Imperial College London

IMPERIAL COLLEGE OF SCIENCE,
TECHNOLOGY AND MEDICINE

DEPARTMENT OF COMPUTING

MENG JOINT MATHEMATICS AND COMPUTING

πACTOR
A π-Calculus Abstraction for ERLANG

AUTHOR
Blaine Rogers
CID: 00987225

SUPERVISORS
Prof. Philippa Gardner
Dr. Emanuele D'Osualdo

SECOND MARKER
Prof. Sophia Drossopoulou

June 20, 2018

Abstract

We propose a procedure for extracting a sound model of an Erlang program in terms of πActor, a novel variant of the π-calculus. The procedure is far from a direct source-to-source translation; in constructing the πActor model of a program, we rely on state-of-the-art abstract interpretation techniques to model control and data flow. We exploit a higher-order control-flow-analysis to produce an intermediate finite state model, and then use that model to construct a set of process reduction definitions for the infinite-state πActor model. Communication and spawning behaviour is preserved between the source program and the abstraction.

Our analysis enables the automatic verification of safety properties of Erlang programs that are unverifiable by previous approaches. We illustrate how the concept of depth-boundedness applies to our πActor model and show that for non-trivial and relevant example programs, interesting coverability queries are decidable on our model. The abstraction can also act as an aid to understand the communication behaviour of a program, or to visualise its communication behaviour.

Acknowledgements

My sincerest and deepest thanks go to Dr. Emanuele D'Osualdo, without whose guidance and support this project would not be anything close to what it is. His ability to answer my often confused or poorly informed questions while dealing with my constant malapropisms is truly remarkable. He has also made the only cup of coffee that I have ever enjoyed.
I would also like to thank Jonathan Sutton for his friendship throughout my time at Imperial. You can tell a great friendship when you and your friend look nothing alike, but people still get your names confused. It has been a great four years.

"...if you believe textbooks, you're liable to have the following schema of research. If A is the question, and B is the answer, then research is a direct path. The problem is that if an experiment doesn't work, or a student gets depressed, it's perceived as something utterly wrong and causes tremendous stress.

I teach my students a different schema. If A is the question, B is the answer, and you start going, and experiments don't work, experiments don't work, experiments don't work, experiments don't work, until you reach a place linked with negative emotions where it seems like your basic assumptions have stopped making sense, like somebody yanked the carpet beneath your feet. And I call this place the cloud. Now you can be lost in the cloud for a day, a week, a month, a year, a whole career, but sometimes, if you're lucky enough and you have enough support, you can see in the materials at hand, or perhaps meditating on the shape of the cloud, a new answer, C, and you decide to go for it. And experiments don't work, experiments don't work, but you get there, and then you tell everyone about it by publishing a paper that reads A arrow C, which is a great way to communicate, but as long as you don't forget the path that brought you there."

– Uri Alon, *Why science demands a leap into the unknown.*

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 The actor model . . . 1
- 1.3 The π-calculus . . . 2
- 1.4 Contributions . . . 2
- 1.5 Outline . . . 3

2 Preliminaries **4**
- 2.1 Notation . . . 4
- 2.2 Mathematical background . . . 4
- 2.3 A brief overview of Core . . . 5
- 2.4 Actor Communicating System (ACS) . . . 10
- 2.5 πActor . . . 11

3 An operational semantics for Core **17**

4 Control flow analysis for Core **22**

5 Generating the πActor model **25**

6 Applications of the πActor model and model checking **28**

7 Evaluation **30**
- 7.1 Case studies . . . 30
- 7.2 Complexity . . . 38
- 7.3 Related work . . . 38

8 Conclusion **39**
- 8.1 Future work . . . 39

9 References **40**

A Concrete reduction rules for the Core semantics **i**

B Abstract reduction rules for the CFA **iv**

C Proof of soundness for the πActor model **vii**
- C.1 A few useful lemmas . . . vii
- C.2 The soundness theorems . . . xii

D Listings for Soter-Compatible Erlang equivalents of Core programs **xxvii**

1 Introduction

1.1 Motivation

Computer systems are becoming increasingly concurrent, particularly in the wake of the internet. Systems designed on the internet are naturally concurrent; an internet application, even if running sequentially on a single server, generally needs to interact concurrently with many independent clients. As we delegate more important tasks to concurrent systems, the need arises for formalisms that make it easier to design and reason about concurrent systems and for tools that can verify their correctness.

It is important to establish that programs are correct. There have been many cases where bugs in computer systems have resulted in loss of profits, loss of property and indeed loss of life. The most widespread way of establishing program correctness is automated testing (unit or otherwise), where programs or their components are run on various inputs and checked to see if they give the appropriate output. This generally involves a large investment of time and effort from the programmer, and results are usually poor. Testing can be used to detect bugs, but cannot establish their absence in general. The traditional alternative to testing is to employ a team of computer scientists to manually verify your code, which is expensive, tedious and error prone. Hence, demand is growing for automatic verification techniques.

Verification problems are interesting because their purest forms are usually undecidable. A typical verification problem is reachability: given two states for a system, is there an execution path for that system that brings it from one to the other? An interesting concrete example of reachability is the halting problem: suppose a program is started with a particular input; is there an execution path that results in the program halting? For Turing-powerful systems, this problem is undecidable, and unfortunately most systems used in practice are Turing-powerful. Hence, the challenge in verification is to come up with an approximation of the system that is expressive enough to describe interesting properties of the system, but not so expressive that verification problems become undecidable or computationally infeasible.

In the context of verifying the behaviour of a program, this approximation takes the form of a sound model of the program semantics. We define a function from states in the concrete execution of the program to states in the execution of the abstract model, and declare the analysis to be sound if every execution trace in the concrete execution has an abstract analogue. More precisely, we would like that if we cannot reach an state in the abstract semantics, then we cannot reach any of the states approximated by that abstract state in the concrete semantics. Properties that can be verified using this notion of soundness are usually called 'safety' properties. We focus here on coverability properties that express some error pattern. In a coverability query, we are given some state A in a model and covering relation (a preorder) between model states and asked if for some state B we can reach from A a state that covers B.

Properties that can be expressed in the form of coverability queries include:

- (unreachability of error locations) Is there a reachable state in which any actor is executing some program location ℓ; is the state with one process that is executing ℓ coverable? ℓ could be the location where an exception is thrown; if the query returns false, then we know that the exception is never thrown.
- (mutual exclusion) Is there a reachable state in which more than one actor is executing the location ℓ? configurations where at least two distinct actors are executing some program location 'l' (aka mutual exclusion)
- (k-boundedness of mailboxes) Is there a reachable state in which more than k messages are in the mailbox of an actor?
- (address privacy) If there a reachable state in which at least two distinct actors both know the address of some third actor?

1.2 The actor model

In the traditional threading model agents (threads) communicate by sharing state. This model is generally considered difficult to reason about, as evidenced by the prevalence of race conditions and deadlocks in multi-threaded code. An alternative is to have agents communicate by message-passing; that is, sending each other messages containing only immutable data. This model tends to be easier to reason about, from both programming and verification perspectives.

One language that implements message-passing concurrency is Erlang. Erlang is a dynamically-typed impure functional language, originally implemented in Prolog for use in the telecommunications industry. Erlang implements the actor model; each process in an Erlang program has a mailbox (a queue of messages), and processes send each other messages asynchronously by appending them to each other's mailboxes. The Erlang runtime has native support for distribution, which has led to its use in a number of successful commercial large-scale projects, including Ericsson's telecom switches, Facebook's chat, WhatsApp and Amazon's SimpleDB.

It is difficult to verify Erlang programs. There are many sources of unboundedness in the state space of an Erlang program: there are unboundedly many processes, each of which is equipped with an unbounded execution stack and mailbox. Higher-order functions are first-class data, and so unboundedly deep closures can be stored. Even if higher-order functions were not first class, Erlang supports data structures that may be nested unboundedly deeply.

Some work has already been done on the verification of Erlang programs. In particular, the Soter tool[8] verifies Erlang programs using an over-approximation in the style of [20]. However, the abstraction used by Soter is quite coarse with regards to process identities. Soter abstracts the processes in a program into a fixed number of classes, such that processes in the same class are indistinguishable when sending messages. A message sent to process in one class might be received by any other process in that class. This leaves us unable to verify many interesting properties. For instance, a common model in client-server applications is that a client will send a request to a server, including its address in the request. The server then responds privately to the client by sending a message to that address. With the coarse abstraction proposed by Soter, we might need to collapse all clients into a single process class, leaving us unable to verify the privacy of the response.

1.3 The π-calculus

The π-calculus[19] is a process algebra. It is an extension of Milner's CCS, reformulated to allow for value-passing and mobility. In the π-calculus, all computation is expressed by the communication of names between processes along channels. Every name can be used as a channel to communicate on, and new, globally unique names can be dynamically at run time. The π-calculus seems a good fit for an abstraction of Erlang more sensitive to process identities; one could give each process a unique name representing its mailbox, creating a new unique name each time a process is spawned.

The π-calculus's ability to create new unique names and send them between processes makes it very powerful; in fact, it is Turing-powerful, and many verification problems (including coverability) are undecidable for it. However, there are certain fragments of π-calculus for which interesting verification problems are decidable. In particular, coverability is decidable for a fragment called depth-bounded π-calculus[18].

With this in mind, we propose a variant of the π-calculus, πActor, carefully designed to be particularly suited to modelling the behaviour of Erlang-style actor model programs.

1.4 Contributions

- We introduce πActor, a variant of the π-calculus designed for the express purpose of modelling Erlang programs. This new calculus inherits decidability results from the π-calculus but allows for the more direct representation of Erlang's inspection of messages before receipt and the easy manipulation of unbounded sets of names. We show that a number of interesting safety properties are expressible in terms of coverability queries on πActor programs.

- We describe Core, a language with syntax and semantics designed to be as close as possible to CoreErlang, with the notable exception that we do not include exception-handling logic. We then propose a procedure for extracting a sound over-approximation of the semantics of a Core program in terms of a πActor program, resulting in a non-uniform analysis. The procedure is far from a direct source-to-source translation; in constructing the πActor model of a program, we rely on state-of-the-art abstract interpretation techniques to model control and data flow. Specifically, we exploit a higher-order control-flow-analysis following the methodology of [20] to produce an intermediate finite state model, and then use that model to construct a set of process reduction definitions for the infinite-state πActor model, a la the procedure for ACS construction[9]. The extracted model preserves the crucial aspects of concurrent actor computation while removing the complexities related to higher-order control flow and complex algebraic datatypes. In so doing, it enables the automatic verification of many safety properties of Erlang programs. The abstract πActor model is agnostic with respect to the model-checking backend used to verify safety properties upon it. The abstraction can also act as an aid to understand the communication behaviour of a program, or to visualise its communication behaviour.

- The model-extraction procedure is proven formally to be sound with respect to the semantics of the original program, in that every concrete execution trace is present in the abstract model; the model is abstract in that it may contain spurious traces. This requires non-trivial manipulation and exploitation of information obtained from the control-flow analysis. In particular, the representation of unbounded data structure containing process ids in the intermediate finite state model requires that we be able to manipulate unboundedly large sets of process ids in the πActor model; this is our motivation for introducing native support for sets into the calculus.

- We demonstrate how the analysis is a strict improvement in precision over the analysis of Soter. Moreover, the complexity of the abstract model generation (though not of model checking) is the same as that of Soter. We illustrate how the concept of depth-boundedness applies to our πActor model and show that while the πActor model is not guaranteed to be depth-bounded, it is for non-trivial and relevant example programs. We show, therefore, that interesting coverability queries for those programs are decidable on our model.

  To the best of our knowledge, our approach offers an analysis strategy that is able to be precise with respect to actor addresses to a degree that is superior to any other analysis for the actor model. Although the approach is demonstrated on the actor model, we believe many of the concepts can be adapted to other forms of message-passing concurrency.

1.5 Outline

In section 2, we define non-standard notation, cover mathematical background including the formal definition of coverability, give an informal overview of Core, and introduce ACS and πActor. In sections 3 to 5, we describe the procedure to extract a πActor model from a Core source. In section 6, we discuss applications of the πActor model and discuss the conditions under which coverability is decidable for it. In section 7 we evaluate the model against the ACS through case studies and give a brief review of related work. Finally, we conclude in section 8 and discuss directions for future work.

2 Preliminaries

We begin this section by introducing some mathematical notation. We then give a brief overview of Core, present ACS, an alternative model against which we evaluate our proposed technique, and describe πActor which will be the target for our abstraction.

2.1 Notation

Definition 2.1 ([finite] power set)**.** Let $\wp(S)$ be the power set of S, and $\bar{\wp}(S) \subseteq \wp(S)$ be the set of finite sets of elements of S (the 'finite power set' of S).

Definition 2.2 (finite sequences)**.** Take S^* to be the set of finite sequences of elements of S. Let $\epsilon \in S^*$ be the empty sequence and let $\cdot$ be overloaded to mean append, prepend and concat for sequences. That is, for $x, x' \in S$ and $\vec{s}, \vec{s'} \in S*$ we let $\cdot$ be right-associative with

- $x \cdot \vec{s}$ being x prepended to $\vec{s}$,
- $\vec{s} \cdot x$ being x appended to $\vec{s}$,
- $x \cdot x' = x \cdot (x' \cdot \epsilon)$,
- $\vec{s} \cdot \vec{s'}$ being $\vec{s}$ concatenated with $\vec{s'}$.

For a finite sequence $\vec{s}$, we may write $x \in \vec{s}$. This should be taken to mean 'x occurs at any position in $\vec{s}$', unless it occurs at the foot of a sum or product (or other iterator context). Where $\vec{s} = s_1 \cdot \ldots \cdot s_n$,

$$\sum_{x \in \vec{s}} E = E[s_1/x] + E[s_2/x] + \cdots + E[s_n/x]$$
$$\prod_{x \in \vec{s}} E = E[s_1/x] \times E[s_2/x] \times \cdots \times E[s_n/x]$$

Where it will not cause confusion, we identify $s \in S$ with $s \cdot \epsilon \in S^*$. The function len : $S^* \to \mathbb{N}$ returns the length of a finite sequence. The function $\lfloor \cdot \rfloor_k : S^* \to S^*$ truncates a sequence $\vec{s}$, keeping its first k elements.

Definition 2.3 (sort)**.** Let sort(S, R) be the function takes a finite set S and a linear ordering R on S and produces the finite sequence of distinct elements of S, sorted according to R.

Definition 2.4 (finite partial functions)**.** Let $S \rightharpoonup T$ be the set of finite partial functions from S to T. For $f, f' \in S \rightharpoonup T$, $s, s_1, \ldots, s_n \in S$, $t_1, \ldots, t_n \in T$, $A \subseteq S$, we write

- $f(s) = \bot$ to mean that f is undefined at s.
- $f[s \mapsto t]$ to mean $(\lambda x.\textbf{if } x = s \textbf{ then } t \textbf{ else } f(x))$, with $f[s_1 \mapsto t_1, \ldots, s_n \mapsto t_n]$ shorthand for $f[s_1 \mapsto t_1] \ldots [s_n \mapsto t_n]$.
- $f \cap A$ with $A \subseteq S$ to mean $(\lambda x.\textbf{if } x \in A \textbf{ then } f(x) \textbf{ else } \bot)$,
- $f \uplus f'$ when the domains of f and f' are disjoint to mean $(\lambda x.\textbf{if } f(x) \neq \bot \textbf{ then } f(x) \textbf{ else } f'(x))$.

We write $[]$ for the function that is everywhere undefined and use $[s_1 \mapsto t_1, \ldots, s_n \mapsto t_n]$ as shorthand for $[][s_1 \mapsto t_1] \ldots [s_n \mapsto t_n]$.

2.2 Mathematical background

Definition 2.5 (transition system)**.** A transition system is a triple $(S, \to, s)$ where $\to \subseteq S \times S$ is the transition relation and $s \in S$ is the initial state. We write $\to *$ for the reflexive transitive closure of $\to$.

Definition 2.6 (reachability problem)**.** Given a transition system $(S, \to, s)$ and a query state $s' \in S$, the reachability problem is to determine if $s \to^* s'$.

Definition 2.7 (preordered transition system)**.** A preordered transition system is a transition system $(S, \rightarrow, s)$ paired with a preorder relation $\sqsubseteq \subseteq S \times S$ (a preorder relation is a reflexive and transitive relation).

Definition 2.8 (coverability problem)**.** Given a preordered transition system $(S, \rightarrow, s, \sqsubseteq)$ and a query state $s' \in S$, the coverability problem is to determine if there is a state $s'' \in S$ for which $s \rightarrow^* s''$ and $s'' \sqsupseteq s$. In the context of coverability queries, we read $s'' \sqsupseteq s$ as 's'' covers s'.

2.3 A brief overview of Core

We now introduce CORE, a modular untyped functional language with actor-model concurrency designed to closely resemble COREERLANG[3], the official intermediate representation of ERLANG.

Definition 2.9 (CORE modules)**.** Fix an (infinite) set of atoms, nominally strings of ASCII characters matching the regular expression

```
'([^']|\\')*'(/[0-9]+)?
```

such that `'spawn'/1`, `'send'/2`, `'self'/0`, `'choice'/2`, `'=:='/2`, `'true'`, `'false'` and `'infinity'` are atoms. Let *atom* range over this set of atoms. Fix also a set of 'variable' names $\mathbb{V}$ ranged over by U and V, nominally strings of ASCII characters which are either alphanumeric or underscore beginning with an upper case letter or an underscore, and a set of labels $\mathcal{L}$ ranged over by ℓ. Then the set of CORE modules $\mathbb{M}$ ranged over by m is given by the grammar in fig. 1, with the following additional constraints:

- For every *atom* `=` U in the exports list, there is a (top level) definition for U in the body.
- No atom appears twice in the exports list.
- The module contains no free names. A name U is bound when it appears on the left side of a definition in a `module`, `let` or `letrec`, when it appears in the argument list of a `fun`, or when it appears in a pattern in a `case` or `receive`. A name is free if it is not bound.

Definition 2.10 (CORE program)**.** A CORE program is a finite set of modules $\mathcal{M} \subset \wp(\mathbb{M})$ satisfying the following properties:

- No two modules in $\mathcal{M}$ have the same name. The name of a module is the atom that appears after the word `module`.
- There is a module with the name `'main'` which exports a nullary function at `'main'/0`.

Remark. We assume from now on that bound variables and labels are unique throughout a program. All programs can be trivially transformed such that this is the case. Note that this means that every variable in a pattern or pattern list must be unique; patterns such as `{A,A}` which require that A and A be equal must check for that equality in a guard. We identify a label ℓ with a node in the abstract syntax tree of the program, and write $\ell : e$ to mean that e is the expression that occurs at ℓ's node. In listings of CORE programs, we let `%` mark the start of a line of comment.

Our language CORE differs from COREERLANG in several important ways:

- We remove support for
  - the throwing and catching of exceptions. The `fail` expression can be thought of as throwing an uncaught, untyped exception, and causes the running process to halt.
  - numbers, characters, strings and binaries, and by extension any string manipulation, arithmetic or binary processing.
  - timeouts in `receive` expressions. All `receive` expressions behave as if they had a timeout of `'infinity'`.
- We remove all but the minimum support for guards in `case` and `receive` expressions. We support only guards that test for equality.

$$
\begin{aligned}
m \in \mathbb{M} &::= \texttt{module}\ atom_0\ \texttt{[}atom_1\ \texttt{=}\ U_1\texttt{,}\ \ \ldots\texttt{,}\ \ atom_m\ \texttt{=}\ U_m\texttt{]}\ defn_1 \ldots defn_n\ \texttt{end} \\
defn &::= U\ \texttt{=}\ \ell_0 : \texttt{fun}\ \texttt{(}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{)}\ \texttt{->}\ \ell_1 : e_1 \\
e, t \in \mathbb{E} &::= U \mid atom \mid \texttt{[]} \mid \texttt{<}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{>} \mid \texttt{\{}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{\}} \mid \texttt{[}U\texttt{|}V\texttt{]} \\
&\quad \mid \texttt{fun}\ \texttt{(}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{)}\ \texttt{->}\ \ell : t \mid \texttt{primop}\ atom\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)} \\
&\quad \mid \texttt{apply}\ U\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)} \mid \texttt{call}\ U_0\ \texttt{:}\ U_1\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)} \\
&\quad \mid \texttt{let}\ \texttt{<}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{>}\ \texttt{=}\ \ell_0 : e_0\ \texttt{in}\ \ell_1 : e_1 \mid \texttt{letrec}\ defn_1 \ldots defn_n\ \texttt{in}\ e \\
&\quad \mid \texttt{case}\ \texttt{<}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{>}\ \texttt{of} \\
&\qquad\qquad \texttt{<}pat_{11}\texttt{,}\ldots\texttt{,}pat_{1n}\texttt{>}\ \texttt{when}\ guard_1\ \texttt{->}\ \ell_1 : e_1 \\
&\qquad\qquad\ \ \vdots \\
&\qquad\qquad \texttt{<}pat_{m1}\texttt{,}\ldots\texttt{,}pat_{mn}\texttt{>}\ \texttt{when}\ guard_m\ \texttt{->}\ \ell_m : e_m \\
&\qquad \texttt{end} \\
&\quad \mid \texttt{receive} \\
&\qquad\qquad \texttt{<}pat_1\texttt{>}\ \texttt{when}\ guard_1\ \texttt{->}\ \ell_1 : e_1 \\
&\qquad\qquad\ \ \vdots \\
&\qquad\qquad \texttt{<}pat_m\texttt{>}\ \texttt{when}\ guard_m\ \texttt{->}\ \ell_m : e_m \\
&\qquad \texttt{end} \\
&\quad \mid \texttt{do}\ \ell_1 : e_1\ \ell_2 : e_2 \mid \texttt{fail} \\
pat &::= U \mid U\ \texttt{=}\ pat \mid atom \mid \texttt{[]} \mid \texttt{\{}pat_1\texttt{,}\ldots\texttt{,}pat_n\texttt{\}} \mid \texttt{[}pat_1\texttt{|}pat_2\texttt{]} \\
guard &::= atom \mid U\ \texttt{=:=}\ V \mid guard_1\ \texttt{\&\&}\ guard_2
\end{aligned}
$$

Figure 1: Specification for the syntax of Core.

- We enforce that the constructors `<_,...,_>`, `{_,...,_}` and `[_|_]` are applied only to names. This allows the easy definition of a closure as a member of $\mathcal{L} \times Env$. By lucky coincidence, this also prevents the nesting of valuelists.
- We enforce that `case`, `apply`, `call` and `primop` expressions are applied only to names. This reduces the number and kind of continuations required in the concrete/abstract semantics.

The syntax chosen for Core is convenient for verification, but sometimes difficult to read. To lighten the load on the reader, we introduce the following convenient shorthand: when $\ell : a$ is a labelled expression that is not a name or a valuelist that occurs as a subexpression of $\ell' : e$ where a name is syntactically expected, then e is shorthand for

$$\ell^\star : \texttt{let}\ U^\star\ \texttt{=}\ \ell : a\ \texttt{in}\ \ell' : e[V^\star/\ell : a]$$

where $V^\star$ is a fresh name and $\ell^\star$ is a fresh label. For example,

$$
\begin{aligned}
& \ell_0 : \texttt{\{}\ell_1 : \texttt{'a'},\ U,\ \ell_2 : \texttt{primop}\ \texttt{'self'/0}\ \texttt{()}\texttt{\}} \\
\text{is short for } & \ell^\star : \texttt{let}\ V^\star\ \texttt{=}\ \ell_1 : \texttt{'a'}\ \texttt{in} \\
& \ell_0 : \texttt{\{}V^\star,\ U,\ \ell_2 : \texttt{primop}\ \texttt{'self'/0}\ \texttt{()}\texttt{\}} \\
\text{is short for } & \ell^\star : \texttt{let}\ V^\star\ \texttt{=}\ \ell_1 : \texttt{'a'}\ \texttt{in} \\
& \ell^{\star\star} : \texttt{let}\ V^{\star\star}\ \texttt{=}\ \ell_2 : \texttt{primop}\ \texttt{'self'/0}\ \texttt{()}\ \texttt{in} \\
& \ell_0 : \texttt{\{}V^\star,\ U,\ V^{\star\star}\texttt{\}}
\end{aligned}
$$

and the final one of these is a valid Core expression. We often omit labels where they are not important.

Semantics

The full formal semantics for Core is given in section 3, but in the interest of making clear the behaviour of Core, we give a sketch of a semantics here.

An environment ρ is a finite partial function from names to closures. A stack frame is one of $\textsc{Let}\langle\vec{U},\rho,e\rangle$ or $\textsc{Do}\langle\rho,e\rangle$, where $\vec{U}\in\mathbb{V}^*$. A state in the computation of a $\textsc{Core}$ program $\mathcal{M}$ can be thought of as a set Π of processes running in parallel. A process $\langle e,\rho,s\rangle^{\iota}_{\mathfrak{m}}$ evaluates an expression e in an environment ρ with a continuation stack s (a finite sequence of stack frames) and a mailbox $\mathfrak{m}$ containing unconsumed messages. ι is the process identifier (pid) of the process, and each process in Π is distinguished by its pid. Purely functional computations performed by each process are interleaved. Letting v range over process ids and the irreducible expressions U, *atom*, `[]`, `<`U_1`, ..., `U_n`>`, `{`U_1`, ..., `U_n`}`, `[`U`|`V`]` and `fun (`U`_1, ..., `U`_n) -> ` e, we give evaluation semantics thus:

- An `apply` U `(`V_1`, ..., `V_n`)` expression, when ρ closes U to an n-ary function, applies that function to the values to which ρ closes $V_1,\ldots,V_n$ and continues by executing its body: when $\rho(U)=($`fun (`V_1'`, ..., `V_n`) -> ` $e,\rho')$,

$$\langle \texttt{apply}\ U\ \texttt{(}V_1\texttt{, ..., }V_n\texttt{)},\rho,s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi\longrightarrow\langle e,\rho[V_i'\mapsto\rho(V_i)],s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi$$

- A `call` U_1 `:` U_2 `(`V_1`, ..., `V_n`)` expression, when ρ closes U_1 and U_2 to atoms, when the function exported by the module U_1 at U_2 is n-ary, applies that exported function to the values to which ρ closes $V_1,\ldots,V_n$: when $\rho(U_1)=(atom_1,\_)$, $\rho(U_2)=(atom_2,\_)$, and there is a module m in $\mathcal{M}$ of the form

  `module` $atom_1$ `[...,` $atom_2$ `=` W`, ...] ...` W `= fun (`V_1'`, ..., `V_n'`) ->` e `... end`

  let ρ' be the initial environment for a module, the smallest environment such that for each definition W `=` e in m, $\rho(W)=(e,\rho')$. Then

$$\langle \texttt{call}\ U_1\ \texttt{:}\ U_2\ \texttt{(}V_1\texttt{, ..., }V_n\texttt{)},\rho,s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi\longrightarrow\langle e,\rho'[V_i'\mapsto\rho(V_i)],s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi$$

- A `case <`$\vec{U}$`> of <`$pats_1$`> when` $guard_1$ `->` e_1 `...<`$pats_m$`> when` $guard_m$ `->` e_m `end` checks its head $\vec{U}$ against each of its clauses in ascending order. When it encounters the first i for which $\vec{U}$ matches pat_i with substitution θ and satisfies $guard_i$, it continues by executing e_i with in the environment extended with the substitution:

$$\langle \texttt{case ... end},\rho,s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi\longrightarrow\langle e_i,\rho\theta,s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi$$

- A `letrec` U_1 `=` e_1 `...` U_n `=` e_n `in` e' expression continues by executing e' with environment ρ', where ρ' is the smallest environment containing ρ such that for each U_i `=` e_i, $\rho'(U_i)=(e_i,\rho')$. Note that the only expressions that can appear on the right hand side of definitions in `letrec` expressions are of the form `fun (_,...,_) -> _`.

$$\langle \texttt{letrec}\ U_1\ \texttt{=}\ e_1\ \texttt{...}\ U_n\ \texttt{=}\ e_n\ \texttt{in}\ e',\rho,s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi\longrightarrow\langle e',\rho',s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi$$

- A `let <`U_1`, ..., `U_n`> =` e `in` e' expression pushes a $\textsc{Let}$ frame onto the stack and continues by executing e:

$$\langle \texttt{let <}U_1\texttt{, ..., }U_n\texttt{> = }e\ \texttt{in}\ e',\rho,s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi\longrightarrow\langle e,\rho,\textsc{Let}\langle U_1\cdot\ldots\cdot U_n,\rho,e'\rangle\cdot s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi$$

- A `do` e e' expression pushes a $\textsc{Do}$ frame onto the stack and continues by executing e:

$$\langle \texttt{do}\ e\ e',\rho,s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi\longrightarrow\langle e,\rho,\textsc{Do}\langle\rho,e'\rangle\cdot s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi$$

- If a process is executing some value v, it inspects its stack. If the stack is empty, the process terminates. Otherwise:

  – if the stack is $\textsc{Do}\langle\rho',e\rangle\cdot s$, the process continues by executing e with environment ρ and stack s.

$$\langle v,\rho,\textsc{Do}\langle\rho',e\rangle\cdot s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi\longrightarrow\langle e,\rho',s\rangle^{\iota}_{\mathfrak{m}}\parallel\Pi$$

  – if the stack is $\textsc{Let}\langle\vec{U},\rho,\ell\rangle\cdot s$, the process continues by executing e with the environment that results from assigning v to $\vec{U}$ in ρ' and stack s.

$$\begin{aligned}&\langle v = \texttt{<}U_1\texttt{,}\ \ \ldots\texttt{,}\ \ U_n\texttt{>}, \rho, \textsc{Let}\langle V_1\cdot\ldots\cdot V_n, \rho', e\rangle\cdot s\rangle^{\iota}_{\mathtt{m}} \parallel \Pi\\ &\qquad\longrightarrow \langle e, \rho'[V_i \mapsto \rho(U_i)], s\rangle^{\iota}_{\mathtt{m}} \parallel \Pi\\ &\langle v \neq \texttt{<\_,}\ \ \ldots\texttt{,}\ \ \texttt{\_>}, \rho, \textsc{Let}\langle V, \rho', e\rangle\cdot s\rangle^{\iota}_{\mathtt{m}} \parallel \Pi\\ &\qquad\longrightarrow \langle e, \rho'[V \mapsto (v,\rho)], s\rangle^{\iota}_{\mathtt{m}} \parallel \Pi\end{aligned}$$

- The `'self'/0` primop returns the pid under which the process is running:

$$\langle\texttt{primop 'self'/0 ()}, \rho, s\rangle^{\iota}_{\mathtt{m}} \parallel \Pi \longrightarrow \langle\iota, \rho, s\rangle^{\iota}_{\mathtt{m}} \parallel \Pi$$

- The `'spawn'/1` primop, when passed a nullary function, evaluates to a fresh pid ι' and creates a new process running the body of that function with pid ι': When $\rho(U) = (\texttt{fun () ->}\ e, \rho')$,

$$\langle\texttt{primop 'spawn'/1 (}U\texttt{)}, \rho, s\rangle^{\iota}_{\mathtt{m}} \parallel \Pi \longrightarrow \langle\iota', \rho, s\rangle^{\iota}_{\mathtt{m}} \parallel \langle e, \rho', \epsilon\rangle^{\iota'}_{\epsilon} \parallel \Pi$$

- The `'send'/2` primop, when passed a pid ι' as its first argument, evaluates to its second argument and appends its second argument to the mailbox of the process with pid ι': When $\rho(U) = (\iota, \_)$, $\rho(V) = (v, \rho'')$,

$$\langle\texttt{primop 'send'/2 (}U\texttt{,}\ V\texttt{)}, \rho, s\rangle^{\iota}_{\mathtt{m}} \parallel \langle e, \rho', s'\rangle^{\iota'}_{\mathtt{m}} \parallel \Pi \longrightarrow \langle v, \rho'', s\rangle^{\iota}_{\mathtt{m}} \parallel \langle e, \rho', s'\rangle^{\iota'}_{\mathtt{m}\cdot(v,\rho'')}$$

- A `receive` `<`pat_1`>` `when` $guard_1$ `->` e_1 $\ldots$`<`pat_m`>` `when` $guard_m$ `->` e_m `end` blocks if the process' mailbox contains no messages that match any of the clauses. Otherwise, it acts like a `case` expression whose head is the first message c in its mailbox that matches a clause in its body with substitution θ with the additional side effect of consuming the message from the mailbox:

$$\langle\texttt{receive}\ \ldots\ \texttt{end}, \rho, s\rangle^{\iota}_{\mathtt{m}\cdot v\cdot\mathtt{m}'} \parallel \Pi \longrightarrow \langle e_i, \rho\theta, s\rangle^{\iota}_{\mathtt{m}\cdot\mathtt{m}'} \parallel \Pi$$

- A `fail` expression causes the process running it to terminate. A process that terminates by executing `fail` is said to have failed.

- If a process encounters a malformed expression (application of an n-ary function to m arguments where $n \neq m$, a case expression where the head does not match any clause, etc), the process transitions to `fail`.

Remark. Note that sending is *non-blocking*, and that mailboxes are not *First-In-First-Out* (FIFO) but *First-In-First-Fireable-Out*. Incoming messages are queued at the end of the mailbox, but the message extracted by a receive construct is not necessarily the first.

Example 2.11 (publisher/subscriber). Listing 1 depicts a Core program that models a publisher/subscriber interaction. The program has two modules, `'publisher'` and `'main'`.

The `'publisher'` module implements a publisher. A publisher acts a go-between for processes that want to publish messages on a given topic and other processes that subscribe to messages on that topic. The publishing processes needn't know the identity or number of processes to whom the message will be forwarded. When a process wants to subscribe to messages on a topic T, it sends the publisher a `{'sub',` T`,` P`}` message, where P is the subscriber's pid. When a process wants to publish a message on topic T, it sends the publisher a `{'pub',` T`,` M`}`, where M is the message to be published. The publisher forwards this message to every subscriber for that topic.

The public face of a publisher is a process running the `Router` function, which we will call the router. The first argument to this function is an (arbitrarily long) list of pids of processes running the `Dispatcher` function, which we call dispatchers. When the router receives a `{'sub',` T`,` P`}` message, it spawns a new dispatcher responsible for forwarding messages on topic T to the process with pid P and then loops, adding the new dispatcher to its list. When it receives a `{'pub',` T`,` M`}` message, it forwards that message to every dispatcher. Note that it does using using `Map`, a higher-order function.

```
module 'publisher' ['new'/0 = New]
  % Spawns a new publisher.
  New = fun () →
    primop 'spawn'/1
      (fun () → apply Router ([]))

  % Creates dispatchers and forwards
  % them messages.
  Router = fun (Dispatchers) →
    receive
      <{'sub', Topic, P}> when 'true' →
        let <F> = fun () →
          apply Dispatcher (Topic, P)
        in
        let <D> = primop 'spawn'/1 (F) in
        apply Router ([D | Dispatchers])

      <M = {'pub', _, _}> when 'true' →
        let <G> = fun (Pid) →
          primop 'send'/2 (Pid, M)
        in
        do apply Map (G, Dispatchers)
        apply Router (Dispatchers)
    end

  % Dispatches messages if they have
  % the correct topic.
  Dispatcher = fun (Topic, P) →
    receive
      <M = {'pub', Topic_, _}>
      when Topic =:= Topic_ →
        do primop 'send'/2 (P, M)
        apply Dispatcher (Topic, P)
      <_> when 'true' →
        apply Dispatcher (Topic, P)
    end

  % Maps F across L.
  Map = fun (F, L) →
    case <L> of
      <[]> when 'true' → []
      <[X | Xs]> when 'true' →
        let <Y> = apply F (X) in
        let <Ys> = apply Map (F, Xs) in
        [Y | Ys]
    end
end
```

```
module 'main' ['main'/0 = Main]
  % Errors if it receives a message on
  % the wrong topic.
  Picky = fun (Topic) →
    receive
      <{'pub', Topic_, _}>
      when Topic =:= Topic_ →
        apply Picky (Topic)
      <_> when 'true' →
        fail
    end

  % Create a new publisher, then spawns
  % infinitely many picky subscribers,
  % sending a message each time.
  Main = fun () →
    let <P> =
      call 'publisher' : 'new'/0 ()
    in
    apply SpawnPickies (P)

  % Nondeterministically selects a
  % topic (either 'a' or 'b').
  % Spawns a picky subscriber for
  % that topic, then sends a message
  % on that topic.
  SpawnPickies = fun (P) →
    let <Topic> =
      primop 'choice'/2 ('a', 'b')
    in
    let <Q> =
      primop 'spawn'/1
        (fun () → apply Picky (Topic))
    in
    do primop 'send'/2
      (P, {'sub', Topic, Q})
    do primop 'send'/2
      (P, {'pub', Topic, 'm'})
    apply SpawnPickies (P)
end
```

Listing 1: A Core program that implements a publisher / subscriber interaction.

A dispatcher filters the messages it receives for relevancy before they are forwarded to a subscriber. Note the use of the guard `Topic =:= Topic_` [1]. Erlang is unusual in that it allows us to inspect the data in messages before they leave the mailbox. If the equality check `Topic =:= Topic_` fails, the message is returned to the mailbox and the next clause/message is checked.

The `'main'` module sets up a system of publishers and subscribers. The subscribers are processes running the function `Picky`, which loops if receives a `{'pub',` T`,` M`}` message on the topic T that it is expecting, or fails if it receives any other message. The `Main` function creates a publisher using an inter-module call expression, then calls `SpawnPickies`. We will see later that Core supports fully dynamic inter-module calls and behaviours using a callback module; both the module to call and the function to call on that module can be varied at run-time.

The `SpawnPickies` function starts by calling a new primop, `'choice'/2`. Occasionally we may introduce new simple primitive operators like this one, usually to provide some nondeterminicity. The `'choice'` family is one of these. `'choice'/`n is an n-ary operator that reduces nondeterministically to any one of its arguments. `SpawnPickies` continues by spawning a picky subscriber for the chosen topic, subscribing to to messages on that topic and sending a message on that topic. It then loops.

An interesting correctness property of this code is that it is error-free, or more precisely that each subscriber receives only messages on the topic to which it subscribes.

2.4 Actor Communicating System (ACS)

We now present the Actor Communicating System (ACS) model, first introduced in [9]. ACS is an abstraction for actor-model programs with an infinite state semantics for which reachability and coverability are decidable.

Definition 2.12 (Actor Communicating System)**.** An Actor Communicating System (ACS) is a sextuple $(P, Q, M, R, \iota_0, q_0)$ where P is a finite set of pid-classes, Q is a finite set of control states, M is a finite set of messages, $\iota_0 \in P$ is the pid-class of the initial process, $q_0 \in Q$ is the initial state of the initial process and R is a finite set of rules of the form $\iota : q \xrightarrow{\lambda} q'$ where $\iota \in P$, $q, q' \in Q$ and λ a label in one of four possible forms:

- τ: a process with pid-class ι performs an internal transition from q to q'
- $?m$ where $m \in M$: a process with pid-class ι transitions from q to q' by consuming a message of type m from its mailbox.
- $\iota'!m$ where $\iota' \in P$, $m \in M$: a process with pid-class ι transitions from q to q', sending a message of type m to a process of pid-class ι.
- $\nu\iota'.q''$ where $\iota' \in P$, $q'' \in Q$: a process with pid-class ι transitions from q to q' and spawns a process with pid-class ι' in control state q''.

We can think of an ACS as an abstraction for actor model programs like those of Core. The control states play the role of the triple of expression, environment and stack, with the caveat that there can be only finitely many of them. M is an abstraction on data. In Core there are an infinite number of messages that can be sent; consider the function

```
F = fun (P, M) → let <M_> = M in do primop 'send'/2 (P, M_) apply F (P, M_)
```

so `apply F (P, 'atom')` sends the messages `'atom'`, `{'atom'}`, `{{'atom'}}`, ... to the pid bound to P. In VAS these must be abstracted to some finite set of messages.

The pid-classes are an abstraction on pids. ACS can track how many processes of each pid-class are in each control state, but cannot distinguish between them during send or receive procedures. A message sent to a process whose pid is in pid-class ι might arrive in the mailbox of *any* process whose pid is in ι. This is the main source of imprecision that we try to correct for in the πActor model.

[1] Erlang has two equality operators, `==` and `=:=`. `==` ignores types when comparing numbers, but `=:=` does not, so `1 == 1.0` but `1 =/= 1.0`. Though in Core we ignore numbers and arithmetic, our intention is that clause guards should respect types when checking if two terms are equal, and so we use the symbol `=:=`.

ACS Rule r	**VAS Rule r**
$\iota : q \xrightarrow{\tau} q'$	$[(\iota, q) \to -1, (\iota, q') \to +1]$
$\iota : q \xrightarrow{?m} q'$	$[(\iota, q) \to -1, (\iota, q') \to +1, (\iota, m) \mapsto -1]$
$\iota : q \xrightarrow{\iota' ! m} q'$	$[(\iota, q) \to -1, (\iota, q') \to +1, (\iota', m) \mapsto +1]$
$\iota : q \xrightarrow{\nu\iota'.q''} q'$	$[(\iota, q) \to -1, (\iota, q') \to (\iota', q'') \mapsto +1]$

Table 1: The transformation $r \mapsto \mathbf{r}$.

The semantics we will give for ACS makes liberal use of counter abstractions. Mailboxes in ACS' semantics discard the ordering of messages, but keep a counter for the number of occurrences of each message in the mailbox. Instead of tracking the execution state of each process, the ACS keeps a counter for the number of processes of each pid in each control state.

Note that ACS is an over-approximation. Consider an ACS with a pid-class ι, three control states q, q' and q'', and two rules $\iota : q \xrightarrow{?a} q'$ and $\iota : q \xrightarrow{?b} q''$. With a FIFFO mailbox, a process in pid-class ι with a mailbox $c \cdot a \cdot b$ can only transition to q'': although both a and b can be fired, a appears before b in the mailbox. In the semantics we will give, the mailbox is abstracted to the counters $[a \mapsto 1, b \mapsto 1, c \mapsto 1]$, with no information about the ordering. So in our semantics, the process can transition nondeterministically to either q or q''. The traces in the FIFFO semantics remain, but an additional spurious traces appear.

The semantics for ACS is given in terms of a Vector Addition System (VAS, also known as a Petri net).

Definition 2.13 (Vector Addition System)**.** A Vector Addition System $\mathcal{V}$ is a pair (I, R) where I is a finite set of indices (called the 'places' of the VAS) and $R \subseteq \mathbb{Z}^I$ is a finite set of rules. A rule is just a vector of integers of dimension $|I|$ indexed by the elements of I.

A VAS $\mathcal{V} = (I, R)$ induces a transition system $[\![\mathcal{V}]\!]$ $(\mathbb{N}^I, \curvearrowright, v_0)$ where $\curvearrowright = \{(v, v + r) : v \in \mathbb{N}^I, r \in R, v + r \in \mathbb{N}^I\}$. We write $v \leq v'$ when for all i in I, $v(i) \leq v'(i)$.

Theorem 2.14. *The coverability problem is decidable for VAS.*[14]

Definition 2.15 (semantics for ACS)**.** The semantics of an ACS $\mathcal{A} = (P, Q, M, R, \iota_0, q_0)$ is the transition system induced by the VAS $\mathcal{V} = (I, R)$ where $I = P \times (Q \uplus M)$ and $\mathbf{R} = \{\mathbf{r} : r \in R\}$ where the transformation $r \mapsto \setminus$ is given in table 1.

Given a state $v \in [\![V]\!]$, the components $v(\iota, q)$ counts the number of processes in the pid-class ι in state q and the component $v(\iota, m)$ counts the number of messages of type m in the (joint) mailbox for pid-class ι.

Since coverability is decidable for the VAS semantics, we can check any safety property that is expressible via coverability on the ACS including

- unreachability of error statements, i.e. $v(\iota, q_{\text{err}}) = 1$ is uncoverable.
- mutual exclusion, i.e. $v(\iota, q_{\text{critical}}) = 2$ is uncoverable.
- boundedness of mailboxes, i.e. states with $\sum_{m \in m} v(\iota, m) = k$ are uncoverable.

2.5 πActor

We turn now to πActor, which we will use as the target for our abstraction. πActor is a variant of the polyadic π-calculus[19], which in turn is an extension of Milner's CCS[2]. Its syntax is influenced by that of the mailbox calculus[6].

Definition 2.16 (πActor)**.** Fix an infinite set of names $\mathfrak{N}$ ranged over by a, b, c, s, t, x, y and a set of set names $\mathfrak{S}$ ranged over by A, B, C. Fix also a set of message tags $\mathfrak{M}$ ranged over by $\mathtt{m}, \mathtt{n}$ and a set of process names $\mathfrak{P}$ ranged over by Proc. Let the set of πActor terms $\mathfrak{T}$ ranged over by P, Q, R be given by the grammar in fig. 2

$$\begin{aligned}
P, Q, R \in \mathfrak{T} &::= \mathbf{0} \mid \textbf{fail} \mid s \in A \mid P \parallel Q \mid M \mid \textsc{Proc}[\, s_1, \dots, s_n \mid A_1, \dots, A_m \,] \mid \boldsymbol{\nu} s.P \mid \boldsymbol{\nu} A.P \\
M, N &::= \pi.P \mid [X_1, \dots, X_m].P \mid M + N \\
\pi &::= \tau \mid \textbf{let } s \in A \mid \textbf{let } A = B \cup C \mid s?\mathtt{m}(A_1, \dots, A_n)[X_1, \dots, X_m] \mid s!\mathtt{m}\langle A_1, \dots, A_n \rangle \\
X &::= A \cap B \neq \emptyset
\end{aligned}$$

Figure 2: Specification for the syntax of πActor.

A πActor program is a pair (P, Δ) where $P \in \mathfrak{T}$ and Δ a finite set of process definitions of the form

$$\textsc{Proc}_i[\, s_1, \dots, s_n \mid A_1, \dots, A_m \,] := M_i$$

such that there are no free set names in P or any definition in Δ and for every term of the form $\textsc{Proc}[\, s_1, \dots, s_n \mid A_1, \dots, A_m \,]$ in P or any M_i, there is exactly one process definition of the form $\textsc{Proc}[\, x_1, \dots, x_n \mid B_1, \dots, B_m \,] = N$ in Δ.

When $m = 0$, we write $\textsc{Proc}[\, s_1, \dots, s_n \mid A_1, \dots, A_m \,]$ as $\textsc{Proc}[\, s_1, \dots, s_n \,]$ and $s?\mathtt{m}(A_1, \dots, A_n)[X_1, \dots, X_m]$ as $s?\mathtt{m}(A_1, \dots, A_n)$. When $I = i_1, \dots, i_n$, we write $\prod_{i \in I} P_i$ for $P_{i_1} \parallel \dots \parallel P_{i_n}$ and $\sum_{i \in I} \pi_i.P_i$ for $\pi_{i_1}.P_{i_1} + \dots + \pi_{i_n}.P_{i_n}$. Terms of the form M are called 'sequential' terms. When

$$\vec{s} = s_1 \cdot \dots \cdot s_n, \qquad \vec{A} = A_1 \cdot \dots \cdot A_m$$

we may write $\boldsymbol{\nu}\vec{s}$, $\boldsymbol{\nu}\vec{A}$, $\textsc{Proc}[\, \vec{s} \mid \vec{A} \,]$, $s!\mathtt{m}\langle \vec{S} \rangle$ or $s?\mathtt{m}(\vec{A})[X_1, \dots, X_l]$. These should be taken as shorthands for $\boldsymbol{\nu} s_1. \cdots .\boldsymbol{\nu} s_n$, $\boldsymbol{\nu} A_1. \cdots .\boldsymbol{\nu} A_m$, $\textsc{Proc}[\, s_1, \dots, s_n \mid A_1, \dots, A_n \,]$, $s!\mathtt{m}\langle A_1, \dots, A_m \rangle$ and $s?\mathtt{m}(A_1, \dots, A_m)[X_1, \dots, X_l]$ respectively. In restrictions, we use juxtapositions for the overloaded append/concatenate $\cdot$.

We may write a set $\{x_1, \dots, x_n\}$ in place of a set name A. A term $P = \textsc{Proc}[\, \vec{s} \mid \dots, \{x_1, \dots, x_n\}, \dots]$ is shorthand for $\boldsymbol{\nu} A.(\textsc{Proc}[\, \vec{s} \mid \dots, A, \dots \,] \parallel \prod_i x_i \in A)$, where $A \notin \mathrm{fv}(P)$.

Definition 2.17 (bound and free names)**.** The name s is bound in $\boldsymbol{\nu} s.P$, **let** $s \in S.P$ and $\textsc{Proc}[\dots, s, \dots \mid \dots] := M$. The set name A is bound in $\boldsymbol{\nu} A.P$, **let** $A = B \cup C.P$, $s?\mathtt{m}(\dots, A, \dots)[\dots].P$ and $\textsc{Proc}[\dots \mid \dots, A, \dots] := M$. A name is free if it is not bound, and we write $\mathrm{fv}(P)$ for the set of free names of P.

Definition 2.18 (normalisation)**.** A set of definitions Δ is said to be in normalised if every definition

$$\textsc{Proc}[\, s_1, \dots, s_n \mid A_1, \dots, A_m \,] := \sum_{i \in I} \pi_i.P_i$$

is such that each P_i contains no sums. A process $P \in \mathfrak{T}$ is said to be normalised if it contains no sums.

Remark 2.18.1. Every process and set of definitions can be normalised while remaining semantically equivalent. See fig. 3 for an example. From now on we consider only normalised πActor programs.

Semantics

Intuitively, a sequential process acts like a thread running finite-control sequential code. The prefixes have the following meanings:

- τ represents an internal transition.
- $[S_1 \cap T_1 \neq \emptyset, \dots, S_n \cap T_n \neq \emptyset]$ represents a check against each of the guards $S_i \cap T_i \neq \emptyset$. If any guard fails, the process cannot continue.
- $s!\mathtt{m}\langle A_1, \dots, A_n \rangle$ represents sending a message of type $\mathtt{m}$ containing name sets $A_1, \dots, A_n$ on the channel with name s.

$\boldsymbol{\nu} ss'.\boldsymbol{\nu} A.(\mathrm{P}[\,s \mid A\,] \parallel s' \in A$
$\quad \parallel \boldsymbol{\nu} s''.\boldsymbol{\nu} B.(s\,\mathtt{!m}\langle B\rangle.\mathbf{0} \parallel s'' \in B))$

$\boldsymbol{\nu} ss'.\boldsymbol{\nu} A.(\mathrm{P}[\,s \mid A\,] \parallel s' \in A$
$\quad \parallel \boldsymbol{\nu} s''.\boldsymbol{\nu} B.(\mathrm{Q}[\,s \mid B\,] \parallel s'' \in B))$

$\mathrm{P}[\,s \mid A\,] := s\mathtt{?m}(B).\mathbf{let}\ s' \in B.s'\mathtt{!n}\langle A\rangle.\mathbf{0}$

$\mathrm{P}[\,s \mid A\,] := s\mathtt{?m}(B).\mathrm{P}'[\,s \mid A, B\,]$
$\mathrm{P}'[\,s \mid A, B\,] := \mathbf{let}\ s' \in B.\mathrm{P}''[\,s, s' \mid A, B\,]$
$\mathrm{P}''[\,s, s' \mid A, B\,] := s'\mathtt{!m}\langle A\rangle.\mathbf{0}$
$\mathrm{Q}[\,s \mid B\,] := s\mathtt{!m}\langle B\rangle.\mathbf{0}$

Figure 3: The two programs above are equivalent. The program on the right is normalised.

- $s\mathtt{?m}(A_1, \ldots, A_n)[B_1 \cap C_1 \neq \emptyset, \ldots, B_m \cap C_m \neq \emptyset]$ represents receiving a message of type $\mathtt{m}$ containing name sets $A_1, \ldots, A_n$ on the channel with name s and binding each S_i to A_i, then checking the guards $B_i \cap C_i \neq \emptyset$. If any guard fails, the message is not received and the process cannot transition.
- **let** $s \in A$ represents nondeterministically selecting a name from S and binding it to s before continuing.

The term $\tau.(P \parallel Q)$ represents spawning a process P and continuing as Q, although the roles of P and Q are interchangeable.

Processes in the π-calculus communicate synchronously over named channels. If a process is trying to send a message, it cannot do so unless another process is ready and willing to receive it. A non-blocking send like the `'send'/2` primop of Core can be achieved by spawning a process that performs the blocking send. The names sent in name sets in messages can be used as the names of channels for later communication after they are extracted using **let**.

Restrictions $\boldsymbol{\nu} s$ are used to make channel names private. In the program

$$\boldsymbol{\nu} s.\,(s\mathtt{!m}\langle B\rangle.P \parallel s\mathtt{?m}(A).Q) \parallel s\mathtt{?m}(A).R,$$

$s\mathtt{!m}\langle B\rangle.p$ may synchronise with $s\mathtt{?m}(A).Q$ to produce $\boldsymbol{\nu} s.(P \parallel Q[S/A]) \parallel s\mathtt{?m}(A).R$, but it cannot synchronise with $s\mathtt{?m}(A).R$ since $s\mathtt{?m}(A).R$ falls outside of scope of the restriction $\boldsymbol{\nu} s$.

These notions in mind, we define structural congruence for πActor programs, which identifies syntactically different programs with semantically equivalent behaviour. We then use this structural congruence to define concisely the semantics of πActor.

Definition 2.19 (structural congruence)**.** Structural congruence $\overset{\Delta}{\equiv}$ for πActor programs with definitions Δ is the smallest congruence closed by α-conversion of bound names, commutativity and associativity of $\parallel$ and $+$ with 0 as the neutral element, commutativity and associativity of $\cup$ with $\emptyset$ as the neutral element, commutativity of $\cap$, and the following laws:

$$\boldsymbol{\nu} s.\mathbf{0} \overset{\Delta}{\equiv} \mathbf{0} \quad \boldsymbol{\nu} A.\mathbf{0} \overset{\Delta}{\equiv} \mathbf{0}$$

$$\boldsymbol{\nu} x.\boldsymbol{\nu} y.P \overset{\Delta}{\equiv} \boldsymbol{\nu} y.\boldsymbol{\nu} x.P \quad \boldsymbol{\nu} A.\boldsymbol{\nu} B.P \overset{\Delta}{\equiv} \boldsymbol{\nu} B.\boldsymbol{\nu} A.P \quad \boldsymbol{\nu} x.\boldsymbol{\nu} A.P \overset{\Delta}{\equiv} \boldsymbol{\nu} A.\boldsymbol{\nu} x.P$$

$$\text{if } s \notin \mathrm{fv}(P), \quad P \parallel \boldsymbol{\nu} s.Q \overset{\Delta}{\equiv} \boldsymbol{\nu} s.(P \parallel Q)$$

$$\text{if } A \notin \mathrm{fv}(P), \quad P \parallel \boldsymbol{\nu} A.Q \overset{\Delta}{\equiv} \boldsymbol{\nu} A.(P \parallel Q) \quad P \parallel s \in A \overset{\Delta}{\equiv} P$$

and if $\textsc{Proc}[\,y_1, \ldots, y_n \mid B_1, \ldots, B_m\,] := M \in \Delta$,

$$\textsc{Proc}[\,x_1, \ldots, x_n \mid A_1, \ldots, A_m\,] \overset{\Delta}{\equiv} M[x_1 \cdot \ldots \cdot x_n / y_1 \cdot \ldots \cdot y_n][A_1 \cdot \ldots \cdot A_m / A_1 \cdot \ldots \cdot A_m]$$

Remark 2.19.1. We omit Δ in $\overset{\Delta}{\equiv}$ when it is unimportant or obvious from context.

Definition 2.20 (standard form)**.** A π-calculus term P is said to be in standard form if it is in the form $\boldsymbol{\nu} s_1 \cdots s_n.\boldsymbol{\nu} A_1 \cdots A_m.(Q_1 \parallel \ldots \parallel Q_l)$ where $Q_1, \ldots, Q_l$ are non-$\mathbf{0}$ sequential processes, **fail** or some $s_i \in A$, each s_i occurs free in some Q_j, and each A_i occurs free in some Q_j. We write $\mathfrak{T}_{\mathrm{sf}}$ for the set of terms in standard form.

Theorem 2.21. *Every* πActor *term is structurally congruent to a term in standard form.*

Remark 2.21.1. Every term in a normalised program in standard form is of the form $\boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{A}.(Q_1 \parallel \cdots \parallel Q_n)$ where each Q_i is **fail**, $s_j \in A$ or of the form $\textsc{Proc}[\, s_1, \ldots, s_m \,|\, A_1, \ldots, A_l \,]$.

Definition 2.22 (normal form)**.** A term $P \in \mathfrak{T}$ is in normal form if it is normalised and in standard form. A definition set Δ is in normal form if it is normalised and each definition in Δ

$$\textsc{Proc}[\, s_1, \ldots, s_n \,|\, A_1, \ldots, A_m \,] := \sum_{i \in I} \pi_i.P_i$$

is such that each P_i is in standard form. A πActor program (P, Δ) if P and Δ are in normal form.

We write $M = \sum_{i \in I} \pi_i.P_i \in \mathfrak{T}_{\text{nf}}$ if each $P_i \in \mathfrak{T}_{\text{sf}}$.

Definition 2.23 (reduction semantics for πActor)**.** The reduction semantics for a normalised πActor program with definitions Δ in normal form is given by a transition system with transitions $P \overset{\Delta}{\twoheadrightarrow} Q$ if

$$\frac{P \overset{\Delta}{\equiv} \boldsymbol{\tau}.\boldsymbol{\nu}\vec{y}.\boldsymbol{\nu}\vec{C}.P' + M \in \mathfrak{T}_{\text{nf}}}{\boldsymbol{\nu}\vec{x}.\boldsymbol{\nu}\vec{B}.(P \parallel Q) \in \mathfrak{T}_{\text{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x}\vec{y}.\boldsymbol{\nu}\vec{B}\vec{C}.(P' \parallel Q)} \quad (\textsc{Tau})$$

$$\frac{P \overset{\Delta}{\equiv} \textbf{let } t \in A.\boldsymbol{\nu}\vec{y}.\boldsymbol{\nu}\vec{C}.P' + M \in \mathfrak{T}_{\text{nf}} \qquad Q = s \in A}{\boldsymbol{\nu}\vec{x}.\boldsymbol{\nu}\vec{B}.(P \parallel Q \parallel R) \in \mathfrak{T}_{\text{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x}\vec{y}.\boldsymbol{\nu}\vec{B}\vec{C}.(P'[s/t] \parallel Q \parallel R)} \quad (\textsc{Let})$$

$$\frac{\begin{array}{c} P \overset{\Delta}{\equiv} \textbf{let } A = B \cup C.\boldsymbol{\nu}\vec{y}.\boldsymbol{\nu}\vec{C}'.P' + M \in \mathfrak{T}_{\text{nf}} \qquad Q = \prod_{i \in I} s_i \in B \parallel \prod_{j \in J} t_j \in C \\ R \neq z \in B \parallel R' \qquad R \neq z \in C \parallel R' \qquad Q' = \prod_{i \in I} s_i \in A \parallel \prod_{j \in J} t_j \in A \end{array}}{\boldsymbol{\nu}\vec{x}.\vec{B}'.(P \parallel Q \parallel R) \in \mathfrak{T}_{\text{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x}\vec{y}.\boldsymbol{\nu}A\vec{B}'\vec{C}'.(P' \parallel Q \parallel Q' \parallel R)} \quad (\textsc{Union})$$

$$\frac{P \overset{\Delta}{\equiv} [A_1 \cap B_1 \neq \emptyset, \ldots, A_n \cap B_n \neq \emptyset].\boldsymbol{\nu}\vec{y}.P' + M \in \mathfrak{T}_{\text{nf}} \qquad Q = \prod_{i=1}^{n}(s_i \in A_i \parallel s_i \in B_i)}{\boldsymbol{\nu}\vec{x}.(P \parallel Q \parallel R) \in \mathfrak{T}_{\text{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x} \cdot \vec{y}.(P' \parallel Q \parallel R)} \quad (\textsc{Match})$$

$$\frac{\begin{array}{c} P \overset{\Delta}{\equiv} s\texttt{!m}\langle A_1, \ldots, A_n \rangle.\boldsymbol{\nu}\vec{y}.\boldsymbol{\nu}\vec{B}'.Q + M_P \in \mathfrak{T}_{\text{nf}} \\ P' \overset{\Delta}{\equiv} s\texttt{?m}(B_1, \ldots, B_n)[C_1 \cap C_1' \neq \emptyset, \ldots, C_m \cap C_m' \cap \neq \emptyset].\boldsymbol{\nu}\vec{z}.\boldsymbol{\nu}\vec{C}'.Q' + M_{P'} \in \mathfrak{T}_{\text{nf}} \\ R_1 = \prod_{i=1}^{n} \prod_{j \in J_i} t_j \in A_i \qquad R_3 = \prod_{i=1}^{n} \prod_{j \in J_i} t_j \in B_i \\ R_2 \neq a \in A_i \parallel R' \qquad R_1 \parallel R_2 \parallel R_3 = \prod_{i=1}^{m}(s_i \in C_i \parallel s_i \in C_i') \parallel R' \end{array}}{\boldsymbol{\nu}\vec{x}.\boldsymbol{\nu}\vec{A}'.(P \parallel P' \parallel R_1 \parallel R_2) \in \mathfrak{T}_{\text{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x}\vec{y}\vec{z}.\boldsymbol{\nu}B_1 \cdots B_n\vec{A}'\vec{B}'\vec{C}'.(Q \parallel Q' \parallel R_1 \parallel R_2 \parallel R_3)} \quad (\textsc{Sync})$$

$$\frac{P \overset{\Delta}{\equiv} P' \in \mathfrak{T}_{\text{sf}} \qquad P' \overset{\Delta}{\twoheadrightarrow} Q' \qquad Q' \overset{\Delta}{\equiv} Q}{P \overset{\Delta}{\twoheadrightarrow} Q} \quad (\textsc{Equiv})$$

Remark 2.23.1. We omit Δ in $\overset{\Delta}{\twoheadrightarrow}$ when it is unimportant or obvious from context.

Example 2.24. In shorter hand, we have $\boldsymbol{\tau}.P \twoheadrightarrow P$ and

- $\boldsymbol{\nu}A.(s\texttt{!m}\langle A \rangle.Q \parallel s' \in A) \parallel \boldsymbol{\nu}B.s\texttt{?m}(B).P \twoheadrightarrow \boldsymbol{\nu}A.(Q \parallel s' \in A) \parallel \boldsymbol{\nu}B.(P \parallel s' \in B)$.
- $[\{s\} \cap \{s, t\} \neq \emptyset].Q \equiv \boldsymbol{\nu}AB.([A \cap B \neq \emptyset].Q \parallel s \in A \parallel s \in B \parallel t \in B)$
  $\twoheadrightarrow \boldsymbol{\nu}AB.(Q \parallel s \in A \parallel s \in B \parallel t \in B)$.
- If $\mathrm{P}[\, s \,|\, A \,] := \textbf{let } s' \in A.\mathrm{P}'[\, s, s' \,|\, A \,]$, then $\mathrm{P}[\, s \,|\, \{s_1, \ldots, s_n\} \,] \twoheadrightarrow \mathrm{P}'[\, s, s_i \,|\, \{s_1, \ldots, s_n\} \,]$ for each $i = 1, \ldots, n$.

- If $\mathrm{P}[\,s \mid B, C\,] := \mathbf{let}\ A = B \cup C.\mathrm{P}'[\,s \mid A\,]$ and $S, T \in \wp(\mathfrak{N})$, then $\mathrm{P}[\,s \mid S, T\,] \twoheadrightarrow \mathrm{P}'[\,s \mid S \cup T\,]$.

We will often write $\textsc{Proc}[\,\vec{s} \mid \ldots, B \cup B', \ldots]$ as shorthand for $\boldsymbol{\nu} A = B \cup B'.\textsc{Proc}[\,\vec{s} \mid \ldots, A, \ldots]$ for some fresh A. When we have a rule like $\textsc{Proc}'[\,\vec{s} \mid A, A'\,] := \pi.\textsc{Proc}[\,\vec{s} \mid S \cup S'\,]$, we write for $S, S' \in \wp(\mathfrak{N})$ write $\textsc{Proc}'[\,\vec{s} \mid S, S'\,] \| Q \twoheadrightarrow \textsc{Proc}[\,\vec{s} \mid S \cup S'\,] ! Q'$ as if it were a one-step transition. This is a harmless abuse of notation that along with our set argument shorthand will greatly increase the legibility of πActor programs.

Coverability

We define a preorder on πActor terms using the term embedding relation. Informally, a term P is embedded in a term Q if Q is the parallel composition of P' with some other terms, where P' and P have the same structure but the set constructs in P' may be supersets of their corresponding set constructs in P.

Definition 2.25 (preorder for sequences $\sqsubseteq\ \subseteq S^* \times S^*$)**.** When $\leq$ is an ordering on S, then when $\vec{x}, \vec{y} \in S^*$, $\vec{x} = x_1, \ldots, x_n$, $\vec{y} = y_1, \ldots, y_m$, we have $\vec{x} \sqsubseteq \vec{y}$ iff $m = n$ and for $i = 1, \ldots, n$, $x_i \leq y_i$.

Definition 2.26 (term embedding $\sqsubseteq$)**.** Let $P, Q \in \mathfrak{T}$ and Δ a definition set such that (P, Δ) and (Q, Δ) are normalised πActor programs. Then $P \sqsubseteq Q$ if and only if

$$\begin{gathered} P \equiv P' \in \mathfrak{T}_{\mathrm{sf}}, \quad P' = \boldsymbol{\nu}\vec{x}.\boldsymbol{\nu}\vec{B}(P_1 \parallel \cdots \parallel P_n), \\ \text{and } Q \equiv Q' \in \mathfrak{T}_{\mathrm{sf}}, \quad Q' = \boldsymbol{\nu}\vec{x}\vec{y}.\boldsymbol{\nu}\vec{B}\vec{C}(P_1 \parallel \cdots \parallel P_n \parallel R). \end{gathered}$$

Remark 2.26.1. Equivalently, when $S, T \in \wp(\mathfrak{N})^*$,

$$\begin{gathered} P \equiv P' \in \mathfrak{T}_{\mathrm{sf}}, \quad P' = \boldsymbol{\nu}\vec{x}.(P_1 \parallel \cdots \parallel P_n), \\ \text{and } Q \equiv Q' \in \mathfrak{T}_{\mathrm{sf}}, \quad Q' = \boldsymbol{\nu}\vec{x}\vec{y}.(Q_1 \parallel \cdots \parallel Q_n \parallel R), \\ \text{and } P_i = \textsc{Proc}_i[\,\vec{z} \mid \vec{S}\,], \quad Q_i = \textsc{Proc}_i[\,\vec{z} \mid \vec{T}\,], \quad \vec{S} \sqsubseteq \vec{T} \quad \text{or } P_i = Q_i = \mathbf{fail}. \end{gathered}$$

(where we take $\subseteq$ as the ordering for sets).

This allows us to define coverability for πActor programs.

Definition 2.27 (coverability for πActor programs)**.** Given a πActor program (P, Δ) and a query state $Q \in \mathfrak{T}$, the coverability problem is to determine if there is a state $Q' \in \mathfrak{T}$ for which $P \xrightarrow{\Delta}\!\!\!\!\rightarrow Q'$ and $Q' \sqsupseteq Q$.

This notion of coverability allows us to express all of the safety properties that can be expressed using an *ACS* and more:

- unreachability of errors: **fail** is uncoverable.
- mutual exclusion:
  - if we simply wish that no two processes be executing the same definition / program location simultaneously, we can check the uncoverability of

    $$\textsc{Critical}[\,s_1, \ldots, s_n \mid \emptyset, \ldots, \emptyset\,] \parallel \textsc{Critical}[\,s'_1, \ldots, s'_n \mid \emptyset, \ldots, \emptyset\,]$$

  - if we want something more precise, say that no two processes are writing to the same resource at the same time, we can ask for more precision with respect to process definition arguments by checking the uncoverability of, say,

    $$\textsc{Write}[\,s \mid \{r\}\,] \parallel \textsc{Write}[\,s' \mid \{r\}\,]$$

- boundedness of mailboxes for individual processes: if we adopt the convention that each process has a unique name s that represents its mailbox, that processes only perform receives on their unique s, and that processes only use non-blocking sends ($\textsc{Send}_{\mathtt{m}}[\,s' \mid S_1, \ldots, S_m\,] \parallel P'$) where

  $$\textsc{Send}_{\mathtt{m}}[\,s \mid A_1, \ldots, A_m\,] := s\,\mathtt{!m}\langle A_1, \ldots, A_m \rangle.\mathbf{0},$$

we can then check that the number of messages of type `m` in the mailbox of any process is bounded above by k by checking the uncoverability of $\textsc{Send}_{\mathtt{m}}[\, s \mid \emptyset, \ldots, \emptyset \,]^k$, where P^i is the parallel composition of i copies of P. As for mutual exclusion, we can be more precise with respect to message contents (arguments to the process definition $\textsc{Send}_m$). If we would like to be more specific with regards to which mailbox is bounded, we can include the relevant processes in our uncoverability query: $\mathrm{P}[\, s, t_1, \ldots, t_n \mid S_1, \ldots, S_n \,] \parallel \textsc{Send}_{\mathtt{m}}[\, s \mid \emptyset, \ldots, \emptyset \,]^k$ checks the k-boundedness of messages of type m in the mailbox of processes that cover $\mathrm{P}[\, s, t_1, \ldots, t_n \mid S_1, \ldots, S_n \,]$.

Relationship with classical π-calculus

We have designed πActor with the aim to use it as an abstraction for the behaviour of Core programs. Our desire is that each send/receive in the abstraction should correspond to a send/receive in the original program, so the communication topology is not polluted by the need to represent control flow or data structures with process communication. However, we would also like to be able to represent programs that maintain unbounded data structures containing pids. To this end, we introduce sets to the calculus as method of collapsing multiple pids into a single class, in a similar manner to the ACS (but with more flexibility). Our slightly odd receive-with-match syntax gives us the ability to model Erlang style inspection of messages before receipt, without needing to use multiple π-calculus sends and receives to represent the process.

We realise that this results in a syntax and semantics that might feel significantly different to the classical π-calculus with which the reader may be more familiar, and one might reasonably ask how the two of them relate. πActor can be encoded into classical π-calculus, and vice-versa. The polyadic π-calculus with match has the syntax

$$\begin{aligned} P, Q &::= \mathbf{0} \mid M \mid P \parallel Q \mid \textsc{Proc}[\, s_1, \ldots s_n \,] \mid \boldsymbol{\nu} s.P \\ M, N &::= \pi.P \mid [x = y].P \mid [x \neq y].P \mid M + N \\ \pi &::= \tau \mid x(s_1, \ldots, s_n) \mid \bar{x}\langle s_1, \ldots, s_n \rangle \end{aligned}$$

and the reduction semantics $P \overset{\Delta}{\twoheadrightarrow} Q$ if

$$\frac{P \overset{\Delta}{\equiv} \boldsymbol{\tau}.\boldsymbol{\nu}\vec{y}.P' + M \in \mathfrak{T}'_{\mathrm{nf}}}{\boldsymbol{\nu}\vec{x}.(P \parallel Q) \in \mathfrak{T}'_{\mathrm{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x} \cdot \vec{y}.(P' \parallel Q)} \ (\textsc{Tau}') \qquad \frac{P \overset{\Delta}{\equiv} [X].\boldsymbol{\nu}\vec{y}.P' + M \in \mathfrak{T}'_{\mathrm{nf}} \qquad X}{\boldsymbol{\nu}\vec{x}.(P \parallel Q) \in \mathfrak{T}'_{\mathrm{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x} \cdot \vec{y}.(P' \parallel Q)} \ (\textsc{Match}')$$

$$\frac{P \overset{\Delta}{\equiv} \bar{s}\langle t_1, \ldots, t_n \rangle.\boldsymbol{\nu}\vec{y}.Q + M_P \in \mathfrak{T}'_{\mathrm{nf}} \qquad P' \overset{\Delta}{\equiv} s(a_1, \ldots, a_n).\boldsymbol{\nu}\vec{z}.Q' + M_{P'} \in \mathfrak{T}'_{\mathrm{nf}}}{\boldsymbol{\nu}\vec{x}.(P \parallel P' \parallel R) \in \mathfrak{T}'_{\mathrm{sf}} \overset{\Delta}{\twoheadrightarrow} \boldsymbol{\nu}\vec{x} \cdot \vec{y} \cdot \vec{z}.(Q \parallel Q'[t_1 \cdot \ldots \cdot t_n / a_1 \cdot \ldots \cdot a_n] \parallel R)} \ (\textsc{Sync}')$$

$$\frac{P \overset{\Delta}{\equiv} P' \in \mathfrak{T}'_{\mathrm{sf}} \qquad P' \overset{\Delta}{\twoheadrightarrow} Q' \qquad Q' \overset{\Delta}{\equiv} Q}{P \overset{\Delta}{\twoheadrightarrow} Q} \ (\textsc{Equiv}')$$

πActor can be encoded into the polyadic π-calculus with match[2] by treating sets as collections of processes. Introduce definitions

$$\begin{aligned} \textsc{Member}[\, x, s \,] &:= \bar{s}.\langle x \rangle.\textsc{Member}[\, x, s \,] \\ \textsc{Union}[\, s, s', t \,] &:= s(x).\bar{t}\langle x \rangle.\textsc{Union}[\, s, s', t \,] + s'(x).\bar{t}\langle x \rangle.\textsc{Union}[\, s, s', t \,] \\ \textsc{Intersect}[\, s, s', a \,] &:= s(x).s'(y).([x = y].\bar{z}\langle\rangle + [x \neq y].\textsc{Intersect}[\, s, s', a \,]) \end{aligned}$$

The selection prefix **let** $y \in S$ then becomes a receive $s(y)$, and the match $[S \cap T \neq \emptyset].P$ becomes $\boldsymbol{\nu} a.(\textsc{Intersect}[\, s, t, a \,] \parallel a().P)$.

[2] Though not in a way that preserves depth-boundedness.

3 An operational semantics for Core

In this section we give a concrete semantics for CORE programs. We use a time-stamped CESK* machine[3] following an approach by Van Horn and Might[20], and taking cues from [9]. This will result in an abstract interpretation that is both parametric and sound by construction.

Definition 3.1 (machine states)**.** Let a state in the semantics be $S = \langle \pi, \mu, \sigma_v, \sigma_k \rangle \in \mathit{State}$ with

$$\begin{aligned} S \in \mathit{State} &:= \mathit{Procs} \times \mathit{Mailboxes} \times \mathit{ValueStore} \times \mathit{KontStore} \\ \pi \in \mathit{Procs} &:= \mathit{Pid} \rightharpoonup \mathit{ProcState} \\ \mu \in \mathit{Mailboxes} &:= \mathit{Pid} \rightharpoonup \mathit{Mailbox} \\ \sigma_v \in \mathit{ValueStore} &:= (\mathit{ClosureAddr} \rightharpoonup \mathit{Closure}) \times (\mathit{PidAddr} \rightharpoonup \mathit{Pid}) \\ \sigma_k \in \mathit{KontStore} &:= \mathit{KontAddr} \rightharpoonup \mathit{Kont}. \end{aligned}$$

An element of *Procs* associates a process with pid ι with its local state $q = \langle f, \rho, c, t \rangle \in \mathit{ProcState}$ where

$$q \in \mathit{ProcState} := \mathit{Pid} \uplus \mathcal{L} \uplus \mathit{ValueAddr} \times \mathit{Env} \times \mathit{KontAddr} \times \mathit{Time}.$$

Its components are:

- the focus, f, which may be a pid, a value address or the label of some subterm of a program.
- an environment ρ which maps names to value addresses,

$$\rho \in \mathit{Env} := \mathbb{V} \rightharpoonup \mathit{ValueAddr}.$$

- a continuation address, which points to a continuation in the store. The continuation indicates what to evaluate next when the current evaluation produces a value.
- a time-stamp, representing the history of the process' computation up until this point.

An element of *Mailboxes* associates a process with pid ι with is mailbox. A mailbox is a finite sequence of value addresses:

$$\mathfrak{m} \in \mathit{Mailbox} := \mathit{ValueAddr}^*.$$

The mailbox is supported by the operations

$$\begin{aligned} \text{mmatch} : &(\mathit{pat} \times \mathit{guard})^* \times \mathit{Mailbox} \times \mathit{Env} \times \mathit{ValueStore} \\ &\to (\mathbb{N} \times \mathit{ValueAddr} \times (\mathbb{V} \rightharpoonup \mathit{Value}) \times \mathit{Mailbox})_\perp \\ \text{enq} : &\ \mathit{ValueAddr} \times \mathit{Mailbox} \to \mathit{Mailbox} \end{aligned}$$

mmatch takes a list of clauses (a clause is a pattern and a guard), a mailbox, the current environment and the current store and returns the index of the matching pattern, a substitution witnessing the match, and the mailbox resulting from the extraction of the matched message. enq enqueues a message into a mailbox. As we are modelling FIFFO mailboxes, we set $\text{enq}(a, \mathfrak{m}) = \mathfrak{m} \cdot a$ and $\text{mmatch}((p_1, g_1) \cdot \ldots \cdot (p_m, g_m), \mathfrak{m}, \rho, \sigma_v) = (i, a, \theta, \mathfrak{m}_1 \cdot \mathfrak{m}_2)$ when

$$\begin{gathered} \mathfrak{m} = \mathfrak{m}_1 \cdot a \cdot \mathfrak{m}_2, \quad \forall a' \in \mathfrak{m}_1 \forall j : \text{match}(p_j, g_j, a', \rho, \sigma_v) = \perp, \\ \theta = \text{match}(p_i, g_i, a, \rho, \sigma_v) \quad \forall j < i : \text{match}(p_j, g_j, a, \rho, \sigma_v) = \perp \end{gathered}$$

and $\perp$ otherwise, where $\text{match}(p, g, a, \rho, \sigma_v)$ seeks to match the value stored at a in σ_v against the pattern p producing witnessing substitution θ and checks that the guard g holds in $\rho\theta$ and σ_v.

An element of *ValueStore* is a partitioned store whose first part associates closure addresses with closures and whose second associates pid addresses with pids. Through a harmless abuse of notation, we let $\sigma_v(x)$ refer to the first partition when x is a closure address and the second when x is a pid address. The reason for this partitioning will become clear when we discuss the construction of addresses. An element of *KontStore* is a store that associates continuation addresses with continuations.

[3]CESK stands for Control, Environment, State and (K)ontinuation

Closures are pairs of program labels and environments. Closures include both closed lambdas (which is standard) and constructor applications. A value is either a closure or a pid.

$$\mathit{Closure} := \mathcal{L} \times \mathit{Env}, \qquad \mathit{Value} = \mathit{Closure} \uplus \mathit{Pid}$$

A continuation $\kappa \in \mathit{Kont}$ takes one of the following forms:

1. Stop, which represents ordinary termination.
2. $\textsc{Let}\langle U_1 \cdot \ldots \cdot U_n, \ell, \rho, c\rangle$, which represents the context $E[$`let <`U_1`, ..., `U_n`> = `$[]$ `in` $e']$ where ρ closes $\ell : e$ to e' and c is the address of the continuation representing the enclosing evaluation context E.
3. $\textsc{Do}\langle \ell, \rho, c\rangle$, which represents the context $E[$`do` $[]$ $e']$ where ρ closes $\ell : e$ to e' and c is the address of the continuation representing the enclosing evaluation context E.

Addresses, Pids and Timestamps

Though the machine supports arbitrary representations of time-stamps, addresses and pids, we follow the lead of [9] and present an instance based on contours. A contour is a string of program labels:

$$t \in \mathit{Time} := \mathcal{L}^*$$

The initial contour is the empty sequence ϵ, and the tick function updates the contour of the process by prepending the label under focus, which always labels an `apply` or `call` expression.

$$\mathrm{tick} : \mathcal{L} \times \mathit{Time} \to \mathit{Time}, \quad \mathrm{tick}(\ell, t) = \ell \cdot t$$

We pull a trick with value addresses that will be extremely useful when constructing the πActor model. We give pids and closures separate address spaces, and embed the pid addresses reachable from the environment of a closure into the address of that closure. Addresses for pids are represented by quadruples containing the pid of the storing process, the variable to which the address will be assigned, the time stamp at the time of the assignment and the pid stored at the address. Addresses for closures are represented by pentuples containing the pid of the storing process, the variables to which the address will be assigned, the time stamp at the time of the assignment, the data stored at the address and the set of pid addresses reachable from the closure. Continuation addresses are represented by pentuples containing the pid of the storing process, the label at the time of storing, the time stamp at the time of storing, the environment at time of storing and the set of pid addresses reachable from the continuation to be stored (through its environment / continuation address), or $\star$ which is the address of the initial continuation Stop.

$$\begin{aligned}
\mathit{PidAddr} &:= \mathit{Pid} \times \mathbb{V} \times \mathit{Time} \times \mathit{Pid} \\
\mathit{ClosureAddr} &:= \mathit{Pid} \times \mathbb{V} \times \mathit{Time} \times \mathit{Data} \times \wp(\mathit{PidAddr}) \\
a \in \mathit{ValueAddr} &:= \mathit{PidAddr} \uplus \mathit{ClosureAddr} \\
c \in \mathit{KontAddr} &:= \{\star\} \uplus (\mathit{Pid} \times \mathcal{L} \times \mathit{Time} \times \mathit{Env} \times \wp(\mathit{PidAddr}))
\end{aligned}$$

The data domain $d \in \mathit{Data}$ is the set of terms of the following grammar, representing Core data structures:

$$d \in \mathit{Data} ::= \iota \mid \mathit{atom} \mid \texttt{fun} \mid \texttt{[]} \mid \texttt{\{}d_1\texttt{, ..., }d_n\texttt{\}} \mid \texttt{[}d_1\texttt{|}d_2\texttt{]}$$

The function res : $\mathit{ValueStore} \times \mathit{Value} \to \mathit{Data}$ resolves all the pointers in a closure through the store σ, returning the corresponding closed term:

$$\begin{aligned}
\mathrm{res}(\sigma_v, \iota) &= \iota \\
\mathrm{res}(\sigma_v, (\ell : e, \rho)) &= e[U / \mathrm{res}(\sigma_v, \sigma_v(\rho(U))) : U \in \mathrm{fv}(e)]
\end{aligned}$$

New addresses are allocated by extracting the relevant components from the execution state:

Definition 3.2 (new, pidaddrs)**.**

$$\begin{aligned}
\mathrm{new}_{\mathrm{pa}} &: \mathit{Pid} \times \mathbb{V} \times \mathit{ProcState} \to \mathit{PidAddr} \\
\mathrm{new}_{\mathrm{pa}}(\iota, U, \langle \iota', \_, \_, t\rangle) &= \langle \iota, U, t, \iota' \rangle \\
\mathrm{new}_{\mathrm{ca}} &: \mathit{Pid} \times \mathbb{V} \times \mathit{ProcState} \times \mathit{Data} \times \mathit{Env} \to \mathit{ClosureAddr} \\
\mathrm{new}_{\mathrm{ca}}(\iota, U, \langle \_, \_, \_, t\rangle, d, \rho) &= \langle \iota, U, t, d, \mathrm{pidaddrs}(\rho)\rangle \\
\mathrm{new}_{\mathrm{ka}} &: \mathit{Pid} \times \mathit{ProcState} \times \to \mathit{KontAddr} \\
\mathrm{new}_{\mathrm{ka}}(\iota, \langle \ell, \rho, c, t\rangle) &= \langle \iota, \ell, \rho, t, \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c)\rangle
\end{aligned}$$

Where the helper function pidaddrs is given by

$$\mathrm{pidaddrs}(\rho) = \bigcup \{\mathrm{pidaddrs}(a) : a \in \mathrm{Im}\,\rho\}$$

$$\begin{aligned}
\mathrm{pidaddrs}(a = \langle \iota, U, t, \iota' \rangle) &= \{a\} \\
\mathrm{pidaddrs}(a = \langle \iota, U, t, d, I \rangle) &= I
\end{aligned}$$

$$\begin{aligned}
\mathrm{pidaddrs}(c = \star) &= \emptyset \\
\mathrm{pidaddrs}(c = \langle \iota, \ell, \rho, t, I \rangle) &= I
\end{aligned}$$

$$\begin{aligned}
\mathrm{pidaddrs}(q = \langle \ell, \rho, c, t \rangle) &= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\
\mathrm{pidaddrs}(q = \langle \iota, \rho, c, t \rangle) &= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\
\mathrm{pidaddrs}(q = \langle a, \rho, c, t \rangle) &= \mathrm{pidaddrs}(a) \cup \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c)
\end{aligned}$$

Pids are identified with the contour of the `'spawn'/1` primop that generated them:

$$\iota \in \mathit{Pid} := \mathcal{L} \times \mathit{Time}$$

The generation of a new pid is defined as

$$\begin{aligned}
\mathrm{new}_{\mathrm{pid}} &: \mathit{Pid} \times \mathcal{L} \times \mathit{Time} \to \mathit{Pid} \\
\mathrm{new}_{\mathrm{pid}}((\ell', t'), \ell, t) &= (\ell, \mathrm{tick}^*(t, \mathrm{tick}(\ell', t')))
\end{aligned}$$

where tick^* is an extension of tick that prepends a whole contour to another. The pid $\iota_0 = (\ell_0, t_0)$ is the pid associated with the starting process.

The transition system

Definition 3.3 (concrete semantics of Core programs)**.** The concrete semantics of a Core program $\mathcal{M}$ is given by a transition system $(\mathit{State}, \to, \mathrm{init}(\mathcal{M}))$, where the transition relation $\to$ is given by the rules in appendix A, some of the more interesting examples of which are presented in fig. 4. The function shrink restricts the domain of an environment to the free variables of an expression:

$$\mathrm{shrink} : \mathit{Env} \times \mathbb{E} \to \mathit{Env}, \quad \mathrm{shrink}(\rho, e) = \rho \cap \mathrm{fv}(e)$$

Before defining init, we introduce functions

$$\begin{aligned}
\mathrm{modenv} &: \bar{\wp}(\mathbb{M}) \times \mathit{Atom} \to \mathit{Env}_\perp, \\
\mathrm{exports} &: \bar{\wp}(\mathbb{M}) \times \mathit{Atom} \times \mathit{Atom} \to \mathcal{L}_\perp.
\end{aligned}$$

modenv takes a program $\mathcal{M}$ and a module name and constructs the initial environment for the module with that name in $\mathcal{M}$. That is,

$$\begin{aligned}
&\texttt{module}\ \mathit{atom}\ \texttt{[}\dots\texttt{]}\ U_1\ \texttt{=}\ \ell_1 : \dots\ \dots\ U_n\ \texttt{=}\ \ell_n : \dots\ \texttt{end} \in \mathcal{M} \\
\iff &\mathrm{modenv}(\mathcal{M}, \mathit{atom}) = [U_1 \mapsto \langle \iota_0, U_1, t_0, \texttt{fun}, \emptyset\rangle, \dots, U_n \mapsto \langle \iota_0, U_n, t_0, \texttt{fun}, \emptyset\rangle]
\end{aligned}$$

If $\mathcal{M}$ is not a valid program, or there is no module in $\mathcal{M}$ with name *atom*, then $\mathrm{modenv}(\mathcal{M}, \mathit{atom}) = \perp$. exports takes a program $\mathcal{M}$, a module name and a function name and returns the label corresponding to the definition of the corresponding exported function in the named module. That is,

$$\begin{gathered}\texttt{module}\ \mathit{atom}_0\ \texttt{[}\ldots\texttt{,}\ \mathit{atom}_1\ \texttt{=}\ U\texttt{,}\ldots\texttt{]}\ \ldots\ U\ \texttt{=}\ \ell:\ldots\ \ldots\ \texttt{end}\in\mathcal{M}\\ \iff \mathrm{exports}(\mathcal{M},\mathit{atom}_0,\mathit{atom}_1)=\ell\end{gathered}$$

If $\mathcal{M}$ is not a valid program, or there is no module in $\mathcal{M}$ with name atom_0, or atom_1 does not appear in the exports list of the module with name atom_0 in $\mathcal{M}$, then $\mathrm{exports}(\mathcal{M},\mathit{atom}_0,\mathit{atom}_1)=\bot$.

These functions in hand, we are ready to define the initial state.

Definition 3.4 (init)**.** Let the initial state of a program $\mathcal{M}$, $\mathrm{init}(\mathcal{M})$, be $\langle\pi_0,\mu_0,\sigma_{v_0},\sigma_{k_0}\rangle$ where

$$\begin{aligned}\ell &= \mathrm{exports}(\mathcal{M},\texttt{'main'},\texttt{'main'/0}),\quad \ell:\texttt{fun ()}\to\ell_0:\ldots\ \texttt{end},\\ \pi_0 &= [\iota_0\mapsto\langle\ell_0,\mathrm{modenv}(\mathcal{M},\texttt{'main'}),\star,t_0\rangle],\\ \mu_0 &= [\iota_0\mapsto\epsilon],\\ G &= \{(\mathit{atom},U,\ell):\texttt{module}\ \mathit{atom}\ \texttt{[}\ldots\texttt{]}\ \ldots\ U\ \texttt{=}\ \ell:\ldots\ \ldots\texttt{end}\in\mathcal{M}\},\\ \sigma_{v_0} &= \biguplus_{(\mathit{atom},U,\ell)\in G}[\langle\iota_0,U,t_0,\texttt{fun},\emptyset\rangle\mapsto\langle\ell,\mathrm{modenv}(\mathcal{M},\mathit{atom})\rangle],\\ \sigma_{k_0} &= [\star\mapsto\textsc{Stop}],\end{aligned}$$

Apply

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell, \rho, c, t \rangle \\
\ell : \texttt{apply}\ U\ \texttt{(}V_1\texttt{,}\dots\texttt{,}V_n\texttt{)} \\
\sigma_v(\rho(U)) = \langle \ell', \rho' \rangle \\
\ell' : \texttt{fun}\ \texttt{(}V'_1\texttt{,}\dots\texttt{,}V'_n\texttt{)}\ \texttt{->}\ \ell''\ \texttt{end} \\
\rho'' = \rho'[V'_1 \mapsto \rho(V_1), \dots, V'_n \mapsto \rho(V_n)] \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell'', \rho'', c, \text{tick}(\ell, t) \rangle]
\end{gathered}$$

Call

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell, \rho, c, t \rangle \\
\ell : \texttt{call}\ U_1\ \texttt{:}\ U_2\ \texttt{(}V_1\texttt{,}\dots\texttt{,}V_n\texttt{)} \\
\sigma_v(\rho(U_1)) = \langle \ell : \mathit{atom}_1, [] \rangle \\
\sigma_v(\rho(U_2)) = \langle \ell : \mathit{atom}_2, [] \rangle \\
\rho' = \text{modenv}(\mathcal{M}, \mathit{atom}_1) \\
\ell' = \text{exports}(\mathcal{M}, \mathit{atom}_1, \mathit{atom}_2) \\
\ell' : \texttt{fun}\ \texttt{(}V'_1\texttt{,}\dots\texttt{,}V'_n\texttt{)}\ \texttt{->}\ \ell''\ \texttt{end} \\
\rho'' = \rho'[V'_1 \mapsto \rho(V_1), \dots, V'_n \mapsto \rho(V_n)] \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell'', \rho'', c, \text{tick}(\ell, t) \rangle]
\end{gathered}$$

Push-Let

$$\begin{gathered}
\text{if } \pi(\iota) = q = \langle \ell, \rho, c, t \rangle \\
\ell : \texttt{let}\ \texttt{<}U_1\texttt{,}\ \dots\texttt{,}\ U_n\texttt{>}\ \texttt{=}\ \ell'\texttt{,}\ \ell'' \\
\kappa = \textsc{Let}\langle U_1 \cdot \dots \cdot U_n, \ell'', \rho, c \rangle \\
c' = \text{new}_{\text{ka}}(\iota, q) \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho, c', t \rangle] \\
\sigma'_k = \sigma_k[c' \mapsto \kappa]
\end{gathered}$$

Pop-Let-Closure

$$\begin{gathered}
\text{if } \pi(\iota) = q = \langle \ell : v, \rho, c, t \rangle \\
\sigma_k(c) = \textsc{Let}\langle U \cdot \epsilon, \ell', \rho', c' \rangle \\
\rho_\ell = \text{shrink}(\rho, \ell) \\
a = \text{new}_{\text{ca}}(\iota, U, q, \text{res}(\sigma_v, \langle \ell, \rho_\ell \rangle), \rho_\ell) \\
\rho'' = \rho'[U \mapsto a] \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho'', c', t \rangle] \\
\sigma'_v = \sigma_v[a \mapsto \langle \ell, \rho_\ell \rangle)]
\end{gathered}$$

Receive

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell, \rho, c, t \rangle \\
\begin{array}{l}
\ell : \texttt{receive} \\
\qquad \texttt{<}\mathit{pat}_1\texttt{>}\ \texttt{when}\ \mathit{guard}_1 \rightarrow \ell_1\texttt{;} \\
\qquad\qquad \vdots \\
\qquad \texttt{<}\mathit{pat}_m\texttt{>}\ \texttt{when}\ \mathit{guard}_m \rightarrow \ell_m \\
\quad \texttt{end}
\end{array} \\
\mathit{clause}_i = (\mathit{pat}_i, \mathit{guard}_i) \\
\mathit{clauses} = \mathit{clause}_1 \cdot \dots \cdot \mathit{clause}_m \\
\text{mmatch}(\mathit{clauses}, \mu(\iota), \rho, \sigma_v) = \langle i, a, \rho', m \rangle \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell_i, \rho \uplus \rho', c, t \rangle] \\
\mu' = \mu[\iota \mapsto m]
\end{gathered}$$

Spawn

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell : \texttt{primop}\ \texttt{'spawn'/1}\ \texttt{(}U\texttt{)}, \rho, c, t \rangle \\
\sigma_v(\rho(U)) = \langle \ell', \rho' \rangle \\
\ell' : \texttt{fun}\ \texttt{()}\ \texttt{->}\ \ell''\ \texttt{end} \\
\iota' = \text{new}_{\text{pid}}(\iota, \ell, t) \\
\text{then } \pi' = \pi\ \begin{array}[t]{l} [\iota\ \mapsto \langle \iota', \rho, c, t \rangle \\ \ \iota' \mapsto \langle \ell'', \rho', \star, t_0 \rangle] \end{array}
\end{gathered}$$

Send

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell : \texttt{primop}\ \texttt{'send'/2}\ \texttt{(}U\texttt{,}V\texttt{)}, \rho, c, t \rangle \\
\sigma_v(\rho(U)) = \iota' \\
\text{then } \pi' = \pi[\iota \mapsto \langle \rho(V), \rho, c, t \rangle] \\
\mu' = \mu[\iota' \mapsto \text{enq}(\rho(V), \mu(\iota'))]
\end{gathered}$$

Self

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell : \texttt{primop}\ \texttt{'self'/0}\ \texttt{()}, \rho, c, t \rangle \\
\text{then } \pi' = \pi[\iota \mapsto \langle \iota, \rho, c, t \rangle]
\end{gathered}$$

Figure 4: Some transition rules for the concrete semantics. Each rule specifies a transition $\langle \pi, \mu, \sigma_v, \sigma_k \rangle \rightarrow \langle \pi', \mu', \sigma'_v, \sigma'_k \rangle$. Unless otherwise stated in the "then" part of the rule, we have that $\pi = \pi'$, $\mu = \mu'$, $\sigma_v = \sigma'_v$ and $\sigma_k = \sigma'_k$. We let the variable $v \in \mathbb{E}$ range over irreducible expressions, excluding valuelists and `fail`; that is, atom, nil, tuple, list and lambda expressions.

4 Control flow analysis for Core

We construct the πACTOR model via a traditional control flow analysis. The only sources of infinity in the state space are the domains *Data*, where we can nest structures infinitely deep[4], *Mailbox*, where we may have an infinitely long queue of unprocessed messages, and *Time*, where we may have contours of infinite length. If we finitise these domains, *State* becomes finite and reachability becomes decidable for $(\mathit{State}, \to, \mathrm{init}(\mathcal{M}))$.

Again following [9], we leave the abstraction of these basic domains as parameters of the abstraction and state the conditions they must satisfy for guaranteeing soundness of the overall analysis.

Definition 4.1 (basic domains abstraction)**.** A basic domains abstraction is a triple $\mathcal{I} = (D, T, M)$ consisting of a data, a time and a mailbox abstraction defined as follows:

(i) A data abstraction is a triple $D = (\widehat{\mathit{Data}}, \alpha_d, \widehat{\mathrm{res}})$ where $\widehat{\mathit{Data}}$ is a flat (discretely ordered) domain of abstract data values and

$$\alpha_d : \mathit{Data} \to \widehat{\mathit{Data}}, \quad \widehat{\mathrm{res}} : \widehat{\mathit{Store}} \times \widehat{\mathit{Value}} \to \wp(\widehat{\mathit{Data}})$$

and α_d is monotonic.

(ii) A time abstraction is a triple $T = (\widehat{\mathit{Time}}, \alpha_t, \widehat{\mathrm{tick}}, \hat{t}_0)$ where $\widehat{\mathit{Time}}$ is a flat domain of abstract contours, $\hat{t}_0 \in \widehat{\mathit{Time}}$ and

$$\alpha_t : \mathit{Time} \to \widehat{\mathit{Time}}, \quad \widehat{\mathrm{tick}} : \mathcal{L} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Time}}$$

and α_t is monotonic.

(iii) A mailbox abstraction is a septuple $M = (\widehat{\mathit{Mailbox}}, \leq_{\mathfrak{m}}, \sqcup_{\mathfrak{m}}, \alpha_{\mathfrak{m}}, \widehat{\mathrm{enq}}, \widehat{\epsilon}, \widehat{\mathrm{mmatch}})$ where $(\widehat{\mathit{Mailbox}}, \leq_{\mathfrak{m}}, \sqcup_{\mathfrak{m}})$ is a join-semilattice with least element $\widehat{\epsilon} \in \widehat{\mathit{Mailbox}}$ and

$$\alpha_{\mathfrak{m}} : \mathit{Time} \to \widehat{\mathit{Time}}, \quad \widehat{\mathrm{enq}} : \widehat{\mathit{ValueAddr}} \times \widehat{\mathit{Mailbox}} \to \widehat{\mathit{Mailbox}}$$

$$\widehat{\mathrm{mmatch}} : (\mathit{pat} \times \mathit{guard})^* \times \widehat{\mathit{Mailbox}} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \wp(\mathbb{N} \times (\mathbb{V} \rightharpoonup \widehat{\mathit{Value}}) \times \widehat{\mathit{Mailbox}})$$

and $\alpha_{\mathfrak{m}}$ and $\widehat{\mathrm{enq}}$ are monotonic on mailboxes.

Given a basic domains abstraction (D, T, M), we define an interpretation of the other domains thus:

$$\begin{aligned}
\widehat{S} \in \widehat{\mathit{State}} &:= \widehat{\mathit{Procs}} \times \widehat{\mathit{Mailboxes}} \times \widehat{\mathit{ValueStore}} \times \widehat{\mathit{KontStore}} \\
\widehat{\pi} \in \widehat{\mathit{Procs}} &:= \widehat{\mathit{Pid}} \rightharpoonup \widehat{\mathit{ProcState}} \\
\widehat{\mu} \in \widehat{\mathit{Mailboxes}} &:= \widehat{\mathit{Pid}} \rightharpoonup \widehat{\mathit{Mailbox}} \\
\widehat{\sigma}_v \in \widehat{\mathit{ValueStore}} &:= (\widehat{\mathit{ClosureAddr}} \rightharpoonup \widehat{\mathit{Closure}}) \uplus (\widehat{\mathit{PidAddr}} \rightharpoonup \widehat{\mathit{Pid}}) \\
\widehat{\sigma}_k \in \widehat{\mathit{KontStore}} &:= \widehat{\mathit{KontAddr}} \rightharpoonup \widehat{\mathit{Kont}} \\
\widehat{q} \in \widehat{\mathit{ProcState}} &:= \widehat{\mathit{Pid}} \uplus \mathcal{L} \uplus \widehat{\mathit{ValueAddr}} \times \widehat{\mathit{Env}} \times \widehat{\mathit{KontAddr}} \times \widehat{\mathit{Time}} \\
\hat{\imath} \in \widehat{\mathit{Pid}} &:= \{\hat{\imath}_0\} \uplus (\mathcal{L} \times \widehat{\mathit{Time}}), \quad \hat{\imath}_0 := \hat{t}_0 \\
\widehat{\rho} \in \widehat{\mathit{Env}} &:= \mathbb{V} \rightharpoonup \widehat{\mathit{ValueAddr}} \\
\widehat{\mathit{PidAddr}} &:= \widehat{\mathit{Pid}} \times \mathbb{V} \times \widehat{\mathit{Time}} \times \widehat{\mathit{Pid}} \\
\widehat{\mathit{ClosureAddr}} &:= \widehat{\mathit{Pid}} \times \mathbb{V} \times \widehat{\mathit{Time}} \times \widehat{\mathit{Data}} \times \wp(\widehat{\mathit{PidAddr}}) \\
\widehat{a} \in \widehat{\mathit{ValueAddr}} &:= \widehat{\mathit{PidAddr}} \uplus \widehat{\mathit{ClosureAddr}} \\
\widehat{c} \in \widehat{\mathit{KontAddr}} &:= \{\star\} \uplus (\widehat{\mathit{Pid}} \times \mathcal{L} \times \widehat{\mathit{Time}} \times \widehat{\mathit{Env}} \times \wp(\widehat{\mathit{PidAddr}})) \\
\widehat{\mathit{Value}} &:= \widehat{\mathit{Pid}} \uplus (\mathcal{L} \times \widehat{\mathit{Env}})
\end{aligned}$$

[4]The infiniteness of the set of atoms is not relevant, since only finitely many atoms can appear in the source of a program, and the infiniteness of the set of pids is due to the infiniteness of the *Time* domain, so will become finite as *Time* does.

and we equip each with an abstraction function defined by an appropriate pointwise extension. For example,

$$\begin{aligned}&\alpha_a : \mathit{ClosureAddr} \to \widehat{\mathit{ClosureAddr}}\\ &\alpha_a(\langle \iota, U, t, d, I\rangle) = \langle \alpha_\iota(\iota), \alpha_U(U), \alpha_t(t), \alpha_d(d), \{\alpha_{\mathit{PidAddr}}(a) : a \in I\}\rangle\end{aligned}$$

We will call all of them α_{CFA} without confusion. The abstract domain $\widehat{\mathit{Kont}}$ is the pointwise abstraction of Kont and we use the constructs $\widehat{\textsc{Stop}}$, $\widehat{\textsc{Let}}\langle\_,\_,\_,\_\rangle$ and $\widehat{\textsc{Do}}\langle\_,\_,\_\rangle$ for the abstractions of $\textsc{Stop}$, $\textsc{Let}\langle\_,\_,\_,\_\rangle$ and $\textsc{Do}\langle\_,\_,\_\rangle$ respectively. The abstract functions $\widehat{\mathrm{new}}_{\mathrm{ca}}$, $\widehat{\mathrm{new}}_{\mathrm{pa}}$, $\widehat{\mathrm{new}}_{\mathrm{ka}}$ and $\widehat{\mathrm{new}}_{\mathrm{pid}}$ and the helper function $\widehat{\mathrm{pidaddrs}}$ are defined exactly as their concrete versions, but on the abstract domains.

When B is a flat domain, the abstraction of a partial map $C = A \rightharpoonup B$ to $\widehat{C} = \widehat{A} \to \wp(\widehat{B})$ is defined as

$$\alpha_C(f) := (\lambda \widehat{a} \in \widehat{A}.\{\alpha_B(b) : (a,b) \in f, \alpha_A(a) = \widehat{a}\})$$

where the preorder on $\widehat{C}$ is $f \leq_{\widehat{C}} \widehat{g} \iff \forall \widehat{a} : \hat{f}(\widehat{a}) \subseteq g(\widehat{a})$.

The operations that support the new abstract domains need to resemble the operations on their concrete counterparts. The correctness conditions below must be satisfied by their instances.

Definition 4.2 (sound basic domains abstraction)**.** A basic domains abstraction $I = (D, T, M)$, $D = (\widehat{\mathit{Data}}, \alpha_d, \widehat{\mathrm{res}})$, $T = (\widehat{\mathit{Time}}, \alpha_t, \widehat{\mathrm{tick}}, \hat{t}_0)$, $M = (\widehat{\mathit{Mailbox}}, \leq_{\mathfrak{m}}, \sqcup_{\mathfrak{m}}, \alpha_{\mathfrak{m}}, \widehat{\mathrm{enq}}, \widehat{\epsilon}, \widehat{\mathrm{mmatch}})$ is sound only if the following conditions are satisfied:

$$\begin{gathered}\alpha_t(\mathrm{tick}(\ell, t)) \leq \widehat{\mathrm{tick}}(\ell, \alpha_t(t))\\ \widehat{\sigma}_v \leq \widehat{\sigma}'_v \wedge \hat{d} \leq \hat{d}' \implies \widehat{\mathrm{res}}(\widehat{\sigma}_v, \hat{d}) \leq \widehat{\mathrm{res}}(\widehat{\sigma}', \hat{d}')\\ \forall \widehat{\sigma}_v \geq \alpha_{\mathrm{CFA}}(\sigma_v) : \alpha_d(\mathrm{res}(\sigma_v, d)) \in \widehat{\mathrm{res}}(\widehat{\sigma}_v, \alpha_{\mathrm{CFA}}(d))\\ \alpha_{\mathfrak{m}}(\mathrm{enq}(d, \mathfrak{m})) \leq \widehat{\mathrm{enq}}(\alpha_{\mathrm{CFA}}(d), \alpha_{\mathfrak{m}}(\mathfrak{m})), \quad \alpha_{\mathfrak{m}}(\epsilon) = \widehat{\epsilon}\end{gathered}$$

and if $\mathrm{mmatch}(\mathit{clauses}, \mathfrak{m}, \rho, \sigma_v) = (i, a, \theta, \mathfrak{m}')$ then $\forall \widehat{\mathfrak{m}} \geq \alpha_{\mathfrak{m}}(\mathfrak{m})$, $\forall \widehat{\sigma}_v \geq \alpha_{\mathrm{CFA}}(\sigma_v)$, $\exists \widehat{\mathfrak{m}}' \geq \alpha_{\mathfrak{m}}(\mathfrak{m}')$ such that

$$(i, \alpha_{\mathrm{CFA}}(a), \alpha_{\mathrm{CFA}}(\theta), \widehat{\mathfrak{m}}') \in \widehat{\mathrm{mmatch}}(\mathit{clauses}, \widehat{\mathfrak{m}}, \alpha_{\mathrm{CFA}}(\rho), \widehat{\sigma}_v)$$

Once the abstract domains are fixed, the rules that define the abstract transition relation are straightforward abstractions of the concrete rules.

Definition 4.3 (abstract semantics for $\textsc{Core}$)**.** Given a basic domains abstraction the abstract (CFA) semantics for a $\textsc{Core}$ program $\mathcal{M}$ are given by a transition system $(\widehat{\mathit{State}}, \rightsquigarrow, \alpha_{\mathrm{CFA}}(\mathrm{init}(\mathcal{M})))$, where the transition relation $\rightsquigarrow$ is given by the rules in appendix B. The abstract equivalents of the rules presented in fig. 4 are presented in fig. 5.

Theorem 4.4 (soundness for CFA)**.** *Given a sound basic domains abstraction, if $S, S' \in \mathit{State}$, $S \to S'$ and $\alpha_{CFA}(S) \leq u$, then there exists $u' \in \widehat{\mathit{State}}$ such that $\alpha_{CFA}(S') \leq u'$ and $u \rightsquigarrow u'$.*

Proof. omitted. We refer the reader to the proof of Theorem 2 in [9]. □

Theorem 4.5 (decideability of CFA)**.** *Given that a sound basic domains abstraction is finite, then the derived abstract transition system is finite. It is also decidable given that the auxiliary operations are computable.*

Throughout this report, we will use the mailbox abstraction

$$\begin{gathered}M_{\mathrm{set}} = (\wp(\widehat{\mathit{ValueAddr}}), \subseteq, \cup, \alpha^{\mathrm{set}}_{\mathfrak{m}}, \widehat{\mathrm{enq}}_{\mathrm{set}}, \emptyset, \widehat{\mathrm{mmatch}}_{\mathrm{set}}), \text{ where}\\ \alpha^{\mathrm{set}}_{\mathfrak{m}}(\mathfrak{m}) := \{\alpha_{\mathrm{CFA}}(d) | d \in \mathfrak{m}\}, \quad \widehat{\mathrm{enq}}_{\mathrm{set}}(\hat{d}, \widehat{m}) := \{\hat{d}\} \cup \widehat{\mathfrak{m}}\\ \widehat{\mathrm{mmatch}}_{\mathrm{set}}(\mathit{clauses}, \widehat{\mathfrak{m}}, \widehat{\rho}, \widehat{\sigma}_v) := \{(i, \widehat{a}, \widehat{\theta}, \widehat{\mathfrak{m}}) : \hat{d} \in \widehat{m}, \widehat{\theta} \in \widehat{\mathrm{match}}(p_i, g_i, \hat{d}, \widehat{\rho}, \widehat{\sigma}_v)\}\end{gathered}$$

The time abstraction will be some $T_k = (\{s : s \in \mathcal{L}^*, \mathrm{len}(s) \leq k\}, \alpha^k_t, \widehat{\mathrm{tick}}_k, t_0)$, where $\alpha^k_t(s) = \lfloor s \rfloor_k$ and $\widehat{\mathrm{tick}}_k(\ell, \hat{t}) = \lfloor \ell \cdot \hat{t} \rfloor_k$, the usual time abstraction for a k-CFA. We will usually use the trivial data abstraction $\widehat{\mathit{Data}}_0 = (\{\dagger\}, \widehat{\mathrm{res}}_\dagger)$ where $\widehat{\mathrm{res}}(\widehat{\sigma}_v, \hat{d}) = \dagger$ for all $\widehat{\sigma}_v$, $\hat{d}$, but may use any k-deep data abstraction D_k. All of these abstractions are sound.

Abs-Apply

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\ell : \texttt{apply}\ U\ \texttt{(}V_1\texttt{,}\dots\texttt{,}V_n\texttt{)}$$
$$\widehat{\sigma}_v(\widehat{\rho}(U)) \ni \langle \ell', \widehat{\rho}'\rangle$$
$$\ell' : \texttt{fun (}V'_1\texttt{,}\dots\texttt{,}V'_n\texttt{)} \to \ell''\ \texttt{end}$$
$$\widehat{\rho}'' = \widehat{\rho}'[V'_1 \mapsto \widehat{\rho}(V_1), \dots, V'_n \mapsto \widehat{\rho}(V_n)]$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell'', \widehat{\rho}'', \widehat{c}, \widehat{\text{tick}}(\ell, \hat{t})\rangle\}]$$

Abs-Call

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\ell : \texttt{call}\ U_1\ \texttt{:}\ U_2\ \texttt{(}V_1\texttt{,}\dots\texttt{,}V_n\texttt{)}$$
$$\widehat{\sigma}_v(\widehat{\rho}(U_1)) \ni \langle \ell : \mathit{atom}_1, []\rangle$$
$$\widehat{\sigma}_v(\widehat{\rho}(U_2)) \ni \langle \ell : \mathit{atom}_2, []\rangle$$
$$\widehat{\rho}' = \widehat{\text{modenv}}(\mathcal{M}, \mathit{atom}_1)$$
$$\ell' = \text{exports}(\mathcal{M}, \mathit{atom}_1, \mathit{atom}_2)$$
$$\ell' : \texttt{fun (}V'_1\texttt{,}\dots\texttt{,}V'_n\texttt{)} \to \ell''\ \texttt{end}$$
$$\widehat{\rho}'' = \widehat{\rho}'[V'_1 \mapsto \widehat{\rho}(V_1), \dots, V'_n \mapsto \widehat{\rho}(V_n)]$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell'', \widehat{\rho}'', \widehat{c}, \widehat{\text{tick}}(\ell, \hat{t})\rangle\}]$$

Abs-Push-Let

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \widehat{q} = \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\ell : \texttt{let <}U_1\texttt{,}\dots\texttt{,}U_n\texttt{> =}\ell'\texttt{,}\ \ell''$$
$$\widehat{\kappa} = \widehat{\textsc{Let}}\langle U_1 \cdot \ldots \cdot U_n, \ell'', \widehat{\rho}, \widehat{c}\rangle$$
$$\widehat{c}' = \widehat{\text{new}}_{\text{ka}}(\hat{\imath}, \widehat{q})$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}, \widehat{c}', \hat{t}\rangle\}]$$
$$\widehat{\sigma}'_k = \widehat{\sigma}_k \sqcup [\widehat{c}' \mapsto \{\widehat{\kappa}\}]$$

Abs-Pop-Let-Closure

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \widehat{q} = \langle \ell : v, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_k(\widehat{c}) \ni \widehat{\textsc{Let}}\langle U \cdot \epsilon, \ell', \widehat{\rho}', \widehat{c}'\rangle$$
$$\widehat{\rho}_\ell = \text{shrink}(\widehat{\rho}, \ell)$$
$$\hat{d} \in \widehat{\text{res}}(\widehat{\sigma}_v, \langle v, \widehat{\rho}_\ell\rangle),$$
$$\widehat{a} = \widehat{\text{newva}}_{\text{data}}(\hat{\imath}, U, \widehat{q}, \hat{d}, \widehat{\rho}_\ell)$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}'[U \mapsto \widehat{a}], \widehat{c}', \hat{t}\rangle\}]$$
$$\widehat{\sigma}'_v = \widehat{\sigma}_v \sqcup [\widehat{a} \mapsto \{\langle \ell, \widehat{\rho}_\ell\rangle\}]$$

Abs-Receive

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\ell : \texttt{receive}\ \mathit{clause}_1\texttt{;}\ \dots\texttt{;}\mathit{clause}_n\ \texttt{end}$$
$$\mathit{clauses} = \mathit{clause}_1 \cdot \ldots \cdot \mathit{clause}_n$$
$$\widehat{\text{mmatch}}(\mathit{clauses}, \widehat{\mu}(\hat{\imath}), \widehat{\sigma}_v) \ni \langle i, \widehat{a}, \widehat{\rho}', \widehat{m}\rangle$$
$$\mathit{clause}_i = \texttt{<}\mathit{pat}_i\texttt{>}\ \texttt{when}\ \mathit{guard}_i \to \ell_i$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell_i, \widehat{\rho} \uplus \widehat{\rho}', \widehat{c}, \hat{t}\rangle\}]$$
$$\widehat{\mu}' = \widehat{\mu} \sqcup [\hat{\imath} \mapsto \widehat{m}]$$

Abs-Spawn

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop 'spawn'/1 (}U\texttt{)}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_v(\widehat{\rho}(U)) \ni \langle \ell', \widehat{\rho}'\rangle$$
$$\ell' : \texttt{fun () } \to \ell''\ \texttt{end}$$
$$\hat{\imath} = \langle \ell', \hat{t}'\rangle$$
$$\hat{\imath}' = \langle \ell, \widehat{\text{tick}}^{\star}(\hat{t}, \widehat{\text{tick}}(\ell', \hat{t}'))\rangle$$
$$\begin{aligned}\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} &\mapsto \{\langle \hat{\imath}', \widehat{\rho}, \widehat{c}, \hat{t}\rangle\} \\ \hat{\imath}' &\mapsto \{\langle \ell'', \widehat{\rho}', \star, \hat{t}_0\rangle\}]\end{aligned}$$

Abs-Send

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop 'send'/2 (}U\texttt{,}V\texttt{)}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_v(\widehat{\rho}(U)) \ni \hat{\imath}'$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \widehat{\rho}(V), \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}]$$
$$\widehat{\mu}' = \widehat{\mu} \sqcup [\hat{\imath}' \mapsto \widehat{\text{enq}}(\widehat{\rho}(V), \widehat{\mu}(\hat{\imath}'))]$$

Abs-Self

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop 'self'/0 ()}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \hat{\imath}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}]$$

Figure 5: Abstract equivalents of the rules presented in fig. 4. As in the concrete semantics, each rule specifies a transition $\langle \widehat{\pi}, \widehat{\mu}, \widehat{\sigma}_v, \widehat{\sigma}_k\rangle \to \langle \widehat{\pi}', \widehat{\mu}', \widehat{\sigma}'_v, \widehat{\sigma}'_k\rangle$. Unless otherwise stated in the "then" part of the rule, we have that $\widehat{\pi} = \widehat{\pi}'$, $\widehat{\mu} = \widehat{\mu}'$, $\widehat{\sigma}_v = \widehat{\sigma}'_v$ and $\widehat{\sigma}_k = \widehat{\sigma}'_k$. We let the variable $v \in \mathbb{E}$ range over irreducible expressions, excluding valuelists and `fail`; that is, atom, nil, tuple, list and lambda expressions.

5 Generating the πActor model

Definition 5.1 (Active components)**.** We note that every transition rule R for $S \to S'$ in the concrete semantics begins by specifying some $q = \langle f, \rho, c, t\rangle = \pi(\iota)$. and ends by associating a new local state $q' = \langle f', \rho', c', t'\rangle$ with ι in π', i.e. $q' = \pi'(\iota)$. We call (ι, q, q') the active components of the transition $S \to S'$ by R. The same is true (modulo abstraction) for rules in the abstraction semantics; that is, every abstract transition rule Abs-R for $\widehat{S} \rightsquigarrow \widehat{S}$ begins by specifying some $\widehat{q} = \langle f, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle \in \widehat{\pi}(\widehat{\iota})$. and ends by associating a new local state $\widehat{q}' = \langle f', \widehat{\rho}', \widehat{c}', \widehat{t}'\rangle$ with $\hat{\iota}$ in $\widehat{\pi}'$, i.e. $\widehat{q}' \in \widehat{\pi}'(\hat{\iota})$. We call $(\hat{\iota}, \widehat{q}, \widehat{q}')$ the active components of the transition $\widehat{S} \rightsquigarrow \widehat{S}'$ by Abs-R.

Definition 5.2 (setname, setname$'$)**.** Fix some total order $\widehat{<_{\mathrm{VA}}}$ on abstract pid addresses. Fix two disjoint families of set names $\mathfrak{A}, \mathfrak{A}' \subset \mathfrak{S}$, $\mathfrak{A} \cap \mathfrak{A}' = \emptyset$, $|\mathfrak{A}| = |\mathfrak{A}'| = |\widehat{PidAddr}|$. Let $<$ be some total order on $\mathfrak{S}$ with $A \in \mathfrak{A}, A' \in \mathfrak{A}' \implies A < A'$. and let setname be an isomorphism from $(\widehat{PidAddr}, \widehat{<_{\mathrm{VA}}})$ to $(\mathfrak{A}, <)$ and setname$'$ an isomorphism from $(\widehat{PidAddr}, \widehat{<_{\mathrm{VA}}})$ to $(\mathfrak{A}', <)$. That is,

$$\begin{aligned}
\forall \widehat{a}, \widehat{a}' \in \widehat{PidAddr},\ \widehat{a} = \widehat{a}' &\iff \mathrm{setname}(\widehat{a}) = \mathrm{setname}(\widehat{a}')\\
\widehat{a} \mathrel{\widehat{<_{\mathrm{VA}}}} \widehat{a}' &\iff \mathrm{setname}(\widehat{a}) < \mathrm{setname}(\widehat{a}')\\
\widehat{a} \mathrel{\widehat{<_{\mathrm{VA}}}} \widehat{a}' &\iff \mathrm{setname}'(\widehat{a}) < \mathrm{setname}'(\widehat{a}')
\end{aligned}$$

Definition 5.3 (defs)**.** For a Core program $\mathcal{M}$ and a sound basic domains abstraction I, the definition set $\Delta = \mathrm{defs}(\mathcal{M})$ is the smallest set of πActor definitions such that the following holds:

Consider the CFA semantics given I, $(\widehat{State}, \rightsquigarrow, \alpha_{\mathrm{CFA}}(\mathrm{init}(\mathcal{M})))$. For each $\widehat{S} \in \widehat{State}$ such that $\alpha_{\mathrm{CFA}}(\mathrm{init}(\mathcal{M})) \rightsquigarrow^* \widehat{S}$, $\widehat{S} = \langle \widehat{\pi}, \widehat{\mu}, \widehat{\sigma}_v, \widehat{\sigma}_k$, For every transition $\widehat{S} \rightsquigarrow \widehat{S}'$ by the rule Abs-Send with active components $(\hat{\iota}, \widehat{q}, \widehat{q}')$, where $\widehat{q} = \langle \ell : \texttt{primop 'send'/2 (}U\texttt{,}V\texttt{)}, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle$ and $\rho(U) = \widehat{a}$, Δ contains

$$\begin{gathered}
\mathrm{Send}_{\widehat{a}}[\, s' \mid args\,] := s'\,!\,\widehat{a}\langle args\rangle.\mathbf{0},\\
args = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}') : \widehat{a}' \in \mathrm{pidaddrs}(\widehat{a})\}, <)
\end{gathered}$$

For every abstract process state $\widehat{q} = \langle f, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle \in \bigcup \mathrm{Im}\, \widehat{\pi}$,

- If $f = \ell : \texttt{fail}$, Δ contains
$$\widehat{q}[\, s \mid args\,] := \boldsymbol{\tau}.\mathbf{fail}$$
- Else if there is no abstract transition in the CFA with active components $(\hat{\iota}, \widehat{q}, \widehat{q}')$, Δ contains
$$\begin{aligned}
\widehat{q}[\, s, s' \mid args\,] &:= \mathbf{0}, && f \in \widehat{Pid}\\
\widehat{q}[\, s \mid args\,] &:= \mathbf{0}, && \text{otherwise}
\end{aligned}$$
- Else if there is an abstract transition in the CFA with active components $(\hat{\iota}, \widehat{q}, \widehat{q}')$, Δ contains
$$\begin{aligned}
\widehat{q}[\, s, s' \mid args\,] &:= \sum \mathbb{P}, && f \in \widehat{Pid}\\
\widehat{q}[\, s \mid args\,] &:= \sum \mathbb{P}, && \text{otherwise}
\end{aligned}$$

In all of the above cases,

$$args = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\widehat{q})\}, <)$$

and $\mathbb{P}$ is the smallest set of sequential terms in $\mathfrak{T}$ for which all of the following is true:

For each transition $\widehat{S} \rightsquigarrow \widehat{S}'$ by rule Abs-R with active components $(\hat{\iota}, \widehat{q}, \widehat{q}')$,

- If Abs-R is Abs-Name, Abs-Letrec, Abs-Push-Let, Abs-Push-Do, Abs-Bad-Apply, Abs-Bad-Call, Abs-Bad-Case, Abs-Bad-Spawn, Abs-Bad-Send or Abs-Bad-Pop-Let then $\mathbb{P}$ contains $\boldsymbol{\tau}.\widehat{q}'[\, s \mid args\,]$.

- If Abs-R is Abs-Apply, Abs-Call, Abs-Pop-Let-Closure, Abs-Pop-Let-ValueAddr or Abs-Pop-Do, then then $\mathbb{P}$ contains $\boldsymbol{\tau}.\widehat{q}'[\, s \mid \mathit{args}' \,]$, where

$$\mathrm{args}' = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\widehat{q}')\}, <).$$

- If Abs-R is Abs-Pop-Let-Pid, where $f = \widehat{\iota}'$ and $\widehat{a} = \widehat{\mathrm{new}_{\mathrm{pa}}}(\widehat{\iota}, U, \widehat{q})$, then $\mathbb{P}$ contains $\boldsymbol{\tau}.\widehat{q}'[\, s \mid \mathit{args}' \,]$, where

$$B_1 \cdot \ldots \cdot B_m = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}') : \widehat{a}' \in \mathrm{pidaddrs}(\widehat{q}')\}, <),$$

$$\mathit{args}' = T_1 \cdot \ldots \cdot T_m, T_i = \begin{cases} B_i \cup \{s'\}, & B_i \in \mathit{args}, B_i = \mathrm{setname}(\widehat{a}) \\ B_i, & B_i \in \mathit{args}, B_i \neq \mathrm{setname}(\widehat{a}) \\ \{s'\}, & B_i \notin \mathit{args}, B_i = \mathrm{setname}(\widehat{a}) \\ \emptyset & B_i \notin \mathit{args}, B_i \neq \mathrm{setname}(\widehat{a}) \end{cases}$$

- If Abs-R is Abs-Case, where $\langle i, \widehat{\theta}\rangle$ is the selected match, and guard_i is the guard of the matched pattern, then let

$$G = \{(\widehat{a}', \widehat{a}'') : \widehat{a}' = (\widehat{\rho}\widehat{\theta})(U),\ \widehat{a}'' = (\widehat{\rho}\widehat{\theta})(V),\ \widehat{a}', \widehat{a}'' \in \widehat{\mathit{PidAddr}},\ U \ \texttt{=:=}\ V \text{ appears in } \mathit{guard}_i\}$$

then $\mathbb{P}$ contains $[\mathit{conds}].\widehat{q}'[\, s \mid \mathit{args} \,]$, where

$$\mathit{conds} = \mathrm{sort}(\{A_1 \cap A_2 \neq \emptyset : (\widehat{a}', \widehat{a}'') \in G, A_1 = \mathrm{setname}(\widehat{a}'), A_2 = \mathrm{setname}(\widehat{a}'')\}, <_\cap)$$

where $<_\cap$ orders terms of the form $A_1 \cap A_2 \neq$ lexicographically by (A_1, A_2).

- If Abs-R is Abs-Receive, where $\langle i, \widehat{a}, \widehat{\theta}, \widehat{\mathfrak{m}}\rangle$ is the selected match, and guard_i is the guard of the matched pattern, then let

$$G = \{(\widehat{a}', \widehat{a}'') : \widehat{a}' = (\widehat{\rho}\widehat{\theta})(U),\ \widehat{a}'' = (\widehat{\rho}\widehat{\theta})(V),\ \widehat{a}', \widehat{a}'' \in \widehat{\mathit{PidAddr}},\ U \ \texttt{=:=}\ V \text{ appears in } \mathit{guard}_i\}$$

$$\mathrm{setname}_{\mathit{clause}_i}(\widehat{a}) = \begin{cases} \mathrm{setname}'(\widehat{a}) & \mathrm{setname}(\widehat{a}) \text{ appears in a pattern in } \mathit{clause}_i \\ \mathrm{setname}(\widehat{a}) & \text{otherwise} \end{cases}$$

then $\mathbb{P}$ contains

$$\begin{aligned} s?\widehat{a}(\mathit{recargs})[\mathit{conds}].\widehat{q}'[\, s \mid \mathit{args}' \,],\ \text{where} \\ \mathit{recargs} = \mathrm{sort}(\{\mathrm{setname}'(\widehat{a}') : \widehat{a}' \in \mathrm{pidaddrs}(\widehat{a})\}, <) \\ \widehat{a}_1 \cdot \ldots \cdot \widehat{a}_m = \mathrm{sort}(\mathrm{pidaddrs}(\widehat{q}'), <) \\ B_i = \mathrm{setname}(\widehat{a}_i) \\ B_i' = \mathrm{setname}'(\widehat{a}_i) \end{aligned}$$

$$\mathit{args}' = T_1 \cdot \ldots \cdot T_m,\ T_i = \begin{cases} B_i \cup B_i', & B_i \in \mathit{args}, B_i' \in \mathit{recargs} \\ B_i, & B_i \in \mathit{args}, B_i' \notin \mathit{recargs} \\ B_i', & B_i \notin \mathit{args}, B_i' \in \mathit{recargs} \\ \emptyset, & B_i \notin \mathit{args}, B_i' \notin \mathit{recargs} \end{cases}$$

$$\mathit{conds} = \mathrm{sort}(\{A_1 \cap A_2 \neq \emptyset : A_1 = (\widehat{a}', \widehat{a}'') \in G, \mathrm{setname}_{\mathit{clause}_i}(\widehat{a}'), A_2 = \mathrm{setname}_{\mathit{clause}_i}(\widehat{a}'')\}, <_\cap)$$

where $<_\cap$ orders terms of the form $A_1 \cap A_2 \neq$ lexicographically by (A_1, A_2).

- If Abs-R is Abs-Self, then $\mathbb{P}$ contains $\boldsymbol{\tau}.\widehat{q}'[\, s, s \mid \mathit{args} \,]$.

- If Abs-R is Abs-Spawn, where $\widehat{\pi}' = \widehat{\pi} \sqcup [\widehat{\iota} \mapsto \{\widehat{q}'\}, \widehat{\iota}' \mapsto \{\widehat{q}''\}]$ then $\mathbb{P}$ contains

$$\begin{aligned} \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\widehat{q}'[\, s, s' \mid \mathit{args} \,] \parallel \widehat{q}''[\, s' \mid \mathit{args}' \,]),\ \text{where} \\ \mathit{args}' = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\widehat{q}'')\}, <) \end{aligned}$$

- If Abs-R is Abs-Send, where $f = \ell : \texttt{primop 'send'/2 (}U\texttt{,}V\texttt{)}$, let $a = \mathrm{setname}(\widehat{\rho}(U)), \widehat{a} = \widehat{\rho}(V)$. then $\mathbb{P}$ contains

  $$\mathbf{let}\ s' \in a.\, (\textsc{Send}_{\widehat{a}}[\, s' \mid \mathit{args}' \,] \parallel \widehat{q}'[\, s \mid \mathit{args} \,])$$

  where $\mathit{args}' = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\widehat{\rho}(V))\}, <)$.

These process definitions for $\mathcal{M}$ in hand, we are ready to define the abstraction function.

Definition 5.4 (name)**.** Fix a family of names $\mathfrak{I} \subset \mathfrak{N}$ with $|\mathfrak{I}| = |\mathit{Pid}|$. and let name be a bijection from Pid to $\mathfrak{I}$.

Definition 5.5 (α_π)**.** Let $S = \langle \pi, \mu, \sigma_v, \sigma_k \rangle \in \mathit{State}$ be a state in the concrete semantics of $\mathcal{M}$. Let $\mathrm{dom}\,\pi = \{\iota_1, \dots, \iota_n\}$. Let $\vec{s} = \mathrm{name}(\iota_1) \cdot \ldots \cdot \mathrm{name}(\iota_n)$. Then

$$\begin{aligned}
\alpha_\pi(S) &= \boldsymbol{\nu} \vec{s}.\, (\alpha_\pi(\pi) \parallel \alpha_\pi(\mu)) \\
\alpha_\pi(\pi) &= \prod_{\iota \in \mathrm{dom}\,\pi} \alpha_\pi(\pi(\iota), \iota) \\
\alpha_\pi(q, \iota) &= \begin{cases} q[\,\mathrm{name}(\iota), \mathrm{name}(\iota') \mid \mathrm{args}(q)\,], & q = \langle \iota', \rho, c, t \rangle \\ q[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], & \text{otherwise} \end{cases} \\
\alpha_\pi(\mu) &= \prod_{\iota \in \mathrm{dom}\,\mu} \prod_{a \in \mu(i)} \textsc{Send}_{\alpha_{\mathrm{CFA}}(a)}[\,\mathrm{name}(\iota) \mid \mathrm{args}(a)\,]
\end{aligned}$$

where

$$\begin{aligned}
\mathrm{args}(q) &= T_1 \cdot \ldots \cdot T_m \\
\widehat{a}_1 \cdot \ldots \cdot \widehat{a}_m &= \mathrm{sort}(\mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q)), \widehat{<_{\mathrm{VA}}}) \\
T_i &= \{\mathrm{name}(\iota) : a = \langle \_, \_, \_, \iota \rangle \in \mathrm{pidaddrs}(q), \alpha_{\mathrm{CFA}}(a) = \widehat{a}_i\},
\end{aligned}$$

$$\begin{aligned}
\mathrm{args}(a) &= T_1 \cdot \ldots \cdot T_m \\
\widehat{a}_1 \cdot \ldots \cdot \widehat{a}_m &= \mathrm{sort}(\mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(a)), \widehat{<_{\mathrm{VA}}}) \\
T_i &= \{\mathrm{name}(\iota) : a' = \langle \_, \_, \_, \iota \rangle \in \mathrm{pidaddrs}(a), \alpha_{\mathrm{CFA}}(a') = \widehat{a}_i\}.
\end{aligned}$$

Remark 5.5.1. $(\alpha_\pi(\mathrm{init}(\mathcal{M})), \mathrm{defs}(\mathcal{M}))$ is a normalised πActor program in standard form.

Theorem 5.6 (soundness for πActor abstraction)**.** *Given a sound basic domains abstraction and a* Core *program* $\mathcal{M}$*, if* $S, S' \in \mathit{State}$ *such that* $\mathrm{init}(\mathcal{M}) \to S \to S'$ *and* $P \sqsupseteq \alpha_\pi(S)$ *then there exists* $P' \sqsupseteq \alpha_\pi(S')$ *with* $P \overset{\Delta}{\twoheadrightarrow}{}^{*} P'$.

Proof. See appendix C $\square$

6 Applications of the πActor model and model checking

Now that we have constructed our πActor model and proved it sound, we would like to make use of it. One can imagine using the model as a program meaning discovery tool, running it through a π-calculus simulator like Stargazer[7] to explore the program's communication topology. Our primary concern here, however, is the automatic verification of safety properties, expressible by negative coverability queries. Can we use the πActor model for this purpose?

Unfortunately, the π-calculus is Turing powerful, and so reachability and coverability are not decidable for arbitrary π-calculus processes, and since there is an obvious encoding from the π-calculus to πActor, we have that reachability and coverability are also not decidable for πActor.

Theorem 6.1. *Coverability and reachability are, in general, undecidable for πActor programs.*

Proof. either by encoding of π-calculus into πActor or the encoding of a two-counter machine into πActor. □

However, there is a class of π-calculus programs for which coverability is known to be decidable: depth-bounded programs[18].

Definition 6.2 (nest$_{\boldsymbol{\nu}}$)**.** The nesting of restrictions in a process, nest$_{\boldsymbol{\nu}}$, is given by

$$\mathrm{nest}_{\boldsymbol{\nu}}(\mathbf{0}) = \mathrm{nest}_{\boldsymbol{\nu}}(\mathbf{fail}) = \mathrm{nest}_{\boldsymbol{\nu}}(s \in A) = \mathrm{nest}_{\boldsymbol{\nu}}(M) = 0 \qquad \mathrm{nest}_{\boldsymbol{\nu}}(\boldsymbol{\nu}s.P) = 1 + \mathrm{nest}_{\boldsymbol{\nu}}(P)$$

$$\mathrm{nest}_{\boldsymbol{\nu}}(\boldsymbol{\nu}A.P) = 1 + \mathrm{nest}_{\boldsymbol{\nu}}(P) \qquad \mathrm{nest}_{\boldsymbol{\nu}}(P \parallel Q) = \max(\mathrm{nest}_{\boldsymbol{\nu}}(P), \mathrm{nest}_{\boldsymbol{\nu}}(Q))$$

Definition 6.3 (depth)**.** The depth of a term $P \in \mathfrak{T}$ is the minimal nesting of restrictions across all terms its congruence class; $\mathrm{depth}(P) = \min(\{\mathrm{nest}_{\boldsymbol{\nu}}(P') : P \equiv P'\})$

Example 6.4. All of these processes are in the same congruence class:

$$\begin{aligned}
P &= \boldsymbol{\nu} ss'wxyz.(\mathrm{P}[\, s \mid \{w, x, y\}, \{w, z\}\,] \parallel \mathrm{Q}[\, s' \mid \{x\}, \{s\}\,]),\\
&= \boldsymbol{\nu} ss'wxyz.\boldsymbol{\nu} ABCD(\mathrm{P}[\, s \mid A, B\,] \parallel w \in A \parallel x \in A \parallel y \in A \parallel w \in B \parallel z \in B \parallel\\
&\qquad \parallel \mathrm{Q}[\, s' \mid C, D\,] \parallel x \in C \parallel s \in D),\\
P' &= \boldsymbol{\nu} sx.(\boldsymbol{\nu} wyz.\boldsymbol{\nu} AB.(\mathrm{P}[\, s \mid A, B\,] \parallel w \in A \parallel x \in A \parallel y \in A \parallel w \in B \parallel z \in B)\\
&\qquad \parallel \boldsymbol{\nu} s'.\boldsymbol{\nu} CD.(\mathrm{Q}[\, s' \mid C, D\,] \parallel x \in C \parallel s \in D)),\\
P'' &= \boldsymbol{\nu} sx.(\boldsymbol{\nu} AB.(\mathrm{P}[\, s \mid A, B\,] \parallel \boldsymbol{\nu} w.(w \in A \parallel w \in B) \parallel x \in A \parallel \boldsymbol{\nu} y.y \in A \parallel \boldsymbol{\nu} z.z \in B)\\
&\qquad \parallel \boldsymbol{\nu} s'.\boldsymbol{\nu} CD.(\mathrm{Q}[\, s' \mid C, D\,] \parallel x \in C \parallel s \in D)),
\end{aligned}$$

$\mathrm{nest}_{\boldsymbol{\nu}}(P) = 10$, $\mathrm{nest}_{\boldsymbol{\nu}}(P') = 7$, $\mathrm{nest}_{\boldsymbol{\nu}}(P'') = 5$. P'' has the shallowest nesting of restrictions of any process in its congruence class, so $\mathrm{depth}(P) = \mathrm{depth}(P') = \mathrm{depth}(P'') = 5$.

Definition 6.5 (depth-boundedness)**.** A term $P \in \mathfrak{T}$ is depth-bounded if there exists some $k_D \in \mathbb{N}$ such that for every term $P' \in \mathfrak{T}$ such that $P \twoheadrightarrow^* P'$, $\mathrm{depth}(P') \leq k_D$.

Theorem 6.6. *If $P \in \mathfrak{T}$ is depth-bounded, then the coverability problem is decidable for P.*

Proof. Term embedding $\subseteq$ can be shown to be a well quasi-ordering for depth-bounded π-calculus terms, and so depth-bounded π-calculus can be shown to be a Well-Structured Transition System (WSTS) for which coverability is decidable[13, 18, 22]. The proofs for the pure π-calculus can be rewritten for our πActor with only minor adjustments. □

Unfortunately, we also have the following result:

Theorem 6.7. *It is undecidable to determine whether an arbitrary term $P \in \mathfrak{T}$ is depth-bounded.*

We therefore seek a class of terms for which depth-boundedness is decidable, and a way to coerce our πActor model such that it is a member of this class. A good candidate may be hierarchical programs. $\mathcal{T}$-compatibile[10], programs are one such class. $\mathcal{T}$-compatibility is a type system, where typeable terms are depth bounded and typeability is decidable for arbitrary *pi*-calculus processes. Such a type system could be easily modified to operate on πActor terms. Although there has been no formal development of the idea, it has been proposed that $\mathcal{T}$-compatibility could be used to gracefully [11, p. 131].

A model checking procedure could be implemented by first running our analysis to generate a πActor abstraction, then checking if the abstraction is $\mathcal{T} - \textit{compatible}$. If it is (and so the abstraction is depth-bounded), we can encode the model into π-calculus and use a model checker for depth bounded π-calculus such as Picasso[1] to verify the property.

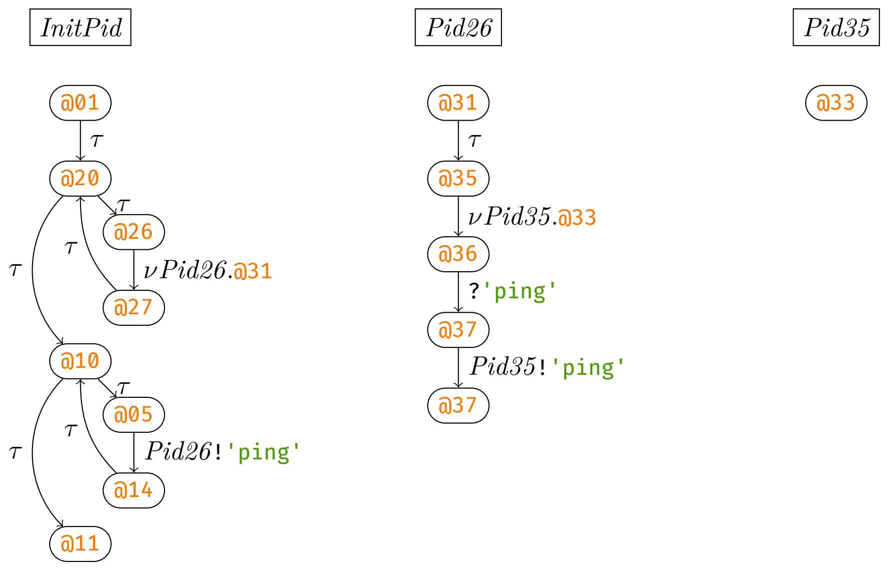

Figure 6: ACS for the program in Listing 2 using a 0-CFA with a 1-depth data abstraction and a mailbox abstraction that ignores the order and number of messages received. Since in a 0-CFA the time domain collapses to a singleton, we omit all reference to it.

7 Evaluation

7.1 Case studies

We start with a very simple example, to demonstrate the πActor abstraction, and the ways in which it differs from our primary point of comparison, the ACS semantics.

Definition 7.1 (generated ACS)**.** [9] Given a Core program $\mathcal{M}$, a sound basic domains abstraction $I = (T, M, D)$ and a sound data abstraction $D_{\text{msg}} = (\widehat{Msg}, \alpha_{\text{msg}}, \widehat{\text{res}}_{\text{msg}})$ for messages, the ACS generated by $\mathcal{M}$, I and D_{msg} is defined as

$$(\widehat{Pid}, \widehat{ProcState}, \widehat{Msg}, R, \alpha_{\text{CFA}}(\iota_0), \alpha_{\text{CFA}}(\pi_0(\iota_0)))$$

where $\text{init}(\mathcal{M}) = \langle \pi_0, \mu_0, \sigma_{v_0}, \sigma_{k_0} \rangle$ and the rules in R are defined by cases as follows. If for $\widehat{S}, \widehat{S}'$ in the abstract semantics of $\mathcal{M}$ and $\widehat{S} \rightsquigarrow \widehat{S}'$ by the rule Abs-R with active components $(\hat{\iota}, \widehat{q}, \widehat{q}')$,

- If Abs-R is Abs-Receive where $\widehat{a}$ is the abstract message matched by $\widehat{\text{mmatch}}$ and $\widehat{m} \in \widehat{\text{res}}_{\text{msg}}(\widehat{\sigma}_v, \widehat{a})$, then
$$\hat{\iota} : \widehat{q} \xrightarrow{?\widehat{m}} \widehat{q}' \in R$$
- If Abs-R is Abs-Send where $\widehat{a}$ is the abstract message that is sent, $\hat{\iota}'$ is the pid-class of the recipient and $\widehat{m} \in \widehat{\text{res}}_{\text{msg}}(\widehat{\sigma}_v, \widehat{a})$, then
$$\hat{\iota} : \widehat{q} \xrightarrow{\hat{\iota}'\,!\,\widehat{m}} \widehat{q}' \in R$$
- If Abs-R is Abs-Spawn where $\hat{\iota}'$ is the pid-class of the spawned process and $\widehat{q}''$ is the process that that becomes associated with it during the transition then
$$\hat{\iota} : \widehat{q} \xrightarrow{\nu\hat{\iota}'.\widehat{q}''} \widehat{q}' \in R$$
- Otherwise,
$$\hat{\iota} : \widehat{q} \xrightarrow{\tau} \widehat{q}' \in R$$

To reduce the complexity of our analysis, we always perform a global store widening on the CFA, as described in [20]. We often use Soter[8] to verify ACS versions of presented examples.

Map send over a list of pids of arbitrary length

Consider first the program in Listing 2. Let $\mathcal{M}$ be the set containing only the module `'main'`. In our analysis, we use the basic domains abstraction $I = (D_0, T_0, M_{\text{set}})$. The generated ACS is given diagrammatically in Figure 6. The πActor abstraction for the program is given by $(\boldsymbol{\nu} s.\texttt{@20}[\,s \mid \emptyset\,], \text{defs}(\mathcal{M}))$ where defs$(\mathcal{M})$ is the following set of definitions:

$$
\begin{aligned}
\texttt{@20}[\,s \mid A\,] &:= \boldsymbol{\tau}.\texttt{@10}[\,s \mid A\,] + \boldsymbol{\tau}.\texttt{@26}_1[\,s \mid A\,] \\
\texttt{@26}_1[\,s \mid A\,] &:= \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\texttt{@35}_1[\,s'\,] \parallel \texttt{@26}_2[\,s, s' \mid A\,]) \\
\texttt{@26}_2[\,s, s' \mid A\,] &:= \boldsymbol{\tau}.\texttt{@20}[\,s \mid A \cup \{s'\}\,] \\
\texttt{@10}[\,s \mid A\,] &:= \boldsymbol{\tau}.\texttt{@11}[\,s\,] + \boldsymbol{\tau}.\texttt{@05}[\,s \mid A\,] \\
\texttt{@11}[\,s\,] &:= \mathbf{0} \\
\texttt{@05}[\,s \mid A\,] &:= \textbf{let } s' \in A.(\textsc{Send}_{\hat{a}}[\,s'\,] \parallel \texttt{@10}[\,s \mid A\,]) \\
\texttt{@35}_1[\,s\,] &:= \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\texttt{@33}[\,s'\,] \parallel \texttt{@35}_2[\,s, s'\,]) \\
\texttt{@35}_2[\,s, s'\,] &:= \boldsymbol{\tau}.\texttt{@36}[\,s \mid \{s'\}\,] \\
\texttt{@36}[\,s \mid B\,] &:= s?a.\texttt{@37}[\,s \mid B\,] \\
\texttt{@37}[\,s \mid B\,] &:= \textbf{let } s' \in B.\textsc{Send}_{\hat{b}}[\,S\,] \\
\texttt{@33}[\,s\,] &:= \mathbf{0}
\end{aligned}
\qquad
\begin{aligned}
\textsc{Send}_{\hat{a}}[\,s\,] &:= s!\hat{a}.\mathbf{0} \\
\textsc{Send}_{\hat{b}}[\,s\,] &:= s!\hat{b}.\mathbf{0}
\end{aligned}
$$

(in the above, we collapse definitions of the form $\textsc{Proc}[\,s \mid \mathit{args}\,] := \boldsymbol{\tau}.\textsc{Proc}'[\,s \mid \mathit{args}\,]$ by replacing all occurrences of $\textsc{Proc}[\,s \mid \mathit{args}\,]$ with $\textsc{Proc}'[\,s \mid \mathit{args}\,]$. We perform this simplification procedure on all πActor listings.)

Remark. The πActor abstraction is quite legible (once the definition collapse has been performed). It discards information about control flow and makes clear the information about communication between processes.

We can see from this example that the translation does just as poorly as the ACS with regards to representing data structures of arbitrary depth; this is expected, since they use the same CFA abstraction to model control flow. However, it performs better with respect to the identities in *Pid35*. In essence, in the πActor abstraction each process maintains its own copy of each process class, and can distinguish when sending between pids that it knows up to their pid class, whereas in the derived ACS, there is a single global copy of each pid-class.

In this example, in the πActor abstraction, processes spawned at label $\texttt{@26}$ executing the process definition $\texttt{@35}_1$ each have a distinct B set, which will only ever contain one name. When that process eventually continues with the definition $\texttt{@37}$, we can be certain that the s' chosen from B will be the same s' that was generated in the body of $\texttt{@35}_1$. We therefore have that processes spawn at label $\texttt{@35}$ executing the definition $\texttt{@33}$ will only ever have one message waiting in their mailbox; that is, the term $\texttt{@33}[\,s\,] \parallel \textsc{Send}_{\hat{b}}[\,s\,] \parallel \textsc{Send}_{\hat{b}}[\,s\,]$ is uncoverable. In the ACS semantics, we cannot even specify this property; the closest we can come is to ask whether the joint mailbox for the pid-class *Pid35* is bounded, which it is not; we can verify this by running Soter on the program in Listing 7 (appendix D).

It is not all good news, however; due to the coarseness of the CFA with regards to data, the initial process $\texttt{@20}[\,s \mid \emptyset\,]$ is not able to distinguish between the processes spawned at $\texttt{@26}$ executing $\texttt{@35}_1$. In the concrete semantics, we have that the mailboxes for processes spawned at $\texttt{@26}$ will only ever contain one message, but in the πActor abstraction those mailboxes may contain arbitrarily many messages.

The πActor program generated for this program is depth-bounded. To see this, consider the reduction sequence:

$$
\begin{aligned}
\boldsymbol{\nu} s.\texttt{@20}[\,s \mid \emptyset\,] &\twoheadrightarrow \boldsymbol{\nu} s.\texttt{@26}_1[\,s \mid \emptyset\,] \twoheadrightarrow \boldsymbol{\nu} ss'.(\texttt{@26}_2[\,s, s' \mid \emptyset\,] \parallel \texttt{@35}_1[\,s'\,]) \twoheadrightarrow \boldsymbol{\nu} ss'.(\texttt{@20}[\,s \mid \{s'\}\,] \parallel \texttt{@35}_1[\,s'\,]) \\
&\twoheadrightarrow^* \boldsymbol{\nu} ss's''.(\texttt{@20}[\,s \mid \{s', s''\}\,] \parallel \texttt{@35}_1[\,s'\,] \parallel \texttt{@35}_1[\,s''\,]) \\
&\equiv \boldsymbol{\nu} A.(\boldsymbol{\nu} s.\texttt{@20}[\,s \mid A\,] \parallel \boldsymbol{\nu} s'.(\texttt{@35}_1[\,s'\,] \parallel s' \in A) \parallel \boldsymbol{\nu} s''.(\texttt{@35}_1[\,s''\,] \parallel s'' \in A))
\end{aligned}
$$

While the set A grows unboundedly in size, the process grows only in breadth, not depth.

```
module 'main'
    [ 'main'/0 = Main ]
    Main = @00: fun () →
        @01: let <Ping> =
            @02: fun (Pid) →
                @03: let <M> = @04: 'ping' in
                @05: primop 'send'/2 (Pid, M)
        in
        @06: let <Pids> = @07: apply GetList () in
        @08: apply Map (Ping, Pids)
    Map = @09: fun (F, L) →
        @10: case <L> of
            <[X | Xs]> when 'true' →
                @12: let <Y> = @13: apply F (X) in
                @14: let <Ys> = @15: apply Map (F, Xs) in
                @16: [Y | Ys]
            <_> when 'true' → @11: []
        end
    GetList = @17: fun () →
        @18: let <B> =
            @19: let <True> = 'true' in
            @20: let <False> = 'false' in
            @21: primop 'choice'/2 (True, False)
        in
        @22: case <B> of
            <'true'> when 'true' → @23: []
            <_> when 'true' →
                @25: let <P> = @26: primop 'spawn'/1 (Proc) in
                @27: let <Ps> = @28: apply GetList () in
                @29: [P | Ps]
        end
    Proc = @30: fun () →
        @31: let <Proc_> = @32: fun () → @33: 'ok' end in
        @34: let <Pid_> = @35: primop 'spawn'/1 (Proc_) in
        @36: receive <A> when 'true' →
            @37: primop 'send'/2 (Pid_, A)
        end
end
```

Listing 2: A program that maps a function that sends a ping message to its argument over a list of pids of arbitrary length.

Locked resources

In the listings for this section, we omit labels where they are not important for the (condensed) translation.

Consider now the program given in Listings 3 to 5. This program is a variant on the Soter example `reslock`, but where many clients communicate with many resources. Note Core's support for the rich dynamic module system of Erlang; the `'reslock'` module is parameterised on another module whose name is given only at runtime.

Let $\mathcal{M}$ be the set that contains only the modules `'main'`, `'cell'` and `'reslock'`. Given the basic domain abstraction $I = (D_0, T_0, M_{\text{set}})$, the πActor abstraction of this program, once it has been simplified, might look like $(\boldsymbol{\nu} s.\textsc{Main}_{\texttt{@01}}[\,s\,], \text{defs}(\mathcal{M}))$ where defs $\mathcal{M}$ contains the following definitions:

$$
\begin{aligned}
&\textsc{Main}_{\texttt{@01}}{}^{1}[\,s\,] := \boldsymbol{\tau}.\textsc{Main}_{\texttt{@01}}{}^{1}[\,s\,] + \boldsymbol{\tau}.\textsc{Main}_{\texttt{@02}}{}^{1}[\,s\,]\\
&\textsc{Main}_{\texttt{@02}}{}^{1}[\,s\,] := \boldsymbol{\tau}.\mathbf{0} + \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@01}}[\,s\,]\\
&\textsc{Main}_{\texttt{@01}}{}^{2}[\,s \mid C\,] :=\\
&\quad \boldsymbol{\tau}.\textsc{Main}_{\texttt{@01}}{}^{2}[\,s \mid C\,] + \boldsymbol{\tau}.\textsc{Main}_{\texttt{@02}}{}^{2}[\,s \mid C\,]\\
&\textsc{Main}_{\texttt{@02}}{}^{2}[\,s \mid C\,] :=\\
&\quad \boldsymbol{\tau}.\textsc{Main}_{\texttt{@02}}{}^{1}[\,s\,] + \boldsymbol{\tau}.\textsc{Main}_{\texttt{@03}}[\,s \mid C\,]\\
&\textsc{Main}_{\texttt{@03}}[\,s \mid C\,] :=\\
&\quad \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\textsc{Reslock}_{\texttt{@08}}[\,s' \mid C\,] \parallel \textsc{Main}_0[\,s, s' \mid C\,])\\
&\textsc{Main}_0[\,s, s' \mid C\,] := \boldsymbol{\tau}.\textsc{Main}_{\texttt{@02}}{}^{2}[\,s \mid C\,]\\
&\textsc{Main}_{\texttt{@04}}[\,s \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@15}}[\,s \mid C\,]\\
&\textsc{Main}_{\texttt{@05}}[\,s \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@13}}[\,s \mid C\,]\\
&\textsc{Main}_{\texttt{@06}}[\,s \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@11}}[\,s \mid C\,]
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{Reslock}_{\texttt{@01}}[\,s\,] :=\\
&\quad \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\textsc{Reslock}_{\texttt{@02}}[\,s'\,] \parallel \textsc{Reslock}_0[\,s, s'\,])\\
&\textsc{Reslock}_0[\,s, s'\,] := \boldsymbol{\tau}.\textsc{Main}_{\texttt{@01}}{}^{2}[\,s \mid \{s\}\,]\\
&\textsc{Reslock}_{\texttt{@02}}[\,s\,] := s\texttt{?a}(S_2').\textsc{Reslock}_{\texttt{@03}}[\,s \mid S_2'\,]\\
&\textsc{Reslock}_{\texttt{@03}}[\,s \mid S_2\,] := \textsc{Reslock}_1[\,s, s \mid S_2\,]\\
&\textsc{Reslock}_1[\,s, s' \mid S_2\,] := \textsc{Reslock}_{\texttt{@04}}[\,s \mid \{s'\}, S_2\,]\\
&\textsc{Reslock}_{\texttt{@04}}[\,s \mid S_0, S_2\,] :=\\
&\quad \textbf{let } s' \in S_2.(\textsc{Send}_{\texttt{b}}[\,s' \mid S_0\,] \parallel \textsc{Reslock}_{\texttt{@05}}[\,s \mid S_0\,])\\
&\textsc{Reslock}_{\texttt{@05}}[\,s \mid S_2\,] :=\\
&\quad s\texttt{?c}(S_3')[S_2 \cap S_3' \neq \emptyset].\textsc{Reslock}_{\texttt{@02}}[\,s\,]\\
&\quad +s\texttt{?d}(S_5')[S_2 \cap S_5' \neq \emptyset].\textsc{Reslock}_{\texttt{@06}}[\,s \mid S_2, S_5'\,]\\
&\quad +s\texttt{?e}(S_4')[S_2 \cap S_4' \neq \emptyset].\textsc{Reslock}_{\texttt{@05}}[\,s \mid S_2\,]\\
&\textsc{Reslock}_{\texttt{@06}}[\,s \mid S_2, S_5\,] :=\\
&\quad \boldsymbol{\tau}.\textsc{Reslock}_2[\,s, s \mid S_2, S_5\,]\\
&\textsc{Reslock}_2[\,s, s' \mid S_2, S_5\,] :=\\
&\quad \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@07}}[\,s \mid \{s'\}, S_2, S_5\,]\\
&\textsc{Reslock}_{\texttt{@07}}[\,s \mid S_1, S_2, S_5\,] :=\\
&\quad \textbf{let } s' \in S_2.(\textsc{Send}_{\texttt{f}}[\,s' \mid S_1\,] \parallel \textsc{Reslock}_{\texttt{@05}}[\,s \mid S_2\,])\\
&\quad +\textbf{let } s' \in S_2.(\textsc{Send}_{\texttt{g}}[\,s' \mid S_1\,] \parallel \textsc{Reslock}_{\texttt{@05}}[\,s \mid S_2\,])
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{Reslock}_{\texttt{@08}}[\,s \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_3[\,s, s \mid C\,]\\
&\textsc{Reslock}_3[\,s, s' \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@09}}[\,s \mid \{s'\}, C\,]\\
&\textsc{Reslock}_{\texttt{@09}}[\,s \mid S_2, C\,] :=\\
&\quad \textbf{let } s' \in C.(\textsc{Send}_{\texttt{a}}[\,r \mid S_2\,] \parallel \textsc{Reslock}_{\texttt{@09}}[\,s \mid C\,])\\
&\textsc{Reslock}_{\texttt{@10}}[\,s \mid C\,] :=\\
&\quad s\texttt{?b}(S_0')[R \cap S_0' \neq \emptyset].\textsc{Main}_{\texttt{@04}}[\,s, s \mid C\,]\\
&\textsc{Reslock}_{\texttt{@11}}[\,s \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_4[\,s, s \mid C\,]\\
&\textsc{Reslock}_4[\,s, s' \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@12}}[\,s \mid \{s'\}, C\,]\\
&\textsc{Reslock}_{\texttt{@12}}[\,s \mid S_3, C\,] := \textbf{let } s' \in C.\textsc{Send}_{\texttt{c}}[\,s' \mid S_3\,]\\
&\textsc{Reslock}_{\texttt{@13}}[\,s \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_5[\,s, s \mid C\,]\\
&\textsc{Reslock}_5[\,s, s' \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@14}}[\,s \mid \{s'\}, C\,]\\
&\textsc{Reslock}_{\texttt{@14}}[\,s \mid S_4, C\,] :=\\
&\quad \textbf{let } s' \in C.(\texttt{e}[\,s' \mid S_4\,] \parallel \textsc{Main}_{@06}[\,s \mid C\,])\\
&\textsc{Reslock}_{\texttt{@15}}[\,s \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_7[\,s, s \mid C\,]\\
&\textsc{Reslock}_7[\,s, s' \mid C\,] := \boldsymbol{\tau}.\textsc{Reslock}_{\texttt{@16}}[\,s \mid \{s'\}, C\,]\\
&\textsc{Reslock}_{\texttt{@16}}[\,s \mid S_5, C\,] :=\\
&\quad \textbf{let } s' \in C.(\textsc{Send}_{\texttt{d}}[\,s' \mid S_5\,] \parallel \textsc{Reslock}_{\texttt{@17}}[\,s \mid C\,])\\
&\textsc{Reslock}_{\texttt{@17}}[\,s \mid C\,] :=\\
&\quad s\texttt{?f}(C')[C \cap C' \neq \emptyset].\textsc{Main}_{\texttt{@05}}[\,s \mid C \cup C'\,]\\
&\quad +s\texttt{?g}(C')[C \cap C' \neq \emptyset].\textsc{Main}_{\texttt{@05}}[\,s \mid C \cup C'\,]
\end{aligned}
$$

We would like to verify that no two processes are executing the critical section at the same time, if those two processes are communicating with the same resource. As with the previous example, the ACS does not give us the means to specify this property. The closest it can get is to ask whether any two processes with pid-class $\textsc{Main}_{\texttt{@03}}$ are executing the critical section simultaneously, which of course there are even in safe runs of the program. The appropriate query for the πActor model is to ask whether any two processes are executing process definitions in the critical section with C arguments that intersect, e.g. $\textsc{Main}_{\texttt{@04}}[\,s \mid \{c\}\,] \parallel \textsc{Reslock}_{\texttt{@14}}[\,s' \mid \{c\}\,]$ is uncoverable, for any pair of process definitions in the critical section. The πActor model can also be used to verify depth-boundedness of mailboxes for the incrementing processes, which for similar reasons cannot be verified on the ACS.

Observe that we can verify these properties because we make use of both halves of mobility:

```
module 'reslock'
    [ 'new'/2 = New
    , 'acquire'/1 = Acquire
    , 'release'/1 = Release
    , 'tell'/2 = Tell
    , 'ask'/2 = Ask
    ]

    New = fun (ModName, InitState) →
        let <Proc> = fun () → apply Unlocked (ModName, InitState) in
        @01: primop 'spawn'/1 (Proc)

    Unlocked = fun (ModName, State) →
        @02: receive <{'acquire', Pid}> when 'true' →
            let <Msg> = {'acquired', @03: primop 'self'/0 ()} in
            do @04: primop 'send'/2 (Pid, Msg)
            apply Locked (ModName, State, Pid)
        end

    Locked = fun (ModName, State, Owner) →
        @05: receive
            <{'release', Pid}> when Pid =:= Owner →
                apply Unlocked (ModName, State)
            <{'request', Pid, Req}> when Pid =:= Owner →
                let <Result> = call ModName : 'handle_request'/2 (State, Req) in
                case <Result> of
                    <{'ok', NewState}> when 'true' →
                        apply Locked (ModName, NewState, Owner)
                    <{'reply', NewState, Reply}> when 'true' →
                        let <Msg> = {'reply', @06: primop 'self'/0 (), Reply} in
                        do @07: primop 'send'/2 (Owner, Msg)
                        apply Locked (ModName, NewState, Owner)
                end
        end

    Acquire = fun (Res) →
        do @09: primop 'send'/2 (Res, {'acquire', @08: primop 'self'/0 ()})
        @10: receive <{'acquired', Res_}> when Res_ =:= Res → 'ok' end

    Release = fun (Res) →
        do @12: primop 'send'/2 (Res, {'release', @11: primop 'self'/0 ()})
        'ok'

    %% Sends a request without waiting for a reply.
    Tell = fun (Res, Req) →
        do @14: primop 'send'/2 (Res, {'request', @13: primop 'self'/0 (), Req})
        'ok'

    %% Sends a request and waits for a reply.
    Ask = fun (Res, Req) →
        do @16: primop 'send'/2 (Res, {'request', @15: primop 'self'/0 (), Req})
        @17: receive <{'reply', Res_, Reply}> when Res_ =:= Res → Reply end
end
```

Listing 3: A module depicting a locked resource behaviour.

```
module 'cell'
    [ 'handle_request'/2 = Handle
    , 'new'/1 = New
    , 'acquire'/1 = Acquire
    , 'release'/1 = Release
    , 'read'/1 = Read
    , 'write'/2 = Write
    ]
    Handle = fun (State, Req) →
        case <Req> of
            <'read'> when 'true' → {'reply', State, State}
            <{'write', NewState}> when 'true' → {'ok', NewState}
        end
    New = fun (InitValue) → call 'reslock' : 'new'/2 ('cell', InitValue)
    Acquire = fun (Cell) → call 'reslock' : 'acquire'/2 (Cell)
    Release = fun (Cell) → call 'reslock' : 'release'/2 (Cell)
    Read = fun (Cell) → call 'reslock' : 'ask'/2 (Cell, 'read')
    Write = fun (Cell, X) → call 'reslock' : 'tell'/2 (Cell, {'write', X})
end
```

Listing 4: A module that instantiates the behaviour in the 'reslock' module with a read/write cell as the enclosed resource.

```
module 'main'
    [ 'main'/0 = Main ]
    Main = fun () →
        let <N> = apply GetPeano () in
        apply Repeat (N, fun () →
            let <Cell> = call 'cell' : 'new'/1 ('zero') in
            let <M> = apply GetPeano () in
            apply Repeat (M, fun () →
               @03: primop 'spawn'/1 (fun → apply Inc (cell))
        ))
    GetPeano = fun () →
        @01: case <primop 'choice'/2 ('true', 'false')> of
            <'true'> when 'true' → 'zero'
            <_> when 'true' → {'succ', apply GetPeano ()}
        end
    % Performs N calls of the nullary function F
    Repeat = fun (N, F) →
        @02: case <N> of
            <{'succ', M}> when 'true' →
                do apply F ()
                call Repeat (M, F)
            <_> when 'true' → 'ok'
        end
    Inc = fun (Cell) →
        do call 'cell' : 'acquire'/1 (Cell)
        %% begin critical section
        @04: let <X> = call 'cell' : 'read'/1 (Cell) in
        @05: do call 'cell' : 'write'/2 (Cell, {'succ', X})
        %% end critical section
        @06: call 'cell' : 'release'/1 (Cell)
end
```

Listing 5: Main module for the reslock example.

processes can learn the names of other processes, but also forget them. Consider the πACTOR analogue of the `Locked`/`Unlocked` routine, process definitions RESLOCK$_{\texttt{@02}}$ through RESLOCK$_{\texttt{@07}}$. When a process makes a successful transition from RESLOCK$_{\texttt{@02}}$, it gains knowledge of the set name S_2, representing the set of pids that the resource considers to own the lock. When it transitions from RESLOCK$_{\texttt{@05}}$ be receiving a `c` message, it forgets the set S_2. We can use this to show that a resource process only ever considers one process to be the owner of the lock at any time.

This πACTOR program is also bounded in depth. At a point in the execution of the program, the set of processes looks something like this: the root process has knowledge of the pid of at most one cell process, and spawns new incrementer processes, immediately forgetting their ids. The resource process is either unlocked, and so knows only its own id, or locked, and knows the id of one incrementer process. There are unboundedly many incrementer processes for each resource, all of which know the pid for that resource. So the term looks something like

$$\begin{aligned}&\boldsymbol{\nu} r_i.(\boldsymbol{\nu} s_0.\textsc{Root}[\,s_0 \mid \{r_i\}\,] \parallel \boldsymbol{\nu} t_{ij}.(\textsc{Res}[\,r_i \mid \{t_{ij}\}\,] \parallel \textsc{Inc}[\,t_{ij} \mid \{r_i\}\,]) \parallel \textstyle\prod_{k\neq j} \boldsymbol{\nu} t_{ik}.\textsc{Inc}[\,t_{ik} \mid \{r_i\}\,])\\ &\parallel \textstyle\prod_{k\neq i} \boldsymbol{\nu} r_k.(\boldsymbol{\nu} t_{kj}.(\textsc{Res}[\,r_k \mid \{t_{kj}\}\,] \parallel \textsc{Inc}[\,t_{kj} \mid \{r_k\}\,]) \parallel \textstyle\prod_{l\neq j} \boldsymbol{\nu} t_{kl}.\textsc{Inc}[\,t_{kj} \mid \{r_k\}\,])\end{aligned}$$

Which, while unbounded in breadth, is certainly bounded in depth.

A note about value addresses

Suppose a process with id *pid0* is running `Foo`:

```
Foo = fun () →
    receive <A> when 'true' →
        receive <B> when 'true' →
            apply F (A, B)
        end
    end
```

We would like to distinguish in the abstraction between the pid stored at `A` and the pid stored at `B`. In most cases we can, but suppose the two are stored at the same value address; perhaps they are being sent by a process running `Bar` with environment $\rho[P \mapsto a]$, store $\sigma_v[a \mapsto \mathit{pid0}]$:

```
Bar = fun (P) →
    let <Q> = primop 'spawn'/1 (G) in
    do primop 'send'/2 (P, Q)
    apply Bar (P)
```

In the current translation, this gives rise to the following process definitions:

$$\begin{aligned}\textsc{Foo}_0[\,s\,] &:= s?\mathtt{q}(Q').\textsc{Foo}_1[\,s \mid Q'\,]\\ \textsc{Foo}_1[\,s \mid Q\,] &:= s?\mathtt{q}(Q').\textsc{Foo}_2[\,s \mid Q \cup Q'\,]\\ \textsc{Foo}_2[\,s \mid Q\,] &:= \boldsymbol{\tau}.\mathrm{F}[\,s \mid Q\,]\\ \textsc{Bar}_0[\,s \mid P\,] &:= \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\mathrm{G}[\,s'\,]|\textsc{Bar}_1[\,s \mid P, \{s'\}\,])\\ \textsc{Bar}_1[\,s \mid P, Q\,] &:= \mathbf{let}\ s' \in P.(\textsc{Send}_{\mathtt{q}}[\,s' \mid Q\,].\textsc{Bar}_2[\,s \mid P, Q\,])\\ \textsc{Bar}_2[\,s \mid P, Q\,] &:= \boldsymbol{\tau}.\textsc{Bar}_0[\,s \mid P\,]\end{aligned}$$

Notice that the continuation as F in the definition of FOO2 has only one set-valued argument; the distinction between the pids received first and second is removed. This distinction might be worth preserving. For instance, suppose `F` is defined along the lines of

```
F = fun (X, Y) → if (X =:= Y) then fail else 'ok' end
```

where

`if` *guard* `then` $\ell_1 : e_1$ `else` $\ell_2 : e_2$ `end` is equivalent to

```
let <$U_1^*$> = 'ok' in
case <$U_1^*$> of
    <'ok'> when $guard$ -> $\ell_1 : e_1$
    <'ok'> when 'true'  -> $\ell_2 : e_2$
end
```

where U_1^* fresh.

The source program is clearly error-free, since we send each pid only once and no two pids are equal. But the translation where

$$\mathrm{F}[\,s \mid Q\,] := [Q \cap Q \neq \emptyset].\mathbf{fail} + \boldsymbol{\tau}.\cdots$$

is clearly not.

In attempting to solve this problem, we notice that value addresses serve two purposes in the concrete semantics: they represent both the shape of the data and the location where it is stored. We can decouple these purposes by increasing the size of the value address space to become

$$\begin{aligned} PidAddr &= (Pid \times \mathcal{L} \times \mathbb{V} \times Time)^2 \times Pid \\ PidAddr &= (Pid \times \mathcal{L} \times \mathbb{V} \times Time)^2 \times Data \times \wp(PidAddr) \end{aligned}$$

where the first $Pid \times \mathcal{L} \times \mathbb{V} \times Time$ represents the pid, program location, variable and time at which the value was first stored and the second $Pid \times \mathcal{L} \times \mathbb{V} \times Time$ represents the pid, program location, variable and time at which the value was most recently assigned. We illustrate by demonstrating the new assignment process. Consider

```
@01: let <X> = @02: primop 'spawn'/1 (F) in
@03: let <Y> = @04: {X} in
@05: 'ok'
```

Then we have the following trace in the concrete semantics:

$$\begin{aligned}
&\langle \pi[\iota \mapsto \langle \texttt{@01}, \rho, c, t\rangle], \mu, \sigma_v, \sigma_k\rangle \\
\to&\langle \pi[\iota \mapsto \langle \texttt{@02}, \rho, \langle \iota, \texttt{@01}, \rho, t\rangle, t], \mu, \sigma_v, \sigma_k^{(1)} = \sigma_k[\langle \iota, \texttt{@01}, \rho, t\rangle \mapsto \textsc{KLet}\langle \texttt{@01}, \texttt{X}, \texttt{@03}, \rho, c\rangle]\rangle \\
\to&\langle (\pi^{(1)} = \pi[\iota' \mapsto ...])[\iota \mapsto \langle \iota', \rho, \langle \iota, \texttt{@01}, \rho, t\rangle, t\rangle], \mu, \sigma_v, \sigma_k^{(1)}\rangle \\
\to&\langle \pi^{(1)}[\iota \mapsto \langle \texttt{@03}, \rho^{(1)} = \rho[\texttt{X} \mapsto \langle \iota, \texttt{@01}, \texttt{X}, t, \iota, \texttt{@01}, \texttt{X}, t, \iota'\rangle], c, t\rangle], \\
&\quad \mu, \\
&\quad \sigma_v^{(1)} = \sigma_v[\langle \iota, \texttt{@01}, \texttt{X}, t, \iota, \texttt{@01}, \texttt{X}, t, \iota'\rangle \mapsto \iota'], \\
&\quad \sigma_k^{(1)}\rangle \\
\to&\langle \pi^{(1)}[\iota \mapsto \langle \texttt{@04}, \rho^{(1)}, \langle \iota, \texttt{@03}, \rho, t\rangle, t\rangle], \mu, \sigma_v^{(1)}, \sigma_k^{(2)} = \sigma_k^{(1)}[\langle \iota, \texttt{@03}, \rho, r\rangle \mapsto \textsc{KLet}\langle \texttt{@03}, \texttt{Y}, \texttt{@05}, \rho, c\rangle\rangle]\rangle \\
\to&\langle \pi^{(1)}[\iota \mapsto \langle \texttt{@05}, \rho^{(1)}[\texttt{X} \mapsto \langle \iota, \texttt{@03}, \texttt{Y}, t, \iota, \texttt{@03}, \texttt{Y}, t, \iota'\rangle], c, t\rangle], \\
&\quad \mu, \\
&\quad \sigma_v^{(1)}[\langle \iota, \texttt{@03}, \texttt{Y}, t, \iota, \texttt{@03}, \texttt{Y}, t, \{\iota'\}, \{\iota'\}\rangle \mapsto \langle \texttt{@04}, [\texttt{X} \mapsto \langle \iota, \texttt{@01}, \texttt{X}, t, \iota, \texttt{@03}, \texttt{Y}, t, \iota'\rangle]\rangle, \\
&\qquad \langle \iota, \texttt{@01}, \texttt{X}, t, \iota, \texttt{@03}, \texttt{Y}, t, \iota'\rangle \mapsto \iota'], \\
&\quad \sigma_k^{(2)}\rangle
\end{aligned}$$

Notice that when the the second assignment happens, the value addresses necessary to reconstruct the value being stored at Y are copied, and the data stored at them duplicated.

Using this new address allocation scheme, the translation above becomes

$$\begin{aligned}
\textsc{Foo}_0[\,s\,] &:= s?\mathtt{q}_1(Q_0').\textsc{Foo}_1[\,s \mid Q_0'\,] \\
\textsc{Foo}_1[\,s \mid Q_0\,] &:= s?\mathtt{q}_2(Q_1').\textsc{Foo}_2[\,s \mid Q_0, Q_1'\,] \\
\textsc{Foo}_2[\,s \mid Q_0, Q_1\,] &:= \boldsymbol{\tau}.\mathrm{F}[\,s \mid Q_0, Q_1\,] \\
\textsc{Bar}_0[\,s \mid P\,] &:= \boldsymbol{\tau}.\boldsymbol{\nu}s'.(\mathrm{G}[\,s'\,] \| \textsc{Bar}_1[\,s \mid P, \{s'\}\,]) \\
\textsc{Bar}_1[\,s \mid P, Q\,] &:= \mathbf{let}\ s' \in P.(\textsc{Send}_{\mathtt{q}}[\,s' \mid Q\,].\textsc{Bar}_2[\,s \mid P, Q\,]) \\
\textsc{Bar}_2[\,s \mid P, Q\,] &:= \boldsymbol{\tau}.\textsc{Bar}_0[\,s \mid P\,] \\
\mathrm{F}[\,s \mid Q_0, Q_1\,] &:= [Q_0 \cap Q_1 \neq \emptyset].\mathbf{fail} + \boldsymbol{\tau}.\cdots
\end{aligned}$$

which is error-free. Formalising this new address allocation procedure and exploring more ways that we can exploit address allocation to gain precision is left as future work.

7.2 Complexity

The price we pay for such a precise abstraction is a drastic increase in complexity. If we use a global store widening, we can compute the πActor abstraction in cubic time with respect to the size of the program[20], the same time complexity as the computation of an ACS. However, model checking for the πActor abstraction (when the relevant problem is even decidable), is currently much less efficient. To the best of our knowledge, no efficient algorithm for computing coverability for depth-bounded systems has been discovered and the exact complexity for checking coverability for depth bounded systems is an open problem, but exponential space is a lower bound[22].

7.3 Related work

In our analysis, we use abstract interpretation to bootstrap our way to a more powerful infinite-state abstraction, which we model-check using an infinite-state procedure. In [4], Colby presents a technique based purely on abstract interpretation that gives rise to a non-uniform analysis (an analysis more precise with respect to pids than our CFA) for concurrent ML. This technique is only effective in programs where the lifetime of processes is clearly linked to a recursive structure in the program.

In [21, 12], powerful frameworks are presented for the abstract interpretation of languages similar to the π-calculus which support non-uniform analyses. Typically, unique pids are represented by sequences of symbols analogous to contours. Standard abstraction techniques are applied until the only source of unboundedness in the model is the length of these contours, which are then abstracted using numerical domains. Such approaches do not yield an abstract model, giving only a yes/don't know answer. However, in principle they could be applied to the πActor programs produced by our analysis.

[15] presents a type system that is able to decide deadlocks for the π-calculus, which is implemented in the TyPiCal tool. In principle, this tool can be applied to πActor models, which amounts to applying the TyPiCal approach to Erlang.

Session types are a very successful approach to specifying and verifying concurrent communication protocols. However, this requires the developer to specify the global protocol that a system should implement, a global type. This is a detailed description of the sequence of communications between the parties involved in the protocol. A well-formed global type implies a number of desirable properties of the protocol. The global type is projected into local types, a type that specifies the actions each single party should implement. The code implementing each party can be type-checked against the local type; if the check is passed, then the whole program conforms to the global type, inheriting its desirable properties. Recently, there has been some effort in finding ways to exploit session types for program analysis/verification by extracting a session type from the program to be verified. [17, 16] apply this approach to Go programs. The first of these extracts types that are a variant of CCS, a predecessor to π-calculus without mobility, and hence models pids more coarsely than our approach. The second is restricted to programs that are essentially finite state, whereas our approach produces a more precise model from any program and can precisely analyse infinite-state models thanks to depth-bounded model checking.

Our analysis is essentially a whole-program approach to verification, and is not very scalable. One can use a modular approach to improve scalability. A popular such approach is Rely/Guarantee, where components of a program are given a Rely condition specifying the changes in the shared environment that the component can tolerate while continuing to function correctly and a Guarantee condition specifying the effects the component has on the shared environment. In the actor model, the Rely/Guarantee framework could be instantiated by considering the contents of actor's mailboxes as the environment. The rely condition for an actor's behaviour would specify conditions that should hold on the contents of its mailbox and the guarantee condition would specify the effects of the actor in terms of messages sent and new actors spawned. In the context of abstract interpretation, this approach simply leads to more efficient methods of computing the fixed point for the CFA[5].

8 Conclusion

Our aim was to develop a sound abstraction for Erlang that is precise with respect to process identities, but for which many safety properties remain decidable. We have been by and large successful in this endeavour.

We have created πActor, a variant of the π-calculus designed for the express purpose of modelling Erlang programs, and shown that a number of interesting safety properties are expressible in terms of coverability queries on πActor programs.

We have formally specified a procedure for extracting a πActor program from a Core program, and proved it to be a sound abstraction of the semantics of the Core program.

We illustrated how the concept of depth-boundedness applies to our πActor model and showed that for non-trivial and relevant example programs, interesting coverability queries are decidable on our model. This includes mutual exclusion for the multiple locked resources example, and mailbox boundedness for a class of programs in the map send example. As such, we have enabled the automatic verification of safety properties of Erlang programs which cannot be expressed on Actor Communicating Systems or other equivalent formalisms. To the best of our knowledge, our approach offers an analysis strategy that is able to be precise with respect to actor addresses to a degree that is superior to any other analysis for the actor model.

8.1 Future work

At present, no tool has been implemented that performs either the model generation procedure or the automatic verification using our model. The implementation of such a tool is left as future work.

In section 7, we hint at a technique for coercing πActor programs into depth boundedness using T-compatibility. Investigating, formalising and implementing this technique is future work. Alternatively, we could investigate alterations to the CFA that cause it to produce a πActor model that is always verifiable.

The πActor model puts us in a good position here to explore the trade-offs between abstraction of the model and precision of properties: the pi-model gives us a precise abstraction, but is not always verifiable. The simpler the source program, the more likely the πActor model is to be in some class of systems with rich decidable properties. For complex programs for which the properties we are interested in are not decidable for the πActor model, all hope might not be lost. We notice the following interesting property: If one deletes all restrictions from the πActor model (or equivalently, moves all restrictions to the highest level), we arrive at a model extremely similar to the ACS, for which, for example, LTL is decidable. It may be possible, by changing the locations of the restrictions in the πActor program (making sure to perform such a procedure in a sound way), to slide between the high precision but low decidability/feasibility of the πActor model and the low precision but high decidability/feasibility of the ACS. Exploring this further is another direction for future work.

9 References

[1] PICASSO: a PI-CAlculus-based Static SOftware analyzer. https://dzufferey.github.io/picasso/pub.html.

[2] Luca Aceto, Kim G Larsen, and Anna Ingólfsdóttir Brics. *An Introduction to Milner's CCS.* BRICS, Department of Computer Science, Aalborg University, 2005. http://www.cs.auc.dk/~luca/SV/intro2ccs.pdf.

[3] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0.3 language specification. 2004.

[4] Christopher Colby and Christopher. Analyzing the communication topology of concurrent programs. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '95*, pages 202–213, New York, New York, USA, 1995. ACM Press. http://portal.acm.org/citation.cfm?doid=215465.215592.

[5] Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. pages 159–179. Springer, Berlin, Heidelberg, 2002. http://link.springer.com/10.1007/3-540-45937-5_13.

[6] Ugo de'Liguoro and Luca Padovani. Mailbox Types for Unordered Interactions. 1 2018. http://arxiv.org/abs/1801.04167.

[7] Emanuele D'Osualdo. Stargazer: A Pi-Calculus Simulator.

[8] Emanuele D'Osualdo, Jonathan Kochems, and C.-H. L. Ong. Soter: an Automatic Safety Verifier for Erlang. In *AGERE! '12*, pages 137–140. ACM, 2012.

[9] Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. Automatic Verification of Erlang-Style Concurrency. 3 2013. http://arxiv.org/abs/1303.2201http://dx.doi.org/10.1007/978-3-642-38856-9_24.

[10] Emanuele D'Osualdo and Luke Ong. A type system for proving depth boundedness in the *pi*-calculus. 2 2015. http://arxiv.org/abs/1502.00944.

[11] Emanuele D'Osualdo. *Verification of Message Passing Concurrent Systems.* PhD thesis, University of Oxford, 2015.

[12] Jérôme Feret. Abstract interpretation of mobile systems. *The Journal of Logic and Algebraic Programming*, 63(1):59–130, 4 2005. https://www.sciencedirect.com/science/article/pii/S1567832604000062.

[13] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 4 2001. http://linkinghub.elsevier.com/retrieve/pii/S030439750000102X.

[14] R. M. Karp and R. E. Miller. Parallel Program Schemata. *Journal of Computer and system Sciences*, page 147–195, 1969.

[15] Naoki Kobayashi and Naoki. A New Type System for Deadlock-Free Processes. In *Proceedings of the 17th international conference on Concurrency Theory*, pages 233–247. Springer-Verlag, 2006. http://link.springer.com/10.1007/11817949_16.

[16] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pages 1137–1148, New York, New York, USA, 2018. ACM Press. http://dl.acm.org/citation.cfm?doid=3180155.3180157.

[17] Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida, Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: liveness and safety for channel-based programming. *ACM SIGPLAN Notices*, 52(1):748–761, 1 2017. http://dl.acm.org/citation.cfm?doid=3093333.3009847.

[18] R Meyer. On boundedness in depth in the *pi*-calculus. In *Foundations of Software Science and Computation Structures*, pages 477–489, 2008.

[19] Robin Milner and David Walker. A Calculus of Mobile Processes, I. 1992. https://ac.els-cdn.com/0890540192900084/1-s2.0-0890540192900084-main.pdf?_tid=11c29b64-003d-11e8-a101-00000aacb360&acdnat=1516712472_00171fbc66e1863f2e06a06d1d28f632.

[20] David Van Horn and Matthew Might. Abstracting Abstract Machines. In *International Conference on Functional Programming*, pages 51–62, 7 2010. http://arxiv.org/abs/1007.4446.

[21] Arnaud Venet. Automatic Determination of Communication Topologies in Mobile Systems. pages 152–167. Springer, Berlin, Heidelberg, 1998. http://link.springer.com/10.1007/3-540-49727-7_9.

[22] Thomas Wies, Damien Zufferey, and Thomas A. Henzinger. Forward Analysis of Depth-Bounded Processes. In *Proceedings of the 13th international conference on Foundations of Software Science and Computational Structures*, pages 94–108. Springer-Verlag, 2010. http://link.springer.com/10.1007/978-3-642-12032-9_8.

"That's all folks!"

– Brian Mitchell

A Concrete reduction rules for the Core semantics

Each rule specifies a transition $\langle \pi, \mu, \sigma_v, \sigma_k \rangle \to \langle \pi', \mu', \sigma_v', \sigma_k' \rangle$. Unless otherwise stated in the "then" part of the rule, we have that $\pi = \pi'$, $\mu = \mu'$, $\sigma_v = \sigma_v'$ and $\sigma_k = \sigma_k'$. We let the variables $v, v_i \in e$ range over irreducible expressions, excluding valuelists and `fail`; that is, atom, nil, tuple, list and lambda expressions. We also assume a known label $\otimes : \texttt{fail}$.

The rule Case depends on the helper function

$$\mathrm{cmatch} : (\mathit{Pat}^* \times \mathit{Guard})^* \times \mathbb{V}^* \times \mathit{Env} \times \mathit{ValueStore} \to (\mathbb{N} \times (\mathbb{V} \rightharpoonup \mathit{ValueAddr}))_\perp$$

which takes a sequence of clauses, a value address and a value store and if there is a match returns the index of the matched clause in the sequence and a substitution witnessing the match. If there is no match, cmatch returns $\perp$. The full definition is given below.

Definition A.1 (cmatch)**.** cmatch is given in terms functions cmatch$'$, pmatch, gmatch and unifiable.

$$\mathrm{cmatch}(\mathit{clauses}, \mathit{names}, \rho, \sigma_v) = \mathrm{cmatch}'(1, \mathit{clauses}, \mathit{names}, \rho, \sigma_v)$$

$$\mathrm{cmatch}'(j, \epsilon, \mathit{names}, \rho, \sigma_v) = \perp$$

$$\begin{aligned}
&\mathrm{cmatch}'(j, (\mathit{pat}_1 \cdot \ldots \cdot \mathit{pat}_n, \mathit{guard}) \cdot \mathit{clauses}, U_1 \cdot \ldots \cdot U_n, \rho, \sigma_v) \\
&= \begin{cases} \langle j, \rho_1 \uplus \cdots \uplus \rho_n \rangle, & \perp \neq \rho_i = \mathrm{pmatch}(\mathit{pat}_i, \rho(U_i), \sigma_v), \\ & \mathrm{gmatch}(\mathit{guard}, \rho \uplus \rho_1 \uplus \cdots \uplus \rho_n, \sigma_v) \\ \mathrm{cmatch}'(j+1, \mathit{clauses}, \mathit{names}, \rho, \sigma_v), & \text{otherwise} \end{cases}
\end{aligned}$$

$$\mathrm{pmatch}(U, a, \sigma_v) = [U \mapsto a]$$

$$\mathrm{pmatch}(U \texttt{ = } \mathit{pat}, a, \sigma_v) = \begin{cases} [U \mapsto a] \uplus \rho & \perp \neq \rho = \mathrm{pmatch}(\mathit{pat}, a, \sigma_v) \\ \perp & \text{otherwise} \end{cases}$$

$$\mathrm{pmatch}(\mathit{atom}, a, \sigma_v) = \begin{cases} [] & \langle \ell : \mathit{atom}, [] \rangle = \sigma_v(a) \\ \perp & \text{otherwise} \end{cases}$$

$$\mathrm{pmatch}(\texttt{[]}, a, \sigma_v) = \begin{cases} [] & \langle \ell : \texttt{[]}, [] \rangle = \sigma_v(a) \\ \perp & \text{otherwise} \end{cases}$$

$$\mathrm{pmatch}(\texttt{\{}\mathit{pat}_1\texttt{,} \ldots \texttt{,}\mathit{pat}_n\texttt{\}}, a, \sigma_v) = \begin{cases} \rho_1 \uplus \cdots \uplus \rho_n & \langle \ell : \texttt{\{}V_1\texttt{,} \ \ldots \texttt{,} \ V_n\texttt{\}}, \rho \rangle = \sigma_v(a), \\ & \perp \neq \rho_i = \mathrm{pmatch}(\mathit{pat}_i, \rho(V_i), \sigma_v) \\ \perp & \text{otherwise} \end{cases}$$

$$\mathrm{pmatch}(\texttt{[}\mathit{pat}_1 \texttt{|} \mathit{pat}_2\texttt{]}, a, \sigma_v) = \begin{cases} \rho_1 \uplus \rho_2 & \langle \ell : \texttt{[}V_1 \texttt{|} V_2\texttt{]}, \rho \rangle = \sigma_v(a), \\ & \perp \neq \rho_i = \mathrm{pmatch}(\mathit{pat}_i, \rho(V_i), \sigma_v) \\ \perp & \text{otherwise} \end{cases}$$

$$\mathrm{gmatch}(\mathit{atom}, \rho, \sigma_v) = (\mathit{atom} = \texttt{'true'})$$

$$\mathrm{gmatch}(U \texttt{ =:= } V, \rho, \sigma_v) = \mathrm{unifiable}(\rho(U), \rho(V), \sigma_v)$$

$$\mathrm{gmatch}(\mathit{guard}_1 \texttt{ \&\& } \mathit{guard}_2, \rho, \sigma_v) = \mathrm{gmatch}(\mathit{guard}_1, \rho, \sigma_v) \wedge \mathrm{gmatch}(\mathit{guard}_2, \rho, \sigma_v)$$

$$\begin{aligned}
\mathrm{unifiable}(a, a', \sigma_v) = {} & (\sigma_v(a) = \sigma_v(a')) \\
& \vee (\sigma_v(a) = \langle \ell : \mathit{atom}, [] \rangle \wedge \sigma_v(a') = \langle \ell' : \mathit{atom}, [] \rangle) \\
& \vee (\sigma_v(a) = \langle \ell : \texttt{[]}, [] \rangle \wedge \sigma_v(a') = \langle \ell' : \texttt{[]}, [] \rangle) \\
& \vee (\sigma_v(a) = \langle \ell : \texttt{\{}U_1\texttt{,} \ldots \texttt{,}U_n\texttt{\}}, \rho \rangle \\
& \quad \wedge \sigma_v(a') = \langle \ell' : \texttt{\{}V_1\texttt{,} \ldots \texttt{,}V_n\texttt{\}}, \rho' \rangle \\
& \quad \wedge (\forall i \in \{1 \ldots n\}. \, \mathrm{unifiable}(\rho(U_i), \rho'(V_i), \sigma_v)) \\
& \vee (\sigma_v(a) = \langle \ell : \texttt{[}U_1 \texttt{|} U_2\texttt{]}, \rho \rangle \\
& \quad \wedge \sigma_v(a') = \langle \ell' : \texttt{[}V_1 \texttt{|} V_2\texttt{]}, \rho' \rangle \\
& \quad \wedge \mathrm{unifiable}(\rho(U_1), \rho'(V_1), \sigma_v) \\
& \quad \wedge \mathrm{unifiable}(\rho(U_2), \rho'(V_2), \sigma_v))
\end{aligned}$$

Name

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell : U, \rho, c, t\rangle \\ \text{then } \pi' = \pi[\iota \mapsto \langle \rho(U), \rho, c, t\rangle]\end{gathered}$$

Apply

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell, \rho, c, t\rangle \\ \ell : \texttt{apply}\ U\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)} \\ \sigma_v(\rho(U)) = \langle \ell', \rho'\rangle \\ \ell' : \texttt{fun (}V'_1\texttt{,}\ldots\texttt{,}V'_n\texttt{)}\ \texttt{->}\ \ell''\ \texttt{end} \\ \rho'' = \rho'[V'_1 \mapsto \rho(V_1), \ldots, V'_n \mapsto \rho(V_n)] \\ \text{then } \pi' = \pi[\iota \mapsto \langle \ell'', \rho'', c, \mathrm{tick}(\ell, t)\rangle]\end{gathered}$$

Call

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell, \rho, c, t\rangle \\ \ell : \texttt{call}\ U_1\ \texttt{:}\ U_2\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)} \\ \sigma_v(\rho(U_1)) = \langle \ell : \mathit{atom}_1, []\rangle \\ \sigma_v(\rho(U_2)) = \langle \ell : \mathit{atom}_2, []\rangle \\ \rho' = \mathrm{modenv}(\mathcal{M}, \mathit{atom}_1) \\ \ell' = \mathrm{exports}(\mathcal{M}, \mathit{atom}_1, \mathit{atom}_2) \\ \ell' : \texttt{fun (}V'_1\texttt{,}\ldots\texttt{,}V'_n\texttt{)}\ \texttt{->}\ \ell''\ \texttt{end} \\ \rho'' = \rho'[V'_1 \mapsto \rho(V_1), \ldots, V'_n \mapsto \rho(V_n)] \\ \text{then } \pi' = \pi[\iota \mapsto \langle \ell'', \rho'', c, \mathrm{tick}(\ell, t)\rangle]\end{gathered}$$

Letrec

$$\begin{gathered}\text{if } \pi(\iota) = q = \langle \ell, \rho, c, t\rangle \\ \ell : \texttt{letrec}\ U_1\ \texttt{=}\ \ell_1\texttt{;}\ \ \ldots\texttt{;}\ U_n\ \texttt{=}\ \ell_n\ \texttt{in}\ \ell' \\ a_i = \mathrm{new}_{\mathrm{ca}}(\iota, U_i, q, \texttt{fun}, \mathrm{shrink}(\rho, \ell_i)) \\ \rho' = \rho[U_1 \mapsto a_1, \ldots, U_n \mapsto a_n] \\ \text{then } \pi' = \pi[\iota \mapsto \langle \ell'', \rho', c, t\rangle] \\ \sigma'_v = \sigma_v[a_1 \mapsto \langle \ell_1, \mathrm{shrink}(\rho', \ell_1)\rangle, \\ \vdots \\ a_n \mapsto \langle \ell_n, \mathrm{shrink}(\rho', \ell_n)\rangle]\end{gathered}$$

Case

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell, \rho, c, t\rangle \\ \ell : \texttt{case}\ \texttt{<}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{>}\ \texttt{of} \\ \texttt{<}\mathit{pat}_{11}\texttt{,}\ldots\texttt{,}\mathit{pat}_{1n}\texttt{>}\ \texttt{when}\ \mathit{guard}_1 \to \ell_1\texttt{;} \\ \vdots \\ \texttt{<}\mathit{pat}_{m1}\texttt{,}\ldots\texttt{,}\mathit{pat}_{mn}\texttt{>}\ \texttt{when}\ \mathit{guard}_m \to \ell_m \\ \texttt{end} \\ \mathit{clause}_i = (\mathit{pat}_{i1} \cdot \ldots \cdot \mathit{pat}_{in}, \mathit{guard}_i) \\ \mathit{clauses} = \mathit{clause}_1 \cdot \ldots \cdot \mathit{clause}_m \\ \mathrm{cmatch}(\mathit{clauses}, U_1 \cdot \ldots \cdot U_n, \rho, \sigma_v) = \langle i, \rho'\rangle \\ \text{then } \pi' = \pi[\iota \mapsto \langle \ell_i, \rho \uplus \rho', c, t\rangle]\end{gathered}$$

Receive

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell, \rho, c, t\rangle \\ \ell : \texttt{receive} \\ \texttt{<}\mathit{pat}_1\texttt{>}\ \texttt{when}\ \mathit{guard}_1 \to \ell_1\texttt{;} \\ \vdots \\ \texttt{<}\mathit{pat}_m\texttt{>}\ \texttt{when}\ \mathit{guard}_m \to \ell_m \\ \texttt{end} \\ \mathit{clause}_i = (\mathit{pat}_i, \mathit{guard}_i) \\ \mathit{clauses} = \mathit{clause}_1 \cdot \ldots \cdot \mathit{clause}_m \\ \mathrm{mmatch}(\mathit{clauses}, \mu(\iota), \rho, \sigma_v) = \langle i, a, \rho', m\rangle \\ \text{then } \pi' = \pi[\iota \mapsto \langle \ell_i, \rho \uplus \rho', c, t\rangle] \\ \mu' = \mu[\iota \mapsto m]\end{gathered}$$

Self

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell : \texttt{primop 'self'/0 ()}, \rho, c, t\rangle \\ \text{then } \pi' = \pi[\iota \mapsto \langle \iota, \rho, c, t\rangle]\end{gathered}$$

Spawn

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell : \texttt{primop 'spawn'/1 (}U\texttt{)}, \rho, c, t\rangle \\ \sigma_v(\rho(U)) = \langle \ell', \rho'\rangle \\ \ell' : \texttt{fun ()}\ \texttt{->}\ \ell''\ \texttt{end} \\ \iota' = \mathrm{new}_{\mathrm{pid}}(\iota, \ell, t) \\ \text{then } \pi' = \pi\ \begin{aligned}[t][\iota &\mapsto \langle \iota', \rho, c, t\rangle \\ \iota' &\mapsto \langle \ell'', \rho', \star, t_0\rangle]\end{aligned}\end{gathered}$$

Send

$$\begin{gathered}\text{if } \pi(\iota) = \langle \ell : \texttt{primop 'send'/2 (}U\texttt{,}V\texttt{)}, \rho, c, t\rangle \\ \sigma_v(\rho(U)) = \iota' \\ \text{then } \pi' = \pi[\iota \mapsto \langle \rho(V), \rho, c, t\rangle] \\ \mu' = \mu[\iota' \mapsto \mathrm{enq}(\rho(V), \mu(\iota'))]\end{gathered}$$

Push-Do

$$\begin{gathered}\text{if } \pi(\iota) = q = \langle \ell : \texttt{do}\ \ell'\texttt{,}\ \ell'', \rho, c, t\rangle \\ \kappa = \textsc{Do}\langle \ell'', \rho, c\rangle \\ c' = \mathrm{new}_{\mathrm{ka}}(\iota, q) \\ \text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho, c', t\rangle] \\ \sigma'_k = \sigma_k[c' \mapsto \kappa]\end{gathered}$$

Pop-Do

$$\begin{gathered}\text{if } \pi(\iota) = \langle v, \rho, c, t\rangle \\ \text{or } \pi(\iota) = \langle f, \rho, c, t\rangle,\ \ f \in \mathit{Pid} \uplus \mathit{ValueAddr} \\ \sigma_k(c) = \textsc{Do}\langle \ell', \rho', c'\rangle \\ \text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho', c', t\rangle]\end{gathered}$$

Push-Let

$$\begin{gathered}
\text{if } \pi(\iota) = q = \langle \ell, \rho, c, t \rangle \\
\ell : \texttt{let <}U_1\texttt{,}\dots\texttt{,}U_n\texttt{> =}\ell'\texttt{, }\ell'' \\
\kappa = \textsc{Let}\langle U_1 \cdot \ldots \cdot U_n, \ell'', \rho, c \rangle \\
c' = \text{new}_{\text{ka}}(\iota, q) \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho, c', t \rangle] \\
\sigma'_k = \sigma_k[c' \mapsto \kappa]
\end{gathered}$$

Pop-Let-Closure

$$\begin{gathered}
\text{if } \pi(\iota) = q = \langle \ell : v, \rho, c, t \rangle \\
\sigma_k(c) = \textsc{Let}\langle U \cdot \epsilon, \ell', \rho', c' \rangle \\
\rho_\ell = \text{shrink}(\rho, \ell) \\
a = \text{new}_{\text{ca}}(\iota, U, q, \text{res}(\sigma_v, \langle \ell, \rho_\ell \rangle), \rho_\ell) \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho'[U \mapsto a], c', t \rangle] \\
\sigma'_v = \sigma_v[a \mapsto \langle \ell, \rho_\ell \rangle)]
\end{gathered}$$

Pop-Let-Pid

$$\begin{gathered}
\text{if } \pi(\iota) = q = \langle \iota', \rho, c, t \rangle \\
\sigma_k(c) = \textsc{Let}\langle U \cdot \epsilon, \ell', \rho', c' \rangle \\
a = \text{new}_{\text{pa}}(\iota, U, q) \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho'[U \mapsto a], c', t \rangle] \\
\sigma'_v = \sigma_v[a \mapsto \iota']
\end{gathered}$$

Pop-Let-ValueAddr

$$\begin{gathered}
\text{if } \pi(\iota) = \langle a, \rho, c, t \rangle \\
\sigma_k(c) = \textsc{Let}\langle U \cdot \epsilon, \ell', \rho', c' \rangle \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho'[U \mapsto a], c', t \rangle]
\end{gathered}$$

Pop-Let-ValueList

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell : \texttt{<}U_1\texttt{,}\dots\texttt{,}U_n\texttt{>}, \rho, c, t \rangle \\
\sigma_k(c) = \textsc{Let}\langle U'_1 \cdot \ldots \cdot U'_n, \ell', \rho', c' \rangle \\
\rho'' = \rho'[U'_1 \mapsto \rho(U_1), \ldots, U'_n \mapsto \rho(U_n)] \\
\text{then } \pi' = \pi[\iota \mapsto \langle \ell', \rho'', c', t \rangle]
\end{gathered}$$

Bad-Apply

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell, \rho, c, t \rangle \\
\ell : \texttt{apply } U \texttt{ (}V_1\texttt{,}\dots\texttt{,}V_n\texttt{)} \\
\sigma_v(\rho(U)) = \langle \ell', \rho' \rangle \\
\ell' \not: \texttt{fun (}V'_1\texttt{,}\dots\texttt{,}V'_n\texttt{) -> }\ell''\texttt{ end} \\
\text{then } \pi' = \pi[\iota \mapsto \langle \otimes, \rho, c, t \rangle]
\end{gathered}$$

Bad-Call

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell, \rho, c, t \rangle \\
\ell : \texttt{call } U_1 \texttt{ : } U_2 \texttt{ (}V_1\texttt{,}\dots\texttt{,}V_n\texttt{)} \\
\text{and } \sigma_v(\rho(U_1)) \neq \langle \ell : \mathit{atom}_1, [] \rangle \\
\text{or } \sigma_v(\rho(U_2)) \neq \langle \ell : \mathit{atom}_2, [] \rangle \\
\text{or } \sigma_v(\rho(U_1)) = \langle \ell : \mathit{atom}_1, [] \rangle \\
\sigma_v(\rho(U_2)) = \langle \ell : \mathit{atom}_2, [] \rangle \\
\text{and } \bot = \text{modenv}(\mathcal{M}, \mathit{atom}_1) \\
\text{or } \bot = \text{exports}(\mathcal{M}, \mathit{atom}_1, \mathit{atom}_2) \\
\text{or } \ell' = \text{modenv}(\mathcal{M}, \mathit{atom}_1) \\
\text{and } \ell' \neq : \texttt{fun (}V'_1\texttt{,}\dots\texttt{,}V'_n\texttt{) -> }\ell''\texttt{ end} \\
\text{then } \pi' = \pi[\iota \mapsto \langle \otimes, \rho, c, t \rangle]
\end{gathered}$$

Bad-Case

$$\text{if } \pi(\iota) = \langle \ell, \rho, c, t \rangle$$

```
ℓ : case <U₁,…,Uₙ> of
        <pat₁₁,…,pat₁ₙ> when guard₁ → ℓ₁;
             ⋮
        <patₘ₁,…,patₘₙ> when guardₘ → ℓₘ
    end
```

$$\begin{gathered}
\mathit{clause}_i = (\mathit{pat}_{i1} \cdot \ldots \cdot \mathit{pat}_{in}, \mathit{guard}_i) \\
\mathit{clauses} = \mathit{clause}_1 \cdot \ldots \cdot \mathit{clause}_m \\
\text{cmatch}(\mathit{clauses}, U_1 \cdot \ldots \cdot U_n, \rho, \sigma_v) = \bot \\
\text{then } \pi' = \pi[\iota \mapsto \langle \otimes, \rho, c, t \rangle]
\end{gathered}$$

Bad-Spawn

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell : \texttt{primop 'spawn'/1 (}U\texttt{)}, \rho, c, t \rangle \\
\sigma_v(\rho(U)) = \langle \ell', \rho' \rangle \\
\ell' \not: \texttt{fun () -> }\ell''\texttt{ end} \\
\text{then } \pi' = \pi[\iota \mapsto \langle \otimes, \rho, c, t \rangle]
\end{gathered}$$

Bad-Send

$$\begin{gathered}
\text{if } \pi(\iota) = \langle \ell : \texttt{primop 'send'/2 (}U\texttt{,}V\texttt{)}, \rho, c, t \rangle \\
\sigma_v(\rho(U)) \neq \iota' \\
\text{then } \pi' = \pi[\iota \mapsto \langle \otimes, \rho, c, t \rangle]
\end{gathered}$$

Bad-Pop-Let

$$\begin{gathered}
\text{if } \pi(\iota) = \langle f, \rho, c, t \rangle \\
\sigma_k(c) = \textsc{Let}\langle U'_1 \cdot \ldots \cdot U'_n, \ell', \rho', c' \rangle \\
\text{and } f \in \mathcal{L},\ f : \texttt{<}U_1\texttt{,}\dots\texttt{,}U_m\texttt{>},\ m \neq n \\
\text{or } f \in \mathcal{L},\ f : v,\ n \neq 1 \\
\text{or } f \in \mathit{Pid} \uplus \mathit{ValueAddr},\ n \neq 1 \\
\text{then } \pi' = \pi[\iota \mapsto \langle \otimes, \rho, c, t \rangle]
\end{gathered}$$

B Abstract reduction rules for the CFA

Each rule specifies a transition $\langle \widehat{\pi}, \widehat{\mu}, \widehat{\sigma}_v, \widehat{\sigma}_k \rangle \to \langle \widehat{\pi}', \widehat{\mu}', \widehat{\sigma}_v', \widehat{\sigma}_k' \rangle$. As with the concrete rules, unless otherwise stated in the "then" part of the rule, we have that $\widehat{\pi} = \widehat{\pi}'$, $\widehat{\mu} = \widehat{\mu}'$, $\widehat{\sigma}_v = \widehat{\sigma}_v'$ and $\widehat{\sigma}_k = \widehat{\sigma}_k'$.

The rules ABS-CASE and ABS-BAD-CASE depend on the helper function

$$\widehat{\text{cmatch}} : (\mathit{Pat}^* \times \mathit{Guard})^* \times \widehat{\mathit{ValueAddr}} \times \widehat{\mathit{ValueStore}} \to \wp((\mathbb{N}, \widehat{\mathit{Env}})_\perp)$$

which takes a sequence of clauses, a value address and a value store and returns a set containing, for each possible match from a resolution of that value address, the matched clause in the sequence and a substitution witnessing the match. If there is a resolution of the address for which there is no match, $\perp$ is also in the returned set. Note that $\widehat{\text{cmatch}}$ discards the ordering of clauses and returns all possible matches for the value addresses provided. This is a simple way of ensuring that the CFA is sound.

Abs-Name

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : U, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \widehat{\rho}(U), \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}]\end{gathered}$$

Abs-Apply

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \ell : \texttt{apply}\ U\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)} \\ \widehat{\sigma}_v(\widehat{\rho}(U)) \ni \langle \ell', \widehat{\rho}'\rangle \\ \ell' : \texttt{fun}\ \texttt{(}V_1'\texttt{,}\ldots\texttt{,}V_n'\texttt{)}\ \rightarrow \ell''\ \texttt{end} \\ \widehat{\rho}'' = \widehat{\rho}'[V_1' \mapsto \widehat{\rho}(V_1), \ldots, V_n' \mapsto \widehat{\rho}(V_n)] \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell'', \widehat{\rho}'', \widehat{c}, \widehat{\text{tick}}(\ell, \hat{t})\rangle\}]\end{gathered}$$

Abs-Call

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \ell : \texttt{call}\ U_1\ \texttt{:}\ U_2\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)} \\ \widehat{\sigma}_v(\widehat{\rho}(U_1)) \ni \langle \ell : \mathit{atom}_1, []\rangle \\ \widehat{\sigma}_v(\widehat{\rho}(U_2)) \ni \langle \ell : \mathit{atom}_2, []\rangle \\ \widehat{\rho}' = \widehat{\text{modenv}}(\mathcal{M}, \mathit{atom}_1) \\ \ell' = \text{exports}(\mathcal{M}, \mathit{atom}_1, \mathit{atom}_2) \\ \ell' : \texttt{fun}\ \texttt{(}V_1'\texttt{,}\ldots\texttt{,}V_n'\texttt{)}\ \rightarrow \ell''\ \texttt{end} \\ \widehat{\rho}'' = \widehat{\rho}'[V_1' \mapsto \widehat{\rho}(V_1), \ldots, V_n' \mapsto \widehat{\rho}(V_n)] \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell'', \widehat{\rho}'', \widehat{c}, \widehat{\text{tick}}(\ell, \hat{t})\rangle\}]\end{gathered}$$

Abs-Letrec

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \widehat{q} = \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \ell : \texttt{letrec}\ U_1\ \texttt{=}\ \ell_1\texttt{;}\ \ldots\texttt{;}\ U_n\ \texttt{=}\ \ell_n\ \texttt{in}\ \ell' \\ \widehat{\sigma}_v(\widehat{\rho}(U)) \ni \langle \ell', \rho'\rangle \\ \widehat{a}_i = \widehat{\text{newva}}_{\text{data}}(\hat{\imath}, U_i, \widehat{q}, \alpha_d(\texttt{fun}), \text{shrink}(\widehat{\rho}, \ell_i)) \\ \widehat{\rho}' = \widehat{\rho}[U_1 \mapsto \widehat{a}_1, \ldots, U_n \mapsto \widehat{a}_n] \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell'', \widehat{\rho}', \widehat{c}, \hat{t}\rangle\}] \\ \widehat{\sigma}_v' = \widehat{\sigma}_v \sqcup [\widehat{a}_1 \mapsto \{\langle \ell_1, \text{shrink}(\widehat{\rho}', \ell_1)\rangle\}, \\ \vdots \\ \widehat{a}_n \mapsto \{\langle \ell_n, \text{shrink}(\widehat{\rho}', \ell_n)\rangle\}]\end{gathered}$$

Abs-Case

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \ell : \texttt{case}\ U\ \texttt{of}\ \mathit{clause}_1\texttt{;}\ \ldots\texttt{;}\mathit{clause}_n\ \texttt{end} \\ \mathit{clauses} = \mathit{clause}_1 \cdot \ldots \cdot \mathit{clause}_n \\ \widehat{\text{cmatch}}(\mathit{clauses}, \widehat{\rho}(U), \widehat{\sigma}_v) \ni \langle i, \widehat{\rho}'\rangle \\ \mathit{clause}_i = \texttt{<}\mathit{pats}_i\texttt{>}\ \texttt{when}\ \mathit{guard}_i\ \rightarrow \ell_i \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell_i, \widehat{\rho} \uplus \widehat{\rho}', \widehat{c}, \hat{t}\rangle\}]\end{gathered}$$

Abs-Receive

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \ell : \texttt{receive}\ \mathit{clause}_1\texttt{;}\ \ldots\texttt{;}\mathit{clause}_n\ \texttt{end} \\ \mathit{clauses} = \mathit{clause}_1 \cdot \ldots \cdot \mathit{clause}_n \\ \widehat{\text{mmatch}}(\mathit{clauses}, \widehat{\mu}(\hat{\imath}), \widehat{\sigma}_v) \ni \langle i, \widehat{a}, \widehat{\rho}', \widehat{m}\rangle \\ \mathit{clause}_i = \texttt{<}\mathit{pat}_i\texttt{>}\ \texttt{when}\ \mathit{guard}_i\ \rightarrow \ell_i \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell_i, \widehat{\rho} \uplus \widehat{\rho}', \widehat{c}, \hat{t}\rangle\}] \\ \widehat{\mu}' = \widehat{\mu} \sqcup [\hat{\imath} \mapsto \widehat{m}]\end{gathered}$$

Abs-Self

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop}\ \texttt{'self'/0}\ \texttt{()}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \hat{\imath}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}]\end{gathered}$$

Abs-Spawn

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop}\ \texttt{'spawn'/1}\ \texttt{(}U\texttt{)}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \widehat{\sigma}_v(\widehat{\rho}(U)) \ni \langle \ell', \widehat{\rho}'\rangle \\ \ell' : \texttt{fun}\ \texttt{()}\ \rightarrow \ell''\ \texttt{end} \\ \hat{\imath} = \langle \ell', \hat{t}'\rangle \\ \hat{\imath}' = \langle \ell, \widehat{\text{tick}}^{\star}(\hat{t}, \widehat{\text{tick}}(\ell', \hat{t}'))\rangle \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \hat{\imath}', \widehat{\rho}, \widehat{c}, \hat{t}\rangle\} \\ \hat{\imath}' \mapsto \{\langle \ell'', \widehat{\rho}', \star, \hat{t}_0\rangle\}]\end{gathered}$$

Abs-Send

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop}\ \texttt{'send'/2}\ \texttt{(}U\texttt{,}V\texttt{)}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \widehat{\sigma}_v(\widehat{\rho}(U)) \ni \hat{\imath}' \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \widehat{\rho}(V), \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}] \\ \widehat{\mu}' = \widehat{\mu} \sqcup [\hat{\imath}' \mapsto \widehat{\text{enq}}(\widehat{\rho}(V), \widehat{\mu}(\hat{\imath}'))]\end{gathered}$$

Abs-Push-Do

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \widehat{q} = \langle \ell : \texttt{do}\ \ell'\texttt{,}\ \ell'', \widehat{\rho}, \widehat{c}, \hat{t}\rangle \\ \widehat{\kappa} = \widehat{\text{Do}}\langle \ell'', \widehat{\rho}, \widehat{c}\rangle \\ \widehat{c}' = \widehat{\text{new}}_{\text{ka}}(\hat{\imath}, \widehat{q}) \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}, \widehat{c}', \hat{t}\rangle\}] \\ \widehat{\sigma}_k' = \widehat{\sigma}_k \sqcup [\widehat{c}' \mapsto \{\widehat{\kappa}\}]\end{gathered}$$

Abs-Pop-Do

$$\begin{gathered}\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle v, \widehat{\rho}, \widehat{c}, \hat{t}\rangle, \\ \text{or } \widehat{\pi}(\hat{\imath}) \ni \langle f, \widehat{\rho}, \widehat{c}, \hat{t}\rangle,\ \ f \in \mathit{Pid} \uplus \mathit{ValueAddr} \\ \widehat{\sigma}_k(\widehat{c}) \ni \widehat{\text{Do}}\langle \ell', \widehat{\rho}', \widehat{c}'\rangle \\ \text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}', \widehat{c}', \hat{t}\rangle\}]\end{gathered}$$

Abs-Push-Let

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \widehat{q} = \langle \ell, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle$$
$$\ell : \texttt{let <}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{> =}\ell'\texttt{,}\ \ell''$$
$$\widehat{\kappa} = \widehat{\text{LET}}\langle U_1 \cdot \ldots \cdot U_n, \ell'', \widehat{\rho}, \widehat{c}\rangle$$
$$\widehat{c}' = \widehat{\text{new}}_{\text{ka}}(\hat{\imath}, \widehat{q})$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}, \widehat{c}', \widehat{t}\rangle\}]$$
$$\widehat{\sigma}_k' = \widehat{\sigma}_k \sqcup [\widehat{c}' \mapsto \{\widehat{\kappa}\}]$$

Abs-Pop-Let-Closure

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \widehat{q} = \langle \ell : v, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_k(\widehat{c}) \ni \widehat{\text{LET}}\langle U \cdot \epsilon, \ell', \widehat{\rho}', \widehat{c}'\rangle$$
$$\widehat{\rho}_\ell = \text{shrink}(\widehat{\rho}, \ell)$$
$$\hat{d} \in \widehat{\text{res}}(\widehat{\sigma}_v, \langle v, \widehat{\rho}_\ell\rangle),$$
$$\widehat{a} = \widehat{\text{newva}}_{\text{data}}(\hat{\imath}, U, \widehat{q}, \hat{d}, \widehat{\rho}_\ell)$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}'[U \mapsto \widehat{a}], \widehat{c}', \hat{t}\rangle\}]$$
$$\widehat{\sigma}_v' = \widehat{\sigma}_v \sqcup [\widehat{a} \mapsto \{\langle \ell, \widehat{\rho}_\ell\rangle\}]$$

Abs-Pop-Let-Pid

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \widehat{q} = \langle \hat{\imath}', \widehat{\rho}, \widehat{c}, \widehat{t}\rangle$$
$$\widehat{\sigma}_k(\widehat{c}) \ni \widehat{\text{LET}}\langle U \cdot \epsilon, \ell', \widehat{\rho}', \widehat{c}'\rangle$$
$$\widehat{a} = \widehat{\text{newva}}_{\text{pid}}(\hat{\imath}, U, \widehat{q})$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}'[U \mapsto \widehat{a}], \widehat{c}', \hat{t}\rangle\}]$$
$$\widehat{\sigma}_v' = \widehat{\sigma}_v \sqcup [\widehat{a} \mapsto \hat{\imath}']$$

Abs-Pop-Let-ValueAddr

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \widehat{a}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_k(\widehat{c}) \ni \widehat{\text{LET}}\langle U \cdot \epsilon, \ell', \widehat{\rho}', \widehat{c}'\rangle$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}'[U \mapsto \widehat{a}], \widehat{c}', \hat{t}\rangle\}]$$

Abs-Pop-Let-ValueList

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{<}U_1\texttt{,}\ldots\texttt{,}U_n\texttt{>}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_k(\widehat{c}) = \widehat{\text{LET}}\langle U_1' \cdot \ldots \cdot U_n', \ell', \widehat{\rho}', \widehat{c}'\rangle$$
$$\widehat{\rho}'' = \widehat{\rho}'[U_1' \mapsto \widehat{\rho}(U_1), \ldots, U_n' \mapsto \widehat{\rho}(U_n)]$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \ell', \widehat{\rho}'', \widehat{c}', \widehat{t}\rangle\}]$$

Abs-Bad-Apply

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle$$
$$\ell : \texttt{apply}\ U\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)}$$
$$\widehat{\sigma}_v(\widehat{\rho}(U)) \ni \langle \ell', \widehat{\rho}'\rangle$$
$$\ell' \not{:} \texttt{fun (}V_1'\texttt{,}\ldots\texttt{,}V_n'\texttt{)} \rightarrow \ell''\ \texttt{end}$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \otimes, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle\}]$$

Abs-Bad-Call

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle$$
$$\ell : \texttt{call}\ U_1\ \texttt{:}\ U_2\ \texttt{(}V_1\texttt{,}\ldots\texttt{,}V_n\texttt{)}$$
$$\text{and } \widehat{\sigma}_v(\widehat{\rho}(U_1)) \ni \langle \ell \not{:} \mathit{atom}_1, []\rangle$$
$$\text{or } \widehat{\sigma}_v(\widehat{\rho}(U_2)) \ni \langle \ell \not{:} \mathit{atom}_2, []\rangle$$
$$\text{or } \widehat{\sigma}_v(\widehat{\rho}(U_1)) \ni \langle \ell : \mathit{atom}_1, []\rangle$$
$$\widehat{\sigma}_v(\widehat{\rho}(U_2)) \ni \langle \ell : \mathit{atom}_2, []\rangle$$
$$\text{and } \bot = \text{modenv}(\mathcal{M}, \mathit{atom}_1)$$
$$\text{or } \bot = \text{exports}(\mathcal{M}, \mathit{atom}_1, \mathit{atom}_2)$$
$$\text{or } \ell' = \text{modenv}(\mathcal{M}, \mathit{atom}_1)$$
$$\text{and } \ell' \neq: \texttt{fun (}V_1'\texttt{,}\ldots\texttt{,}V_n'\texttt{)} \rightarrow \ell''\ \texttt{end}$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \otimes, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle\}]$$

Abs-Bad-Case

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle$$
$$\ell : \texttt{case}\ U\ \texttt{of}\ \mathit{clause}_1\texttt{;}\ \ldots\texttt{;}\mathit{clause}_n\ \texttt{end}$$
$$\mathit{clauses} = \mathit{clause}_1 \cdot \ldots \cdot \mathit{clause}_n$$
$$\widehat{\text{cmatch}}(\mathit{clauses}, \widehat{\rho}(U), \sigma_v) \ni \bot$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \otimes, \widehat{\rho}, \widehat{c}, \widehat{t}\rangle\}]$$

Abs-Bad-Spawn

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop 'spawn'/1 (}U\texttt{)}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_v(\widehat{\rho}(U)) \ni \langle \ell', \widehat{\rho}'\rangle$$
$$\ell' \not{:} \texttt{fun ()} \rightarrow \ell''\ \texttt{end}$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \otimes, \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}]$$

Abs-Bad-Send

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle \ell : \texttt{primop 'send'/2 (}U\texttt{,}V\texttt{)}, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_v(\widehat{\rho}(U)) \ni v, v \notin \widehat{\mathit{Pid}}$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \otimes, \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}]$$

Abs-Bad-Pop-Let

$$\text{if } \widehat{\pi}(\hat{\imath}) \ni \langle f, \widehat{\rho}, \widehat{c}, \hat{t}\rangle$$
$$\widehat{\sigma}_k(\widehat{c}) \ni \widehat{\text{LET}}\langle U_1' \cdot \ldots \cdot U_n', \ell', \widehat{\rho}', \widehat{c}'\rangle$$
$$\text{and } f \in \mathcal{L},\ f : \texttt{<}U_1\texttt{,}\ldots\texttt{,}U_m\texttt{>},\ m \neq n$$
$$\text{or } f \in \mathcal{L},\ f : v,\ n \neq 1$$
$$\text{or } f \in \widehat{\mathit{Pid}} \uplus \widehat{\mathit{ValueAddr}},\ n \neq 1$$
$$\text{then } \widehat{\pi}' = \widehat{\pi} \sqcup [\hat{\imath} \mapsto \{\langle \otimes, \widehat{\rho}, \widehat{c}, \hat{t}\rangle\}]$$

C Proof of soundness for the πActor model

Theorem 5.6 (soundness for πActor abstraction)**.** *Given a sound basic domains abstraction and a* Core *program $\mathcal{M}$, if $S, S' \in State$ such that* $\text{init}(\mathcal{M}) \to S \to S'$ *and $P \sqsupseteq \alpha_\pi(S)$ then there exists $P' \sqsupseteq \alpha_\pi(S')$ with $P \overset{\Delta}{\twoheadrightarrow}{}^{*} P'$.*

Proof. Readily follows from Theorems C.1 and C.2. □

Theorem C.1. *for any π*Actor *program (P, Δ), if $P \sqsubseteq P'$ and $P \twoheadrightarrow^{\Delta} Q$ then there exists Q' with $P' \twoheadrightarrow^{\Delta} Q'$ and $Q' \sqsupseteq Q$.*

Proof. Readily follows from Definitions 2.23 and 2.26. □

Theorem C.2. *For some program $\mathcal{M}$, let $S, S' \in State$ such that* $\text{init}(\mathcal{M}) \to^* S \to S'$*. Then there exists $P \in \mathfrak{T}$ with $\alpha_\pi(S) \twoheadrightarrow^{\text{defs}(\mathcal{M})} P$ and $P \sqsupseteq \alpha_\pi(S')$.*

It is Theorem C.2 that we will spend the rest of this section proving.

C.1 A few useful lemmas

Let $S = \langle \pi, \mu, \sigma_v, \sigma_k \rangle \in State$, $S' = \langle \pi', \mu', \sigma'_v, \sigma'_k \rangle \in State$ be concrete states in the semantics of the program $\mathcal{M}$. We are interested in proving Theorem 5.6. We start by proving a few useful lemmas.

Lemma C.3. $\text{pidaddrs}(a) \subseteq \text{pidaddrs}(a') \implies \text{pidaddrs}(\alpha_{CFA}(a)) \subseteq \text{pidaddrs}(\alpha_{CFA}(a'))$

Proof. Trivial by construction. □

Lemma C.4. *If*

$$\text{pidaddrs}(q') \subseteq \text{pidaddrs}(q)$$
$$\sigma_v \cap \text{pidaddrs}(q') = \sigma'_v \cap \text{pidaddrs}(q')$$

then if

$$args = \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{CFA}(q))\}, <)$$
$$args' = \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{CFA}(q'))\}, <)$$

we have

$$args'[\text{args}(q)/args] \sqsupseteq \text{args}(q').$$

Proof. Let $\text{args}(q) = T_1 \cdot \ldots \cdot T_p$, $\text{args}(q') = T'_1 \cdot \ldots \cdot T'_q$ where

$$\widehat{a}_1 \cdot \ldots \cdot \widehat{a}_p = \text{sort}(\text{pidaddrs}(\alpha_{\text{CFA}}(q)), \widehat{<_{\text{VA}}})$$
$$T_i = \{\text{name}(\iota) : a = \langle \_, \_, \_, \iota \rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a) = \widehat{a}_i\}$$
$$\widehat{a}'_1 \cdot \ldots \cdot \widehat{a}'_q = \text{sort}(\text{pidaddrs}(\alpha_{\text{CFA}}(q')), \widehat{<_{\text{VA}}})$$
$$T'_i = \{\text{name}(\iota) : a = \langle \_, \_, \_, \iota \rangle \in \text{pidaddrs}(q'), \alpha_{\text{CFA}}(a) = \widehat{a}'_i\}$$

Now Lemma C.3 gives us that $\text{pidaddrs}(\alpha_{\text{CFA}}(q')) \subseteq \text{pidaddrs}(\alpha_{\text{CFA}}(q))$, which in turn gives us $\forall i \exists j. \widehat{a}'_i = \widehat{a}_j$ and $A \in args' \implies A \in args$. $A_1 \cdot \ldots \cdot A_n = args \implies (A_i = A_j \iff i = j)$, so

$$args' = \text{filter}(args, (\lambda A.\{\text{addr}(A) \in \text{pidaddrs}(\alpha_{\text{CFA}}(q'))\})),$$
$$args'[\text{args}(q)/args] = T''_1 \cdot \ldots \cdot T''_q,$$
$$T''_i = \{\text{name}(\iota) : a = \langle \_, \_, \_, \iota \rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a) = \widehat{a}'_i\}.$$

So if $T'_i \subseteq T''_i$, we are done.

Suppose $a \in \text{pidaddrs}(q')$, $\alpha_{\text{CFA}}(a) = \widehat{a}'_i$. Then $a \in \text{pidaddrs}(q)$, $\alpha_{\text{CFA}}(a) = \widehat{a}'_i$. Now

$$\sigma_v \cap \text{pidaddrs}(q') = \sigma'_v \cap \text{pidaddrs}(q')$$

gives us $\sigma_v(a) = \iota \iff \sigma'_v(a) = \iota$. So $s \in T'_i \implies s \in T''_i$ and $T'_i \subseteq T''_i$. □

Corollary C.5. *If*

$$\begin{gathered}\mathrm{pidaddrs}(q') = \mathrm{pidaddrs}(q) \\ \sigma_v \cap \mathrm{pidaddrs}(q') = \sigma'_v \cap \mathrm{pidaddrs}(q')\end{gathered}$$

then $\mathrm{args}(q) = \mathrm{args}(q')$.

Proof. Let

$$\begin{aligned} \mathit{args} &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <), \\ \mathit{args}' &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q'))\}, <). \\ \mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q'), &\quad \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q')) = \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q)) \\ \Longrightarrow \mathrm{pidaddrs}(q') \subseteq \mathrm{pidaddrs}(q), &\quad \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q')) \subseteq \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q)) \end{aligned}$$

so by Lemma C.4, we have

$$\mathit{args}'[\mathrm{args}(q)/\mathit{args}] \sqsupseteq \mathrm{args}(q').$$

Let

$$\begin{aligned} \widehat{a}_1 \cdot \ldots \cdot \widehat{a}_p &= \mathrm{sort}(\mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q)), \widehat{<_{\mathrm{VA}}}), \\ \widehat{a}'_1 \cdot \ldots \cdot \widehat{a}'_m &= \mathrm{sort}(\mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q')), \widehat{<_{\mathrm{VA}}}). \end{aligned}$$

Clearly if $\mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q)) = \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q'))$, then by Lemma C.3 $p = q$ and $\widehat{a}_i = \widehat{a}'_i$. So $\mathit{args} = \mathit{args}'$ and

$$\mathit{args}'[\mathrm{args}(q)/\mathit{args}] = \mathrm{args}(q).$$

Hence $\mathrm{args}(q) \sqsupseteq \mathrm{args}(q')$. But this argument in symmetric in q, q', σ_v and σ'_v, so we have also that $\mathrm{args}(q) \sqsubseteq \mathrm{args}(q')$. So $\mathrm{args}(q) = \mathrm{args}(q')$ (this is easy to check from the definition of $\sqsubseteq$). □

Lemma C.6. *Suppose that for some finite set of pids I, for $\iota \in \mathrm{dom}\,\pi$ such that $\iota \notin I$, we have $\pi(\iota) = \pi'(\iota)$ and*

$$\sigma_v \cap \mathrm{pidaddrs}(\pi(\iota)) = \sigma'_v \cap \mathrm{pidaddrs}(\pi(\iota)).$$

Let $\mathrm{dom}\,\pi \setminus I = \{\iota_1, \ldots, \iota_n\}$, $\mathrm{dom}\,\pi \cap I = \{\iota'_1, \ldots, \iota'_m\}$, $\mathrm{dom}\,\pi' \cap I = \{\iota''_1, \ldots, \iota''_l\}$. Let $\vec{s} = \mathrm{name}(\iota_1) \cdot \ldots \cdot \mathrm{name}(\iota_n)$, $\vec{s}' = \mathrm{name}(\iota'_1) \cdot \ldots \cdot \mathrm{name}(\iota'_m)$, $\vec{s}'' = \mathrm{name}(\iota''_1) \cdot \ldots \cdot \mathrm{name}(\iota''_l)$. Then

$$\begin{gathered} \alpha_\pi(S) = \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}'.(\alpha_\pi(\pi \cap I) \parallel \mathit{Processes} \parallel \alpha_\pi(\mu)) \\ \alpha_\pi(S') = \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}''.(\alpha_\pi(\pi' \cap I) \parallel \mathit{Processes} \parallel \alpha_\pi(\mu')) \\ \mathit{Processes} = \alpha_\pi(\pi \cap (\mathrm{dom}\,\pi \setminus I)) \end{gathered}$$

Proof. Let $\mathrm{dom}\,\pi \setminus I = J$, $\mathrm{dom}\,\pi \cap I = I'$, $\mathrm{dom}\,\pi' \cap I = I''$. We have that for $\iota \in J$, $\alpha_\pi(\pi(\iota), \iota) = \alpha_\pi(\pi'(\iota), \iota)$ by Corollary C.5. Now

$$\begin{aligned} \alpha_\pi(\pi) &= \prod_{\iota \in \mathrm{dom}\,\pi} \alpha_\pi(\pi(\iota), \iota) \\ &= (\prod_{\iota \in I'} \alpha_\pi(\pi(\iota), \iota)) \parallel \prod_{\iota \in J} \alpha_\pi(\pi(\iota), \iota) \\ &= \alpha_\pi(\pi \cap I) \parallel \mathit{Processes} \\ \alpha_\pi(\pi') &= \prod_{\iota \in \mathrm{dom}\,\pi} \alpha_\pi(\pi'(\iota), \iota) \\ &= (\prod_{\iota \in I''} \alpha_\pi(\pi'(\iota), \iota)) \parallel \prod_{\iota \in J} \alpha_\pi(\pi'(\iota), \iota) \\ &= (\prod_{\iota \in I''} \alpha_\pi(\pi'(\iota), \iota)) \parallel \prod_{\iota \in J} \alpha_\pi(\pi(\iota), \iota) \\ &= \alpha_\pi(\pi' \cap I) \parallel \mathit{Processes} \end{aligned}$$

So

$$\begin{aligned}\alpha_\pi(S) &= \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}'.\,(\alpha_\pi(\pi) \parallel \alpha_\pi(\mu)) \\ &= \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}'.(\alpha_\pi(\pi \cap I) \parallel \mathit{Processes} \parallel \alpha_\pi(\mu)) \\ \alpha_\pi(S') &= \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}''.\,(\alpha_\pi(\pi') \parallel \alpha_\pi(\mu')) \\ &= \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}''.(\alpha_\pi(\pi' \cap I) \parallel \mathit{Processes} \parallel \alpha_\pi(\mu'))\end{aligned}$$ □

Lemma C.7. *Suppose that for some finite set of pids I, for $\iota \in \operatorname{dom}\mu$ such that $\iota \notin I$, we have $\mu(\iota) = \mu'(\iota)$ and for $a \in \mu(\iota)$,*

$$\sigma_v \cap \operatorname{pidaddrs}(a) = \sigma'_v \cap \operatorname{pidaddrs}(a).$$

Let $\operatorname{dom}\mu \setminus I = J$. Then

$$\begin{aligned}\alpha_\pi(\mu) &= \alpha_\pi(\mu \cap I) \parallel \mathit{Mail} \\ \alpha_\pi(\mu) &= \alpha_\pi(\mu' \cap I) \parallel \mathit{Mail} \\ \mathit{Mail} &= \alpha_\pi(\mu \cap J)\end{aligned}$$

Proof. As above. Use Corollary C.5 and equivalence for π-calculus terms. □

Lemma C.8. *and for some finite set of pids I, for $\iota \in \operatorname{dom}\pi$ such that $\iota \notin I$, we have $\pi(\iota) = \pi'(\iota)$ and*

$$\sigma_v \cap \operatorname{pidaddrs}(\pi(\iota)) = \sigma'_v \cap \operatorname{pidaddrs}(\pi(\iota)),$$

and for another finite set of pids J, for $\iota \in \operatorname{dom}\mu$ such that $\iota \notin I$, we have $\mu(\iota) = \mu'(\iota)$ and for $a \in \mu(\iota)$,

$$\sigma_v \cap \operatorname{pidaddrs}(a) = \sigma'_v \cap \operatorname{pidaddrs}(a),$$

there exists $P \in \mathfrak{T}$ with

$$\begin{aligned}\alpha_\pi(\pi \cap I) \parallel \alpha_\pi(\mu \cap J) &\twoheadrightarrow P, \\ P &\sqsupseteq \alpha_\pi(\pi' \cap I) \parallel \alpha_\pi(\mu' \cap J),\end{aligned}$$

then there exists P' such that $\alpha_\pi(S) \twoheadrightarrow P'$, $P' \sqsupseteq \alpha_\pi(S')$.

Proof. Let $\operatorname{dom}\pi \setminus I = J = \{\iota_1, \ldots, \iota_n\}$, $\operatorname{dom}\pi \cap I = I' = \{\iota'_1, \ldots, \iota'_m\}$, $\operatorname{dom}\pi' \cap I = I'' = \{\iota''_1, \ldots, \iota''_l\}$, Let $\vec{s} = \operatorname{name}(\iota_1) \cdot \ldots \cdot \operatorname{name}(\iota_n)$, $\vec{s}' = \operatorname{name}(\iota'_1) \cdot \ldots \cdot \operatorname{name}(\iota'_m)$, $\vec{s}'' = \operatorname{name}(\iota''_1) \cdot \ldots \cdot \operatorname{name}(\iota''_l)$. By Lemmas C.6 and C.7 we have that

$$\begin{aligned}\alpha_\pi(S) &= \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}'.(\alpha_\pi(\pi \cap I) \parallel \alpha_\pi(\mu \cap J) \parallel \mathit{Processes} \parallel \mathit{Mail}) \\ \alpha_\pi(S') &= \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}''.(\alpha_\pi(\pi' \cap I) \parallel \alpha_\pi(\mu' \cap J) \parallel \mathit{Processes} \parallel \mathit{Mail})\end{aligned}$$

$\alpha_\pi(\pi \cap I) \parallel \alpha_\pi(\mu \cap J) \twoheadrightarrow P$, $P \sqsupseteq \alpha_\pi(\pi' \cap I) \parallel \alpha_\pi(\mu' \cap J)$ implies

$$\begin{gathered}P \equiv \boldsymbol{\nu}\vec{s}'''.((\prod_{\iota \in \operatorname{dom}\pi \cap I} P_\iota) \parallel (\prod_{\iota \in \operatorname{dom}\mu \cap J} M_\iota) \parallel R) \text{ where} \\ \forall \iota \in I,\ P_\iota \sqsupseteq \alpha_\pi(\pi'(\iota), \iota) \\ \forall \iota \in J,\ M_\iota \sqsupseteq \prod_{a \in \mu'(\iota)} \textsc{Send}_{\alpha_{\text{CFA}}(a)}[\,\operatorname{name}(\iota) \mid \operatorname{args}(a)\,] \\ \alpha_\pi(S) \twoheadrightarrow P' = \boldsymbol{\nu}\vec{s}.\boldsymbol{\nu}\vec{s}'.\boldsymbol{\nu}\vec{s}'''.((\prod_{\iota \in \operatorname{dom}\pi \cap I} P_\iota) \parallel (\prod_{\iota \in \operatorname{dom}\mu \cap J} M_\iota) \parallel R \parallel \mathit{Processes} \parallel \mathit{Mail})\end{gathered}$$

And clearly this $P' \sqsupseteq \alpha_\pi(S')$. □

Definition C.9. A value address a is relevant in some store σ_v if $a \in \operatorname{dom}\sigma_v$ and $\sigma_v(a) \in \mathit{Pid}$.

Remark C.9.1. Colloquially, a value address is relevant in a store if a pid is stored at that value address in that store.

Lemma C.10. *If* $S = \langle \pi, \mu, \sigma_v, \sigma_k \rangle$ *is such that* $\mathrm{init}(\mathcal{M}) \to^* S$ *for some program* $\mathcal{M}$*, then*

$$\sigma_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma_v \wedge a \in \mathit{PidAddr}$$

Proof. Proceed by induction on the number of transitions from $\mathrm{init}(\mathcal{M})$ to S. That is, if $\mathrm{init}(\mathcal{M}) \to^* S$ then $\mathrm{init}(\mathcal{M}) \to^n S$ for some $n \in \mathbb{N}$, and we induct over n.

Base case: Suppose $S = \mathrm{init}(\mathcal{M})$. Recall Definition 3.4:

Definition 3.4 (init)**.** Let the initial state of a program $\mathcal{M}$, $\mathrm{init}(\mathcal{M})$, be $\langle \pi_0, \mu_0, \sigma_{v_0}, \sigma_{k_0} \rangle$ where

$$\begin{aligned}
\ell &= \mathrm{exports}(\mathcal{M}, \texttt{'main'}, \texttt{'main'/0}), \quad \ell : \texttt{fun ()} \to \ell_0 : \ldots \texttt{ end},\\
\pi_0 &= [\iota_0 \mapsto \langle \ell_0, \mathrm{modenv}(\mathcal{M}, \texttt{'main'}), \star, t_0 \rangle],\\
\mu_0 &= [\iota_0 \mapsto \epsilon],\\
G &= \{(\mathit{atom}, U, \ell) : \texttt{module } \mathit{atom} \texttt{ [...]} \ldots U \texttt{ = } \ell : \ldots \ldots \texttt{end} \in \mathcal{M}\},\\
\sigma_{v_0} &= \biguplus_{(\mathit{atom}, U, \ell) \in G} [\langle \iota_0, U, t_0, \texttt{fun}, \emptyset \rangle \mapsto \langle \ell, \mathrm{modenv}(\mathcal{M}, \mathit{atom}) \rangle],\\
\sigma_{k_0} &= [\star \mapsto \textsc{Stop}],
\end{aligned}$$

Clearly, $\operatorname{Im} \sigma_{v_0} \cap \mathit{Pid} = \emptyset$ and $\operatorname{dom} \sigma_v \cap \mathit{PidAddr} = \emptyset$.

Inductive step: Suppose $\mathrm{init}(\mathcal{M}) \to^{n-1} S' = \langle \pi', \mu', \sigma'_v, \sigma'_k \rangle$ with $S' \to S$. Then

$$\begin{aligned}
&(\sigma'_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma'_v \wedge a \in \mathit{PidAddr})\\
\implies &(\sigma_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma_v \wedge a \in \mathit{PidAddr}).
\end{aligned}$$

Proceed by case analysis on the rule used to make the transition $S' \to S$. If $\sigma'_v = \sigma_v$, the implication follows trivially. So we need only consider rules that alter the value store, i.e. Letrec, Pop-Let-Closure and Pop-Let-Pid.

- If the transition was made by Letrec, then

  $$\begin{aligned}
  \sigma_v = \sigma'_v[a_1 &\mapsto \langle \ell_1, \mathrm{shrink}(\rho', \ell_1) \rangle,\\
  &\vdots\\
  a_n &\mapsto \langle \ell_n, \mathrm{shrink}(\rho', \ell_n) \rangle]
  \end{aligned}$$

  where $a_i = \mathrm{newva}_{\mathrm{data}}(\iota, U_i, q, \texttt{fun}, \mathrm{shrink}(\rho, \ell_i))$. So σ_v differs from σ'_v at at most $a_1, \ldots, a_n$. Clearly, $a_1, \ldots, a_n$ do not have $\sigma_v(a_i) \in \mathit{Pid}$ and $a_1, \ldots, a_n \notin \mathit{PidAddr}$. Assume

  $$\sigma'_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma'_v \wedge a \in \mathit{PidAddr}.$$

  Then

  $$a \notin \mathit{PidAddr} \implies \sigma'_v(a) \notin \mathit{Pid}.$$

  Hence $a_1, \ldots, a_n$ not relevant in σ'_v. So since σ_v differs from σ'_v at at most $a_1, \ldots, a_n$ and $a_1, \ldots, a_n$ not relevant in either σ_v or σ'_v, and

  $$\sigma'_v(a) \in \mathit{Pid} \implies a \in \mathit{PidAddr},$$

  if $\sigma_v(a') \in \mathit{Pid}$, we have $a' \notin \{a_1, \ldots a_n\}$ so $\sigma_v(a') = \sigma'_v(a')$ and hence $\sigma'_v(a') \in \mathit{Pid}$, so $a' \in \mathit{PidAddr}$. So

  $$\sigma_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma_v \wedge a \in \mathit{PidAddr}.$$

- If the transition was made by Pop-Let-Closure, then $\sigma_v = \sigma'_v[a \mapsto \langle \ell, \mathrm{shrink}(\rho, \ell) \rangle]$ where $a = \mathrm{newva}_{\mathrm{data}}(\iota, U, q, \mathrm{res}(\sigma_v, \langle \ell, \mathrm{shrink}(\rho, \ell) \rangle), \mathrm{shrink}(\rho, \ell))$. So σ_v differs from σ'_v at at most a. Clearly, $\sigma_v(a) \notin \mathit{Pid}$ and $a \notin \mathit{PidAddr}$. Again, assuming

  $$\sigma'_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma'_v \wedge a \in \mathit{PidAddr}.$$

  we have that $\sigma_v(a) \notin \mathit{Pid}$. So since σ_v differs from σ'_v at at most a and a not relevant in either σ_v or σ'_v, it is clear that

  $$\sigma_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma_v \wedge a \in \mathit{PidAddr}.$$

- If the transition was made by Pop-Let-Pid, then $\sigma_v = \sigma'_v[a \mapsto \iota']$ where $a = \mathrm{newva}_{\mathrm{pid}}(\iota, U, q)$. So σ_v differs from σ'_v at at most a. Clearly, $\sigma_v(a) \in \mathit{Pid}$ and $a \in \mathit{PidAddr}$. Assume

  $$\sigma'_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma'_v \wedge a \in \mathit{PidAddr}.$$

  So if $\sigma_v(a') \in \mathit{Pid}$, either $a' = a$ and $a \in \mathit{PidAddr}$ by definition or $a' \neq a$ and $\sigma_v(a') = \sigma'_v(a')$ so $\sigma'_v(a') \in \mathit{Pid}$ and $a' \in \mathit{PidAddr}$. So

  $$\sigma_v(a) \in \mathit{Pid} \iff a \in \operatorname{dom} \sigma_v \wedge a \in \mathit{PidAddr}. \qquad \square$$

Corollary C.11. *If $S = \langle \pi, \mu, \sigma_v, \sigma_k \rangle$, $S' = \langle \pi', \mu', \sigma'_v, \sigma'_k \rangle \in \mathit{State}$ such that $\mathrm{init}(\mathcal{M}) \to^* S$, $\mathrm{init}(\mathcal{M}) \to^* S'$. Then if $S \to S'$ by any rule other than* Pop-Let-Pid*, $\sigma_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\} = \sigma'_v \cap \{a : \sigma'_v(a) \in \mathit{Pid}\}$.*

Proof. Clearly if $\sigma_v = \sigma'_v$ the result holds trivially. So we need only consider the rules that alter σ_v, i.e. Letrec, Pop-Let-Closure and Pop-Let-Pid. By Lemma C.10 and the contrapositive, we have that

$$a \notin \mathit{PidAddr} \implies \sigma_v(a) \notin \mathit{Pid}, \qquad (1)$$
$$a \notin \mathit{PidAddr} \implies \sigma'_v(a) \notin \mathit{Pid}. \qquad (2)$$

- If the transition was made by Letrec, then

  $$\begin{aligned}\sigma'_v = \sigma_v[a_1 &\mapsto \langle \ell_1, \mathrm{shrink}(\rho', \ell_1)\rangle,\\ &\vdots\\ a_n &\mapsto \langle \ell_n, \mathrm{shrink}(\rho', \ell_n)\rangle]\end{aligned}$$

  where $a_i = \langle \iota, U_i, \texttt{fun}, t\rangle$. So σ'_v differs from σ_v at at most $a_1, \ldots, a_n$. By 1 and 2, we have that $a_1, \ldots, a_n$ not relevant in σ_v or σ'_v. Assume $a' \in \mathrm{dom}(\sigma_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})$. So $\sigma_v(a') \in \mathit{Pid}$ and hence $a' \neq a_1, \ldots, a_n$. So $\sigma'_v(a') = \sigma_v(a')$, so

  $$(\sigma_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})(a') = (\sigma'_v \cap \{a : \sigma'_v(a) \in \mathit{Pid}\})(a').$$

  Assume $a' \in \mathrm{dom}(\sigma'_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})$. So $\sigma_v(a') \in \mathit{Pid}$ and hence $a' \neq a_1, \ldots, a_n$. So $\sigma'_v(a') = \sigma_v(a')$, so

  $$(\sigma_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})(a') = (\sigma'_v \cap \{a : \sigma'_v(a) \in \mathit{Pid}\})(a').$$

- If the transition was made by Pop-Let-Closure, then $\sigma'_v = \sigma_v[a \mapsto \langle \ell, \mathrm{shrink}(\rho, \ell)\rangle]$ where $a = \langle \iota, U, \mathrm{res}(v, \rho, \sigma_v), t\rangle$. So σ'_v differs from σ_v at at most a. By 1 and 2, we have that a not relevant in σ_v or σ'_v. Assume $a' \in \mathrm{dom}(\sigma_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})$. So $\sigma_v(a') \in \mathit{Pid}$ and hence $a' \neq a$. So $\sigma'_v(a') = \sigma_v(a')$, so

  $$(\sigma_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})(a') = (\sigma'_v \cap \{a : \sigma'_v(a) \in \mathit{Pid}\})(a').$$

  Assume $a' \in \mathrm{dom}(\sigma'_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})$. So $\sigma_v(a') \in \mathit{Pid}$ and hence $a' \neq a$. So $\sigma'_v(a') = \sigma_v(a')$, so

  $$(\sigma_v \cap \{a : \sigma_v(a) \in \mathit{Pid}\})(a') = (\sigma'_v \cap \{a : \sigma'_v(a) \in \mathit{Pid}\})(a'). \qquad \square$$

Definition C.12 (Accessible addresses)**.** Let $\xhookrightarrow{\sigma_k} \subset \mathit{KontAddr}^2$ be the relation with

$$c \xhookrightarrow{\sigma_k} c' \iff \sigma_k(c) = \textsc{Let}\langle \_, \_, \_, c'\rangle \vee \sigma_k(c) = \textsc{Do}\langle \_, \_, c'\rangle.$$

Let $\xhookrightarrow{\sigma_k}{}^*$ be the reflexive transitive closure of $\xhookrightarrow{\sigma_v}$.

The set of continuation addresses accessible from a continuation address c or a process state $q = \langle f, \rho, c, t\rangle$ in a store σ_k is given by

$$\begin{aligned}&\mathrm{accessible}_k : \mathit{KontAddr} \times \mathit{KontStore} \to \wp(\mathit{KontAddr})\\ &\mathrm{accessible}_k(c, \sigma_k) = \{c' : c \xhookrightarrow{\sigma_k}{}^* c'\}\\ &\mathrm{accessible}_k : \mathit{ProcState} \times \mathit{KontStore} \to \wp(\mathit{KontAddr})\\ &\mathrm{accessible}_k(\langle f, \rho, c, t\rangle, \sigma_k) = \mathrm{accessible}_k(c, \sigma_k)\end{aligned}$$

Let $\overset{\sigma_v}{\hookrightarrow} \subset (\mathit{ValueAddr} \uplus \mathit{Env})^2$ (read $x \overset{\sigma_v}{\hookrightarrow} y$ as 'y directly accessible from x via σ_v') be the relation with

$$\begin{aligned} a \overset{\sigma_v}{\hookrightarrow} \rho &\iff \sigma_v(a) = \langle \ell, \rho \rangle, \\ \forall \sigma_v,\ \rho \overset{\sigma_v}{\hookrightarrow} a &\iff a \in \operatorname{Im} \rho \end{aligned}$$

so $\overset{\sigma_v}{\hookrightarrow}$ forms a directed bipartite graph on $\mathit{ValueAddr}$ and Env. Then $\overset{\sigma_v}{\hookrightarrow}{}^2 \subset \mathit{ValueAddr}^2 \uplus \mathit{Env}^2$; that is, it relates value addresses with value addresses and environments with environments. Let $\overset{\sigma_v}{\hookrightarrow}{}^{2*}$ be the reflexive transitive closure of $\overset{\sigma_v}{\hookrightarrow}{}^2$.

The set of value addresses accessible from a value address v, an environment ρ, a continuation address c, or a process state $q = \langle f, \rho, c, t \rangle$ in a store σ_v, σ_k is given by

$$\begin{aligned}
&\text{accessible}_v : \mathit{ValueAddr} \times \mathit{ValueStore} \to \wp(\mathit{ValueAddr}) \\
&\text{accessible}_v(a, \sigma_v) = \{a' : a \overset{\sigma_v}{\hookrightarrow}{}^{2*} a'\} \\
&\text{accessible}_v : \mathit{Env} \times \mathit{ValueStore} \to \wp(\mathit{ValueAddr}) \\
&\text{accessible}_v(\rho, \sigma_v) = \bigcup \{\text{accessible}_v(a, \sigma_v) : \rho \overset{\sigma_v}{\hookrightarrow} a\} \\
&\text{accessible}_v : \mathit{KontAddr} \times \mathit{ValueStore} \times \mathit{KontStore} \to \wp(\mathit{ValueAddr}) \\
&\text{accessible}_v(c, \sigma_v, \sigma_k) \\
&\qquad = (\bigcup \{\text{accessible}_v(\rho', \sigma_v) : c' \in \text{accessible}_k(c, \sigma_k),\ \textsc{Let}\langle \_, \_, \rho', \_ \rangle = \sigma_k(c')\}) \\
&\qquad\ \cup (\bigcup \{\text{accessible}_v(\rho', \sigma_v) : c' \in \text{accessible}_k(c, \sigma_k),\ \textsc{Do}\langle \_, \rho', \_ \rangle = \sigma_k(c')\}) \\
&\text{accessible}_v : \mathit{ProcState} \times \mathit{ValueStore} \times \mathit{KontStore} \to \wp(\mathit{ValueAddr}) \\
&\text{accessible}_v(\langle f, \rho, c, t \rangle, \sigma_v, \sigma_k) = \text{accessible}_v(f, \sigma_v) \cup \text{accessible}_v(\rho, \sigma_v) \cup \text{accessible}_v(c, \sigma_v, \sigma_k)
\end{aligned}$$

where $\text{accessible}_v(f, \sigma_v) = \emptyset$ when $f \notin \mathit{ValueAddr}$.

Lemma C.13. *If $S = \langle \pi, \mu, \sigma_v, \sigma_k \rangle, \in$ State such that $\text{init}(\mathcal{M}) \to^* S$, Then*

1. *for all $a \in \operatorname{dom} \sigma_V$,*
   (a) *if $a' \in \text{accessible}_v(a, \sigma_v)$ then $\text{pidaddrs}(a') \subseteq \text{pidaddrs}(a)$*
   (b) *if $a' \in \text{pidaddrs}(a)$ then $a'\, \text{accessible}_v(a, \sigma_v)$.*
2. *for all $c \in \operatorname{dom} \sigma_k$, if $c' \in \text{accessible}_{vk}(c, \sigma_k)$ then $\text{pidaddrs}(c') \subseteq \text{pidaddrs}(c)$*

and the same hold in the abstract semantics.

Proof. This can be shown by induction, but should be true by construction. □

C.2 The soundness theorems

Theorem C.2. *For some program $\mathcal{M}$, let $S, S' \in$ State such that $\text{init}(\mathcal{M}) \to^* S \to S'$. Then there exists $P \in \mathfrak{T}$ with $\alpha_\pi(S) \twoheadrightarrow^{\text{defs}(\mathcal{M})} P$ and $P \sqsupseteq \alpha_\pi(S')$.*

Proof. Let $S = \langle \pi, \mu, \sigma_v, \sigma_k \rangle$, $S' = \langle \pi', \mu', \sigma'_v, \sigma'_k \rangle$. Proceed by case analysis on the rule used to make the transition $S \to S'$.

- Suppose $S \to S'$ by the rule Name, with active components (ι, q, q'). Let $I = \{\iota\}$; clearly if $\iota' \notin I$, $\pi(\iota') = \pi'(\iota')$, and

$$\sigma_v \cap \text{pidaddrs}(\pi(\iota')) = \sigma'_v \cap \text{pidaddrs}(\pi(\iota')),$$

  . We also have $\mu = \mu'$, so by Lemma C.8 we have that to show there exists P such that $\alpha_\pi(S) \rightsquigarrow P$, $P \sqsupseteq \alpha_\pi(S')$ it is enough to show that there exists P such that $\alpha_\pi(q, \iota) \twoheadrightarrow P$, $P \sqsupseteq \alpha_\pi(q', \iota)$.

$$\begin{aligned} \alpha_\pi(q, \iota) &= \alpha_{\text{CFA}}(q)[\,\text{name}(\iota) \mid \text{args}(q)\,], \\ \alpha_\pi(q', \iota) &= \alpha_{\text{CFA}}(q')[\,\text{name}(\iota) \mid \text{args}(q')\,]. \end{aligned}$$

Since the transition was made by NAME, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule ABS-NAME with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So if

$$\mathit{args} = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <)$$

we have a process definition in the abstraction

$$\alpha_{\mathrm{CFA}}(q)[\, s \mid \mathit{args}\,] := \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\, s \mid \mathit{args}\,] + \ldots$$

So

$$\begin{aligned}
&\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \\
&\quad \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathit{args}\,][\mathrm{args}(q)/\mathit{args}] \\
&\quad \equiv \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,].
\end{aligned}$$

So if $\mathrm{args}(q) = \mathrm{args}(q')$, we are done. By Corollaries C.5 and C.11 it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$, $\mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q)) = \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q'))$.

$q = \langle \ell : U, \rho, c, t\rangle$ and $q' = \langle \rho(U), \rho, c, t\rangle$. Since $\rho(U) \in \mathrm{Im}\,\rho$ we have

$$\begin{aligned}
\mathrm{pidaddrs}(q') &= \mathrm{pidaddrs}(\rho(U)) \cup \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\
&= \mathrm{pidaddrs}(\rho(U)) \cup \left(\bigcup\{\mathrm{pidaddrs}(a) : a \in \mathrm{Im}\,\rho\}\right) \cup \mathrm{pidaddrs}(c) \\
&= \left(\bigcup\{\mathrm{pidaddrs}(a) : a \in \mathrm{Im}\,\rho\}\right) \cup \mathrm{pidaddrs}(c) \\
&= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\
&= \mathrm{pidaddrs}(q).
\end{aligned}$$

- Suppose $S \to S'$ by the rule LETREC, with active components (ι, q, q'). Let $I = \{\iota\}$; clearly if $\iota' \notin I$, $\pi(\iota') = \pi'(\iota')$ and since $\sigma_v = \sigma'_v \cap \mathrm{dom}\,\sigma_v$, $\sigma_k = \sigma'_k$ $\mathrm{pidaddrs}(\pi(\iota')) = \mathrm{pidaddrs}(\pi(\iota'))$. ($\sigma'_v = \sigma_v[a_1 \mapsto x_1, \ldots a_n \mapsto x_n]$ where $a_1, \ldots, a_n \notin \mathrm{dom}\,\sigma_v$ by construction and trivially all addresses pidaddrs from $\pi(\iota')$ are in $\mathrm{dom}\,\sigma_v$ again by construction). So by Lemma C.8 we have that to show there exists P such that $\alpha_\pi(S) \rightsquigarrow P$, $P \sqsupseteq \alpha_\pi(S')$ it is enough to show that there exists P such that $\alpha_\pi(q, \iota) \twoheadrightarrow P$, $P \sqsupseteq \alpha_\pi(q', \iota)$.

$$\begin{aligned}
\alpha_\pi(q, \iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\
\alpha_\pi(q', \iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].
\end{aligned}$$

Since the transition was made by LETREC, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule ABS-LETREC with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So if

$$\mathit{args} = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <)$$

we have a process definition in the abstraction

$$\alpha_{\mathrm{CFA}}(q)[\, s \mid \mathit{args}\,] := \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\, s \mid \mathit{args}\,] + \ldots$$

So as in the previous case,

$$\begin{aligned}
&\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \\
&\quad \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,].
\end{aligned}$$

So if $\mathrm{args}(q) = \mathrm{args}(q')$, we are done. By Corollaries C.5 and C.11 it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$.

$q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \ell'', \rho', c, t\rangle$, $\rho' = \rho[U_1 \mapsto a_1, \ldots, U_m \mapsto a_m]$, $a_i = \mathrm{newva}_{\mathrm{data}}(\iota, U_i, q, \texttt{fun}, \mathrm{shrink}(\rho, \ell_i))$ where $a_1, \ldots, a_m \notin \mathrm{dom}\,\sigma_v$, $U_1, \ldots, U_m \notin \mathrm{dom}\,\rho$ by construction. So $\mathrm{pidaddrs}(a_i) = \mathrm{pidaddrs}(\mathrm{shrink}(\rho, \ell_i))$

by Definition 3.2, and it is easy to show $\mathrm{pidaddrs}(\mathrm{shrink}(\rho, \ell_i)) \subseteq \mathrm{pidaddrs}(\rho)$.

$$\begin{aligned}
\mathrm{pidaddrs}(q') &= \mathrm{pidaddrs}(\rho') \cup \mathrm{pidaddrs}(c) \\
&= \left(\bigcup \{\mathrm{pidaddrs}(a) : a \in \mathrm{Im}\,\rho'\}\right) \cup \mathrm{pidaddrs}(c) \\
&= \left(\bigcup_i \mathrm{pidaddrs}(a_i)\right) \cup \left(\bigcup \{\mathrm{pidaddrs}(a) : a \in \mathrm{Im}\,\rho\}\right) \cup \mathrm{pidaddrs}(c) \\
&= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\
&= \mathrm{pidaddrs}(q).
\end{aligned}$$

- Suppose $S \to S'$ by the rule Push-Let, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q, \iota) \twoheadrightarrow \alpha_\pi(q', \iota)$.

  $$\begin{aligned}
  \alpha_\pi(q, \iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\
  \alpha_\pi(q', \iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].
  \end{aligned}$$

  Since the transition was made by Push-Let, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Push-Let with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So if

  $$\mathit{args} = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <)$$

  we have a process definition in the abstraction

  $$\alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] := \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}\,] + \ldots$$

  So as in the above cases,

  $$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,].$$

  and if $\mathrm{args}(q) = \mathrm{args}(q')$, we are done. By Corollaries C.5 and C.11 it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$.

  $q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \ell', \rho, c', t\rangle$, $c' = \mathrm{new}_{\mathrm{ka}}(\iota, q) = \langle \iota, \ell, t, \rho, \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c)\rangle$.

  $$\begin{aligned}
  \mathrm{pidaddrs}(q') &= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c') \\
  &= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\
  &= \mathrm{pidaddrs}(q).
  \end{aligned}$$

- Suppose $S \to S'$ by the rule Push-Do, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q, \iota) \twoheadrightarrow \alpha_\pi(q', \iota)$.

  $$\begin{aligned}
  \alpha_\pi(q, \iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\
  \alpha_\pi(q', \iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].
  \end{aligned}$$

  Since the transition was made by Push-Do, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Push-Do with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So if

  $$\mathit{args} = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <)$$

  we have a process definition in the abstraction

  $$\alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] := \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}\,] + \ldots$$

  So as in the above cases,

  $$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,].$$

  and if $\mathrm{args}(q) = \mathrm{args}(q')$, we are done. By Corollaries C.5 and C.11 it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$.

  $q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \ell', \rho, c', t\rangle$, $c' = \mathrm{new}_{\mathrm{ka}}(\iota, q) = \langle \iota, \ell, t, \rho, \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c)\rangle$.

  $$\begin{aligned}
  \mathrm{pidaddrs}(q') &= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c') \\
  &= \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\
  &= \mathrm{pidaddrs}(q).
  \end{aligned}$$

- Suppose $S \to S'$ by Bad-Apply, Bad-Call, Bad-Case, Bad-Spawn, Bad-Send or Bad-Pop-Let with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q,\iota) \twoheadrightarrow \alpha_\pi(q',\iota)$.

$$\begin{aligned}\alpha_\pi(q,\iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,],\\ \alpha_\pi(q',\iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].\end{aligned}$$

Since the transition was made by Bad-Apply, Bad-Call, Bad-Case, Bad-Spawn, Bad-Send or Bad-Pop-Let, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by Abs-Bad-Apply, Abs-Bad-Call, Abs-Bad-Case, Abs-Bad-Spawn, Abs-Bad-Send or Abs-Bad-Pop-Let with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So if

$$\mathit{args} = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <)$$

we have a process definition in the abstraction

$$\alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] := \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}\,] + \ldots$$

So as in the above cases,

$$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,].$$

and if $\mathrm{args}(q) = \mathrm{args}(q')$, we are done. By Corollaries C.5 and C.11 it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$. Since $q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \otimes, \rho, c, t\rangle$, this holds trivially by definition.

- Suppose $S \to S'$ by the rule Apply, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q,\iota) \twoheadrightarrow \alpha_\pi(q',\iota)$.

$$\begin{aligned}\alpha_\pi(q,\iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,],\\ \alpha_\pi(q',\iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].\end{aligned}$$

Since the transition was made by Apply, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Apply with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned}\alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] &:= \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}'\,] + \ldots\\ \mathit{args} &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <)\\ \mathit{args}' &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q'))\}, <)\end{aligned}$$

So

$$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}]$$

and if $\mathit{args}'[\mathrm{args}(q)/\mathit{args}] \sqsupseteq \mathrm{args}(q')$, we are done. By Lemma C.4 and Corollary C.11 it is enough to show $\mathrm{pidaddrs}(q) \supseteq \mathrm{pidaddrs}(q')$.

$q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \ell'', \rho'', c, \mathrm{tick}(t,t')\rangle$, $\rho'' = \rho'[V_1' \mapsto \rho(V_1), \ldots, V_n' \mapsto \rho(V_n)]$, $\langle \ell, \rho'\rangle = \sigma_v(\rho(U))$.

Since ρ' accessible from ρ in σ_v it follows by Lemma C.13 that $\mathrm{pidaddrs}(\rho') \subseteq \mathrm{pidaddrs}(\rho)$. Each $\rho(V_i)$ also has $\mathrm{pidaddrs}(\rho(V_i)) \subseteq \rho$. So $\mathrm{pidaddrs}(\rho'') \subseteq \mathrm{pidaddrs}(\rho)$ and

$$\begin{aligned}\mathrm{pidaddrs}(q') &= \mathrm{pidaddrs}(\rho'') \cup \mathrm{pidaddrs}(c)\\ &\subseteq \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c)\\ &= \mathrm{pidaddrs}(q).\end{aligned}$$

- Suppose $S \to S'$ by the rule Call, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q,\iota) \twoheadrightarrow \alpha_\pi(q',\iota)$.

$$\begin{aligned}\alpha_\pi(q,\iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,],\\ \alpha_\pi(q',\iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].\end{aligned}$$

Since the transition was made by CALL, we have that $\alpha_{\text{CFA}}(S) \rightsquigarrow \alpha_{\text{CFA}}(S')$ by the rule ABS-CALL with active components $(\alpha_{\text{CFA}}(\iota), \alpha_{\text{CFA}}(q), \alpha_{\text{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned}\alpha_{\text{CFA}}(q)[\, s \mid \mathit{args} \,] &:= \boldsymbol{\tau}.\alpha_{\text{CFA}}(q')[\, s \mid \mathit{args}' \,] + \ldots\\ \mathit{args} &= \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))\}, <)\\ \mathit{args}' &= \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{\text{CFA}}(q'))\}, <)\end{aligned}$$

So

$$\alpha_{\text{CFA}}(q)[\, \text{name}(\iota) \mid \text{args}(q) \,] \twoheadrightarrow \alpha_{\text{CFA}}(q')[\, \text{name}(\iota) \mid \mathit{args}' \,][\text{args}(q)/\mathit{args}]$$

and if $\mathit{args}'[\text{args}(q)/\mathit{args}] \sqsupseteq \text{args}(q')$, we are done. By Lemma C.4 and Corollary C.11 it is enough to show $\text{pidaddrs}(q) \supseteq \text{pidaddrs}(q')$.

$q = \langle \ell, \rho, c, t \rangle$, $q' = \langle \ell'', \rho'', c, \text{tick}(t, t') \rangle$, $\rho'' = \rho'[V_1' \mapsto \rho(V_1), \ldots, V_n' \mapsto \rho(V_n)]$, $\rho' = \text{modenv}(\mathcal{M}, \mathit{atom}_1)$, $\sigma_v(\rho(U_1)) = \langle \ell : \mathit{atom}_1, [] \rangle$,

As ρ' is produced by modenv, we have that $\operatorname{Im} \rho' \cap \mathit{PidAddr} = \emptyset$ and $a \in \operatorname{Im} \rho' \implies \text{pidaddrs}(a) = \{\}$. So

$$\begin{aligned}\text{pidaddrs}(\rho'') &= \bigcup \{\text{pidaddrs}(a) : a \in \operatorname{Im} \rho''\}\\ &= \left(\bigcup \{\text{pidaddrs}(a) : a \in \operatorname{Im} \rho'\}\right) \cup \bigcup_i \text{pidaddrs}(\rho(V_i))\\ &= \emptyset \cup \bigcup_i \text{pidaddrs}(\rho(V_i))\\ &\subseteq \text{pidaddrs}(\rho)\end{aligned}$$

So as in the previous case, $\text{pidaddrs}(q') = \text{pidaddrs}(q)$.

- Suppose $S \to S'$ by the rule POP-LET-CLOSURE, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q, \iota) \twoheadrightarrow \alpha_\pi(q', \iota)$.

$$\begin{aligned}\alpha_\pi(q, \iota) &= \alpha_{\text{CFA}}(q)[\, \text{name}(\iota) \mid \text{args}(q) \,],\\ \alpha_\pi(q', \iota) &= \alpha_{\text{CFA}}(q')[\, \text{name}(\iota) \mid \text{args}(q') \,].\end{aligned}$$

Since the transition was made by POP-LET-CLOSURE, we have that $\alpha_{\text{CFA}}(S) \rightsquigarrow \alpha_{\text{CFA}}(S')$ by the rule ABS-POP-LET-CLOSURE with active components $(\alpha_{\text{CFA}}(\iota), \alpha_{\text{CFA}}(q), \alpha_{\text{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned}\alpha_{\text{CFA}}(q)[\, s \mid \mathit{args} \,] &:= \boldsymbol{\tau}.\alpha_{\text{CFA}}(q')[\, s \mid \mathit{args}' \,] + \ldots\\ \mathit{args} &= \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))\}, <)\\ \mathit{args}' &= \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{\text{CFA}}(q'))\}, <)\end{aligned}$$

So

$$\alpha_{\text{CFA}}(q)[\, \text{name}(\iota) \mid \text{args}(q) \,] \twoheadrightarrow \alpha_{\text{CFA}}(q')[\, \text{name}(\iota) \mid \mathit{args}' \,][\text{args}(q)/\mathit{args}]$$

and if $\mathit{args}'[\text{args}(q)/\mathit{args}] \sqsupseteq \text{args}(q')$, we are done. By Lemma C.4 and Corollary C.11 it is enough to show $\text{pidaddrs}(q) \supseteq \text{pidaddrs}(q')$.

$q = \langle \ell, \rho, c, t \rangle$, $q' = \langle \ell', \rho'', c', t \rangle$, $\rho'' = \rho'[U \mapsto a]$, $\text{LET}\langle U \cdot \epsilon, \ell', \rho', c' \rangle = \sigma_k(c)$, $a = \text{newva}_{\text{data}}(\iota, U, t, \text{res}(\sigma_v, \langle \ell, \text{shrink}$

So by Definition 3.2 we have $\text{pidaddrs}(a) = \text{pidaddrs}(\text{shrink}(\rho, \ell))$,

$$\text{pidaddrs}(\rho'') = \text{pidaddrs}(a) \cup \text{pidaddrs}(\rho') = \text{pidaddrs}(\text{shrink}(\rho, \ell)) \cup \text{pidaddrs}(\rho').$$

By construction, $\text{pidaddrs}(c) = \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')$. So

$$\begin{aligned}\text{pidaddrs}(q) &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(c)\\ &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')\\ &\supseteq \text{pidaddrs}(\text{shrink}(\rho, \ell)) \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')\\ &= \text{pidaddrs}(\rho'') \cup \text{pidaddrs}(c')\\ &= \text{pidaddrs}(q').\end{aligned}$$

- Suppose $S \to S'$ by the rule Pop-Let-ValueAddr, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q,\iota) \twoheadrightarrow \alpha_\pi(q',\iota)$.

$$\begin{aligned} \alpha_\pi(q,\iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\ \alpha_\pi(q',\iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,]. \end{aligned}$$

  Since the transition was made by Pop-Let-ValueAddr, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Pop-Let-ValueAddr with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned} \alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] &:= \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}'\,] + \dots \\ \mathit{args} &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <) \\ \mathit{args}' &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q'))\}, <) \end{aligned}$$

  So

$$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}]$$

  and if $\mathit{args}'[\mathrm{args}(q)/\mathit{args}] \sqsupseteq \mathrm{args}(q')$, we are done. By Lemma C.4 and Corollary C.11 it is enough to show $\mathrm{pidaddrs}(q) \supseteq \mathrm{pidaddrs}(q')$.

  $q = \langle a, \rho, c, t\rangle$, $q' = \langle \ell', \rho'', c', t\rangle$, $\rho'' = \rho'[U \mapsto a]$, $\textsc{Let}\langle U\cdot\epsilon, \ell', \rho', c'\rangle = \sigma_k(c)$, So $\mathrm{pidaddrs}(\rho'') = \mathrm{pidaddrs}(a) \cup \mathrm{pidaddrs}(\rho')$. By construction, $\mathrm{pidaddrs}(c) = \mathrm{pidaddrs}(\rho') \cup \mathrm{pidaddrs}(c')$. So

$$\begin{aligned} \mathrm{pidaddrs}(q) &= \mathrm{pidaddrs}(a) \cup \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\ &= \mathrm{pidaddrs}(a) \cup \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(\rho') \cup \mathrm{pidaddrs}(c') \\ &\supseteq \mathrm{pidaddrs}(a) \cup \mathrm{pidaddrs}(\rho') \cup \mathrm{pidaddrs}(c') \\ &= \mathrm{pidaddrs}(\rho'') \cup \mathrm{pidaddrs}(c') \\ &= \mathrm{pidaddrs}(q'). \end{aligned}$$

- Suppose $S \to S'$ by the rule Pop-Let-ValueList, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q,\iota) \twoheadrightarrow \alpha_\pi(q',\iota)$.

$$\begin{aligned} \alpha_\pi(q,\iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\ \alpha_\pi(q',\iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,]. \end{aligned}$$

  Since the transition was made by Pop-Let-ValueList, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Pop-Let-ValueList with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned} \alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] &:= \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}'\,] + \dots \\ \mathit{args} &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <) \\ \mathit{args}' &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q'))\}, <) \end{aligned}$$

  So

$$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}]$$

  and if $\mathit{args}'[\mathrm{args}(q)/\mathit{args}] \sqsupseteq \mathrm{args}(q')$, we are done. By Lemma C.4 and Corollary C.11 it is enough to show $\mathrm{pidaddrs}(q) \supseteq \mathrm{pidaddrs}(q')$.

  $q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \ell', \rho'', c', t\rangle$, $\rho'' = \rho'[U'_1 \mapsto \rho(U_1), \dots, U'_n \mapsto \rho(U_n)]$, $\textsc{Let}\langle U'_1 \cdot \dots \cdot U'_n, \ell', \rho', c'\rangle = \sigma_k(c)$, So

$$\mathrm{pidaddrs}(\rho'') = \left(\bigcup_i \mathrm{pidaddrs}(\rho(U_i))\right) \cup \mathrm{pidaddrs}(\rho') \subseteq \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(\rho').$$

By construction, $\text{pidaddrs}(c) = \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')$. So

$$\begin{aligned}
\text{pidaddrs}(q) &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(c) \\
&= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c') \\
&\supseteq \text{pidaddrs}(\rho'') \cup \text{pidaddrs}(c') \\
&= \text{pidaddrs}(q').
\end{aligned}$$

- Suppose $S \to S'$ by the rule Pop-Do, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q,\iota) \twoheadrightarrow \alpha_\pi(q',\iota)$.

$$\begin{aligned}
\alpha_\pi(q,\iota) &= \alpha_{\text{CFA}}(q)[\,\text{name}(\iota) \mid \text{args}(q)\,], \\
\alpha_\pi(q',\iota) &= \alpha_{\text{CFA}}(q')[\,\text{name}(\iota) \mid \text{args}(q')\,].
\end{aligned}$$

Since the transition was made by Pop-Do, we have that $\alpha_{\text{CFA}}(S) \rightsquigarrow \alpha_{\text{CFA}}(S')$ by the rule Abs-Pop-Do with active components $(\alpha_{\text{CFA}}(\iota), \alpha_{\text{CFA}}(q), \alpha_{\text{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned}
\alpha_{\text{CFA}}(q)[\,s \mid \mathit{args}\,] &:= \boldsymbol{\tau}.\alpha_{\text{CFA}}(q')[\,s \mid \mathit{args}'\,] + \dots \\
\mathit{args} &= \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))\}, <) \\
\mathit{args}' &= \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{\text{CFA}}(q'))\}, <)
\end{aligned}$$

So

$$\alpha_{\text{CFA}}(q)[\,\text{name}(\iota) \mid \text{args}(q)\,] \twoheadrightarrow \alpha_{\text{CFA}}(q')[\,\text{name}(\iota) \mid \mathit{args}'\,][\text{args}(q)/\mathit{args}]$$

and if $\mathit{args}'[\text{args}(q)/\mathit{args}] \sqsupseteq \text{args}(q')$, we are done. By Lemma C.4 and Corollary C.11 it is enough to show $\text{pidaddrs}(q) \supseteq \text{pidaddrs}(q')$.

$q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \ell', \rho', c', t\rangle$, $\text{Do}\langle \ell', \rho', c'\rangle = \sigma_k(c)$, By construction, $\text{pidaddrs}(c) = \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')$. So

$$\begin{aligned}
\text{pidaddrs}(q) &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(c) \\
&= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c') \\
&\supseteq \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c') \\
&= \text{pidaddrs}(q').
\end{aligned}$$

- Suppose $S \to S'$ by the rule Pop-Let-Pid, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q,\iota) \twoheadrightarrow \alpha_\pi(q',\iota)$. $q = \langle \iota', \rho, c, t\rangle$, so

$$\begin{aligned}
\alpha_\pi(q,\iota) &= \alpha_{\text{CFA}}(q)[\,\text{name}(\iota), \text{name}(\iota') \mid \text{args}(q)\,], \\
\alpha_\pi(q',\iota) &= \alpha_{\text{CFA}}(q')[\,\text{name}(\iota) \mid \text{args}(q')\,].
\end{aligned}$$

Since the transition was made by Pop-Let-Pid, we have that $\alpha_{\text{CFA}}(S) \rightsquigarrow \alpha_{\text{CFA}}(S')$ by the rule Abs-Pop-Let-Pid with active components $(\alpha_{\text{CFA}}(\iota), \alpha_{\text{CFA}}(q), \alpha_{\text{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned}
\alpha_{\text{CFA}}(q)[\,s, s' \mid \mathit{args}\,] &:= \boldsymbol{\tau}.\alpha_{\text{CFA}}(q')[\,s \mid \mathit{args}'\,] + \dots \\
\mathit{args} &= \text{sort}(\{\text{setname}(\widehat{a}) : \widehat{a} \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))\}, <) \\
B_1 \cdot \ldots \cdot B_m &= \text{sort}(\{\text{setname}(\widehat{a}') : \widehat{a}' \in \text{pidaddrs}(\alpha_{\text{CFA}}(q'))\}, <), \\
\mathit{args}' = T_1 \cdot \ldots \cdot T_m, T_i &= \begin{cases} B_i \cup \{s'\}, & B_i \in \mathit{args}, B_i = \text{setname}(\widehat{a}'), \in \\ B_i, & B_i \in \mathit{args}, B_i \neq \text{setname}(\widehat{a}') \\ \{s'\}, & B_i \notin \mathit{args}, B_i \neq \text{setname}(\widehat{a}') \\ \emptyset & B_i \notin \mathit{args}, B_i \neq \text{setname}(\widehat{a}') \end{cases}
\end{aligned}$$

So

$$\alpha_{\text{CFA}}(q)[\,\text{name}(\iota), \text{name}(\iota') \mid \text{args}(q)\,] \twoheadrightarrow \alpha_{\text{CFA}}(q')[\,\text{name}(\iota) \mid \mathit{args}'\,][\text{args}(q)/\mathit{args}]$$

and if $\mathit{args}'[\text{args}(q)/\mathit{args}] \sqsupseteq \text{args}(q')$, we are done.

$q = \langle \iota', \rho, c, t\rangle$, $q' = \langle \ell', \rho'', c', t\rangle$, $\rho'' = \rho'[U \mapsto a]$, $\textsc{Let}\langle U \cdot \epsilon, \ell', \rho', c'\rangle = \sigma_k(c)$, $a = \text{newva}_{\text{pid}}(\iota, U, q) = \langle \iota, U, t, \iota'\rangle$. So

$$\text{pidaddrs}(\rho'') = \text{pidaddrs}(a) \cup \text{pidaddrs}(\rho') = \{a\} \cup \text{pidaddrs}(\rho').$$

By construction, $\text{pidaddrs}(c) = \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')$. So $\text{pidaddrs}(q) = \text{pidaddrs}(\rho) \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')$, $\text{pidaddrs}(q') = \{a\} \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')$. Since a is fresh by construction we have $a \notin \text{Im}\,\rho$, $\text{pidaddrs}(q) \cap \text{pidaddrs}(q') = \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c')$.

Now

$$\begin{gathered}
\mathit{args}'[\text{args}(q)/\mathit{args}] = T''_1 \cdot \ldots \cdot T''_m[\text{args}(q)/\mathit{args}] \\
\text{args}(q) = T_1 \cdot \ldots \cdot T_m \\
\widehat{a}_1 \cdot \ldots \cdot \widehat{a}_m = \text{sort}(\text{pidaddrs}(\alpha_{\text{CFA}}(q)), \widehat{<_{\text{VA}}}) \\
T_i = \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \widehat{a}_i\}, \text{args}(q') = T'_1 \cdot \ldots \cdot T'_p \\
\widehat{a}'_1 \cdot \ldots \cdot \widehat{a}'_p = \text{sort}(\text{pidaddrs}(\alpha_{\text{CFA}}(q')), \widehat{<_{\text{VA}}}) \\
T'_i = \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q'), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}
\end{gathered}$$

Clearly since name is an order isomorphism we have m = p, so if $T''_i[\text{args}(q)/\mathit{args}] \subseteq T'_i$ for each i then $\mathit{args}'[\text{args}(q)/\mathit{args}] \sqsupseteq \text{args}(q')$ and we are done.

$\alpha_{\text{CFA}}(q) = \langle \alpha_{\text{CFA}}(\iota'), \alpha_{\text{CFA}}(\rho), \alpha_{\text{CFA}}(c), \alpha_{\text{CFA}}(t)\rangle$, $\alpha_{\text{CFA}}(q') = \langle \ell', \alpha_{\text{CFA}}(\rho'), \alpha_{\text{CFA}}(c'), \alpha_{\text{CFA}}(t)\rangle$. Let $\widehat{a} = \widehat{\text{newva}}_{\text{pid}}(\alpha_{\text{CFA}}(\iota), U, \alpha_{\text{CFA}}(q)) = \langle \alpha_{\text{CFA}}(\iota), U, \alpha_{\text{CFA}}(t), \alpha_{\text{CFA}}(\iota')\rangle$ be the value address constructed by Abs-Pop-Let-Pid. Clearly $\alpha_{\text{CFA}}(a) = \widehat{a}$. Continue by cases on the construction of T_i:

- Suppose $B_i \in \mathit{args}$ and $B_i = \text{setname}(\alpha_{\text{CFA}}(a))$. Then $T''_i = B_i \cup \{s'\}$. Since $B_i \in \mathit{args}$ there is some $\widehat{a}_j \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))$ with $\text{setname}(a_j) = B_i$, so

  $$T''_i[\text{args}(q)/\mathit{args}] = T_j \cup \{s'\} = \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \widehat{a}_j\} \cup \{s'\}$$

  But $\{s'\} = \{\text{name}(\iota')\} = \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(a), \alpha_{\text{CFA}}(a') = \alpha_{\text{CFA}}(a)\}$ Since setname an isomorphism, we have $\widehat{a}'_i = \alpha_{\text{CFA}}(a) = \widehat{a}_j$ so

  $$T''_i[\text{args}(q)/\mathit{args}] = \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q) \cup \text{pidaddrs}(a), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}$$

  $$\begin{aligned}
  \text{pidaddrs}(q) \cup \text{pidaddrs}(a) &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c') \cup \{a\} \\
  &\supseteq \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c') \cup \{a\} \\
  &= \text{pidaddrs}(q'), \mathit{so}
  \end{aligned}$$

  $$\begin{aligned}
  T''_i[\text{args}(q)/\mathit{args}] &\supseteq \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q'), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\} \\
  &= T'_i
  \end{aligned}$$

- Suppose $B_i \in \mathit{args}$ and $B_i \neq \text{setname}(\alpha_{\text{CFA}}(a))$. Then $T''_i = B_i$. Since $B_i \in \mathit{args}$ there is some $\widehat{a}_j \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))$ with $\text{setname}(a_j) = B_i$, so

  $$T''_i[\text{args}(q)/\mathit{args}] = T_j = \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \widehat{a}_j\}$$

  Since setname an isomorphism, we have $\widehat{a}'_i = \widehat{a}_j$ and $\widehat{a}'_i \neq \alpha_{\text{CFA}}(a)$. So

  $$\begin{aligned}
  T''_i[\text{args}(q)/\mathit{args}] &= \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\} \\
  T'_i &= \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q'), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\} \\
  &= \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota''\rangle \in \text{pidaddrs}(q') \setminus \{a\}, \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}
  \end{aligned}$$

  $$\begin{aligned}
  \text{pidaddrs}(q) &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c') \\
  &\supseteq \text{pidaddrs}(\rho') \cup \text{pidaddrs}(c') \\
  &= \text{pidaddrs}(q') \setminus \{a\}, \mathit{so}
  \end{aligned}$$

  $$T''_i[\text{args}(q)/\mathit{args}] \supseteq T'_i$$

- Suppose $B_i \notin \mathit{args}$ and $B_i = \mathrm{setname}(\alpha_{\mathrm{CFA}}(a))$. Then $T_i'' = \{s'\}$ and $T_i''[\mathrm{args}(q)/\mathit{args}] = \{s'\}$ Since setname an isomorphism, we have $\widehat{a}_i' \neq \alpha_{\mathrm{CFA}}(a)$. Now $\{s'\} = \{\mathrm{name}(\iota')\} = \{\mathrm{name}(\iota'') : a' = \langle \_, \_, \_, \iota'' \rangle \in \{a\}, \alpha_{\mathrm{CFA}}(a') = \widehat{a}_i'\}$. Since

$$\mathrm{pidaddrs}(q') = \mathrm{pidaddrs}(\rho') \cup \mathrm{pidaddrs}(c') \cup \{a\},$$
$$\mathrm{pidaddrs}(q) \cap \mathrm{pidaddrs}(q') = \mathrm{pidaddrs}(\rho') \cup \mathrm{pidaddrs}(c'),$$

clearly if $a' \in \mathrm{pidaddrs}(q')$ and $a' \neq a$ then $a' \in \mathrm{pidaddrs}(q)$. Then by construction $\alpha_{\mathrm{CFA}}(a') \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))$ But then $\alpha_{\mathrm{CFA}}(a') = \widehat{a}_j$ for some j and $\mathrm{setname}(\widehat{a}_j) \in \mathit{args}$. So since $B_i = \mathrm{setname}(\widehat{a}_i')$, $B_i \notin \mathit{args}$, by the contrapositive we must have $\widehat{a}_i' \neq \alpha_{\mathrm{CFA}}(a')$ for all $a' \in \mathrm{pidaddrs}(q'), a' \neq a$. So

$$\begin{aligned} T_i' &= \{\mathrm{name}(\iota'') : a' = \langle \_, \_, \_, \iota'' \rangle \in \mathrm{pidaddrs}(q'), \alpha_{\mathrm{CFA}}(a') = \widehat{a}_j\} \\ &= \{\mathrm{name}(\iota'') : a' = \langle \_, \_, \_, \iota'' \rangle \in \{a\}, \alpha_{\mathrm{CFA}}(a') = \widehat{a}_j\} \\ &= \{s'\} = T_i''[\mathrm{args}(q)/\mathit{args}] \end{aligned}$$

- Suppose $B_i \notin \mathit{args}$ and $B_i \neq \mathrm{setname}(\alpha_{\mathrm{CFA}}(a))$. This case is impossible, since $B_i = \mathrm{setname}(\widehat{a}_i')$, $\widehat{a}_i' \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q'))$, so $\widehat{a}_i' = \alpha_{\mathrm{CFA}}(a')$ for some $a' \in \mathrm{pidaddrs}(q')$ by construction. But by arguments in the above cases, this a' is in neither $\mathrm{pidaddrs}(q) \cap \mathrm{pidaddrs}(q')$ nor $\{a\}$, so $a' \notin \mathrm{pidaddrs}(q')$, a contradiction.

- Suppose $S \to S'$ by the rule Case, with active components (ι, q, q'), where $\langle i, \theta \rangle$ is the selected match and guard_i is the guard of the matched pattern. Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q, \iota) \twoheadrightarrow \alpha_\pi(q', \iota)$.

$$\alpha_\pi(q, \iota) = \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,],$$
$$\alpha_\pi(q', \iota) = \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].$$

Since the transition was made by Case, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Case with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$, where $\langle i, \alpha_{\mathrm{CFA}}(\theta) \rangle$ is the selected match. So if

$$\mathit{args} = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <)$$
$$G = \{(\widehat{a}', \widehat{a}'') : \widehat{a}' = (\alpha_{\mathrm{CFA}}(\rho) \uplus \alpha_{\mathrm{CFA}}(\theta))(U),\ \widehat{a}'' = (\alpha_{\mathrm{CFA}}(\rho) \uplus \alpha_{\mathrm{CFA}}(\theta))(V),$$
$$\widehat{a}', \widehat{a}'' \in \widehat{\mathit{PidAddr}},\ U \mathrel{=:=} V \text{ appears in } \mathit{guard}_i\}$$
$$\mathit{conds} = \mathrm{sort}(\{A_1 \cap A_2 \neq \emptyset : (\widehat{a}', \widehat{a}'') \in G, A_1 = \mathrm{setname}(\widehat{a}'), A_2 = \mathrm{setname}(\widehat{a}'')\}, <_\cap)$$

we have a process definition in the abstraction

$$\alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] := [\mathit{conds}].\alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}\,] + \ldots$$

So if $\mathit{cond}[\mathrm{args}(q)/\mathit{args}]$ holds for every $\mathit{cond} \in \mathit{conds}$,

$$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,].$$

So if $\mathit{cond}[\mathrm{args}(q)/\mathit{args}]$ holds for every $\mathit{cond} \in \mathit{conds}$ and $\mathrm{args}(q) = \mathrm{args}(q')$, we are done.

By Corollaries C.5 and C.11 for $\mathrm{args}(q) = \mathrm{args}(q')$, it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$.

$q = \langle \ell, \rho, c, t \rangle$, $q' = \langle \ell_i, \rho'', c, t \rangle$, $\rho'' = \rho \uplus \theta$, $\langle i, \theta \rangle = \mathrm{cmatch}(\mathit{clauses}, \rho(U), \sigma_v)$. From Definition A.1, it is clear that $\mathrm{Im}\,\theta \subseteq \mathrm{accessible}_v(\rho, \sigma_v)$, and by construction if $a' \in \mathrm{accessible}_v(a, \sigma_V)$ then $\mathrm{pidaddrs}(a') \subseteq \mathrm{pidaddrs}(a)$. so $\mathrm{pidaddrs}(\rho) = \mathrm{pidaddrs}(\rho'')$. Now trivially $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$.

Since $S \to S'$ we have that $\rho \uplus \theta$ satisfies every guard in guard_i; that is, for each $U \mathrel{=:=} V$ in guard_i we have $\mathrm{unifiable}(\rho(U), \rho(V), \sigma_v)$. Each $U \mathrel{=:=} V$ in guard_i with $(\rho \uplus \theta)(U), (\rho \uplus \theta)(V) \in \mathit{PidAddr} \implies (\alpha_{\mathrm{CFA}}(\rho) \uplus \alpha_{\mathrm{CFA}}(\theta))(U), (\alpha_{\mathrm{CFA}}(\rho) \uplus \alpha_{\mathrm{CFA}}(\theta))(V) \in \widehat{\mathit{PidAddr}}$ corresponds to an element of G and hence to some $\mathit{cond} = A_1 \cap A_2 \neq \emptyset$ in conds where

$A_1 = \mathrm{setname}((\alpha_{\mathrm{CFA}}(\rho) \uplus \alpha_{\mathrm{CFA}}(\theta))(U)), A_2 = \mathrm{setname}((\alpha_{\mathrm{CFA}}(\rho) \uplus \alpha_{\mathrm{CFA}}(\theta))(V))$. Since $\mathrm{dom}\,\theta \subseteq \mathrm{accessible}_v(\rho, \sigma_v)$ we have that $A_1, A_2 \in \mathit{args}$ and so

$$\begin{aligned}
\mathit{cond}[\mathrm{args}(q)/\mathit{args}] = T \cap T' \neq \emptyset \\
T = \{\mathrm{name}(\iota') : a' = \langle \_, \_, \_, \iota' \rangle \in \mathrm{pidaddrs}(q), \alpha_{\mathrm{CFA}}(a') = (\alpha_{\mathrm{CFA}}(\rho) \uplus \alpha_{\mathrm{CFA}}(\theta))(U))\} \\
\supseteq \{\iota'\}, \quad (\rho \uplus \theta)(U) = \langle \_, \_, \_, \iota' \rangle \\
T' \supseteq \{\iota''\}, \quad (\rho \uplus \theta)(V) = \langle \_, \_, \_, \iota'' \rangle
\end{aligned}$$

Since $\rho \uplus \theta$ satisfies $U \mathrel{=:=} V$, we have that $\sigma_v((\rho \uplus \theta)(U)) = \sigma_v((\rho \uplus \theta)(V))$, so $\iota' = \iota''$ and $T \cap T' \neq \emptyset$.

- Suppose $S \to S'$ by the rule Receive, with active components (ι, q, q'). Let $I = \{\iota\}$; clearly if $\iota' \notin I$, $\pi(\iota') = \pi'(\iota')$, $\mu(\iota') = \mu'(\iota')$,

$$\sigma_v \cap \mathrm{pidaddrs}(\pi(\iota')) = \sigma'_v \cap \mathrm{pidaddrs}(\pi(\iota')),$$

and for $a \in \mu(\iota')$,

$$\sigma_v \cap \mathrm{pidaddrs}(a) = \sigma'_v \cap \mathrm{pidaddrs}(a),$$

Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show

$$\alpha_\pi(q, \iota) \parallel \alpha_\pi(\mu \cap \{\iota\}) \twoheadrightarrow \alpha_\pi(q', \iota) \parallel \alpha_\pi(\mu' \cap \{\iota\}).$$

$$\begin{aligned}
\alpha_\pi(q, \iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\
\alpha_\pi(q', \iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].
\end{aligned}$$

Let $\langle i, a, \rho', m \rangle = \mathrm{mmatch}(\mathit{clauses}, \mu(\iota), \sigma_v)$ be the pattern index, address, substitution and new mailbox produced by mmatch during the transition and guard_i be the guard of the matched pattern. Then $\mu'(\iota) = m$, and by definition of mmatch this m is $\mu(\iota)$ with the first occurrence of a omitted. So

$$\alpha_\pi(\mu \cap \{\iota\}) = \textsc{Send}_{\alpha_{\mathrm{CFA}}(a)}[\,\mathrm{name}(\iota) \mid \mathrm{args}(a)\,] \parallel \alpha_\pi(\mu' \cap \{\iota\})$$

Since the transition was made by Receive, we have that $\alpha_{\mathrm{CFA}}(S) \leadsto \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Receive with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$, where

$$\langle i, \alpha_{\mathrm{CFA}}(a), \alpha_{\mathrm{CFA}}(\rho'), \widehat{m} \rangle = \widehat{\mathrm{mmatch}}(\mathit{clauses}, \alpha_{\mathrm{CFA}}(\mu)(\alpha_{\mathrm{CFA}}(\iota)), \alpha_{\mathrm{CFA}}(\sigma_v))$$

is the pattern index, address, substitution and new mailbox produced by $\widehat{\mathrm{mmatch}}$ during the transition. So we have process definitions in the abstraction

$$\begin{gathered}
\alpha_{\text{CFA}}(q)[\,s \mid \mathit{args}\,] := s\mathbf{?}\alpha_{\text{CFA}}(a)(\mathit{recargs})[\mathit{conds}].\alpha_{\text{CFA}}(q')[\,s \mid \mathit{args}'\,] + \dots \\
\textsc{Send}_{\alpha_{\text{CFA}}(a)}[\,s \mid \mathit{args}''\,] := s\,\mathbf{!}\,\alpha_{\text{CFA}}(a)\langle \mathit{args}''\rangle.\mathbf{0}, \\
\mathit{args} = \operatorname{sort}(\{\operatorname{setname}(\widehat{a}) : \widehat{a} \in \operatorname{pidaddrs}(\alpha_{\text{CFA}}(q))\}, <) \\
\mathit{recargs} = \operatorname{sort}(\{\operatorname{setname}'(\widehat{a}') : \widehat{a}' \in \operatorname{pidaddrs}(\alpha_{\text{CFA}}(a))\}, <) \\
\widehat{a}_1 \cdot \ldots \cdot \widehat{a}_m = \operatorname{sort}(\operatorname{pidaddrs}(\alpha_{\text{CFA}}(q')), <) \\
B_i = \operatorname{setname}(\widehat{a}_i) \\
B_i' = \operatorname{setname}'(\widehat{a}_i) \\
\mathit{args}'' = S_1 \cdot \ldots \cdot S_m,\ S_i = \begin{cases} B_i \cup B_i', & B_i \in \mathit{args}, B_i' \in \mathit{recargs} \\ B_i, & B_i \in \mathit{args}, B_i' \notin \mathit{recargs} \\ B_i', & B_i \notin \mathit{args}, B_i' \in \mathit{recargs} \\ \emptyset, & B_i \notin \mathit{args}, B_i' \notin \mathit{recargs} \end{cases} \\
\mathit{args}'' = \operatorname{sort}(\{\operatorname{setname}(\widehat{a}') : \widehat{a}' \in \operatorname{pidaddrs}(\alpha_{\text{CFA}}(a))\}, <) \\
G = \{(\widehat{a}', \widehat{a}'') : \widehat{a}' = (\alpha_{\text{CFA}}(\rho) \uplus \alpha_{\text{CFA}}(\theta))(U),\ \widehat{a}'' = (\alpha_{\text{CFA}}(\rho) \uplus \alpha_{\text{CFA}}(\theta))(V), \\
\widehat{a}', \widehat{a}'' \in \widehat{\mathit{PidAddr}},\ U \texttt{ =:= } V \text{ appears in } \mathit{guard}_i\} \\
\operatorname{setname}_{\mathit{clause}_i}(\widehat{a}) = \begin{cases} \operatorname{setname}'(\widehat{a}) & \operatorname{setname}(\widehat{a}) \text{ appears in a pattern in } \mathit{clause}_i \\ \operatorname{setname}(\widehat{a}) & \text{otherwise} \end{cases} \\
\mathit{conds} = \operatorname{sort}(\{A_1 \cap A_2 \neq \emptyset : (\widehat{a}', \widehat{a}'') \in G, A_1 = \operatorname{setname}_{\mathit{clause}_i}(\widehat{a}'), A_2 = \operatorname{setname}_{\mathit{clause}_i}(\widehat{a}'')\}, <_\cap)
\end{gathered}$$

So if $\mathit{cond}[\operatorname{args}(q)/\mathit{args}][\operatorname{args}(a)/\mathit{recargs}]$ holds for every $\mathit{cond} \in \mathit{conds}$,

$$\begin{aligned}
\alpha_\pi(q,\iota) \parallel \alpha_\pi(\mu \cap \{\iota\}) &\equiv \alpha_{\text{CFA}}(q)[\,\operatorname{name}(\iota) \mid \operatorname{args}(q)\,] \\
&\quad \parallel \textsc{Send}_{\alpha_{\text{CFA}}(a)}[\,\operatorname{name}(s) \mid \operatorname{args}(a)\,] \\
&\quad \parallel \alpha_\pi(\mu' \cap \{\iota\}) \\
&\equiv (\operatorname{name}(\iota)\mathbf{?}\alpha_{\text{CFA}}(a)(\mathit{recargs})[\mathit{conds}].\alpha_{\text{CFA}}(q')[\,s \mid \mathit{args}'\,] + \dots)[\operatorname{args}(q)/\mathit{args}] \\
&\quad \parallel \operatorname{name}(\iota)\,\mathbf{!}\,\alpha_{\text{CFA}}(a)\langle \operatorname{args}(a)\rangle.\mathbf{0} \\
&\quad \parallel \alpha_\pi(\mu' \cap \{\iota\}) \\
&\twoheadrightarrow \alpha_{\text{CFA}}(q')[\,s \mid \mathit{args}'\,][\operatorname{args}(q)/\mathit{args}][\operatorname{args}(a)/\mathit{recargs}] \parallel \alpha_\pi(\mu' \cap \{\iota\})
\end{aligned}$$

So if $\mathit{cond}[\operatorname{args}(q)/\mathit{args}][\operatorname{args}(a)/\mathit{recargs}]$ holds for every $\mathit{cond} \in \mathit{conds}$ and $\mathit{args}'[\operatorname{args}(q)/\mathit{args}][\operatorname{args}(a)/\mathit{recargs}] \sqsupseteq \operatorname{args}(q')$, we are done.

$q = \langle \ell, \rho, c, t\rangle$, $\langle i, a, \theta, m\rangle = \operatorname{mmatch}(\mathit{clauses}, \mu(\iota), \sigma_v)$. Since $S \to S'$ we have that $\rho \uplus \theta$ satisfies every guard in guard_i; that is, for each $U \texttt{ =:= } V$ in guard_i we have $\operatorname{unifiable}(\rho(U), \rho(V), \sigma_v)$. Each $U \texttt{ =:= } V$ in guard_i with $(\rho \uplus \theta)(U), (\rho \uplus \theta)(V) \in \mathit{PidAddr} \implies (\alpha_{\text{CFA}}(\rho) \uplus \alpha_{\text{CFA}}(\theta))(U), (\alpha_{\text{CFA}}(\rho) \uplus \alpha_{\text{CFA}}(\theta))(V) \in \widehat{\mathit{PidAddr}}$ corresponds to an element of G and hence to some $\mathit{cond} = A_1 \cap A_2 \neq \emptyset$ in conds where $A_1 = \operatorname{setname}_{\mathit{clause}_i}((\alpha_{\text{CFA}}(\rho) \uplus \alpha_{\text{CFA}}(\theta))(U))$, $A_2 = \operatorname{setname}_{\mathit{clause}_i}((\alpha_{\text{CFA}}(\rho) \uplus \alpha_{\text{CFA}}(\theta))(V))$. Since $\operatorname{dom}\theta \subseteq \operatorname{accessible}_v(a, \sigma_v)$ we for A_i that $A_i \in \mathit{args}$ or $A_i \in \mathit{recargs}$. Let $I_i = \operatorname{pidaddrs}(q)$ if $A_i \in \mathit{args}$ and $I_i = \operatorname{pidaddrs}(a)$ if $A_i \in \mathit{recargs}$.

$$\begin{gathered}
\mathit{cond}[\operatorname{args}(q)/\mathit{args}][\operatorname{args}(a)/\mathit{recargs}] = T \cap T' \neq \emptyset \\
T = \{\operatorname{name}(\iota') : a' = \langle \_, \_, \_, \iota'\rangle \in I, \alpha_{\text{CFA}}(a') = (\alpha_{\text{CFA}}(\rho) \uplus \alpha_{\text{CFA}}(\theta))(U))\} \\
\supseteq \{\iota'\}, \quad (\rho \uplus \theta)(U) = \langle \_, \_, \_, \iota'\rangle \\
T' \supseteq \{\iota''\}, \quad (\rho \uplus \theta)(V) = \langle \_, \_, \_, \iota''\rangle
\end{gathered}$$

Since $\rho \uplus \theta$ satisfies $U \texttt{ =:= } V$, we have that $\sigma_v((\rho \uplus \theta)(U)) = \sigma_v((\rho \uplus \theta)(V))$, so $\iota' = \iota''$ and $T \cap T' \neq \emptyset$.

$q = \langle \ell, \rho, c, t\rangle$, $q' = \langle \ell_i, \rho \uplus \rho', c', t\rangle$, $\langle i, a, \rho', m\rangle = \operatorname{mmatch}(\mathit{clauses}, \mu(\iota), \sigma_v)$ Everything in ρ'

is accessible from a, so by construction $\text{pidaddrs}(\rho') \subseteq \text{pidaddrs}(a)$. So

$$\begin{aligned}
\text{pidaddrs}(q) &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(c'),\\
\text{pidaddrs}(q') &= \text{pidaddrs}(\rho) \cup \text{pidaddrs}(c') \cup \text{pidaddrs}(\rho')\\
&= \text{pidaddrs}(q) \cup \text{pidaddrs}(\rho')\\
&\subseteq \text{pidaddrs}(q) \cup \text{pidaddrs}(a)
\end{aligned}$$

Now

$$\begin{aligned}
\text{args}(q) &= T_1 \cdot \ldots \cdot T_n\\
\widehat{a}_1 \cdot \ldots \cdot \widehat{a}_n &= \text{sort}(\text{pidaddrs}(\alpha_{\text{CFA}}(q)), \widehat{<_{\text{VA}}})\\
T_i = \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \widehat{a}_i\}, \text{args}(q') &= T'_1 \cdot \ldots \cdot T'_m\\
\widehat{a}'_1 \cdot \ldots \cdot \widehat{a}'_m &= \text{sort}(\text{pidaddrs}(\alpha_{\text{CFA}}(q')), \widehat{<_{\text{VA}}})\\
T'_i = \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle \in \text{pidaddrs}(q'), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}\ \text{args}(a) &= T''_1 \cdot \ldots \cdot T''_l\\
\widehat{a}''_1 \cdot \ldots \cdot \widehat{a}''_l &= \text{sort}(\text{pidaddrs}(\alpha_{\text{CFA}}(a)), \widehat{<_{\text{VA}}})\\
T''_i &= \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle \in \text{pidaddrs}(a), \alpha_{\text{CFA}}(a') = \widehat{a}''_i\}.
\end{aligned}$$

so if $S_i[\text{args}(q)/\mathit{args}][\text{args}(a)/\mathit{recargs}] \supseteq T'_i$ for each i then $\mathit{args}'[\text{args}(q)/\mathit{args}][\text{args}(a)/\mathit{recargs}] \sqsupseteq \text{args}(q')$ and we are done.

Continue by cases on the construction of S_i:

- Suppose $B_i \in \mathit{args}$ and $B'_i \in \mathit{recargs}$. Then $S_i = B_i \cup B'_i$. Since $B_i \in \mathit{args}$ there is some $\widehat{a}_j \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))$ with $\text{setname}(\widehat{a}_j) = B_i$. Since $B'_i \in \mathit{recargs}$ there is some $\widehat{a}''_k \in \text{pidaddrs}(\alpha_{\text{CFA}}(a))$ with $\text{setname}'(\widehat{a}''_k) = B'_i$. setname and setname$'$ are isomorphisms, $\widehat{a}'_i = \widehat{a}_j = \widehat{a}''_k$. So

$$\begin{aligned}
S_i&[\text{args}(q)/\mathit{args}][\text{args}(a)/\mathit{recargs}] = T_j \cup T''_k\\
&= \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle = \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}\\
&\quad \cup \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle = \text{pidaddrs}(a), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}\\
&= \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle = \text{pidaddrs}(q) \cup \text{pidaddrs}(a), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}\\
&\supseteq T'_i = \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle = \text{pidaddrs}(q'), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}
\end{aligned}$$

- Suppose $B_i \in \mathit{args}$ and $B'_i \notin \mathit{recargs}$. Then $S_i = B_i$. Since $B_i \in \mathit{args}$ there is some $\widehat{a}_j \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))$ with $\text{setname}(\widehat{a}_j) = B_i$. Since setname is a bijection, we have $\widehat{a}'_i = \widehat{a}_j$. Suppose $a' \in \text{pidaddrs}(a)$. Then by construction $\alpha_{\text{CFA}}(a') \in \text{pidaddrs}(\alpha_{\text{CFA}}(a))$ But then $\alpha_{\text{CFA}}(a') = \widehat{a}''_k$ for some k and $\text{setname}'(\widehat{a}''_k) \in \mathit{recargs}$. So since $B'_i = \text{setname}'(\widehat{a}'_i)$, $B'_i \notin \mathit{recargs}$, by the contrapositive we must have $\widehat{a}'_i \neq \text{setname}(\alpha_{\text{CFA}}(a'))$ for any $a' \in \text{pidaddrs}(a)$.

$$\text{pidaddrs}(q') = \text{pidaddrs}(q) \cup \text{pidaddrs}(\rho'),\ \text{pidaddrs}(\rho') \subseteq \text{pidaddrs}(a),$$

  So

$$\begin{aligned}
T'_i &= \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle \in \text{pidaddrs}(q'), \alpha_{\text{CFA}}(a') = \widehat{a}'_i\}\\
&= \{\text{name}(\iota') : a' = \langle\_,\_,\_,\iota'\rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \widehat{a}_j\}\\
&= T_j = S_i[\text{args}(q)/\mathit{args}][\text{args}(a)/\mathit{recargs}]
\end{aligned}$$

- Suppose $B_i \notin \mathit{args}$ and $B'_i \in \mathit{recargs}$. Then $S_i = B'_i$. Since $B'_i \in \mathit{recargs}$ there is some $\widehat{a}''_k \in \text{pidaddrs}(\alpha_{\text{CFA}}(a))$ with $\text{setname}'(\widehat{a}''_k) = B'_i$. Since setname$'$ is a bijection, we have $\widehat{a}'_i = \widehat{a}''_k$. Suppose $a' \in \text{pidaddrs}(q)$. Then by construction $\alpha_{\text{CFA}}(a') \in \text{pidaddrs}(\alpha_{\text{CFA}}(q))$ But then $\alpha_{\text{CFA}}(a') = \widehat{a}_j$ for some j and $\text{setname}(\widehat{a}_j) \in \mathit{args}$. So since $B_i = \text{setname}(\widehat{a}'_i)$, $B_i \notin \mathit{args}$, by the contrapositive we must have $\widehat{a}'_i \neq \text{setname}(\alpha_{\text{CFA}}(a'))$ for any $a' \in \text{pidaddrs}(q)$.

$$\text{pidaddrs}(q') = \text{pidaddrs}(q) \cup \text{pidaddrs}(\rho'),\ \text{pidaddrs}(\rho') \subseteq \text{pidaddrs}(a),$$

So

$$\begin{aligned} T'_i &= \{\mathrm{name}(\iota') : a' = \langle \_, \_, \_, \iota' \rangle \in \mathrm{pidaddrs}(q'), \alpha_{\mathrm{CFA}}(a') = \widehat{a}'_i\} \\ &= \{\mathrm{name}(\iota') : a' = \langle \_, \_, \_, \iota' \rangle \in \mathrm{pidaddrs}(\rho'), \alpha_{\mathrm{CFA}}(a') = \widehat{a}''_k\} \\ &\subseteq \{\mathrm{name}(\iota') : a' = \langle \_, \_, \_, \iota' \rangle \in \mathrm{pidaddrs}(a), \alpha_{\mathrm{CFA}}(a') = \widehat{a}''_k\} \\ &= T''_k = S_i[\mathrm{args}(q)/\mathit{args}][\mathrm{args}(a)/\mathit{recargs}] \end{aligned}$$

– Suppose $B_i \notin \mathit{args}$ and $B'_i \notin \mathit{recargs}$. This case is impossible, since

$$\mathrm{pidaddrs}(q') = \mathrm{pidaddrs}(q) \cup \mathrm{pidaddrs}(\rho'),\ \ \mathrm{pidaddrs}(\rho') \subseteq \mathrm{pidaddrs}(a),$$

and by arguments in the above cases we have $\widehat{a}'_i \neq \mathrm{setname}(\alpha_{\mathrm{CFA}}(a'))$ for any $a' \in \mathrm{pidaddrs}(q) \cup \mathrm{pidaddrs}(a) \supset \mathrm{pidaddrs}(q')$. So we have a contradiction, and we must be in one of the other cases.

- Suppose $S \to S'$ by the rule Self, with active components (ι, q, q'). Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q, \iota) \twoheadrightarrow \alpha_\pi(q', \iota)$.

$$\begin{aligned} \alpha_\pi(q, \iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\ \alpha_\pi(q', \iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota), \mathrm{name}(\iota) \mid \mathrm{args}(q')\,]. \end{aligned}$$

Since the transition was made by Self, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Self with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned} \alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] &:= \boldsymbol{\tau}.\alpha_{\mathrm{CFA}}(q')[\,s, s \mid \mathit{args}\,] + \dots \\ \mathit{args} &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <) \end{aligned}$$

So

$$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,]$$

and if $\mathrm{args}(q) = \mathrm{args}(q')$, we are done. By Corollaries C.5 and C.11 it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$. Since $q = \langle \ell, \rho, c, t \rangle$, $q' = \langle \iota, \rho, c, t \rangle$, this holds trivially.

- Suppose $S \to S'$ by the rule Spawn, with active components (ι, q, q'). Let $q' = \langle \iota', \rho, c, t \rangle$, $q'' = \pi'(\iota')$. By construction $\iota' \notin \mathrm{dom}\,\pi$. Let $I = \{\iota, \iota'\}$; clearly if $\iota'' \notin I$, $\pi(\iota'') = \pi'(\iota'')$,

$$\sigma_v \cap \mathrm{pidaddrs}(\pi(\iota'')) = \sigma'_v \cap \mathrm{pidaddrs}(\pi(\iota'')),$$

We also have $\mu = \mu'$, so by Lemma C.8, to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show $\alpha_\pi(q, \iota) \twoheadrightarrow \alpha_\pi(q', \iota) \parallel \alpha_\pi(q'', \iota')$.

$$\begin{aligned} \alpha_\pi(q, \iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\ \alpha_\pi(q', \iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota), \mathrm{name}(\iota') \mid \mathrm{args}(q')\,], \\ \alpha_\pi(q'', \iota') &= \alpha_{\mathrm{CFA}}(q'')[\,\mathrm{name}(\iota') \mid \mathrm{args}(q'')\,], \end{aligned}$$

Since the transition was made by Spawn, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule Abs-Spawn with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$. So we have a process definition in the abstraction

$$\begin{aligned} \alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] &:= \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\alpha_{\mathrm{CFA}}(q')[\,s, s' \mid \mathit{args}\,] \parallel \alpha_{\mathrm{CFA}}(q'')[\,s' \mid \mathit{args}'\,]) + \dots \\ \mathit{args} &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <) \\ ' &= \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q''))\}, <) \end{aligned}$$

So

$$\alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,]$$

$$\begin{aligned}\alpha_\pi(q,\iota) &\equiv \boldsymbol{\tau}.\boldsymbol{\nu} s'.(\alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota), s' \mid \mathrm{args}(q)\,] \\ &\qquad \parallel \alpha_{\mathrm{CFA}}(q'')[\,s' \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}']) + \ldots \\ &\twoheadrightarrow \boldsymbol{\nu} s'.(\alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota), s' \mid \mathrm{args}(q)\,] \\ &\qquad \parallel \alpha_{\mathrm{CFA}}(q'')[\,s' \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}']) \\ &\equiv \boldsymbol{\nu}\,\mathrm{name}(\iota').(\alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota), \mathrm{name}(\iota') \mid \mathrm{args}(q)\,] \\ &\qquad \parallel \alpha_{\mathrm{CFA}}(q'')[\,\mathrm{name}(\iota') \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}'])\end{aligned}$$

as long as $\mathrm{name}(\iota') \notin \mathrm{args}(q)$. We have $\mathrm{name}(\iota') \notin \mathrm{args}(q)$, since this would require that $\langle \_, \_, \_, \iota' \rangle \in \mathrm{pidaddrs}(q)$, and this cannot be since ι' is fresh by construction. So if $\mathrm{args}(q) = \mathrm{args}(q')$ and $\mathit{args}'[\mathrm{args}(q)/\mathit{args}'] \sqsupseteq \mathrm{args}(q'')$, we are done. By Lemma C.4 and Corollaries C.5 and C.11 it is enough to show $\mathrm{pidaddrs}(q) = \mathrm{pidaddrs}(q')$, $\mathrm{pidaddrs}(q) \supseteq \mathrm{pidaddrs}(q'')$. Since $q = \langle \ell, \rho, c, t \rangle$, $q' = \langle \iota', \rho, c, t \rangle$, the first of these holds trivially. For the second, note that $q'' = \langle \ell'', \rho', \star, t_0 \rangle$, $\langle \ell', \rho' \rangle = \sigma_v(\rho(U))$. So by construction $\mathrm{pidaddrs}(\rho') \subseteq \mathrm{pidaddrs}(\rho(U)) \subseteq \mathrm{pidaddrs}(\rho)$ and

$$\begin{aligned}\mathrm{pidaddrs}(q') &= \mathrm{pidaddrs}(\rho') \\ &\subseteq \mathrm{pidaddrs}(\rho) \cup \mathrm{pidaddrs}(c) \\ &= \mathrm{pidaddrs}(q)\end{aligned}$$

- Suppose $S \to S'$ by the rule SEND, with active components (ι, q, q'). Let

  $$q = \langle \ell : \texttt{primop 'send'/2 (}U\texttt{,}V\texttt{)}, \rho, c, t \rangle,$$

  $\iota' = \sigma_v(\rho(U))$. Let $I = \{\iota\}$, $J = \{\iota'\}$; clearly if $\iota'' \notin I$, $\pi(\iota'') = \pi'(\iota'')$,

  $$\sigma_v \cap \mathrm{pidaddrs}(\pi(\iota'')) = \sigma'_v \cap \mathrm{pidaddrs}(\pi(\iota'')),$$

  and if $\iota'' \notin J$, $\mu(\iota'') = \mu'(\iota'')$, and for $a \in \mu(\iota'')$,

  $$\sigma_v \cap \mathrm{pidaddrs}(a) = \sigma'_v \cap \mathrm{pidaddrs}(a),$$

  Then by Lemma C.8 we have that to show $\alpha_\pi(S) \twoheadrightarrow \alpha_\pi(S')$ it is enough to show

  $$\alpha_\pi(q,\iota) \parallel \alpha_\pi(\mu \cap \{\iota'\}) \twoheadrightarrow \alpha_\pi(q',\iota) \parallel \alpha_\pi(\mu' \cap \{\iota'\}).$$

  $$\begin{aligned}\alpha_\pi(q,\iota) &= \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,], \\ \alpha_\pi(q',\iota) &= \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q')\,].\end{aligned}$$

  Let $\rho(V) = a$. Then $\mu'(\iota') = \mathrm{enq}(a, \mu(\iota')) = \mu(\iota') \cdot a$, so

  $$\alpha_\pi(\mu' \cap \{\iota'\}) = \mathrm{SEND}_{\alpha_{\mathrm{CFA}}(a)}[\,\mathrm{name}(\iota') \mid \mathrm{args}(a)\,] \parallel \alpha_\pi(\mu \cap \{\iota'\})$$

  Since the transition was made by SEND, we have that $\alpha_{\mathrm{CFA}}(S) \rightsquigarrow \alpha_{\mathrm{CFA}}(S')$ by the rule ABS-SEND with active components $(\alpha_{\mathrm{CFA}}(\iota), \alpha_{\mathrm{CFA}}(q), \alpha_{\mathrm{CFA}}(q'))$, where $\alpha_{\mathrm{CFA}}(\iota') = \alpha_{\mathrm{CFA}}(\sigma_v)(\alpha_{\mathrm{CFA}}(\rho)(U))$. So we have process definitions in the abstraction

  $$\alpha_{\mathrm{CFA}}(q)[\,s \mid \mathit{args}\,] := \mathbf{let}\ s' \in \mathrm{setname}(\alpha_{\mathrm{CFA}}(a)).\,\big(\mathrm{SEND}_{\alpha_{\mathrm{CFA}}(a)}[\,s' \mid \mathit{args}'\,] \parallel \alpha_{\mathrm{CFA}}(q')[\,s \mid \mathit{args}]\big) + \ldots$$
  $$\mathit{args} = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(q))\}, <),\ \mathit{args}' = \mathrm{sort}(\{\mathrm{setname}(\widehat{a}) : \widehat{a} \in \mathrm{pidaddrs}(\alpha_{\mathrm{CFA}}(a))\}, <).$$

  So

  $$\begin{aligned}\alpha_\pi(q,\iota) \parallel \alpha_\pi(\mu \cap \{\iota'\}) &\equiv \alpha_{\mathrm{CFA}}(q)[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \parallel \alpha_\pi(\mu \cap \{\iota\}) \\ &\equiv (\mathbf{let}\ s' \in (\mathrm{setname}(\alpha_{\mathrm{CFA}}(a))[\mathrm{args}(q)/\mathit{args}]).(\\ &\qquad \mathrm{SEND}_{\alpha_{\mathrm{CFA}}(a)}[\,s' \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}] \\ &\qquad \parallel \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,]) + \ldots) \\ &\quad \parallel \alpha_\pi(\mu \cap \{\iota\}) \\ &\twoheadrightarrow \alpha_{\mathrm{CFA}}(q')[\,\mathrm{name}(\iota) \mid \mathrm{args}(q)\,] \\ &\qquad \parallel \mathrm{SEND}_{\alpha_{\mathrm{CFA}}(a)}[\,\mathrm{name}(\iota') \mid \mathit{args}'\,][\mathrm{args}(q)/\mathit{args}] \\ &\qquad \parallel \alpha_\pi(\mu \cap \{\iota\})\end{aligned}$$

if $\text{name}(\iota') \in \text{setname}(\alpha_{\text{CFA}}(a))[\text{args}(q)/\mathit{args}]$. So if $\text{name}(\iota') \in \text{setname}(\alpha_{\text{CFA}}(a))[\text{args}(q)/\mathit{args}]$, $\text{args}(q) = \text{args}(q')$, $\mathit{args}'[\text{args}(q)/\mathit{args}] \sqsupseteq \text{args}(a)$, we are done. By Lemma C.4 and Corollaries C.5 and C.11 for the last two it is enough to show $\text{pidaddrs}(q) = \text{pidaddrs}(q')$, $\text{pidaddrs}(q) \supseteq \text{pidaddrs}(a)$.

If $\sigma_v(a) = \iota'$, then by construction $a = \langle \_, \_, \_, \iota' \rangle$. Since $a \in \operatorname{Im} \rho$ w have $\text{pidaddrs}(a) = \{a\} \subseteq \text{pidaddrs}(\rho)$, so $a \in \text{pidaddrs}(\rho) \subseteq \text{pidaddrs}(q)$. Now again by construction $\alpha_{\text{CFA}}(a) \in \text{pidaddrs}(\alpha_{\text{CFA}}(\rho)) \subseteq \text{pidaddrs}(\alpha_{\text{CFA}}(q))$. So $\text{setname}(\alpha_{\text{CFA}}(a)) \in \mathit{args}$ and

$$\text{setname}(\alpha_{\text{CFA}}(a))[\text{args}(q)/\mathit{args}] = \{\text{name}(\iota'') : a' = \langle \_, \_, \_, \iota'' \rangle \in \text{pidaddrs}(q), \alpha_{\text{CFA}}(a') = \alpha_{\text{CFA}}(a)\}$$

So $\text{name}(\iota') \in \text{setname}(\alpha_{\text{CFA}}(a))[\text{args}(q)/\mathit{args}]$.

$q = \langle \ell, \rho, c, t \rangle$, $q' = \langle a, \rho, c, t \rangle$. $\{a\} \subseteq \text{pidaddrs}(q)$, so clearly $\text{pidaddrs}(q) = \text{pidaddrs}(q')$. □

D Listings for Soter-Compatible Erlang equivalents of Core programs

```
% equivalent of the 'publisher' module
new() -> spawn(fun () -> router([]) end).

router(Dispatchers) ->
  receive
    {sub, Topic, P} ->
      F = fun () ->
        dispatcher(Topic, P)
      end,
      D = spawn(F),
      router([D | Dispatchers]);
    {pub, Topic, Msg} ->
      M = {pub, Topic, Msg},
      G = fun (Pid) -> Pid ! M end,
      map(G, Dispatchers),
      router(Dispatchers)
  end.

dispatcher(Topic, P) ->
  receive
    {pub, Topic, Msg} ->
      M = {pub, Topic, Msg},
      P ! M,
      dispatcher(Topic, P);
    _ ->
      dispatcher(Topic, P)
  end.

map(F, L) ->
  case L of
    [] -> [];
    [X | Xs] ->
      Y = F(X),
      Ys = map(F, Xs),
      [Y, Ys]
  end.
```

```
% equivalent of 'main' module
picky(Topic) ->
  receive
    {pub, Topic, _} ->
      picky(Topic);
    _ -> ?soter_error(bad_topic)
  end.

main() ->
  P = new(),
  spawn_pickies(P).

spawn_pickies(P) ->
  Topic = ?SoterOneOf(['a', 'b']),
  Q = spawn(fun () -> picky(Topic) end),
  P ! {sub, Topic, Q},
  P ! {pub, Topic, m},
  spawn_pickies(P).
```

Listing 6: Soter-compatible ERLANG source for Listing 1.

```
-uncoverable("mail␣>␣1").

main() ->
  Ping = fun (Pid) -> Pid ! 'ping' end,
  Pids = getList(),
  map(Ping, Pids).

map(F, L) ->
  case L of
    [X | Xs] ->
      Y = F(X),
      Ys = map(F, Xs),
      [Y | Ys];
    _ -> []
   end.

getList() ->
  B = ?SoterOneOf(['true', 'false']),
  ca\textit{}se B of
    'true' -> [];
    _ ->
       P = spawn(fun() -> proc() end),
       Ps = getList(),
       [P | Ps]
  end.

proc() ->
  Pid_ = spawn(fun () -> ?label_mail("mail"), 'ok' end),
  receive A -> Pid_ ! A end.
```

Listing 7: Soter-compatible Erlang source for Listing 2.

```
% equivalent of the 'reslock' module
-module(reslock).

reslock_new(Handler, InitState) ->
  spawn(fun () ->
    unlocked(Handler, InitState)
  end).

unlocked(Handler, State) ->
  receive {acquire, Pid} ->
    Pid ! {acquired, self()},
    locked(Handler, State, Pid)
  end.

locked(Handler, State, Owner) ->
  receive
    {release, Owner} ->
      unlocked(Handler, State);
    {request, Owner, Req} ->
      Result = Handler(State, Req),
      case Result of
        {ok, NewState} ->
          locked(ModName, NewState, Owner);
        {reply, NewState, Reply} ->
          Owner ! {reply, self(), Reply},
          locked(ModName, NewState, Owner)
      end
  end.

reslock_acquire(Res) ->
    Res ! {acquire, self()},
    receive {acquired, Res} -> ok end.

reslock_release(Res) ->
    Res ! {release, self()},
    ok.

tell(Res, Req) ->
    Res ! {request, self(), Req},
    ok.

ask(Res, Req) ->
    Res ! {request, self(), Req},
    receive {reply, Res, Reply} -> Reply end.
```

```
% equivalent of the 'cell' module
handle(State, Req) ->
  case Req of
    read -> {reply, State, State};
    {write, NewState} -> {ok, NewState}
  end.

cell_new(InitValue) ->
  Handler = fun reslock:handle/2
  reslock_new(Handler, InitValue).

cell_acquire(Cell) -> reslock_acquire(Cell).
cell_release(Cell) -> reslock_release(Cell).
read(Cell) -> ask(Cell, read).
write(Cell, X) -> tell(Cell, {write, X}).

% equivalent of the 'main' module
main() ->
  repeat(?any_peano(), fun () ->
    Cell = cell_new(z),
    repeat(?any_peano(), fun () ->
      spawn(fun -> inc(Cell) end)
  end) end).

repeat(N, F) ->
  case <N> of
    {s, M} -> F(), repeat(M, F);
    _ -> ok
  end.

inc(Cell) ->
  cell_acquire(Cell),
  ?label(critical),
  X = read(Cell),
  write(Cell, {s, X}),
  %% end critical section
  cell_release(Cell).
```

Listing 8: Soter-compatible ERLANG source for Listings 3 to 5.