \mathit{tick}(\varsigma)\end{aligned}$$

The transition function for `release` ensures that the lock is correctly unlocked before continuing the evaluation, but does not check whether the lock was previously locked or not. The same lock can thus be released more than once.

$$\begin{aligned}\overbrace{\langle \texttt{(release } v\texttt{)}, \rho, \sigma, a, t\rangle}^{\varsigma} \to & \langle \texttt{\#t}, \rho, \sigma[\rho(v) \mapsto \texttt{\#unlocked}], a, u\rangle \\ & \text{where } u = \mathit{tick}(\varsigma)\end{aligned}$$

8.2.2 Abstract Semantics

Remember that in the abstract machine, there might be more than one concrete value associated with an address in the store. In certain situations, we might not know whether a lock is locked or not. This is the case if two different locks, one locked and one unlocked, have been assigned to the same address.

One possible solution for this problem is to assume that there is only a finite number of locks created in the programs. When this is the case, we can assign a different address space for locks and ensure that every lock gets assigned a different address in this address space. The precise value of a lock will then always be known.

Under this assumption, we can give the following abstract transition rule for `acquire`, in which we can safely do a strong update on the value of the lock. This is because we know that the lock is the only value residing at this address, and it has a finite number of possible values.

$$\begin{aligned}\overbrace{\langle \texttt{(acquire } v\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle}^{\hat{\varsigma}} \to & \langle \texttt{\#t}, \hat{\rho}, \hat{\sigma}[\hat{\rho}(v) \mapsto \{\texttt{\#locked}\}], \hat{a}, \hat{u}\rangle \\ & \text{if } \hat{\sigma}(\hat{\rho}(v)) \sqcap \{\texttt{\#unlocked}\} \neq \bot \\ & \text{where } \hat{u} = \mathit{tick}(\hat{\varsigma})\end{aligned}$$

The abstract transition rule for `release` is a straightforward translation of the concrete one:

$$\begin{aligned}\overbrace{\langle \texttt{(release } v\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t}\rangle}^{\hat{\varsigma}} \to & \langle \texttt{\#t}, \hat{\rho}, \hat{\sigma}[\hat{\rho}(v) \mapsto \{\texttt{\#unlocked}\}], \hat{a}, \hat{u}\rangle \\ & \text{where } \hat{u} = \mathit{tick}(\hat{\varsigma}).\end{aligned}$$

This complete the definition of the PCESK$_L$ machine, which replaces the transition rule of the PCESK machine for `cas` by the two transition rules for `acquire` and `release`.

8.3 Adaptation of the Analyses for PCESK$_L$

The analyses defined in Chapter 5 were based on `cas` as the only concurrency primitive. We have to adapt some of these analyses to use locks instead.

8.3.1 Conflict Analysis

Thanks to our formulation of the conflict analysis using the relations *Read* and *Write*, the adaptation is trivial. Since the language does not have `cas` anymore, but just `set!` as a write expression, the definition of the *Write* relation becomes the following:

$$\mathit{Write}(\langle \texttt{(set! } v \ \_\texttt{)}, \hat{\rho}, \_, \_\rangle, \hat{\rho}(v))$$

This is the only change required to detect conflicts in the PCESK$_L$ machine. The definitions of *RWConflict*, *WWConflict* and *Conflict* stay the same as before.

8.3.2 Race Condition Analysis

As the CScheme$_L$ language does not include `cas`, no race condition can happen due to an unretried `cas`. Therefore, the unretried `cas` detection is not needed anymore. The race condition analysis reduces to the conflict analysis.

$$\mathit{RaceCondition}(e) \Leftrightarrow Conflict(e)$$

8.3.3 Deadlock Analysis

Instead of having to inspect potentially long paths in the state graph to detect a deadlock, deadlocks can now be found by looking at individual states and whether they have a successor or not. Indeed, if there exists a reachable state where every context in this state is currently evaluating either an `acquire` or a `join`, and if this state has no successor, we reached a deadlock. This analysis is formalized by the relation $\mathit{Deadlock}_L \subset \mathit{Exp}$:

$$\begin{aligned}\mathit{Deadlock}_L(e) \Leftrightarrow & (V, E) = \widehat{geval}(e), \exists \langle \hat{T}_1, \hat{\sigma}_1 \rangle \in V \wedge \\ & \nexists (\langle \hat{T}_1, \hat{\sigma}_1 \rangle, \langle \hat{T}_2, \hat{\sigma}_2 \rangle) \in E \wedge \\ & \forall \hat{c} \in \mathit{range}(\hat{T}_1), \\ & (\hat{c} = \langle \texttt{(acquire \_)}, \_, \_, \_ \rangle \vee \hat{c} = \langle \texttt{(join \_)}, \_, \_, \_ \rangle).\end{aligned}$$

The `cas`-based deadlock analysis was unable to detect deadlocks that involved a loss of precision on the value of the variable written to by `cas`. Here, since there cannot be a loss of precision on the value of locks, and locks are the only values that can be used to do synchronization between threads (previously, any value could be used with `cas`), we do not have this problem anymore. Every deadlock should be detected by this analysis.

Another problem of the `cas`-based deadlock analysis was that we did not check whether multiple `join`s were actually blocked by a deadlocked thread (e.g. thread 1 is blocked and thread 2 and 3 joins on thread 1) . If we had a `cas` that contained a cycle to itself, and that all the other contexts in this state were `join`s, we considered this state to be a deadlock, while it could be that the `join`s did not block each other. Here, it is not necessary to check this, since we verify that the state of the deadlock does not have a successor. If there are multiple `join`s but no deadlock, a successor state will exist and no deadlock will be detected.

Because we have locks as first-class member of the language, deadlocks are easier to find in the state graph since they actually represent a *dead* branch of the graph, while we had to look for multiple states in our previous analysis.

8.4 PCESK$_L$ Benchmarks

Benchmarks have been done for the PCESK$_L$ machine on similar examples as for the PCESK machine, rewritten with first-class locks. The results are given in Figures 8.4 and 8.5 (the raw numbers are in Table 8.1), for $k = 0$.

A striking difference with the PCESK machine is the gain of about an order of magnitude in both the size of the state space and the analysis time. This can be seen on Figures 8.2 and 8.3 which compare the size of the state space and the analysis time for the PCESK and PCESK$_L$ machines on equivalent examples. The *PCESK best* bars correspond to the best size and the best time for the PCESK machine, and are generally not for an execution with the same improvements (an execution with abstract garbage collection might produce a smaller state space but take more time to produce it). The only example where the PCESK machine performs similarly as the PCESK$_L$ machine is

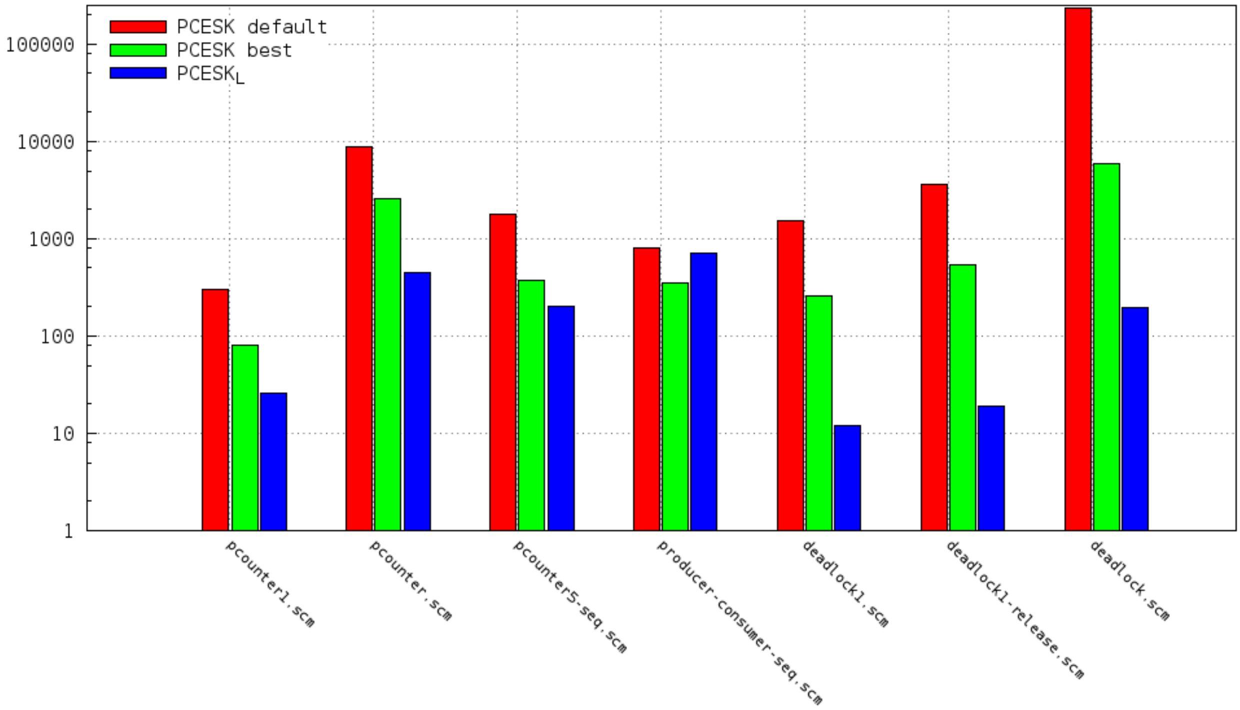

Figure 8.2: Comparison between the number of states in the state graph computed by the PCESK and the PCESK$_L$ machines.

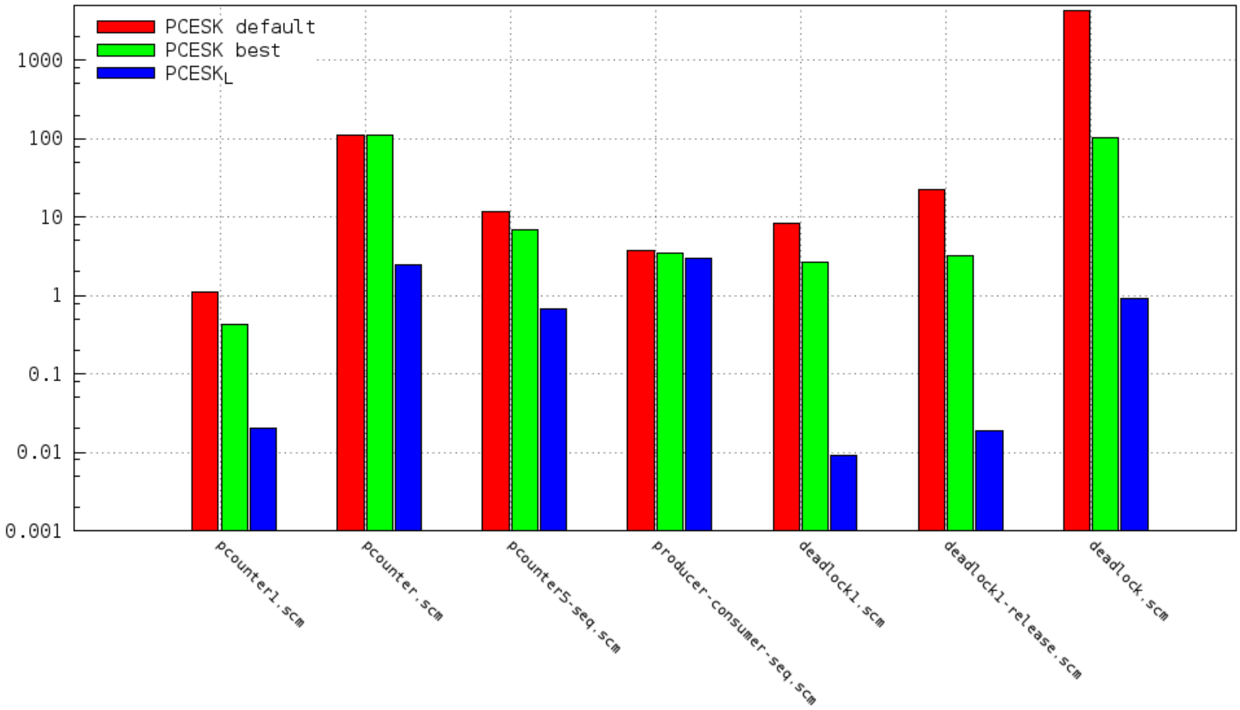

Figure 8.3: Comparison between the time to compute the state graph for the PCESK and the PCESK$_L$ machines.

the `producer-consumer-seq.scm` example. This can be explained by the fact that this example contains two threads that run one after the other (hence the `-seq`) and thus does not introduce many interleavings (just between one of the thread and the main thread).

However, the `pcounter5-seq.scm` example also uses threads that run one after the other, and the difference between the PCESK and PCESK$_L$ machines is higher. This difference might be explained by the fact that this example uses 5 threads, which introduce more interleavings with the main thread.

The `deadlock.scm` example has 7407 states with the PCESK machine, and took 104 seconds to analyze (with garbage collection enabled), while it only has 199 states, for less than 1 second (0.918s) with the PCESK$_L$ machine. This is 37 times less states and 113 times less analysis time.

This improvement follows from the fact that the PCESK$_L$ machine is better suited to detect deadlocks since it has first-class locks, but we also see the improvements for programs not involving deadlocks, such as the `pcounter.scm`, which was implemented with `cas` in the PCESK machine. With the PCESK machine, at its best (state subsumption, BFS exploration) it had 2578 states and took 475 seconds to analyze. With the PCESK$_L$ machine, it has 443 states and takes 2.48s to analyze. This is more than 5 times less states and 191 times less analysis time.

Thanks to this improvement in performance, the PCESK$_L$ machine is able to analyze more complex programs: the `pcounter3.scm` example is the parallel counter with three threads, the `deadlock3.scm` example contains a deadlock involving three locks and three threads and the `incdec.scm` example is Example 13 rewritten with locks. Those three examples were not practically analyzable by the PCESK machine but are by the PCESK$_L$ machine.

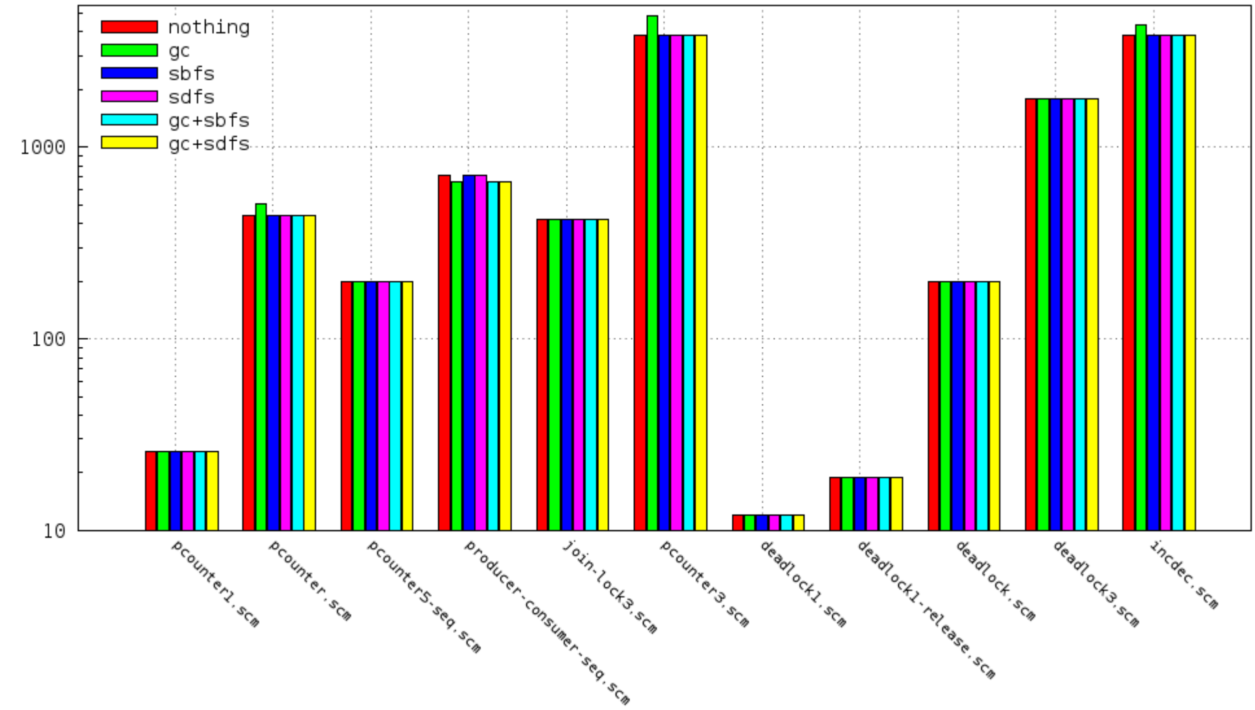

Figure 8.4: Number of states in the state graph computed by the PCESK$_L$ machine.

On Figures 8.4 and 8.5, we can see that both garbage collection and state subsumption do not improve the size of the state space anymore, and tend to increase the analysis time. It thus seems that for the PCESK$_L$ machine, a reasonable default is to turn off those improvements, whereas they were still useful in some specific cases for the PCESK machine.

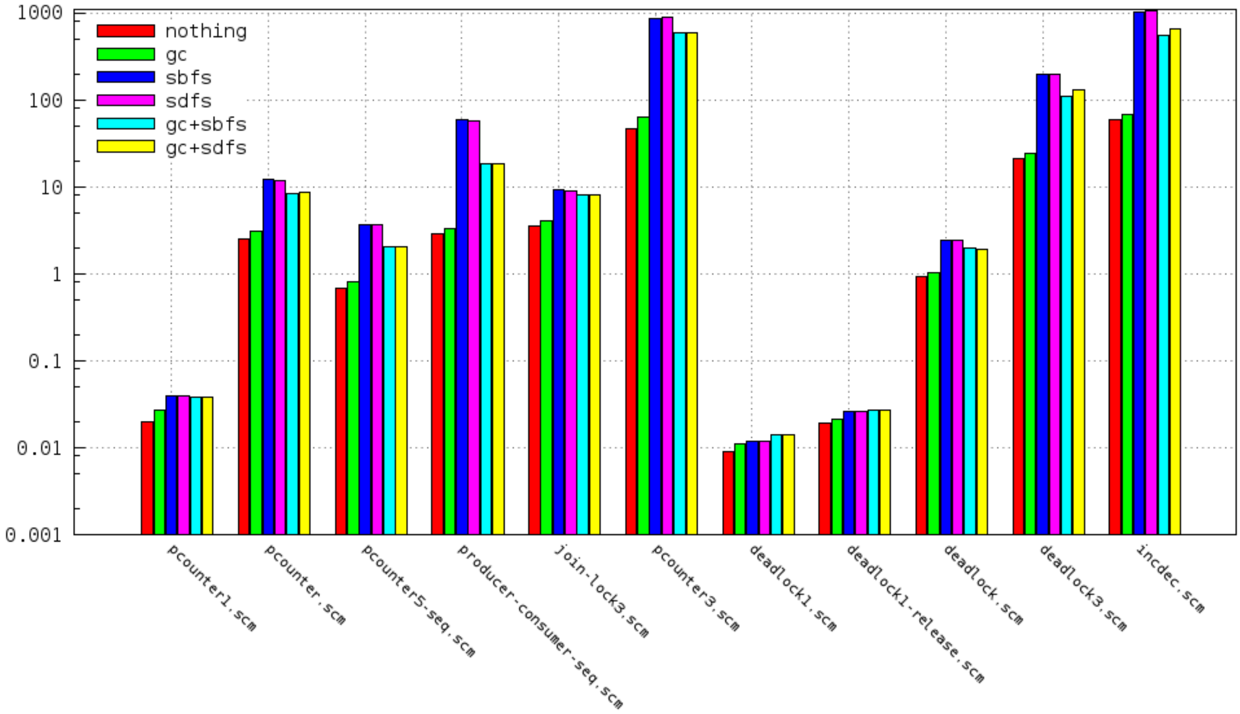

Figure 8.5: Time taken by the PCESK$_L$ machine to compute the state graph.

example	nothing		gc		sbfs		sdfs		gc+sbfs		gc+sdfs	
	#	t	#	t	#	t	#	t	#	t	#	t
pcounter1.scm	26	0.020	26	0.027	26	0.039	26	0.039	26	0.038	26	0.038
pcounter.scm	443	2.480	507	3.138	443	12.306	443	12.006	443	8.414	443	8.669
pcounter5-seq.scm	200	0.682	200	0.817	200	3.619	200	3.670	200	2.040	200	2.041
producer-consumer-seq.scm	710	2.937	662	3.346	710	59.347	710	58.382	662	18.666	662	18.440
join-lock3.scm	422	3.564	422	4.073	422	9.363	422	9.118	422	8.032	422	8.174
pcounter3.scm	3827	47.282	4827	64.053	3827	873.849	3827	899.201	3827	597.058	3827	602.982
deadlock1.scm	12	0.009	12	0.011	12	0.012	12	0.012	12	0.014	12	0.014
deadlock1-release.scm	19	0.019	19	0.021	19	0.026	19	0.026	19	0.027	19	0.027
deadlock.scm	199	0.918	199	1.017	199	2.461	199	2.450	199	1.955	199	1.927
deadlock3.scm	1793	21.404	1793	24.609	1793	197.430	1793	198.929	1793	111.376	1793	130.022
incdec.scm	3829	59.940	4349	68.612	3829	1043.559	3829	1053.622	3829	561.269	3829	648.868

Table 8.1: Number of states in the state graph computed (#), and time taken to compute it (t) by the PCESK$_L$ machine.

8.5 Validation of the Adapted Analyses

We show in this section that the analyses we built on top of the PCESK$_L$ machine work as expected. To use the PCESK$_L$ machine instead of the PCESK machine in the implementation, the flag `-l` is used.

8.5.1 Race Condition Analysis

The race condition analysis of the implementation is in fact exactly the same as for the PCESK machine. It will detect conflicts involving `set!` and `cas`, and since no `cas` should be present in the input program, it is the same as using the `setconflict` target (unretried `cas` will also be looked at, but for the same reason, nothing will be found).

We can verify that the locks correctly guard the critical sections by using this analysis on correct programs. For the examples containing race conditions due to `set!` conflicts, since the analysis is the same as before, they are still detected. We run the analysis on the following programs. The results are shown in Table 8.2.

- `pcounter.scm`: Example 26, page 76, is the parallel counter example involving two threads, written with first-class locks.
- `pcounter3.scm`: similar to `pcounter.scm`, but involving three threads.
- `incdec.scm`: similar to Example 13, page 47, involving three threads either increasing or decreasing the value of a shared lock, rewritten to use first-class locks instead of `cas`.
- `pcounter-race.scm`: Example 12, page 46, containing a race condition due to a critical section reachable by multiple threads.

Example	Length	Threads	Locks	Expected	`conflicts`		
					found	tp	fp
`pcounter.scm`	32	2	1	0	0	0	0
`pcounter3.scm`	38	3	1	0	0	0	0
`incdec.scm`	52	3	1	0	0	0	0
`pcounter-race.scm`	24	2	0	2	2	2	0
				Precision	100%		
				Recall	100%		
				Accuracy	100%		

Table 8.2: Results of the race condition analysis on the PCESK$_L$ machine.

Because the conflict analysis remains mostly the same as in the PCESK machine, it is expected that it still correctly detects errors such as in `pcounter-race.scm`, while correctly detecting nothing in correct examples. The main difference with the `cas`-based approach is that we are now able to analyze bigger programs, such as `pcounter3.scm` and `incdec.scm`, whose CScheme equivalents were not analyzable in a reasonable amount of time.

8.5.2 Deadlock Analysis

The target for the deadlock analysis of the PCESK$_L$ machine is the target `ldeadlocks`. We run this analysis on the following examples. The results are given in Table 8.3.

- `deadlock1.scm`: Example 27, page 83, containing a deadlock involving one thread and one (initially locked) lock.
- `deadlock.scm`: Example 28, page 84, containing a deadlock involving two threads and two locks.
- `deadlock3.scm`: Example 29, page 84, containing a deadlock involving three threads and three locks.
- `pcounter.scm`, `pcounter3.scm`, `incdec.scm`, examples from the previous section, not containing any deadlocks.

Example 27 (Deadlock involving one lock and two threads)**.** This program is similar as the one in Example 23, as it has one deadlock created by two threads trying to acquire the same lock.

```
(letrec ((lock #unlocked)
         (t1 (spawn (acquire lock))))
  (acquire lock)
  (join t1))
```

Example 28 (Deadlock involving two locks and two threads)**.** This program is the similar as the one in Example 24. Two threads try to acquire two locks in a different order, creating a deadlock.

```
(letrec ((a #unlocked)
         (b #unlocked)
         (t1 (spawn (begin
                      (acquire a)
                      (acquire b)
                      (release b)
                      (release a))))
         (t2 (spawn (begin
                      (acquire b)
                      (acquire a)
                      (release a)
                      (release b)))))
  (join t1)
  (join t2))
```

Example 29 (Deadlock involving three locks and three threads)**.** This example contains a possible deadlock as there is a possible dependency cycle between the locks. If thread `t1` acquires lock `a`, followed by thread `t2` acquiring lock `b`, and thread `t3` acquiring lock `c` and then trying to acquire lock `a`, there will be a deadlock. Indeed, thread `t1` is blocked as it cannot acquire lock `b` (held by thread `t2`), thread `t2` is blocked as lock `c` is held by thread `t3`, thread `t3` is blocked as lock `a` is held by thread `t1`, and the main thread is blocked as it waits thread `t1`.

```
(letrec ((a #unlocked)
         (b #unlocked)
         (c #unlocked)
         (t1 (spawn (begin
                      (acquire a)
                      (acquire b)
                      (release b)
                      (release a))))
         (t2 (spawn (begin
                      (acquire b)
                      (acquire c)
                      (release c)
                      (release b))))
         (t3 (spawn (begin
                      (acquire c)
                      (acquire a)
                      (release a)
                      (release c)))))
  (join t1)
  (join t2)
  (join t3))
```

As expected, the deadlocks are detected, and no deadlock is detected on the examples not containing any. Also, this analysis is able to detect deadlocks on programs larger than those analyzable by the PCESK machine.

Example	Length	Threads	Locks	Expected	`ldeadlocks`		
					found	tp	fp
`deadlock1.scm`	13	0	1	2	2	2	0
`deadlock.scm`	39	2	2	1	1	1	0
`deadlock3.scm`	58	3	3	1	1	1	0
`deadlock1-release.scm`	16	0	1	1	1	1	0
`pcounter.scm`	32	2	1	0	0	0	0
`pcounter3.scm`	38	3	1	0	0	0	0
`incdec.scm`	52	3	1	0	0	0	0
				Precision	100%		
				Recall	100%		
				Accuracy	100%		

Table 8.3: Results of the deadlock analysis on the PCESK$_L$ machine.

8.6 Conclusion

In this chapter, we addressed the precision problem of our `cas`-based deadlock analysis. Values on which a `cas` was performed could become too abstract, either by living at the same address as another value or as the result of a complex operation on the value. This prevented the analyses to distinguish between a `cas` that will always fail from a correctly used `cas`.

The PCESK$_L$ machine solves this problem by replacing `cas` with first-class locks, managed by `acquire` and `release`. We defined the semantics of the PCESK$_L$ machine and adapted both the race condition analysis and the deadlock analysis. The formulation of the race condition analysis is simpler than in the `cas`-based approach, because the only write operation is `set!`, and there is no need to filter out some conflicts that appear in correct uses of `cas`. The deadlock analysis is also simpler, and does not have the precision problem of the `cas`-based approach, so we do not need to throw away possible deadlocks.

Our benchmarks show that both the size of the state graph and the analysis time are greatly reduced in the PCESK$_L$ machine, compared to the PCESK machine. This allows us to analyze larger programs that were not analyzable in a reasonable amount of time with the PCESK machine. Finally, we showed that both the race condition analysis and the deadlock analysis compute the expected results.

Chapter 9

Conclusion

In this dissertation, we explored the possibility of analyzing concurrency constructs in higher-order programs. Being able to perform such analyses is important, as concurrent programs tend to contain bugs that are particularly hard to find. We took the abstract interpretation approach, basing our work on Might and Van Horn's PCESK machine. We described this machine along with some improvements. Next, we improved the original May-Happen-in-Parallel analysis for the PCESK machine, so that it detected fewer false positives. This MHP analysis served as the basis to build a conflict analysis, which was able to detect read/write and write/write conflicts in higher-order programs. We then presented an unretried `cas` analysis, as unretried `cas` may introduce race conditions in programs. Together, the conflict analysis and the unretried `cas` analysis form a race condition analysis. We also defined a deadlock analysis, which has some limitations due to the inevitable loss of precision in the PCESK machine.

We conclude from our experiments that using `cas` as a synchronization primitive makes the analyses complex, and that the running times of the analyses are high. Therefore, we reviewed the language we analyze to use first-class locks instead of `cas`, which leads to improvements in both the definition of the analyses and in the running time of the analyses. The race condition analysis becomes simpler, as it only has to take `set!` into account as a write operation. The lock-based deadlock analysis has fewer problems than the `cas`-based variant, mainly because we avoid losing precision on lock values, which was not the case for the general values on which `cas` acts. Having a shorter analysis time allows us to analyze larger and more complex programs that were previously not practically analyzable by the original PCESK machine.

9.1 Limitations of the Approach

The main limitation of the approach used in this work is the size of the state space that grows with the size of the analyzed program and the number of threads. Using abstract garbage collection and state subsumption on the CESK machine leads to good improvements, but Chapter 7 shows that this is no longer the case on the PCESK machine. Indeed, the potential decrease in state space size requires a non-negligible increase in analysis time. This is even more true when we use first-class locks, as there is no longer any decrease in state space by using abstract garbage collection and state subsumption (Section 8.4).

In order to analyze programs that are longer than a few tens of lines of code, it becomes indispensable to have a mechanism to reduce the total number of states. The same problem arises with model-checking, and we could take inspiration from techniques used in model-checking to reduce the size of the state space for concurrent programs, such as partial-order reduction [GvLH$^+$96]. In fact, our implementation supports a naive version of cartesian partial-order reduction [GFYS07], but we chose not to discuss it in this work because it often increases the number of false negatives of our analyses. Indeed, to reduce the size of the state space, it needs to drop information about the possible interleavings, making analyses based on the May-Happen-in-Parallel analysis more unsound. More work can be done to adapt these state space reduction techniques to abstract interpretation.

Two open question remain after our validation of the analyses on various examples. First, enabling garbage collection can lead to an *increase* of the size of the state space in the PCESK machine, as described in Section 7.3. This seems to be because two previously equivalent states can become different if the garbage collector collects more garbage in one state than the other. Further investigation is needed to find out why the garbage collector does not reclaim the same addresses in two states that were previously equivalent. The second question comes from the validation of the deadlock analysis on the PCESK machine (Section 6.5). In the `deadlock.scm` example, two false positives are detected. Due to the size of the state space, it is not feasible to investigate by hand to find the reason of those false positives. We found no simpler example that still exhibits these false positives. Knowing why these false positives occur could improve the precision of the deadlock analysis.

9.2 Future Work

In this dissertation, we define analyses for race condition and deadlock detection, but concurrent programs might also contain *livelocks*. Because a program still executes different expressions during a livelock, defining an analysis to detect them in the PCESK machine would probably have similar limitations as the deadlock analysis. Since a cycle corresponding to a livelock will contain branches with successful lock acquirements, it seems that a program with a livelock will not be distinguishable from a program correctly acquiring a lock.

Our deadlock analysis on the PCESK$_L$ machine makes the assumption that the program we analyze contains a finite number of locks, in order to avoid dealing with the possible loss of precision in the store addresses. A way to relax this assumption would be to do a strong update on the lock value and creating two branches in the state graph when accessing a lock whose value has been abstracted. Each branch would correspond to the path taken by the program if the lock was either already locked, or not locked. If the address space of lock values is separated from the other addresses, this approach should correctly approximate the behavior of the program.

The $\widehat{newtid}$ abstract thread allocation mechanism we used makes the assumption that only a finite number of threads are spawned. Might and Van Horn [MVH11] describe other approaches, but we did not include those in our implementation, as the increasing counter allocation scheme did not cause any problems in our examples. More work could be done to investigate whether this scheme is sufficient in the general case, or if other approaches should be used.

While our deadlock analysis on the PCESK machine is less powerful than the one on the PCESK$_L$ machine, it is still a useful analysis, even if there is room for improvement. Improving it would require finding a solution to avoid loss of precision on addresses used inside `cas` expressions in the program, and on the values stored in those addresses.

We demonstrated that changing the synchronization primitive from `cas` to locks both simplifies the definition of the analyses, and improves their running time. Therefore, it could be worth investigating other, higher-level synchronization primitives, such as monitors. Also, adapting the PCESK machine to concurrency models different from the shared-state thread view are worth investigating, for example to detect deadlocks in programs using actor-based [HBS73] or CSP-based [Hoa85] paradigms.

Finally, while our race condition analysis works as expected on the examples we presented, we did not prove that the analysis is sound. It should be tested on more examples to better understand its limitations, or be proven sound if it does not seem possible to find an example that defeats it.

9.3 Summary

In this dissertation we show that it is possible to automatically detect race conditions and deadlocks in higher-order programs through abstract interpretation. Other approaches are described in related work (Sections 2.3 and 2.4), although none use abstract interpretation to detect concurrency bugs. We based our work on Might and Van Horn's PCESK machine, which uses `cas` as the sole synchronization primitive. We show that it is possible to reason about race conditions and deadlocks on this machine, but that there is a precision cost forcing the analyses to be unsound to avoid detecting too many false positives. We describe a solution to this problem by introducing PCESK$_L$ machine, which uses locks as the only concurrency primitive. This machine not only makes the concurrency analyses more simple to formalize, it also produces more precise results in less time than was the case with the PCESK machine. The PCESK$_L$ machine also represents a step forward in the complexity of programs our analyses can handle.

Bibliography

[AB01] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 68–75. IEEE, 2001.

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for safe locking: Static race detection for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[AFMG12] Elvira Albert, Antonio E Flores-Montoya, and Samir Genaim. Analysis of may-happen-in-parallel in concurrent objects. In *Formal Techniques for Distributed Systems*, pages 35–51. Springer, 2012.

[APM$^+$07] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 805–806. ACM, 2007.

[Bir04] Andrew Birrell. Implementing condition variables with semaphores. *Computer Systems*, pages 29–37, 2004.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Notices*, volume 37, pages 211–230. ACM, 2002.

[Boy04] Chandrasekhar Boyapati. *Safejava: a unified type system for safe programming.* PhD thesis, Massachusetts Institute of Technology, 2004.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *ACM SIGPLAN Notices*, volume 36, pages 56–69. ACM, 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CDH$^+$00] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Pasareanu, Hongjun Zheng, et al. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[FA99] Cormac Flanagan and Martín Abadi. Types for safe locking. In *Programming Languages and Systems*, pages 91–108. Springer, 1999.

[FF00] Cormac Flanagan and Stephen N Freund. Type-based race detection for java. In *ACM SIGPLAN Notices*, volume 35, pages 219–232. ACM, 2000.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Notices*, volume 38, pages 338–349. ACM, 2003.

[Fra04] Keir Fraser. *Practical lock-freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[FS07] Lars-Åke Fredlund and Hans Svensson. Mcerlang: a model checker for a distributed functional programming language. In *ACM SIGPLAN Notices*, volume 42, pages 125–136. ACM, 2007.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. *Model Checking Software*, page 95, 2007.

[GvLH$^+$96] Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[HD01] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In *CONCUR 2001—Concurrency Theory*, pages 39–58. Springer, 2001.

[Hoa85] Charles Antony Richard Hoare. *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[Hol04] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[HP04a] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[HP04b] David Hovemeyer and William Pugh. Finding concurrency bugs in java. In *Proc. of PODC*, volume 4, 2004.

[Jag95] Suresh Jagannathan. Locality abstractions for parallel and distributed computing. In *Theory and Practice of Parallel Programming*, pages 320–345. Springer, 1995.

[JW94] Suresh Jagannathan and Stephen Weeks. Analyzing stores and references in a parallel symbolic language. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 294–305. ACM, 1994.

[Mig10] Matthew Might. Tutorial: Small-step cfa. 2010.

[Mig11] Matthew Might. Abstract interpreters for free. In *Static Analysis*, pages 407–421. Springer, 2011.

[MS06] Matthew Might and Olin Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. *ACM SIGPLAN Notices*, 41(9):13–25, 2006.

[MVH11] Matthew Might and David Van Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. In *Static Analysis*, pages 180–197. Springer, 2011.

[NA07] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. *ACM SIGPLAN Notices*, 42(1):327–338, 2007.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

[NPSG09] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, pages 386–396. IEEE Computer Society, 2009.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Shi91] Olin Shivers. *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, 1991.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[Tur08] Franklyn Turbak. *Design concepts in programming languages.* MIT press, 2008.

[Ur08] Shmuel Ur. Testing and debugging concurrent software. 2008.

[VHM10] David Van Horn and Matthew Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

[vP04] Christoph von Praun. *Detecting synchronization defects in multi-threaded object-oriented programs.* PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[WJP94] Stephen Weeks, Suresh Jagannathan, and James Philbin. A concurrent abstract interpreter. *Lisp and Symbolic Computation*, 7(2-3):173–193, 1994.

[WTE05] Amy Williams, William Thies, and Michael D Ernst. Static deadlock detection for java libraries. In *ECOOP 2005-Object-Oriented Programming*, pages 602–629. Springer, 2005.

[ZR12] Eduardo Zambon and Arend Rensink. Graph subsumption in abstract state space exploration. In *GRAPHITE*, pages 35–49, 2012.