A General Method for Rendering Static Analyses for Diverse Concurrency Models Modular

Quentin Stiévenart[a], Jens Nicolay[a], Wolfgang De Meuter[a], Coen De Roover[a]

[a]*Software Languages Lab, Vrije Universiteit Brussel, Belgium,*
{qstieven,jnicolay,wdmeuter,cderoove}@vub.ac.be

Abstract

Shared-memory multi-threading and the actor model both share the notion of processes featuring communication, respectively by modifying shared state and by sending messages. Existing static analyses for concurrent programs either model every possible process interleavings and therefore suffer from the state explosion problem, or feature modularity but lack in precision or in their support for dynamic processes. In this paper we present a general method for obtaining a scalable analysis of concurrent programs featuring dynamic process creation. Our ModConc method transforms an abstract concurrent semantics modeling processes and communication into a modular static analysis treating the behavior of processes separately from their communication. We present ModConc in a generic way and demonstrate its applicability by instantiating it for multi-threaded and actor-based programs. The resulting analyses are evaluated in terms of precision, performance, scalability, and soundness. While a typical non-modular static analysis time out on half of our 56 benchmarks with a 30 minutes timeout, analyses resulting from the application of ModConc can analyze all of them in less than 30 seconds, while remaining on par in terms of precision. Analyzing concurrent processes in isolation while modeling their communications is the key ingredient in supporting scalable analysis of concurrent programs featuring dynamic process communication.

Keywords: static analysis, modular analysis, concurrency, actors, threads
2000 MSC: 68N15, 68N19, 68N30

1. Introduction

In most concurrent programming models, programs consist of entities called *processes* that run concurrently to each other and that interfere through *communication effects*. Interprocess communication effects can be accesses and modifications to shared variables in thread models, or messages exchanged between different actors in the actor model [4, 45]. At run time, a single process in a concurrent program may create an unbounded number of additional processes. This combination of process creation and communication effects in concurrent programs results in highly dynamic control flow and data flow.

Static analyses for concurrent programs have been proposed, and are discussed in details in Section 8. Most of these existing analyses either explicitly take into account every possible interleaving of concurrent executions at points where interprocess communication occurs, rendering them non-scalable as they are subject to the state explosion problem [94], or are modular but limited in one of the following important properties: automation, precision, or support for programs in which the set of processes evolves dynamically. Performing static analysis of concurrent programs featuring unbounded processes in an automated, scalable, and precise way is therefore still an open problem, which we tackle in this paper.

Scalability of a static analysis can be achieved by *modularizing* it according to the general framework proposed by Cousot and Cousot [21]. A modular analysis treats the behavior of *components* (in our case, processes) separately from their *interferences* (in our case, communication). Modular static analysis has been explored in the context of shared-memory concurrency by Miné et al. [71] and for synchronous sequential processes by Midtgaard et al. [66]. However, these and similar analyses are limited to programs with a known and fixed number of processes and therefore do not support analyzing programs where processes can be created dynamically.

In a modular analysis, the analysis of a component can trigger other components for re-analysis. This violates the notion of *compositionality* of an analysis, in which the results of the analysis for a whole program are the composition of the results of the analysis of each component. In a modular analysis, the results of the analysis follow from a fixed point obtained in the analysis of each component, having taken other components that interfere with it into account. In the case of concurrent programs, a thread that accesses a variable that is modified by another thread has to be reconsidered for analysis, as is one actor to which another actor may send a message.

This paper proposes ModConc, an approach for the modular static analysis of concurrent programs with unbounded dynamic process creation. This sets it apart from the aforementioned modular analyses for concurrent programs [71, 66], which require a static process topology. We design ModConc as a general technique to design scalable modular analyses of concurrent programs based on their concurrent semantics. We demonstrate the use of ModConc to render an AAM-style analysis [50] of concurrent programs scalable. AAM is a well-studied abstract interpretation method, and is used here to analyze the behavior of a single process in a flow-insensitive and context-insensitive manner. We demonstrate our approach on two concurrency models supporting dynamic process creation: threads and actors. Note that the choice of AAM and of the sensitivities is made only to demonstrate the application of ModConc, and that ModConc is not limited to such analyses.

The core insight behind ModConc is that the dynamic behavior of a process is entirely defined by its code and its communication effects. We therefore construct an *intra-process analysis* that analyzes a single process in isolation to infer the processes created and communication effects generated by this process under a given set of input conditions. The intra-process analysis is based on a modified version of the program semantics, replacing concurrent operations by operations that denote the corresponding generated effects but otherwise do not modify the analysis state of other processes in any way. The information obtained from the intra-process analyses is subsequently used by an *inter-process analysis* to compute the set of processes that interfered with the analyzed processes and therefore require (additional) intra-process analysis. When no new interprocess communication effects can be discovered, a sound over-approximation of the behavior of all processes in the program is obtained. The result is a modular —in the sense of Cousot and Cousot [21]— whole-program analysis for concurrent programs that infers the set of all running processes and their communication effects in a sound and scalable manner.

A ModConc analysis is capable of inferring properties of concurrent programs that form the foundation of tool support for addressing pressing problems in software engineering such as program comprehension, bug detection and program verification. These inferred properties concern the processes created and their communication effects in addition to the traditional data and control flow properties computed by analyses for sequential programs. Our evaluation demonstrates that modular analyses designed with ModConc

scale linearly with both the number of abstract processes created and the number of communication effects. The analyses do not suffer from the state explosion problem and are therefore able to analyze concurrent programs from a benchmark suite that consists of the actor-based programs from the well-known Savina benchmark suite [56] and their shared-memory multi-threaded equivalent, in a matter of seconds.

To summarize, the contributions of this paper are the following.

- An extension of the framework by Cousot and Cousot of modular analysis [21] to concurrent programs with dynamic process creation.
- The application of this extension to both thread-based and actor-based concurrency.
- The formalization, empirical validation, and discussion of termination, soundness, and complexity of the approach on an analysis for thread-based and actor-based concurrency.
- The construction of a benchmark suite composed of programs exposing dynamic creation of threads in a shared-memory concurrency setting, similar to the Savina benchmark suite for actor programs.

2. Context: Models of Concurrency and Their Dynamic Behavior

Most concurrency models share the concepts of *processes* and *communication*. A *process* is a unit of computation isolated from other processes, except for interferences (communications) that can occur between processes. In the thread model, a thread is a process and communication happens through shared variables, locks, and thread joining. In the actor programming paradigm, an actor is a process and communication happens through the exchange of messages.

Communication effects may trigger new computations, influence already-running computations, or be used for synchronization. Process creation is one of many possible communication effects, and an inherent part of concurrent models is the ability of a running process to dynamically create an unbounded number of new processes. Unbounded dynamic process creation is often ignored by existing static analyses for concurrent programs (see Section 8), which can only handle a fixed set of processes.

2.1. Dynamic Concurrent Example: factorial

We explore dynamic concurrency through a concurrent divide-and-conquer approach to a factorial computation. Note that for our examples we use a Scheme-like language, of which we define the core semantics in the next section. Because of the commutativity of multiplication, computation of a factorial can easily be divided in smaller chunks of work. Consider the computation of $100! = \Pi_1^{100} i$, and how it can be split into sub-computations as represented in Fig. 1. We will represent each of these sub-computations as a process in the following thread-based and actor-based implementations.

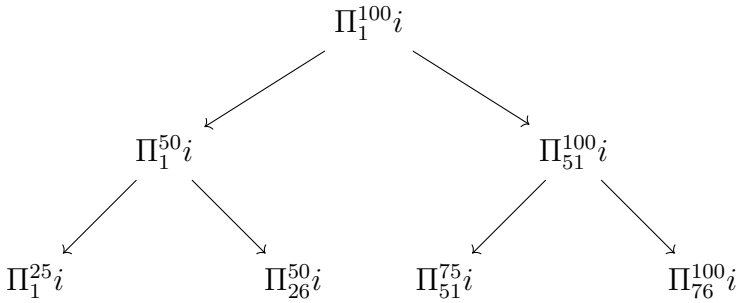


Figure 1: Representation of a concurrent computation of a factorial.

Thread-based approach. The program in Listing 1 implements every node of the computation tree as a thread. Every thread performs a recursive call to `fact-thread` that either splits the node in two subtrees to perform the concurrent computation and to multiply the results from the two subtrees, or computes the partial product directly. $\Pi_a^b i$ is implemented as `(fact-thread a b)`.

Function `split` splits an interval in multiple intervals, for example `(split 1 100)` may evaluate to `'((1 . 50) (51 . 100))`. Global variable `FragmentSize` specifies the cut-off factor (expressed as the size of an interval) at or below which an interval is computed sequentially rather than concurrently. Function `product` sequentially computes the product of integers in a given interval, i.e., `(product a b)` computes $\Pi_a^b i$ sequentially.

```
1 (define (fact-thread from to)
2   (if (<= (- to from) FragmentSize)
3       (product from to)
4       (let ((steps (split from to)))
5         (foldl * 1
6                (map (lambda (t) (join t))
7                     (map (lambda (bounds)
8                            (spawn
```

```
 9                              (fact-thread (car bounds)
10                                           (cdr bounds))))
11                          steps))))))
12 (define (fact n)
13   (let* ((t (spawn (fact-thread 1 n)))
14          (res (join t)))
15     (printf "fact(~a) = ~a~n" n res)))
16 (fact (read-integer))
```

Listing 1: Thread-based program that computes a factorial.

Actor-based approach. In the actor-based program in Listing 2 every node of the computation tree is implemented as an actor that starts with behavior `fact-actor`. Upon reception of a `compute` message, `fact-actor` either directly computes the product of factors and sends the result back to its parent, or it splits the computation in newly created actors and changes its behavior to `fact-actor-wait`. The `fact-actor-wait` behavior waits for receiving partial products through `computed` messages, and multiplies partial products together until it has received all expected partial products. It then sends the result to its parent. A master actor creates the root `fact-actor` and displays the results when the computation is finished.

```
 1 (define fact-actor
 2   (actor ()
 3     (compute (from to parent)
 4       (if (<= (- to from) FragmentSize)
 5         (let ((partial-fact (product from to)))
 6           (send parent computed partial-fact)
 7           (terminate))
 8         (let ((steps (split from to)))
 9           (map (lambda (bounds)
10             (send (create fact-actor)
11                     compute (car bounds) (cdr bounds) self))
12             steps)
13           (become fact-actor-wait 0 (length steps) 1 parent))))))
14 (define fact-actor-wait
15   (actor (received fragments current parent)
16     (computed (result)
17       (let ((new-result (* current result)))
18         (if (= (+ received 1) fragments)
19           (begin
20             (send parent computed new-result)
21             (terminate))
22           (become fact-actor-wait
23             (+ received 1) fragments new-result parent))))))
24 (define master-actor
25   (actor ()
26     (compute (n)
27       (send (create fact-actor) compute 1 n self)
28       (become master-actor))
29     (computed (res)
30       (printf "result = ~a~n" res))
```

```
31       (become master-actor)))
32 (define act (create master-actor))
33 (send act compute (read-integer))
```

Listing 2: Actor program that computes a factorial.

2.2. Process-Modular Analysis Designs

ModConc analyses follow what we call a *process-modular design*. We first discuss what such a design is, before delving in the details of ModConc. Such a design can be applied to the static analysis of concurrent programs that share a number of commonalities already introduced: the notion of *process* as a unit of computation, and interferences between the processes that can be communicated as *communication effects*. Possible communication effects include accesses to shared state, the sending and the receiving of messages, synchronization across processes, process creation and termination, etc. This design enables the analysis to overcome two challenging problems in the static analysis concurrent programs:

1. **Dynamic process creation**: In both the thread-based and the actor-based factorial program, the number of processes created is dynamic and depends on user input. ModConc supports this possibly unbounded number of processes by mapping an infinite domain of processes to a finite domain of *abstract processes*, and by performing intra-process analysis on each of these abstract processes to infer information about other processes that are created or otherwise affected through communication.
2. **State explosion** In both factorial programs, the number of possible interleavings between the processes is high: multiple processes are executed concurrently and their execution may interleave in an exponential number of possible ways. This is known as the state explosion problem in analysis of concurrent programs [94]. Mitigations for this problem have been studied in the context of model checking, leading to partial-order reduction techniques [39, 33] which reduce the number of interleavings a model checker has to explore, rendering it more scalable and enabling the verification of more complex programs. However, such techniques do not solve the explosion problem and still expose exponential behavior in the worst case.
   The design of ModConc is inspired by Cousot and Cousot [21] and does not explicitly model process interleavings in the analysis. These interleavings are still implicitly accounted for: any communication that

occurs between processes will trigger the re-analysis of the affected process to take it into account. The fact that the interleavings are not explicitly modeled prevents ModConc analyses of suffering from the state explosion problem.

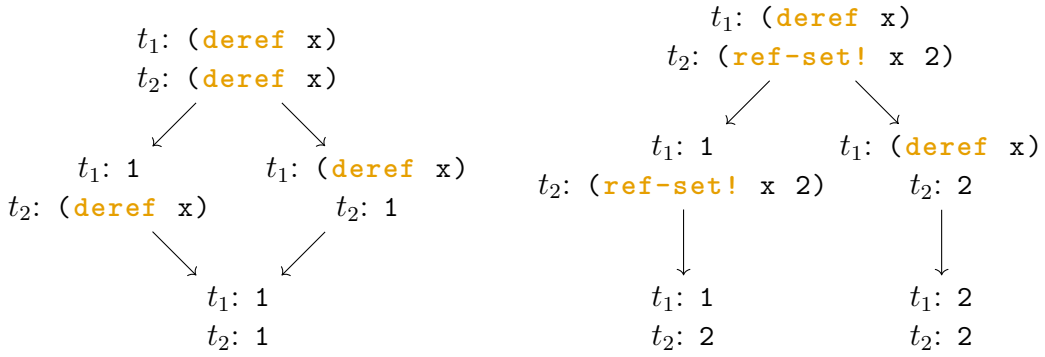
To illustrate the difference between a static analysis that explicitly models process interleavings and a static analysis that follows a process-modular design, consider a program with two threads, t_1 and t_2, where each thread is ready to perform a state transition.

(a) If the operation performed by each transition does not influence the applicability or the outcome of the transition of the other thread, having t_1 transition first and then only t_2 or having t_2 transition first and then only t_1 does not influence the result of the program. This is, for example, the case when both transitions read from, but do not write to, the contents of a reference. This situation is depicted in Fig. 2a. In this case there are two persistent sets from the initial state: the set containing only the transition of t_1, and the set containing only the transition of t_2. An analysis may explore only one of the two interleavings, and still remain sound.

(b) If the transitions do influence each other, then the end result of the program will depend on the interleaving, and an analysis has to take into account both interleavings. This is for example the case if t_1 modifies a reference and at the same time t_2 accesses the same reference. This situation is depicted in Fig. 2b. In this case, there is only one persistent set at the initial state, containing both enabled transitions. This means that if t_1 transitions first, the result of the program may be different from the result where t_2 transitions first. For an analysis to be sound, it needs to account for both interleavings in its results.

Static analyses that explicitly model interleavings may benefit from reduction techniques to explore a single interleaving in the first case, but have to model both interleavings in the second case due to the interference. In the general case, reduction techniques do not reduce the worst-case time complexity of an analysis, which remains exponential.

With a process-modular analysis design inspired from the modular analysis notion of [21], all interleavings are accounted for through over-approximation, but all interleavings are not explicitly explored separately. Instead, the

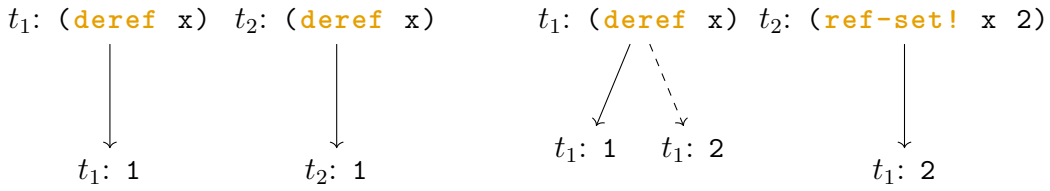


(a) Non-interfering processes.

(b) Interfering processes.

Figure 2: Representation of the concurrent execution of two threads in an all-interleavings analysis. Each program state represents the current expression evaluated or the value reached by each thread (written $t_i : e$ or $t_i : v$ for each thread). Edges represent transitions performed in the execution of the program.

program is analyzed on a per-process basis. Each process is analyzed in isolation, and every interleaving of the analyzed process with other processes is deemed possible. Processes that may conflict need to be analyzed more than once to account for possible conflicts, hence a process-modular analysis will iterate over multiple analyses of the set of processes. However, it can be ensured that the maximal number of iterations does not grow exponentially, thereby ensuring scalability.



(a) Non-interfering threads.

(b) Interfering threads.

Figure 3: Representation of the concurrent execution of two threads in a process-modular analysis. Each node denotes the current expression evaluated by a thread or the value reached by this thread. Edges represent transitions performed during the analysis of a thread. Plain edges are transitions explored during the first iteration of a process-modular analysis, and dashed edges are explored during the second iteration.

We revisit our previous example in the setting of a process-modular analysis. There are two threads to analyze: t_1 and t_2, and each thread is

analyzed in isolation.

(a) In the case of non-interfering transitions, each thread is analyzed just as any sequential program, and the complexity of the analysis becomes the complexity of a sequential analysis multiplied by the number of threads. This is depicted in Fig. 3a. Note that the interleavings of the different threads are not explicitly represented, but rather the result of the analysis is a graph per thread describing the evolution of each thread separately.

(b) In the case of interfering transitions, a first iteration of the analysis analyzes each thread in separation until completion. This is depicted by the plain edges in Fig. 3b. Thread t_2 performs an operation conflicting with thread t_1, and therefore thread t_1 is analyzed again to account for the changes that occur from the execution of thread t_2. The dashed transition of Fig. 3b is therefore also explored.

Although this is a very synthetic example, we can already see that the number of transitions explored is reduced compared to an analysis that explores all interleavings or to an analysis that performs state space reduction. Indeed, in the case of the non-interfering threads, an analysis with state space reduction and a process-modular analysis only need to explore two transitions instead of the four possible transitions. In the case of interfering processes, an analysis with state space reduction needs to explore all four transitions, while a process-modular analysis only explores three transitions. As the number of processes and the number of transitions grow, this difference in the number of transitions and the number of states that have to be explored by each analysis grows as well.

The concept of modular analysis has been formalized by Cousot and Cousot [21], whoi present a general-purpose method to design modular analyses. These ideas have been applied in the context of thread-based programs by using concepts from either assume-guarantee reasoning [31, 44], rely-guarantee reasoning [71, 73], or separation logic [41], and this in a setting limited to a statically known number of executed threads. Recent developments [66, 67] propose process-modular analyses for synchronous message-passing programs, but again limited to programs composed of a constant set of processes that is known a priori.

Process-modular analyses may fare very well in terms of scalability [72], as they are not subject to the state explosion problem. The main challenge

compared to existing work is that dynamic creation of processes is inherent to modern concurrent programs and must be supported by the analysis.

3. The ModConc Approach

Non-modular analyses explicitly explore all possible interleavings of the transition relation of a concurrent semantics (or a sound subset thereof) to derive how a concurrent program evolves at every program point. A non-modular analysis concurrently keeps track of the state of all processes and may step one of them at any given point. Every interference between two processes (process creation and interprocess communication) is immediately effected on the global analysis state.

Analyses resulting from our ModConc approach rely on the same transition relation as a non-modular analysis, but consist of two cleary distinct and alternating phases.

1. In the *intra-process analysis* phase, single processes are analyzed *in isolation* until a fixed point is reached. During this phase, all communication effects are accumulated in a set instead of being applied on the global analysis state.
2. The *inter-process analysis* phase uses the effects accumulated during the intra-process analysis to update the global analysis state and to launch additional intra-process analyses for processes that were created or impacted by these state updates.

From this description it is clear that ModConc analyses, like all sound non-modular analyses, take into account all process creation and communication that may occur during program execution (no information is "lost") but, unlike existing non-modular analyses, do not have to deal with all possible process interleavings at all program points at which interprocess communication occurs. It is for this reason that ModConc analyses scale with respect to process creation and interprocess communication in concurrent programs.

In the remainder of this section we detail ModConc as a step-by-step design method (Section 3.1) and illustrate how a resulting analysis behaves on our previously introduced example programs (Section 3.2). We then discuss important properties of the resulting analyses (Section 3.3) that are termination, soundness, and complexity. ModConc is applied to threads and to actors in the remainder of this paper.

3.1. Steps to Apply ModConc

The ModConc method for constructing a modular analysis for concurrent programs consists of the following steps:

1. Specification of an *operational semantics* for the input language featuring concurrency, modeled by a transition relation.
2. Modification of the operational semantics into a *sequentialized transition relation* annotated with communication effects.
3. Construction of an *intra-process analysis* that infers communication effects, based on the sequentialized semantics.
4. Construction of an *inter-process analysis* that drives intra-process analysis until no new communication effects are inferred.

3.1.1. Step 1: Operational semantics for the input language

The operational semantics of a concurrent programming language can be modeled by a transition relation that specifies how the program state (including the state of its set of processes) evolves in a step-wise fashion. Two kinds of transitions can be distinguished.

1. *Sequential transitions* model sequential operations affecting a single process. In the thread-based and actor-based factorial examples in our Scheme dialect, sequential constructs modeled by this type of transition include `define`, `let`, `+`, and `if`.
2. *Concurrent transitions* model concurrent operations that affect more than one process. In our examples, concurrent operations include `spawn`, `join`, `create`, and `send`.

3.1.2. Step 2: Sequentializing semantics by delaying effects

The second step consists in constructing a sequentialized version from the concurrent semantics for a single process. This modified semantics honors the existing semantics of sequential transitions, but replaces the semantics of concurrent transitions. Each of these transitions are replaced by a transition that only acts on the state of the process performing the transition, and that generates a communication effect describing the effect of applying the transition on the rest of the program state. The result of this step is an intra-process transition relation annotated with a set of communication effects, which is non-empty for concurrent operations.

Examples of communication effects include the process creation effect, generated upon the creation of a new process and containing the state of the

created process; a send communication effect that is generated upon execution of the `send` primitive and that contains information about the target process and the content of the message. We formally define effects for thread-based and actor-based concurrency in Sections 5 and 6, respectively.

3.1.3. Step 3: Intra-process analysis for inferring communication effects

Computing the fixed point of a process using the sequentialized transition relation results in an intra-process analysis that infers communication effects. The intra-process analysis must be parameterized by specific input values for processes, generally including abstractions of the memory heap (called *value store*) and continuation stack (called *continuation store*). Input values also include values returned by other threads in multi-threaded programs, and actor mailboxes in actor programs.

The intra-process analysis explores all possible behaviors of the analyzed process under the assumptions given by the input values, and infers the communication effects as well as changes to the value store and continuation store. To infer communication effects containing values, the analysis needs to be able to reason over the flow of values in a program. In this paper, we rely on an analysis for sequential programs in which the analysis steps over abstract states that over-approximate concrete program states. The result of stepping through a process is a flow graph that models control and value flow of that process. The intra-process analysis finishes when no new behavior can be inferred.

3.1.4. Step 4: Inter-process analysis for driving the intra-process analysis

The inter-process analysis gathers the communication effects inferred by the intra-process analysis on a set of processes and decides the next set of processes that require intra-process analysis. This next set of processes contains newly created processes and processes that depend on the inferred communication effects. For example, if a thread t_1 is *joining* thread t_2, and the intra-process analysis of thread t_2 infers the return value of this process, which is considered to be a communication effect, then the intra-process analysis of thread t_1 must be performed again to include this information. Similarly, based on the communication effect inferred by the intra-process analysis of actor a_1 when it sends a message to actor a_2, the inter-process analysis triggers the intra-process analysis of actor a_2.

Because a concurrent program starts by executing its main process, the inter-process analysis starts with an intra-process analysis of the main process.

The inter-process analysis is finished when no new communication effects can be inferred. The resulting inter-process analysis driving the intra-process analysis is what we refer to in this paper as a modular ModConc analysis.

3.2. Example runs

Consider the thread formulation of our factorial example in Listing 1. We describe each iteration of the modular analysis on this example, and depict the abstraction of this example as inferred by ModConc in Fig. 4. Each iteration consists of one or more intra-process analyses that, except for the main process in the initial iteration, are scheduled by the inter-process analysis.

1. The analysis starts with the intra-process analysis of main thread t_0. The intra-process analysis infers that thread t_0 creates thread t_1 to evaluate expression `(fact-thread 1 n)` (line 13), and that t_0 joins on the result of t_1 (line 14). Therefore, in this initial iteration, the intra-process analysis for thread t_0 cannot fully analyze this thread as its behavior depends on the newly created thread t_1.
2. In the second iteration thread t_1 is analyzed. The intra-process analysis infers that t_1 may terminate with the resulting integer value of `(product from to)` (line 3), and that new threads may be created to evaluate `(fact-thread (car bounds) (cdr bounds))` (line 9) after which they are immediately joined (line 6). For the sake of the example we assume a single thread t_2 may be created (which, in an abstract analysis setting, may represent multiple concrete threads created at this point).
3. The third iteration reanalyzes thread t_0 as new join information has been discovered in the previous iteration. No new communication effects are discovered for t_0 in this iteration. This iteration also analyzes t_2 because it was created in the previous iteration and, similar to the effects inferred for t_1 in the previous interation, infers termination with an integer return value and the creation of (again) t_2 as effects.
4. The fourth iteration reanalyzes t_1 and t_2 because new information about the return value of t_2 has been discovered in the previous iteration and both t_1 and t_2 join on this abstract process. The analysis of t_1 and t_2 again detects integer return values and the creation of t_2. Because all of these effects were already inferred and no new effects have been discovered, the analysis reaches a fixed point for threads t_1 and t_2, completing their analysis.

5. The fifth and final iteration reanalyzes thread t_0 because it depends on the result of thread t_1 for which new join information (the integer result value of t_1) has been discovered in the previous iteration. No previously unencountered effects are generated for t_0, completing the analysis of t_0 and of the entire program.

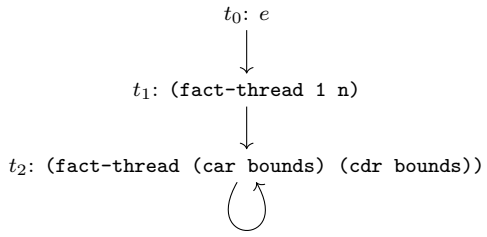


Figure 4: Abstract representation of the factorial program with threads. Nodes represent abstract threads, while an edge from node t_i to t_j indicates that thread t_i creates thread t_j. Note that the unbouded chain of `fact-thread` threads are abstracted to finitely many abstract threads with a self-reference. e is the program under analysis of Listing 1.

Consider now the actor formulation of the factorial example in Listing 2. The analysis of this program using ModConc is very similar to that of the thread-based version: both analyses start analyzing the `main` process, which creates an initial process that is then analyzed in the next iteration. Both analyses terminate once all processes have been analyzed taking in consideration all communication effects that affect them. The differences are in the fact that, in the actor variant, messages are used as the means of communication for actors as opposed to the use of thread joining, and in the fact that the actor version has an extra process for the `master` actor, of which no counterpart is present in the thread version. Other than that, both analyses execute in a similar way and we do not detail the execution of the analysis on this second example.

3.3. Theoretical Properties

3.3.1. Termination

In this paper, we specify the intra-process and inter-process analyses as a fixed-point computation of monotone transfer functions over a finite state space. Termination of these analyses is therefore ensured by Tarski's fixed-point theorem [91].

3.3.2. Soundness

An intra-process analysis is sound only if it relies on a sound sequentialized transition relation that infers communication effects. Soundness of the inter-process analysis relies on the soundness of the intra-process analysis it drives. If the intra-process analysis is sound, the set of communication effects inferred by the analysis of a process is a sound over-approximation of the set of communication effects that may appear at run time. Using this sound over-approximation, the inter-process analysis will at least re-trigger the intra-process analysis of all affected processes. When a fixed point is reached by the inter-process analysis, all processes have been analyzed with soundly over-approximated input values, which results in a sound inter-process analysis.

3.3.3. Complexity

A modular analysis designed with our ModConc approach adds a linear factor (proportional to the total number of abstract communication effects) to the complexity of its underlying sequential analysis. The analysis time grows with the number of abstract processes created by the program under analysis and with other communication effects such as the number of abstract message sends in actor programs or write operations on shared variables in thread programs.

The inter-process analysis will at most have to recompute an intra-process analysis a number of times linear to the number of communication effects. As the number of abstract processes increases, the running time of the analysis increases in proportion.

Because the number of types of communication effects is fixed, the complexity of the overall analysis adds a linear factor to the complexity of the sequential analysis. If the sequential analysis is polynomial, the modular concurrent analysis remains polynomial—unlike a non-modular analysis which becomes exponential. As an example, with the AAM formulation used in this paper—which exhiits a cubic worst-case time complexity—the resulting ModConc analyses have a worst-case time complexity of $\mathcal{O}(|Exp|^4)$ where $|Exp|$ is the number of expression of the program. This because the number of abstract communication effects is bounded by the number of expressions in the program under analysis.

3.4. Applications

The information inferred by ModConc analyses includes the possible running processes, the communication effects they can perform as well an

over-approximation of their data-flow and control-flow in the form of flow graphs, under any possible input and interleaving. This information can serve as a foundation for building a number of client analyses.

In the domain of program comprehension, information from the communication effects such as message sends and process creation can be used to derive *communication topology analyses* [19, 65] in order to provide information about the different processes that may execute in a concurrent program, and more specifically which process creates which other process and how such processes communicate. This information is directly present in the communication effects inferred by the analysis.

Also, the over-approximation of the data-flow and control-flow inferred from the analysis in the form of *flow graphs* can serve as a foundation to bootstrap other analyses, which then do not need to extract data- and control-flow information again as this has already been done by ModConc. This can serve for applications in the domain of defect detection or program verification.

4. Base Language: λ_0

In this work we explore both thread-based (Section 5) and actor-based concurrency (Section 6) on top of a Scheme-like base language λ_0. This base language is based on the λ-calculus in A-Normal Form [34], underlining the fact that ModConc readily supports higher-order languages. This restricted language is chosen to enable focusing on the core principles of ModConc, while still preserving the challenging aspects of combining higher-order with concurrent models that feature mutable shared-memory or actors. Features other than functions and variable bindings (e.g., imperative constructs and classes) have long been investigated by the related work and are of an orthogonal nature to the content of this paper: in the application of ModConc, no assumption is made about the base language and ModConc is therefore not limited with respect to these orthogonal features of the base language.

In this section we start by defining the syntax of λ_0, and specify its abstract semantics. For completeness, the concrete semantics of λ_0 is defined in Appendix A, and we highlight here in gray the parts of the abstract semantics that have undergone changes due to the abstraction.

4.1. Syntax

The syntax of λ_0 is defined in Fig. 5. It supports function definition, function application, and recursive bindings through `letrec`. Our implementation

(see Section 7.1) supports an extended version of this language with more data types, operations, and language constructs, which brings it closer to full-fledged Scheme.

λ_0

Syntax

$e \in Exp ::= ae$
$\quad\quad | \quad (f\ ae)$
$\quad\quad | \quad \texttt{(letrec ((}x\ e_1\texttt{))}\ e_2\texttt{)}$

$f, ae \in AExp ::= x \ | \ lam$
$lam \in Lam ::= \texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)}$
$x, y \in Var$ a finite set of names

Figure 5

4.2. Abstract Semantics

The abstract operational semantics of λ_0, defined in Fig. 6, is a non-deterministic transition relation $(\ \widehat{\rightsquigarrow}\ ) : \widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore} \times \mathcal{P}(\widehat{Effect}) \times \widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore}$, resulting from abstraction of the transition relation of the concrete semantics using the AAM approach [50], and mapping from one process state, store and continuation store to a successor process state, store and continuation store, and generating one or more effects during this step.

We leave the set of effects undefined at this point as λ_0 does not generate any effect. We use this set of effects when extending λ_0 with concurrent extensions in Sections 5 and 6.

State space. A process state $\hat{\varsigma}$ is composed of a control component $\hat{c}$ and a continuation address $\widehat{k}$. The control dictates whether a state is an evaluation state (**ev**) in which an expression is evaluated in a given environment, or a continuation state (**ko**) in which a value has been reached. The continuation address points to the continuation of the computation in the continuation store. Environments ρ map variables to addresses, value stores $\hat{\sigma}$ map addresses to values, and continuation stores $\widehat{\Xi}$ map continuation addresses to continuations. A non-empty continuation $\hat{\kappa}$ consists of a frame $\hat{\phi}$ and a continuation address $\widehat{k}$ that points to the next continuation in the continuation store. The continuations threaded through the continuation store form a continuation stack in a way that is suited for abstraction [50]. Language λ_0 only requires a single type of continuation frame, $\mathbf{letrec}(e, \hat{a}, \hat{\rho})$, for evaluating `letrec` expressions,

λ_0

Abstract state space

$$\hat{\varsigma} \in \widehat{\Sigma} = \widehat{Control} \times \widehat{KAddr} \qquad \widehat{\Xi} \in \widehat{KStore} = \widehat{KAddr} \rightharpoonup \mathcal{P}(\widehat{Kont})$$

$$\hat{c} \in \widehat{Control} ::= \mathbf{ev}(e, \hat{\rho}) \qquad \hat{\kappa} \in \widehat{Kont} ::= \hat{\phi} : \widehat{k} \mid \epsilon$$

$$\mid \mathbf{ko}(\hat{v}) \qquad \hat{\phi} \in \hat{\Phi} ::= \mathbf{letrec}(e, \hat{a}, \hat{\rho})$$

$$\hat{\rho} \in \widehat{Env} = Var \rightharpoonup \widehat{Addr} \qquad \hat{v} \in \widehat{Val} ::= \mathbf{clo}(lam, \hat{\rho})$$

$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Val}) \qquad \hat{a} \in \widehat{Addr} \text{ a finite set of addresses}$$

$$\widehat{k} \in \widehat{KAddr} \text{ a finite set of addresses}$$

Abstract transition relation

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(ae, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \mathbin{\widehat{\rightsquigarrow}} \langle \mathbf{ko}(\hat{v}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Atomic}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash f \mathbin{\widehat{\Downarrow}} \mathbf{clo}((\lambda\ \ (x)\ \ e), \hat{\rho}') \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \hat{v} \qquad \hat{a} = \widehat{alloc}(x, \hat{\sigma})}{\langle \mathbf{ev}((f\ ae), \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \mathbin{\widehat{\rightsquigarrow}} \langle \mathbf{ev}(e, \hat{\rho}'[x \mapsto \hat{a}]), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{App}$$

$$\frac{\hat{a} = \widehat{alloc}(x, \hat{\sigma}) \qquad \widehat{k}' = \widehat{kalloc}(e_2, \hat{\rho}, \hat{\sigma}, \widehat{\Xi}) \qquad \hat{\rho}' = \hat{\rho}[x \mapsto \hat{a}]}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(letrec ((}x\ e_1\texttt{))}\ e_2\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \mathbin{\widehat{\rightsquigarrow}} \\ \langle \mathbf{ev}(e_1, \hat{\rho}'), \widehat{k}' \rangle, \hat{\sigma}, \widehat{\Xi} \sqcup [\widehat{k}' \mapsto \left\{\mathbf{letrec}(e_2, \hat{a}, \hat{\rho}') : \widehat{k}\right\}]\end{array}} \textsc{Letrec1}$$

$$\frac{\widehat{\Xi}(\widehat{k}) \ni \mathbf{letrec}(e, \hat{a}, \hat{\rho}) : \widehat{k}'}{\langle \mathbf{ko}(\hat{v}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \mathbin{\widehat{\rightsquigarrow}} \langle \mathbf{ev}(e, \hat{\rho}), \widehat{k}' \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{Letrec2}$$

Atomic evaluation $\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \hat{v}$

$$\frac{\hat{v} \in \hat{\sigma}(\hat{\rho}(x))}{\hat{\rho}, \hat{\sigma} \vdash x \mathbin{\widehat{\Downarrow}} \hat{v}} \textsc{Var} \qquad \frac{}{\hat{\rho}, \hat{\sigma} \vdash lam \mathbin{\widehat{\Downarrow}} \mathbf{clo}(lam, \hat{\rho})} \textsc{Lambda}$$

Figure 6: Abstract state space and abstract semantics of λ_0.

where e is the body expression, $\hat{a}$ is the address of the newly bound variable being evaluated, and $\hat{\rho}$ is the binding environment. Closures (**clo**) are values that pair a lambda expression with a binding environment.

Due to the abstraction, the set of addresses $\widehat{Addr}$ and $\widehat{KAddr}$ are finite, as they are the only sources of infiniteness in the concrete state space. The propagation of these changes through the state space has as effect that the value store may map a single address to more than one value, and the fact that a continuation store may map a single address to more than one continuation.

Transition relation. The transition relation encodes the common call-by-value λ-calculus semantics. Rule ATOMIC evaluates an atomic expression using atomic evaluation. Rule APP evaluates a function application by atomically evaluating the operator and its operand, and stepping into the resulting function body with an extended environment and value store in which parameters are bound to their corresponding argument values. Rules LETREC1 and LETREC2 encode the semantics of `letrec`. First a new address is allocated for the bound variable. The evaluation then steps into the expression computing the value of this variable, pushing a frame on top of the continuation stack. When the bound value has been evaluated, the store is extended to incorporate it and evaluation continues with the body of the `letrec`. To maintain soundness under the abstraction, updates to the store are modeled as store joins ($\hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}]$[1]).

Atomic evaluation. To evaluate of a variable reference, the variable's address in the environment and its value in the store (rule VAR). Note that more than one value may reside at the same address in the store due to abstraction. Atomic evaluation of a lambda expression returns a closure that pairs the lambda expression with the current binding environment (**clo**).

Address allocation. As part of the abstraction, the domains of addresses $\widehat{Addr}$ and $\widehat{KAddr}$ have been rendered finite, but we have not yet defined these domains. We provide here an instanciation of these domains of the corresponding allocation functions which create elements of these domains. The allocation strategy used here results in a context-insensitive analysis. The use of a widening of the store into a global store in our implementation further results in a flow-insensitive analysis.

[1]The $\sqcup$ operator is defined here as $\forall x, (\hat{\sigma}_1 \sqcup \hat{\sigma}_2)(x) = \hat{\sigma}_1(x) \cup \hat{\sigma}_2(x)$.

λ_0

Allocation

$$\hat{a} \in \widehat{Addr} = Exp \qquad \widehat{alloc}(e, \hat{\sigma}) = e$$
$$\widehat{k} \in \widehat{KAddr} = Exp \times \widehat{Env} \qquad \widehat{kalloc}(e, \hat{\rho}, \hat{\sigma}, \widehat{\Xi}) = (e, \hat{\rho})$$

Analyzing a λ_0 program. In order to analyze a λ_0 program, we need to explore all process states that are reachable according to the transition relation. This can be expressed as the fixed point of the following transfer function, i.e., $\mathrm{lfp}(\mathcal{F}^e)$ is an over-approximation of the set of all reachable states of program e.

λ_0

Transfer function

$$\widehat{\mathcal{F}}^e(S) = S \cup \left\{ (\langle \mathbf{ev}(e, []), \widehat{k}_0 \rangle, [], [\widehat{k}_0 \mapsto \{\epsilon\}]) \right\} \cup \bigcup_{\substack{(\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}) \in S \\ \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \,\widehat{\rightsquigarrow}\, \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'}} (\hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}')$$

5. Shared-Memory Concurrency with Threads: λ_τ

We now add support for shared-memory concurrency to the base language λ_0 defined in the previous section by adding three new concepts.

1. **Thread creation and joining**. Threads can be created to compute a value in a different process, and a thread can *join* another thread to obtain the final value of the computation performed by that other thread. Thread joining is a blocking operation and is a form of synchronization. Note that the notion of thread joining (one thread blocking until another thread finishes its execution) is not to be confused with the notion of store joining used in the semantics (when the information contained in two stores at the same address is merged).
2. **Mutable references**. Thread-based programs usually share mutable state, but λ_0 is free of side-effects. We therefore introduce references into the language, which hold a value that can be read or modified by any thread.

3. **Locks**. Race conditions may be caused by two concurrent threads that access a reference. We therefore introduce locks for developers to model critical sections in concurrent programs to avoid these race conditions. The language is extended with locks as they are a semantically simple form of synchronization, but other forms of synchronization could be considered as well (e.g., synchronized variables and methods).

5.1. Syntax of λ_τ

The language λ_0 extended by the three aforementioned features results in the thread-based concurrent higher-order language λ_τ, of which the syntax is given in Fig. 7. We refer back to Listing 1 for a program written in an extended version of λ_τ supported by our implementation.

λ_τ

Syntax

$e \in \mathit{Exp} ::= \ldots$

`(spawn` e`)`	`(join` ae`)`	
`(ref` ae`)`	`(deref` ae`)`	`(ref-set` ae ae`)`
`(new-lock)`	`(acquire` ae`)`	`(release` ae`)`

Figure 7: Syntax of the thread-based concurrent higher-order language λ_τ.

The `spawn` construct takes as argument an expression to evaluate in a new thread and returns a process identifier that identifies the newly created thread. When a thread finishes its execution, the result of its evaluated expression serves as the return value of that thread. A thread can `join` another thread, blocking the former until the latter finishes its execution and returns a value.

Values can be wrapped into a reference through the `ref` primitive. The `deref` primitive reads the content of a reference, and `ref-set` updates the content of a reference to a new value.

The `new-lock` primitive creates a new lock that can be used to protect critical regions. A lock is acquired through the `acquire` primitive and released through the `release` primitive. A lock acquired by a process can only be released by the same process. If a process tries to acquire a lock that is held by another process, the call to `acquire` blocks until the lock is released.

5.2. Abstract Semantics of λ_τ

We extend the state space and transition relation of λ_0 to account for the concurrent features of λ_τ in three steps: we add thread management primitives, references, and locks. Note that the transition relation here works on the level of a single process, and is annotated with effects to describe the concurrent operations that should be performed on the global state. We discuss the interleavings in the execution of multiple processes later in this section. Also, the local-thread state is not explicit: everything resides in a single store, shared among all processes. Processes have access only to the portion of the store that is visible in the environment $\hat{\rho}$ in which they are executed, and this include state that is shared among multiple threads as well as thread-local state.

Similarly as for λ_0, we directly introduce the abstract semantics and highlight the differences with the concrete semantics in gray. The full concrete semantics is available in Appendix A.

Thread management. Process identifiers are first-class values as they are returned by `spawn` and passed along to `join` to manage threads (Fig. 8). Just like addresses, we leave process identifiers unspecified, but a natural number formulation can be used in a concrete setting. Creating a thread with `spawn` generates a *create* communication effect denoted by $\mathbf{c}(\hat{p}, \hat{\varsigma})$, where $\hat{p}$ is the process identifier and $\hat{\varsigma}$ is the initial state of the created thread (rule Spawn). Joining on a thread generates a *join* communication effect denoted by $\mathbf{j}(\hat{p}, v)$, where $\hat{p}$ is the process identifier of the thread which is joined and v is the return value of this thread.

References. We add first-class references as values to the language (Fig. 9) to support mutability. A reference $\mathbf{ref}(\hat{a})$ contains an address a which is associated to a value in the store. Creating a reference allocates a new address and returns a reference bound to that address (rule Ref). Reading from a reference $\mathbf{ref}(\hat{a})$ returns the value in the store at address $\hat{a}$ and generates a *read* communication effect $\mathbf{w}(\hat{a})$ (rule Deref). Writing to a reference changes the value in the store at the address $\hat{a}$ contained in the reference, and generates a *write* communication effect $\mathbf{r}(\hat{a})$.

Locks. First-class locks are added as values (Fig. 10) in a similar way as with references. A lock is formally represented as $\mathbf{lock}(\hat{a})$, where $\hat{a}$ is an address in the value store that either points to **unlocked** if the lock is unlocked, or to $\mathbf{locked}(\hat{p})$ if the lock is held by process $\hat{p}$. Creating a new lock allocates

λ_τ

Abstract state space

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

$$\hat{p} \in \widehat{PID} \text{ a finite set of process identifiers}$$

Effects

$$\widehat{eff} \in \widehat{Effect} ::= \mathbf{c}(\hat{p}, \hat{\varsigma}) \mid \mathbf{j}(\hat{p}, \hat{v})$$

Transition relation

$$\frac{}{\langle \mathbf{ev}(\texttt{(spawn } e\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{c}(\hat{p}, \langle \mathbf{ev}(e, \hat{\rho}), \hat{k}_0\rangle)} \langle \mathbf{ko}(\mathbf{pid}(\hat{p})), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Spawn}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{pid}(\hat{p})}{\langle \mathbf{ev}(\texttt{(join } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{j}(\hat{p}, \hat{v})} \langle \mathbf{ko}(\hat{v}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Join}$$

Figure 8: Abstract semantics of thread management in the λ_τ language.

an address in the store and associates it to **unlocked** (rule NewLock). Acquiring a lock requires the lock to be unlocked and sets it to a **locked**$(\hat{p})$ value, generating an **acq**$(\hat{p}, \hat{a})$ effect (rule Acquire). Releasing a lock requires the lock to be locked by the same process $\hat{p}$ and changes its value to **unlocked**, generating effect **rel**$(\hat{p}, \hat{a})$ (rule Release).

Process allocation. Figure 11 depicts the abstract allocation strategy for threads, where a process identifier is abstracted by the expression it evaluates. This allocation strategy results in an analysis where abstract process identifiers are not sensitive to the history of the program under analysis. The abstract process allocation function therefore simply extracts the expressions being evaluated by the process.

5.3. Non-Modular Analysis of λ_τ

In order to perform a non-modular analysis of λ_τ programs, one can define a transition relation that acts not on *program states*, but rather on process states: $(\mathrel{\widehat{\Rightarrow}}_{\hat{p}}) : \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore} \times \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}$, which we provide in Appendix B. This transition relation performs a step on a process $\hat{p}$ in a process map $\hat{\pi} \in \widehat{\Pi}$, which are mappings fom process identifiers to their

λ_τ

Abstract state space

$$\hat{v} \in \widehat{Val} ::= \ldots \mid \mathbf{ref}(\hat{a})$$

Effects

$$\widehat{eff} \in \widehat{Effect} ::= \ldots \mid \mathbf{r}(\hat{a}) \mid \mathbf{w}(\hat{a})$$

Transition relation

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

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{ref}(\hat{a}) \qquad \hat{v} \in \hat{\sigma}(\hat{a})}{\langle \mathbf{ev}(\texttt{(deref } ae\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{r}(\hat{a})}{\rightsquigarrow} \langle \mathbf{ko}(\hat{v}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Deref}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{ref}(\hat{a}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(\texttt{(ref-set } ae\ ae'\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{w}(\hat{a})}{\rightsquigarrow} \langle \mathbf{ko}(\hat{v}), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{RefSet}$$

Figure 9: Abstract semantics of references in the λ_τ language.

corresponding process states. From this transition relation, we can then define a transfer function similarly as for λ_0, with the addition that the process that performs the step is non-deterministically picked. The fixed point of this transfer function, $\mathrm{lfp}(\widehat{\mathcal{F}}^e)$ over-approximates all reachable program states under all possible process interleavings.

5.4. Applying ModConc to λ_τ

Instead of performing a non-modular, all-interleavings analysis, we instantiate ModConc to λ_τ programs. The first two steps of the application of ModConc consists in defining the semantics of the language under analysis and an abstract and sequentialized version of this semantics, which is what we have so far.

Intra-process analysis. The third step of ModConc is to define the intra-process analysis, which fully explores one thread in our case, given a number of assumptions. This intra-process analysis, defined in Fig. 13 along with its state space, is parameterized by the following values:

- The process identifier of the process under analysis: $\hat{p}$,

λ_τ

Abstract state space

$$\hat{v} \in \widehat{Val} ::= \ldots \mid \mathbf{lock}(\hat{a}) \mid \mathbf{locked}(\hat{p}) \mid \mathbf{unlocked}$$

Effects

$$\widehat{eff} \in \widehat{Effect} ::= \ldots \mid \mathbf{acq}(\hat{p}, \hat{a}) \mid \mathbf{rel}(\hat{p}, \hat{a})$$

Transition relation

$$\frac{\hat{a} = \widehat{alloc}(\texttt{(new-lock)}, \hat{\sigma})}{\langle \mathbf{ev}(\texttt{(new-lock)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \overset{\widehat{\leadsto}}{\leadsto} \langle \mathbf{ko}(\mathbf{lock}(\hat{a})), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\mathbf{unlocked}\}], \widehat{\Xi}} \textsc{NewLock}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{lock}(\hat{a}) \qquad \hat{\sigma}(\hat{a}) \ni \mathbf{unlocked}}{\langle \mathbf{ev}(\texttt{(acquire } ae\texttt{)}, \hat{\rho}) \rangle, \widehat{k}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{acq}(\hat{p}, \hat{a})}{\rightsquigarrow} \langle \mathbf{ko}(\mathbf{lock}(\hat{a})), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\mathbf{locked}(\hat{p})\}], \widehat{\Xi}} \textsc{Acquire}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{lock}(\hat{a}) \qquad \hat{\sigma}(\hat{a}) \ni \mathbf{locked}(\hat{p})}{\langle \mathbf{ev}(\texttt{(release } ae\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{rel}(\hat{p}, \hat{a})}{\rightsquigarrow} \langle \mathbf{ko}(\mathbf{lock}(\hat{a})), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\mathbf{unlocked}\}], \widehat{\Xi}} \textsc{Release}$$

Figure 10: Abstract semantics of locks in the λ_τ language.

- The initial process state of the process under analysis: $\hat{\varsigma}_0$,
- Over-approximations of the value store and continuation store when this process is executed: $\hat{\sigma}_0$ and $\widehat{\Xi}_0$,
- A *thread-join store*, which contains return values of other threads: $\widehat{J}$.

The state space of the analysis consists of the set of reachable process states (S), the value and continuation stores ($\hat{\sigma}$ and $\widehat{\Xi}$)), and the sets related to effects. The latter comprise the set of created processes (C), the set of processes on which a process may join (in the *thread join* sense) (P), and the set of addresses on which a process depends (A). This last set corresponds to addresses that are read from or written to, or locks that are accessed.

The transfer function operates as follows with respect to generated effects and effect sets:

λ_τ

Process identifier allocation

$$\hat{p} \in \widehat{PID} = Exp \qquad \widehat{palloc}(\mathbf{ev}(e, \_), \_) = e$$

Figure 11: Allocation of process identifiers in λ_τ.

λ_τ

Non-modular transfer function

$$\hat{\mathcal{F}}^e(S) = \left\{([\mathbf{main} \mapsto \langle \mathbf{ev}(e, []), \widehat{k}_0 \rangle], [], [\widehat{k}_0 \mapsto \epsilon])\right\}$$
$$\cup \left\{(\hat{\pi}', \hat{\sigma}', \widehat{\Xi}') \mid (\hat{\pi}, \hat{\sigma}, \widehat{\Xi}) \in S \wedge \hat{p} \in dom(\hat{\pi}) \wedge \hat{\pi}, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\Rightarrow}}_{\hat{p}} \hat{\pi}', \hat{\sigma}', \widehat{\Xi}'\right\}$$

Figure 12: Transfer function for non-modular analysis of λ_τ.

1. The initial state of the process, with the corresponding stores, is visited.
2. Effect-less transitions are trivially taken and generate no communication effects.
3. Transitions that create a process add an element to the set of created processes.
4. Transitions that join on another process $\hat{p}'$ add the dependency on process identifier $\hat{p}'$ to the corresponding set.
5. Other transitions that read from a reference, write to a reference, acquire a lock, or release a lock at a specific address register the address accessed as a dependency in the corresponding set of addresses. In the case of locks, the value of $\hat{p}$ inside the communication effect must correspond to the current process identifier.

Inter-process analysis. The inter-process transfer function uses a process map $\hat{\pi} \in \widehat{\Pi} = \widehat{PID} \to (\widehat{IntraState} \times \widehat{\Sigma})$ that maps every process identifier to the most recent intra-process analysis state and to the initial state of the corresponding process. This transfer function relies on the following functions defined in Fig. 14.

- $explore : \widehat{\Pi} \times \widehat{PID} \times \widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore} \times \widehat{JStore} \to \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore} \times \widehat{JStore}$ requires the current process map $\hat{\pi}$ and performs an intra-process anal-

λ_τ

Intra-process state space

$$\begin{aligned}\widehat{\mathit{IntraState}} &= \mathcal{P}(\widehat{\Sigma}) \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \\ &\quad \times \mathcal{P}(\mathit{Created}) \times \mathcal{P}(\widehat{\mathit{PID}}) \times \mathcal{P}(\widehat{\mathit{Addr}}) \\ \mathit{Created} &= \widehat{\mathit{PID}} \times \widehat{\Sigma}\end{aligned}$$

Intra-process transfer function

$$\widehat{\mathcal{F}}^{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{J}}(\langle S,\hat{\sigma},\widehat{\Xi},C,P,A\rangle) = \langle\{\hat{\varsigma}\},\hat{\sigma}_0,\widehat{\Xi}_0,\varnothing,\varnothing,\varnothing\rangle \tag{1}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \rightsquigarrow \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing,\varnothing\rangle \tag{2}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{c}(\hat{p}',\hat{\varsigma}_2)}{\rightsquigarrow} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}' \\ \hat{p}'=\widehat{palloc}(\hat{\varsigma}_2)}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\{(\hat{p}',\hat{\varsigma}_2)\},\varnothing,\varnothing\rangle \tag{3}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{j}(\hat{p}',\hat{v})}{\rightsquigarrow} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}' \\ \hat{v}\in\widehat{J}(\hat{p}')}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\{\hat{p}'\},\varnothing\rangle \tag{4}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\widehat{eff}}{\rightsquigarrow} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}' \\ \widehat{eff}\in\{\mathbf{w}(\hat{a}),\mathbf{r}(\hat{a}),\mathbf{acq}(\hat{p},\hat{a}),\mathbf{rel}(\hat{p},\hat{a})\}}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing,\{\hat{a}\}\rangle \tag{5}$$

Figure 13: Transfer function for the intra-process analysis for λ_τ.

ysis on the actor with the process identifier $\hat{p}$, initial state $\hat{\varsigma}$, using the value store $\hat{\sigma}$, the continuation store $\widehat{\Xi}$ and the join store $\widehat{J}$. It returns a process map, a value store, a continuation store and a join store containing the information resulting from the intra-process analysis.

- $\mathit{created} : \widehat{\Pi} \to \mathcal{P}(\widehat{\mathit{PID}} \times \widehat{\Sigma})$ returns the set of all created threads inferred by the intra-process analyses, described by their process identifier and initial thread state.
- $\mathit{joins} : \widehat{\Pi} \times \widehat{\mathit{JStore}} \to \mathcal{P}(\widehat{\mathit{PID}} \times \widehat{\Sigma})$ returns the set of threads (described as a pair of process identifier and initial thread state) that join a thread for which the return value has been inferred and stored in the join store $\widehat{J}$.
- $\mathit{conflicts} : \widehat{\Pi} \to \mathcal{P}(\widehat{\mathit{PID}} \times \widehat{\Sigma})$ returns the set of threads (described as a pair of process identifier and initial thread state) that may be in conflict with other threads because they access the same reference or lock.

λ_τ

$$\mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \widehat{J}) = \langle [\hat{p} \mapsto (s, \hat{\varsigma})], \hat{\sigma}', \widehat{\Xi}', \widehat{J}' \rangle$$
$$\text{where } s = \mathrm{lfp}(\widehat{\mathcal{F}}_{\hat{p},\hat{\varsigma},\hat{\sigma},\widehat{\Xi},\widehat{J}})$$
$$\text{and } (\_, \hat{\sigma}', \widehat{\Xi}', \_, \_, \_) = s$$
$$\text{and } \widehat{J}' = \bigsqcup_{\substack{\hat{p} \in dom(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((S,\_,\_,\_,\_,\_,\_),\_) \\ \langle \mathbf{ko}(v), \widehat{k}_0 \rangle \in S}} [\hat{p} \mapsto \{v\}]$$
$$\mathit{conflicts}(\hat{\pi}) = \bigcup_{\substack{\hat{p},\hat{p}' \in dom(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_,\_,\_,\_,\_,A),\hat{\varsigma}) \\ \hat{\pi}(\hat{p}') = ((\_,\_,\_,\_,\_,A'),\hat{\varsigma}') \\ A \cap A' \neq \varnothing}} \{(\hat{p}, \hat{\varsigma}), (\hat{p}', \hat{\varsigma}')\}$$
$$\mathit{joins}(\hat{\pi}, \widehat{J}) = \bigcup_{\substack{\hat{p} \in dom(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_,\_,\_,\_,P,\_),\hat{\varsigma}) \\ \exists \hat{p}' \in P, \widehat{J}(\hat{p}) \neq \varnothing}} (\hat{p}, \hat{\varsigma})$$
$$\mathit{created}(\hat{\pi}) = \bigcup_{\substack{\hat{p} \in dom(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_,\_,\_,C,\_,\_,\_),\_)}} C$$

Figure 14: Auxiliary functions used by the inter-process transfer function for λ_τ.

Figure 15 depicts the inter-process transfer function $\widehat{\mathcal{G}}^e : \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \times \widehat{\mathit{JStore}} \to \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \times \widehat{\mathit{JStore}}$. This transfer function performs the following operations.

1. The main thread is analyzed by the intra-process analysis.
2. Each newly-created thread discovered by a previous intra-process analysis (as extracted by the *created* function) is analyzed.
3. Each thread that joins another thread for which a previous intra-process analysis has inferred a return value (as extracted by the *joins* function) is reconsidered for analysis to account for the possibly new inferred return values.
4. Each thread that accesses a memory address that is part of the set of conflicting addresses between different threads (as extracted by the *conflicts* function) is reconsidered for analysis to account for a possible change in value at the conflicting address.

The inter-process analysis of a λ_τ program e is then the computation of the fixed point of this transfer function, $\mathrm{lfp}(\widehat{\mathcal{G}}^e)$, and results in an over-approximation of the set of all reachable thread states contained in a process map, as well as an over-approximation of the value store and continuation store.

λ_τ

$$\widehat{\mathcal{G}}^e(\langle \hat{\pi}, \hat{\sigma}, \widehat{\Xi}, \widehat{J} \rangle) = \mathit{explore}([], \hat{p}_0, \langle \mathbf{ev}(e, []), \widehat{k}_0 \rangle, \hat{\sigma}, \widehat{\Xi}, \widehat{J}) \tag{1}$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma}) \in \mathit{created}(\hat{\pi})} \mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \widehat{J}) \tag{2}$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma}) \in \mathit{joins}(\hat{\pi}, J)} \mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \widehat{J}) \tag{3}$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma}) \in \mathit{conflicts}(\hat{\pi})} \mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \widehat{J}) \tag{4}$$

Figure 15: Inter-process transfer function for λ_τ.

Soundness of this analysis follows from the argument given in Section 3.3.2. The inter-process analysis adds a linear factor, proportional to the number of abstract processes created, the number of join operations performed, and the number of addresses accessed (for locks and references), to the complexity of the intra-process analysis. When the intra-process analysis is polynomial, as

is the case for the intra-process analysis defined here, the modular analysis retains a polynomial complexity.

6. Actor-Based Concurrency

In this section we add support for actors to the base language λ_0 from Section 4 and then show how ModConc can be applied to this extended language. We present this on a higher-level, as most of the developments are similar to the ones made for λ_τ in the previous section. We add two new concepts to λ_0.

1. **Actor definition**, *creation*, and *evolution*. Actors can be created by instantiating some defined actor behavior, resulting in the creation of a new process. Actors are associated with *state variables*, which characterize the state of the actor. An actor can update its behavior by *becoming* a behavior with updated state variables.
2. **Messages**. Actors can send messages to and receive messages from other actors. Message handling is central to the concept of actors and therefore each actor is defined with message handlers for each type of message the actor may receive. A message is associated with a *tag* to indicate its type, which is used to select the appropriate message handler in the receiving actor. Messages can contain any number of values of any type.

6.1. Actor Language: λ_α

Extending the base language λ_0 with actors and messages results in the actor-based concurrent higher-order language λ_α, of which the syntax extensions are given in Fig. 16. The factorial definition in Listing 2 is an example of a program written in an extended version of λ_α.

The `actor` primitive defines an actor behavior, associating the types of messages the behavior can receive with a corresponding message processing body. Primitive `create` spawns a new actor from a given behavior and returns its process identifier, while `become` changes the behavior of the current actor. Primitive `send` sends a message to a specific actor identified by its process identifier. Messages exchanged between actors consist of a *tag* t (a simple name) and an argument. Like variable names, tags are syntactic elements of which there are a finite number within a program.

λ_α

Syntax

$$
\begin{array}{lll}
e \in \mathit{Exp} ::= \ldots & & ae \in \mathit{AExp} ::= \ldots \mid \mathit{act} \\
\quad \mid \texttt{(create } ae\ ae\texttt{)} & & \mathit{act} \in \mathit{Act} ::= \texttt{(actor (}x\texttt{)} \\
\quad \mid \texttt{(send } ae\ t\ ae\texttt{)} & & \qquad\qquad\qquad\qquad \texttt{(}t\ \texttt{(}y\texttt{)}\ e\texttt{)}^*\texttt{)} \\
\quad \mid \texttt{(become } ae\ ae\texttt{)} &&
\end{array}
$$

Syntactic functions

$$
\begin{aligned}
\textsc{var}(\texttt{(actor (}x\texttt{)} \ldots\texttt{)}) &= x \\
\textsc{handler}(\texttt{(actor (}x\texttt{)} \ldots \texttt{(}t\ \texttt{(}y\texttt{)}\ e\texttt{)} \ldots\texttt{)}, t) &= (y, e)
\end{aligned}
$$

Figure 16: Syntax of the actor-based concurrent higher-order λ_α language.

To simplify the presentation, but without loss of generality, λ_α is limited to actors and message handlers with one argument. Our implementation, described in Section 7.1, supports an arbitrary number of arguments.

6.2. Abstract Semantics of λ_α

The abstract semantics of λ_α is defined in a similar manner as for λ_τ. We provide the full abstract semantics in Fig. 17 and highlight here the differences and similarities with the semantics of λ_τ.

State space. Actor definitions are first-class values (**actdef**) in the language, whereas threads have no definitions as they are simply defined by their expressions e. A process state ($\hat{\varsigma}$) has the usual control component $\hat{c}$ and continuation address $\widehat{k}$, but has one more component: the current behavior $\hat{b}$ of the actor. An actor can have as behavior either an actor definition coupled to its extended definition environment (**act**), or the behavior of the main actor (**main**). Compared to λ_0 and λ_τ, the control component has an additional possible value: an actor can be **wait**ing for a message.

Actors can generate three kinds of communication effects. Creating a new actor generates a *create* effect $\mathbf{c}(\hat{p}, \hat{\varsigma})$, stating that an actor with initial state $\hat{\varsigma}$ and process identifier $\hat{p}$ is created. This effect is identical to the one for λ_τ. The other effects differ: actors have no shared states nor locks, but rather communicate by exchanging messages. An actor can therefore send a message with tag t and argument $\hat{v}$ to an actor with process identifier $\hat{p}$, generating

communication effect $\mathbf{snd}(\hat{p}, t, \hat{v})$, and an actor receiving a message with tag t and argument $\hat{v}$ generates $\mathbf{rcv}(t, \hat{v})$ as communication effect.

Transition relation. Atomic evaluation is extended to support actor definitions. Evaluating an actor definition yields an **actordef** value that pairs the definition with the binding environment (rule ACTOR).

To create an actor, the behavior and arguments given to `create` are evaluated and a *create* communication effect is generated containing the state of the new actor, initially **wait**ing for messages. The creating actor then reaches the process identifier of the created actor as value (rule CREATE). An actor can change its behavior through a `become` statement that binds the new value of its argument to the given value and sets the actor to a **wait** state (rule BECOME).

Sending a message requires evaluating the first argument to the process identifier of the receiving actor, evaluating the message argument, and generating the corresponding communication effect (rule SEND). Receiving a message depends on a **rcv** communication effect, and requires extracting the handler corresponding to the tag of the received message, binding the handler parameter to the received value, and evaluating the handler body (rule RECEIVE).

Just as for λ_τ, we propose in Fig. 18 an allocation strategy that doesn't preserve process history information: an abstract process identifier of an actor is the syntactic behavior definition with which this actor has been spawned.

6.3. Modular Analysis of λ_α

We skip the description of non-modular analysis of λ_α as it is similar to the one of λ_τ, and instead focus on highlighting the similarities and differences in the modular analysis of λ_α with respect to the one of λ_τ.

Intra-process analysis. The intra-process analysis of λ_α is similar to the one of λ_τ: it explores every process state reachable from the transition relation, and store the impact of communication effects in sets that are used by the inter-process analysis. As for λ_τ, the set of processes created by the process under analysis is computed (C). The only other set needed is the set of messages sent to other actors (M). The intra-process analysis transfer function is provided in Fig. 19, and performs the following operations.

1. The initial state and stores are part of the intra-process analysis state. The initial sets of created processes and sent messages are empty.

λ_α

Abstract state space

$$\hat{\varsigma} \in \widehat{\Sigma} = \widehat{Control} \times \widehat{Beh} \times \widehat{KAddr}$$

$$\hat{c} \in \widehat{Control} ::= \ldots \mid \mathbf{wait}$$

$$\hat{v} \in \widehat{Val} ::= \ldots \mid \mathbf{actdef}(act, \hat{\rho})$$

$$\hat{b} \in \widehat{Beh} ::= \mathbf{act}(act, \hat{\rho}) \mid \mathbf{main}$$

$$\hat{p} \in \widehat{PID} \text{ a finite set of process identifiers}$$

Effects

$$\widehat{eff} \in \widehat{Effect} ::= \mathbf{c}(\hat{p}, \hat{\varsigma}) \mid \mathbf{snd}(\hat{p}, t, v) \mid \mathbf{rcv}(t, v)$$

Atomic evaluation

$$\frac{}{\hat{\rho}, \hat{\sigma} \vdash act \mathbin{\widehat{\Downarrow}} \mathbf{actdef}(act, \hat{\rho})} \textsc{Actor}$$

Transition relation

$$\frac{\begin{array}{c}\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{act}(act, \hat{\rho}') \\ \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v} \qquad \hat{a} = \widehat{alloc}(x, \hat{\sigma}) \qquad x = \textsc{var}(act)\end{array}}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(create } ae\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{c}(\hat{p}, \langle \mathbf{wait}, \mathbf{act}(act, \hat{\rho}'[x \mapsto \hat{a}]), \widehat{k}_0 \rangle)} \\ \langle \mathbf{ko}(\mathbf{pid}(\hat{p})), \hat{b}, \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}\end{array}} \textsc{Create}$$

$$\frac{\begin{array}{c}\hat{a} = \widehat{alloc}(x, \hat{\sigma}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{actdef}(act, \hat{\rho}') \\ \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v} \qquad \hat{b} = \mathbf{act}(act, \hat{\rho}'[x \mapsto \hat{a}]) \qquad x = \textsc{var}(act)\end{array}}{\langle \mathbf{ev}(\texttt{(become } ae\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\rightsquigarrow}} \langle \mathbf{wait}, \hat{b}', \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{Become}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{pid}(\hat{p}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(\texttt{(send } ae\ t\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{snd}(\hat{p}, t, \hat{v})} \langle \mathbf{ko}(\hat{v}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Send}$$

$$\frac{\hat{a} = \widehat{alloc}(y, \hat{\sigma}) \qquad (y, e) = \textsc{handler}(act, t)}{\begin{array}{c}\langle \mathbf{wait}, \mathbf{act}(act, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{rcv}(t, \hat{v})} \\ \langle \mathbf{ev}(e, \hat{\rho}[y \mapsto \hat{a}]), \mathbf{act}(act, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}\end{array}} \textsc{Receive}$$

Figure 17: Abstract semantics of the λ_α language.

λ_α

Allocation

$$\hat{p} \in \widehat{PID} = \widehat{Beh} \qquad \widehat{palloc}(\langle \_, \hat{b}, \_ \rangle, \_) = \hat{b}$$

Figure 18: Abstract allocation strategy for the λ_α language.

2. Effect-less transitions are taken without generating any communication effect.
3. Transitions that process a message or change the behavior of the actor are taken without adding information to the set of created actors and messages sent.
4. Transitions that create a process add an element to the set of created processes.
5. Transitions that receive a message extract the message content from the mailbox.

Inter-process analysis. The inter-process analysis for λ_α, as for the one for λ_τ, operates on a process map. A difference lies in the fact that this process map also contains an abstract mailbox for each actor: $\pi \in \Pi = PID \to (\widehat{IntraState} \times \Sigma \times Mbox)$.

The transfer function relies on the following auxiliary functions, defined in Fig. 20, and which works is similar ways as the auxiliary function for λ_τ.

- $explore : \widehat{PID} \times \widehat{\Sigma} \times \widehat{Mbox} \times \widehat{Store} \times \widehat{KStore} \to \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}$ performs an intra-process analysis on the actor with the process identifier $\hat{p}$, initial state $\hat{\varsigma}$, mailbox $\widehat{mb}$, using value store $\hat{\sigma}$ and continuation store Ξ. It returns a process map, value store and continuation store containing the information resulting from the intra-process analysis.

- $created : \widehat{\Pi} \to \mathcal{P}(\widehat{PID} \times \widehat{\Sigma})$ returns the set of all created actors computed by the analysis, described by their process identifier and their initial actor state.

- $sent : \widehat{\Pi} \times \widehat{PID} \to \mathcal{P}(\widehat{Message})$ returns the set of messages sent to actor with process identifier $\hat{p}$.

λ_α

Intra-process analysis state space

$$\begin{aligned}\widehat{\mathit{IntraState}} &= \mathcal{P}(\widehat{\Sigma}) \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \\ &\quad \times \mathcal{P}(\mathit{Created}) \times \mathcal{P}(\mathit{Sent}) \\ \mathit{Created} &= \widehat{\mathit{PID}} \times \widehat{\Sigma} \\ \mathit{Sent} &= \widehat{\mathit{PID}} \times \widehat{\mathit{Message}}\end{aligned}$$

Intra-process analysis transfer function

$$\widehat{\mathcal{F}}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}}(\langle S,\hat{\sigma},\widehat{\Xi},C,M\rangle) = \langle\{\hat{\varsigma}_0\},\hat{\sigma}_0,\widehat{\Xi}_0,\varnothing,\varnothing\rangle \tag{1}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \rightsquigarrow \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing\rangle \tag{2}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\widehat{\mathit{eff}}}{\rightsquigarrow} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}' \\ (\widehat{\mathit{eff}}=\mathbf{b}(\hat{b},\hat{v}))\vee(\widehat{\mathit{eff}}=\mathbf{rcv}(t,\hat{v})\wedge(t,\hat{v})\in\widehat{mb})}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing\rangle \tag{3}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{c}(\hat{p}',\hat{\varsigma}_2)}{\rightsquigarrow} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\{(\hat{p}',\hat{\varsigma}_2)\},\varnothing\rangle \tag{4}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{snd}(\hat{p}',t,\hat{v})}{\rightsquigarrow} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\{(\hat{p}',(t,\hat{v}))\}\rangle \tag{5}$$

Figure 19: Transfer function for the intra-process analysis for λ_α.

λ_α

$$explore(\hat{p}, \hat{\varsigma}, \widehat{mb}, \hat{\sigma}, \widehat{\Xi}) = \langle [\hat{p} \mapsto (s, \hat{\varsigma}, \widehat{mb})], \hat{\sigma}', \widehat{\Xi}' \rangle$$
$$\text{where } s = \text{lfp}(\widehat{\mathcal{F}}_{\hat{p},\hat{\varsigma},\hat{\sigma},\widehat{\Xi},\widehat{mb}}) \text{ and } (\_, \hat{\sigma}', \widehat{\Xi}', \_, \_) = s$$

$$created(\hat{\pi}) = \bigcup_{\substack{\hat{p} \in dom(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_,\_,\_,C,\_),\_,\_)}} C \qquad sent(\hat{\pi}, \hat{p}) = \bigcup_{\substack{\hat{p} \in dom(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_,\_,\_,\_,M),\_,\_) \\ (\hat{p},\hat{m}) \in M}} \{\hat{m}\}$$

Figure 20: Auxiliary functions used in the inter-process analysis for λ_α.

Finally, the inter-process analysis for λ_α explores all processes based on the information inferred from previous intra-process analysis, in a similar way as for λ_τ. It is depicted in Fig. 21.

1. The main process has an empty mailbox and is analyzed with the intra-process analysis.
2. Each process to which a message is sent—as extracted from the results of previous intra-process analyses by the *sent* function—is re-analyzed with an updated mailbox.
3. Each process that has been created—as extracted from the result of previous intra-process analyses by the *created* function—is analyzed, starting with an empty mailbox.

As for λ_τ, the result of running the inter-process analysis is a process map that maps each abstract process to its set of reachable process states, and its set of generated effects. In addition, an over-approximation of the mailbox of each actor is also computed.

The soundness and complexity of the modular analysis for λ_α follow in the same way as they do for λ_τ: the soundness of the inter-process analysis follows from the soundness argument given in Section 3.3.2, and the analysis remains polynomial, where the number of abstract processes created and abstract messages sent add a linear factor to the complexity of the intra-process analysis.

λ_α

$$\widehat{\mathcal{G}}_e(\langle \hat{\pi}, \hat{\sigma}, \widehat{\Xi} \rangle) = \mathit{explore}(\hat{p}_0, \langle \mathbf{ev}(e, []), \mathbf{main}(e), \widehat{k}_0 \rangle, \varnothing, \hat{\sigma}, \widehat{\Xi}) \tag{1}$$

$$\sqcup \bigsqcup_{\substack{\hat{\pi}(\hat{p})=(\_,\hat{\varsigma},\widehat{mb}) \\ \widehat{mb}' = \widehat{mb} \cup \mathit{sent}(\hat{\pi},\hat{p})}} \mathit{explore}(\hat{p}, \hat{\varsigma}, \widehat{mb}', \hat{\sigma}, \widehat{\Xi}) \tag{2}$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma}) \in \mathit{created}(\hat{\pi})} \mathit{explore}(\hat{p}, \hat{\varsigma}, \varnothing, \hat{\sigma}, \widehat{\Xi}) \tag{3}$$

Figure 21: Inter-process analysis transfer function for λ_α.

7. Empirical Validation

We applied the modular analyses described in this paper to a number of benchmark programs to compute flow graphs over-approximating the behavior of concurrent programs written in λ_τ and λ_α, and to deduce the communication effects that may be generated by each process. We performed a number of experiments on our implementation of the analyses and their result.

7.1. Implementation

We applied ModConc to both the thread-based language λ_τ and the actor-based language λ_α. We employed the Scala-AM static analysis framework [90, 88] to implement support for these languages and their analyses. We made this implementation available through a replication package[2]. The base Scheme language supported by Scala-AM goes beyond λ_0, being a large subset of R5RS Scheme [2] with support for lists, cons cells, vectors, and around 100 standard Scheme primitives and other Scheme constructs such as named lets, do-notation, etc.

Our implementation also contains an optimized non-modular analysis of both λ_τ and λ_α. The non-modular analysis and the modular ModConc

[2]Available at `https://github.com/acieroid/scala-am/tree/modularthreads` for the thread-based language, and at `https://github.com/acieroid/scala-am/tree/modularactors` for the actor-based language.

analyses share the same code base, and we applied a number of common optimizations in their implementation. The optimizations applied to the non-modular analyses are essential to improve their scalability in order to support more than a handful of benchmarks, while the optimization applied to the modular analyses is a common application of general knowledge about fixed points. Altogether, this shared code base and optimizations ensure that when comparing non-modular and modular versions of the analyses, we do compare the impact of ModConc on the results of the analyses.

Strong updates in non-modular analyses. The implementations of the non-modular concurrency analyses take advantage of strong updates over the process map where possible [68]. With this improvement, when an abstract process identifier maps to a unique abstract process in a process map, joins can be replaced by updates. As soon as a process identifier may map to more than one concrete process, joins still have to be used to ensure soundness. This yields an important improvement for the analysis of programs in which some of the abstract processes are created only a fixed number of times. However, as soon as an abstract process is created more than once (e.g., for worker threads created at the same syntactic location in a loop), strong updates cannot be performed anymore, so we do not consider this improvement crucial for analyzing programs that create unboundedly many processes.

Macro-stepping in non-modular analyses. A crucial optimization for the non-modular analyses is the use of abstract macro-stepping [3, 89]. Under this optimization, a non-modular analysis explores the behavior of a single process until it performs an operation that may influence other processes. This drastically reduces the number of interleavings to explore, yet preserves soundness.

Fixed-point computations in modular analyses. The fixed-point computations of the modular analyses are only performed when necessary. If during an iteration of the inter-process analysis a process depends on communication effects for which the process was already analyzed, then that process is not reanalyzed because the result of the analysis would yield identical results.

7.2. Benchmarks

We validate our analyses on two sets of 28 benchmark programs, one set for each concurrency model. These benchmark programs are listed in Table 1

and available online[3].

Our actor programs stem from the widely-used Savina benchmark suite [56], which consists of 28 actor programs written in Scala. These benchmark programs exhibit multiple instances of actors and dynamic process creation. We translated each benchmark program—ranging from 102 to 616 lines of Scala code—to our extended version of λ_α, resulting in programs that range from 17 to 293 lines of code. Programs of this size are beyond what is currently supported by existing non-modular analyses for actor programs, as demonstrated in Section 7.4.

For multi-thread programs there exist no benchmark suite equivalent to the Savina suite for actor progams. We therefore created our own suite of benchmark programs inspired by common concurrency problems and literature, including usual concurrent algorithms such as the alternating bit protocol, the Dekker algorithm, the producer-consumer problem, the dining philosophers problem; multi-threaded implementations of common computer science problems such as matrix multiplication, factorial computation, sorting algorithm, sudoku solution checker; and implementation of concurrency models (actors, atomics and software transactional memory) on top of threads. These benchmarks range from 40 to 219 lines of λ_τ code.

For our evaluation, we executed all benchmark programs under Scala 2.12.2 using Java 1.8.0_102 on a Mid-2014 MacBook Pro with a 2.8 GHz Intel Core i7 and 16 GB of RAM, and we report on the average timing of 20 runs after 10 warmup runs.

7.3. Soundness Testing

Section 3.3.2 outlines the soundness proofs for the analyses described in this paper. In addition, we provide empirical evidence for the soundness of our implementation of ModConc for λ_τ and λ_α through *soundness testing* [77, 5]. To this end, we verify that all information recorded during concrete runs of each benchmark program is indeed over-approximated by the analysis of the same program. No unsound results were reported for any of the benchmark programs, i.e., the implementation of the analyses over-approximate every value that was observed during the concrete runs.

[3] `https://github.com/acieroid/scala-am` in the directory `actor/savina` of the branch `modularactors` for actor benchmarks, and in the directory `threads/suite` of the branch `modularthreads` for the thread benchmarks.

7.4. Performance

We compare the performance of the non-modular analyses to the modular ModConc analyses for λ_τ and λ_α programs. Table 1 lists the results of our experiments. It is evident that our modular analyses scale beyond non-modular ones, as they are able to analyze all programs in our benchmark suite in under 30 seconds, while non-modular analyses can only analyze slightly more than half of them before timing out after 30 minutes.

Actors			Threads		
Benchmark	NonMod	Mod	Benchmark	NonMod	Mod
PP	0.11	0.03	ABP	1.01	0.04
COUNT	0.09	0.02	COUNT	51.08	0.04
FJT	0.03	0.02	DEKKER	0.27	0.01
FJC	ϵ	ϵ	FACT	∞	0.38
THR	233.74	0.02	MATMUL	∞	7.09
CHAM	1499.94	0.05	MCARLO	1134.67	0.02
BIG	∞	0.06	MSORT	∞	0.34
CDICT	21.25	0.09	PC	12.13	0.03
CSLL	∞	0.08	PHIL	0.41	0.03
PCBB	102.99	0.66	PHILD	4.82	0.03
PHIL	∞	0.05	PP	0.60	0.02
SBAR	∞	0.08	RINGBUF	∞	0.10
CIG	1.40	0.05	RNG	0.71	0.04
LOGM	∞	0.11	SUDOKU	∞	0.15
BTX	2.50	0.08	TRAPR	0.42	0.05
RSORT	12.98	0.03	ATOMS	8.71	0.10
FBANK	619.85	0.15	STM	∞	19.76
SIEVE	0.09	0.03	NBODY	∞	1.26
UCT	∞	0.85	SIEVE	0.64	0.16
OFL	∞	5.75	CRYPT	∞	20.48
TRAPR	10.24	0.08	MCEVAL	∞	5.23
PIPREC	0.38	0.05	QSORT	∞	0.19
RMM	∞	0.45	TSP	∞	0.87
QSORT	∞	1.41	BCHAIN	0.46	0.24
APSP	∞	1.06	LIFE	663.71	5.69
SOR	∞	2.39	PPS	56.67	0.54
ASTAR	∞	0.26	MINIMAX	∞	7.74
NQN	∞	0.55	ACTORS	∞	1.61
Analyzed	15/28	28/28	Analyzed	15/28	28/28

Table 1: Performance evaluation on our two benchmark suites. The *NonMod* columns indicate timing for non-modular analyses, and *Mod* columns indicate timing for ModConc modular analyses. Times are in seconds. We employed a timeout of two minutes, after which we denote an analysis time as infinite (∞). We denote the analysis time as ϵ if the analysis completes under 1 millisecond.

7.5. Precision

Similar to the empirical verification of soundness above, we also measure the precision of the different analyses according to the precision evaluation method of our previous work [89], by comparing the maximum-precision abstraction of the observed values[4] in concrete runs of the programs with the abstract values computed for each analysis. To that end, we aggregate all observed values during each of the 1000 concrete runs of each benchmark program, and we compare these values after abstraction to the results of the analysis. Values computed by the analysis for which no corresponding concrete value has been observed are called *spurious* and result in loss of precision. The more spurious elements an analysis produces, the less precise the results of the analysis become. We count the spurious elements for each benchmark program and report on these numbers in Table 2

For λ_τ programs, we observe created threads, joined threads, read and written addresses, and acquired and released locks. For λ_α programs, we observe created actors, received messages and `become` statements executed.

We conclude that the precision of each non-modular analysis is identical to the precision of its modular counterpart for those benchmark programs on which both analyses terminate (i.e., do not time out). Overall, the precision of modular thread analysis is 90% on the benchmark suite for λ_τ programs[5], and the precision of the modular actor analysis is 94% on the benchmark suite for λ_α programs[6]. We investigated each false positive to determine what caused them and why they are higher for the thread-based benchmarks than for the actor-based ones. A first difference lies in the fact - more effects - read/write/acq/rel effects more complex than send/receive - slightly more complex control-flow

Note that the reported number of spurious elements are merely an upper-bound on the number of potential spurious elements, as they correspond to elements that have not been observed in any concrete execution but that may be present in unexplored concrete executions, and are therefore accounted for in the results of the analysis.

[4]For each concrete run of a benchmark, concrete values are observed and recorded. After all concrete runs of a benchmarks, the set of observed values is abstracted into the abstract domains of λ_τ and λ_α. These abstracted values correspond to the maximal precision that can be attained by an analysis with that abstract domain.

[5]380 true positives and 44 spurious elements, $\frac{380}{380+44} = 0.896$.

[6]317 true positives and 20 spurious elements, $\frac{317}{317+20} = 0.941$.

Actors				Threads			
Benchmark	Observed	NonMod	Mod	Benchmark	Observed	NonMod	Mod
PP	9	0	0	ABP	20	0	0
COUNT	8	0	0	COUNT	6	4	4
FJT	3	0	0	DEKKER	16	5	5
FJC	2	0	0	FACT	21	–	8
THR	5	–	0	MATMUL	40	–	0
CHAM	10	–	0	MCARLO	12	–	0
BIG	10	–	0	MSORT	12	–	0
CDICT	13	0	0	PC	15	0	0
CSLL	16	–	0	PHIL	4	0	0
PCBB	13	0	0	PHILD	8	0	0
PHIL	13	–	0	PP	6	0	0
SBAR	19	–	0	RINGBUF	19	–	2
CIG	9	0	0	RNG	6	0	0
LOGM	15	–	0	SUDOKU	58	–	0
BTX	9	0	0	TRAPR	2	0	0
RSORT	10	0	0	ATOMS	6	0	0
FBANK	38	–	0	STM	11	–	7
SIEVE	8	0	0	NBODY	25	–	0
UCT	20	–	8	SIEVE	4	2	2
OFL	13	–	0	CRYPT	2	–	2
TRAPR	7	0	0	MCEVAL	2	–	2
PIPREC	8	0	0	QSORT	18	–	0
RMM	14	–	0	TSP	12	–	8
QSORT	6	–	0	BCHAIN	7	0	0
APSP	5	–	1	LIFE	16	–	4
SOR	12	–	12	PPS	10	0	0
ASTAR	11	–	0	MINIMAX	4	–	0
NQN	11	–	0	ACTORS	18	–	0
Total	317	–	20	Total	380	–	44

Table 2: Precision evaluation on our benchmark suite. The *Observed* columns provide the number of observed values in all of the concrete executions of each benchmark. The *NonMod* columns provide the number of *spurious* elements, or false positives, for non-modular analyses. A dash (–) is used when the analysis does not terminate on a benchmark. The *Mod* columns provide the number of spurious elements for modular analyses. The final line provides the total count of observed elements and spurious elements.

7.6. Scalability

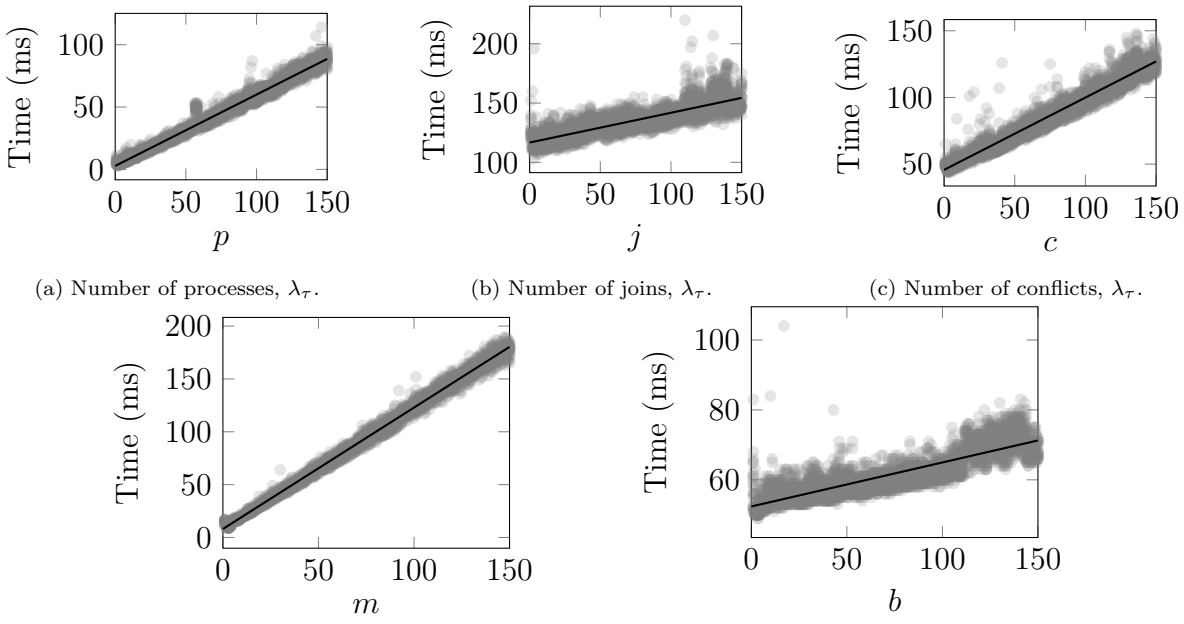
We empirically verify the scalability of the modular analyses. To this end, we generate synthetic benchmark programs for each added factor to the complexity of the analysis (e.g., number of thread joins, number of actors created, ...). Each benchmark has a parameter that will increase the number of the factor in question. We expect to see the time of analysis increase linearly with the increasing values of the parameter in each benchmark.

We ran each benchmark 20 times after 10 warmup runs for values of each parameter ranging from 1 to 150. Figure 22 depicts the results of our experiments. These results empirically demonstrate that the modular analysis for λ_τ scales linearly with the complexity of the sequential analysis, adding as factors the number of different process created, the number of joins and the number of conflicts. We also demonstrate that the modular analysis for λ_α scales linearly with the complexity of the sequential analysis, adding only the number of different actor behaviors created and the number of different kinds of messages sent as factors.

In conclusion, these experiments support the claim that our modular thread analysis scales linearly with the number of abstract process created, the number of joins performed, and the number of conflicts on addresses, and that our modular actor analysis scales with the number of different actor behaviors created and the number of different kinds of messages sent (Section 3.3.3).

8. Related Work

Our ModConc method derives a modular concurrency analysis from a sequentialized concurrency analysis that collects communication effects instead of directly applying them. In this paper, the sequential analysis we use is inspired by Van Horn and Might's work on *Abstracting Abstract Machines* (AAM) [50]. An AAM intra-process analysis can be parameterized in such a way that it is able to infer communication effects of processes with sufficient precision. The resulting ModConc analysis that drives the mutually dependent intra-process and inter-process fixed-point computation to analyze the whole concurrent program is completely automatic, requiring no interaction with the application developer. It approximates the control flow and data flow in concurrent programs, while scaling beyond non-modular analysis techniques. We discuss here other related analysis methods for concurrent programs.



(a) Number of processes, λ_τ.

(b) Number of joins, λ_τ.

(c) Number of conflicts, λ_τ.

(d) Number of messages sent, λ_α.

(e) Number of behaviors, λ_α.

Figure 22: Scalability evaluation. Each graph corresponds to a benchmark which is parameterized by a value (p for abstract processes, j for join operations, c for conflicts, m for number of messages, and b for number of actor behaviors) that is increased from 1 to 150, and shows the running time of the analysis in function of the value of the given parameter.

8.1. Dynamic Analyses

There exist many dynamic methods to analyze concurrent programs, which are generally *unsound* and are hence used for bug detection purposes. Examples include *FindBugs* [51], which performs syntactic checks on Java programs to detect shared-memory concurrency bugs, and *Dialyzer*, for detecting race conditions in Erlang programs [17]. However, the aim of our work is to over-approximate the behavior of concurrent programs in a *sound* manner, at the cost of being subject to imprecisions. Unsound tools are, in contrast, aimed at precisely detecting certain defects.

There exists a number of dynamic analyses based on automated testing: dCUTE [81], Basset [63], Bita [93] and Concuerror [16] perform automated testing of actor-based programs, while jCUTE [82, 84, 83] is a concolic tester for multi-threaded Java programs supporting dynamic process creation. They rely on concolic testing [80] and incorporate partial-order reduction as a way to reduce the number of interleavings to investigate during the analysis. Partial-order reduction methods reduce the state space that has to be explored by an analysis, by identifying equivalent process interleavings. ModConc does not rely on such state space reduction technique as it does not explicitly model interleavings, and is therefore not impacted by state explosion due to interleavings.

A large body of work has been dedicated to the detection of race conditions and of deadlocks in multi-threaded programs. A recent survey [49] describes 43 race detectors, all of which are focused on bug detection rather than on sound analyses, as are most dynamic deadlock detectors [35, 6, 96, 28, 76]. Again, the approach taken by ModConc is the opposite: it provides a sound over-approximation of the possible behaviors of the program under analysis, through static analysis.

8.2. Model Checking

A large effort has been devoted to verifying concurrent systems using model checking techniques [46, 20, 42, 40, 36, 39, 33, 64, 92, 18].

Most model checking tools (e.g., SPIN [46]) require the program to be modeled in a formal specification language. Exceptions to this include Bandera [20], Java PathFinder [42], VeriSoft [40], and McErlang [36]. Bandera, Java PathFinder, and VeriSoft are able to deduce the specification from the program source, while McErlang implements an Erlang virtual machine that supports temporal logic queries. All these tools perform verification by taking as input temporal logic formulas that should hold during the program

execution. These formulas are verified against a non-modular analysis of the model. To mitigate the state explosion problem, state-space reduction techniques are used to avoid exploring redundant interleavings. Techniques such as partial-order reduction have been proposed in the context of shared-memory concurrency [39, 33] and actor-based concurrency [64, 92] to this end. However, even though model checkers can analyze large programs, they are still subject to state space explosion. Moreover, model checkers require the number of processes to be fixed for one verification run: on a generic program with an unspecified number of processes, the model checker has to be run with one process, two processes, etc. ModConc, on the other hand, results in scalable analyses that do not require state-space reduction techniques and can deal with an unbounded, unspecified, number of processes. Also, ModConc-based analyses provide an automatic over-approximation of the possible executions of a program without requiring any user-defined specifications. Future tools can then use this over-approximation as input in order to verify specific properties.

Atig et al. have studied decidability results for concurrent programs with shared memory [9] and locks [10], proposing restricted models under which some properties become decidable, enabling model checkers to prove such properties under specific assumptions. In comparison, ModConc, following the abstract interpretation approach, uses abstractions to overcome the decidability barrier at the cost of precision, but without limiting the program under analysis.

Lal and Reps have introduced the notion of *context-bounded analysis* [62], in which partial correctness guarantees are given by verifying a program within a bound on the number of context switches. ModConc does not make any assumption on the number of context switches.

8.3. Abstract Interpretation

Might and Van Horn extended the AAM approach to a concurrent, shared-memory concurrency setting [68, 87]. The resulting analysis is a non-modular analysis of threads.

Another approach to abstract interpretation for concurrent higher-order programs is Jagannathan et al.'s work [95, 59, 58]. This abstract interpreter computes the possible values of every variable at each program point in order to perform compiler optimizations, and supports both dynamic process creation and higher-order functions. Unlike our approach, this analysis is performed globally on the program which negatively impact scalability, while

our approach performs intra-process analyses that are managed by an inter-process analysis to scale.

Huch founded a line of work on abstract interpretation of actor-based programs [52, 53, 54, 55] for automatic analysis of actor-based concurrency. This work was continued by D'Osualdo in Soter [27, 26, 25]. Soter performs automatic analysis of Erlang programs to verify the absence of errors, and requires user-defined code annotations to verify bounds on mailboxes and mutual exclusion. Soter proceeds in two phases: the first phase computes a model of the program under analysis, and the second phase performs model checking on this model with a property to verify. Like our approach, Soter supports programs with an unbounded number of dynamically created actors. However, unlike our modular approach, Soter's scalability is limited because it performs a non-modular analysis.

Our previous work on non-modular verification of actor-based concurrency [89] is a fully automated technique for verifying the absence of error states and for inferring bounds on mailboxes. As an analysis that immediately applies inter-process communication effects to the global analysis state, it is subject to the state explosion problem and does not scale beyond synthetic programs. The non-modular analysis, however, is able to reason about the order of messages received by each actor. This is in contrast to the actor-based ModConc analysis presented in this work, which approximates a mailbox as an unordered set of messages.

Miné [71] introduces a modular abstract interpretation that reasons over the values of variables in a concurrent, shared-memory setting. This technique has been integrated into the AstréeA static analyzer [70] and has been used to verify very large programs, being able to analyze programs of up to 1.7 million lines of code with 15-predefined running threads in a few days. ModConc analyses have not been tested on programs of that order of magnitude. Our technique performs modular abstract interpretation too, but in addition supports dynamic process creation and scales linearly with the number of dynamic processes created.

Similarly, Midtgaard et al. [66] introduce an *iterated process analysis* for synchronous message passing programs consisting of two processes. Like our ModConc analyses, this analysis is based on an intra-process analysis that analyzes each process in separation, and an inter-process analysis that combines the results in a global fixed-point loop. The approach is able to analyze the order of messages in the possible executions of a programs, while our actor formulation of ModConc does not infer this information. However,

the analysis of Midtgaard et al. [66] is limited to programs with two processes, while ModConc focuses on supporting dynamic process creation.

8.4. Proof systems

Techniques based on proof systems for Erlang such as Rebeca [85, 86], the Erlang Verification Tool [7, 8], and the work by Dam and Fredlund [24] require proof system expertise to prove the correctness of a program. ModConc analyses are completely automatic and require no such expertise.

Concurrent separation logic [78] enables reasoning about thread-based concurrent systems in a modular way, focusing on resource usage. Techniques have been proposed to infer logic formulas for (non-concurrent) separation logic [14], but we are not aware of automatic inference for concurrent separation logic, nor of the existence of similar techniques for actor systems. There exists a number of tools and methods to verify concurrent programs annotated with specifications in concurrent separation logic [11, 57], or in other similar logics, such as rely-guarantee [60], assume-guarantee methods [32], and others [61]. These approaches lack in terms of inference and require user-provided specification of components (procedures and/or processes) in order to verify programs. This is not a concern for ModConc, because our technique requires no user intervention or annotation.

8.5. Type Systems

There exist various type systems that aim at ensuring specific properties about concurrent systems, both for thread-based concurrency [13, 12, 30, 12, 29, 1, 79, 6, 96, 75, 76, 28] and actor-based concurrency [23, 74, 47, 48, 22]. Such type systems support unbounded creation of dynamic processes, as ModConc does. The main differences are the domain of application: type systems have a very specific application (e.g., proving the absence of deadlocks) and are hard to re-use for a different application, while a ModConc analysis can be used as input for a number of applications, as described in Section 3.4.

Type systems for concurrent programs focus mostly on eliminating race conditions and deadlocks. Multiple extensions of the Java type system have been described to ensure the absence of certain concurrent properties such as race conditions [13, 12, 30, 15] and deadlocks [12, 29], some with support for type inference [1, 79, 97], relieving the developper from annotating the program by hand. ModConc does not require any annotation from the developper, and is not focused on detecting such defects but rather on analyzing control and data flow as a way to build other analyses.

Dagnat and Pantel [23] introduce a type-based static analysis that infers interfaces for actors in a subset of Erlang with the goal of detecting orphan messages, i.e., messages sent to but never handled by an actor. Our approach can be used to generate the same information, but reasons about more general control flow and data flow properties. In the domains of active objects and process networks, both related to actors, addition of implicit synchronsation on futures has been proposed in order to enable verification of the absence of deadlocks [38, 37, 43].

Session types enable the verification of adherence of programs to a protocol. Mostrous and Vasconcelos have studied session typing for Erlang [74] for ensuring that sent messages have the expected types. Unlike our approach, this type system is limited to programs that exhibit no unbounded behavior and are written in a specific programming style.

In multiparty session types [47], the protocol is expressed as a *global type* and can involve multiple parties. Typical multiparty session types rely on channels used by at most two entities at a time, while in the actor model any number of actors can send messages to an actor. Multiparty asynchronous session types solve this limitation [48] by allowing an arbitrary number of parties participating in a session. However, the global type that specifies the system requires advanced knowledge about the topology of the system, while actor systems are inherently dynamic (see Section 2). ModConc does not rely on such a global type.

Behavioral types have been studied in the context of actors [22]. They deal with the dynamic topology of the actor system, but are currently restricted to programs that only perform finite computations. ModConc, in contrast, is based on a sequential analysis that readily supports infinite sequential computations, and we have shown that our analysis always terminates.

8.6. Modular Analysis

The concept of a modular analysis as used in this paper has been formalized by Cousot and Cousot [21], which presents different general-purpose methods to design modular analyses. These ideas have been applied in the context of thread-based programs by using concepts from either assume-guarantee reasoning [31, 44], rely-guarantee reasoning [71, 73], or separation logic [41], and this in a setting limited to a statically known number of executed threads. Recent developments have applied modular designs to static analyses of message-passing programs [66] in a way that is similar to this work, but limited to programs composed of only two processes. In contrast, our analyses

support dynamic process creation where the number of created process may be unbounded and we have shown that our approach can be used to analyze both thread-based and actor-based programs. The main challenge compared to existing work based on Cousot's techniques is that dynamic creation of processes is inherent to contemporary concurrent programs must be supported by static analyses, which is the focus of ModConc. As attested by Miné and Delmas [72], modular analyses for concurrency have been identified as being able to fare very well in terms of scalability, as they are not subject to the state explosion problem.

Note that a number of analysis approaches are, by design, modular and even compositional: the analysis of the whole program is divided in the analysis of its components, of which the results can be composed together. This is the case for type systems and for most proof-based approaches, and leads to scalable analyses. We provide here a method that is modular but not compositional: the addition of a new component in the program analysis may influence the analysis of existing components. This loss of compositionality is necessary in order to obtain an automated analysis of programs that can perform for example unrestricted side effects: the addition of a component could modify a variable used in another component without restrictions.

9. Conclusion and Future Work

This paper describes ModConc, a method to derive scalable modular static analyses for concurrent programs. Scalability is achieved by avoiding the state explosion problem that arises when an analysis models every possible process interleaving at every point that processes can possibly interfere. Instead, ModConc uses an intra-process analysis to analyze the behavior of a single process in isolation to infer which processes and communication effects by this process generates. The information obtained from the intra-process analyses is used by an inter-process analysis that, based on effects computed by the intra-process analysis, triggers additional intra-process analyses of newly created processes and of processes that interfere with the analyzed process. When no new interprocess communication effects can be discovered, a sound over-approximation of the behavior of all processes in the program has been computed. The result is a sound and modular whole-program analysis for concurrent programs that infers the set of all running processes and their communication effects in a scalable manner.

We applied ModConc to obtain an analysis for multi-threaded programs and an analysis for actor-based programs, both of which scale linearly in the number of processes created and in the number of other communication effects (thread joins, thread conflicts, actor message sends).

We also evaluated the performance and precision of analyses derived by ModConc. These analyses are able to analyze the entire Savina benchmark suite [56] and an equivalent suite for threads in a matter of seconds, whereas non-modular analyses fail to analyze most of the benchmarks using the same timeout of 30 minutes. Moreover, our modular analyses generate few spurious elements and achieve an overall precision that is identical to the non-modular analyses against which we compare with respect to detecting process creation and communication effects within processes.

We have used context-insensitive allocation strategies in this paper, both for addresses in the value store and in the continuation store, as well as for process identifiers. We identify as future work the investigation of other sensitivities. First, with a global-store widening approach as used in this paper, we lose potential flow-sensitivity, as all values of the same variable are merged together. Avoiding relying on the global store within each component of the analysis would enable regaining flow sensitivity. Second, to obtain context-sensitive analysis, the intra-process analysis would need to be instrumented with a notion of history, that can then be used when allocating addresses for the value store. This follows from traditional ways of obtaining context-sensitivity in AAM-based analyses [50]. Finally, a novel notion of process-sensitivity could be investigated: the allocation of process identifiers could take into account history related to the operations performed by the parent processes or to the history of process creation. This would increase the number of components to take into account by the inter-process analysis, for a possibly increased precision: the analysis could distinguish processes created at the same call site but at different points in time in the execution of the program.

Another avenue for future work consists in retaining ordering information between different processes. This may improve precision sufficiently to enable the verification of stronger properties about processes, such as the properties used by Midtgaard et al. [67] in the context of two-processes communications. In our current formulation of the analysis for actors for example, it is not possible to reason about the order in which messages appear in a mailbox. This information can be useful to prove properties such as absence of errors or bounds on mailboxes [89].

Finally, while we have applied our approach to moderately-sized program-

ming languages and programs, existing modular analyses have been shown to scale to real-world languages such as C and to applications of millions of line of code [70]. Supporting larger languages and programs would require extensive engineering effort. However, many of the aspects of static analyses that are required to support these languages are orthogonal to the contributions of this paper. For example, the support for imperative constructs, or object-oriented programming with inheritance has been investigated in previous work [69, 77] and can incorporated in ModConc. Using ModConc to build analyses for other synchronization mechanism or concurrency primitives such as reentrant locks or software transactional memory should not face foundational barriers, as these can also be modeled similarly as how locks are modeled here. Nevertheless, supporting these different models and languages would further advance the support for concurrency in static analysis.

In summary, this paper demonstrates that it is possible to construct scalable, sound, and precise analyses for concurrent programs that do not suffer from the state explosion problem. This can be achieved by taking advantage of the fact that concurrent programs consist of processes that only interfere through communication effects. By relying on straightforward and well-understood sequential analyses for single processes, we are able to build a sound, modular, scalable analysis for thread-based and actor-based concurrent programs.

Acknowledgments

Quentin Stiévenart is funded by the strategic research program titled "Foundations of Programming Models for Next-Generation Computing Platforms" funded by Vrije Universiteit Brussel. Jens Nicolay is funded by the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[2] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. Adams, D. P. Friedman, E. Kohlbecker, G. Steele, D. H. Bartley, R. Halstead, et al.

Revised5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[4] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1986.

[5] E. S. Andreasen, A. Møller, and B. B. Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 31–36. ACM, 2017.

[6] C. Artho and A. Biere. Applying static analysis to large-scale, multithreaded Java programs. In *13th Australian Software Engineering Conference (ASWEC 2001), 26-28 August 2001, Canberra, Australia*, pages 68–75. IEEE Computer Society, 2001.

[7] T. Arts, M. Dam, L. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In C. Kirchner and H. Kirchner, editors, *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, volume 1421 of *Lecture Notes in Computer Science*, pages 38–41. Springer, 1998.

[8] T. Arts and T. Noll. Verifying generic Erlang client-server implementations. In M. Mohnen and P. W. M. Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, volume 2011 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2000.

[9] M. F. Atig, A. Bouajjani, K. Narayan Kumar, and P. Saivasan. On bounded reachability analysis of shared memory systems. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[10] M. F. Atig, A. Bouajjani, K. Narayan Kumar, and P. Saivasan. Verification of asynchronous programs with nested locks. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 93. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[11] S. Blom and M. Huisman. The vercors tool for verification of concurrent programs. In *International Symposium on Formal Methods*, pages 127–131. Springer, 2014.

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

[13] C. Boyapati and M. C. Rinard. A parameterized type system for race-free Java programs. In L. M. Northrop and J. M. Vlissides, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001.*, pages 56–69. ACM, 2001.

[14] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009.

[15] E. Castegren and T. Wrigstad. Reference capabilities for concurrency control. In *ECOOP 2016, July 17–22, Rome, Italy*, 2016.

[16] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 154–163, 2013.

[17] M. Christakis and K. Sagonas. Static detection of race conditions in Erlang. In M. Carro and R. Peña, editors, *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2010.

[18] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *International Conference on Tools*

and Algorithms for the Construction and Analysis of Systems, pages 570–574. Springer, 2005.

[19] C. Colby. Analyzing the communication topology of concurrent programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995*, pages 202–213, 1995.

[20] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 439–448. ACM, 2000.

[21] P. Cousot and R. Cousot. Modular static program analysis. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.

[22] S. Crafa. Behavioural types for actor systems. *CoRR*, abs/1206.1687, 2012.

[23] F. Dagnat and M. Pantel. Static analysis of communications for Erlang. In *Proceedings of 8th International Erlang/OTP User Conference*, 2002.

[24] M. Dam and L. Fredlund. On the verification of open distributed systems. In K. M. George and G. B. Lamont, editors, *Proceedings of the 1998 ACM symposium on Applied Computing, SAC'98, Atlanta, GA, USA, February 27 - March 1, 1998*, pages 532–540. ACM, 1998.

[25] E. D'Osualdo. *Verification of Message Passing Concurrent Systems*. PhD thesis, University of Oxford, United Kingdom, 2015.

[26] E. D'Osualdo, J. Kochems, and C. L. Ong. Automatic verification of Erlang-style concurrency. In F. Logozzo and M. Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 454–476. Springer, 2013.

[27] E. D'Osualdo, J. Kochems, and L. Ong. Soter: an automatic safety verifier for Erlang. In G. A. Agha, R. H. Bordini, A. Marron, and A. Ricci, editors, *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, pages 137–140. ACM, 2012.

[28] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In M. L. Scott and L. L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 237–252. ACM, 2003.

[29] C. Flanagan and M. Abadi. Types for safe locking. In S. D. Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer, 1999.

[30] C. Flanagan and S. N. Freund. Type-based race detection for Java. In M. S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 219–232. ACM, 2000.

[31] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In D. L. Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

[32] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1-3):153–183, 2005.

[33] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, pages 110–121. ACM, 2005.

[34] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In R. Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993.

[35] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, pages 215–228, 2011.

[36] L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 125–136. ACM, 2007.

[37] E. Giachino, L. Henrio, C. Laneve, and V. Mastandrea. Actors may synchronize, safely! In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, pages 118–131. ACM, 2016.

[38] E. Giachino, N. Kobayashi, and C. Laneve. Deadlock analysis of unbounded process networks. In *International Conference on Concurrency Theory*, pages 63–77. Springer, 2014.

[39] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[40] P. Godefroid. Model checking for programming languages using verisoft. In P. Lee, F. Henglein, and N. D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 174–186. ACM Press, 1997.

[41] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 266–277. ACM, 2007.

[42] K. Havelund and T. Pressburger. Model checking JaVa programs using JaVa PathFinder. *STTT*, 2(4):366–381, 2000.

[43] L. Henrio, C. Laneve, and V. Mastandrea. Analysis of synchronisations in stateful active objects. In *International Conference on Integrated Formal Methods*, pages 195–210. Springer, 2017.

[44] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer, 2003.

[45] C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.

[46] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[47] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008.

[48] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

[49] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *Softw. Test., Verif. Reliab.*, 25(3):191–217, 2015.

[50] D. V. Horn and M. Might. Abstracting abstract machines. In P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 51–62. ACM, 2010.

[51] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[52] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In D. Rémi and P. Lee, editors, *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, pages 261–272. ACM, 1999.

[53] F. Huch. *Verification of Erlang programs using abstract interpretation and model checking*. PhD thesis, RWTH Aachen University, Germany, 2001.

[54] F. Huch. Model checking Erlang programs - abstracting recursive function calls. *Electr. Notes Theor. Comput. Sci.*, 64:195–219, 2002.

[55] F. Huch. Model checking Erlang programs - ltl-propositions and abstract interpretation. In P. Dadam and M. Reichert, editors, *INFORMATIK 2004 - Informatik verbindet, Band 2, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Ulm, 20.-24. September 2004*, volume 51 of *LNI*, pages 438–448. GI, 2004.

[56] S. M. Imam and V. Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In E. G. Boix, P. Haller, A. Ricci, and C. Varela, editors, *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, pages 67–80. ACM, 2014.

[57] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

[58] S. Jagannathan. Locality abstractions for parallel and distributed computing. In T. Ito and A. Yonezawa, editors, *Theory and Practice of*

Parallel Programming, International Workshop TPPP'94, Sendai, Japan, November 7-9, 1994, Proceedings, volume 907 of *Lecture Notes in Computer Science*, pages 320–345. Springer, 1994.

[59] S. Jagannathan and S. Weeks. Analyzing stores and references in a parallel symbolic language. In *LISP and Functional Programming*, pages 294–305, 1994.

[60] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[61] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN Notices*, volume 50, pages 637–650. ACM, 2015.

[62] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[63] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha. A framework for state-space exploration of Java-based actor programs. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 468–479. IEEE Computer Society, 2009.

[64] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In D. S. Rosenblum and G. Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.

[65] M. Martel and M. Gengler. Communication topology analysis for concurrent programs. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, pages 265–286, 2000.

[66] J. Midtgaard, F. Nielson, and H. R. Nielson. Iterated process analysis over lattice-valued regular expressions. In J. Cheney and G. Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 132–145. ACM, 2016.

[67] J. Midtgaard, F. Nielson, and H. R. Nielson. A parametric abstract domain for lattice-valued regular expressions. In X. Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 338–360. Springer, 2016.

[68] M. Might and D. V. Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. In E. Yahav, editor, *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 180–197. Springer, 2011.

[69] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In *ACM Sigplan Notices*, volume 45, pages 305–315. ACM, 2010.

[70] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), 2012.

[71] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In K. L. McMillan and X. Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2014.

[72] A. Miné and D. Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In A. Girault and N. Guan, editors, *2015 International Conference on Embedded Software, EMSOFT 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 65–74. IEEE, 2015.

[73] R. Monat and A. Miné. Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In A. Bouajjani and D. Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 386–404. Springer, 2017.

[74] D. Mostrous and V. T. Vasconcelos. Session typing for a featherweight Erlang. In W. D. Meuter and G. Roman, editors, *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6721 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2011.

[75] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In M. I. Schwartzbach and T. Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 308–319. ACM, 2006.

[76] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 386–396. IEEE, 2009.

[77] J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover. Purity analysis for JavaScript through abstract interpretation. *Journal of Software: Evolution and Process*, pages e1889–n/a, 2017. e1889 smr.1889.

[78] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[79] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 83–94. ACM, 2005.

[80] K. Sen. Concolic testing. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 571–572, 2007.

[81] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[82] K. Sen and G. Agha. CUTE and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 419–423, 2006.

[83] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers*, pages 166–182, 2006.

[84] K. Sen and G. A. Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical report, 2006.

[85] M. Sirjani, F. S. de Boer, and A. Movaghar-Rahimabadi. Modular verification of a component-based actor language. *J. UCS*, 11(10):1695–1717, 2005.

[86] M. Sirjani and M. M. Jaghoori. Ten years of analyzing actors: Rebeca experience. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer, 2011.

[87] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover. Detecting concurrency bugs in higher-order programs through abstract interpretation. In M. Falaschi and E. Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 232–243. ACM, 2015.

[88] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover. Building a modular static analysis framework in Scala (tool paper). In A. Biboudis,

M. Jonnalagedda, S. Stucki, and V. Ureche, editors, *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 105–109. ACM, 2016.

[89] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover. Mailbox abstractions for static analysis of actor programs. In P. Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 25:1–25:30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[90] Q. Stiévenart, M. Vandercammen, W. De Meuter, and C. De Roover. Scala-AM: A modular static analysis framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 85–90. IEEE Computer Society, 2016.

[91] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[92] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In H. Giese and G. Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.

[93] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 114–124. IEEE, 2013.

[94] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996.

[95] S. Weeks, S. Jagannathan, and J. Philbin. A concurrent abstract interpreter. *Lisp and Symbolic Computation*, 7(2-3):173–193, 1994.

[96] A. L. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In A. P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 602–629. Springer, 2005.

[97] J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Cooperative types for controlling thread interference in java. *Science of Computer Programming*, 112:227–260, 2015.

A. Concrete Semantics

The concrete semantics of λ_0 is provided in Fig. A.23, and a concrete allocation strategy is provided in Fig. A.24. The concrete semantics of λ_τ is provided in Fig. A.27, and a concrete process allocation strategy for λ_τ is provided in Fig. A.28. The concrete semantics of λ_α is provided in Fig. A.29, and a concrete process allocation strategy for λ_α is provided in Fig. A.30.

λ_0

State space

$$\begin{aligned}
\varsigma \in \Sigma &= \mathit{Control} \times \mathit{KAddr} & \Xi \in \mathit{KStore} &= \mathit{KAddr} \rightharpoonup \mathit{Kont} \\
c \in \mathit{Control} &::= \mathbf{ev}(e, \rho) & \kappa \in \mathit{Kont} &::= \phi : k \mid \epsilon \\
&\mid \mathbf{ko}(v) & \phi \in \Phi &::= \mathbf{letrec}(e, a, \rho) \\
\rho \in \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Addr} & v \in \mathit{Val} &::= \mathbf{clo}(\mathit{lam}, \rho) \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Val} & a \in \mathit{Addr} &\text{ a set of addresses} \\
& & k \in \mathit{KAddr} &\text{ a set of addresses}
\end{aligned}$$

Transition relation

$$\frac{\rho, \sigma \vdash \mathit{ae} \Downarrow v}{\langle \mathbf{ev}(\mathit{ae}, \rho), k\rangle, \sigma, \Xi \rightsquigarrow \langle \mathbf{ko}(v), k\rangle, \sigma, \Xi} \textsc{Atomic}$$

$$\frac{\rho, \sigma \vdash f \Downarrow \mathbf{clo}((\lambda\ (x)\ e), \rho') \qquad \rho, \sigma \vdash \mathit{ae} \Downarrow v \qquad a = \mathit{alloc}(x, \sigma)}{\langle \mathbf{ev}((f\ \mathit{ae}), \rho), k\rangle, \sigma, \Xi \rightsquigarrow \langle \mathbf{ev}(e, \rho'[x \mapsto a]), k\rangle, \sigma[a \mapsto v], \Xi} \textsc{App}$$

$$\frac{a = \mathit{alloc}(x, \sigma) \qquad k' = \mathit{kalloc}(e_2, \rho, \sigma, \Xi) \qquad \rho' = \rho[x \mapsto a]}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(letrec ((}x\ e_1\texttt{))}\ e_2\texttt{)}, \rho), k\rangle, \sigma, \Xi \rightsquigarrow \\ \langle \mathbf{ev}(e_1, \rho'), k'\rangle, \sigma, \Xi[k' \mapsto \mathbf{letrec}(e_2, a, \rho') : k]\end{array}} \textsc{Letrec1}$$

$$\frac{\Xi(k) = \mathbf{letrec}(e, a, \rho) : k'}{\langle \mathbf{ko}(v), k\rangle, \sigma, \Xi \rightsquigarrow \langle \mathbf{ev}(e, \rho), k'\rangle, \sigma[a \mapsto v], \Xi} \textsc{Letrec2}$$

Atomic evaluation

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

Figure A.23: Concrete state space and concrete semantics for λ_0 programs.

λ_0

Allocation

$$Addr = \mathbb{N} \qquad alloc(x, \Xi) = |\operatorname{Dom}(\sigma)|$$
$$KAddr = \mathbb{N} \qquad kalloc(e, \rho, \sigma, \Xi) = |\operatorname{Dom}(\Xi)|$$

Figure A.24: Concrete allocation strategy for λ_0 programs.

λ_τ

State space

$$v \in Val ::= \ldots \mid \mathbf{pid}(p)$$
$p \in PID$ a set of process identifiers

Effects

$$eff \in Effect ::= \mathbf{c}(p, \varsigma) \mid \mathbf{j}(p, v)$$

Transition relation

$$\frac{}{\langle \mathbf{ev}(\texttt{(spawn } e\texttt{)}, \rho), k\rangle, \sigma, \Xi \overset{\mathbf{c}(\hat{p}, \langle \mathbf{ev}(e, \hat{\rho}), \hat{k}_0\rangle)}{\rightsquigarrow} \langle \mathbf{ko}(\mathbf{pid}(p)), k\rangle, \sigma, \Xi} \textsc{Spawn}$$

$$\frac{\rho, \sigma \vdash ae \Downarrow \mathbf{pid}(p)}{\langle \mathbf{ev}(\texttt{(join } ae\texttt{)}, \rho), k\rangle, \sigma, \Xi \overset{\mathbf{j}(p,v)}{\rightsquigarrow} \langle \mathbf{ko}(v), k\rangle, \sigma, \Xi} \textsc{Join}$$

Figure A.25: Concrete semantics of thread management in the λ_τ language.

B. Non-Modular Analysis of λ_τ

We provide in Fig. B.31 a transition relation that acts on program states for λ_τ, as used in Section 5.3.

λ_τ

State space

$$v \in \mathit{Val} ::= \ldots \mid \mathbf{ref}(a)$$

Effects

$$\mathit{eff} \in \mathit{Effect} ::= \ldots \mid \mathbf{r}(a) \mid \mathbf{w}(a)$$

Transition relation

$$\frac{a = \mathit{alloc}(ae, \sigma) \qquad \rho, \sigma \vdash ae \Downarrow v}{\langle \mathbf{ev}(\texttt{(ref } ae\texttt{)}, \rho), k\rangle, \sigma, \Xi \rightsquigarrow \langle \mathbf{ko}(\mathbf{ref}(a)), k\rangle, \sigma[a \mapsto v], \Xi} \textsc{Ref}$$

$$\frac{\rho, \sigma \vdash ae \Downarrow \mathbf{ref}(a) \qquad v = \sigma(a)}{\langle \mathbf{ev}(\texttt{(deref } ae\texttt{)}, \rho), k\rangle, \sigma, \Xi \overset{\mathbf{r}(a)}{\rightsquigarrow} \langle \mathbf{ko}(v), k\rangle, \sigma, \Xi} \textsc{Deref}$$

$$\frac{\rho, \sigma \vdash ae \Downarrow \mathbf{ref}(a) \qquad \rho, \sigma \vdash ae' \Downarrow v}{\langle \mathbf{ev}(\texttt{(ref-set } ae\ ae'\texttt{)}, \rho), k\rangle, \sigma, \Xi \overset{\mathbf{w}(a)}{\rightsquigarrow} \langle \mathbf{ko}(v), k\rangle, \sigma[a \mapsto v], \Xi} \textsc{RefSet}$$

Figure A.26: Concrete semantics of references in the λ_τ language.

λ_τ

State space

$$v \in \mathit{Val} ::= \ldots \;\mid\; \mathbf{lock}(a) \;\mid\; \mathbf{locked}(p) \;\mid\; \mathbf{unlocked}$$

Effects

$$\mathit{eff} \in \mathit{Effect} ::= \ldots \;\mid\; \mathbf{acq}(p, a) \;\mid\; \mathbf{rel}(p, a)$$

Transition relation

$$\frac{a = \mathit{alloc}(\texttt{(new-lock)}, \sigma)}{\langle \mathbf{ev}(\texttt{(new-lock)}, \rho), k\rangle, \sigma, \Xi \rightsquigarrow \quad \langle \mathbf{ko}(\mathbf{lock}(a)), k\rangle, \sigma[a \mapsto \mathbf{unlocked}], \Xi}\ \textsc{NewLock}$$

$$\frac{\rho, \sigma \vdash ae \Downarrow \mathbf{lock}(a) \qquad \sigma(a) = \mathbf{unlocked}}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(acquire } ae\texttt{)}, \rho)\rangle, k, \sigma, \Xi \overset{\mathbf{acq}(p,a)}{\rightsquigarrow} \\ \langle \mathbf{ko}(\mathbf{lock}(a)), k\rangle, \sigma[a \mapsto \mathbf{locked}(p)], \Xi\end{array}}\ \textsc{Acquire}$$

$$\frac{\rho, \sigma \vdash ae \Downarrow \mathbf{lock}(a) \qquad \sigma(a) = \mathbf{locked}(p)}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(release } ae\texttt{)}, \rho), k\rangle, \sigma, \Xi \overset{\mathbf{rel}(p,a)}{\rightsquigarrow} \\ \langle \mathbf{ko}(\mathbf{lock}(a)), k\rangle, \sigma[a \mapsto \mathbf{unlocked}], \Xi\end{array}}\ \textsc{Release}$$

Figure A.27: Concrete semantics of locks in the λ_τ language.

λ_τ

Process identifier allocation

$$p \in \mathit{PID} = \mathbb{N} \qquad \mathit{palloc}(\_, \pi) = |\operatorname{Dom}(\pi)|$$

Figure A.28: Allocation of concrete process identifiers in λ_τ.

λ_α

State space

$$\varsigma \in \Sigma = \mathit{Control} \times \mathit{Beh} \times \mathit{KAddr}$$
$$c \in \mathit{Control} ::= \ldots \mid \mathbf{wait}$$
$$v \in \mathit{Val} ::= \ldots \mid \mathbf{actdef}(\mathit{act}, \rho)$$
$$b \in \mathit{Beh} ::= \mathbf{act}(\mathit{act}, \rho) \mid \mathbf{main}$$
$$p \in \mathit{PID} \text{ a set of process identifiers}$$

Effects

$$\mathit{eff} \in \mathit{Effect} ::= \mathbf{c}(p, \varsigma) \mid \mathbf{snd}(p, t, v) \mid \mathbf{rcv}(t, v)$$

Atomic evaluation

$$\frac{}{\rho, \sigma \vdash \mathit{act} \Downarrow \mathbf{actdef}(\mathit{act}, \rho)} \textsc{Actor}$$

Transition relation

$$\frac{\begin{array}{c}\rho, \sigma \vdash ae \Downarrow \mathbf{act}(\mathit{act}, \rho') \\ \rho, \sigma \vdash ae' \Downarrow v \qquad a = \mathit{alloc}(x, \sigma) \qquad x = \textsc{var}(\mathit{act})\end{array}}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(create } ae\ ae'\texttt{)}, \rho), b, k\rangle, \sigma, \Xi \xrightarrow{\mathbf{c}(p, \langle \mathbf{wait}, \mathbf{act}(\mathit{act}, \rho'[x \mapsto a]), k_0\rangle)} \\ \langle \mathbf{ko}(\mathbf{pid}(p)), b, k\rangle, \sigma[a \mapsto v], \Xi\end{array}} \textsc{Create}$$

$$\frac{\begin{array}{c}a = \mathit{alloc}(x, \sigma) \qquad \rho, \sigma \vdash ae \Downarrow \mathbf{actdef}(\mathit{act}, \rho') \\ \rho, \sigma \vdash ae' \Downarrow v \qquad b = \mathbf{act}(\mathit{act}, \rho'[x \mapsto a]) \qquad x = \textsc{var}(\mathit{act})\end{array}}{\langle \mathbf{ev}(\texttt{(become } ae\ ae'\texttt{)}, \rho), b, k\rangle, \sigma, \Xi \rightsquigarrow \langle \mathbf{wait}, b', k\rangle, \sigma[a \mapsto v], \Xi} \textsc{Become}$$

$$\frac{\rho, \sigma \vdash ae \Downarrow \mathbf{pid}(p) \qquad \rho, \sigma \vdash ae' \Downarrow v}{\langle \mathbf{ev}(\texttt{(send } ae\ t\ ae'\texttt{)}, \rho), b, k\rangle, \sigma, \Xi \xrightarrow{\mathbf{snd}(p, t, v)} \langle \mathbf{ko}(v), b, k\rangle, \sigma, \Xi} \textsc{Send}$$

$$\frac{a = \mathit{alloc}(y, \sigma) \qquad (y, e) = \textsc{handler}(\mathit{act}, t)}{\begin{array}{c}\langle \mathbf{wait}, \mathbf{act}(\mathit{act}, \rho), k\rangle, \sigma, \Xi \xrightarrow{\mathbf{rcv}(t, v)} \\ \langle \mathbf{ev}(e, \rho[y \mapsto a]), \mathbf{act}(\mathit{act}, \rho), k\rangle, \sigma[a \mapsto v], \Xi\end{array}} \textsc{Receive}$$

Figure A.29: Concrete semantics of the λ_α language.

λ_α

Allocation

$$p \in \mathit{PID} = \mathbb{N} \qquad \mathit{palloc}(\_, \pi) = |\operatorname{Dom}(\pi)|$$

Figure A.30: Concrete allocation strategy for the λ_α language.

λ_τ

Program state space

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

Program transition relation

$$\frac{\hat{\pi}(\hat{p}) \ni \hat{\varsigma} \qquad \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\rightsquigarrow}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\Rightarrow}}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \{\hat{\varsigma}'\}], \hat{\sigma}', \widehat{\Xi}'} \textsc{NoEff}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \hat{\varsigma} \qquad \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{c}(\hat{p}', \hat{\varsigma}'')}{\rightsquigarrow} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}' \qquad \hat{p}' = \widehat{\mathit{palloc}}(\hat{\varsigma}'', \hat{\pi})}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\Rightarrow}}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \{\hat{\varsigma}'\}, \hat{p}' \mapsto \{\hat{\varsigma}''\}], \hat{\sigma}', \widehat{\Xi}'} \textsc{CreateEff}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \hat{\varsigma} \qquad \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{j}(\hat{p}', \hat{v})}{\rightsquigarrow} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}' \qquad \hat{\pi}(\hat{p}') \ni \langle \mathbf{ko}(\hat{v}), \widehat{k}_0 \rangle}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\Rightarrow}}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \{\hat{\varsigma}'\}], \hat{\sigma}', \widehat{\Xi}'} \textsc{JoinEff}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \hat{\varsigma} \qquad \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \overset{\widehat{\mathit{eff}}}{\rightsquigarrow} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}' \\ \widehat{\mathit{eff}} \in \{\mathbf{w}(\hat{a}), \mathbf{r}(\hat{a}), \mathbf{acq}(\hat{p}, \hat{a}), \mathbf{rel}(\hat{p}, \hat{a})\}\end{array}}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \mathrel{\widehat{\Rightarrow}}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \{\hat{\varsigma}'\}], \hat{\sigma}', \widehat{\Xi}'} \textsc{OtherEffs}$$

Figure B.31: Program transition relation for non-modular analysis of λ_τ programs.