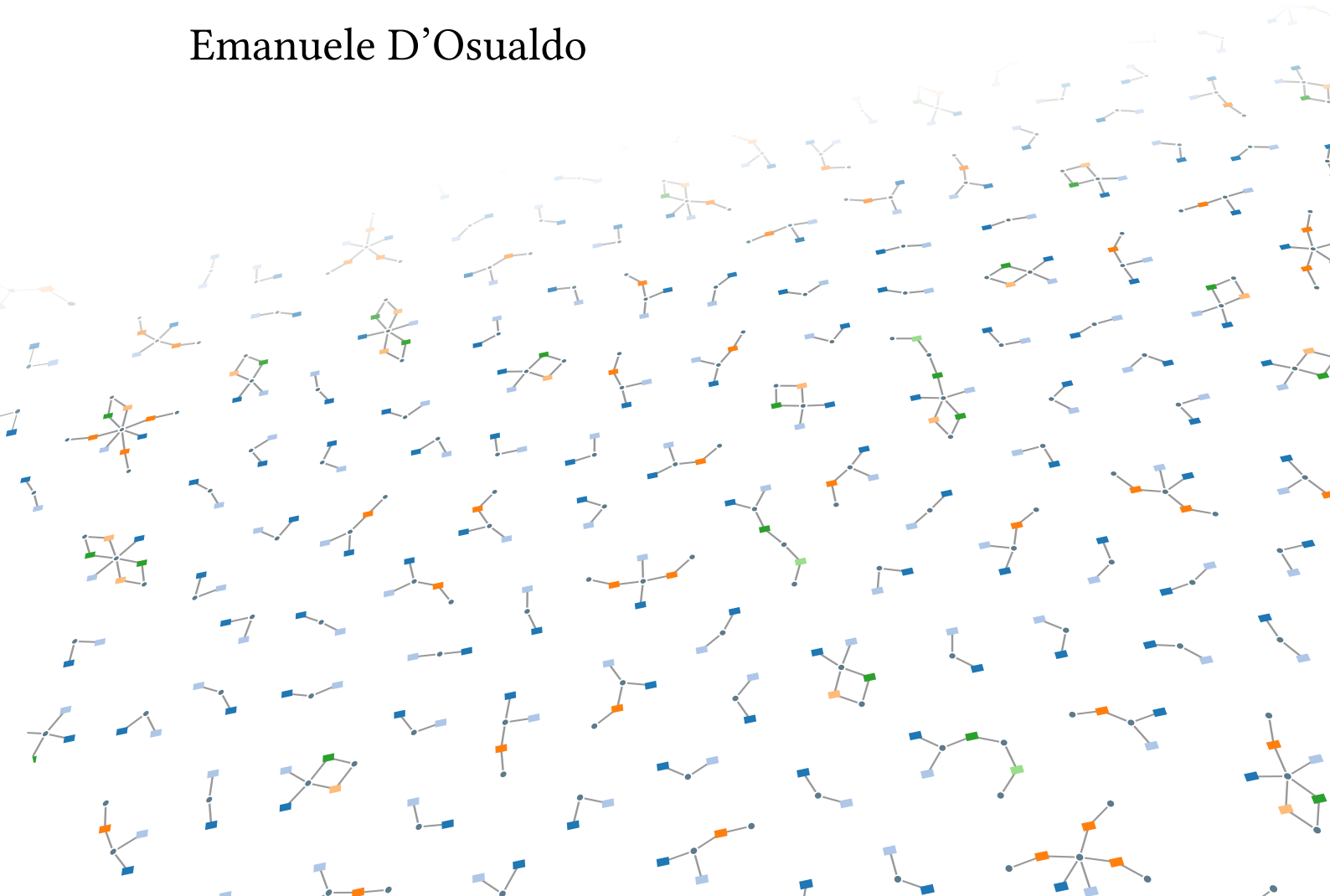
University of Oxford
Department of Computer Science

Doctor of Philosophy in Computer Science

Verification of Message Passing Concurrent Systems

Emanuele D'Osualdo



University of Oxford
Department of Computer Science

Doctor of Philosophy in Computer Science

Verification of Message Passing Concurrent Systems

Emanuele D'Osualdo

Merton College, University of Oxford

Supervisor

Prof. C.-H. Luke Ong
University of Oxford

Examiners

Prof. Simon Gay
University of Glasgow

Prof. Hongseok Yang
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science

Hilary Term 2015

Final version submitted September 2015.
The officially deposited version can be found at the Oxford University Research Archive:
http://ora.ox.ac.uk/objects/uuid:f669b95b-f760-4de9-a62a-374d41172879

This version was produced on 25th October 2016 and contains corrections of minor typos.

Typeset using LaTeX and the Memoir class. The main text uses the Linux Libertine font, the monospaced text uses DejaVu Sans Mono.

Most of the illustrations are produced using the Ti*k*z package.

The illustration on the cover is an artistic view of a π-calculus system.
The illustration "Cada Homem É Uma Ilha", on the title page of Part I,
is courtesy of Vicente Sardinha © 1988.

Abstract

This dissertation is concerned with the development of fully-automatic methods of verification, for message-passing based concurrent systems.

In the first part of the thesis we focus on Erlang, a dynamically typed, higher-order functional language with pattern-matching algebraic data types extended with asynchronous message-passing. We define a sound parametric control-flow analysis for Erlang, which we use to bootstrap the construction of an abstract model that we call *Actor Communicating System* (ACS). ACS are given semantics by means of *Vector Addition Systems* (VAS), which have rich decidable properties. We exploit VAS model checking algorithms to prove properties of Erlang programs such as unreachability of error states, mutual exclusion, or bounds on mailboxes. To assess the approach empirically, we constructed Soter, a prototype implementation of the verification method, thereby obtaining the first fully-automatic, infinite-state model checker for a core concurrent fragment of Erlang.

The second part of the thesis addresses one of the major sources of imprecision in the ACS abstraction: process identities. To study the problem of algorithmically verifying models where process identities are accurately represented we turn to the π-calculus, a process algebra based around the notion of *name* and *mobility*. The full π-calculus is Turing-powerful so we focus on the *depth-bounded* fragment introduced by Roland Meyer, which enjoys decidability of some verification problems. The main obstacle in using depth-bounded terms as a target abstract model, is that depth-boundedness of arbitrary π-terms is undecidable. We therefore consider the problem of identifying a fragment of depth-bounded π-calculus for which membership is decidable. We define the first such fragment by means of a novel type system for the π-calculus. Typable terms are ensured to be depth-bounded. Both type-checking and type inference are shown to be decidable. The constructions are based on the novel notion of $\mathcal{T}$-compatibility, which imposes a hierarchy between names. The type system's main goal is proving that this hierarchy is preserved under reduction, even in the presence of unbounded name creation and mobility.

Acknowledgements

First, I would like to thank my supervisor Prof. Luke Ong: I am grateful to him for holding me to a high research standard and for teaching me how research is done, with great patience. I would like to thank Prof. Hongseok Yang, who was in the commission for my transfer and confirmation of status, for the detailed, honest and constructive feedback. I am particularly grateful to my examiners Prof. Hongseok Yang and Prof. Simon Gay for the careful work of review and for making my viva a very enjoyable experience. I am also grateful to Prof. Roland Meyer who provided encouraging and insightful feedback on the second part of this thesis. Thanks also to Dr. Damien Zufferey for the useful discussions on depth-boundedness.

I acknowledge the financial support from the Scatcherd European Scholarship and Merton College that funded my DPhil studies.

A big thank you is due to the awesome Julie Sheppard, the Graduate Studies Administrator, who never missed an email and restlessly provided prompt and reliable advice. Without her I would still be waiting for the Queen's signature on a form for misreading the regulations.

Every member of the LOng group contributed in one way or another to the completion of this thesis. Thank you all, in order of appearance: Robin Neatherway, Steven Ramsay, Martin "φ" Lester, Michael Vanden Boom, Jonathan Kochems, Chang Yan, Marcelo Sousa, Egor Ianovski, and Conrad Cotton-Barratt. It has been an honor to share our natural-light-deprived office with you, both on a professional and a personal level.

A special thanks goes to Dr. Ani Calinescu who proved with her kindness that you can be yourself even in a pool of sharks.

To Prof. Marino Miculan, Prof. Marco Comini and Prof. Angelo Montanari: it is thanks to your support, advice and, not least, your example that I choose this path. It is perhaps not a coincidence that my thesis should be on π-calculus, static analysis and model checking! Similarly, I feel deeply indebted to many of my teachers, the influence of whom is present in this dissertation: Diana Bitto, Giuseppina Trifiletti, Franca Alborini, Roberto Grison, Renato Miani, Sebastiano Sonego. The impact they had on me both personally and professionally is inestimable.

Many friends have helped me stay sane through these difficult years. I am deeply grateful for their support and affection. Thank you Enrico Sandretti, Gabriele & Francesca

Puppis, Caterina Guardini, Nunzio Turtulici, Francesca Ferasin, Michele De Marchi; Vikaran Khanna, Kristina Gedgaudaitė, Johannes Moeller, Charlie Taylor, Christian Matek, Alessandro Zocco, Adèle Jatteau. Your example has been a constant source of inspiration.

Thanks to Vicente Sardinha for letting me use his wonderfully quirky illustration "Cada Homem É Uma Ilha".

❧

Un ringraziamento speciale va alla mia famiglia. Francesca e Tommaso: quando il gioco si fa duro ci siete sempre per me. Alice e Elena: la lontananza si paga in due; grazie per tenere duro. Mamma Luciana e papà Claudio: tutto quello di buono che sono riuscito a fare parte dal vostro amore, la vostra cura, la vostra fiducia, il vostro generoso e incondizionato sostegno.

Un sentito grazie va anche a Silvia Driussi ed Enzo Sandretti per il loro continuo supporto e incoraggiamento.

Ma soprattutto voglio ringraziare la mia compagna Anna Sandretti. Il suo sostegno e la sua fiducia sono stati fondamentali per la realizzazione di questa tesi. Hai creduto in me nei momenti bui, hai gioito con me nei momenti felici. Ti sei presa cura di me come solo una vera amica può fare.

Contents

List of Figures

List of Definitions

Chapter 1

Introduction

A living cell. In the nucleus a chain of messenger RNA is being assembled by a large number of proteins around the template offered by the DNA. In the meantime, in the cytoplasm, enzymes are bonding translation RNA molecules to amino acids. In a short while, the messenger RNA will reach the cytoplasm where ribosomal units will navigate through it to mark the spots where the translation RNA carrying the right amino acid will land. Peptide bonds will form between adjacent amino acids and a protein will have taken shape.

A colony of bacteria. They are accumulating in their host's system. Each bacterium has very limited knowledge of its surroundings. It knows that attacking the host when the colony is too small would mean certain defeat. But they have a plan. They produce a molecule and monitor its concentration in their proximity. When a certain threshold is exceeded, they know the colony must have grown big enough for the big attack.

A room like many others. A man is buying a book on an online shop. He clicks 'Buy now', but little does he know that his gesture will generate an enormous amount of signals, travelling through the network, all conspiring and cooperating to communicate his intent and actuate it. The browser sends a packet of data with his request which gets dispatched by many routers before reaching a server that will decide by which machine it will be processed. Algorithms will process the signal, updating the records of supply and the personal records of the buyer. This information is stored redundantly and is distributed in physically separate places: it will take some time for the wave of new information to reach all of them. Meanwhile, the payment request reaches the bank and a protocol is initiated to ensure that all the parties agree on the terms of the transaction. All the user sees is a banner asking him to 'like' the book on his profile. Another click and we are back to the start.

What do these three situations have in common? In all three cases, the essence of what is happening can be captured as a large ensemble of independent agents interacting locally to obtain a global effect. Despite their ubiquity, our formal understanding of this kind of discrete dynamic systems is still quite primitive. Understanding what are the laws

that regulate communication between concurrent processes is one of the great challenges of modern science.

This dissertation is concerned with computational systems organised as an evolving collection of processes interacting through message passing. With the growing number of sensitive tasks that we delegate to algorithms, the development of methods to certify their trustworthiness is of paramount importance. Formal methods are here to help us: they provide tools to formalise the notion of correctness and to prove it. In the process of developing these tools we deepen our understanding of the underlying laws and structure of communicating systems.

Our contribution is grounded on formal methods and specifically on techniques that allow correctness proofs to be found algorithmically. Two big ideas enable this line of work: abstraction and symbolic semantic reasoning. An abstraction removes details that are not relevant, or conservatively simplifies the view of the system until a property is evident or easier to prove. Designing good abstractions requires insight on what is the essence of the phenomenon under study. Symbolic semantic reasoning on the other hand allows the expression and manipulation of potentially infinite objects (behaviours) by finite means.

We focus on an industrial strength programming language: Erlang. Its concurrency model is a particularly clean and succinct concurrent extension of a functional language. The contribution of this thesis to the field is twofold. First, we propose an automatic abstraction technique which is able to exploit infinite state model checking to prove safety properties of Erlang programs. This part culminates in the development of a prototype tool. Second, we analyse one of the major shortcomings of the proposed abstraction: the representation of the identity of unboundedly many processes participating in interactions. We propose the concept of $\mathcal{T}$-compatibility as a way to organise identities according to a hierarchy. Systems obeying the hierarchy can be model-checked more accurately using powerful symbolic representations of the reachable terms.

1.1 An Erlang Primer

Let us overview the key features of Erlang through an example. We present a simplified program implementing a server. The server expects two kinds of requests: a ping or a visit. A ping request is dealt with by sending a ping back. The server wants to keep track of visits using a counter stored in a database. We want to avoid blocking the server while replying to a request, so when access to the database is needed a worker process is spawned to deal with the request and answer to the client directly. In this way the server can answer to other requests while the workers interact with the database and the clients.

Erlang has built-in support for running multiple processes concurrently either locally or in a distributed environment, where processes run on physically remote nodes. The server process runs the `serve` procedure below:

```
serve(DB) →
  receive
    {ping , P} →
      P ! {pong, self()};
    {visit, C} →
      spawn(fun()→ worker(DB, fun(X)→ X+1 end, C) end)
  end,
  serve(DB).
```

A `receive` construct declares how to react to incoming messages. A send instruction takes the form `P ! Msg` where `P` is a *process identifier* (pid), which acts as an address, and `Msg` is the message to be sent, which can consist of an arbitrary Erlang value.

Erlang's communication model is based on asynchronous message passing. Every process is equipped with an unbounded buffer of incoming messages called its *mailbox*. A `receive` instruction inspects the mailbox looking for messages that match a pattern and reacts if one is found, or blocks, waiting for more messages, if no matching message is currently in the mailbox.

According to the definition of the `serve` function, the server process expects to receive messages of two forms, `{ping, P}`, a tuple tagged with the atomic value `ping` and carrying a pid `P` as a payload, or `{visit, C}` a tuple tagged with the atomic value `visit` and carrying a pid `C` as a payload.[1]

In the case of a ping message, the server reacts by sending a `pong` reply to the process `P`. Since communication is asynchronous, the server does not wait the process at `P` to receive the message before continuing and serving the next request.

In the case of a visit message, the primitive `spawn` is used to create a new process running the function `worker`, which we will describe later. Once the process is initialised, the `spawn` primitive returns the pid of the newly created process to the caller.

A notable feature used in this definition is higher-order: the `worker` function takes a function as second argument. This function will be used by workers to compute the new value to store in the database from the old value.

The database is represented by a process running `database(State)`; the variable `State` will hold the current contents of the database. The process reacts to messages requesting to write or read the value it stores. In order to prevent data-races, the database must be locked before writing/reading, and unlocked afterwards.

[1]Despite the confusing syntax, curly braces do not denote sets in Erlang. A term `{A,B}` should be read as a tuple constructor applied to two arguments.

```
database(State) →
  receive {lock, C} →
    C ! {ready, self()},
    lockeddb(State, C)
  end.

lockeddb(State, Owner) →
  receive
    {read, Owner} →
      Owner ! {val, self(), State},
      lockeddb(State, Owner);
    {write, X, Owner} →
      lockeddb(X, Owner);
    {unlock, Owner} →
      database(State)
  end.
```

The worker updates the database by following the locking protocol, and then sends a reply to the client assigned to it:

```
worker(DB, F, C) →
  DB ! {lock, self()},
  receive {ready, DB} →
    % Start of critical section
    DB ! {read, self()},
    receive {val, DB, X} →
      DB ! {write, F(X), self()},
      DB ! {unlock, self()},
    % End of critical section
      C  ! {reply, X}
    end
  end.
```

The locking mechanism should ensure mutual exclusion between the workers: the critical section is executed at most by one worker at once.

The function `client` defines a simple client that visits the server repeatedly:

```
client(Server) →
  Server ! {visit, self()},
  receive {reply, _} →
    client(Server)
  end.
```

The entry point of the program is the function `main`, which sets up a system composed of a single database, a single server and `N` clients.

```
main(N) →
  DB = spawn(fun()→ database(0) end),
  S  = spawn(fun()→ serve(DB) end),
  spawn_clients(N, S).

spawn_clients(0, S) → ok;
spawn_clients(N, S) →
  spawn(fun()→ client(S) end),
  spawn_clients(N-1, S).
```

We are interested in safety properties: a (user-specified) set of bad configurations can never be reached. In our example, a first safety property we consider is mutual exclusion of the workers, that is, no two workers can think they both acquired the lock on the database. This can be formalised by stating that the number of processes running the critical section of `worker` cannot exceed 1.

A second property we consider is about the mailbox of the clients: the number of messages in the mailbox of each client is bounded by 1. In fact, after updating the database, each worker sends a single message to the client assigned to it; no two workers are assigned the same client, so the property should hold. We use these two properties to illustrate how Soter works.

Note that, since we let `N` range over naturals, proving these properties requires reasoning about an unbounded number of processes and interactions.

1.2 Soter's verification approach

In Part I, we study the problem of automatically verifying properties of programs like the one just presented. Clearly, an automated approach cannot hope to be able to prove arbitrary safety properties of arbitrary Erlang programs: it is an undecidable task. To side-step this problem, static analysis and model checking methods adopt a strategy based on approximation: the semantics of the input program is over-approximated so that if no bad state can be reached in the approximation it cannot be present in the exact semantics. The challenge is then finding the right trade-off between *decidability* of the safety of the approximation and the *precision* of the approximation itself. If the approximate semantics is too precise the analysis could take too much time or become undecidable, if it is too imprecise there will be too many false alarms.

We identify an abstract model that is expressive enough to capture properties as mutual exclusion or bounds on mailboxes, and allows to check these properties automatically at the same time. We call these models *Actor Communicating Systems* (ACS). They incorporate our choices on how the various features of Erlang should be approximated.

The first important constraint of ACS is that they only support a finite set of control states. On the contrary, each Erlang process can possess an infinite state space:

data structures are not bounded in size, general recursion requires an unbounded stack and higher-order allows arbitrarily nested closures to be created. Additionally, the control-flow of each individual process is dependent on side effects such as spawning or communication.

We address this problem by defining a parametric abstract interpretation of Erlang programs that is able to extract a finite approximation of the control-flow, taking in account internal computation and interaction between processes.

The result of this analysis is then used in a second phase to construct an ACS model of the program. The ACS model computed by one of the possible instances of our analysis from our running example is graphically rendered in Figure 1.1.

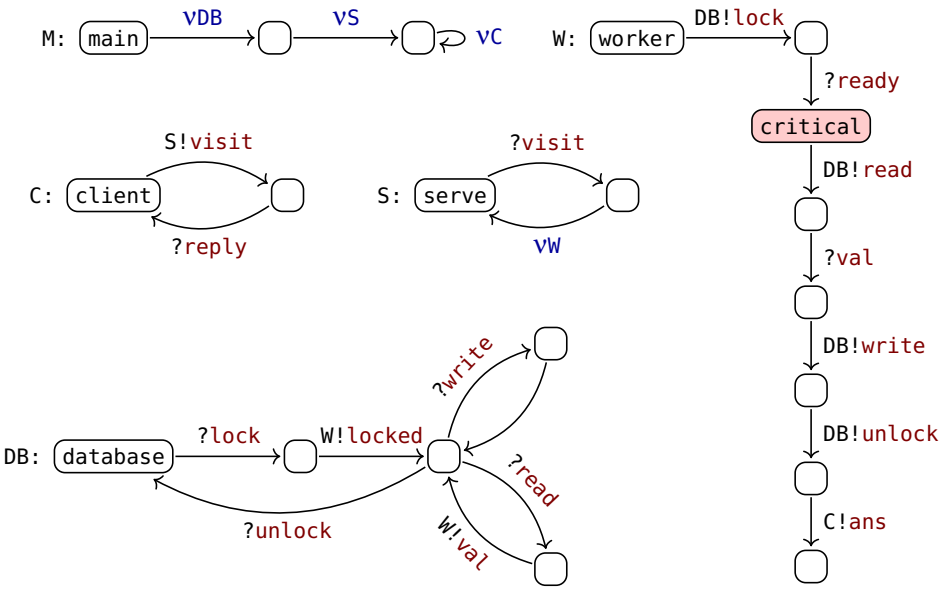


Figure 1.1 – An abstract model of a server. The marked control state represents the start of the critical section of a worker.

The rounded boxes represent control states, the labels of transitions represent side effects: `νP` is a spawn action, `P!msg` is a send and `?msg` is a receive. Even before explaining what is the semantics of the side effects, we can make three observations on the analysis. First, the data values, and hence the messages, are abstracted to a finite set. While the analysis itself is parametric on the data abstraction, often simple choices give enough precision to prove interesting properties. Second, the analysis had to take into account the non trivial control-flow of higher-order computation in order to soundly over-approximate the possible transitions between control states. Third, the analysis is whole-program, which means that the control-flow is analysed assuming the system which is setup by `main` is isolated and autonomous. In our example this implies that,

since no process is sending ping requests, the server's abstraction does not include a transition receiving a `ping` message.

So far, we only explained our choices regarding the abstraction of the control-flow. While the analysis extracting the ACS model has a very coarse view of the concurrency model, we equip ACS of an infinite-state semantics that is capable of capturing salient properties of spawning and message-passing.

Let us first consider process spawning. Erlang processes can spawn other processes dynamically, as for example done by the server when reacting to a visit. Furthermore, the unique pid generated by a spawn, can be stored, communicated and used as an address, allowing the network of processes—the so-called *communication topology* of the system—to evolve during time. We approximate this phenomenon using the concept of *pid-classes*: we cluster the spawned processes grouping them into a finite set of abstract identifiers (the pid-classes). A pid-class has two functions: it identifies the possible control-flow of each spawned process associated to it, and it serves as an abstract address to which messages can be sent. In the case of our server example, the analysis identified 5 pid-classes, `M`, `C`, `S`, `W` and `DB`. Every time an ACS transition labelled with `νP` is executed, for a pid-class `P`, a new process with the abstract control-flow associated to `P` is created. Since the addresses of processes with the same pid-class cannot be distinguished in the ACS semantics, to represent a configuration it is sufficient to keep a counter for each couple pid-class/control state. A spawn action is then executed by incrementing the counter of the initial state of the control-flow graph associated with the pid-class to be spawned. The way pid-classes are generated by the analysis is also a parameter; in our example we use a simple instance that generates a distinct pid-class for each syntactic occurrence of `spawn` in the program.

Our design choices on pid-classes determine an abstraction on the semantics of message-passing: the mailboxes of processes in the same pid-class are conflated. When an ACS transition labelled with `P!msg` is executed, the message is sent to a mailbox associated with the pid-class `P`, which will include the original addressee but also any other process sharing the same pid-class.

A final abstraction is needed to represent (the conflated) mailboxes. Erlang's mailboxes have a quite expressive buffering and matching mechanism. The messages are enqueued by order of arrival and are examined by `receive` starting from the oldest non-processed message. Keeping this sequencing information in ACS would make proving safety properties undecidable even after the pid-class abstraction. We therefore apply a second counter abstraction: the exact order of arrival is lost and the ACS semantics only keeps a counter for each different kind of message in each mailbox. Specifically, a send action `P!msg` in the ACS has the effect of incrementing the counter associated to messages `msg` in the mailbox of the pid-class `P`. Symmetrically, a receive action `?msg` in an ACS can be executed when the corresponding counter is not zero, with the effect of decrementing the counter.

The analysis builds the ACS in a way that guarantees that if bad states cannot be

reached in the ACS semantics, they cannot be reached in the original program. A state of an ACS is an assignment of values to each of the counters of its semantics. We say that a state of the ACS s covers another state s' just if each counter has a greater (or equal) value in s than in s'. The *coverability problem* for an ACS asks, given two states s and s', whether it is possible to reach from s a state covering s'. Our choice of the semantics for ACS models makes them equivalent to *Vector Addition Systems* (VAS), also known as *Petri nets*. From this correspondence we know that coverability is decidable for ACS.

In our running example we can formulate the mutual exclusion property as an ACS coverability problem by asking whether from the initial state one can cover the state assigning 2 to the counter associated with the control state marked in Figure 1.1 (and 0 to all the other counters). This is not the case in the ACS model, proving that in no reachable configuration two workers can be executing the critical section at the same time.

The verification method outlined above (and detailed in Chapter 5), has been implemented in a tool called Soter, that we present in Chapter 6. A web interface for the tool is available at `http://mjolnir.cs.ox.ac.uk/soter/`.

While the approach proves to be successful for the first property, it fails for the second one: the Soter tool would report a false alarm when checking that the mailbox of each client cannot hold more than one message at a time. The reason why Soter is not able to verify this is due to the pid-class abstraction: the mailboxes of all the clients are conflated under the same pid-class `C`; Soter cannot distinguish between the (safe) state where two clients have one message each and the (unsafe) one where one of the two clients holds both messages. Part II of this dissertation describes an attempt to remedy this limitation.

1.3 Process Identities and the π-calculus

In Part II we direct our attention to the issue of precisely representing process identities in message-passing programs. In order to abstract away irrelevant details and orthogonal issues, we move away from Erlang and focus on π-calculus models of concurrent computation.

The π-calculus is a concise yet expressive model of concurrent computation, introduced in Milner, Parrow and Walker [MPW92]. The purpose of this calculus is to model mobile systems with a very small number of fundamental primitives. Communication is synchronous and is done via channels: a process that knows the name of a channel c can listen on it for messages using the input action $c(x)$ or send messages over it using the output action $\overline{c}\langle x\rangle$. A system exhibits mobility if its *communication topology*, i.e. the graph linking processes that share channels, changes over time.

In the π-calculus a new private channel x can be created using the restriction operator $\nu x.P$: the process P now knows the name of x but it can, for instance, output it over a public channel c with an action $\overline{c}\langle x\rangle$. At this point any other process listening on c can

receive the name x and use it for further communication. This mechanism offers two main advantages over ACS. First, the pid-class abstraction is not needed as a new pid can be represented accurately using restrictions. Second, ACS do not exhibit mobility: pid-classes can be understood as static global channels; moreover, since messages in an ACS consist of simple constants a process cannot acquire knowledge of a pid-class it did not know before by receiving it in a message.

However, the price one pays for this increase in expressivity is that π-calculus, unlike ACS, is Turing-complete and, as a consequence, its verification problems are undecidable. The strategy we follow to overcome this difficulty is to constrain the shape of the systems we consider so that 1) some class of safety properties becomes decidable, 2) we can still represent an unbounded number of names precisely and 3) we can still express some form of mobility.

To illustrate our key observations, let us go back to the server example. Soter could not see that each client is assigned to a single worker. While ACS are not powerful enough to represent this fact accurately, a π-calculus model can. Instead of presenting a π-term for the whole program, let us concentrate on the pattern that is the key to obtain the desired coupling between clients and worker: the client/server pattern.

In essence, the client/server pattern assumes that the server is listening for requests on a public many-to-one channel; the client sends, along with its request, a private channel to the server. When the server is ready to reply, it sends the answer to the client using the private channel in the request. This ensures that the answer reaches the right client and that client only, without the need for the server to know all the names of the clients in advance.

Here we summarise the pattern as found in Erlang and in the π-calculus: the client sends the request and starts to listen on its private channel (its own mailbox in Erlang) for a reply:

```
Server ! {request, self()},
receive {answer, X} → ...
```

$$\nu self.(\overline{server}\langle self\rangle \parallel self(x).\cdots)$$

The server runs an infinite loop replying to requests:

```
serve(Y) →
  receive {request, Client} →
    Client ! {answer, Y},
    serve(Y)
  end.
```

$$!(server(client).\overline{client}\langle y\rangle)$$

On the π-calculus side, the expression $P \parallel Q$ represents the parallel composition of two processes P and Q. Dots between actions mean that the actions should be performed in sequence. The replication operator $!P$ can be interpreted as the parallel composition of an infinite number of copies of P. In other words, $!(\pi.P)$ can reduce to $!(\pi.P) \parallel P$ by executing the action π.

A minimal implementation of the pattern in the π-calculus is the following term:

$$\nu s.\Big(\underbrace{!\big(\tau.\nu c.(\overline{s}\langle c\rangle \parallel c(x))\big)}_{C} \parallel \underbrace{!\big(s(x).\nu y.\overline{x}\langle y\rangle\big)}_{S} \Big)$$

where τ is an internal action. After executing the internal action of C three times we obtain the term

$$\nu s.\big(C \parallel S \parallel \nu c_1.(\overline{s}\langle c_1\rangle \parallel c_1(x)) \parallel \nu c_2.(\overline{s}\langle c_2\rangle \parallel c_2(x)) \parallel \nu c_3.(\overline{s}\langle c_3\rangle \parallel c_3(x))\big)$$

where we disambiguated the three occurrences of νc by α-renaming them. Now if we let S react to $\overline{s}\langle c_3\rangle$ we obtain

$$\nu s.\big(C \parallel S \parallel \nu c_1.(\overline{s}\langle c_1\rangle \parallel c_1(x)) \parallel \nu c_2.(\overline{s}\langle c_2\rangle \parallel c_2(x)) \parallel \nu c_3.\nu y.(\overline{c_3}\langle y\rangle \parallel c_3(x))\big)$$

which represents the situation where two clients sent pending requests and the server just sent a reply to the third client.

While arbitrary terms may represent more complex structures, this pattern has a property that enables automatic verification: it is *depth-bounded*. Depth boundedness is a condition proposed in [Mey08] that, when met, makes some verification problems decidable. To understand the intuition behind the concept of depth, let us inspect the communication topology of the client/server pattern after the steps described above, in Figure 1.2. The figure shows an hypergraph with processes for nodes and names as hyperedges. A node is connected to an hyperedge representing a name when it knows that name.

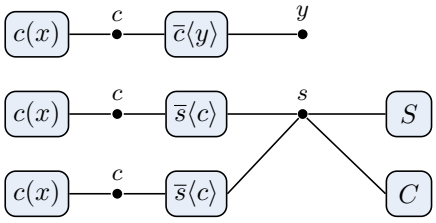


Figure 1.2 – A graphical representation of a reachable term in the client/server pattern.

A *simple path* is a path with no repeated edges. The notion of depth of a term is tightly related to the length of the simple paths in the communication topology of the term. The maximum length of the simple paths of Figure 1.2 is 3. A system is depth-bounded if there exists a bound on the length of the simple paths of any reachable term.

It can be proven that in the client/server pattern the length of the simple paths never exceeds 4. Our system is depth-bounded and we can prove that no client receives more than an answer by requiring that no reachable term can contain

$$\nu c.\big(\nu y.\overline{c}\langle y\rangle \parallel \nu z.\overline{c}\langle z\rangle \parallel c(x)\big) \parallel \nu s.(S \parallel C)$$

as a sub-system.[2] Checking this kind of properties is decidable for depth-bounded systems [Mey08].

Not all systems are depth-bounded, however. Consider for example the following Erlang program

```
ring() → self() ! token, master(self()).

master(Head) →
  receive Msg →
    New = spawn(fun()→ slave(Head) end),
    New ! Msg,
    master(New)
  end.

slave(Next) →
  receive Msg →
    Next ! Msg,
    slave(Next)
  end.
```

A process running `ring` bootstraps a ring of two processes, a master and a slave. Slaves simply forward any received message to the next process in the ring. When the token reaches the master, a new slave is added to the ring and the token is forwarded to it.

The communication topology of the ring example evolves as in Figure 1.3. It is clear

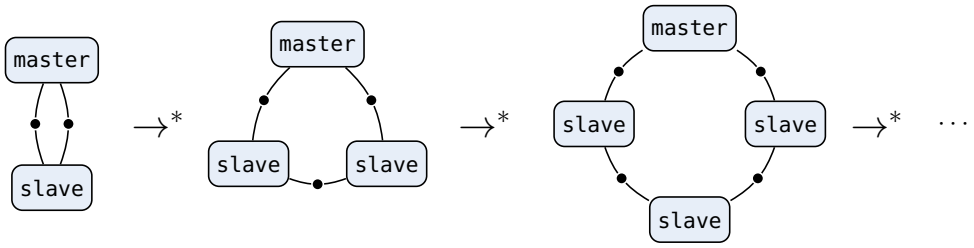
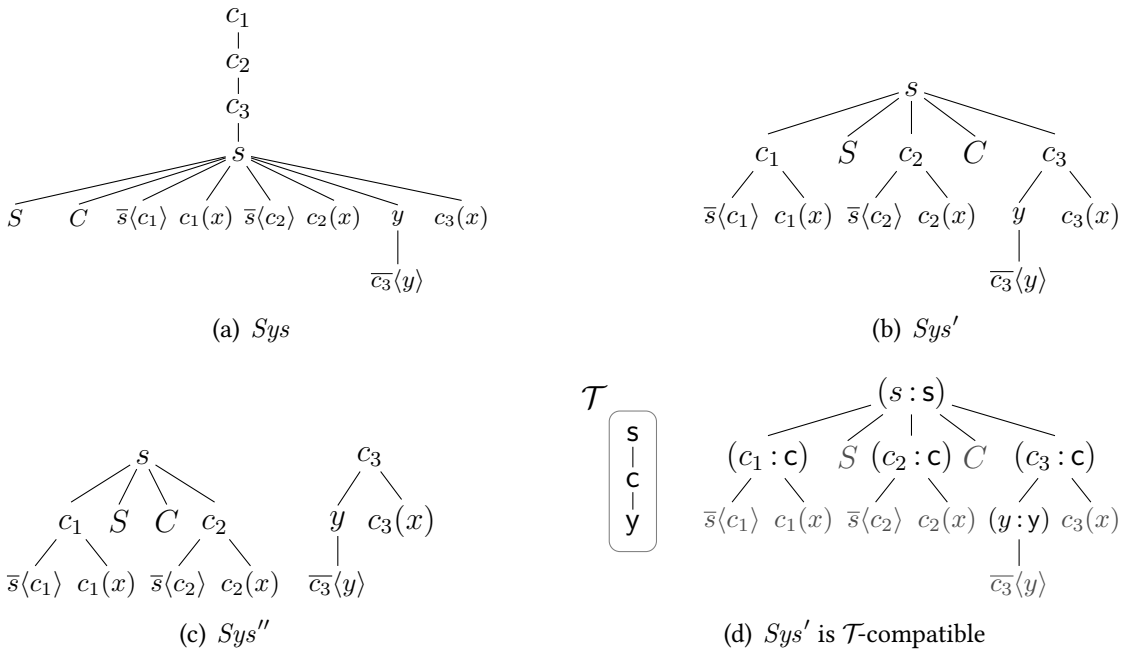


Figure 1.3 – The ring example is not bounded in depth.

that the length of the simple paths in the reachable terms is not bounded and therefore the system is not depth-bounded.

When trying to apply verification procedures developed for depth-bounded systems to arbitrary programs, we are blocked by a serious issue: depth boundedness is an undecidable property [Mey09b]. The main contribution of Part II is a characterisation of the structure of some depth-bounded systems which enables the definition of a meaningful and expressive static fragment of the depth-bounded π-calculus. By 'static fragment' here we mean a set of terms with decidable membership.

[2]The precise meaning of 'sub-system' will be formalised in Definition 9.6.



(a) Sys (b) Sys'

(c) Sys'' (d) Sys' is $\mathcal{T}$-compatible

Figure 1.4 – Forest representations of the Sys term.

1.4 Hierarchical Systems

The goal of Part II is to devise a method to statically determine if an arbitrary π-calculus term is depth-bounded or not. In order to find such a method we introduce the novel concept of $\mathcal{T}$-compatibility and a type system which enforces invariance of $\mathcal{T}$-compatibility under reduction, which implies depth boundedness.

Let us explain the key ideas behind this construction using the client/server pattern example. We observe that the bound on the depth of the system is carrying very little information on the structure of the reachable terms. We want to more strongly characterise the shape of the reachable terms.

In addition to communication topologies, there is a second simple representation that can help us visualise the structure of terms: a version of the abstract syntax tree of the term.

Take the term represented in Figure 1.2:

$$Sys = \nu s.\nu c_1.\nu c_2.\nu c_3.\big(S \parallel C \parallel \overline{s}\langle c_1\rangle \parallel c_1(x) \parallel \overline{s}\langle c_2\rangle \parallel c_2(x) \parallel (\nu y.\overline{c_3}\langle y\rangle) \parallel c_3(x)\big)$$

its forest representation is depicted in Figure 1.4(a). The internal nodes are labelled with restrictions, the leaves with the active processes, i.e. processes that are ready to perform an action. Parallel composition is implicitly represented by (unordered) branching.

In the π-calculus there is a very important relation between terms called *structural congruence* ($\equiv$): it equates terms that differ only in irrelevant presentation details, but not in their behaviour. For instance we have that terms can be α-renamed, and the parallel operator is commutative and associative, i.e. $P \parallel Q \equiv Q \parallel P$ and $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$. The structural congruence laws for restriction tell us that the order of restrictions is irrelevant—$\nu x.\nu y.P \equiv \nu y.\nu x.P$—and that the scope of a restriction can be extended to processes not referring to the restricted name—that is $(\nu x.P) \parallel Q \equiv \nu x.(P \parallel Q)$ when x does not appear free in Q—without altering the meaning of the term. The former law is called *exchange*, the latter one is called *scope extrusion*.

By using the laws of structural congruence we can derive a number of terms that are equivalent to Sys. We give two examples:

$$Sys' = \nu s.\Big(S \parallel C \parallel C_1 \parallel C_2 \parallel \nu c_3.\big((\nu y.\overline{c_3}\langle y\rangle) \parallel c_3(x)\big)\Big)$$
$$Sys'' = \nu s.\big(S \parallel C \parallel C_1 \parallel C_2\big) \parallel \nu c_3.\big((\nu y.\overline{c_3}\langle y\rangle) \parallel c_3(x)\big)$$

where $C_i = \nu c_i.\big(\overline{s}\langle c_i\rangle \parallel c_i(x)\big)$. Their forest representation is shown in Figures 1.4(b) and 1.4(c) respectively.

The depth of a term P corresponds precisely to the minimum height of the trees in the forest representation of terms structurally equivalent to P. In the case of the term Sys, we have that the tree in Figure 1.4(a) has height 6, the tree in Figure 1.4(b) has height 4 and the one in Figure 1.4(c) has height 3. It turns out that there is no term equivalent to Sys that has a forest representation shallower than the one of Sys'' and therefore the depth of Sys (and of Sys' and Sys'') is 3. To prove Sys is bounded in depth by 3 we would need to show that any term reachable from it has depth at most 3.

Our technique to prove this depth bound on every reachable term relies on two ideas.

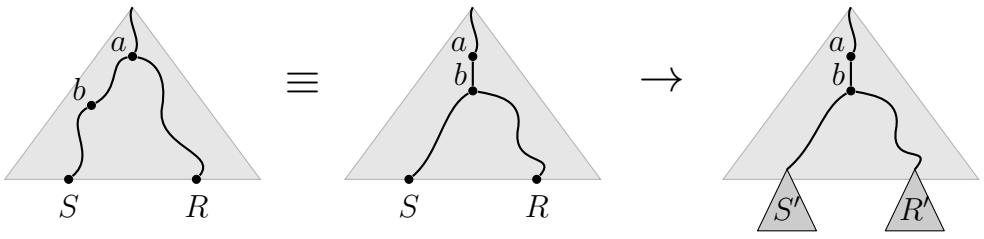
Idea 1: $\mathcal{T}$-compatibility The depth of a term is not a trivial concept: computing it requires reasoning about the whole structural congruence class of the term. We observe that different structural presentations of a term, like the three presentations of Sys in Figure 1.4, essentially differ in the order in which one considers restrictions. The tree in Figure 1.4(a) is the one constructed by considering the restrictions in the order $\nu c_1 < \nu c_2 < \nu c_3 < \nu s < \nu y$. By using the exchange and scope extrusion laws we can obtain the tree in Figure 1.4(b) which corresponds to the order $\nu s < \nu c_i$, for $1 \leq i \leq 3$ and $\nu c_3 < \nu y$. Now, proving a bound on depth is made hard by two factors: one needs to consider the set of reachable terms, and possibly reshuffle the structural presentation of each to obtain the bound. In the Sys example reachable terms all resemble Sys itself but will contain more copies of the subterms $\nu c.(\overline{s}\langle c\rangle \parallel c(x))$ and $\nu y.(\overline{c}\langle y\rangle \parallel c(x))$. It is clear that there is no use in considering the structural representations of these reachable terms that correspond to the orders on restrictions where all the instances of νc come before than s: the forests constructed that way will necessarily have paths $c_1, \cdots, c_n, s$ which are not helpful in proving the bound on depth.

The lesson we learn from this observation is that only some orderings of restrictions are sensible choices when trying to bound the depth of a term. In the example, the only sensible choice is $\nu s < \nu c$. In our view, the root cause of this phenomenon is the different *degree of sharing* of restrictions originating from νs and from νc. Namely, all the unbounded number of processes knowing the same instance of c will all know the same instance of s, but the processes knowing the same s need not know the same instance of c. If the term only generates a finite number of distinct names this is not an issue, but when the number of names is unbounded this has a huge effect on proving depth bounds.

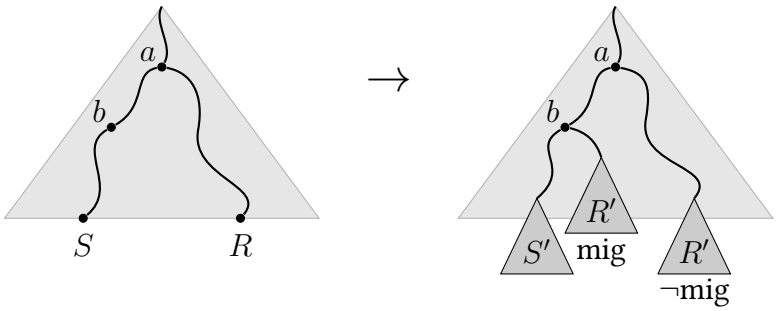
To formalise (static) assumptions on the informal concept of relative degree of sharing of restrictions, we introduce the notion of $\mathcal{T}$-compatibility. Restrictions νx are annotated with base types t_x, $\mathcal{T}$ is a forest of base types where we interpret the parent relation as the 'degree of sharing' order. A forest representation of a term is $\mathcal{T}$-compatible if every path $n_1 \dots n_k$ from its root is tagged with types $t_1 \dots t_k$ so that the order prescribed by $\mathcal{T}$ is strictly respected, i.e. every t_i is an ancestor of t_{i+1} in $\mathcal{T}$ and $t_i \neq t_{i+1}$. For example, Figure 1.4(d) shows a $\mathcal{T}$ with nodes $\{\mathsf{s}, \mathsf{c}, \mathsf{y}\}$, which encodes our observations on the degree of sharing of the restrictions in Sys. The tree in Figure 1.4(d) is the forest representation of Sys' when the restrictions are annotated with base types from $\mathcal{T}$ as shown. All the paths respect the order on base types: the parent relation of $\mathcal{T}$ is respected and no base type is repeated in any path. The forest is therefore $\mathcal{T}$-compatible.

Idea 2: $\mathcal{T}$-compatible migration The second idea concerns the use we want to make of $\mathcal{T}$-compatibility to prove depth boundedness. We observe that when $\mathcal{T}$ has finite height, invariance of $\mathcal{T}$-compatibility under reduction entails depth boundedness: if every reachable term is $\mathcal{T}$-compatible it always has a forest representation with height less or equal than the height of $\mathcal{T}$. The main enemy of invariance of $\mathcal{T}$-compatibility is mobility. During execution, a term may acquire knowledge of a name that makes it impossible to rearrange the forest representation so that $\mathcal{T}$ is respected. This is the case of terms unbounded in depth.

We identify a pattern in the communications that does not affect $\mathcal{T}$-compatibility. Suppose we are presented with a $\mathcal{T}$-compatible term P containing two processes $S = \overline{a}\langle b \rangle.S'$ and $R = a(x).R'$ ready to communicate over a channel a. After sending the message b, S continues as the process S' while upon reception of b, R binds x to b and continues as R' which now acquired knowledge of b. Schematically, the traditional understanding of this transaction is: first use extrusion to include R under the scope of b, and then make R and S react



Our view of the synchronisation instead is: upon communication, the sender continues in-place as S', while R' is split in two parts, one that uses the message (the *migrating* one) and one that does not. The migrating portion of R' is 'installed' under b so that it can make use of the acquired name, while the non migrating one can simply continue in-place. Pictorially:



Crucially, the reaction context stays constant. This means that if the starting term is $\mathcal{T}$-compatible, the context of the *reactum* is $\mathcal{T}$-compatible as well. We can then focus on imposing constraints on the use of names of R' so that the scopes of the migrating part are compatible with the place in which the term is 'transplanted'.

Invariance of $\mathcal{T}$-compatibility is ensured by the type system introduced in Chapter 10 by first realising that the type of the variable x above needs to agree with that of the message b that will be bound to it, and, second, requiring that the types of the names used by the migrating part of R' are compatible with $\mathcal{T}$ when put below b.

By using these ideas, our type system is able to statically accept π-calculus terms encoding systems like the server example presented in Section 1.1 and reject the ones unbounded in depth like the ring example of Section 1.3. The type system can be used to *check* that a given $\mathcal{T}$ is respected by the behaviour of a term but also to *infer* a suitable $\mathcal{T}$ when it exists. Once typability of a term is established, very rich properties, like the bound on the client's mailboxes of the server example, can be decided without losing precision on the identities of names.

Of course the type system is approximate in the sense that there are depth-bounded systems that cannot be typed. The fragment of terms that can be typed is however rather interesting as it includes terms which generate an unbounded number of names and exhibit mobility.

1.5 Outline

This dissertation is split into two parts. Part I details Soter's verification method for Erlang programs. After an overview of Erlang and of the kind of systems we are studying, in Chapter 4 we present λActor, a prototypical fragment of Erlang, and we explain and

motivate the choices behind the definition of ACS, our abstract models. In Chapter 5 we give operational semantics to λActor in a way that readily supports abstraction and proceed in defining a sound parametric abstract interpretation for λActor programs. We show some useful instantiations of the analysis and how we can use the analysis to build a sound ACS from a program. Chapter 6 presents Soter, our prototype implementation of the verification method. Soter takes as input an Erlang module and produces an ACS abstraction that can be automatically checked against user-supplied properties. The chapter concludes with some empirical results on some benchmarks and a case study on an Erlang program implementing Eratosthenes' sieve. In Chapter 7 we review related approaches.

Part II is dedicated to the definition and study of the notion of (typably) hierarchical systems. In Chapter 9 we overview the syntax and reduction semantics of the π-calculus and the concepts of depth and depth-boundedness. We introduce the main technical contributions of this part in Chapter 10: the notion of $\mathcal{T}$-compatibility and its relation with depth-boundedness and a type system to prove depth-boundedness through invariance of $\mathcal{T}$-compatibility. Related work is presented in Chapter 11 where we dedicate some attention to *Nested Data Class Memory Automata* which are a class of automata over infinite alphabet which can encode the behaviour of typably hierarchical systems when one is concerned with coverability. In Chapter 12 we discuss some possible extensions of the type system and a semantic notion of hierarchical systems which is independent from the type system.

In Chapter 13 we summarise the contributions of the two parts and outline some future directions.

Chapter 2

Preliminaries

In this chapter we briefly review some material from the literature that we will use in the rest of the dissertation.

Basic Notation

We denote the cardinality of a set A by $|A|$ and the powerset of A as $\mathscr{P}(A)$. Given two sets A and B we write $A \uplus B$ for their disjoint union. We write A^* for the set of finite sequences of elements of the set A, and ϵ for the null sequence. Let $a \in A$ and $l, l' \in A^*$, we overload '$\cdot$' so that it means insertion at the top $a \cdot l$, at the bottom $l \cdot a$ or concatenation $l \cdot l'$. We write l_i for the i-th element of l. The set of finite partial functions from A to B is denoted $A \rightharpoonup B$. We often write a finite partial function with the notation $f = [\, a_1 \mapsto b_1,\ \ldots,\ a_n \mapsto b_n \,]$ to indicate that $f(a_i) = b_i$. Given $f \colon A \rightharpoonup B$ we define $f[a \mapsto b] := (\lambda x.\ \textbf{if}\ (x{=}a)\ \textbf{then}\ b\ \textbf{else}\ f(x))$ and write $[\,]$ for the everywhere undefined function. To indicate that f is undefined on a we write $f(a) = \bot$; the domain of f is the set $\mathrm{dom}(f) = \{a \in A \mid f(a) \neq \bot\}$.

2.1 Transition Systems

A transition system is a tuple $(S, \to, s)$ where S is a set of configurations, $(\to) \subseteq (S \times S)$ is the transition relation and $s \in S$ is the initial state. We denote by $\to^*$ the transitive closure of $\to$. The *reachability problem* asks whether a state can be reached in a finite number of steps from the initial state.

Definition 2.1 (Reachability Problem)**.** Given a transition system $(S, \to, s)$, and a query state $t \in S$, determine if $s \to^* t$.

A relation $R \subseteq S_1 \times S_2$ between two transition systems $(S_1, \to_1, s_1)$ and $(S_2, \to_2, s_2)$, is a *simulation* if $s\ R\ t$ implies that for each $s' \in S_1$ such that $s \to_1 s'$ there is a

$t' \in S_2$ such that $t \to_2 t'$ and $s' \; R \; t'$. A relation R between two transition systems is a *bisimulation* if both R and R^{-1} are simulations. that is, if $s \; R \; t$ then:

(A) for each $s' \in S_1$ such that $s \to_1 s'$ there is a $t' \in S_2$ such that $t \to_2 t'$ and $s' \; R \; t'$;

(B) for each $t' \in S_2$ such that $t \to_2 t'$ there is a $s' \in S_1$ such that $s \to_1 s'$ and $s' \; R \; t'$.

Two transition systems $(S_1, \to_1, s_1)$ and $(S_2, \to_2, s_2)$ are said to be *bisimilar* if there exists a bisimulation between S_1 and S_2 relating s_1 and s_2.

A *preordered* transition system is a transition system $(S, \to, s)$ equipped with a preorder relation $\sqsubseteq$ on S. The *coverability problem* asks, given a state t, whether it is possible to reach a state greater than t.

Definition 2.2 (Coverability Problem)**.** Given a preodered transition system $(S, \to, s, \sqsubseteq)$ and a query state $t \in S$, determine if there exists $s' \in S$ such that $s \to^* s'$ and $t \sqsubseteq s'$.

The *coverability problem* can be used as an approximation of reachability: if $s \to^* s'$ then, by reflexivity of $\sqsubseteq$, s' is coverable.

2.2 Well Structured Transition Systems

Well Structured Transition Systems are a uniform framework that can be used to prove decidability results of models of computation.

Definition 2.3 (Well Structured Transition System [FS01; ACJT96])**.** A *well quasi ordering* (wqo) on a set X is a reflexive, transitive binary relation $(\sqsubseteq) \subseteq X \times X$ such that any infinite sequence of elements $x_0, x_1, x_2 \ldots$ from X contains a pair $x_i \sqsubseteq x_j$ with $i < j$.

A *Well Structured Transition System* (WSTS) is a transition system $(S, \to, s_0)$ equipped with a well quasi order $\sqsubseteq$ on states which is a simulation,[1] i.e. $s \to s'$ and $s \sqsubseteq t$ then there must exist t' such that $t \to t'$ and $s' \sqsubseteq t'$.

Given a preorder $(X, \sqsubseteq)$, the *upper-closure* of a set $Y \subseteq X$ is

$$\uparrow Y := \{x \mid x \in X, y \in Y, y \sqsubseteq x\};$$

Y is said to be *upward-closed* just if $Y = \uparrow Y$. A fundamental property of wqos is that any upward-closed set can be expressed as the upper-closure of a finite set of elements.

In a preordered transition system, the *pred-basis* of a state s, $\mathrm{pb}(s)$ is the set of minimal elements of the set of predecessors of $\uparrow s$, i.e. $\mathrm{pb}(s) := \min \{t \mid t \to s', s \sqsubseteq s'\}$. In a WSTS, by virtue of the wqo, the pred-basis is always finite, although not necessarily computable.

[1] This requirement can be weakened in general, see [FS01]

Theorem 2.1 (WSTS Coverability [FS01])**.** *The coverability problem is decidable for WSTS with 1) decidable wqo and 2) effective pred-basis.*

The algorithm to prove coverability for WSTS works by calculating the transitive closure of the pred-basis starting from the coverability query. The pred-basis is computable and ensured by the wqo to be a finite set; the wqo also guarantees that iterating the pred-basis eventually terminates with a fixpoint. Then coverability can be decided by checking whether the initial state is in the upper-closure of the fixpoint. This last check is decidable by decidability of the wqo.

2.3 Vector Addition Systems

One of the prototypical examples of WSTS is the *Petri net* model, also known as *Vector Addition Systems*.

Definition 2.4 (Vector Addition System)**.** A *Vector Addition System* (VAS) $\mathcal{V}$ 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*. Thus a rule is just a vector of integers of dimension $|I|$, whose components are indexed (i.e. named) by the elements of I.

The *state transition system* $[\![\mathcal{V}]\!]$ induced by a VAS $\mathcal{V} = (I, R)$ has state-set $\mathbb{N}^I$ and transition relation $\{(\mathbf{v}, \mathbf{v} + \mathbf{r}) \mid \mathbf{v} \in \mathbb{N}^I, \mathbf{r} \in R, \mathbf{v} + \mathbf{r} \in \mathbb{N}^I\}$. We write $\mathbf{v} \leq \mathbf{v}'$ just if for all i in I, $\mathbf{v}(i) \leq \mathbf{v}'(i)$.

The order $\leq$ on $\mathbb{N}^I$, for a finite I, is a wqo and the pred-basis can be shown to be decidable for VAS, so Vector Addition Systems are WSTS and have decidable coverability.

Theorem 2.2. *The coverability problem for VAS is decidable.*

VAS coverability has been known to be decidable since [KM69] and in fact the WSTS framework arose as a generalisation and systematisation of this and other similar results. VAS coverability has been proved to be Expspace-complete [Rac78]. VAS reachability is also known to be decidable although its complexity is open.

Part I

Soter: Infinite-State Verification for Erlang

Illustration: “Cada Homem É Uma Ilha”, courtesy of Vicente Sardinha © 1988.

Chapter 3

Overview

3.1 Contributions

This part of the thesis describes an approach for the automated verification of Erlang programs using a combination of static analysis and infinite-state model checking.

In devising a verification method for Erlang programs we face numerous challenges. The sequential core language is a dynamically typed, higher-order functional language with pattern-matching algebraic data types. These features alone pose serious challenges for automatic analysis tools and in fact many of the efforts in the Erlang analysis literature concentrate on taming the complexity of the sequential fragment. In addition, we aim at modelling and accurately analysing the concurrency model of Erlang, which is based on asynchronous message-passing. This adds two highly non-trivial features: unbounded dynamic spawning of processes and unbounded communication buffers (called mailbox, each associated with a process).

We focus on the core features of Erlang, which we formalise defining the λActor language. We give semantics to λActor in a form that supports the application of abstract interpretation and systematically derive a provably sound parametric control-flow analysis. The analysis is then used to bootstrap the construction of an abstract model of the semantics that we call *Actor Communicating System* (ACS). ACS are given semantics by means of *Vector Addition Systems* (VAS), also known as Petri nets, which have rich decidable properties. We exploit a class of properties called *coverability queries* to prove properties of Erlang programs such as unreachability of error states, mutual exclusion, or bounds on mailboxes. To assess the empirical effectiveness of the approach, we constructed Soter, a tool implementation of the verification method, thereby obtaining the first fully-automatic, infinite-state model checker for a core concurrent fragment of Erlang.

In the rest of this chapter, we put the problem domain in context with an overview of the approach to concurrency of Erlang and its relevance. In Chapter 4 we present the λActor language and analyse the consequences of each of its features on the problem of

automatic analysis. We motivate and formalise the design choices we make in response to the challenges listed above, and define our abstract model, ACS. Chapter 5 presents the concrete and abstract semantics of λActor in a parametric way and the soundness argument. After showing how the ACS model is extracted from the analysis and establishing the relation between the program's and the ACS' semantics, we discuss limitations and possible extensions of the analysis. The implementation is described in Chapter 6 and its practical effectiveness is demonstrated with empirical results on a set of benchmarks. The chapter also includes a tutorial-style case study showing how Soter can be used in practice to prove properties of Erlang modules. Related work is discussed in Chapter 7.

Chapters 5 and 6 present joint work with Jonathan Kochems and Luke Ong which has been published in [DKO12; DKO13b]. The contributions of the author are the design and definition of the concrete and abstract semantics, the ACS generation procedure and most of the benchmarks and examples. The bulk of the implementation has been programmed by Jonathan Kochems. The author implemented some modules for the simplifications of the ACS, the encoding of ACS into BFC models and the frontend as well as the web interface.

3.2 The Actor Model

A model of concurrency provides metaphors, concepts, formal structures and patterns for the representation, design and (algorithmic) analysis of the behaviour of systems containing a variable number of components which evolve simultaneously and interact with each other. Our world can be seen as inherently concurrent and, with such a general goal, a model of concurrency has the potential for being applied to—and explain—a wide variety of phenomena, computational or otherwise, which involves interaction among parties evolving in parallel.

We are concerned with computational concurrent systems. There are many technological innovations which are pushing towards the development of general models of concurrent computation, as opposed to sequential. Modern computer architectures are increasingly using parallel designs to improve performance; this approach has proven much more scalable and cost effective than just continuing to miniaturize single chips. At the same time, the ever increasing pervasive use of the Internet and distributed computing makes the research on accurate and effective models of concurrency urgent.

Concurrent software is hard to write. The plethora of synchronisation mechanisms that coexist in modern languages shows that there is no clear answer to the question: *how should we structure concurrent interaction?* The traditional "threads over shared state" model has a prominent role in the theory and practice of concurrency and many techniques for structuring and analysing concurrency have been developed for this paradigm. Unfortunately this approach is showing its shortcomings in terms of scalability, modularity and ease of formal reasoning [Lee06]. The advent of massive concurrency

through client-cloud computing has galvanized interest in alternative models; this is not a problem arising only in software systems but in hardware design too: the current trend of boosting hardware performance using multi-core designs and network-on-chip technology, is also pushing for a shift of paradigm.

Quite interestingly, the same shift happened in the theoretical developments leading to the invention of *process algebra* [Bae05]. As sharply noted by Baeten, these innovations were enabled by abandoning input/output semantics in favour of labelled transition systems, and overcoming the notion of global variables by representing exchange of information explicitly via message-passing.

In [HBS73], Hewitt and his collaborators proposed a new model with the precise goal of making concurrent programming (in the context of Artificial Intelligence) more modular and structured. Instead of adding concurrency primitives to a sequential (stateful) language, the *Actor Model* models concurrency as a fundamental concept, which gives rise to sequential computation as a special case of concurrent interaction [Hew77].

Actors are a metaphor for the unit of concurrency: this model views the world as a collection of independent opaque entities (the actors) which interact by asynchronously exchanging messages. Actors run in parallel and are opaque in the sense that they share no state; they can interact only through direct asynchronous message passing.

Each actor possesses an unbounded mailbox which stores the received messages waiting to be processed. In reaction to a message an actor can send messages, create new actors, update its internal state. An address is associated to each mailbox and an actor can send a message only to actors it knows the address of. An actor knows only a finite number of addresses at a given time and addresses can be sent as messages.

Communication is asynchronous: when an actor sends a message it can continue executing without waiting for the receiver to process the message; when an actor wants to receive a message it processes the first interesting message in the mailbox or, in the case it could not find any, it blocks, waiting for an interesting message to arrive.

One big advantage of this view is that the autonomy of actors frees the programmer from the burden of explicitly managing threads and synchronizing them; this autonomy also enables distribution and mobility: the whole system can be distributed among different interconnected machines and actors can be dynamically allocated and moved between different nodes; since the communication is transparent to the programmer, message-passing can be carried out by inter-process communication or over a network, at will.

After its initial proposal, this model has been further explored in many directions. In his PhD thesis [Agh85], Agha describes a programming language suitable for distributed computing based exclusively on the Actor Model, giving explicit syntax, semantics and an abstract machine. Further exploration of the semantics of this paradigm is presented in Clinger's PhD thesis [Cli81]. The Actor Model influenced many programming languages and concurrency formalism as we will briefly see in the next sections. For a more detailed

historical account see [Hew10, Appendix 2].

Actor-inspired programming languages

In the next section we will give an overview of Erlang, one of the most prominent languages directly based on the Actor Model. Apart from Erlang and experimental or domain specific languages, the Actor Model is often not supported natively by languages but implemented in libraries. Examples of such libraries include ActorFoundry among others for Java, Haskell-Actor for Haskell; similar libraries exist for C++, F# and Python.

Perhaps the most notable is Scala's Actor Library [HO09]. Scala [Oa04] is a multi-paradigm language built on top of the JVM and as such is fully interoperable with Java (all existing Java libraries are available in Scala). Scala is an attempt to merge typical features of a modern higher-order statically typed functional language with the object oriented paradigm, providing compilation to Java Byte Code. Other two models of concurrency are available to Scala in the form of two libraries: classical threads are provided by bindings to Java's Thread objects, and a CSP-inspired communication channels system by the Communicating Scala Objects library.

3.3 Erlang

One of the most successful and faithful implementations of the Actor Model is Erlang. Originally designed to program fault-tolerant distributed systems at Ericsson in the late 80s, Erlang is now a widely used, open-sourced language with support for higher-order functions, concurrency, communication, distribution, on-the-fly code upgrading, and multiple platforms support [AVW93; Arm10]. In the words of its father, the key observations behind Erlang's design are:

> The world is concurrent.
> Things in the world don't share data.
> Things communicate with messages.
> Things fail.
>
> – Joe Armstrong [CT09]

The first three sentences summarize the essence of the Actor Model; the last one puts the emphasis on the fault-tolerance features of Erlang systems.

The sequential part of Erlang is a higher order, dynamically typed, call-by-value functional language with pattern-matching algebraic data types. Following the Actor Model, a concurrent Erlang computation consists of a (dynamic) network of processes that communicate by message passing. The functional paradigm offers isolation and information hiding principles; processes take the role of actors, messages are immutable values. Every process has a unique identifier (which is data that may be sent as message),

and is equipped with an unbounded mailbox. Messages are sent asynchronously: the sender does not block awaiting receipt. Messages are retrieved from the mailbox, not FIFO, but by a pattern-matching mechanism. A process may block while waiting for a message that matches a certain pattern to arrive in its mailbox.

Syntax and features of Erlang

Erlang is *not* statically typed, although all types are checked dynamically to ensure type-safety at runtime. Due to this dynamic view of types, all the data structures are built using only atoms (literals), few built-in data-types as integers, and tuples (denoted by $\{x_1,\ldots,x_n\}$); the few other constructors, such as lists or records, are just syntactic sugar. A list of declarations of the form $f(pat_1,\ldots,pat_n) \to e$ defines an eager uncurried function f specified by cases (the non necessarily linear patterns pat_i). For example, the map function can be implemented in Erlang as following:

```
map(F, nil) → nil;
map(F, {X, Xs}) → {F(X), map(F, Xs)}.
```

Creation of new processes can be done using the `spawn` primitive which in essence takes an expression as argument, creates a new process evaluating that expression and returns to the caller a unique token, the *process identifier* (pid), associated to the new process. Pids are unique and cannot be inspected, forged nor guessed but can be stored in data structures and sent as messages. The `self` primitive returns the pid of the process calling it.

Communication primitives include the non-blocking send of a message `M` to a pid `P`, denoted by (`P ! M`), and the receive construct of the form

```
receive
  pat1 → e1;
  ...
  patn → en
end
```

While sends never block, receives work as follows: the first message in the process' mailbox is matched against each pattern in textual order; if no pattern is matched the second message in the mailbox is considered and so on; if no message in the mailbox matches any of the patterns, the process blocks waiting for more messages. When a message matches a pattern, the message is removed from the mailbox and the process continues with the branch corresponding to the matched pattern, binding the appropriate pattern variables.

A common Erlang pattern is the so-called *behaviours*: i.e. a module implementing a general purpose protocol, parametrised over another module implementing functions called *callbacks* to which the behaviour delegates control in the appropriate phases of the

protocol. Note that Erlang's dynamic module references can be simulated with higher-order parameters, which is general enough to express in full the dynamic module system of Erlang.

More advanced features of the language

Erlang is an extremely dynamic language. The interpreter does not perform any sanity check on the code; if something goes wrong during the execution, e.g. a function is applied to the wrong number of arguments or the type of a value does not match the one of the operation that is about to be applied to it, the runtime will simply throw an exception. Indeed, the philosophy of Erlang pragmatics is often summarized with the motto 'let it crash' [Arm10]: assuming the application undergoes extensive testing, only *heisenbugs* will possibly still be present; heisenbugs are those bugs which reveal themselves only in very rare interleavings of actions; since, by definition, these errors are very unprobable, restarting the faulty component would get rid of the problem for the current execution. Erlang's language support for this style of programming is made concrete by 'try-catch' exception handling, timeouts in receive instructions, and, more importantly, by the `monitor`/`link` primitives. The global architecture of a typical large scale Erlang program includes two logically separated parts: a set of processes implementing the application logic, and a set of processes forming the *supervision tree* (using `monitor`/`link`) with the specific role of continuously checking that each component of the system is running fine and react to faults with *local* interventions.

Erlang has a very dynamic module system which provides support for both some object-oriented like patterns and on-the-fly code replacement. Module names are regular values (atoms) and lookup of function definitions is dynamic.

Other language features include binary strings representation and matching, a mechanism for interfacing Erlang processes with processes running foreign language code, parametric modules, which increase reusability and registration of global name-pid associations, for easy access to global resources.

Implementation details

The success of Erlang as an industrial scale programming language has been made possible by the great efficiency of its implementation. In order to support the Actor Model and yet being of practical use, the runtime must scale really well with respect to the number of processes running in parallel and the overhead for message-passing communication. Erlang's runtime fulfills the first requirement by internally managing processes, instead of using threads at the system level, with an extremely minimal footprint. Creating new processes and sending and receiving messages are all very lightweight operations; due to isolation of processes and immutability of data, sends can be done both via copying or by reference as it is more convenient, and the garbage collection can dispose the memory

of short-lived processes very efficiently. The Erlang runtime can create a new process in less than a microsecond and run millions of processes simultaneously. The overhead for the bookeeping of process is less than a kilobyte per process. Message passing and context switching take only hundreds of nanoseconds. Message passing is transparent to the programmer so the runtime naturally supports a distributed deployment. The module system coupled with higher-order allows upgrading systems without taking them out of service. The result is a highly scalable and lightweight runtime that supports massive fault-tolerant distributed systems.

The compilation step takes Erlang code and translates it to a concise intermediate language called Core Erlang [Car01]; this intermediate representation gets analyzed and optimized, then it is translated to Erlang's virtual machine byte-code by the BEAM compiler or compiled to native code by the HiPE[1] compiler.

Industrial relevancy

The standard Erlang distribution is under active development. Many common general patterns are formalized and implemented as modules of the *Open Telecoms Platform* (OTP), a large set of libraries designed for building telecom systems bundled together with the standard Erlang distribution, providing a standardized way of performing the most common tasks needed to build a reliable system. Examples of the generic communication patterns offered by the OTP are the server-client, finite state machine, event handler, error management behaviours. The 2010 OTP system includes 49 subsystems including a real-time relational database (Mnesia), an H.248 stack implementation (Megaco), tools for building documentation (docbuilder), a debugging tool and a bug-finding tool (Dialyzer).

Many real-world applications are powered by Erlang. The AXD301, an asynchronous transfer mode switch developed by Ericsson, was written by a large programming team and is composed of more than 1.6 million lines of Erlang code. It proved that the Erlang (OTP) programming principles scale to realistic large system very well. Many other projects outside Ericcson are written in Erlang: Amazon's SimpleDB,[2] Yahoo! Harvester[3] and Facebook's Chat[4] are some notable examples.

Analysing such a rich language is a demanding engineering problem. We will focus on the core features of higher-order functional computation, dynamic process spawning and send/receive. The techniques used to analyse this core have all the ingredients that are needed to extend the approach to the full language.

[1] see https://www.it.uu.se/research/group/hipe
[2] see http://aws.amazon.com/simpledb
[3] see http://drdobbs.com/high-performance-computing/220600332
[4] see Eugene Letuchy, Erlang Factory Conference 2009 http://www.erlang-factory.com

Chapter 4

Models of Message-Passing Concurrency

We start this chapter with a formalisation of the fragment of Erlang we will focus on. Then we discuss the features of the language and the challenges they pose for automatic verification. Guided by this analysis we explore the design space for (decidable) abstract models and propose Actor Communicating Systems as a viable tradeoff between precision and decidability.

4.1 A Prototypical Fragment of Erlang

We now introduce λActor, a prototypical untyped functional language with actor concurrency. λActor is inspired by single-node *Core Erlang* [Car01]—the official intermediate representation of Erlang code—without built-in functions and fault-tolerance features. It exhibits in full the higher-order features of Erlang, with message-passing concurrency and dynamic process creation. The syntax of λActor is defined as follows:

$$
\begin{aligned}
e \in \mathit{Exp} ::= {} & x \mid \mathtt{c}(e_1, \ldots, e_n) \mid e_0(e_1, \ldots, e_n) \mid \mathit{fun} \\
& \mid \mathtt{letrec}\ f_1(x_1, \ldots, x_{k_1}) = e_1 . \cdots f_n(x_1, \ldots, x_{k_n}) = e_n .\ \mathtt{in}\ e \\
& \mid \mathtt{case}\ e\ \mathtt{of}\ \mathit{pat}_1 \to e_1; \ldots; \mathit{pat}_n \to e_n\ \mathtt{end} \\
& \mid \mathtt{receive}\ \mathit{pat}_1 \to e_1; \ldots; \mathit{pat}_n \to e_n\ \mathtt{end} \\
& \mid \mathtt{send}(e_1, e_2) \mid \mathtt{spawn}(e) \mid \mathtt{self}() \\
\mathit{fun} ::= {} & \mathtt{fun}(x_1, \ldots, x_n) \to e \\
\mathit{pat} ::= {} & x \mid \mathtt{c}(\mathit{pat}_1, \ldots, \mathit{pat}_n)
\end{aligned}
$$

where c ranges over a fixed finite set Σ of constructors.

Syntactic conventions For ease of comparison we keep the syntax close to Core Erlang. Variable names begin with an uppercase letter. We write '_' for an unnamed

unbound variable. Instead of writing n-tuples as a constructor application `tuple(`$e_1 \ldots e_n$`)` we use Erlang's syntax `{`$e_1 \ldots e_n$`}`. Sequencing $X = e_1$`,` e_2 is a shorthand for `(fun(X) ->`e_2`)(`e_1`)` and we we omit brackets for nullary constructors. The character '`%`' marks the start of a line of comment. We write $\mathrm{fv}(e)$ for the free variables of an expression and we define a λActor program $\mathcal{P}$ to be a closed λActor expression, i.e. $\mathrm{fv}(e) = \emptyset$.

Labels We associate a unique label ℓ to each sub-expression e of a program and indicate that e is labelled by ℓ by writing $\ell\colon e$. Take a term $\ell\colon (\ell_0\colon e_0(\ell_1\colon e_1, \ldots, \ell_n\colon e_n))$, we define $\ell.\mathrm{arg}_i := \ell_i$ and $\mathrm{arity}(\ell) := n$.

Reduction semantics

The fully formal specification of the operational semantics of λActor is given in Definition 5.1 and described in Section 5.1, but to illustrate λActor's concurrency model we sketch a small-step reduction semantics here. The rewrite rules for function application and λ-abstraction are identical to call-by-value λ-calculus; we write evaluation contexts as $E[\,]$. A state of the computation of a λActor program is a set Π of processes running in parallel. A process $\langle e \rangle^{\iota}_{\mathfrak{m}}$, identified by the pid ι, evaluates an expression e with mailbox $\mathfrak{m}$ holding unconsumed messages. Purely functional reductions performed by each process are independently interleaved. A `spawn` construct, `spawn(fun()->`e`)`, evaluates to a fresh pid ι' and creates a new process $\langle e \rangle^{\iota'}_{\epsilon}$, with pid ι':

$$\langle E[\texttt{spawn(fun()->}e\texttt{)}] \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \quad \longrightarrow \quad \langle E[\iota'] \rangle^{\iota}_{\mathfrak{m}} \parallel \langle e \rangle^{\iota'}_{\epsilon} \parallel \Pi$$

A `send` construct, `send(`ι, v`)`, evaluates to the message v with the side-effect of appending it to the mailbox of the receiver process ι; thus send is non-blocking:

$$\langle E[\texttt{send(}\iota, v\texttt{)}] \rangle^{\iota'}_{\mathfrak{m}'} \parallel \langle e \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \quad \longrightarrow \quad \langle E[v] \rangle^{\iota'}_{\mathfrak{m}'} \parallel \langle e \rangle^{\iota}_{\mathfrak{m} \cdot v} \parallel \Pi$$

The evaluation of a `receive` construct, `receive` p_1 `->`e_1`;` $\ldots$ p_n `->`e_n `end`, will block if the mailbox of the process in question contains no message that matches any of the patterns p_i. Otherwise, the first message m that matches a pattern, say p_i, is consumed by the process, and the computation continues with the evaluation of e_i. The pattern-matching variables in e_i are bound by a substitution θ to the corresponding matching subterms of the message m; if more than one pattern matches the message, then the first in textual order is fired

$$\langle E[\texttt{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \texttt{end}] \rangle^{\iota}_{\mathfrak{m} \cdot m \cdot \mathfrak{m}'} \parallel \Pi \quad \longrightarrow \quad \langle E[e_i\ \theta] \rangle^{\iota'}_{\mathfrak{m} \cdot \mathfrak{m}'} \parallel \Pi;$$

Note that mailboxes are *not First-In-First-Out* (FIFO) buffers, but rather *First-In-First-Fireable-Out* (FIFFO): incoming messages are queued at the end of the mailbox, and the message that a receive construct extracts is not necessarily the first.

```
letrec

%%% LOCKED RESOURCE BEHAVIOUR MODULE
res_start(Res) =
  spawn(fun() → res_free(Res)).
res_free(Res) =
  receive {lock, P} →
    send(P, {acquired, self()}),
    res_locked(Res, P)
  end.
res_locked(Res, P) =
  receive
    {req, P, Cmd} →
      case Res(P, Cmd) of
        {NewRes, ok} →
          res_locked(NewRes, P);
        {NewRes, {reply, A}} →
          send(P, {ans, self(), A}),
          res_locked(NewRes, P)
      end;
    {unlock, P} → res_free(Res)
  end.

% Locked Resource API
res_lock(Q) =
  send(Q, {lock, self()}),
  receive {acquired, Q} → ok end.
res_unlock(Q) =
  send(Q, {unlock, self()}).
res_request(Q, Cmd) =
  send(Q, {req, self(), Cmd}),
  receive {ans, Q, X} → X end.
res_do(Q, Cmd) =
  send(Q, {req, self(), Cmd}).


%%% CELL IMPLEMENTATION MODULE
cell_start() =
  res_start(cell(zero)).
cell(X) = fun(_P, Cmd) →
  case Cmd of
    {write, Y} → {cell(Y), ok};
    read → {cell(X), {reply, X}}
  end.

% Cell API
cell_lock(C) = res_lock(C).
cell_unlock(C) = res_unlock(C).
cell_read(C) = res_request(C, read).
cell_write(C, X) =
  res_do(C, {write, X}).

%%% INCREMENT CLIENT
inc(C) =
  cell_lock(C),
  cell_write(C, {succ, cell_read(C)}),
  cell_unlock(C).
add_to_cell(M, C) =
  case M of
    zero       → ok;
    {succ, M'} →
      spawn(fun() → inc(C)),
      add_to_cell(M', C)
  end.

%%% ENTRY POINT
in  C = cell_start(),
    add_to_cell(N, C).
```

Figure 4.1 – Locked Resource (running example).

Example 4.1 (Locked Resource). Figure 4.1 shows an example λActor program. The code has three logical parts, which would constitute three modules in Erlang:

1. the definition of a generic 'resource' *behaviour*[1] (lines 3–34),
2. an instantiation of the behaviour implementing a cell (lines 37–51),
3. a client making use of the cell's definitions and setting up a system (lines 53–68).

The first part defines the protocol that governs the lock-controlled, concurrent access of a shared resource by a number of clients. A resource is viewed as a generic server implementing the locking protocol, parametrised on a function that specifies how to react to requests. Note the use of higher-order arguments and return values. The function `res_start` creates a new process that runs an unlocked (`res_free`) instance of the resource. When unlocked, a resource waits for a `{lock, P}` message to arrive from a client `P`. Upon receipt of such a message, an acknowledgement message is sent back to the client and the control is yielded to `res_locked`. When locked (by a client `P`), a resource can accept requests `{req,P,Cmd}` from `P`—and from `P` only—for an unspecified command `Cmd` to be executed.

Note that the variable `P` in the *patterns* of `res_locked` is *bound*. An unusual feature of Erlang, it allows us in this example to match only messages sent by the pid currently bound to `P`; this feature corresponds, operationally, to equality checks on the values that would be bound to the bound variables by the match; this equality check is however performed *before* extracting the message from the queue so that, should the check fail, the message is skipped and the next message can be processed.

After running the requested command, the resource is expected to return the updated resource handler and an answer, which may be the atom `ok`, which requires no additional action, or a couple `{reply, Ans}` which signals that the answer `Ans` should be sent back to the client. Here we exploit higher-order features in two ways. First a resource is modelled as a functional argument that gets called when appropriate. Secondly this call itself yields a functional value that represents the updated resource. When an unlock message is received from `P` the control is given back to `res_free`. Note that the mailbox matching mechanism allows multiple locks and requests to be sent asynchronously to the mailbox of the locked resource without causing conflicts: the pattern matching in the locked state ensures that all the pending lock requests get delayed for later consumption once the resource is unlocked. The functions `res_lock`, `res_unlock`, `res_request`, `res_do` encapsulate the locking protocol, hiding it from the user who can then use this API as if it was purely functional.

The second part implements a simple shared resource that holds a natural number, which is encoded using the constructors `{succ, _}` and `zero`, and allows a client to read its value or overwrite it with a new one. Similarly as before, we provide the user with an

[1] see Section 3.3

API that hides the details of the protocol. Without lock messages, a shared resource with such a protocol easily leads to inconsistencies: it is a perfect application for our locking behaviour.

The last part defines the function `inc` which accesses a locked cell to increment its value. The function `add_to_cell` adds `M` to the contents of the cell by spawning `M` processes incrementing it concurrently. Finally the entry-point of the program sets up a process with a shared locked cell and then calls `add_to_cell`. Note that `N` is a free variable; to make the example a program we can either close it by setting `N` to a constant or make it range over all natural numbers with the extension described in Section 5.2.

An interesting correctness property of this code is mutual exclusion of the lock-protected region (i.e. line 56) of the concurrent instances of `inc`.

4.2 The Quest for a Decidable Abstraction

Concurrent programs are hard to write. They are just as hard to verify. In the case of Erlang programs, the inherent complexity of the verification task can be seen from several diverse sources of infinity in the state space.

(∞ 1) Data domains, and hence the message space, are unbounded: functions may return, and variables may be bound to, terms of an arbitrary size.

(∞ 2) General recursion requires a (process local) call-stack.

(∞ 3) Higher-order functions are first-class values; closures can be passed as parameters or returned.

(∞ 4) An unbounded number of processes can be spawned dynamically.

(∞ 5) Mailboxes have unbounded capacity.

Our goal is to verify safety properties of Erlang-like programs *automatically*, using a combination of static analysis and infinite-state model checking. The challenge is that one must reason about the asynchronous communication of an unbounded set of messages, across an unbounded set of Turing-powerful processes. Let us consider the items above and the constraints they force on us in the search for a suitable abstract model.

The main reason why these sources of unboundedness are difficult to handle algorithmically is that each of them easily supports encoding Turing-powerful models of computation, such as Minsky machines, i.e. finite automata equipped with two counters that can be increased, decreased and tested for zero.

In the presence of pattern-matching algebraic data types, the (sequential) functional fragment of λActor is already Turing powerful: counters can be easily encoded by using only two constructors, one for the successor and one for zero. Restricting it to a

pushdown (equivalently, first-order) fragment but allowing concurrent execution would enable, using very primitive synchronization, the simulation of a Turing-powerful finite automaton with two stacks. In [OR11], Ong and Ramsay propose a clean restriction on the use of pattern-matching that enables automatic verification of sequential higher-order functional programs. In their setting, a pattern is called weak if no variable bound by it can be used in the body of the function. Purely functional higher-order programs with arbitrary constructors are not Turing-powerful and can be verified against a rich class of properties. Unfortunately this restriction is not enough in the concurrent case. With only weak patterns, three processes, mailboxes with capacity 1 and either a) order-1 functions, finite data, or b) order-0 functions, constructors of arity 2, we can encode 2-counters machines. Thus the abstract model will need to represent items $(\infty\, 1)$ to $(\infty\, 3)$ very coarsely.

Handling an unbounded number of distinct pids is also problematic. Even when mailboxes have bounded capacity and processes are finite state machines, the ability to send pids in messages is fatal for decidability. We will elaborate on this in Section 9.5 and find ways to handle unboundedly many pids in Part II. For now, we will allow unbounded process creation but collapse the unbounded set of dynamic pids into a finite set of static process addresses called *pid-classes*. A pid-class is a bag (multiset) of pids which will be undistinguishable as addresses in the abstract model. Sending a message to a pid-class $\hat{\imath}$ corresponds to sending a message to *any* of the pids belonging to it.

FIFFO mailboxes are also highly expressive on their own. A single finite-control process equipped with a FIFFO or FIFO mailbox can encode a Turing-powerful queue automaton in the sense of Minsky. Unbounded queues are inherently more powerful than stacks: a single unbounded queue can encode the tape of a Turing machine. The literature considered a number of ways to weaken mailboxes so that some properties become decidable. For instance, if the message alphabet consists of only one message, a finite collection of finite-control processes can be simulated by Petri nets [AJ93]. If we impose a bound on channel capacities instead, we obtain a finite state system. A quite prolific weakening is *Lossy Channel Systems* [AJ93]: they postulate that mailboxes are lossy, that is, messages can be discarded before and after any transition, a feature that effectively captures the behaviour of unreliable channel communication. Not only do these systems allow to accurately and concisely model error-tolerant protocols but they also enjoy the decidability of some interesting verification problems such as (backward) control-state reachability. In our context however, since Erlang's semantics enforces some guarantees on the delivery of messages, the lossiness assumption would be of little use. Instead, we propose to weaken mailboxes using *counter abstraction*: a mailbox is simply a bag of messages in no particular order.

Thus constrained, we opt for a model of concurrent computation that has finite control, a finite number of messages, unordered mailboxes, and a finite number of *process classes*.

4.3 Actor Communicating Systems

Motivated by the previous analysis, we introduce *Actor Communicating System* (ACS), which models the interaction of an unbounded set of communicating processes. An ACS has a finite set of control states Q, a finite set of *pid-classes* P, a finite set of messages M, and a finite set of transition rules. ACS models are infinite state: the mailbox of a process has unbounded capacity, and the number of processes in an ACS may grow arbitrarily large. However the set of pid-classes is fixed, and processes of the same pid-class are not distinguishable.

Definition 4.1 (Actor Communicating System)**.** An *Actor Communicating System* (ACS) $\mathcal{A}$ is a tuple $\langle P, Q, M, R, \iota_0, q_0 \rangle$ 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 $\iota \colon q \xrightarrow{\lambda} q'$ where $\iota \in P$, $q, q' \in Q$ and λ is a label that can take one of four possible forms:

- τ, which represents an internal (sequential) transition of a process of pid-class ι
- $?m$ with $m \in M$: a process of pid-class ι extracts (and reads) a message m from its mailbox
- $\iota'!m$ with $\iota' \in P$, $m \in M$: a process of pid-class ι sends a message m to a process of pid-class ι'
- $\nu\iota'.\, q''$ with $\iota' \in P$ and $q'' \in Q$: a process of pid-class ι spawns a new process of pid-class ι' that starts executing from q''

We now give semantics to ACS. In light of the discussion in the previous section, we apply a *counter abstraction* on mailboxes and processes. Mailboxes disregard the ordering of messages, but track the number of occurrences of every message. For process spawining we count, for each control-state of each pid-class, the number of processes in that pid-class that are currently in that state.

It is important to make sure that such an abstraction contains all the behaviours of the semantics that uses FIFFO mailboxes: if there is a term in the mailbox that matches a pattern, then the corresponding branch is non-deterministically fired. To see the difference, take the ACS that has one process (named ι), three control states q, q_1 and q_2, and two rules $\iota \colon q \xrightarrow{?a} q_1$, $\iota \colon q \xrightarrow{?b} q_2$. When equipped with a FIFFO mailbox containing the sequence $c\,a\,b$, the process can only evolve from q to q_1 by consuming a from the mailbox, since it can skip c but will find a matching message (and thus not look further into the mailbox) before reaching the message b. In contrast, the VAS semantics would let q evolve non-deterministically to both q_1 and q_2, consuming a or b respectively: the mailbox is abstracted to $[\, a \mapsto 1,\ b \mapsto 1,\ c \mapsto 1 \,]$ with no information on whether a or

b arrived first. However, the abstracted semantics does contain the traces of the FIFFO semantics.

The semantics of an ACS is a state transition system where states have a finite number of counters (with values in $\mathbb{N}$) that support increment and decrement (when non-zero) operations. Such infinite-state systems are known as Vector Addition Systems (VAS), which are equivalent to Petri nets (see Definition 2.4).

Definition 4.2 (Semantics of ACS)**.** The semantics of an ACS $\mathcal{A} = (P, Q, M, R, \iota_0, q_0)$ is the transition system induced by the VAS $\mathcal{V} = (I, \mathbf{R})$ where $I = P \times (Q \uplus M)$ and $\mathbf{R} = \{\mathbf{r} \mid r \in R\}$). The transformation $r \mapsto \mathbf{r}$ is defined as follows.[2]

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

Given a $[\![\mathcal{V}]\!]$-state $\mathbf{v} \in \mathbb{N}^I$, the component $\mathbf{v}(\iota, q)$ counts the number of processes in the pid-class ι currently in state q, while the component $\mathbf{v}(\iota, m)$ is the sum of the number of occurrences of the message m in the mailboxes of the processes of the pid-class ι.

While infinite-state, many non-trivial properties are decidable on VAS including reachability, coverability and place boundedness; for more details see [FS01]. In this thesis we focus on *coverability*, which is Expspace-complete [Rac78]: given two states s and t, is it possible to reach from s a state t' that covers t (i.e. $t' \leq t$)?

Which kinds of correctness properties of λActor programs can one specify by coverability of an ACS? We will be using ACS to *over-approximate* the semantics of a λActor program, so if a state of the ACS is not coverable, then it is not reachable in any execution of the program. It follows that we can use coverability to express safety properties such as:

1. unreachability of error program locations, i.e. $\mathbf{v}(\iota, q_{\text{err}}) = 1$ is uncoverable;
2. mutual exclusion, i.e. $\mathbf{v}(\iota, q_{\text{critical}}) = 2$ is uncoverable;
3. boundedness of mailboxes: is it possible to reach a state where the mailbox of pid-class ι has more than k messages? I.e. states with $\sum_{m \in M} \mathbf{v}(\iota, m) = k$ are uncoverable.

[2] All unspecified components of the vectors $\mathbf{r}$ as defined in the table are set to zero.

Chapter 5

Verifying λActor programs

The goal of this chapter is defining an algorithm to extract a sound ACS description of an arbitrary λActor program. By sound we mean that the semantics of the ACS should be simulating the semantics of the program: there should exist an abstraction function from states of the program to states of the ACS such that if a state of the program is reachable, its abstraction can be reached in the VAS semantics of the ACS.

The procedure we propose is divided in two parts. First we tackle sources of infinity (∞ 1) to (∞ 3) with a parametric abstract interpretation resposible for extracting the salient control-flow information. In order to do this in a general way we define a store-based operational semantics from which a control-flow analysis can be derived systematically. Second, from the analysis we extract an ACS model that is shown to be a sound approximation of the input program.

To obtain a control-flow analysis for λActor we apply the methodology proposed by Might and Horn [MH11]. The key observation, explained lucidly in [Mig10], is the following. When the domain describing states of the operational semantics is not recursive, defining an abstraction often can be done by abstracting some basic domains and lifting the abstraction in a generic way on the domains constructed combining these basic ones. In the case of higher-order functional programs, the domain is recursive: a closure is a program expression coupled with an environment, and environments map variables to closures. A circular dependency as the above can be solved by introducing a level of indirection in one of the circular references. In the example, one can introduce a *store* mapping *addresses* to closures: environments map variables to addresses, thus removing the recursive structure in the domain definition. Figure 5.1 shows the operation pictorially: we go from a domain with cyclic dependencies (on the right) to an isomorphic acyclic one (on the left) by introducing a store.

In Section 5.1 we give an operational semantics to λActor, structured in a way that makes the semantic domain non-recursive and ready to be abstracted effectively. Then we proceed by formalising the abstraction in Section 5.2. The chapter concludes with the procedure to extract from the analysis an ACS, and the argument for its correctness.

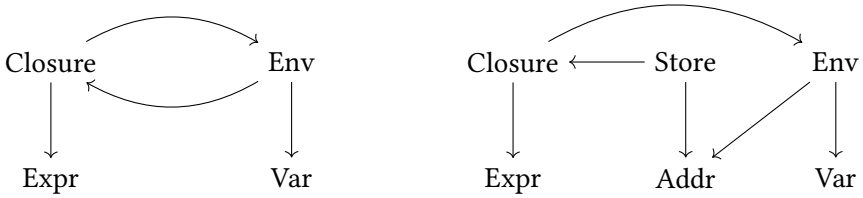


Figure 5.1 – Dependencies in the concrete semantics domain before and after introducing an indirection through a store.

5.1 An Operational Semantics for λActor

In this section, we define an operational semantics for λActor using a *time-stamped CESK* machine*,[1] following the methodology advocated by Van Horn and Might [VM10]. As explained in the introduction, such machines make use of *store-allocated continuations* to separate the recursive structure in its state space from the recursion in a programs's control/data flow. As we shall illustrate in Section 5.2, such a formalism is key to a *transparently sound* and *parametric* abstract interpretation.

A concrete machine semantics

Without loss of generality, we assume that in a λActor program, variables are distinct, and constructors and cases are only applied to (bound) variables. The λActor machine defines a transition system on *(global) states*, which are elements of the set $State$

$$\begin{aligned} s \in State &:= Procs \times Mailboxes \times Store \\ \pi \in Procs &:= Pid \rightharpoonup ProcState \\ \mu \in Mailboxes &:= Pid \rightharpoonup Mailbox \end{aligned}$$

An element of $Procs$ associates a process with its *(local) state*, and an element of $Mailboxes$ associates a process with its mailbox. We split the $Store$ into two partitions

$$\sigma \in Store := (VAddr \rightharpoonup Value) \times (KAddr \rightharpoonup Kont)$$

each with its address space, to separate *values* and *continuations*. By abuse of notation $\sigma(x)$ shall mean the application of the first component when $x \in VAddr$ and of the second when $x \in KAddr$.

The *local state* of a process is a tuple

$$q \in ProcState := (ProgLoc \uplus Pid) \times Env \times KAddr \times Time.$$

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

Its components are:

1. a pid, or a *program location*[2] which is a subterm of the program, labelled with its occurrence; whenever it is clear from the context, we shall omit the label;
2. an environment, which is a map from variables to pointers to values
$$\rho \in \mathit{Env} := \mathit{Var} \rightharpoonup \mathit{VAddr};$$
3. a pointer to a continuation, which indicates what to evaluate next when the current evaluation returns a value;
4. a time-stamp, which will be described later.

Values are either closures or pids:

$$d \in \mathit{Value} := \mathit{Closure} \uplus \mathit{Pid} \qquad \mathit{Closure} := \mathit{ProgLoc} \times \mathit{Env}$$

Note that, as defined, closures include both functions (which is standard) as well as constructor terms.

All the domains we define are naturally partially ordered: $\mathit{ProgLoc}$ and Var are discrete partial orders, all the others are defined by the appropriate pointwise extensions.

Mailbox and message passing

A *mailbox* is just a finite sequence of values: $\mathfrak{m} \in \mathit{Mailbox} := \mathit{Value}^*$. We denote the empty mailbox by ϵ. A mailbox is supported by two operations:

$$\begin{aligned} \mathrm{mmatch}\colon\ & \mathit{pat}^* \times \mathit{Mailbox} \times \mathit{Env} \times \mathit{Store} \to (\mathbb{N} \times (\mathit{Var} \rightharpoonup \mathit{Value}) \times \mathit{Mailbox})_\perp \\ \mathrm{enq}\colon\ & \mathit{Value} \times \mathit{Mailbox} \to \mathit{Mailbox} \end{aligned}$$

The function mmatch takes a list of patterns, a mailbox, the current environment and a store (for resolving pointers in the values stored in the mailbox) and returns the index of the matching pattern, a substitution witnessing the match, and the mailbox resulting from the extraction of the matched message. To model *Erlang-style* FIFFO mailboxes we set $\mathrm{enq}(d, \mathfrak{m}) := \mathfrak{m} \cdot d$ and define:

$$\mathrm{mmatch}(p_1 \ldots p_n, \mathfrak{m}, \rho, \sigma) := (i, \theta, \mathfrak{m}_1 \cdot \mathfrak{m}_2)$$

such that

$$\begin{array}{ll} \mathfrak{m} = \mathfrak{m}_1 \cdot d \cdot \mathfrak{m}_2 & \forall d' \in \mathfrak{m}_1.\, \forall j.\, \mathrm{match}_{\rho,\sigma}(p_j, d') = \perp \\ \theta = \mathrm{match}_{\rho,\sigma}(p_i, d) & \forall j < i.\, \mathrm{match}_{\rho,\sigma}(p_j, d) = \perp \end{array}$$

where $\mathrm{match}_{\rho,\sigma}(p, d)$ seeks to match the term d against the pattern p, following the pointers ρ to the store σ if necessary, and returning the witnessing substitution if matchable, and $\perp$ otherwise.

[2] Precisely a program location is a node in the abstract syntax tree of the program being analysed.

Evaluation contexts as continuations

Next we represent (in an inside-out manner) evaluation contexts as continuations. A continuation consists of a *tag* indicating the shape of the evaluation context, a pointer to a continuation representing the enclosing evaluation context, and, in some cases, a program location and an environment. Thus $\kappa \in \mathit{Kont}$ consists of the following constructs:

- Stop represents the empty context.
- $\mathsf{Arg}_i\langle \ell, v_0 \ldots v_{i-1}, \rho, a \rangle$ represents the context

$$E[v_0(v_1, \ldots, v_{i-1}, [\,], e'_{i+1}, \ldots, e'_n)]$$

  where $e_0(e_1, \ldots, e_n)$ is the subterm located at ℓ; ρ closes the terms $e_{i+1}, \ldots, e_n$ to $e'_{i+1}, \ldots, e'_n$ respectively; the address a points to the continuation representing the enclosing evaluation context E.

Addresses, pids and time-stamps

While the machine supports arbitrary concrete representations of time-stamps, addresses and pids, we present here an instance based on *contours* [Shi91] which shall serve as the reference semantics of λActor, and the basis for the abstraction of Section 5.2.

A way to represent a *dynamic* occurrence of a symbol is the history of the computation at the point of its creation. We record history as *contours* which are strings of program locations

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

The initial contour is just the empty sequence $t_0 := \epsilon$, while the tick function updates the contour of the process in question by prepending the current program location, which is always a function call (see rule **Apply**):

$$\begin{aligned} &\mathrm{tick}\colon \mathit{ProgLoc} \times \mathit{Time} \to \mathit{Time} \\ &\mathrm{tick}(\ell, t) := \ell \cdot t \end{aligned}$$

Addresses for values ($b \in \mathit{VAddr}$) are represented by tuples comprising the current pid, the variable in question, the bound value and the current time stamp. Addresses for continuations ($a, c \in \mathit{KAddr}$) are represented by tuples comprising the current pid, program location, environment and time (i.e. contour); or $*$ which is the address of the initial continuation (Stop).

$$\begin{aligned} \mathit{VAddr} &:= \mathit{Pid} \times \mathit{Var} \times \mathit{Data} \times \mathit{Time} \\ \mathit{KAddr} &:= (\mathit{Pid} \times \mathit{ProgLoc} \times \mathit{Env} \times \mathit{Time}) \uplus \{*\} \end{aligned}$$

The *data domain* ($\delta \in \mathit{Data}$) is the set of closed λActor terms; the function $\mathrm{res} : \mathit{Store} \times \mathit{Value} \to \mathit{Data}$ resolves all the pointers of a value through the store σ, returning the corresponding closed term:

$$\begin{aligned} \mathrm{res}(\sigma, \iota) &:= \iota \\ \mathrm{res}(\sigma, (e, \rho)) &:= e[x \mapsto \mathrm{res}(\sigma, \sigma(\rho(x))) \mid x \in \mathrm{fv}(e)] \end{aligned}$$

New addresses are allocated by extracting the relevant components from the context at that point:

$$\begin{aligned} &\mathrm{new}_{\mathrm{kpush}} : \mathit{Pid} \times \mathit{ProcState} \to \mathit{KAddr} \\ &\mathrm{new}_{\mathrm{kpush}}(\iota, \langle \ell, \rho, \_, t \rangle) := (\iota, \ell.\mathsf{arg}_0, \rho, t) \\ &\mathrm{new}_{\mathrm{kpop}} : \mathit{Pid} \times \mathit{Kont} \times \mathit{ProcState} \to \mathit{KAddr} \\ &\mathrm{new}_{\mathrm{kpop}}(\iota, \kappa, \langle \_, \_, \_, t \rangle) := (\iota, \ell.\mathsf{arg}_{i+1}, \rho, t) \\ &\qquad \text{where } \kappa = \mathsf{Arg}_i \langle \ell, \ldots, \rho, \_ \rangle \\ &\mathrm{new}_{\mathrm{va}} : \mathit{Pid} \times \mathit{Var} \times \mathit{Data} \times \mathit{ProcState} \to \mathit{VAddr} \\ &\mathrm{new}_{\mathrm{va}}(\iota, x, \delta, \langle \_, \_, \_, t \rangle) := (\iota, x, \delta, t) \end{aligned}$$

Remark 5.1. To enable data abstraction in our framework, the address of a value contains the data to which the variable is bound: by making appropriate use of the embedded information in the abstract semantics, we can fine-tune the data-sensitivity of our analysis, as we shall illustrate in Section 5.2. However when no data abstraction is needed, this data component can safely be discarded.

Following the same scheme, pids ($\iota \in \mathit{Pid}$) can be identified with the contour of the `spawn` that generated them: $\mathit{Pid} := (\mathit{ProgLoc} \times \mathit{Time})$. Thus the generation of a new pid is defined as

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

where tick^* is just the simple extension of tick that prepends a whole sequence to another. Note that the new pid contains the pid that created it as a sub-sequence: it is indeed part of its history (dynamic context). The pid $\iota_0 := (\ell_0, \epsilon)$ is the pid associated with the starting process, where ℓ_0 is just the root of the program.

Remark 5.2. Note that the only sources of infinity for the state space are time, mailboxes and the data component of value addresses. If these domains are finite then the state space is finite and hence reachability is decidable.

It is possible to present a more general version of the concrete machine semantics. We can reorganise the machine semantics so that components such as Time, Pid, $\mathit{Mailbox}$,

$KAddr$ and $VAddr$ are presented as *parameters* (which may be instantiated as the situation requires). In this thesis we present a contour-based machine, which is general enough to illustrate our method of verification.

Now that the state space is set up, we can define the operational semantics of λActor.

Definition 5.1 (Concrete Semantics of λActor)**.** The *concrete semantics* of λActor is the (non-deterministic) transition relation on states $(\rightarrow) \subseteq State \times State$ defined by the rules in Figure 5.2. The transition $s \rightarrow s'$ is defined by a case analysis of the shape of s. We present the rules for application, message passing and process creation; we omit the other rules (letrec, case and treatment of pids as returned value) since they follow the same shape.

The rules for the purely functional reductions are a simple lifting of the corresponding rules for the sequential CESK* machine: when the currently selected process is evaluating a variable, its address is looked up in the environment and the corresponding value is fetched from the store and returned, as dictated by rule **Vars**. Rule **Apply** is used when evaluating an application: control is given to each argument—including the function to be applied—in turn; Rules **FunEval** and **ArgEval** are then applied, collecting the values in the continuation. After all arguments have been evaluated, new values are recorded in the environment (and the store), and control is given to the body of the function to be applied. Rule **Receive** can only fire if mmatch returns a valid match from the mailbox of the process. In case there is a match, control is passed to the expression in the matching clause, and the substitution θ witnessing the match is used to generate the bindings for the variables of the pattern. When applying a send, according to rule **Send**, the recipient's pid is first extracted from the continuation, and enq is then called to dispatch the evaluated message to the designated mailbox. When applying a spawn with rule **Spawn**, the argument must be an evaluated nullary function; a new process with a fresh pid is then created whose code is the body of the function.

One can easily add rules for run-time errors such as wrong arity in function application, non-exhaustive patterns in cases, sending to a non-pid and spawning a non-function.

Functional reductions

ArgEval

$$\begin{array}{l}\text{if } \pi(\iota) = \langle v, \rho, a, t\rangle \\ \quad \sigma(a) = \kappa = \mathsf{Arg}_i\langle \ell, d_0 \ldots d_{i-1}, \rho', c\rangle \\ \quad\quad d_i := (v, \rho) \\ \quad\quad b := \mathrm{new}_{\mathrm{kpop}}(\iota, \kappa, \pi(\iota)) \\ \text{then } \pi' = \pi[\,\iota \mapsto \langle \ell.\mathsf{arg}_{i+1}, \rho', b, t\rangle\,] \\ \quad\quad \sigma' = \sigma[\,b \mapsto \mathsf{Arg}_{i+1}\langle \ell, d_0 \ldots d_i, \rho', c\rangle\,]\end{array}$$

Apply

$$\begin{array}{l}\text{if } \pi(\iota) = \langle v, \rho, a, t\rangle,\ \mathrm{arity}(\ell) = n \\ \quad \sigma(a) = \kappa = \mathsf{Arg}_n\langle \ell, d_0 \ldots d_{n-1}, \rho', c\rangle \\ \quad\quad d_0 = (\texttt{fun}(x_1 \ldots x_n) \to e, \rho_0) \quad d_n := (v, \rho) \\ \quad\quad b_i := \mathrm{new}_{\mathrm{va}}(\iota, x_i, \mathrm{res}(\sigma, d_i), \pi(\iota)) \\ \quad\quad t' := \mathrm{tick}(\ell, \pi(\iota)) \\ \text{then } \pi' = \pi[\,\iota \mapsto \langle e, \rho'[x_1 \to b_1 \ldots x_n \to b_n], c, t'\rangle\,] \\ \quad\quad \sigma' = \sigma[\,b_1 \mapsto d_1,\ \ldots,\ b_n \mapsto d_n\,]\end{array}$$

Communication

Receive

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \texttt{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \texttt{end}, \rho, a, t\rangle \\ \quad \mathrm{mmatch}(p_1 \ldots p_n, \mu(\iota), \rho, \sigma) = (i, \theta, \mathfrak{m}) \\ \quad \theta = [\,x_1 \mapsto d_1,\ \ldots,\ x_k \mapsto d_k\,] \\ \quad b_j := \mathrm{new}_{\mathrm{va}}(\iota, x_j, \mathrm{res}(\sigma, d_j), \pi(\iota)) \\ \quad \rho' := \rho[\,x_1 \mapsto b_1,\ \ldots,\ x_k \mapsto b_k\,] \\ \text{then } \pi' = \pi[\,\iota \mapsto \langle e_i, \rho', a, t\rangle\,] \\ \quad\quad \mu' = \mu[\,\iota \mapsto \mathfrak{m}\,] \\ \quad\quad \sigma' = \sigma[\,b_1 \mapsto d_1,\ \ldots,\ b_k \mapsto d_k\,]\end{array}$$

Send

$$\begin{array}{l}\text{if } \pi(\iota) = \langle v, \rho, a, t\rangle \\ \quad \sigma(a) = \kappa = \mathsf{Arg}_2\langle \ell, d, \iota', \_, c\rangle \\ \quad\quad d = (\texttt{send}, \_) \\ \text{then } \pi' = \pi[\,\iota \mapsto \langle v, \rho, c, t\rangle\,] \\ \quad\quad \mu' = \mu[\,\iota' \mapsto \mathrm{enq}((v, \rho), \mu(\iota'))\,]\end{array}$$

FunEval

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \ell\colon (e_0(e_1, \ldots, e_n)), \rho, a, t\rangle \\ \quad\quad b := \mathrm{new}_{\mathrm{kpush}}(\iota, \pi(\iota)) \\ \text{then } \pi' = \pi[\,\iota \mapsto \langle e_0, \rho, b, t\rangle\,] \\ \quad\quad \sigma' = \sigma[\,b \mapsto \mathsf{Arg}_0\langle \ell, \epsilon, \rho, a\rangle\,]\end{array}$$

Vars

$$\begin{array}{l}\text{if } \pi(\iota) = \langle x, \rho, a, t\rangle \\ \quad \sigma(\rho(x)) = (v, \rho') \\ \text{then } \pi' = \pi[\,\iota \mapsto \langle v, \rho', a, t\rangle\,]\end{array}$$

Initial state

Init The initial state associated with a program $\mathcal{P}$ is $s_{\mathcal{P}} := \langle \pi_0, \mu_0, \sigma_0\rangle$ where

$$\begin{array}{l}\pi_0 = [\,\iota_0 \mapsto \langle \mathcal{P}, [], *, t_0\rangle\,] \\ \mu_0 = [\,\iota_0 \mapsto \epsilon\,] \\ \sigma_0 = [\,* \mapsto \mathsf{Stop}\,]\end{array}$$

Process creation

Spawn

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \texttt{fun()} \to e, \rho, a, t\rangle \\ \quad \sigma(a) = \mathsf{Arg}_1\langle \ell, d, \rho', c\rangle \\ \quad\quad d = (\texttt{spawn}, \_) \\ \quad\quad \iota' := \mathrm{new}_{\mathrm{pid}}(\iota, \ell, \vartheta) \\ \text{then} \\ \quad \pi' = \pi\begin{bmatrix}\iota \mapsto \langle \iota', \rho', c, t\rangle, \\ \iota' \mapsto \langle e, \rho, *, t_0\rangle\end{bmatrix} \\ \quad \mu' = \mu[\,\iota' \mapsto \epsilon\,]\end{array}$$

Self

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \texttt{self()}, \rho, a, t\rangle \\ \text{then } \pi' = \pi[\,\iota \mapsto \langle \iota, \rho, a, t\rangle\,]\end{array}$$

Figure 5.2 – Operational Semantics Rules. The tables define the transition relation $s = \langle \pi, \mu, \sigma, \vartheta\rangle \to \langle \pi', \mu', \sigma', \vartheta'\rangle = s'$ by cases; the primed components of the state are identical to the non-primed components, unless indicated otherwise in the "then" part of the rule. The meta-variable v stands for terms that cannot be further rewritten such as λ-abstractions, constructor applications and un-applied primitives.

5.2 Parametric Abstract Interpretation

Now that the concrete operational semantics is setup with a non-recursive state space, we can proceed in defining an analysis based on it. In Remark 5.2 we identify Time, $\mathit{Mailbox}$ and Data as responsible for the unboundedness of the state space. Our strategy is abstracting these basic domains and lift these abstractions pointwise to the composed domains. Instead of fixing the abstractions of the basic domains, we leave them as parameters and state the conditions they must satisfy for guaranteeing soundness of the overall analysis.

Definition 5.2 (Basic domains abstraction)**.** A *basic domains abstraction* is a triple $\mathcal{I} = \langle \mathcal{D}, \mathcal{T}, \mathcal{M} \rangle$ consisting of a data, a time and a mailbox abstraction, defined as follows.

(i) A *data abstraction* is a triple $\mathcal{D} = \langle \widehat{\mathit{Data}}, \alpha_{\mathrm{d}}, \widehat{\mathrm{res}} \rangle$ where $\widehat{\mathit{Data}}$ is a flat (i.e. discretely ordered) domain of abstract data values and

$$\begin{aligned} \alpha_{\mathrm{d}} &\colon \mathit{Data} \to \widehat{\mathit{Data}} \\ \widehat{\mathrm{res}} &\colon \widehat{\mathit{Store}} \times \widehat{\mathit{Value}} \to \mathscr{P}(\widehat{\mathit{Data}}). \end{aligned}$$

and α_{d} is monotone.

(ii) A *time abstraction* is a tuple $\mathcal{T} = \langle \widehat{\mathit{Time}}, \alpha_{\mathrm{t}}, \widehat{\mathrm{tick}}, \hat{t}_0 \rangle$ where $\widehat{\mathit{Time}}$ is a flat domain of abstract contours, $\hat{t}_0 \in \widehat{\mathit{Time}}$, and

$$\begin{aligned} \alpha_{\mathrm{t}} &\colon \mathit{Time} \to \widehat{\mathit{Time}} \\ \widehat{\mathrm{tick}} &\colon \mathit{ProgLoc} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Time}}. \end{aligned}$$

and α_{t} is monotone.

(iii) A *mailbox abstraction* is a tuple $\mathcal{M} = \langle \widehat{\mathit{Mailbox}}, \leq_{\mathrm{m}}, \sqcup_{\mathrm{m}}, \alpha_{\mathrm{m}}, \widehat{\mathrm{enq}}, \hat{\epsilon}, \widehat{\mathrm{mmatch}} \rangle$ where $(\widehat{\mathit{Mailbox}}, \leq_{\mathrm{m}}, \sqcup_{\mathrm{m}})$ is a join-semilattice with least element $\hat{\epsilon} \in \widehat{\mathit{Mailbox}}$,

$$\begin{aligned} \alpha_{\mathrm{m}} &\colon \mathit{Mailbox} \to \widehat{\mathit{Mailbox}} \\ \widehat{\mathrm{enq}} &\colon \widehat{\mathit{Value}} \times \widehat{\mathit{Mailbox}} \to \widehat{\mathit{Mailbox}} \\ \widehat{\mathrm{mmatch}} &\colon \mathit{pat}^* \times \widehat{\mathit{Mailbox}} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \mathscr{P}(\mathbb{N} \times (\mathit{Var} \rightharpoonup \widehat{\mathit{Value}}) \times \widehat{\mathit{Mailbox}}) \end{aligned}$$

and α_{m} and $\widehat{\mathrm{enq}}$ are monotone on mailboxes.

An abstract interpretation of the basic domains determines an interpretation of the other abstract domains as follows.

$$\begin{aligned}
\widehat{State} &:= \widehat{Procs} \times \widehat{Mailboxes} \times \widehat{Store} \\
\widehat{Procs} &:= \widehat{Pid} \to \mathscr{P}(\widehat{ProcState}) \\
\widehat{ProcState} &:= (ProgLoc \uplus \widehat{Pid}) \times \widehat{Env} \times \widehat{KAddr} \times \widehat{Time} \\
\widehat{Store} &:= (\widehat{VAddr} \to \mathscr{P}(\widehat{Value})) \times (\widehat{KAddr} \to \mathscr{P}(\widehat{Kont})) \\
\widehat{Mailboxes} &:= \widehat{Pid} \to \widehat{Mailbox} \\
\widehat{Value} &:= \widehat{Closure} \uplus \widehat{Pid} \\
\widehat{Closure} &:= ProgLoc \times \widehat{Env} \\
\widehat{Env} &:= Var \rightharpoonup \widehat{VAddr} \\
\widehat{Pid} &:= (ProgLoc \times \widehat{Time}) \uplus \{\hat{\imath}_0\} \\
\hat{\imath}_0 &:= \hat{t}_0
\end{aligned}$$

each equipped with an abstraction function defined by an appropriate pointwise extension, the details of which can be found in Appendix A. We will call all of them α since it will not introduce ambiguities. To distinguish this abstraction from the one to the ACS semantics, we call the abstraction function on states $\alpha_{\text{cfa}} \colon State \to \widehat{State}$. The abstract domain $\widehat{Kont}$ is the pointwise abstraction of $Kont$, and we will use the same tags as those in the concrete domain. The abstract functions $\widehat{\text{new}_{\text{kpush}}}$, $\widehat{\text{new}_{\text{kpop}}}$, $\widehat{\text{new}_{\text{va}}}$ and $\widehat{\text{new}_{\text{pid}}}$, 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 $\widehat{C} = \widehat{A} \to \mathscr{P}(\widehat{B})$ is defined as

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

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

The operations on the parameter domains need to 'behave' with respect to the abstraction functions: the standard correctness conditions listed below must be satisfied by their instances. These conditions amount to requiring that what we get from an application of a concrete auxiliary function is adequately represented by the abstract result of the application of the abstract counterpart of that auxiliary function. The partial orders on the domains are standard pointwise extensions of partial orders of the parameter domains.

Definition 5.3 (Sound basic domains abstraction)**.** A basic domains abstraction $\mathcal{I}$ is *sound* just if the following conditions are met by the auxiliary operations:

$$\alpha_{\mathrm{t}}(\mathrm{tick}(\ell, t)) \leq \widehat{\mathrm{tick}}(\ell, \alpha_{\mathrm{t}}(t)) \tag{5.1}$$

$$\hat{\sigma} \leq \hat{\sigma}' \wedge \hat{d} \leq \hat{d}' \implies \widehat{\mathrm{res}}(\hat{\sigma}, \hat{d}) \leq \widehat{\mathrm{res}}(\hat{\sigma}', \hat{d}') \tag{5.2}$$

$$\forall \hat{\sigma} \geq \alpha(\sigma).\ \alpha_{\mathrm{d}}(\mathrm{res}(\sigma, d)) \in \widehat{\mathrm{res}}(\hat{\sigma}, \alpha(d)) \tag{5.3}$$

$$\alpha_{\mathrm{m}}(\mathrm{enq}(d, \mathfrak{m})) \leq \widehat{\mathrm{enq}}(\alpha(d), \alpha_{\mathrm{m}}(\mathfrak{m})) \qquad \alpha_{\mathrm{m}}(\epsilon) = \hat{\epsilon} \tag{5.4}$$

if $\mathrm{mmatch}(\vec{p}, \mathfrak{m}, \rho, \sigma) = (i, \theta, \mathfrak{m}')$ then

$$\forall \hat{\mathfrak{m}} \geq \alpha(\mathfrak{m}).\ \forall \hat{\sigma} \geq \alpha(\sigma).\ \exists \hat{\mathfrak{m}}' \geq \alpha(\mathfrak{m}').\ (i, \alpha(\theta), \hat{\mathfrak{m}}') \in \widehat{\mathrm{mmatch}}(\vec{p}, \hat{\mathfrak{m}}, \alpha(\rho), \hat{\sigma}) \tag{5.5}$$

Following the Abstract Interpretation framework, one can exploit the soundness constraints to derive, by algebraic manipulation, the definitions of the abstract auxiliary functions which would then be correct by construction [MJ08].

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

Definition 5.4 (Abstract Semantics of λActor)**.** The non-deterministic *abstract transition relation* on abstract states $(\rightsquigarrow) \subseteq \widehat{State} \times \widehat{State}$ is defined by the rules in Figure 5.3. These rules are the abstract counterparts of the rules for the operational semantics of Figure 5.2. When referring to a particular program $\mathcal{P}$, its *abstract semantics* is the portion of the graph reachable from $s_{\mathcal{P}}$.

The formal link between the concrete and abstract semantics is the following soundness result, which states that the abstract transitions simulate the concrete ones.

Theorem 5.1 (Soundness of Analysis)**.** *Given a sound abstraction of the basic domains, if* $s \to s'$ *and* $\alpha_{cfa}(s) \leq u$*, then there exists* $u' \in \widehat{State}$ *such that* $\alpha_{cfa}(s') \leq u'$ *and* $u \rightsquigarrow u'$*.*

One of the substantial advantages of using the Abstract Interpretation methodology is that the proofs of soundness follow now by construction. In fact the theorem above is proved by case analysis and unfoldings of definitions. We only give a brief sketch of the proof which can be found in full in Appendix A.

Proof (*Sketch*). The proof is by case analysis on the rule justifying a transition $s \to s'$. We show the case for rule **Receive** for illustration, the other cases follow the same scheme.

Functional abstract reductions

AbsArgEval

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \langle v, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\ \quad \hat{\sigma}(\hat{a}) \ni \hat{\kappa} = \mathsf{Arg}_i\langle \ell, \hat{d}_0 \ldots \hat{d}_{i-1}, \hat{\rho}', \hat{c}\,\rangle \\ \qquad \hat{d}_i := (v, \hat{\rho}) \quad \hat{b} := \widehat{\mathrm{new}_{\mathrm{kpop}}}(\hat{\imath}, \hat{\kappa}, \hat{q}\,) \\ \text{then } \hat{\pi}' = \hat{\pi} \sqcup [\,\hat{\imath} \mapsto \{\langle \ell.\mathsf{arg}_{i+1}, \hat{\rho}', \hat{b}, \hat{t}\,\rangle\}\,] \\ \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\,\hat{b} \mapsto \{\mathsf{Arg}_{i+1}\langle \ell, \hat{d}_0 \ldots \hat{d}_i, \hat{\rho}', \hat{c}\,\rangle\}\,]\end{array}$$

AbsApply

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \hat{q} = \langle v, \hat{\rho}, \hat{a}, \hat{t}\,\rangle,\ \mathrm{arity}(\ell) = n \\ \quad \hat{\sigma}(\hat{a}) \ni \mathsf{Arg}_n\langle \ell, \hat{d}_0 \ldots \hat{d}_{n-1}, \hat{\rho}', \hat{c}\rangle \\ \qquad \hat{d}_0 = (\texttt{fun}(x_1 \ldots x_n) \to e, \hat{\rho}_0) \quad \hat{d}_n := (v, \hat{\rho}) \\ \qquad \hat{\delta}_i \in \widehat{\mathrm{res}}(\hat{\sigma}, \hat{d}_i,) \quad \hat{b}_i := \widehat{\mathrm{new}_{\mathrm{va}}}(\hat{\imath}, x_i, \hat{\delta}_i, \hat{q}\,) \\ \quad \hat{\rho}'' := \hat{\rho}'[\,x_1 \mapsto \hat{b}_1,\ \ldots,\ x_n \mapsto \hat{b}_n\,] \\ \text{then } \hat{\pi}' = \hat{\pi} \sqcup [\,\hat{\imath} \mapsto \{\langle e, \hat{\rho}'', \hat{c}\,, \widehat{\mathrm{tick}}(l, \hat{t}\,)\rangle\}\,] \\ \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\,\hat{b}_1 \mapsto \{\hat{d}_1\},\ \ldots,\ \hat{b}_n \mapsto \{\hat{d}_n\}\,]\end{array}$$

Abstract communication

AbsReceive

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \hat{q} = \langle e, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\ \quad e = \texttt{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \texttt{end} \\ \quad \widehat{\mathrm{mmatch}}(p_1 \ldots p_n, \hat{\mu}(\hat{\imath}), \hat{\rho}, \hat{\sigma}) \ni (i, \hat{\theta}, \hat{\mathfrak{m}}) \\ \quad \hat{\theta} = [\,x_1 \mapsto \hat{d}_1,\ \ldots,\ x_k \mapsto \hat{d}_k\,] \\ \quad \hat{\delta}_j \in \widehat{\mathrm{res}}(\hat{\sigma}, \hat{d}_j) \quad \hat{b}_j := \widehat{\mathrm{new}_{\mathrm{va}}}(\hat{\imath}, x_j, \hat{\delta}_j, \hat{q}\,) \\ \quad \hat{\rho}' := \hat{\rho}[\,x_1 \mapsto \hat{b}_1,\ \ldots,\ x_k \mapsto \hat{b}_k\,] \\ \text{then } \hat{\pi}' = \hat{\pi} \sqcup [\,\hat{\imath} \mapsto \{\langle e_i, \hat{\rho}', \hat{a}, \hat{t}\,\rangle\}\,] \\ \qquad \hat{\mu}' = \hat{\mu}[\,\hat{\imath} \mapsto \hat{\mathfrak{m}}\,] \\ \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\,\hat{b}_1 \mapsto \{\hat{d}_1\},\ \ldots,\ \hat{b}_k \mapsto \{\hat{d}_k\}\,]\end{array}$$

AbsSend

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \langle v, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\ \quad \hat{\sigma}(\hat{a}) \ni \mathsf{Arg}_2\langle \ell, \hat{d}, \hat{\imath}', \_, \hat{c}\rangle \\ \qquad \hat{d} = (\texttt{send}, \_) \\ \text{then } \hat{\pi}' = \hat{\pi} \sqcup [\,\hat{\imath} \mapsto \{\langle v, \hat{\rho}, \hat{c}, \hat{t}\,\rangle\}\,] \\ \qquad \hat{\mu}' = \hat{\mu}[\,\hat{\imath}' \mapsto \widehat{\mathrm{enq}}((v, \hat{\rho}), \hat{\mu}(\hat{\imath}'))\,]\end{array}$$

AbsFunEval

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \hat{q} = \langle \ell\colon (e_0(e_1, \ldots, e_n)), \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\ \quad \hat{b} := \widehat{\mathrm{new}_{\mathrm{kpush}}}(\hat{\imath}, \hat{q}) \\ \text{then } \hat{\pi}' = \hat{\pi} \sqcup [\,\hat{\imath} \mapsto \{\langle e_0, \hat{\rho}, \hat{b}, \hat{t}\,\rangle\}\,] \\ \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\,\hat{b} \mapsto \{\mathsf{Arg}_0\langle \ell, \epsilon, \hat{\rho}, \hat{a}\rangle\}\,]\end{array}$$

AbsVars

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \langle x, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\ \quad \hat{\sigma}(\hat{\rho}(x)) \ni (v, \hat{\rho}') \\ \text{then } \hat{\pi}' = \hat{\pi} \sqcup [\,\hat{\imath} \mapsto \{\langle v, \hat{\rho}', \hat{a}, \hat{t}\,\rangle\}\,]\end{array}$$

Initial abstract state

AbsInit The initial state associated with a program $\mathcal{P}$ is

$$\hat{s}_{\mathcal{P}} := \alpha(s_{\mathcal{P}}) = \langle \hat{\pi}_0, \hat{\mu}_0, \hat{\sigma}_0\rangle$$

$$\begin{array}{l}\text{where } \hat{\pi}_0 = [\,\hat{\imath}_0 \mapsto \{\langle \mathcal{P}, [], *, \hat{t}_0\rangle\}\,] \\ \qquad \hat{\mu}_0 = [\,\hat{\imath}_0 \mapsto \hat{\epsilon}\,] \\ \qquad \hat{\sigma}_0 = [\,* \mapsto \{\mathsf{Stop}\}\,]\end{array}$$

Abstract process creation

AbsSpawn

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \langle \texttt{fun}() \to e, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\ \quad \hat{\sigma}(\hat{a}) \ni \mathsf{Arg}_1\langle \ell, \hat{d}, \hat{\rho}', \hat{c}\rangle \\ \qquad \hat{d} = (\texttt{spawn}, \_) \\ \qquad \hat{\imath}' := \widehat{\mathrm{new}_{\mathrm{pid}}}(\hat{\imath}, \ell, \hat{t}\,) \\ \text{then} \\ \qquad \hat{\pi}' = \hat{\pi} \sqcup \begin{bmatrix} \hat{\imath} \mapsto \{\langle \hat{\imath}', \hat{\rho}', \hat{c}, \hat{t}\,\rangle\} \\ \hat{\imath}' \mapsto \{\langle e, \hat{\rho}, *, \hat{t}_0\rangle\} \end{bmatrix} \\ \quad \hat{\mu}' = \hat{\mu} \sqcup [\,\hat{\imath}' \mapsto \hat{\epsilon}\,]\end{array}$$

AbsSelf

$$\begin{array}{l}\text{if } \hat{\pi}(\hat{\imath}) \ni \langle \texttt{self}(), \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\ \text{then } \hat{\pi}' = \hat{\pi} \sqcup [\,\hat{\imath} \mapsto \{\langle \hat{\imath}, \hat{\rho}, \hat{a}, \hat{t}\,\rangle\}\,]\end{array}$$

Figure 5.3 – Rules defining the Abstract Semantics. The tables describe the conditions under which a transition $\hat{s} = \langle \hat{\pi}, \hat{\mu}, \hat{\sigma}\rangle \rightsquigarrow \langle \hat{\pi}', \hat{\mu}', \hat{\sigma}'\rangle = \hat{s}'$ can fire; the primed versions of the components of the states are identical to the non-primed ones unless indicated otherwise in the "then" part of the corresponding rule. We write $\sqcup$ for the join operation of the appropriate domain.

Assume $s = \langle \pi, \mu, \sigma \rangle \to \langle \pi', \mu', \sigma' \rangle = s'$ is proved using rule **Receive**, and let $e = \texttt{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \texttt{end}$. Then

$$\begin{aligned} q &= \pi(\iota) = \langle e, \rho, a, t \rangle & q' &= \langle e_i, \rho', a, t \rangle \\ (i, \theta, \mathfrak{m}) &= \mathrm{mmatch}(p_1 \ldots p_n, \mu(\iota), \rho, \sigma) & \pi' &= \pi[\, \iota \mapsto q' \,] \\ \theta &= [\, x_1 \mapsto d_1,\ \ldots,\ x_k \mapsto d_k \,] & \rho' &= \rho[\, x_1 \mapsto b_1,\ \ldots,\ x_k \mapsto b_k \,] \\ b_j &= \mathrm{new}_{\mathrm{va}}(\iota, x_j, \delta_j, q) & \mu' &= \mu[\, \iota \mapsto \mathfrak{m} \,] \\ \delta_j &= \mathrm{res}(\sigma, d_j) & \sigma' &= \sigma[\, b_1 \mapsto d_1,\ \ldots,\ b_k \mapsto d_k \,]. \end{aligned}$$

Now let $u = \langle \hat{\pi}, \hat{\mu}, \hat{\sigma} \rangle$ such that $\alpha_{\mathrm{cfa}}(s) \leq u$, which implies $\alpha(\pi) \leq \hat{\pi}$, $\alpha(\mu) \leq \hat{\mu}$, and $\alpha(\sigma) \leq \hat{\sigma}$. We prove that there exists a u' such that $u \rightsquigarrow u'$ by application of rule **AbsReceive** and $\alpha_{\mathrm{cfa}}(s') \leq u'$. Since $\alpha(\pi) \leq \hat{\pi}$, we have $\hat{q} = \langle e, \hat{\rho}, \hat{a}, \hat{t} \,\rangle \in \hat{\pi}(\hat{\iota})$ where we write $\hat{\rho} := \alpha(\rho)$, $\hat{a} := \alpha(a)$, $\hat{t} = \alpha_{\mathrm{t}}(t)$ and $\hat{\iota} = \alpha_{\mathrm{pid}}(\iota)$. From $\alpha(\mu) \leq \hat{\mu}$ we know that $\alpha_{\mathrm{m}}(\mu(\iota)) \leq_{\mathrm{m}} \hat{\mu}(\hat{\iota})$. From the assumption of soundness of the mailbox abstraction by condition (5.5) and $\alpha(\sigma) \leq \hat{\sigma}$ we can infer

$$(i, \alpha(\theta), \hat{\mathfrak{m}}) \in \widehat{\mathrm{mmatch}}(\vec{p}, \hat{\mu}(\hat{\iota}), \hat{\rho}, \hat{\sigma})$$

with $\alpha_{\mathrm{m}}(\mathfrak{m}) \leq_{\mathrm{m}} \hat{\mathfrak{m}}$. The abstract substitution is

$$\alpha(\theta) = [\, x_1 \mapsto \alpha(d_1),\ \ldots,\ x_k \mapsto \alpha(d_k) \,]$$

Again by soundness of the basic domain abstraction we have $\hat{\delta}_j := \alpha_{\mathrm{d}}(\delta_j) \in \mathrm{res}(\hat{\sigma}, \alpha(d_j))$; to obtain new abstract variable addresses we can now set $\hat{b}_j := \widehat{\mathrm{new}_{\mathrm{va}}}(\hat{\iota}, x_j, \hat{\delta}_j, \hat{q}\,)$, for $j = 1, \ldots, k$. Rule **AbsReceive** is thus applicable to u and $u' = \langle \hat{\pi}', \hat{\mu}', \hat{\sigma}' \rangle$ yielding

$$\begin{aligned} \hat{\pi}' &:= \hat{\pi} \sqcup [\, \hat{\iota} \mapsto \hat{q}' \,] & \hat{q}' &:= \langle e_i, \hat{\rho}', \hat{a}, \hat{t} \,\rangle \\ \hat{\mu}' &:= \hat{\mu}[\, \hat{\iota} \mapsto \hat{\mathfrak{m}} \,] & \hat{\rho}' &:= \hat{\rho}[\, x_1 \mapsto \hat{b}_1,\ \ldots,\ x_k \mapsto \hat{b}_k \,] \\ \hat{\sigma}' &:= \hat{\sigma} \sqcup [\, \hat{b}_1 \mapsto \alpha(d_1),\ \ldots,\ \hat{b}_k \mapsto \alpha(d_k) \,] & \hat{u}' &:= \langle \hat{\pi}', \hat{\mu}', \hat{\sigma}', \hat{\vartheta} \rangle \end{aligned}$$

Now we can show that $\alpha_{\mathrm{cfa}}(s') \leq u'$ component by component:

i) $\alpha(\pi') \leq \hat{\pi} \leq \hat{\pi} \sqcup [\, \hat{\iota} \mapsto \hat{q}' \,] = \hat{\pi}'$

ii) by $\alpha_{\mathrm{m}}(\mathfrak{m}) \leq_{\mathrm{m}} \hat{\mathfrak{m}}$, $\alpha(\mu') = \alpha(\mu[\, \iota \mapsto \mathfrak{m} \,]) \leq \hat{\mu}[\, \hat{\iota} \mapsto \hat{\mathfrak{m}} \,] = \hat{\mu}'$

iii) $\alpha(\sigma') = \alpha(\sigma[\, b_1 \mapsto d_1, \ldots, b_k \mapsto d_k \,]) \leq \hat{\sigma} \sqcup [\, \hat{b}_1 \mapsto \alpha(d_1), \ldots, \hat{b}_k \mapsto \alpha(d_k) \,] = \hat{\sigma}'$

which completes the proof of this case. The other cases are similarly proved by straightforward unfolding of the definitions and use of the soundness of the basic domain abstraction assumption. □

Now that we have defined a sound abstract semantics we give sufficient conditions for its computability.

Theorem 5.2 (Decidability of Analysis)**.** *If a given (sound) abstraction of the basic domains is finite, then the derived abstract transition relation defined in Figure 5.3 is finite; it is also decidable, if the associated auxiliary operations (in Definition 5.3) are computable.*

Proof. The proof is by a simple inspection of the rules: all the individual rules are decidable and the state space is finite. □

Before showing how we can derive a sound ACS model from this analysis, we briefly present some useful possible instantiations of the abstract basic domains.

A simple mailbox abstraction

Abstract mailboxes need to be finite too in order for the analysis to be computable. By abstracting addresses (and data) to a finite set, values, and thus messages, become finite too. The only unbounded dimension of a mailbox becomes then the length of the sequence of messages. We then abstract mailboxes by losing information about the sequence and collecting all the incoming messages in an un-ordered set:

$$\mathcal{M}_{\text{set}} := \langle \mathscr{P}(\widehat{Value}), \subseteq, \cup, \alpha_{\text{set}}, \widehat{\text{enq}}_{\text{set}}, \emptyset, \widehat{\text{mmatch}}_{\text{set}} \rangle$$

where the abstract version of enq is the insertion in the set, as easily derived from the soundness requirement; the matching function is similarly derived from the correctness condition: writing $\vec{p} = p_1 \dots p_n$

$$\begin{aligned} \alpha_{\text{set}}(\mathfrak{m}) &:= \{\alpha(d) \mid \exists i.\ \mathfrak{m}_i = d\} \\ \widehat{\text{enq}}_{\text{set}}(\hat{d}, \hat{\mathfrak{m}}) &:= \{\hat{d}\} \cup \hat{\mathfrak{m}} \\ \widehat{\text{mmatch}}_{\text{set}}(\vec{p}, \hat{\mathfrak{m}}, \hat{\rho}, \hat{\sigma}) &:= \{(i, \hat{\theta}, \hat{\mathfrak{m}}) \mid \hat{d} \in \hat{\mathfrak{m}}, \hat{\theta} \in \hat{\text{match}}_{\hat{\rho},\hat{\sigma}}(p_i, \hat{d})\} \end{aligned}$$

We omit the straightforward proof that this constitutes a sound abstraction.

Abstracting data

Had we not included data in the value addresses in the definition of $VAddr$, cutting contours would have been sufficient to make this domain finite. A simple solution is to discard the value completely by using the trivial data abstraction $\widehat{Data}_0 := \{\dagger\}$ which is sound. If more precision is needed, any finite data-abstraction would do: the analysis would then be able to distinguish states that differ only because of different bindings in their frame.

We present here a data abstraction particularly well-suited to languages with algebraic data-types such as λActor: the abstraction $\lfloor e \rfloor_{\hat{\sigma},D}$ discards every sub-term of e that is nested at a deeper level than a parameter D.

$$\begin{aligned}
\lfloor (e, \hat{\rho}) \rfloor_{\hat{\sigma},0} &:= \{\dagger\} \\
\lfloor (\texttt{fun}\ldots, \hat{\rho}) \rfloor_{\hat{\sigma},D+1} &:= \{\dagger\} \\
\lfloor (\texttt{c}(x_1 \ldots x_n), \hat{\rho}) \rfloor_{\hat{\sigma},D+1} &:= \{\texttt{c}(\hat{\delta}_1 \ldots \hat{\delta}_n) \mid \hat{d}_i \in \hat{\sigma}(\hat{\rho}(x_i)), \hat{\delta}_i \in \lfloor \hat{d}_i \rfloor_{\hat{\sigma},D}\}
\end{aligned}$$

where † is a placeholder for discarded subterms.

An analogous D-deep abstraction can be easily defined for concrete values and we use the same notation for both; we use the notation $\lfloor \delta \rfloor_D$ for the analogous function on elements of $Data$.

We define $\mathcal{D}_D = \langle \widehat{Data}_D, \alpha_{\mathrm{D}}, \widehat{\mathrm{res}}_{\mathrm{D}} \rangle$ to be the 'depth-D' data abstraction where

$$\begin{aligned}
\widehat{Data}_{D+1} &:= \{\dagger\} \cup \{\texttt{c}(\hat{\delta}_1 \ldots \hat{\delta}_n) \mid \hat{\delta}_i \in \widehat{Data}_D\} \\
\alpha_{\mathrm{D}}(\delta) &:= \lfloor \delta \rfloor_D \\
\widehat{\mathrm{res}}_{\mathrm{D}}(\hat{\sigma}, \hat{d}\,) &:= \lfloor \hat{d} \rfloor_{\hat{\sigma},D}
\end{aligned}$$

The proof of its soundness is easy and we omit it.

Abstracting time

Let us now define a specific time abstraction that amounts to a concurrent version of a standard k-CFA. A k-CFA is an analysis parametric in k, which is able to distinguish dynamic contexts up to the bound given by k. We proceed as in standard k-CFA by truncating contours at length k to obtain their abstract counterparts:

$$\begin{aligned}
\widehat{Time}_k &:= \textstyle\bigcup_{0 \le i \le k} ProgLoc^i \\
\alpha^k_{\mathrm{t}}(\ell_1 \ldots \ell_k \cdot t) &:= \ell_1 \ldots \ell_k
\end{aligned}$$

Putting it all together

The simplest analysis we can then define is the one induced by the basic domains abstraction $\mathcal{I}_0 := \langle \widehat{Data}_0, \widehat{Time}_0, \widehat{Mailbox}_{\mathrm{set}} \rangle$. With this instantiation many of the domains collapse in to singletons. Implementing the analysis as it is would lead however to an exponential algorithm because it would record separate store and mailboxes for each abstract state. To get a better complexity bound, we apply a widening following the lines of [VM10, Section 7]: instead of keeping a separate store and separate mailboxes for each state we can join them keeping just a global copy of each (see Section 5.4). This reduces significantly the space we need to explore: the algorithm becomes polynomial time in the size of the program (which is reflected in the size of $ProgLoc$).

Considering other abstractions for the basic domains easily leads to exponential algorithms; in particular, the state-space grows linearly wrt the size of abstract data so the complexity of the analysis using $\widehat{Data}_D$ is exponential in D.

Dealing with open programs

Often it is useful to verify an open expression where its input is taken from a regular set of terms (see [OR11]). We can reproduce this in our setting by introducing a new primitive `choice` that non-deterministically calls one of its arguments. For instance, an interesting way of closing `N` in Example 4.1 would be by binding it to `any_num()`:

```
letrec
    ...
    any_num() = choice(
        fun() → zero,
        fun() → {succ, any_num()}
    ).
in C = cell_start(), add_to_cell(any_num(), C).
```

Now the uncoverability of the state where more than one instance of `inc` is running the protected section would prove that mutual exclusion is ensured for any number of concurrent copies of `inc`.

5.3 Generating the Actor Communicating System

The CFA algorithm we presented allows us to derive a sound 'flat' representation of the control-flow of the program. The analysis takes into account higher-order computation and (limited) information about synchronization. Now that we have this rough scheme of the possible transitions, we can 'guard' those transitions with actions which must take place in their correspondence; these guards, in the form of 'receive a message of this form' or 'send a message of this form' or 'spawn this process' cannot be modelled faithfully while retaining decidability of useful verification problems, as noted in Section 4.3. The best we can do, while remaining sound, is to relax the synchronization and process creation primitives with counting abstractions and use the guards to restrict the applicability of the transitions. In other words, these guarded (labelled) rules will form the definition of an ACS that simulates the semantics of the input λAcTor program.

First let us fix some terminology. We identify a common pattern of the rules in Figure 5.3. In each rule **R**, the premise distinguishes an abstract pid $\hat{\iota}$ and an abstract process state $\hat{q} = \langle e, \hat{\rho}, \hat{a}, \hat{t} \rangle$ associated with $\hat{\iota}$ i.e. $\hat{q} \in \hat{\pi}(\hat{\iota})$ and the conclusion of the rule associates a new abstract process state—call it $\hat{q}'$—with $\hat{\iota}$ i.e. $\hat{q}' \in \hat{\pi}'(\hat{\iota})$. Henceforth we shall refer to $(\hat{\iota}, \hat{q}, \hat{q}')$ as the *active components* of the rule **R**. We write $(\hat{\iota}, \hat{q}, \hat{q}') \models_{\mathbf{R}} \hat{s} \rightsquigarrow \hat{s}'$ if $\hat{s} \rightsquigarrow \hat{s}'$ is proved by rule **R** with active components $(\hat{\iota}, \hat{q}, \hat{q}')$.

Definition 5.5 (Generated ACS)**.** Given a λActor program $\mathcal{P}$, a sound basic domains abstraction $\mathcal{I} = \langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$ and $\mathcal{D}_{\text{msg}} = \langle \widehat{Msg}, \alpha_{\text{msg}}, \widehat{\text{res}}_{\text{msg}} \rangle$, a sound data abstraction for messages, the *Actor Communicating System generated by* $\mathcal{P}$, $\mathcal{I}$ and $\mathcal{D}_{\text{msg}}$ is defined as

$$\mathcal{A}_{\mathcal{P}} := \langle \widehat{Pid}, \widehat{ProcState}, \widehat{Msg}, R, \alpha(\iota_0), \alpha(\pi_0(\iota_0)) \rangle$$

where $s_{\mathcal{P}} = \langle \pi_0, \mu_0, \sigma_0, t_0 \rangle$ is the initial state (according to **Init**) with $\pi_0 = [\iota_0 \mapsto \langle \mathcal{P}, [], *, t_0 \rangle]$ and the rules in R are defined by cases as follows. If $(\hat{\iota}, \hat{q}, \hat{q}') \models_{\mathbf{R}} \hat{s} \rightsquigarrow \hat{s}'$, then:

1. If **R** is **AbsFunEval** or **AbsArgEval** or **AbsApply**, then

$$\hat{\iota} : \hat{q} \xrightarrow{\tau} \hat{q}' \in R \tag{AcsTau}$$

2. If **R** is **AbsReceive** where $\hat{d} = (p_i, \hat{\rho}')$ is the abstract message matched by $\widehat{\text{mmatch}}$ and $\hat{m} \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d})$, then

$$\hat{\iota} : \hat{q} \xrightarrow{?\hat{m}} \hat{q}' \in R \tag{AcsRec}$$

3. If **R** is **AbsSend** where $\hat{d}$ is the abstract value that is sent and $\hat{m} \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d})$, then

$$\hat{\iota} : \hat{q} \xrightarrow{\hat{\iota}'!\hat{m}} \hat{q}' \in R \tag{AcsSend}$$

4. If **R** is **AbsSpawn** where $\hat{\iota}'$ is the new abstract pid that is generated in the premise of the rule, which gets associated with the process state $\hat{q}'' = \langle e, \hat{\rho}, * \rangle$, then

$$\hat{\iota} : \hat{q} \xrightarrow{\nu\hat{\iota}'.\hat{q}''} \hat{q}' \in R \tag{AcsSp}$$

As we will make precise later, keeping $\widehat{Pid}$ and $\widehat{ProcState}$ small is of paramount importance for the model checking of the generated ACS to be feasible. This is the main reason why we keep the message abstraction independent from the data abstraction: this allows us to increase precision with respect to types of messages, which is computationally cheap, and keep the expensive precision on data as low as possible. Note that these two 'dimensions' are in fact independent and a more precise message space enhances the precision of the ACS even when using $\widehat{Data_0}$ as the data abstraction.

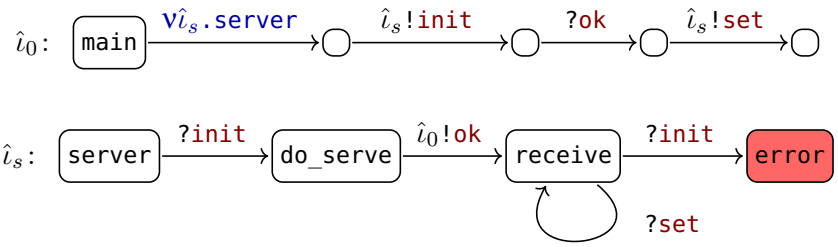
In our examples (and in our implementation) we use a $\widehat{Data_D}$ abstraction for messages where D is the maximum depth of the receive patterns of the program.

It is important to note that most of the decidable properties of the generated ACS are not even expressible on the CFA graph alone: being able to predicate on the contents of the counters means we can decide boundedness, mutual exclusion and many other expressive properties. The next example shows one simple way in which the generated ACS can be more precise than the bare CFA graph.

Example 5.3 (Generated ACS improving on CFA). Given the following program:

```
letrec
  server() =
    receive {init, P, X} →
      send(P, ok), do_serve(X)
    end.
  do_serve(X) =
    receive
        {init, _, _} → error;
        {set , Y}    → do_serve(Y);
        {get , P}    → send(P,X),
                       do_serve(X);
    end.
in  S = spawn(server),
    send(S, {init, self(), a}),
    receive ok → send(S, {set, b}) end.
```

our algorithm would output the following ACS starting from 'main':[3]



The error state is reachable in the CFA graph but not in its Parikh semantics: the token `init` is only sent once and never after `ok` is sent back to the main process. Once `init` has been consumed in the transition from 'server' to 'do_serve' the counter for it will remain set to zero forever.

Soundness

We now show that the semantics of the generated ACS is indeed a sound approximation of the semantics of λActor programs. To formalise the soundness argument we need to relate the states of the concrete operational semantics and the states of the generated ACS' semantics.

Definition 5.6 (Abstraction function)**.** The abstraction function

$$\alpha_{\text{acs}} \colon \mathit{State} \to (\widehat{\mathit{Pid}} \times (\widehat{\mathit{ProcState}} \uplus \widehat{\mathit{Msg}}) \to \mathbb{N})$$

[3]Labels are abbreviated to unclutter the picture; for example `{init,†,†}` is abbreviated with `init`.

relating concrete states and states of an ACS is defined as

$$\alpha_{\text{acs}}(\langle \pi, \mu, \sigma \rangle) := \begin{cases} (\hat{\iota}, \hat{q}) \mapsto \left| \{ \iota \mid \alpha(\iota) = \hat{\iota}, \alpha(\pi(\iota)) = \hat{q} \} \right| \\ (\hat{\iota}, \hat{m}) \mapsto \left| \{ (\iota, i) \mid \alpha(\iota) = \hat{\iota}, \alpha_{\text{msg}}(\text{res}(\sigma, \mu(\iota)_i)) = \hat{m} \} \right| \end{cases}$$

Theorem 5.3 (Soundness of generated ACS)**.** *For all choices of $\mathcal{I}$ and $\mathcal{D}_{\text{msg}}$, for all concrete states s and s', if $s \to s'$ and $\alpha_{acs}(s) \leq \mathbf{v}$ then there exists $\mathbf{v}'$ such that $\alpha_{acs}(s') \leq \mathbf{v}'$, and $\mathbf{v} \to_{\text{acs}} \mathbf{v}'$.*

The full proof of the theorem can be found in Appendix B, but we show the case of message reception here for illustration; the other cases are analogous.

Proof (*Sketch*)*.* As in Theorem 5.1, we proceed by case analysis on the rule justifying $s \to s'$. We show the argument for rule **Receive**.

By applying Theorem 5.1 to the abstract state $\hat{s} = \alpha_{\text{cfa}}(s)$ we have $\hat{s} \rightsquigarrow \hat{s}'$ using abstract rule **AbsReceive** with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota} = \alpha(\iota)$, $\hat{q} = \alpha(q)$ and $\hat{q}' = \alpha(q')$. Observe that $s = \langle \pi, \sigma, \mu \rangle$ and $\hat{s} = \langle \hat{\pi}, \hat{\sigma}, \hat{\mu} \rangle$ where $\hat{\pi} = \alpha(\pi)$, $\hat{\sigma} = \alpha(\sigma)$ and $\hat{\mu} = \alpha(\mu)$. Let the message matched by mmatch and extracted from $\mu(\iota)$ be $d = (p_i, \rho')$, then by soundness of the basic domain abstraction, Equation (5.5) ensures that a message $\hat{d} = (p_i, \hat{\rho}')$ can be matched in $\hat{\mu}(\hat{\iota})$ by $\widehat{\text{mmatch}}$, with $\hat{\rho}' = \alpha(\rho')$. We can therefore assume $\hat{d}$ is indeed the message matched during $\hat{s} \rightsquigarrow \hat{s}'$. Since the message abstraction is a sound data abstraction we know that $\hat{m} := \alpha_{\text{msg}}(\text{res}(\sigma, d)) \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d})$ and hence we have $\mathbf{r} := \hat{\iota} \colon \hat{q} \xrightarrow{?\hat{m}} \hat{q}' \in R$. Additionally we know

1. $\alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) \geq 1$,
2. $\alpha_{\text{acs}}(s)(\hat{\iota}, \hat{m}) \geq 1$,
3. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1$,
4. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1$ and
5. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{m}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{m}) - 1$

since d is the message extracted from $\mu(\iota)$ and $\hat{m} = \alpha_{\text{msg}}(\text{res}(\sigma, d))$. Now assume $\alpha_{\text{acs}}(s) \leq \mathbf{v}$, this implies $\mathbf{v}(\hat{\iota}, \hat{q}) \geq 1$ and $\mathbf{v}(\hat{\iota}, \hat{m}) \geq 1$ and so we can define

$$\mathbf{v}' := \mathbf{v} \begin{bmatrix} (\hat{\iota}, \hat{q}) \mapsto \mathbf{v}(\hat{\iota}, \hat{q}) - 1, \\ (\hat{\iota}, \hat{m}) \mapsto \mathbf{v}(\hat{\iota}, \hat{m}) - 1, \\ (\hat{\iota}, \hat{q}') \mapsto \mathbf{v}(\hat{\iota}, \hat{q}') + 1 \end{bmatrix}.$$

It is then clear that, using rule $\mathbf{r} \in R$, $\mathbf{v} \to_{\text{acs}} \mathbf{v}'$ and, from items 1–5 above, $\alpha_{\text{acs}}(s') \leq \mathbf{v}'$, as desired. □

Corollary 5.4. *Let $\mathcal{A}_{\mathcal{P}}$ be the ACS derived from a given λACTOR program $\mathcal{P}$. We have $[\![\mathcal{A}_{\mathcal{P}}]\!]$ simulates the semantics of $\mathcal{P}$: for each $\mathcal{P}$-run $s \to s_1 \to s_2 \to \ldots$, there exists a $[\![\mathcal{A}_{\mathcal{P}}]\!]$-run $\mathbf{v} \to_{\text{acs}} \mathbf{v}_1 \to_{\text{acs}} \mathbf{v}_2 \to_{\text{acs}} \ldots$ such that $\alpha_{acs}(s) = \mathbf{v}$ and for all i, $\alpha_{acs}(s_i) \leq \mathbf{v}_i$.*

Simulation preserves all paths so reachability (and coverability) is preserved.

Corollary 5.5. *If there is no $\mathbf{v} \geq \alpha_{acs}(s')$ such that $\alpha_{acs}(s) \to^*_{\text{acs}} \mathbf{v}$ then $s \not\to^* s'$.*

Example 5.4 (ACS Generated from Example 4.1). Consider Example 4.1 with the parametric entry point of Section 5.2, and the 0-CFA analysis of Section 5.2. A (simplified) pictorial representation of the ACS generated by our procedure is shown in Figure 5.4.

The three pid-classes correspond to the starting process $\hat{\iota}_0$ and the two static calls of spawn in the program, the one for the shared cell process $\hat{\iota}_{\text{c}}$ and the other, $\hat{\iota}_{\text{i}}$, for all the processes running `inc`.

The first component of the ACS, the starting one, just spawns a shared cell and an arbitrary number of concurrent copies of the third component; these actions increment the counter associated with states 'res_free' and 'inc_0'. The second component represents the intended protocol quite closely; note that by abstracting messages they essentially become tokens and the payload is ignored. The rules of the third component clearly show its sequential behaviour. The entry point is $(\hat{\iota}_0, \text{cell\_start})$.

The VAS semantics is accurate enough in this case to prove mutual exclusion of, say, state 'inc_2', which is protected by locks. Let's say for example that $n > 0$ processes of pid-class $\hat{\iota}_{\text{i}}$ reached state 'inc_1'; each of them sent a `lock` message to the cell; note that now the message does not contain the pid of the requester so all these messages are indistinguishable; moreover the order of arrival is lost, we just count them. Suppose that $\hat{\iota}_{\text{c}}$ is in state 'res_free'; since the counter for `lock` is n and hence not zero, the rule labeled with ?`lock` is enabled; however, once fired the counter for 'res_free' is zero and the rule is disabled. Now exactly one `ack` can be sent to the 'collective' mailbox of pid-class $\hat{\iota}_{\text{i}}$ so the rule receiving the ack is enabled; but as long as it is fired, the only `ack` message is consumed and no other $\hat{\iota}_{\text{i}}$ process can proceed. This holds until the lock is released and so on. Hence only one process at a time can be in state 'inc_2'. This property can be stated as a coverability problem: can $[\,(\hat{\iota}_{\text{i}}, \text{inc}_2) \mapsto 2\,]$ be covered? Since the ACS semantics is given in terms of a VAS, the property is decidable and the answer can be algorithmically calculated. As we saw the answer is negative and then, by soundness, we can infer it holds in the actual semantics of the input program too.

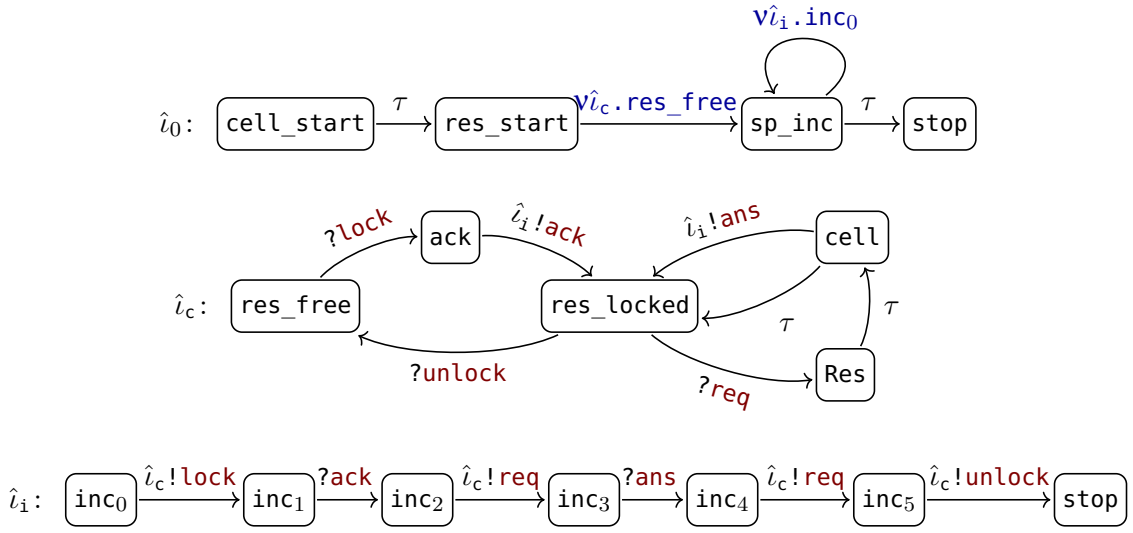


Figure 5.4 – ACS generated by the algorithm from Example 4.1.

5.4 An Efficient Algorithm for ACS Generation

The ACS generation procedure of Definition 5.5 is based on the analysis of Definition 5.4 so it is parametric in the basic domain abstraction. The complexity of the ACS generation depends on the choice of these parameters. However, even using the simplest instance with $\mathcal{I}_0$, the set of abstract states is exponential in the size of the input program. An algorithm enumerating the abstract states reachable from the initial one using $\rightsquigarrow$ would then be exponential time. Van Horn and Might [VM10] propose to use a widening to obtain a polynomial time 0-CFA algorithm for the exploration of reachable abstract states. We observe that if $\hat{s} \rightsquigarrow s'$ then $\hat{s} \leq \hat{s}'$; if two states $\hat{s}_1$ and $\hat{s}_1$ are successors of $\hat{s}$, i.e. $\hat{s} \rightsquigarrow s_1$ and $\hat{s} \rightsquigarrow s_2$, then $\hat{s}_1 \rightsquigarrow \hat{s}_1 \sqcup \hat{s}_2$ and $\hat{s}_2 \rightsquigarrow \hat{s}_1 \sqcup \hat{s}_2$. Therefore by joining the successors at each step of the exploration, we remain sound and do not introduce new spurious transitions. This procedure corresponds to keeping a global store, mailbox and pid to process state map, common to every reachable state and joining the corresponding components of new reachable states on the global ones at each step of the analysis until a fixpoint is reached.

The general algorithm for the exploration of the abstract transition system is shown in Figure 5.5. The procedure iterates over an increasing chain of abstract states, the maximum length of which is polynomial in the size of the program and the size of $\widehat{ProcState}$, $\widehat{Kont}$ and $\widehat{Pid}$. When using the $\mathcal{I}_0$ abstraction described at the end of Section 5.2, $\widehat{ProcState}$ and $\widehat{Kont}$ are polynomial and $\widehat{Pid}$ is linear in the size of the program. Assuming $|\mathcal{D}_{\text{msg}}|$ to be independent from the size of $\mathcal{P}$, such instantiation yields an overall

$\textsc{AcsGen}_{\mathcal{I},\mathcal{D}_{\text{msg}}}(\mathcal{P})$
  $\hat{s} := \bot$
  $\hat{s}' := \hat{s}_{\mathcal{P}}$ (from rule **AbsInit**)
  $R := \emptyset$
  while $\hat{s}' \neq \hat{s}$ **do**
    $\hat{s} := \hat{s}'$
    for each $\hat{s}''$ such that $(\hat{\iota}, \hat{q}, \hat{q}') \models_{\mathbf{R}} \hat{s}' \rightsquigarrow \hat{s}''$ **do**
      $\hat{s}' := \hat{s}' \sqcup \hat{s}''$
      $R := R \cup \{\hat{\iota} \colon \hat{q} \xrightarrow{\lambda} \hat{q}' \mid \lambda$ obtained using Definition 5.5$\}$
  return $(\hat{s}, R)$

Figure 5.5 – The ACS generation algorithm.

polynomial time procedure.

More precise instantiations for $\mathcal{I}$ easily lead to exponential algorithms. For instance, using the $\widehat{Data}_D$ abstraction the size of $\widehat{ProcState}$ grows exponentially in D.

Henceforth we call *dimension* of an ACS the dimension of the VAS underlying its VAS semantics. The complexity of coverability on VAS is Expspace in the dimension of the VAS; hence for the approach to be practical, it is critical to keep the dimension of the generated ACS small.

Our algorithm produces an ACS with dimension $(|\widehat{ProcState}| + |\widehat{Msg}|) \times |\widehat{Pid}|$. Assuming $|\widehat{Msg}|$ to be a constant, using the $\mathcal{I}_0$ basic domain abstraction ensures that the dimension of the generated ACS is polynomial in the size of the program, in the worst case. Due to the parametricity of the abstract interpretation we can adjust for the right levels of precision and speed. For example, if the property at hand is not sensitive to pids, one can choose a coarser pid abstraction.

It is also possible to greatly reduce the size of $\widehat{ProcState}$: we observe that many of the control states result from intermediate functional reductions; such reductions performed by different processes are independent, thanks to the actor model paradigm. This allows for the use of preorder reductions. In our prototype, as described in Section 6.1, we implemented a simple reduction that safely removes states which only represent internal functional transitions, irrelevant to the property at hand. This has proven to be a simple yet effective transformation yielding a significant speedup. We conjecture that, after the reduction, the cardinality of $\widehat{ProcState}$ is quadratic only in the number of `send`, `spawn` and `receive` of the program.

5.5 Limitations

As we saw in the examples, the ACS abstraction is quite powerful in that it allows to state and algorithmically verify very precise concurrency-relevant properties. As we discussed in Chapter 4, natural ways of increasing precision almost inevitably render the analysis undecidable. Although quite expressive, ACS models are however abstract, and thus there are programs and properties that, inescapably, cannot be proved using any of the presented abstractions.

A first cause of imprecision is non-trivial control-flow. This can be counteracted by increasing the context-sensitivity of the CFA, incurring in a considerable performance cost. On the positive side, many practical improvements of CFA have been recently proposed in the literature [EMH10; JLMH13], and they can be easily incorporated into the first phase of our analysis.

Related to control-flow approximation is the issue of representing higher-order call-stack behaviour. To appreciate this effect consider the program in Figure 5.6: it defines a higher-order combinator that spawns a number of identical workers, each applied to a different task in a list. It then waits for all the workers to return a result before collecting them in a list which is subsequently returned. The desired property is that the combinator only returns when every worker has sent back its result. Unfortunately to prove this property stack reasoning is required, which is beyond the capabilities of an ACS.

If call-stacks are problematic, faithful representations of mailboxes can be even harder to integrate in our framework. The current abstraction deliberately forgets the sequence of message arrival to attain decidability. Unfortunately, in some cases the communication protocol may rely on the exact sequence of arrival for correctness. Consider the program in Figure 5.7: it defines a simple function that discards a message in the mailbox and feeds the next to its functional argument and so on in a loop. Another process sends a 'bad argument' and a good one in alternation such that only the good ones are fed to the function.

A safety property of interest is that the function is never called with a bad argument. This cannot be proved because sequential information of the mailboxes, which is essential for the verification, is lost in the counter abstraction.

We observe however that the imprecision with respect to mailboxes is mitigated by the fact that, in practice, Erlang programmers often intentionally *avoid* relying on the exact sequence of arrival of messages. This happens for two reasons. First, the FIFFO mechanism of message matching assures that even if the mailbox contains messages in a certain order, it is the receiver that decides in which order to fetch them by only including certain patterns in the receives. To a certain extent, this information is used by our analysis. Second, in a distributed setting the message-delivery guarantees are weaker: if two messages are sent to an actor from two different remote nodes, their sequence in the receiver's mailbox cannot be assumed to reflect the order in which the messages were sent [see CS05].

```
letrec
  worker(Task) = ...

  spawn_wait(F, L) = spawn_wait'(F, fun()→[], L).

  spawn_wait'(F, G, L) =
    case L of
      []     → G();
      [T|Ts] →
        S = self(),
        C = spawn( fun() → send(S, {ans, self(), F(T) }) ),
        F' = fun() →
               receive
                 {ans, C, R} → [ R | G() ]
               end,
        spawn_wait'(F, F', Ts)
    end.

in spawn_wait(worker, [task1, task2, ...]).
```

Figure 5.6 – A program that Soter cannot verify because of the stack.

A final notable source of imprecision is the one introduced by pid-classes. The fact that pid-classes are required to be finite has a number of consequences. The most obvious effect is that the mailboxes of different processes get merged, adding imprecision to the loss of sequencing information. Secondly, the topology of the system is static. By topology of the system here we mean the graph that connects an actor with all the actors with pids known to it. In general, the number of actors, and hence of distinct pids, grows with time and since pids are values and can be exchanged as messages, actors can become connected or disconnected dynamically. This complexity is not expressible with pid-classes. Unlike the case of sequences in mailboxes, many idiomatic concurrency patterns make heavy use of both, the uniqueness of pids and the dynamic evolution of the topology. A paradigmatic example is the one of servers: when a client sends a request to a public server, the request message usually contains the pid of the client to allow the server to reply privately to the request. With the pid-classes abstraction we cannot model the fact that no client other than the requester will receive the answer, nor the fact that the server acquires (and then discards) knowledge about the pid of the client during the exchange.

To remedy to this limitation, in Part II we turn to a model of computation that can express these patterns accurately, the π-calculus. We will propose a way to characterise

```
1  letrec
2    no_a(X) = case X of
3      a → error;
4      b → ok
5    end.
6
7    send_b(P) = send(P, b), send_a(P).
8    send_a(P) = send(P, a), send_b(P).
9
10   stutter(F) = receive _ → unstut(F) end.
11   unstut(F)  = receive X → F(X), stutter(F) end.
12
13 in P = spawn(fun() → stutter(no_a)),
14    send_a(P).
```

Figure 5.7 – A program that Soter cannot verify because of the sequencing in mailboxes.

systems with a very precise representation of process identities, that support automatic model checking procedures.

Chapter 6

Soter: a Verification Tool for Erlang

In this chapter we present the tool we built to validate the approach presented in the previous chapters. We first give an overview of the implementation design, and then look into its performance and how it can be used by means of a case study. The impatient reader can jump to Section 6.2 for some benchmarks.

6.1 The Soter Tool

Soter is an experimental, prototype Haskell implementation of the analysis presented in the previous sections. It accepts a substantial (concurrent) subset of the Erlang language: supported features include algebraic data-types with pattern-matching, higher-order, spawning of new processes, asynchronous communication.

As presented in Figure 6.1, Soter's workflow has three phases. In phase 1, the input Erlang module with correctness annotations is compiled using the standard Erlang compiler `erlc` to a module of *Core Erlang*—the official intermediate representation of Erlang. The code is then normalised in preparation for the next phase: the code is transformed to a form that satisfies the assumptions made in Chapter 5. Correctness properties, expressible in various forms, are specified by annotating the program source. The user can insert assertions, or label program points and mailboxes and then state constraints they must satisfy.

For example, the resource-locking program in Figure 4.1 can be annotated by labelling the critical section of the client:

```
inc(C) =
  cell_lock(C),
  ?label(critical),
  cell_write(C, {succ, cell_read(C)}),
  cell_unlock(C).
```

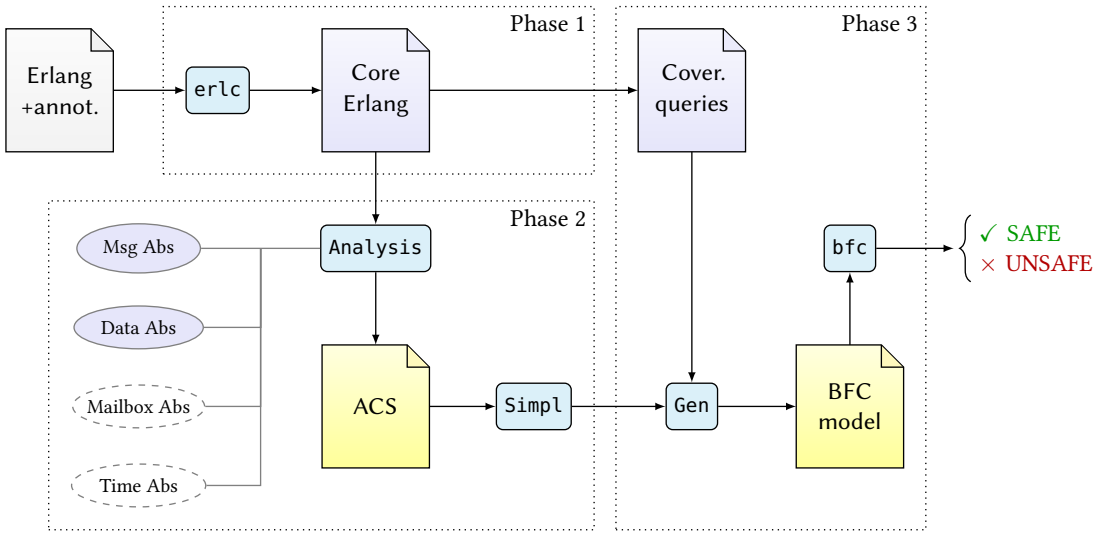


Figure 6.1 – Soter's workflow. The input annotated program goes through 3 phases. Phase 1 converts it to Core Erlang and extracts the properties to be verified. Phase 2 runs the CFA-style analysis using the parameter base abstract domains (only message and data abstractions can be currently tweaked). Phase 3 generates the BFC model from the simplified ACS and the properties. Then BFC is invoked to verify if the model is safe.

Then mutual exclusion can be expressed by the directive

```
-uncoverable("critical >= 2").
```

which requires that the number of processes running the line labelled with `critical` should never exceed 1.

The main purpose of phase 2 is to soundly generate the ACS rules. It implements the algorithm in Figure 5.5 with an optimisation for computing the fixpoint more efficiently (but with the same worst case complexity). The basic domain abstractions are set to the ones presented in Section 5.2: $\mathcal{D}_D$, $\mathcal{T}_0$, $\mathcal{M}_{\text{set}}$ and $\mathcal{D}_M$ as messages abstraction, where D and M are parameters controlling the depth of the data and message abstraction respectively. By default D is set to zero, so calls to the same function with different arguments are merged in the abstract model; the runtime of the analysis is exponential in D. M is by default set to $D+P$ where P is the maximum depth of the receive patterns of the program; using large values for M does not incur the same slowdown as adjusting D. In future releases, we plan to introduce parameters to tune the precision of the abstraction so that users can control the context sensitivity of the analysis.

In phase 3, Soter generates a VAS in the format of BFC [KKW12], which is a fast coverability checker for VAS with transfer arcs, developed by Alexander Kaiser. For each property Soter needs to prove, BFC is called internally with the BFC model and a query representing the safety property.

Soter can be run in three modes: 'analysis only' which produces the ACS, skipping phase 3; 'verify assertions' which extracts the properties from user annotations; 'verify absence-of-errors' which generates BFC queries asserting the absence of runtime exceptions. Currently, not all the exceptions that the Erlang runtime can throw are represented; the supported ones include sending a message to a non-pid value, applying a function with the wrong arity, and spawning a non-functional value. A notable omission is pattern-matching failures.

Phases 1 and 2 are polytime in the size of the input program. Despite the EXPSPACE-completeness of the Petri net coverability problem, phase 3 is surprisingly efficient; see the outcome of the experiments in Table 6.1.

Web interface

In addition to a command-line interface, we have built a web interface for Soter available at `http://mjolnir.cs.ox.ac.uk/soter/`. The user interface allows easy input of Erlang programs and allows the user to try Soter directly from the browser, without having to install any additional software. A library of annotated example programs is available to be tried and modified. Soter presents the generated abstract model as a labelled graph for easy visualisation, and reports in detail on the performance and results of the verification.

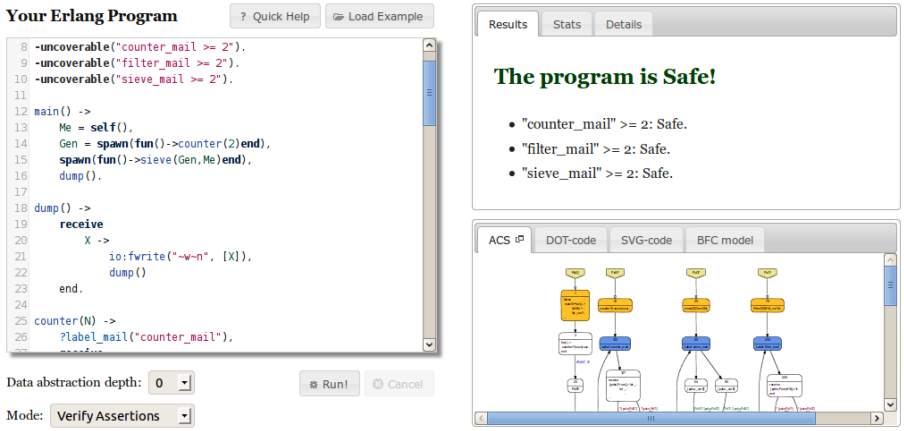


Figure 6.2 – Soter's web interface.

6.2 Empirical Evaluation

In Table 6.1 we summarise our experimental results. Many of the examples are higher-order and use dynamic (and unbounded) process creation and non-trivial synchronization. Example 4.1 appears as reslock and Soter proves mutual exclusion of the clients' critical

section. concdb is the example program of [Huc99] for which we prove mutual exclusion. pipe is inspired by the 'pipe' example of [KNY95]; the property proved here is boundedness of mailboxes. sieve is a dynamically spawning higher-order concurrent implementation of the Erathostene's sieve which we will present in detail in the next section;Soter can prove all the mailboxes are bounded.

Example	LOC	P	SAFE?	ABS		ACS SIZE		TIME		
				D	M	Places	Ratio	Analysis	BFC	Total
reslock	356	1	yes	0	2	40	10%	0.56	0.82	1.48
sieve	230	3	yes	0	2	47	19%	0.26	2.46	2.76
concdb	321	1	yes	0	2	67	12%	1.10	5.19	6.46
state_factory	295	2	yes	0	1	22	4%	0.59	0.02	0.75
pipe	173	1	yes	0	0	18	8%	0.15	0.00	0.18
ring	211	1	yes	0	2	36	9%	0.55	0.25	0.88
parikh	101	1	yes	0	2	42	41%	0.05	0.07	0.13
unsafe_send	49	1	no	0	1	10	38%	0.02	0.00	0.02
safe_send	82	1	no*	0	1	33	36%	0.05	0.00	0.06
safe_send	82	4	yes	1	2	82	34%	0.23	0.06	0.32
firewall	236	1	no*	0	2	35	10%	0.36	0.02	0.44
firewall	236	1	yes	1	3	74	10%	2.38	0.00	2.69
finite_leader	555	1	no*	0	2	56	20%	0.35	0.01	0.40
finite_leader	555	1	yes	1	3	97	23%	0.75	0.86	1.70
stutter	115	1	no*	0	0	15	19%	0.04	0.00	0.05
howait	187	1	no*	0	2	29	14%	0.19	0.00	0.22

Table 6.1 – Soter Benchmarks. The number of lines of code refers to the compiled Core Erlang. The **P** column indicates the number of properties which need to be proved. The columns **D** and **M** indicate the data and message abstraction depth respectively. In the **SAFE?** column, "no*" means that the program satisfies the properties but the verification was inconclusive; "no" means that the program is not safe and Soter finds a genuine counterexample. **Places** is the number of places of the underlying Petri net after the simplification; **Ratio** is the ratio of the number of places of the generated Petri net before and after the simplification. All times are in seconds.

All example programs, annotated with coverability queries, can be viewed and verified using Soter at `http://mjolnir.cs.ox.ac.uk/soter/`.

6.3 A Simple Case Study

We illustrate the workings of Soter by an example. Figure 6.3 shows an implementation of Eratosthenes' sieve by Rob Pike.[1] We use here Erlang's syntax.

[1] see "Concurrency and message passing in Newsqueak", `http://youtu.be/hB05UFqOtFA`

```
1  main() →
2    Gen = spawn( fun() →
3             counter(2) end ),
4    spawn( fun() →
5      sieve(Gen,self()) end ),
6    dump().
7
8  dump() →
9    receive X →
10     io:write(X), dump()
11   end.
12
13 counter(N) →
14   ?label_mail("counter_mail"),
15   receive {poke, From} →
16     From ! {ans, N},
17     counter(N+1)
18   end.
19
20 sieve(In, Out) →
21   ?label_mail("sieve_mail"),
22   In ! {poke, self()},
23   receive {ans,X} →
24     Out ! X,
25     F = spawn(fun()→
26       filter(divisible_by(X), In)
27     end),
28     sieve(F,Out)
29   end.
30
31 filter(Test, In) →
32   ?label_mail("filter_mail"),
33   receive {poke, From} →
34     filter(Test, In, From)
35   end.
36
37 filter(Test, In, Out) →
38   In ! {poke, self()},
39   receive {ans,Y} →
40     case Test(Y) of
41       false→ Out ! {ans,Y},
42              filter(Test, In);
43       true → filter(Test, In, Out)
44     end
45   end.
46
47 -ifdef(SOTER).
48   divisible_by(X) →
49     fun(Y) → ?any_bool() end.
50 -else.
51   divisible_by(X) →
52     fun(Y) → case Y rem X of
53                0 → true;
54                _ → false
55              end
56     end.
57 -endif.
```

Figure 6.3 – Eratosthenes' Sieve, actor style.

The actor defined by `counter` provides the sequence of natural numbers as responses to `poke` messages, starting from 2; the `dump` actor prints everything it receives. The `sieve` actor's goal is to send all prime numbers in sequence as messages to the `dump` actor; to do so it pokes its current `In` actor waiting for a prime number. After forwarding the received prime number, it creates (`spawn`) a new `filter` process, which becomes its new `In` actor. The filter actor, when poked, queries its `In` actor until a number satisfying `Test` is received and then it forwards it; the test (an higher-order parameter) is initialized by `sieve` to be a divisibility check that tests if the received number is divisible by the last prime produced. The overall effect is a growing chain of `filter` actors each filtering multiples of the primes produced so far; at one end of the chain there is the counter, at the other the sieve that forwards the results to `dump`.

Soter does not have native support for arithmetic operations, so line 48 defines a stub to be used by Soter that returns true or false non-deterministically, thus soundly approximating the definition based on division given in line 51.

The communication here is synchronous in spirit: whenever a message is sent, the sender actor blocks waiting for a reply. To check this is the case, we can verify the property that every mailbox contains in fact at most one message at any time. To be able to express this constraint we label the mailboxes we are interested in with the `?label_mail()` macro: the instructions in lines 14, 21 and 32 mark the mailbox of any process that may execute them with the corresponding label.

Then we can insert the following lines at the beginning of the module

```
-uncoverable("counter_mail >= 2").
-uncoverable("filter_mail >= 2").
-uncoverable("sieve_mail >= 2").
```

which state the property we want to prove. The directive `-uncoverable` is ignored by the Erlang compiler but it is interpreted by Soter as a property to be proved: all the states satisfying the constraint are considered to be 'bad states'. These inequalities state that if the total number of messages in the labelled mailboxes exceed the given bound, we are in a bad state.

Soter allows the user to mark program locations as well using the macro `?label()`; the inequalities in this case state that the total number of processes executing the labelled instruction at the same time must be less that the given bound.

When executed on the code in Figure 6.3, Soter will compute the ACS in Figure 6.4; its semantics is a sound approximation of the actual semantics of the program. A VAS description of it, incorporating the property, is then generated and fed to BFC to check for the uncoverability of bad states; in this instance BFC successfully proves the program safe.

The ACS is also rendered in graphical form in the web interface. The graph resulting from analysing the sieve example is shown unedited in Figure 6.5.

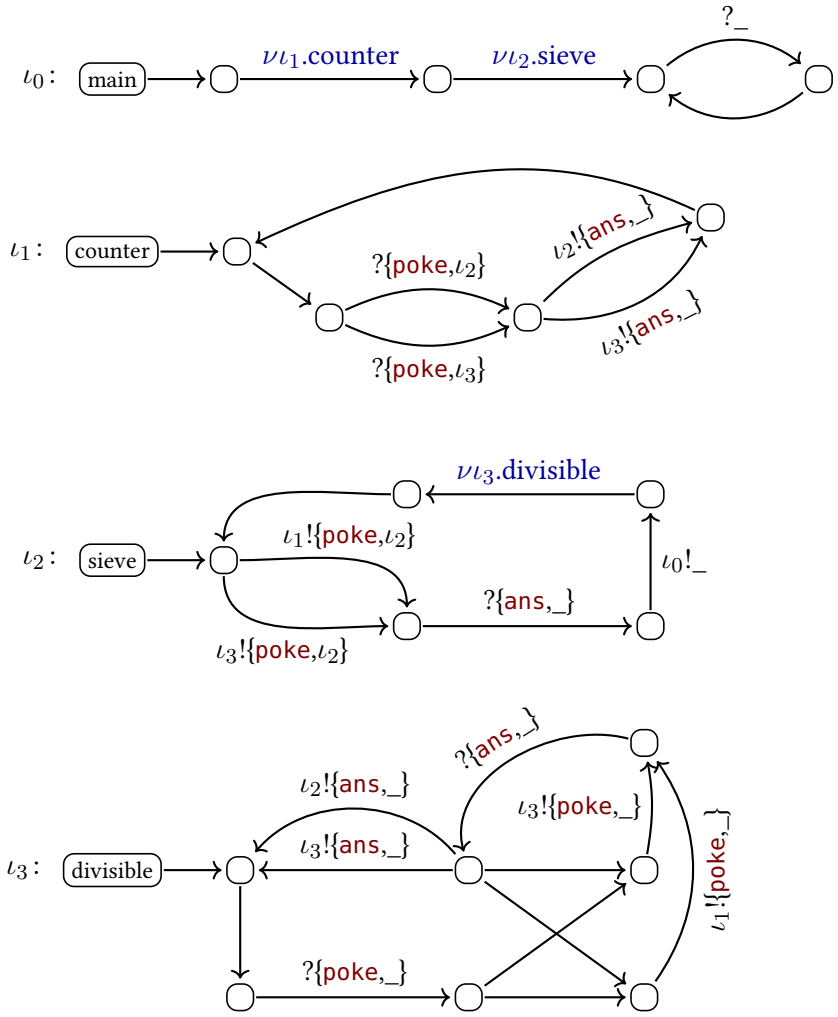


Figure 6.4 – A simplified view of the ACS generated by Soter from the sieve example. The ι_0 component represents the starting process which sets up the counter agent (ι_1) and the sieve agent (ι_2) and then becomes the dump agent. During its execution, the sieve agent spawns new filter agents, all represented by the ι_3 component.

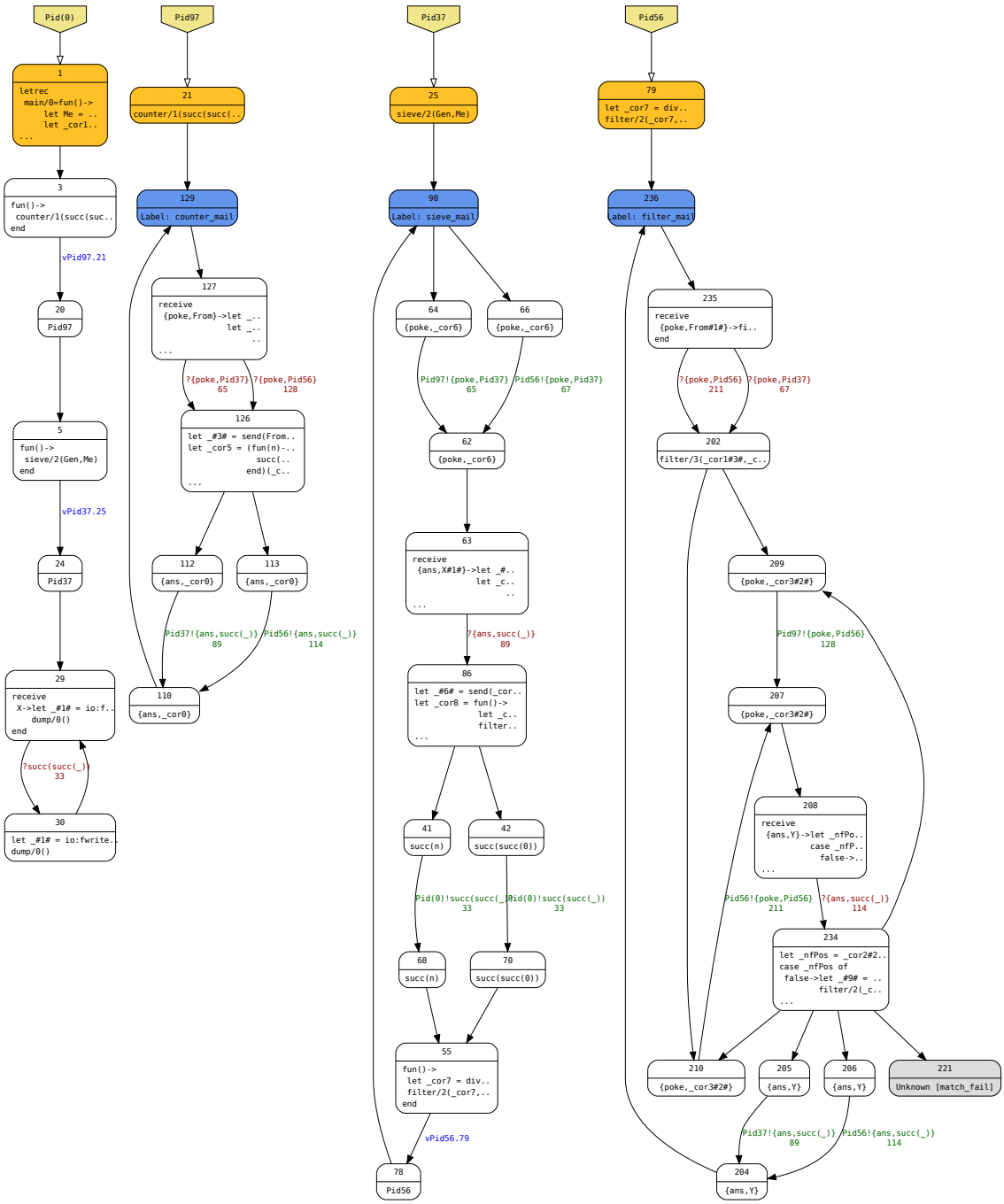


Figure 6.5 – The graphical representation of the ACS generated by Soter from sieve.

Chapter 7

Related Work

7.1 Semantics of Erlang

In his PhD thesis [Fre01], Fredlund gives a small-step operational semantics for the standard intermediate language Core Erlang; the formulation is valid for programs running on a single node. The semantics is presented in two layers, the *expression layer* which deals with the purely functional computation and relays communication effects to the second layer, the *process layer*. The latter describes the operational semantics of processes which are expressions equipped with a process identifier and a mailbox. In this work a logic for specification based on the μ-calculus and a proof system, supported by a proof assistant, is introduced. Successively, in [CS05; SF07], Fredlund's semantics is revised adding a third layer accounting for the distributed version of Erlang. The added layer defines a labelled transition system giving semantics to *multi-node* Erlang systems. This extended semantics accurately models the differences in how Erlang handles local and distributed message delivery. An advantage of Fredlund's layered approach is a clean treatment of behavioural congruence results and increased modularity.

Our semantics is consistent with Fredlund's single node semantics. However, ours defines an unlabelled transition system: we aim at modelling closed systems and the analysis is not compositional. Extending our framework to support the distributed case is straightforward.

7.2 The Many Faces of Soundness

In defining our analysis, we placed a lot of emphasis on soundness. Different definitions and uses of soundness are however employed in the literature. In many cases the property checking portion of a verification or bug-finding initiative is applied to an abstract model extracted from the program by means of some static analysis. In general, a static analysis algorithm tries to extrapolate approximate information on the dynamic behaviour of the

program. Depending on how this information is used, one may need to put different constraints on the kind of approximation in order for the constructions based on it to be correct. We identify three groups of soundness criteria:

1. *over-approximation*: the one used most often in verification, including Soter's approach,

2. *under-approximation*: as found in bug-finding frameworks,

3. *the gray area*: optimisations some times do not need strong soundness guarantees.

The nature of *bug-finding* can be seen as the dual of the verification approach: semantically, bug-finding checks a specification against an *under-approximation* of the semantics of a program; if an error is found then it must be contained in the actual semantics, if there is no error then the program could still contain undetected bugs. On the contrary, verification usually checks a specification against an *over-approximation* of the semantics; if no error is found then the program is bug-free, otherwise the error found can be either spurious or genuine.

When the static analysis information is used to direct compiler optimisations, weaker soundness criteria may be enough. This happens for instance in those cases where the static information is used to choose between two semantically equivalent machine implementations; the choice does not alter the semantics, only the performance. In these cases the precision of the abstraction allows for more aggressive optimisations but unsoundness does not cause the transformations to fail. In [CSW06], for example, a data-flow and escape analysis is computed in order to estimate which data is going to be sent in messages. This information is then used to implement a compiler optimization for an hybrid abstract machine that allows both send by copy or by reference. The optimization is able to automatically infer where it is advantageous to send by reference and generates code to store the relevant data into the shared memory rather than into the private process heap. Since message passing is completely transparent to the programmer and data that gets exchanged by message is immutable, both send-by-copy and send-by-reference are completely equivalent choices; the analysis does not need a strong correctness property such as "it marks all the data that may eventually be part of a message as shared" because any marking would give rise to the same behaviour; the only difference is in the performance of the generated code.

Similar ideas are used in approaches based on *soft-typing*. A soft-typing system is a hybrid of dynamic and static typing: a static type inference/checking algorithm is used, in conjunction with a dynamic type checker, to provide useful information to the compiler for optimizations and optional warnings to the programmer. Programs that do not have a static type would be annotated (where possible) with static type information, instead of rejected, by a soft-type system. Type safety is ultimately ensured by the dynamic

type system. A soft-typing system addressing only sequential computation, has been proposed for Erlang by Nyström [Nys03].

Even weaker guarantees are needed by testing frameworks. Although the scope of testing might seem very far from the one of verification, modern testing tools for concurrent code usually are faced with similar challenges. Automatic tests generation and exhaustive exploration of possible interleavings of concurrent tests require sophisticated techniques that take in account static information about the behaviour of the system.

Although Soter's notion of soundness is the over-approximating one, its CFA-like phase could be reused for hybrid approaches. Its explicit support for concurrency, the parametricity of the presentation and the fully formal definition makes it a good starting point for extensions of previous work on Erlang tooling.

7.3 Static Analysis

Control-flow analysis

As we mentioned, automated approaches to analysis of concurrent programs typically rely on static analysis to reconstruct information about control-flow that can be finitely representable.

CFA was originally proposed for pure λ-calculus [Jon81] and later for *continuation passing style* Scheme with side-effects [Shi88]. Roughly speaking, the purpose of CFA is to compute an over-approximation of the set of the possible expressions that can be evaluated at each program point. There are many equivalent formulations of 0-CFA, the simplest instance of CFA.

Jones and Andersen [JA07] give a grammar based formulation of 0-CFA in the context of term rewriting. In [Hei92; Hei94], a set-based formulation of a similar analysis for ML is presented; roughly speaking, the algorithm over-approximates the sets of values that can be returned by each program point during execution by first collecting set constraints induced by the structure of the program and then solving them; the constraint solver is based on regular tree grammars and relies on the interpretation of inclusions as production rules. Jagannathan and Weeks [JW95] unified and extended previous work by showing that the same analysis could be thought as an instance of Abstract Interpretation and that Shiver's 0-CFA is in fact equivalent to Heintze's set-based formulation. Van Horn and Might [VM10] recently proposed the general methodology, again based on Abstract Interpretation of an operational semantics, by which one can derive a sound CFA algorithm; as described in the introduction of Chapter 5, we apply this methodology to Erlang's semantics to extract key information to bootstrap the construction of an ACS model.

Variants of CFA, with varying degrees of precision and speed, include the context sensitive k-CFA [Shi91], Γ-CFA [MS06] where the abstract semantics implements some

optimizations usually performed in the concrete one, as garbage collection, enabling improvements in the precision and performance of the analysis algorithm, and CFA2 [VS10] where call-stacks are abstracted using more powerful abstract domains. For a thorough and accurate survey of the history and variants of CFA see [Mid12].

Concurrency specific analyses

A number of static analyses for the sequential fragment of Erlang have been proposed in the literature: examples include data-flow [CSW06], control-flow [Nys03; LS06] and escape [CS10] analyses. Concurrency primitives are either not supported, or represented very coarsely by non-deterministic choices (as for instance in [CS10; CS11]).

Might and Horn [MH11] derive a CFA for a multi-threaded extension of Scheme. The concurrency model therein is thread-based, and uses a compare-and-swap primitive. Our contribution, in addition to extending the methodology to Actor concurrency, is to use the derived parametric abstract interpretation to bootstrap the construction of an infinite-state abstract model for automated verification.

Colby [Col95] analyse the channel communication patterns of *Concurrent ML* (CML) programs. CML is based on strongly typed channels and synchronous message passing, unlike the Actor-based concurrency model of Erlang. The aim of the analysis is answering the question 'which occurrences of "send" can match which occurrences of "receive"?' to enable compiler optimisations. The identity of channels plays in CML the same role of pids in Erlang. Notably, Colby's analysis is non-uniform: it can distinguish unboundedly many names in some CML-idiomatic patterns. In Part II we will analyse the problem of removing the limitations imposed by using a finite number of pid-classes.

Reppy and Xiao [RX07] consider the problem of categorising the use of channels in CML with respect to five groups: one-shot, one-to-one, one-to-many, many-to-one, many-to-many. Different groups are better implemented with different data structures. The analysis has two phases: first, a type-informed modular CFA is used to extract an 'extended control-flow graph' of a module, second, the graph is analysed to infer the correct categorisation of channels (identified with their creation points). Interestingly, the modularity of the analysis is enabled by the encapsulation guarantees given by strong typing. Unfortunately, module boundaries are not as strongly delimited in Erlang.

Venet [Ven98] proposed an abstract interpretation framework for the analysis of π-calculus, later extended to other process algebras by Feret [Fer05] and applied to CAP, a process calculus based on the Actor model, by Garoche [GPT06]. In particular, Feret's non-standard semantics can be seen as an alternative to Van Horn and Might's methodology, but tailored for process calculi.

7.4 Abstract Model Checking

In an ICFP'99 paper [Huc99], Huch presents an approach to verify Erlang programs that combines abstract interpretation with model checking. He focuses on the following class of programs:

a) rules have order at most one,

b) tail recursion is the only form of recursion (subsequently relaxed in a follow-up paper [Huc02]),

c) mailboxes are bounded, and

d) programs spawn a fixed, statically determined, number of processes.

Given a user-supplied data abstraction function, the input program is then transformed to an abstract finite-state system; if a path property (e.g. LTL definable) can be proved for the abstract system, then it holds for the input Erlang program. Though the approach can verify rich correctness properties, the assumptions are rather restrictive. The bounds of assumptions c) and d), which are not decidable in general and hard to derive, must be established (manually) beforehand. More importantly, very often useful and correct reactive programs do not have such bounds; on the converse, when such bounds exist, they are usually a substantial part of the property one would like to prove automatically. In contrast, our method can verify Erlang programs of every finite order, with no restriction on the size of mailboxes, or the number of processes that may be spawned. Since our method of verification is by transformation to a *decidable infinite-state system* that simulates the input program, it is capable of greater accuracy.

McErlang

McErlang is a model checker for Erlang programs developed by Fredlund and Svensson [FS07]. Given an input program, a correctness property in the form of a Büchi automaton, and a user-supplied data abstraction, an abstract model of the program generated on-the-fly is explored. McErlang implements a fully-fledged Erlang runtime system replacement with support for a substantial part of the language and OTP libraries. McErlang's runtime simulates the program exploring all relevant interleavings, abstracting states and providing counterexamples when found.

An advantage of McErlang is that the specification and abstraction provided by the user is in the form of Erlang modules: the programmer does not need to learn a new language for the specifications and can use all the features of Erlang to express them. The approach is however not fully automatic: the burden of providing sound abstractions and properties that entail the correctness and termination of the model checking is entirely on the user. Soter's aim is instead that of achieving a higher degree of automation providing

a simple way to state properties and a fully automated method to verify them. While McErlang can verify liveness properties in addition to safety ones, this can only be done when the abstract model is finite state.

7.5 Type Systems for Erlang

Erlang's dynamic type system is one of the features that makes it so versatile and apt to support a iterative refinement development methodology. Many of the patterns used in real-world applications heavily rely on features that are hard or impossible to support in a statically typed language. Even if Erlang's types are only checked at runtime, they have been used in practice as a documentation tool. Many attempts at analysis of Erlang rightfully tried to harness and build on this body of information in the form of type annotations.

Marlow and Wadler [MW97] were the first to propose a static typing system for Erlang. In their paper, they define a type inference algorithm and a procedure for checking if an inferred type conforms to a user-supplied type signature; it uses a sophisticated set of subtyping constraints to minimise false positives. They choose a type language which is expressive enough to give types to common programming patterns; it supports disjoint unions, a limited form of complement, and recursive types. The type system however, supports only the sequential purely functional first-order fragment. They propose an extension that can handle higher-order but this causes the checking algorithm to be incomplete. Although Marlow and Wadler's proposal has significantly raised the level of type awareness in the Erlang community, their actual type system never caught on in the Erlang community.

Nyström's soft-type system [Nys03] also considers only the pure functional fragment of Erlang; it defines a specification language very similar to that of [MW97] including parametric type and function specifications. The idea is that the programmer is encouraged to supply type annotations at module interfaces. A data flow analysis (based on 0-CFA) is then used to compute over-approximations of the possible values of expressions and other constructs. The analysis will report inconsistencies in these annotations and warn about all program points where type clashes may occur. However because of the inherently dynamically-typed programming style favoured by Erlang programmers, an excessive amount of warnings are triggered, especially if no type annotations are provided by the programmer [LS06].

Some years later, Lindahl and Sagonas [LS06] introduced a new kind of typing called *success types*. Success typing "never disallows the use of a function that will not result in a type clash during runtime". They have the property that if a function is used in a way not allowed by its success typing, then it will definitely fail. At the cost of missing some bugs, success typing never generates false positives, positioning the approach in the *under-approximating* category. An important feature of this algorithm is compositionality

of inference: this allows the performance to scale up very well. On the other hand, success typing cannot be used to establish correctness. The specification language is closely related to the one of [MW97]: it is a subtyping system with parametric types, unions, function types. Recursive types are not supported a part from the built-in list type.

This system is at the heart of the TypEr tool [LS05] which automatically annotates Erlang code with type information, and it is also used in the popular Dialyzer tool (see Section 7.7).

Soter can be extended to exploit type information as follows. Since Soter's analysis is whole-program, it needs a closed program entry point. Soter could be used to analyse a single module by closing inputs/free variables or unknown modules by stubs that conform to the type annotations of these unknowns. Partial support for stubs and input generation is already implemented.

7.6 Session Types

A rather different approach to typing is presented in [IK01] where behavioural types for the π-calculus are introduced generalising previous work. These type systems are based on two key ideas: first, types should carry information about the sequence of actions that can happen on a channel, and not just a static invariant; second, by decorating types with capabilities and obligations, liveness properties could be checked automatically. Roughly speaking, the type system can extract from a π-term P a type which is itself a CCS-like term simulating P. Properties of the type (such as absence of locks) can then be transferred back to P by virtue of this simulation. A vast literature explores this approach further as a potential type-based foundation for structuring message-passing concurrency, a line of work revolving around the notion of *Session Types* [THK94; HVK98].

Typically, analysis done using behavioural types involves a compositional type theoretic procedure checking or synthesising types for processes and channels, followed by a non-compositional, global analysis on the types themselves, to check that the desired properties are met by the system. The ACS generation procedure has the same goal as the type-inference phase, but it is not compositional and thus potentially able to be more precise with respect to control-flow. The ACS model checking phase is also guided by user-provided assertions, while Session Types generally aim at guaranteeing generic global progress properties.

From a methodological point of view, ACS and Session Types are very different: Session Types are intended to be a specification device, while ACS are merely, at this stage, an internal abstract model; Session Types require the language using them to include the concept of *session* as primitive so that the various dialogues between agents can be structured in protocols; Soter's approach instead tries to analyse Erlang programs without assumptions on how the system is structured.

7.7 Bug-finding and Testing

Dialyzer

Dialyzer[1] (DIscrepancy AnaLYZer for ERlang programs) [LS06; CS10; CS11] is a popular bug finding tool included in the standard Erlang/OTP distribution. It evolved from a simple tool to discover typos to a more advanced analyzer capable to statically detect runtime errors and, recently, communication errors such as orphan messages or unreachable receive patterns. Dialyzer uses success types as specifications to find bugs; these specifications can be both given as code annotations or inferred by the TypEr module. Dialyzer firstly builds a control-flow, data-flow and escape [PG92] analysis of the program generating a call graph [CSW06]; this step ignores the effects of concurrent primitives. Then various kinds of analyses are performed using the information gained from the call graph: success types are derived and checked for errors; patterns for common errors in the use of primitives are looked for; these include race conditions in the use of certain built-in functions [CS10] and some simple message passing errors [CS11].

Testing Erlang programs

Claessen et al. [Cla+09] study the problem of testing and debugging concurrent, distributed Erlang applications, focusing on finding race conditions. They use a combination of three tools, QuickCheck [CH00], PULSE (a custom scheduler) and a visualizer, to enhance the possibility of detection in unit testing. QuickCheck distinguishes between two kind of property checks: sequential and parallel. The sequential ones work by testing API calls in a single process and checking pre/post-conditions. The parallel checks are tailored to test *linearizability* [HW87] of the API of a module, which amounts to require the API calls to behave atomically: a parallel test is passed if the observed results are compatible with a possible sequential execution of the calls. Conceptually, the parallel check works by running all the (independent) calls in parallel and then checking if the result equals the result of one of the possible sequential execution.

PropEr[2] (PROPerty-based testing tool for ERlang) [PS11] is a QuickCheck-inspired property based testing tool for Erlang but, unlike QuickCheck, it is open-source. One advantage over QuickCheck is that PropEr exploits type specifications, when present, to automatically generate test cases.

[1] see http://www.it.uu.se/research/group/hipe/dialyzer
[2] see http://proper.softlab.ntua.gr

Part II

Typably Hierarchical Systems

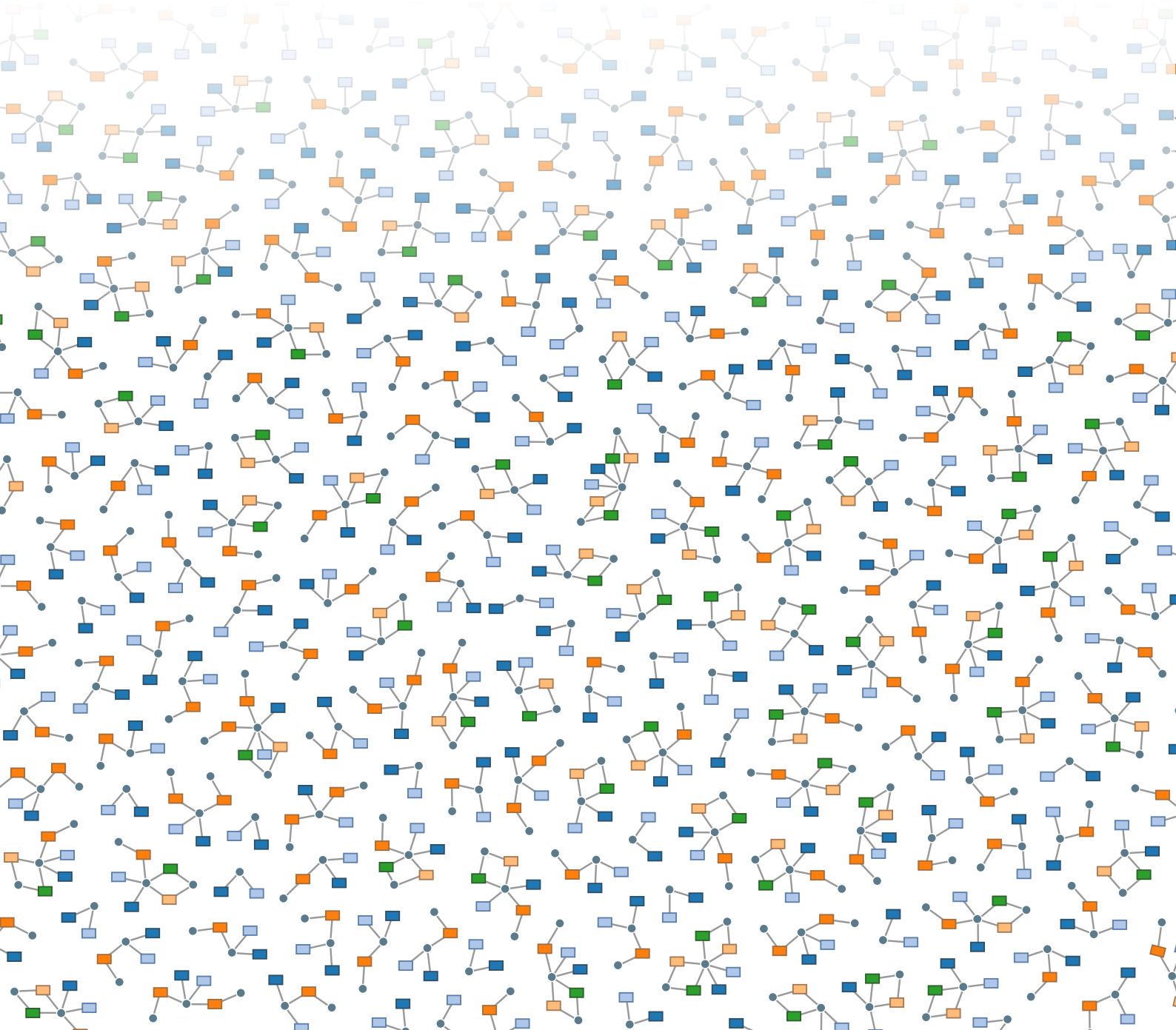


Illustration: artistic view of a reachable configuration of Example 9.2.

Chapter 8

Beyond Petri Nets

8.1 Motivation

In Part I we presented an algorithm to extract a VAS (equivalently, Petri net) model of an Erlang program. As we have outlined in Section 5.5, the extracted Petri Net models have several expressivity limitations. In this second part of the dissertation we address a specific limitation by defining a model which is richer than Petri nets and could serve as a new backend for Soter's verification pathway.

In the Petri net abstraction that Soter uses, the unboundedly many processes that can be created at runtime are partitioned into a finite number of classes. Each member of a class shares its mailbox with the other members. In other words, the addresses of unboundedly many processes may be abstracted to a single abstract address; once a message is sent to the abstract address it becomes impossible to determine which process was the intended recipient. The question motivating the work presented in this part of the dissertation is:

Can we be more precise with respect to the identities of processes,
while retaining decidability of the analysis?

We answer this question positively by studying a model of computation which goes beyond Petri nets but retains decidability of some key verification problems.

Let us illustrate the issue we want to address with an example. Consider the following Erlang definition:

```
1 distribute([], 0) → 0;
2 distribute([], N) →
3     receive
4       {ans, Y} →  Y + distribute([], N-1)
5     end;
6 distribute([X | Xs], N) →
```

```
7       C = spawn(worker),
8       C ! {task, self(), X},
9       distribute(Xs, N+1).
```

The definition of `worker` is left as a parameter, we only assume it never calls `self`. When called on a list `L` of n items, `distribute(L, 0)` would spawn n processes running the function `worker` and sending to each a message `{task, X}`. Then it will wait for n answers `{ans, Y}` from the workers and return the sum of the answers. A simple property of this system is that each worker process' mailbox will hold at most one message at all times. This property could be of interest for optimisation purposes or could have implications on the absence of errors. Suppose the definition of `worker` is of the form:

```
1  worker() →
2      receive {task, P, X} →
3          ...,
4          receive
5              {task, P, Y} → error("too much work");
6              ...
7          end
8      end.
```

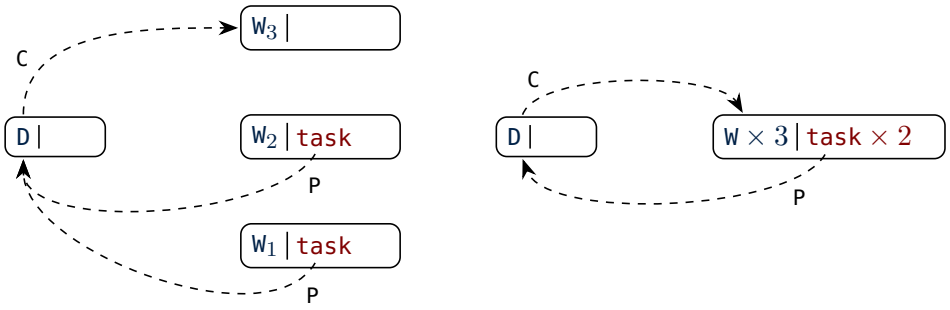
then absence of error can only be proved by proving that there is never going to be a message firing the inner receive. Note that, without a bound on the length of `L`, the number of distinct workers is unbounded.

Soter's abstraction is not powerful enough to prove this property and indeed it reports a false alarm in this case because in the generated Petri net all workers get assigned the same abstract pid. A worker in the Petri net can then receive messages addressed to other members of its class, and "steal" the second task from them. Figure 8.1 shows a view of the system after 3 items of the list are consumed. Figure 8.1(a) presents the concrete configuration when the worker with pid $\texttt{W}_3$ has just been created and the process with pid `D`, running `distribute`, is about to send a task to it. Figure 8.1(b) is the abstract Petri net view of the same configuration.

The route we want to follow to solve this problem is to keep the identities of each worker distinct in the abstraction: each of the workers will have its private mailbox. The improved abstraction would still need data, mailbox and control-flow abstractions but will be able to represent pids more accurately.

From Petri nets to π-calculus

In their seminal work, Milner, Parrow and Walker [MPW92] proposed and studied the π-calculus, a process calculus for mobile systems. The π-calculus is a concise yet



(a) Concrete (b) Abstract

Figure 8.1 – A scheme of the configuration after three iterations of the `distribute` function. Processes are represented by rounded rectangles split in two: the pid (in blue) and the contents of the mailbox (in red). The arrows point to pids known by the process and are labelled with the variable holding the pid.

expressive model of concurrent computation. Its view of a concurrent system is a set of processes exchanging messages over channels, either private or public. Both processes and private channels can be created dynamically. A key feature of the calculus is mobility: a private channel name can be sent as a message over a public one and later used to exchange messages with an initially disconnected party. The *communication topology* of a π-calculus system, i.e., the graph linking processes that share channels, is therefore dynamically evolving, in contrast to those of simpler process calculi such as CCS.

The `distribute` example above could be faithfully modelled by a π-calculus term by encoding pids as channel names. Intuitively, spawning a new worker would correspond to creating a new private name w_i representing the pid of the new process; another private name d is associated with the pid of the process running `distribute`. Asynchronously sending a message to a worker is realised by creating a π-calculus process ready to output the message d (that would be the result of `self()` in `distribute`) over w_i, effectively appending the message to the (unordered) mailbox of the worker. Figure 8.2 shows the π-calculus representation of the configuration of Figure 8.1(a).

The ability to send names over channels allows the communication topology to change over time: after sending a task, `distribute` forgets the pid of the worker and "connects" with a new one; similarly, once the worker consumes the message in its mailbox, it acquires the pid of the distributor.

From a verification point of view, proving properties of π-calculus terms is challenging: the full π-calculus is Turing-complete. As a consequence, a lot of research effort has been devoted to defining fragments of π-calculus that could be verified automatically while retaining as much expressivity as possible. To date, the most expressive fragment that has decidable verification problems is the *depth-bounded π-calculus* [Mey08]. Roughly

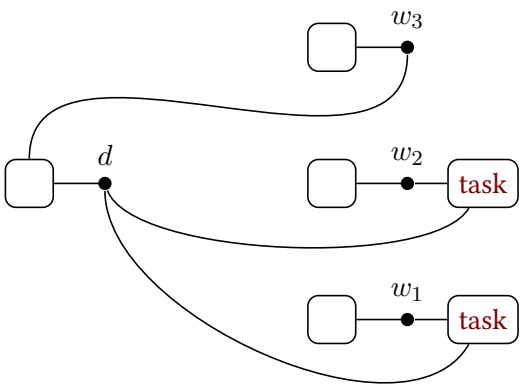


Figure 8.2 – A π-calculus view of Figure 8.1. The black circles represent private channels,the rounded boxes represent processes.

speaking, the depth of a π-calculus term can be understood as the maximum length of the simple (i.e non looping) paths in the communication topology of the term. A term is depth-bounded if there exists a $k \in \mathbb{N}$ such that the maximal nested depth of restriction of each reachable term is bounded by k. Notably, depth-bounded systems can have an infinite state-space and generate unboundedly many names.

The π-calculus abstraction of the `distribute` example we outlined is depth-bounded, even when the length of the list `L` is unbounded. It would be possible then to automatically prove that, for example, the number of tasks in each mailbox cannot exceed 1.

Unfortunately, depth boundedness is a *semantic* property of a Turing-powerful language, so it is undecidable whether a given arbitrary π-calculus term is depth-bounded. It has recently been proven that the problem becomes decidable if the bound k is fixed [RM14] but the complexity is very high. The decision procedure needs coverability of depth-bounded systems as a subroutine, which is more general than coverability of name-bounded systems, which in turn has been shown to have a non-primitive recursive lower bound [HMM13]. It is only safe to apply algorithms for the verification of depth-bounded systems on programs that can be proven depth-bounded. Since this check is undecidable in general, two approaches are possible:

1. a subset of depth-bounded systems with decidable membership is defined so that the check can be automated (even if it can reject some depth-bounded terms);

2. the abstraction is made capable of abstracting away sources of unboundedness in depth so that the abstract model is guaranteed to have a depth bounded by some measure.

In the rest of the thesis we show a possible attack to this problem following the first approach, and hint at how one can realise the second.

8.2 Contributions and Outline

The first contribution presented in this part is a novel fragment of π-calculus which we call *typably hierarchical*, which is a proper subset of the depth-bounded π-calculus. This fragment is defined by means of a type system with decidable checking and inference. The typably hierarchical fragment is rather expressive: it includes terms that are unbounded in the number of private channels and exhibit mobility, i.e. the communication topology changes over time.

The type system itself is based on the novel notion of $\mathcal{T}$*-compatibility*, where $\mathcal{T}$ is a given finite forest. We start from the observation that the communication topologies of depth-bounded terms exhibit a hierarchical structure: channels are organisable into layers with decreasing degree of sharing. Consider the example of an unbounded number of clients communicating with their local server: a message from a client containing a private channel is sent to the server's channel, the server replies to the client's request on the client's private channel. While the server's channel is shared among all the clients, the private channel of each client is shared only between itself and the server. As described in Section 1.4, $\mathcal{T}$-compatibility formalises and generalises this intuition. Roughly speaking, we associate to each channel name a *base type* which is a node in a (finite) forest $\mathcal{T}$. The forest $\mathcal{T}$ represents the hierarchical relationship between channels: it is the blueprint according to which one can organise the relationship between channels in each reachable term.

We believe that the notion of $\mathcal{T}$-compatibility has potential as a specification device: it allows the user to specify the desired relationship between channels instead of just a numeric bound on depth.

After presenting a primer on π-calculus, depth boundedness and some preliminary definitions in Chapter 9, we present the concept of typably hierarchical terms in Chapter 10 and prove the main result stating the soundness of the type system characterising them. Chapter 10 concludes with two fully worked out examples taken from Chapter 1.

In Chapter 11 we study the relation between typably hierarchical systems and other known models of computation. First we turn to the question: *is there an automata-based model that can represent the same set of systems?* Since the typably hierarchical restriction could be counterintuitive or too indirect, an automata-based presentation could help shed light on the fragment. We answer this question by means of an encoding of typably hierarchical terms into *Nested Data Class Memory Automata* (NDCMA) [CMO15], a class of automata over data-words (i.e. finite words over infinite alphabets). The translation is not 100% accurate: the coverability problem of the two models are equivalent, but for other properties such as reachability the automaton model is less expressive. To highlight the difference between the two models we also present an encoding from NDCMA to typably hierarchical terms which clearly shows the limited way in which NDCMA can simulate mobility. The two encodings are heavily based on the notion of $\mathcal{T}$-compatibility

and open an approach to fruitful interactions between process algebra and automata over infinite alphabets.

We then show how typably hierarchical terms can be seen as a proper extension of a variant of CCS called CCS$^!$ studied in [He11]. Every CCS$^!$ term, seen as a π-term with only 0-arity channels, can be typed by our type system but CCS$^!$ terms show no mobility, unlike typably hierarchical ones.

After reviewing other related approaches, we sketch some directions for future work in Chapter 12, including extensions of the type system and applications. In Section 12.2 we discuss a semantic notion of hierarchical system.

Chapter 9

The π-calculus and Depth Boundedness

9.1 The π-calculus

We use a π-calculus with guarded replication to express recursion [Mil92]. Fix a universe $\mathcal{N}$ of *names* representing channels and messages occurring in communications. The syntax follows the grammar:

$$\begin{array}{rcll} \mathcal{P} \ni P, Q & ::= & \mathbf{0} \mid \nu x.P \mid P_1 \parallel P_2 \mid M \mid !M & \text{process} \\ M & ::= & M + M \mid \pi.P & \text{choice} \\ \pi & ::= & a(x) \mid \overline{a}\langle b \rangle \mid \boldsymbol{\tau} & \text{prefix} \end{array}$$

Structural congruence is defined as the smallest congruence closed by α-conversion of bound names commutativity and associativity of choice and parallel composition with $\mathbf{0}$ as the neutral element, and the following laws for restriction, replication and scope extrusion:

$$\nu x.\mathbf{0} \equiv \mathbf{0} \qquad \nu x.\nu y.P \equiv \nu y.\nu x.P \qquad !M \equiv M \parallel !M$$

$$P \parallel \nu a.Q \equiv \nu a.(P \parallel Q) \quad (\text{if } a \notin \mathrm{fn}(P))$$

In $P = \pi.Q$, we call Q the *continuation* of P and will often omit Q altogether when $Q = \mathbf{0}$. In a term $\nu x.P$ we will occasionally refer to P as the *scope* of x. The name x is bound in both $\nu x.P$, and in $a(x).P$. We will write $\mathrm{fn}(P)$, $\mathrm{bn}(P)$ and $\mathrm{bn}_\nu(P)$ for the set of free, bound and restriction-bound names in P, respectively. A sub-term is *active* if it is not under a prefix. A name is active when it is bound by an active restriction. The set $\mathrm{active}_\nu(P)$ is the set of the active names of P. Terms of the form M and $!M$ are called *sequential*. We write $\mathcal{S}$ for the set of all sequential terms. $\mathrm{seq}(P)$ is the set of all active sequential processes of P. We will occasionally write P^i for the parallel composition of i copies of P.

Intuitively, a sequential process acts like a thread running finite-control sequential code. A term as $\boldsymbol{\tau}.(P \parallel Q)$ is the equivalent of spawning a process Q and continuing as P—although in this context the roles of P and Q are interchangeable. Interaction is done by *synchronous* communication over channels. A prefix $a(x)$ is a blocking receive on the channel a binding the message to the variable x. A prefix $\overline{a}\langle b\rangle$ is a blocking send of the message b over the channel a. Here b is itself the name of a channel that can be later used for further communication: an essential feature for mobility. A non-blocking send, as the send primitive of Erlang, can be simulated by spawning a new process doing a blocking send. Restrictions are used to make a channel name private; a restricted name can be sent as a message. A replication $!(\pi.P)$ can be understood as having a server that can spawn a new copy of P every time some other process tries to communicate with it. In other words it behaves like an infinite parallel composition $(\pi.P \parallel \pi.P \parallel \cdots)$. Note that channels are unary; extending our work to the polyadic case is straightforward (see Section 10.6) but we only consider the unary case for conciseness.

Remark 9.1. A remark about recursive behaviour. Unlike the case of process calculi without mobility, in the π-calculus representing (tail) recursive processes using process definitions or replication is equivalent [Mil93, Section 3.1]. Take for example the variant of π-calculus presented in [Mey08]: the syntax does not include replication; a term can contain process 'calls' $K\lfloor x_1, \ldots, x_n\rfloor$ where K is a process name; a finite set of process definitions $K\lfloor x_1, \ldots, x_n\rfloor := P$ gives meaning to the calls; the body of the definitions can freely contain calls to process names. Unfolding of a definition is implemented with a reduction step $K\lfloor a_1, \ldots, a_n\rfloor \to P[\, a_i/x_i\,]$. We can accurately simulate this mechanism by introducing a globally free name $call_K$ for each process name K and replacing every instance of $K\lfloor x_1, \ldots, x_n\rfloor$ with an output prefix $\overline{call_K}\langle x_1, \ldots, x_n\rangle$ and each definition $K\lfloor x_1, \ldots, x_n\rfloor := P$ with $!\big(call_K(x_1, \ldots, x_n).P\big)$.

We will often rely on the following mild assumption, that the choice of names is unambiguous, especially when selecting a representative for a congruence class.

Name Uniqueness. *Each name in P is bound at most once and* $\mathrm{fn}(P) \cap \mathrm{bn}(P) = \emptyset$.

As we will see in the rest of the chapter, the notions of depth and of hierarchy between names rely heavily on structural congruence. In particular, given a certain structure on names, there will be a specific representative of the structural congruence class that exhibits the desired properties. Nevertheless, we cannot assume the input term is always presented as that specific representative; worse yet, when the structure on names is not fixed, as in the case of type inference, we cannot fix any particular representative and be sure it will witness the desired properties. So, instead, in the semantics and in the type system, we manipulate a neutral representative called *normal form*, which is a variant of the *standard form* [Mil99]. In this way we are not distracted by the particular syntactic representation we are presented with.

$$\mathrm{nf}(\mathbf{0}) := \mathbf{0} \qquad \mathrm{nf}(\pi.P) := \pi.\,\mathrm{nf}(P) \qquad \mathrm{nf}(\boldsymbol{\nu} x.P) := \boldsymbol{\nu} x.\,\mathrm{nf}(P)$$

$$\mathrm{nf}(M + M') := \mathrm{nf}(M) + \mathrm{nf}(M') \qquad \mathrm{nf}(!M) := !(\mathrm{nf}(M))$$

$$\mathrm{nf}(P \parallel Q) := \begin{cases} \mathrm{nf}(P) & \text{if } \mathrm{nf}(Q) = \mathbf{0} \neq \mathrm{nf}(P) \\ \mathrm{nf}(Q) & \text{if } \mathrm{nf}(P) = \mathbf{0} \\ \boldsymbol{\nu} X_P X_Q.(N_P \parallel N_Q) & \text{if } \mathrm{nf}(Q) = \boldsymbol{\nu} X_Q.N_Q, \mathrm{nf}(P) = \boldsymbol{\nu} X_P.N_P \\ & \text{and } \mathrm{active}_{\boldsymbol{\nu}}(N_P) = \mathrm{active}_{\boldsymbol{\nu}}(N_Q) = \emptyset \end{cases}$$

Figure 9.1 – Definition of the $\mathrm{nf} : \mathcal{P} \to \mathcal{P}_{\mathsf{nf}}$ function.

We say that a term P is in *normal form* ($P \in \mathcal{P}_{\mathsf{nf}}$) if it is in standard form and each of its inactive subterms is also in normal form. Formally, each process in normal form follows the grammar

$$\begin{aligned} \mathcal{P}_{\mathsf{nf}} \ni N ::= {} & \boldsymbol{\nu} x_1.\cdots\boldsymbol{\nu} x_n.(A_1 \parallel \cdots \parallel A_m) \\ A ::= {} & \pi_1.N_1 + \cdots + \pi_n.N_n \\ \mid {} & !(\pi_1.N_1 + \cdots + \pi_n.N_n) \end{aligned}$$

where the sequences $x_1 \ldots x_n$ and $A_1 \ldots A_m$ may be empty; when they are both empty the normal form is the term $\mathbf{0}$. We further assume w.l.o.g. that a normal form satisfies Name Uniqueness.

Since the order of appearance of the restrictions, sequential terms or choices in a normal form is irrelevant in the technical development of our results, we use the following abbreviations. Given a finite set of indexes $I = \{i_1, \ldots, i_n\}$ we write $\prod_{i \in I} A_i$ for $(A_{i_1} \parallel \cdots \parallel A_{i_n})$, which is $\mathbf{0}$ when I is empty; and $\sum_{i \in I} \pi_i.N_i$ for $(\pi_{i_1}.N_{i_1} + \cdots + \pi_{i_n}.N_{i_n})$. This notation is justified by commutativity and associativity of the parallel and choice operators. We also write $\boldsymbol{\nu} X.P$ or $\boldsymbol{\nu} x_1\, x_2 \cdots x_n.P$ for $\boldsymbol{\nu} x_1.\cdots\boldsymbol{\nu} x_n.P$ when $X = \{x_1, \ldots, x_n\}$, or just P when X is empty; this is justified by the structural laws of restrictions. When X and Y are disjoint sets of names, we use juxtaposition for union.

Every process $P \in \mathcal{P}$ is structurally congruent to a process in normal form. The function $\mathrm{nf} : \mathcal{P} \to \mathcal{P}_{\mathsf{nf}}$, defined in Figure 9.1, extracts, from a term, a structurally equivalent normal form.

We are interested in the reduction semantics of a π-term, which can be described using the following rule.

Definition 9.1 (Reduction Semantics of π-calculus)**.** The reduction semantics of π-calculus is defined by the transition system on π-terms, with transitions satisfying $P \to Q$ if

(i) $P \equiv \nu W.(S \parallel R \parallel C) \in \mathcal{P}_{\mathsf{nf}}$,

(ii) $S = (\overline{a}\langle b\rangle.\nu Y_s.S') + M_s$,

(iii) $R = (a(x).\nu Y_r.R') + M_r$,

(iv) $Q \equiv \nu W Y_s Y_r.(S' \parallel R'[\,b/x\,] \parallel C)$,

or if

(i) $P \equiv \nu W.(\boldsymbol{\tau}.\nu Y.P' \parallel C) \in \mathcal{P}_{\mathsf{nf}}$,

(ii) $Q \equiv \nu W Y.(P' \parallel C)$.

We define the set of reachable configurations as $\mathrm{Reach}(P) := \{\, Q \mid P \to^* Q \,\}$, writing $\to^*$ to mean the reflexive, transitive closure of $\to$.

Note that the use of structural congruence takes care of unfolding replications, if necessary.

Example 9.2 (Server/Client system). Consider the term $\nu s\ c.P$ where:

$$P = !S \parallel !C \parallel !M \qquad S = s(x).\nu d.\overline{x}\langle d\rangle$$
$$C = c(m).(\overline{s}\langle m\rangle \parallel m(y).\overline{c}\langle m\rangle) \qquad M = \boldsymbol{\tau}.\nu m.\overline{c}\langle m\rangle$$

The term $!S$, which is presented in normal form, represents a server listening to a port s for a client's requests. A request is a channel x that the client sends to the server for exchanging the response. After receiving x the server creates a new name d and sends it over x. The term $!M$ creates unboundedly many clients, each with its own private mailbox m. A client on a mailbox m repeatedly sends requests to the server and concurrently waits for the answer on the mailbox before recursing. An example run of the system:

$$\begin{aligned}\nu s\ c.P &\to \nu s\ c\ m.(P \parallel \overline{c}\langle m\rangle)\\ &\to \nu s\ c\ m.(P \parallel \overline{s}\langle m\rangle \parallel m(y).\overline{c}\langle m\rangle)\\ &\to \nu s\ c\ m\ d.(P \parallel \overline{m}\langle d\rangle \parallel m(y).\overline{c}\langle m\rangle)\\ &\to \nu s\ c\ m\ d.(P \parallel \overline{c}\langle m\rangle) \equiv \nu s\ c\ m.(P \parallel \overline{c}\langle m\rangle)\end{aligned}$$

Example 9.3 (Stack-like system). Consider the normal form $\nu X.(!S \parallel \overline{s}\langle a\rangle)$ where $X = \{s, n, v, a\}$ and

$$S = s(x).\nu b.\big((\overline{v}\langle b\rangle.\overline{n}\langle x\rangle) \parallel \overline{s}\langle b\rangle\big)$$

The term $\overline{s}\langle a\rangle$ represents a stack with top element a; a term $\overline{v}\langle b\rangle.\overline{n}\langle a\rangle \parallel \overline{s}\langle b\rangle$ indicates that the top value is b, the next is a and the stack now starts from b. The stack is in an infinite loop that pushes new names (copies of b). An example run:

$$\begin{aligned}&\nu X.(!S \parallel \overline{s}\langle a\rangle)\\ &\to \nu X.(!S \parallel \nu b.((\overline{v}\langle b\rangle.\overline{n}\langle a\rangle) \parallel \overline{s}\langle b\rangle))\\ &\to \nu X.(!S \parallel \nu b\, b'.((\overline{v}\langle b\rangle.\overline{n}\langle a\rangle) \parallel (\overline{v}\langle b'\rangle.\overline{n}\langle b\rangle) \parallel \overline{s}\langle b'\rangle))\end{aligned}$$

The π-calculus is a Turing-powerful model of computation. A finite-control machine with two stacks is a well-known Turing-powerful model, and Example 9.3 shows the key construction to simulate it in the π-calculus: the (unbounded) nesting of restrictions is key in enabling the representation of an unbounded stack (or counter). The example can be extended to support the pop, push and emptyness operations (see [Mil93]). The same expressivity result is also demonstrated by several encodings of the λ-calculus [MPW92; Mil92].

9.2 Forest Representation of Terms

In the technical developement of our ideas, we will manipulate the structure of terms in non-trivial ways. To make these manipulations easier we view a term as a forest representing (part of) its abstract syntax tree. We only aim to capture the active portion of the term, so the active sequential processes will be the leaves of its forest view. Unordered branches represent parallel composition and inner nodes correspond to restrictions.

A *forest* is a simple, acyclic, directed graph $f = (N_f, \prec_f)$ such that the edge relation, $\prec_f^{-1} : N_f \rightharpoonup N_f$, is the *parent* map where $n_1 \prec n_2$ means that "n_1 is the parent of n_2", which is defined on every node of the forest except the *root*(s). We write $\leq_f$ and $<_f$ for the reflexive transitive and the transitive closure of $\prec_f$ respectively. A *path* is a sequence of nodes, $n_1 \ldots n_k$, such that for each $i < k$, $n_i \prec_f n_{i+1}$. Henceforth we assume that all forests are finite. Thus every node of a forest has a unique path to a root (and it follows that that root is unique).

An *L-labelled forest* is a pair $\varphi = (f_\varphi, \ell_\varphi)$ where f_φ is a forest and $\ell_\varphi : N_\varphi \to L$ is a labelling function on nodes. Given a path $n_1 \ldots n_k$ of f_φ, its *trace* is the induced sequence $\ell_\varphi(n_1) \ldots \ell_\varphi(n_k)$. By abuse of language, a *trace* is an element of L^* which is the trace of some path in the forest. We write $\mathrm{traces}(\varphi)$ for the set of traces of the labelled forest.

We define L-labelled forests inductively from the empty forest $(\emptyset, \emptyset)$. We write $\varphi_1 \uplus \varphi_2$ for the disjoint union of forests φ_1 and φ_2, and $l[\varphi]$ for the forest with a single root which is labelled with $l \in L$, and has the respective roots of the forest φ as children. Since the

choice of the set of nodes is irrelevant, we will always interpret equality between forests up to isomorphism (i.e. a bijection on nodes respecting parent and labeling).

Definition 9.2 (Forest representation)**.** We represent the structural congruence class of a term $P \in \mathcal{P}$ with the set of labelled forests $\mathcal{F}[\![P]\!] := \{\mathrm{forest}(Q) \mid Q \equiv P\}$ with labels in $\mathrm{active}_{\mathsf{v}}(P) \uplus \mathrm{seq}(P)$ where $\mathrm{forest}(Q)$ is defined as

$$\mathrm{forest}(Q) := \begin{cases} (\emptyset, \emptyset) & \text{if } Q = \mathbf{0} \\ Q[(\emptyset, \emptyset)] & \text{if } Q \text{ is sequential} \\ x[\mathrm{forest}(Q')] & \text{if } Q = \mathsf{v}x.Q' \\ \mathrm{forest}(Q_1) \uplus \mathrm{forest}(Q_2) & \text{if } Q = Q_1 \parallel Q_2 \end{cases}$$

Note that only leaves are labelled with sequential processes.

The *restriction height*, $\mathrm{height}_{\mathsf{v}}(\mathrm{forest}(P))$, is the length of the longest path formed of nodes labelled with names in $\mathrm{forest}(P)$.

Clearly, for any $P \in \mathcal{P}$, $\mathrm{depth}(P) = \min\{\mathrm{height}_{\mathsf{v}}(\varphi) \mid \varphi \in \mathcal{F}[\![P]\!]\}$.

Lemma 9.1. *Let φ be a forest with labels in $\mathcal{N} \uplus \mathcal{S}$. Then $\varphi = \mathrm{forest}(Q)$ with $Q \equiv Q_\varphi$ where*

$$\begin{aligned} Q_\varphi &:= \mathsf{v}X_\varphi.\textstyle\prod_{(n,A)\in I} A \\ X_\varphi &:= \{\ell_\varphi(n) \in \mathcal{N} \mid n \in N_\varphi\} \\ I &:= \{(n, A) \mid \ell_\varphi(n) = A \in \mathcal{S}\} \end{aligned}$$

provided

i) $\forall n \in N_\varphi$*, if* $\ell_\varphi(n) \in \mathcal{S}$ *then* n *has no children in* φ*, and*

ii) $\forall n, n' \in N_\varphi$*, if* $\ell_\varphi(n) = \ell_\varphi(n') \in \mathcal{N}$ *then* $n = n'$*, and*

iii) $\forall n \in N_\varphi$*, if* $\ell_\varphi(n) = A \in \mathcal{S}$ *then for each* $x \in X_\varphi \cap \mathrm{fn}(A)$ *there exists* $n' <_\varphi n$ *such that* $\ell_\varphi(n') = x$*.*

Proof. We proceed by induction on the structure of φ. The base case is when $\varphi = (\emptyset, \emptyset)$, for which we have $Q_\varphi = \mathbf{0}$ and $\varphi = \mathrm{forest}(\mathbf{0})$.

When $\varphi = \varphi_0 \uplus \varphi_1$ we have that if conditions 9.1.i, 9.1.ii and 9.1.iii hold for φ, they must hold for φ_0 and φ_1 as well, hence we can apply the induction hypothesis to them obtaining $\varphi_i\, \mathrm{forest}(Q_i)$ with $Q_i \equiv Q_{\varphi_i}$ ($i \in \{0, 1\}$). We have $\varphi = \mathrm{forest}(Q_0 \parallel Q_1)$ by definition of forest, and we want to prove that $Q_0 \parallel Q_1 \equiv Q_\varphi$. By condition 9.1.ii on φ, X_{φ_0} and X_{φ_1} must be disjoint; furthermore, by condition 9.1.iii on both φ_0 and φ_1 we can conclude that $\mathrm{fn}(Q_{\varphi_i}) \cap X_{\varphi_{1-i}} = \emptyset$. We can therefore apply scope extrusion: $Q_0 \parallel Q_1 \equiv Q_{\varphi_0} \parallel Q_{\varphi_1} \equiv \mathsf{v}X_{\varphi_0}X_{\varphi_1}.(P_{\varphi_0} \parallel P_{\varphi_1}) = Q_\varphi$.

The last case is when $\varphi = l[\varphi']$. Suppose conditions 9.1.i, 9.1.ii and 9.1.iii hold for φ. We distinguish two cases. If $l = A \in \mathcal{S}$, by 9.1.i we have $\varphi' = (\emptyset, \emptyset)$, $\varphi = \mathrm{forest}(A)$ and $A = Q_\varphi$. If $l = x \in \mathcal{N}$ then we observe that conditions 9.1.i, 9.1.ii and 9.1.iii hold for φ' under the assumption that they hold for φ. Therefore $\varphi' = \mathrm{forest}(Q')$ with $Q' \equiv Q_{\varphi'}$, and, by definition of forest, $\varphi = \mathrm{forest}(\nu x.Q')$. By condition 9.1.ii we have $x \notin X_{\varphi'}$ so $\nu x.Q' \equiv \nu x.Q_{\varphi'} \equiv \nu(X \cup \{x\}).P_{\varphi'} = Q_\varphi$. □

9.3 Communication Topology

We now briefly present the standard notion of *communication topology* of a term [MPW92; Mil99]. We will not use it as a technical tool but just as an aid to intuition.

While the forest representation of a term reflects its syntactic structure, the communication topology abstracts away from it by focussing on normal forms and reflects the underlying communication network of a term. Two sequential terms are only able to communicate directly by using a channel they both know. The underlying network we are referring to is the graph connecting sequential terms that share channels.

To give formal meaning to this intuition we use *hypergraphs*. An L-labelled hypergraph is a tuple $(V, E, \mathrm{link}, \lambda)$ where V is a finite set of vertices, E is a finite set of hyper-edges, $\mathrm{link}\colon E \to \mathscr{P}(V)$ is the map defining the set of nodes linked by each edge, and $\lambda\colon V \to L$ is the vertex labelling function. Two vertices $v, v' \in V$ are linked by a hyper-edge e just if $\mathrm{link}(e) \supseteq \{v, v'\}$. A path of length n in a hypergraph is a finite sequence $v_1 e_1 v_2 \ldots v_n e_n v_{n+1}$ with $e_i \supseteq \{v_i, v_{i+1}\}$.

Definition 9.3 (Communication Topology)**.** The *communication topology* of a normal form $\nu X.\prod_{i\in I} A_i$ is the hypergraph

$$\mathcal{G}[\![\nu X.\textstyle\prod_{i\in I} A_i]\!] = (I, X, \mathrm{link}, \lambda)$$

with labels in $L = \{A_i \mid i \in I\}$, where $\mathrm{link}(x) = \{i \mid x \in \mathrm{fn}(A_i)\}$ and $\lambda(i) = A_i$. The communication topology of an arbitrary π-term P is $\mathcal{G}[\![\mathrm{nf}(P)]\!]$.

Example 9.4. Recall the definitions of Example 9.2: $P = !S \parallel !C \parallel !M$ where

$$S = s(x).\nu d.\overline{x}\langle d\rangle \qquad C = c(m).(\overline{s}\langle m\rangle \parallel m(y).\overline{c}\langle m\rangle) \qquad M = \boldsymbol{\tau}.\nu m.\overline{c}\langle m\rangle$$

Starting from $\nu s\ c.P$ after some steps we can reach the following term:

$$Q = \nu s\ c\ m_1\ m_2\ d_1.\big(P \parallel m_1(y).\overline{c}\langle m_1\rangle \parallel \overline{s}\langle m_1\rangle \parallel m_2(y).\overline{c}\langle m_2\rangle \parallel \overline{m_2}\langle d_1\rangle\big)$$

The communication topology of this normal form is shown in Figure 9.2.

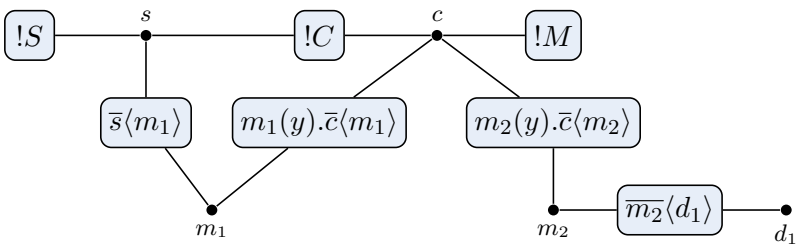


Figure 9.2 – Example of communication topology. Black dots represent hyper-edges, rounded rectangles sequential processes.

9.4 The 'tied to' relation

In this section we present a number of basic relations that help understanding the forest structure of a term. They will have an important role in the definition of the type system described in Chapter 10.

Definition 9.4 (Linked to, tied to, migratable)**.** Given a normal form $P = \nu X.\prod_{i\in I} A_i$ we say that A_i is *linked to* A_j *in* P, written $i \leftrightarrow_P j$, if $\mathrm{fn}(A_i) \cap \mathrm{fn}(A_j) \cap X \neq \emptyset$. We define the *tied-to* relation as the transitive closure of $\leftrightarrow_P$. I.e. A_i is *tied to* A_j, written $i \frown_P j$, if $\exists k \in I.\, i \leftrightarrow_P k \wedge k \frown_P j$. Furthermore, we say that a name y is *tied to* A_i *in* P, written $y \triangleleft_P i$, if $\exists j \in I.\, y \in \mathrm{fn}(A_j) \wedge j \frown_P i$. Given an input-prefixed normal form $a(y).P$ where $P = \nu X.\prod_{i\in I} A_i$, we say that A_i *is migratable in* $a(y).P$, written $\mathrm{Mig}_{a(y).P}(i)$, if $y \triangleleft_P i$.

These definitions have an intuitive meaning with respect to the communication topology of a normal form P: two sequential subterms are linked if they are connected by an edge in $\mathcal{G}[\![P]\!]$, and are tied to each other if there exist a path between them.

The following lemma indicates how the tied-to relation fundamentally constrains the possible shape of the forest of a term.

Lemma 9.2. *Let $P = \nu X.\prod_{i\in I} A_i \in \mathcal{P}_{\mathsf{nf}}$, if $i \frown_P j$ then any forest $\varphi \in \mathcal{F}[\![P]\!]$ containing two leaves labelled with A_i and A_j respectively, will be such that these leaves belong to the same tree (i.e. have a common ancestor in φ).*

Proof. We show that the claim holds in the case where A_i is linked to A_j in P. From this, a simple induction over the length of linked-to steps required to prove $i \frown_P j$, can prove the lemma.

Suppose $i \leftrightarrow_P j$. Let $Y = \mathrm{fn}(A_i) \cap \mathrm{fn}(A_j) \cap X$, we have $Y \neq \emptyset$. Both A_i and A_j are in the scope of each of the restrictions bounding names $y \in Y$ in any of the processes Q in the congruence class of P, hence, by definition of forest, the nodes labelled with A_i and A_j generated by $\mathrm{forest}(Q)$ will have a node labelled with y as common ancestor. □

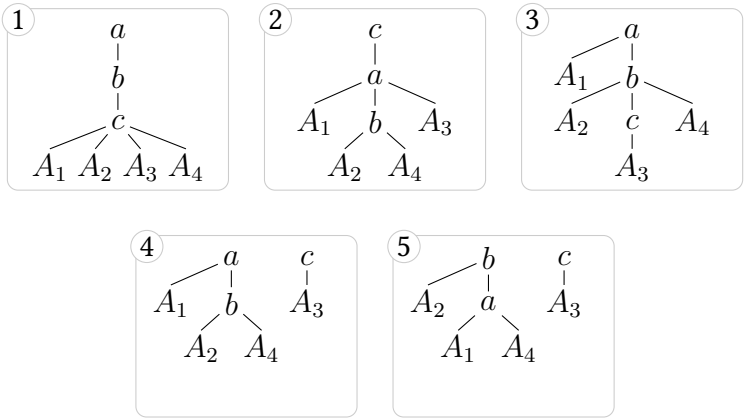


Figure 9.3 – Examples of forests in $\mathcal{F}[\![P]\!]$ of Example 9.5: $P = \nu a\, b\, c.(A_1 \parallel A_2 \parallel A_3 \parallel A_4)$ where $A_1 = a(x)$, $A_2 = b(x)$, $A_3 = c(x)$ and $A_4 = \overline{a}\langle b\rangle$.

The migratable predicate may seem obscure at first. Intuitively, it partitions the continuation of an input in two: the part of the continuation that is tied to the message and the part that is not. After the input action has been executed, only the migratable terms will need to be put under the scope of the name in the message.

Example 9.5. Take the normal form $P = \nu a\, b\, c.(A_1 \parallel A_2 \parallel A_3 \parallel A_4)$ where $A_1 = a(x)$, $A_2 = b(x)$, $A_3 = c(x)$ and $A_4 = \overline{a}\langle b\rangle$. We have $1 \leftrightarrow_P 4$, $2 \leftrightarrow_P 4$, therefore $1 \frown_P 2 \frown_P 4$ and $a \lhd_P 2$. In Figure 9.3 we show some of the forests in $\mathcal{F}[\![P]\!]$. Forest 1 represents $\mathrm{forest}(P)$. The fact that A_1, A_2 and A_4 are tied is reflected by the fact that none of the forests place them in disjoint trees. Now suppose we select only the forests in $\mathcal{F}[\![P]\!]$ that have a as an ancestor of b: in all the forests in this set, the nodes labelled with A_1, A_2 and A_4 have a as common ancestor (as in forests 1, 2, 3 and 4). In particular, in these forests A_2 is necessarily a descendent of a even if a is not one of its free names.

9.5 Depth-bounded Systems

We now review some of the notions introduced in [Mey08]. The main concept we are going to use is the one of *depth*.

Definition 9.5 (nest_ν, depth, depth-bounded term [Mey08])**.** The *nesting of restrictions* of a term is given by the function

$$\begin{aligned} \mathrm{nest}_\nu(M) := \mathrm{nest}_\nu(!M) := \mathrm{nest}_\nu(\mathbf{0}) &:= 0 \\ \mathrm{nest}_\nu(\nu x.P) &:= 1 + \mathrm{nest}_\nu(P) \\ \mathrm{nest}_\nu(P \parallel Q) &:= \max(\mathrm{nest}_\nu(P), \mathrm{nest}_\nu(Q)). \end{aligned}$$

The *depth* of a term is defined as the minimal nesting of restrictions in its congruence class:

$$\mathrm{depth}(P) := \min\{\mathrm{nest}_\nu(Q) \mid P \equiv Q\}.$$

A term $P \in \mathcal{P}$ is *depth-bounded* if there exists a $k \in \mathbb{N}$ such that for each $Q \in \mathrm{Reach}(P)$, $\mathrm{depth}(Q) \leq k$.

Remark 9.6. In [Mey08] these definitions are given on terms in a special form called *fragments*. It is easy to prove that our definition of nest_ν coincides with the one in [Mey08] on fragments and that for any fragment F and non-fragment P, if $F \equiv P$ then $\mathrm{nest}_\nu(P) \geq \mathrm{nest}_\nu(F)$. As a consequence our definition of depth coincides with the one in [Mey08].

The intuition behind the notion of depth is as follows. Consider a term $P = \nu a.\nu b.\nu c.\big(a(x) \parallel \overline{b}\langle c\rangle \parallel c(y)\big)$. We have $\mathrm{nest}_\nu(P) = 3$ but this does not reflect the *inherent* nesting of scopes induced by where names are used. In fact one can use scope extrusion to rearrange the structure of the term so that scopes are minimised. In our example we can note that although $a(x)$ is in the scope of b and c, since it does not use them it can be extruded; we obtain $P \equiv Q = \nu a.a(x) \parallel \nu b.\nu c.\big(\overline{b}\langle c\rangle \parallel c(y)\big)$ which gives $\mathrm{nest}_\nu(Q) = 2$. We can apply the same reasoning with $c(y)$ and by using exchange and scope extrusion we can obtain $P \equiv Q' = \nu a.a(x) \parallel \nu c.\big((\nu b.\overline{b}\langle c\rangle) \parallel c(y)\big)$ in which the scopes cannot be shrunk further. We still have $\mathrm{nest}_\nu(Q') = 2$, which happens to be also the depth of P.

In terms of forest representation, the nesting of restrictions becomes the height of the forest.

Lemma 9.3. *For every $P \in \mathcal{P}$, $\mathrm{nest}_\nu(P) = \mathrm{height}_\nu(\mathrm{forest}(P))$.*

Proof. Straightforward from the definition of forest. □

The concept of depth boundedness is less intuitive. In the communication topology interpretation, the concept of depth has a tight relationship with the maximum length of the simple paths. A path $v_1 e_1 v_2 \ldots v_n e_n v_{n+1}$ in $\mathcal{G}[\![P]\!]$ is *simple* if it does not repeat hyper-edges, i.e. $e_i \neq e_j$ for all $i \neq j$. A term is depth-bounded if and only if there exists a bound on the length of the simple paths of the communication topology of each reachable term [Mey08]. This allows terms to grow unboundedly in *breadth*, i.e. the degree $|\mathrm{link}(e)|$ of hyper-edges in the communication topology.

When the depth is 0, no restrictions can ever become active. Terms of depth bounded by 0 are expressive enough to represent Petri nets (see Section 11.5). Roughly speaking, a depth-bounded system is allowed to grow unboundedly in the number of parallel components and in the number of fresh used names. Indeed its state-space can be infinite. The restriction on the depth limits however the information that is accessible to each process at any given time. A sequential process in any given state can manipulate only a fixed number of names (through its free variables). Thus the only way for a sequential

process to maintain a general "data structure" of unbounded size is to nest scopes so that each name in the structure is available to it after some interaction. The interaction needed to access a particular name forms the interface of the data structure. This is the kind of structure exploited by Example 9.3 to represent an unbounded stack. Other less powerful unbounded data structures as bags (multisets) do not require unbounded nesting of scopes.

Example 9.7 (Term with unbounded depth). Recall the definition of the term S in Example 9.3:

$$S = s(x).\nu b.\big(\underbrace{(\overline{v}\langle b\rangle.\overline{n}\langle x\rangle)}_{N_{bx}} \parallel \overline{s}\langle b\rangle\big)$$

The term $\nu s\, a.(!S \parallel \overline{s}\langle a\rangle)$ is unbounded in depth: the number of nested copies of b grows every time a push is performed; it is not possible to extrude their scope to reduce the number of nested levels.

The communication topology after an arbitrary number of reductions looks like the one in Figure 9.4: the longest simple path keeps growing in length.

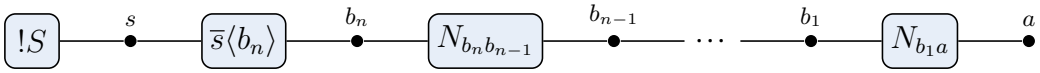


Figure 9.4 – The simple paths of a term with unbounded depth keep growing longer.

Example 9.8 (Depth-bounded term). The term in Example 9.2 is depth-bounded: all the reachable terms are congruent to terms of the form

$$Q_{ijk} = \nu s\, c.\big(P \parallel N^i \parallel Req^j \parallel Ans^k\big)$$

for some $i, j, k \in \mathbb{N}$ where $N = \nu m.\overline{c}\langle m\rangle$, $Req = \nu m.(\overline{s}\langle m\rangle \parallel m(y).\overline{c}\langle m\rangle)$, $Ans = \nu m.(\nu d.\overline{m}\langle d\rangle \parallel m(y).\overline{c}\langle m\rangle)$ and by Q^n we mean the parallel composition of n copies of the term Q. For any i, j, k, $\mathrm{nest}_\nu(Q_{ijk}) \leq 4$: the longest chain of nested restrictions is s, c, m, d. The length of simple paths in the communication topology of Q_{ijk} never exceeds 5.

Note that the terms of both Examples 9.7 and 9.8 are not *name bounded* (in the sense of [HMM13]): the number of active restrictions in the reachable terms is not bounded.

As we noted in the previous section, the π-calculus is Turing-powerful. Its power derives largely from unbounded nesting of restrictions. The depth-bounded fragment is still pretty expressive: reachability, i.e. given P and Q determining if $P \to^* Q$, is undecidable for depth-bounded systems, already at depth 1 [Mey09b]. This sets this fragment apart from Petri nets, which have a decidable reachability problem. However, Meyer [Mey08] proved that the depth-bounded fragment is in fact not Turing-powerful:

termination becomes decidable. To prove this result, Meyer shows that when P is depth-bounded, the term embedding order is a *well quasi ordering* (wqo) on the reachable terms.

Definition 9.6 (Term embedding)**.** A *forest embedding* from a forest φ_1 to a forest φ_2 is an injective function $f : N_{\varphi_1} \to N_{\varphi_2}$ preserving parent and labels, i.e. $\ell_{\varphi_1}(n) = \ell_{\varphi_2}(f(n))$ and $n \prec_{\varphi_1} n' \implies f(n) \prec_{\varphi_2} f(n')$. A forest embedding is *rooted* if it maps roots to roots.

The *term embedding order* is the relation $\preceq$ on π-terms where for two π-terms P and Q, $P \preceq Q$ if there exist P', Q' such that $P' \equiv P$, $Q' \equiv P$, and a rooted forest embedding f from $\mathrm{forest}(P')$ to $\mathrm{forest}(Q')$.

In terms of communication topologies, term embeddings corresponds to label preserving sub-hypergraph isomorphisms. That is, $P \preceq Q$ if and only if there exists a subset of nodes and edges of $\mathcal{G}[\![P]\!]$ that form an hypergraph isomorphic to $\mathcal{G}[\![Q]\!]$. Term embedding can be thought as the formalisation of the intuitive notion of *sub-system* in the π-calculus.

Recall from Chapter 2 that the *pred-basis* of a state s is the (finite) set of minimal elements of the set of predecessors of states greater than s. When a WSTS has decidable wqo and pred-basis, the coverability problem becomes decidable.

Theorem 9.4 (Depth-bounded systems are well structured [Mey08])**.** *If P is a depth-bounded π-term then $(\mathrm{Reach}(P), \to)$ ordered by $\preceq$ is a WSTS with decidable pred-basis.*

Theorem 9.4 enables a host of results: decidability of termination and (backwards) coverability of system of depth-bounded by a known k are direct consequences. Coverability is a particularly useful notion in this context. The term embedding order is a very informative relation: proving that a term is uncoverable proves that it cannot occur in any reachable term. This generalises control-state reachability and many other safety properties.

In [WZH10] it is further proved that (forward) coverability is decidable even when the depth bound k is not known *a priori*. The proof is based on the *Expand, Enlarge and Check* (EEC) algorithm [GRV06]. To instantiate the EEC coverability procedure on depth-bounded systems, Wies, Zufferey and Henzinger extend the representation of terms with an operator similar to replication, obtaining a language able to finitely represent every downward-closed set (with respect to term embedding).

The cover set is the downward closure of the reachable terms or, equivalently, the set of all coverable terms. While it always has a finite representation for depth-bounded systems, it is, somewhat counterintuitively, not computable, even when the bound of depth is known. Building on [WZH10], in [ZWH12] an approximate algorithm for computing the *cover set* of a system of depth-bounded by k is presented.

Example 9.9. A possible coverability query for the depth-bounded term of Example 9.2 is the term

$$\nu s\, c\, m\, d.(P \parallel \overline{m}\langle d\rangle \parallel \overline{m}\langle d\rangle \parallel m(y).\bar{c}\langle m\rangle)$$

If proven uncoverable, we could be certain that the mailbox of clients, here represented by the name m, will never hold more than one message $\overline{m}\langle d\rangle$. In fact the query can be algorithmically shown to be uncoverable.

A tool like Picasso [ZWH12] could even compute the set of coverable terms in the symbolic form (using the abbreviations of Example 9.8)

$$\nu s\, c.\big(P \parallel N^* \parallel Req^* \parallel Ans^*\big)$$

which symbolically represents the set $\{\nu s\, c.\big(P \parallel N^i \parallel Req^j \parallel Ans^k\big) \mid i, j, k \in \mathbb{N}\}$.

Besides enabling the design of procedures for deciding such important verification problems, depth boundedness can be useful as a correctness property of a system in itself. Consider, for example, a system modelling an unbounded number of processes, each maintaining a private queue of tasks and communicating via message-passing. In the π-calculus, structures such as lists and queues are typically modelled using private channels to represent the "next" pointers. Proving a bound in depth k for such a system would guarantee that none of the queues grows unboundedly, which is an oft-desired resource-usage property.

Unfortunately, given an arbitrary π-term, it is undecidable to determine if it is depth-bounded.

Theorem 9.5 ([Mey09b]). *Determining whether a π-term P is depth-bounded or not is undecidable.*

The theorem is proved by showing that the folklore encoding of 2-counters machines into π-calculus (which uses a variation of the stack process of Example 9.3 to implement counters with zero-tests) preserves termination. Then, if checking whether a term is depth-bounded were decidable, one could decide termination of a 2-counters machine M by

1. taking the π-calculus encoding P of M
2. deciding if P is depth-bounded:
   - if it is, then termination can be decided;
   - if it is not, then it does not terminate.

This is clearly a contradiction since termination for 2-counters machines is undecidable.

All the analyses mentioned above rely on the assumption of depth-boundedness and may even require a known bound on the depth to terminate. Theorem 9.5 makes it hard

to apply this machinery on arbitrary terms. In [RM14], among other classification results, the problem of determining whether a term P is bounded in depth by a given natural number k is proved to be decidable. The algorithm however involves solving several coverability problems as sub-steps, implying a prohibitively high complexity, with a non-primitive recursive lower bound. The results presented in the rest of the chapter try to compensate this problem by devising an expressive, yet computationally cheap, static check for depth-boundedness.

Chapter 10

Typably Hierarchical Topologies

10.1 Proving Depth-Boundedness Using $\mathcal{T}$-compatibility

In this section we will introduce the concept of *$\mathcal{T}$-compatibility*, which is a central tool in our constructions. In Section 9.5 we presented the notion of depth-boundedness. The main reason why depth-boundedness is an undecidable property is that it is a property of the set of *reachable terms*, which can be arbitrarily difficult to capture using finite means. Another difficulty in proving a bound in depth comes from the definition of depth itself: for each reachable term P, one may need to consider every term structurally congruent to P to determine its depth. In this section we show how the notion of $\mathcal{T}$-compatibility can give more structure to a proof of depth-boundedness of a term. This structure is the principle behind the type system presented in the rest of the chapter.

Suppose we want to show that a (unannotated) term P is bounded in depth by k. We can first try to characterise $\mathrm{Reach}(P)$, as we did for instance in Example 9.8 obtaining that every reachable term is congruent to

$$Q_{ijk} = \nu s\, c.\big(P \parallel \big(\nu m.\bar{c}\langle m\rangle\big)^i \parallel Req^j \parallel Ans^k\big)$$

for some $i, j, k \in \mathbb{N}$. Then usually one can see patterns in how the restrictions are used: in the example, it is easy to see that if we choose to represent Q_{ijk} by nesting c under the restrictions νm we would end up with unbounded chains of restrictions of the form $\nu m_1.\nu m_2.\cdots\nu m_i.\nu c.Q'$, for any choice of i. This equivalent view of the terms Q_{ijk} is clearly inadequate to prove depth-boundedness of P. The reason why nesting instances of m under c looks intuitively more promising is that c has a higher *degree of sharing* than m: if you group all the components that share the same c, they will not all share the same m. By contrast, if you group all the components that share the same m, they will all share the same c. This would be true even if we replicated the whole initial term.

We propose to capture this intuition by using a forest $\mathcal{T}$ that places c higher in the hierarchy of names than m. More precisely, in our example we could map the restriction νc to a node $n_c \in N_\mathcal{T}$ and the restrictions νm to a node $n_m \in N_\mathcal{T}$ such that n_c is an ancestor of n_m. But then, what does it mean for a term to respect the hierarchy? Given a map $\mu\colon \mathcal{N} \to \mathcal{T}$ from names to nodes in $\mathcal{T}$ and a forest $\varphi = (N_\varphi, \prec_\varphi, \ell_\varphi)$ with labels in $\mathcal{N} \uplus \mathcal{S}$, we write φ^μ for the forest $(N_\varphi, \prec_\varphi, \ell)$ with labels in $(\mathcal{N} \times \mathcal{T}) \uplus \mathcal{S}$ where $\ell(n) = (\ell_\varphi(n), \mu(\ell_\varphi(n)))$ if $\ell_\varphi(n) \in \mathcal{N}$ and $\ell(n) = \ell_\varphi(n)$ otherwise. We say a forest φ is $(\mu, \mathcal{T})$-*compatible* if every trace $((x_1, t_1) \dots (x_k, t_k)\ A)$ of φ^μ satisfies $t_1 <_\mathcal{T} t_2 \dots <_\mathcal{T} t_k$. If we can show that every reachable term can be represented by a $(\mu, \mathcal{T})$-compatible forest, then the term is depth-bounded.

Proposition 10.1. *A term $P \in \mathcal{P}$ is depth-bounded if and only if there exists a finite forest $\mathcal{T}$ and a function $\mu\colon \mathcal{N} \to \mathcal{T}$ such that for each $Q \in \mathrm{Reach}(P)$, $\mathcal{F}[\![Q]\!]$ contains a $(\mu, \mathcal{T})$-compatible forest.*

Proof. The $\Leftarrow$-direction is straightforward. For the $\Rightarrow$-direction, let P be bounded in depth by k and $\mathcal{T}$ be the forest $1 \prec 2 \prec \dots \prec k$. By hypothesis every reachable term Q is congruent to a term Q' with $\mathrm{nest}_\nu(Q') \leq k$. Because of this bound, we can α-rename Q' to a term Q'' where every restriction with nesting level i is renamed to x_i [Mey08]. By construction $\mathrm{forest}(Q'')$ is $(\mu, \mathcal{T})$-compatible for $\mu(x_i) = i$. □

The existential quantification on μ and $\varphi \in \mathcal{F}[\![P]\!]$ makes this notion as expressive as depth-boundedness itself: our aim is to weaken it so that it still implies boundedness in depth but can be automated. To make the proposition effective we restrict the power of μ so that the choice of φ becomes trivial.

$\mathcal{T}$-compatibility

Henceforth we will fix a *finite forest of base types* $(\mathcal{T}, \prec)$. We write $\leq$ and $<$ for the reflexive transitive and the transitive closure of $\prec$, respectively.

Types are of the form

$$\tau ::= \ t \mid t[\tau]$$

where $t \in \mathcal{T}$ is a base type. A name with type t cannot be used as a channel but can be used as a message; a name with type $t[\tau]$ can be used to transmit a name of type τ. We will write $\mathrm{base}(\tau)$ for t when $\tau = t[\tau']$ or $\tau = t$. By abuse of notation we write, for a set of types X, $\mathrm{base}(X)$ for the set of base types of the types in X. The structure of our types is closely related to the notion of sorts of [Mil93] where no particular significance is attached to base types, which instead are central in our work. For more details on the relation with other type systems see Section 11.5. An environment Γ is a partial map from names to types, which we will write as a set of *type assignments*, $x : \tau$. Given a set of names X and an environment Γ, we write $\Gamma(X)$ for the set $\{\Gamma(x) \mid x \in X \cap \mathrm{dom}(\Gamma)\}$.

Given two environments Γ and Γ' with $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$, we write $\Gamma\Gamma'$ for their union. For a type environment Γ we define

$$\min_{\mathcal{T}}(\Gamma) := \{(x:\tau) \in \Gamma \mid \forall (y:\tau') \in \Gamma.\ \mathrm{base}(\tau') \not< \mathrm{base}(\tau)\}.$$

Definition 10.1 (Annotated term)**.** A *$\mathcal{T}$-annotated π-term* (or simply *annotated π-term*) $P \in \mathcal{P}^{\mathcal{T}}$ has the same syntax as regular π-terms except restrictions take the form $\nu x:\tau$. In the abbreviated form νX, X is a set of type assignments. The semantics is the same, except type annotations get copied when a name is duplicated or renamed by structural congruence. The definition of forest representation is also extended to annotated π-terms by changing the case when $Q = \nu x:\tau.Q'$ to $(x,t)[\mathrm{forest}(Q')]$, where $\mathrm{base}(\tau) = t$. The forests in $\mathcal{F}[\![P]\!]$ will thus have labels in $(\mathrm{active}_\nu(P) \times \mathcal{T}) \uplus \mathrm{seq}(P)$. We write $\mathcal{F}_{\mathcal{T}}$ for the set of forests with labels in $(\mathcal{N} \times \mathcal{T}) \uplus \mathcal{S}$. The set $\mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$ contains all the annotated π-terms in normal form.

Now we give the formal definition of $\mathcal{T}$-compatibility on annotated terms. It coincides with the notion of $(\mu, \mathcal{T})$-compatibility of a term, when $\mu(x) = \mathrm{base}(\tau)$ where τ is the type annotation of x in the term.

Definition 10.2 ($\mathcal{T}$-compatibility)**.** Let $P \in \mathcal{P}^{\mathcal{T}}$ be an annotated π-term. A forest $\varphi \in \mathcal{F}[\![P]\!]$ is said to be *$\mathcal{T}$-compatible* if for every trace $((x_1,t_1)\ldots(x_k,t_k)\,A)$ in φ it holds that $t_1 < t_2 \ldots < t_k$. P is said to be *$\mathcal{T}$-compatible* if $\mathcal{F}[\![P]\!]$ contains a $\mathcal{T}$-compatible forest. A term is *$\mathcal{T}$-shaped* if each of its subterms is $\mathcal{T}$-compatible.

In this way, the mapping from names to types is determined *statically*, even when one considers reachable terms: the reduction relation (and α-renaming) does not alter annotations.

Example 10.1. Let us fix $\mathcal{T}$ to be the forest $\mathsf{s} \prec \mathsf{c} \prec \mathsf{m} \prec \mathsf{d}$. The normal form in Example 9.2 is $\mathcal{T}$-compatible when s and c are annotated with types τ_s and τ_c respectively, with $\mathrm{base}(\tau_s) = \mathsf{s}$ and $\mathrm{base}(\tau_c) = \mathsf{c}$; indeed we have

$$\mathrm{forest}(\nu(s:\tau_s)\ (c:\tau_c).P) = (s,\mathsf{s})\big[(c,\mathsf{c})[\,!S[(\emptyset,\emptyset)] \uplus !C[(\emptyset,\emptyset)] \uplus !M[(\emptyset,\emptyset)]\,]\big].$$

By annotating m and d with types with base type m and d respectively, the term is also $\mathcal{T}$-shaped.

Since $\mathcal{T}$-compatibility is a condition on types, α-renaming does not interfere with it.

Lemma 10.2. *If $\mathrm{forest}(P)$ is $\mathcal{T}$-compatible then for any term Q which is an α-renaming of P, $\mathrm{forest}(Q)$ is $\mathcal{T}$-compatible.*

Lemma 10.3. *Let $P = \nu X.\prod_{i\in I} A_i$ be a $\mathcal{T}$-compatible normal form, $Y \subseteq X$ and $J \subseteq I$. Then $P' = \nu Y.\prod_{j\in J} A_j$ is $\mathcal{T}$-compatible.*

Proof. Take a $\mathcal{T}$-compatible forest $\varphi \in \mathcal{F}[\![P]\!]$. By Lemma 10.2 we can assume without loss of generality that $\varphi = \mathrm{forest}(Q)$ where proving $Q \equiv P$ does not require α-renaming. Clearly, removing the leaves that do not correspond to sequential terms indexed by Y does not affect the $\mathcal{T}$-compatibility of φ. Similarly, if a restriction $(x : \tau) \in X$ is not in Y, we can remove the node of φ labelled with $(x, \mathrm{base}(\tau))$ by making its parent the new parent of its children. This operation is unambiguous under Name Uniqueness and does not affect $\mathcal{T}$-compatibility, by transitivity of $<$. We then obtain a forest φ' which is $\mathcal{T}$-compatible and that, by Lemma 9.1, is the forest of a term congruent to the desired normal form P'. □

It is clear from the definition that if a π-term P is $\mathcal{T}$-compatible then $\mathrm{depth}(P)$ is bounded by the length of the longest strictly increasing chain in $\mathcal{T}$; since $\mathcal{T}$ is assumed to be finite, the bound on the depth is finite.

Proposition 10.4. *Let $\mathcal{T}$ be a forest and P an annotated π-term. If every $Q \in \mathrm{Reach}(P)$ is $\mathcal{T}$-compatible, then P is depth-bounded.*

Example 10.2. Fix $\mathcal{T}$ to be the forest $\mathsf{n} \prec \mathsf{v} \prec \mathsf{s} \prec \mathsf{a}$ and take the term of Example 9.3 annotating it with types such that the base types of the names n, v, s, a and b are $\mathsf{n}, \mathsf{v}, \mathsf{s}, \mathsf{a}$ and a respectively. The term $\nu n\ v\ s\ a.(!S \parallel \overline{s}\langle a\rangle)$ is $\mathcal{T}$-compatible, but the term

$$Q = \nu n\ v\ s\ a\ b\ b'.(!S \parallel (\overline{v}\langle b\rangle.\overline{n}\langle a\rangle) \parallel (\overline{v}\langle b'\rangle.\overline{n}\langle b\rangle) \parallel \overline{s}\langle b'\rangle),$$

reachable from it, is not: b and b' have the same base type a but need to be in the same trace in any forest of $\mathcal{F}[\![Q]\!]$. As we have shown in Example 9.7, this term is not bounded in depth, so there cannot be any finite $\mathcal{T}$ such that every reachable term is $\mathcal{T}$-compatible.

10.2 A canonical representative

While many forests in $\mathcal{F}[\![P]\!]$ can be witnesses of the $\mathcal{T}$-compatibility of P, we want to characterise the shape of a witness that *must* exist if P is $\mathcal{T}$-compatible. Such forest is identified by $\Phi_{\mathcal{T}}(\mathrm{nf}(P))$. Before showing the definition, let us briefly explain the intent of Φ. The type system we are going to define in Section 10.3 aims to guarantee that $\mathcal{T}$-compatibility is invariant under reduction for typable terms. We will need to show that starting from a typable $\mathcal{T}$-compatible term P, any step will reduce it to a (typable) $\mathcal{T}$-compatible term. The hypothesis of $\mathcal{T}$-compatibility of P can be used to extract a $\mathcal{T}$-compatible forest φ from $\mathcal{F}[\![P]\!]$. It is very important to select the shallowest such φ so to minimise the chances to unnecessarily increase the depth of the term it represents. As Lemma 10.6 underlines, the forest extracted by Φ is the shallowest among all the $\mathcal{T}$-compatible forests in $\mathcal{F}[\![P]\!]$.

Definition 10.3 ($\Phi_{\mathcal{T}}$)**.** The function $\Phi_{\mathcal{T}} \colon \mathcal{P}^{\mathcal{T}}_{\mathsf{nf}} \to \mathcal{F}_{\mathcal{T}}$ is defined inductively as

$$\begin{aligned}\Phi_{\mathcal{T}}(\textstyle\prod_{i\in I} A_i) &:= \biguplus_{i\in I} \{A_i[]\} \\ \Phi_{\mathcal{T}}(\nu X.\textstyle\prod_{i\in I} A_i) &:= \Big(\biguplus \Big\{ (x, \mathrm{base}(\tau))[\Phi_{\mathcal{T}}(\nu Y_x.\textstyle\prod_{j\in I_x} A_j)] \;\Big|\; (x:\tau) \in \min_{\mathcal{T}}(X) \Big\}\Big) \\ &\quad \uplus \Phi_{\mathcal{T}}(\nu Z.\textstyle\prod_{r\in R} A_r)\end{aligned}$$

where $X \neq \emptyset$, $P = \nu X.\prod_{i\in I} A_i$ and

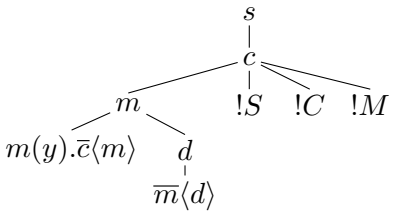
$$\begin{aligned} I_x &= \{i \in I \mid x \lhd_P i\} \\ R &= I \setminus \big(\textstyle\bigcup_{(x:\tau)\in \min_{\mathcal{T}}(X)} I_x \big) \\ Y_x &= \{(y:\tau) \in X \mid \exists i \in I_x.\, y \in \mathrm{fn}(A_i)\} \setminus \min_{\mathcal{T}}(X) \\ Z &= X \setminus \big(\textstyle\bigcup_{(x:\tau)\in \min_{\mathcal{T}}(X)} Y_x \cup \{x:\tau\} \big) \end{aligned}$$

We omit the subscript from $\Phi_{\mathcal{T}}$ when irrelevant or clear from the context.

Example 10.3. In the run shown in Example 9.2, after three steps we reach

$$Q = \nu s\, c\, m\, d.(P \parallel \overline{m}\langle d\rangle \parallel m(y).\overline{c}\langle m\rangle).$$

The forest $\Phi_{\mathcal{T}}(Q)$, when $\mathcal{T}$ and types annotations are as in Example 10.1, is



where the nodes show only the name components of their labels for conciseness. Note how the scope of names is minimised while respecting $\mathcal{T}$-compatibility.

Consider the term P in Example 9.5, with annotations $a : \mathsf{a}[\mathsf{b}[t]]$, $b : \mathsf{b}[t]$ and $c : \mathsf{c}[t']$. Forests 4 and 5 of Figure 9.3 represent $\Phi_{\mathcal{T}}(P)$ when $\mathcal{T}$ is $\mathsf{a} \prec \mathsf{b}$ and $\mathsf{b} \prec \mathsf{a}$ respectively.

The following lemma establishes the formal link between Φ and $\mathcal{T}$-compatibility.

Lemma 10.5. *Let $P \in \mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$. Then:*

a) $\Phi_{\mathcal{T}}(P)$ is a $\mathcal{T}$-compatible forest;

b) $\Phi_{\mathcal{T}}(P) \in \mathcal{F}[\![P]\!]$ if and only if P is $\mathcal{T}$-compatible;

c) if $P \equiv Q \in \mathcal{P}^{\mathcal{T}}$ then $\Phi_{\mathcal{T}}(P) \in \mathcal{F}[\![Q]\!]$ if and only if Q is $\mathcal{T}$-compatible.

Proof. Item a) is an easy induction on the cardinality of X.

Item b) requires more work. By item a) $\Phi(P)$ is $\mathcal{T}$-compatible so $\Phi(P) \in \mathcal{F}[\![P]\!]$ proves that P is $\mathcal{T}$-compatible.

To prove the $\Leftarrow$-direction we assume that $P = \nu X.\prod_{i\in I} A_i$ is $\mathcal{T}$-compatible and proceed by induction on the cardinality of X to show that $\Phi(P) \in \mathcal{F}[\![P]\!]$. The base case is when $X = \emptyset$: $\Phi(P) = \Phi(\prod_{i\in I} A_i) = \biguplus_{i\in I} \{A_i[]\} = \mathrm{forest}(\prod_{i\in I} A_i) = \mathrm{forest}(P) \in \mathcal{F}[\![P]\!]$. For the induction step, we observe that $X \neq \emptyset$ implies $\min_{\mathcal{T}}(X) \neq \emptyset$ so, $Z \subset X$ and for each $(x:\tau) \in \min_{\mathcal{T}}(X)$, $Y_x \subset X$ since $x \notin Y_x$. This, together with Lemma 10.3, allows us to apply the induction hypotesis on the terms $P_x = \nu Y_x.\prod_{j\in I_x} A_j$ and $P_R = \nu Z.\prod_{r\in R} A_r$, obtaining that there exist terms $Q_x \equiv P_x$ and $Q_R \equiv P_R$ such that $\mathrm{forest}(Q_x) = \Phi(P_x)$ and $\mathrm{forest}(Q_R) = \Phi(P_R)$ where all the forests $\mathrm{forest}(Q_x)$ and $\mathrm{forest}(Q_R)$ are $\mathcal{T}$-compatible. Let $Q = \prod \{\nu(x:\tau).Q_x \mid (x:\tau) \in \min_{\mathcal{T}}(X)\} \parallel Q_R$, then $\mathrm{forest}(Q) = \Phi(P)$. To prove the claim we only need to show that $Q \equiv P$. We have $Q \equiv \prod \{\nu(x:\tau).\nu Y_x.\prod_{j\in I_x} A_j \mid (x:\tau) \in \min_{\mathcal{T}}(X)\} \parallel P_R$ and we want to apply extrusion to get $Q \equiv \nu Y_{\min}.\left(\prod_{i\in I_{\min}} A_i\right) \parallel P_R$ for $I_{\min} = \biguplus \{I_x \mid (x:\tau) \in \min_{\mathcal{T}}(X)\}$, $Y_{\min} = \min_{\mathcal{T}}(X) \uplus \biguplus \{Y_x \mid (x:\tau) \in \min_{\mathcal{T}}(X)\}$ which adds an obligation to prove that

i) I_x are all pairwise disjoint so that $I_{\min}$ is well-defined,

ii) Y_x are all pairwise disjoint and all disjoint from $\min_{\mathcal{T}}(X)$ so that $Y_{\min}$ is well-defined,

iii) $Y_x \cap \mathrm{fn}(A_j) = \emptyset$ for every $j \in I_z$ with $z \neq x$ so that we can apply the extrusion rule.

To prove condition i), assume by contradiction that there exists an $i \in I$ and names $x, y \in \min_{\mathcal{T}}(X)$ with $x \neq y$, such that both x and y are tied to A_i in P. By transitivity of the tied-to relation, we have $I_x = I_y$. By Lemma 9.2 all the A_j with $j \in I_x$ need to be in the same tree in any forest $\varphi \in \mathcal{F}[\![P]\!]$. Since P is $\mathcal{T}$-compatible there exist such a φ which is $\mathcal{T}$-compatible and has every A_j as label of leaves of the same tree. This tree will include a node n_x labelled with $(x, \mathrm{base}(X(x)))$ and a node n_y labelled with $(y, \mathrm{base}(X(y)))$. By $\mathcal{T}$-compatible of φ and the existence of a path between n_x and n_y we infer $\mathrm{base}(X(x)) < \mathrm{base}(X(y))$ or $\mathrm{base}(X(y)) < \mathrm{base}(X(x))$ which contradicts the assumption that $x, y \in \min_{\mathcal{T}}(X)$.

Condition ii) follows from condition i): suppose there exists a $(z:\tau) \in X \cap Y_x \cap Y_y$ for $x \neq y$, then we would have that $z \in \mathrm{fn}(A_i) \cap \mathrm{fn}(A_j)$ for some $i \in I_x$ and $j \in I_y$, but then $i \frown_P j$, meaning that $i \in I_y$ and $j \in I_x$ violating condition i). The fact that $Y_x \cap \min_{\mathcal{T}}(X) = \emptyset$ follows from the definition of Y_x. The same reasoning proves condition iii).

Now we have $Q \equiv \nu Y_{\min}.\left(\prod_{i\in I_{\min}} A_i\right) \parallel \nu Z.\prod_{r\in R} A_r$ and we want to apply extrusion again to get $Q \equiv \nu Y_{\min} Z.\prod \{A_i \mid i \in (I_{\min} \uplus R)\}$ which is sound under the following conditions:

iv) $Y_{\min} \cap Z = \emptyset$,

v) $I_{\min} \cap R = \emptyset$,

vi) $Z \cap \mathrm{fn}(A_i) = \emptyset$ for all $i \notin R$

of which the first two hold trivially by construction, while the last follows from condition viii) below, as a name in the intersection of Z and a $\mathrm{fn}(A_i)$ would need to be in X but not in $Y_{\min}$. To be able to conclude that $Q \equiv P$ it remains to prove that

vii) $I = I_{\min} \uplus R$ and

viii) $X = Y_{\min} \uplus Z$

which are also trivially valid by inspection of their definitions. This concludes the proof for item b).

Finally, for every $Q \in \mathcal{P}^{\mathcal{T}}$ such that $Q \equiv P$, $\Phi(P) \in \mathcal{F}[\![Q]\!]$ if and only if $\Phi(P) \in \mathcal{F}[\![P]\!]$ by definition of $\mathcal{F}[\![-]\!]$; since $\Phi(P)$ is $\mathcal{T}$-compatible we can infer that Q is $\mathcal{T}$-compatible if and only if $\Phi(P) \in \mathcal{F}[\![Q]\!]$, which proves item c). □

In light of Lemma 10.5, we can turn the computation of $\Phi_{\mathcal{T}}(P)$ into an algorithm to check $\mathcal{T}$-compatibility of P: it is sufficient to compute $\Phi_{\mathcal{T}}(P)$ and check at each step that the sets I_x, R form a partition of I and the sets Y_x, Z form a partition of X. If the checks fail $\Phi_{\mathcal{T}}(P) \notin \mathcal{F}[\![P]\!]$ and P is not $\mathcal{T}$-compatible, otherwise the obtained forest is a witness of $\mathcal{T}$-compatibility.

Lemma 10.6. *Let $P = \nu X.\prod_{i\in I} A_i \in \mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$ be a $\mathcal{T}$-compatible normal form. Then for every trace $((x_1, t_1) \dots (x_k, t_k)\ A_j)$ in the forest $\Phi(P)$, for every $i \in \{1, \dots, k\}$, we have $x_i \lhd_P j$ (i.e. x_i is tied to A_j in P).*

Proof. Straightforward from the definition of I_x in Φ: when a node labelled by (x, t) is introduced, its subtree is extracted from a recursive call on a term that contains all and only the sequential terms that are tied to x. □

Remark 10.4. $\Phi(P)$ satisfies conditions 9.1.i, 9.1.ii and 9.1.iii of Lemma 9.1.

10.3 A Type System for Hierarchical Topologies

We now define a type system to prove depth boundedness. Our goal is to use Proposition 10.4 by devising a type system, parametrised over $\mathcal{T}$, such that typability implies invariance of $\mathcal{T}$-compatibility under reduction. Typability of a $\mathcal{T}$-compatible term P would then imply that every term reachable from it is $\mathcal{T}$-compatible, entailing depth boundedness of P.

$$\frac{\begin{array}{c}\forall i \in I.\, \Gamma, X \vdash_{\mathcal{T}} A_i \\ \forall i \in I.\, \forall x : \tau_x \in X.\, x \lhd_P i \implies \mathrm{base}(\Gamma(\mathrm{fn}(A_i))) < \mathrm{base}(\tau_x)\end{array}}{\Gamma \vdash_{\mathcal{T}} \nu X.\prod_{i\in I} A_i}\ \textsc{Par}$$

$$\frac{\forall i \in I.\, \Gamma \vdash_{\mathcal{T}} \pi_i.P_i}{\Gamma \vdash_{\mathcal{T}} \sum_{i\in I} \pi_i.P_i}\ \textsc{Choice} \qquad \frac{\Gamma \vdash_{\mathcal{T}} A}{\Gamma \vdash_{\mathcal{T}} !A}\ \textsc{Repl} \qquad \frac{\Gamma \vdash_{\mathcal{T}} P}{\Gamma \vdash_{\mathcal{T}} \boldsymbol{\tau}.P}\ \textsc{Tau}$$

$$\frac{a : t_a[\tau_b] \in \Gamma \qquad b : \tau_b \in \Gamma \qquad \Gamma \vdash_{\mathcal{T}} Q}{\Gamma \vdash_{\mathcal{T}} \overline{a}\langle b\rangle.Q}\ \textsc{Out}$$

$$\frac{\begin{array}{c}a : t_a[\tau_x] \in \Gamma \qquad \Gamma, x : \tau_x \vdash_{\mathcal{T}} \nu X.\prod_{i\in I} A_i \\ \mathrm{base}(\tau_x) < t_a \vee \big(\forall i \in I.\ \mathrm{Mig}_{a(x).P}(i) \implies \mathrm{base}(\Gamma(\mathrm{fn}(A_i) \setminus \{a\})) < t_a\big)\end{array}}{\Gamma \vdash_{\mathcal{T}} a(x).\nu X.\prod_{i\in I} A_i}\ \textsc{In}$$

Figure 10.1 – A type system for proving depth boundedness. The term P stands for $\nu X.\prod_{i\in I} A_i$.

A judgement $\Gamma \vdash_{\mathcal{T}} P$ means that $P \in \mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$ can be typed under assumptions Γ, over the tree $\mathcal{T}$; we say that P is *typable* if $\Gamma \vdash_{\mathcal{T}} P$ is provable for some Γ and $\mathcal{T}$. An arbitrary term $P \in \mathcal{P}^{\mathcal{T}}$ is said to be *typable* if its normal form is. The typing rules are presented in Figure 10.1.

The type system presents several non-standard features. First, it is defined on normal forms as opposed to general π-terms. This choice is motivated by the fact that different syntactic presentations of the same term may be misleading when trying to analyse the relation between the structure of the term and $\mathcal{T}$. The rules need to guarantee that a reduction will not break $\mathcal{T}$-compatibility, which is a property of the congruence class of the term. As justified by Lemma 9.2, the scope of names in a congruence class may vary, but the tied-to relation puts constraints on the structure that must be obeyed by all members of the class. Therefore the type system is designed around this basic concept, rather than the specific scoping of any representative of the structural congruence class. Second, no type information is associated with the typed term, only restricted names hold type annotations. Third, while the rules are compositional, the constraints on base types have a global flavour due to the fact that they involve the structure of $\mathcal{T}$ which is a global parameter of typing proofs.

Let us illustrate intuitively how the constraints enforced by the rules guarantee preservation of $\mathcal{T}$-compatibility. Consider the term

$$P = \nu e\, a.\Big(\nu b.\big(\overline{a}\langle b\rangle.A_0\big) \parallel \nu d.\big(a(x).Q\big)\Big)$$

with $Q = \nu c.(A_1 \parallel A_2 \parallel A_3)$, $A_0 = b(y)$, $A_1 = \overline{x}\langle c\rangle$, $A_2 = c(z).\overline{a}\langle e\rangle$ and $A_3 = \overline{a}\langle d\rangle$. Let $\mathcal{T}$ be the forest with $t_e \prec t_a \prec t_b \prec t_c$ and $t_a \prec t_d$, where t_x is the base type of the (omitted) annotation of the restriction νx, for $x \in \{a, b, c, d, e\}$. The reader can check that $\mathrm{forest}(P)$ is $\mathcal{T}$-compatible.

In the traditional understanding of mobility, we would interpret the communication of b over x as an application of scope extrusion to include $\nu d.\big(a(x).Q\big)$ in the scope of b and then syncronisation over a with the application of the substitution $[\,b/x\,]$ to Q; note that the substitution is only valid because the scope of b has been extended to include the receiver.

Our key observation is that we can instead interpret this communication as a migration of the subcomponents of Q that do get their scopes changed by the reduction, from the scope of the receiver to the scope of the sender. For this operation to be sound, the subcomponents of Q migrating to the sender's scope cannot use the names that are in the scope of the receiver but not of the sender.

In our specific example, after the synchronisation between the prefixes $\overline{a}\langle b\rangle$ and $a(x)$, b is substituted to x in A_1 resulting in the term $A_1' = \overline{b}\langle c\rangle$ and A_0, A_1', A_2 and A_3 become active. The scope of A_0 can remain unchanged as it cannot know more names than before as a result of the communication. By contrast, A_1 now knows b as a result of the substitution $[\,b/x\,]$: A_1 needs to migrate under the scope of b. Since A_1 uses c as well, the scope of c needs to be moved under b; however A_2 uses c so it needs to migrate under b with the scope of c. A_3 instead does not use neither b nor c so it can avoid migration and its scope remains unaltered.

This information can be formalised using the tied-to relation: on one hand, A_1 and A_2 need to be moved together because $1 \frown_Q 2$ and they need to be moved because $x \triangleleft_Q 1, 2$. On the other hand, A_3 is not tied to neither A_1 nor A_2 in Q and does not know x, thus it is not migratable. After reduction, our view of the reactum is the term

$$\nu a.\Big(\nu b.\big(A_0 \parallel \nu c.(A_1' \parallel A_2)\big) \parallel \nu d.A_3\Big)$$

the forest of which is $\mathcal{T}$-compatible. Rule Par, applied to A_1 and A_2, ensures that c has a base type that can be nested under the one of b. Rule In does not impose constraints on the base types of A_3 because A_3 is not migratable. It does however check that the base type of e is an ancestor of the one of a, thus ensuring that both receiver and sender are already in the scope of e. The base type of a does not need to be further constrained since the fact that the synchronisation happened on it implies that both the receiver and the sender were already under its scope; this implies, by $\mathcal{T}$-compatibility of P, that c can be nested under a.

We now describe the purpose of the rules of the type system in more detail. Most of the rules just drive the derivation through the structure of the term. The crucial constraints are checked by Par, In and Out.

The Out rule The main purpose of rule Out is enforcing types to be consistent with the dataflow of the process: the type of the argument of a channel a must agree with the types of all the names that may be sent over a. This is a very coarse sound over-approximation of the dataflow; if necessary it could be refined using well-known techniques from the literature but a simple approach is sufficient here to type interesting processes.

The Par rule Rule Par is best understood imagining the normal form to be typed, P, as the continuation of a prefix $\pi.P$. In this context a reduction exposes each of the active sequential subterms of P which need to have a place in a $\mathcal{T}$-compatible forest for the reactum. The constraint in Par can be read as follows. A 'new' leaf A_i may refer to names already present in the forests of the reaction context; these names are the ones mentioned in both $\mathrm{fn}(A_i)$ and Γ. Then we must be able to insert A_i so that we can find these names in its path. However, A_i must belong to a tree containing all the names in X that are tied to it in P. So by requiring every name tied to A_i to have a base type greater than any name in the context that A_i may refer to, we make sure that we can insert the continuation in the forest of the context without violating $\mathcal{T}$-compatibility. Note that $\Gamma(\mathrm{fn}(A_i))$ contains only types that annotate names both in Γ and $\mathrm{fn}(A_i)$, that is, names which are not restricted by X and are referenced by A_i (and therefore come from the context).

The In rule Rule In serves two purposes: on the one hand it requires the type of the messages that can be sent through a to be consistent with the use of the variable x which will be bound to the messages; on the other hand, it constrains the base types of a and x so that synchronisation can be performed without breaking $\mathcal{T}$-compatibility.

The second purpose is achieved by distinguishing two cases, represented by the two disjuncts of the condition on base types of the rule. In the first case, the base type of the message is an ancestor of the base type of a in $\mathcal{T}$. This implies that in any $\mathcal{T}$-compatible forest representing $a(x).P$, the name b sent as message over a is already in the scope of P. Under this circumstance, there is no real mobility, P does not know new names by the effect of the substitution $[\,b/x\,]$, and the $\mathcal{T}$-compatibility constraints to be satisfied are in essence unaltered.

The second case is more complicated as it involves genuine mobility. This case also requires a slightly non-standard feature: not only do the premises predicate on the direct subcomponents of an input prefixed term, but also on the direct subcomponents of the continuation. This is needed to be able to separate the continuation in two parts: the one requiring migration and the one that does not. The situation during execution is depicted in Figure 10.2. The non migratable sequential terms behave exactly as the case of the first disjunct: their scope is unaltered. The migratable ones instead are intended to be inserted as descendent of the node representing the message b in the forest of the reaction context.

For this to be valid without rearrangement of the forest of the context, we need all the names in the context that are referenced in the migratable terms, to be also in the

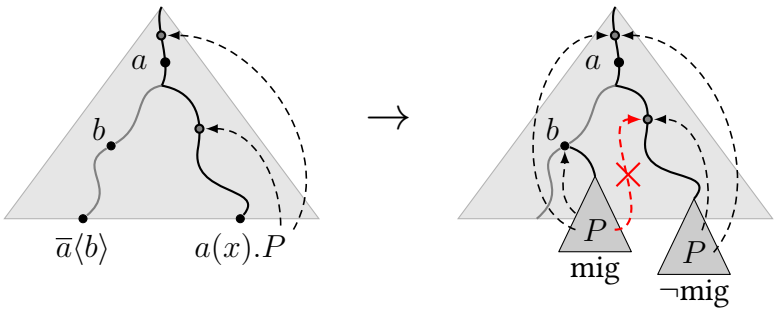


Figure 10.2 – Explanation of constraints imposed by rule In. The dashed lines represent references to names restricted in the reduction context. When adding the migratable continuation of P under b, there cannot be references to names not already in the path to b. Assuming the forests are $\mathcal{T}$-compatible, this can be enforced by requiring the names referred by the migratable portion of P to be smaller than a, which is surely a common ancestor of receiver and sender.

scope at b; we make sure this is the case by requiring the free names of any migratable A_i that are from the context (i.e. in Γ) to have base types smaller than the base type of a. The set $\mathrm{base}(\Gamma(\mathrm{fn}(A_i) \setminus \{a\}))$ indeed represents the base types of the names in the reaction context referenced in a migratable continuation A_i. In fact a is a name that needs to be in the scope of both the sender and the receiver at the same time, so it needs to be a common ancestor of sender and receiver in any $\mathcal{T}$-compatible forest. Any name in the reaction context and in the continuation of the receiver, with a base type smaller than the one of a, will be an ancestor of a—and hence of the sender, the receiver and the node representing the message—in any $\mathcal{T}$-compatible forest. Clearly, remembering a is not harmful as it must be already in the scope of receiver and sender, so we exclude it from the constraint.

Example 10.5. Take the normal form in Example 9.2. Let us fix $\mathcal{T}$ to be the forest $\mathsf{s} \prec \mathsf{c} \prec \mathsf{m} \prec \mathsf{d}$ and annotate the normal form with the following types:

$$s : \tau_s = \mathsf{s}[\tau_m] \qquad c : \tau_c = \mathsf{c}[\tau_m] \qquad m : \tau_m = \mathsf{m}[\mathsf{d}] \qquad d : \mathsf{d}$$

We want to prove $\emptyset \vdash_{\mathcal{T}} \nu s\, c.P$. We can apply rule Par: in this case there are no conditions on types because, being the environment empty, we have $\mathrm{base}(\emptyset(\mathrm{fn}(A))) = \emptyset$ for every active sequential term A of P. Let $\Gamma = \{(s : \tau_s), (c : \tau_c)\}$. The rule requires $\Gamma \vdash_{\mathcal{T}} !S$, $\Gamma \vdash_{\mathcal{T}} !C$ and $\Gamma \vdash_{\mathcal{T}} !M$, which can be proved by proving typability of S, C and M under Γ by rule Repl.

To prove $\Gamma \vdash_{\mathcal{T}} S$ we apply rule In; we have $s : \mathsf{s}[\tau_m] \in \Gamma$ and we need to prove that $\Gamma, x : \tau_m \vdash_{\mathcal{T}} \nu d.\overline{x}\langle d\rangle$. No constraints on base types are generated at this step since the migratable sequential term $\nu d.\overline{x}\langle d\rangle$ does not contain free variables typed by Γ making $\Gamma(\mathrm{fn}(\nu d.\overline{x}\langle d\rangle) \setminus \{a\}) = \Gamma(\{x\})$ empty. Next, $\Gamma, x : \tau_m \vdash_{\mathcal{T}} \nu d.\overline{x}\langle d\rangle$ can be proved by applying rule Par which amounts to checking $\Gamma, x : \tau_m, d : \mathsf{d} \vdash_{\mathcal{T}} \overline{x}\langle d\rangle.\mathbf{0}$ (by a simple

application of Out and the axiom $\Gamma, x : \tau_m, d : \mathsf{d} \vdash_{\mathcal{T}} \mathbf{0}$) and verifying the condition—true in $\mathcal{T}$—$\mathrm{base}(\tau_m) < \mathrm{base}(\tau_d)$: in fact d is tied to $\overline{x}\langle d\rangle$ and, for $\Gamma' = \Gamma \cup \{x : \tau_m\}$,

$$\mathrm{base}(\Gamma'(\mathrm{fn}(\overline{x}\langle d\rangle))) = \mathrm{base}(\Gamma'(\{x, d\})) = \mathrm{base}(\{\tau_m\}).$$

The proof for $\Gamma \vdash_{\mathcal{T}} M$ is similar and requires $\mathsf{c} < \mathsf{m}$ which is true in $\mathcal{T}$.

Finally, we can prove $\Gamma \vdash_{\mathcal{T}} C$ using rule In; both the two continuation $A_1 = \overline{s}\langle m\rangle$ and $A_2 = m(y).\overline{c}\langle m\rangle$ are migratable in C and since $\mathrm{base}(\tau_m) < \mathrm{base}(\tau_c)$ is false we need the other disjunct of the condition to be true. This amounts to checking that

$$\mathrm{base}(\Gamma(\mathrm{fn}(A_1) \setminus \{c\})) = \mathrm{base}(\Gamma(\{s, m\})) = \mathrm{base}(\{\tau_s\}) = \mathsf{s} < \mathsf{c}$$

(note $m \notin \mathrm{dom}(\Gamma)$) and $\mathrm{base}(\Gamma(\mathrm{fn}(A_a) \setminus \{c\})) = \mathrm{base}(\Gamma(\emptyset)) < \mathsf{c}$ (that holds trivially).

To complete the typing we need to show $\Gamma, m : \tau_m \vdash_{\mathcal{T}} A_1$ and $\Gamma, m : \tau_m \vdash_{\mathcal{T}} A_2$. The former can be proved by a simple application of Out which does not impose further constraints on $\mathcal{T}$. The latter is proved by applying In which requires $\mathrm{base}(\tau_c) < \mathsf{m}$, which holds in $\mathcal{T}$.

Note how, at every step, there is only one rule that applies to each subproof.

Example 10.6. There is no choice for (a finite) $\mathcal{T}$ that would make the normal form in Example 9.3 typeable. To see why, one can build the proof tree without assumptions on $\mathcal{T}$ obtaining that:

1. the restrictions must be annotated with types consistent with the type assignments

$$s : t_s[t] \qquad v : t_v[t] \qquad n : t_n[t] \qquad a : t \qquad b : t$$

2. $\mathcal{T}$ must satisfy the constraint that the base type assigned to b must be strictly greater than the one assigned to x, which is inconsistent with $s : t_s[t], b : t$.

10.4 Soundness

In this section we show how the type system can be used to prove depth-boundedness. Theorem 10.9 will show how typability is preserved by reduction. Theorem 10.10 establishes the main property of the type system: if a term is typable then $\mathcal{T}$-shapedness is invariant under reduction. This allows us to conclude that if a term is $\mathcal{T}$-shaped and typable, then every term reachable from it will be $\mathcal{T}$-shaped and, therefore, it is depth-bounded.

We start with some simple properties of the type system.

Lemma 10.7. *Let $P \in \mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$ and Γ, Γ' be type environments.*

a) if $\Gamma \vdash_{\mathcal{T}} P$ then $\mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)$;

b) if $\mathrm{dom}(\Gamma') \cap \mathrm{bn}(P) = \emptyset$ *and* $\mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)$, *then* $\Gamma \vdash_{\mathcal{T}} P$ *if and only if* $\Gamma\Gamma' \vdash_{\mathcal{T}} P$*;*

c) if $P \equiv P' \in \mathcal{P}_{\mathsf{nf}}^{\mathcal{T}}$ *then,* $\Gamma \vdash_{\mathcal{T}} P$ *if and only if* $\Gamma \vdash_{\mathcal{T}} P'$*.*

The subtitution lemma states that substituting names without altering the types preserves typability.

Lemma 10.8 (Substitution)**.** *Let* $P \in \mathcal{P}_{\mathsf{nf}}^{\mathcal{T}}$ *and* Γ *be a typing environment such that* $\Gamma(a) = \Gamma(b)$*. Then it holds that if* $\Gamma \vdash_{\mathcal{T}} P$ *then* $\Gamma \vdash_{\mathcal{T}} P[\,b/a\,]$*.*

Proof. We prove the lemma by induction on the structure of P. The base case is when $P \equiv \mathbf{0}$, where the claim trivially holds.

For the induction step, let $P \equiv \nu X.\prod_{i\in I} A_i$ with $A_i = \sum_{j\in J} \pi_{ij}.P_{ij}$, for some finite sets of indexes I and J. Since the presence of replication does not affect the typing proof, we can safely ignore that case as it follows the same argument. Let us assume $\Gamma \vdash_{\mathcal{T}} P$ and prove that $\Gamma \vdash_{\mathcal{T}} P[\,b/a\,]$.

From $\Gamma \vdash_{\mathcal{T}} P$, for each $i \in I$ and $x : \tau_x \in X$ we have

$$\Gamma, X \vdash_{\mathcal{T}} A_i \tag{10.1}$$

$$x \lhd_P i \implies \mathrm{base}(\Gamma(\mathrm{fn}(A_i))) < \mathrm{base}(\tau_x) \tag{10.2}$$

To extract from this assumptions a proof for $\Gamma \vdash_{\mathcal{T}} P[\,b/a\,]$, we need to prove that (10.1) and (10.2) hold after the substitution.

Since the substitution does not apply to names in X and the 'tied to' relation is only concerned with names in X, the only relevant effect of the substitution is modifying the set $\mathrm{fn}(A_i)$ to $\mathrm{fn}(A_i[\,b/a\,]) = \mathrm{fn}(A_i) \setminus \{a\} \cup \{b\}$ when $a \in \mathrm{fn}(A_i)$; But since $\Gamma(a) = \Gamma(b)$ by hypothesis, we have $\mathrm{base}(\Gamma(\mathrm{fn}(A_i[\,b/a\,]))) < \mathrm{base}(\tau_x)$.

It remains to prove (10.1) holds after the substitution as well. This amounts to prove for each $j \in J$ that $\Gamma' \vdash_{\mathcal{T}} \pi_{ij}.P_{ij} \implies \Gamma' \vdash_{\mathcal{T}} \pi_{ij}.P_{ij}[\,b/a\,]$, where $\Gamma' = \Gamma \cup X$; we prove this by cases.

Suppose $\pi_{ij} = \overline{\alpha}\langle\beta\rangle$ for two names α and β, then from $\Gamma' \vdash_{\mathcal{T}} \pi_{ij}.P_{ij}$ we know the following

$$\alpha : t_\alpha[\tau_\beta] \in \Gamma' \qquad \beta : \tau_\beta \in \Gamma' \tag{10.3}$$

$$\Gamma' \vdash_{\mathcal{T}} P_{ij} \tag{10.4}$$

Condition (10.3) is preserved after the substitution because it involves only types so, even if α or β are a, their types will be left untouched after they get substituted with b from the hypothesis that $\Gamma(a) = \Gamma(b)$. Condition (10.4) implies $\Gamma' \vdash_{\mathcal{T}} P_{ij}[\,b/a\,]$ by inductive hypothesis.

Suppose now that $\pi_{ij} = \alpha(x)$ and $P_{ij} \equiv \nu Y.\prod_{k\in K} A'_k$ for some finite set of indexes K; by hypothesis we have:

$$\alpha : t_\alpha[\tau_x] \in \Gamma' \tag{10.5}$$

$$\Gamma', x : \tau_x \vdash_{\mathcal{T}} P_{ij} \tag{10.6}$$

$$\mathrm{base}(\tau_x) < t_\alpha \vee \forall k \in K.\ \mathrm{Mig}_{\pi_{ij}.P_{ij}}(k) \implies \mathrm{base}(\Gamma'(\mathrm{fn}(A'_k) \setminus \{\alpha\})) < t_\alpha \tag{10.7}$$

Now x and Y are bound names so they are not altered by substitutions. The substitution $[\,b/a\,]$ can therefore only be affecting the truth of these conditions when $\alpha = a$ or when $a \in \mathrm{fn}(A'_k) \setminus (Y \cup \{x\})$. Since we know a and b are assigned the same type by Γ and $\Gamma \subseteq \Gamma'$, condition (10.5) still holds when substituting a for b. Condition (10.6) holds by inductive hypotesis. The first disjunct of condition (10.7) depends only on types, which are not changed by the substitution, so it holds after applying it if and only if it holds before the application. To see that the second disjunct also holds after the substitution we observe that the 'migratable' condition depends on x and $\mathrm{fn}(A'_k) \cap Y$ which are preserved by the substitution; moreover, if $a \in \mathrm{fn}(A'_k) \setminus \{\alpha\}$ then $\Gamma'(\mathrm{fn}(A'_k) \setminus \{\alpha\}) = \Gamma'(\mathrm{fn}(A'_k[\,b/a\,]) \setminus \{\alpha\})$.

This shows that the premises needed to derive $\Gamma', x : \tau'_x \vdash_{\mathcal{T}} \pi_{ij}.P_{ij}[\,b/a\,]$ are implied by our hypothesis, which completes the proof. □

Before we state the main theorem, we define the notion of P-safe type environment, which is a simple restriction on the types that can be assigned to names that are free at the top-level of a term.

Definition 10.4 (P-safe environment)**.** A type environment Γ is said to be *P-safe* if for each $x \in \mathrm{fn}(P)$ and $(y : \tau) \in \mathrm{bn}_\nu(P)$, $\mathrm{base}(\Gamma(x)) < \mathrm{base}(\tau)$.

Theorem 10.9 (Subject Reduction)**.** *Let P and Q be two terms in $\mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$ and Γ be a P-safe type environment. If $\Gamma \vdash_{\mathcal{T}} P$ and $P \to Q$, then $\Gamma \vdash_{\mathcal{T}} Q$.*

Proof. We will only prove the result for the case when $P \to Q$ is caused by a synchronising send and receive action since the $\boldsymbol{\tau}$ action case is similar and simpler. From $P \to Q$ we know that $P \equiv \nu W.(S \parallel R \parallel C) \in \mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$ with $S \equiv (\overline{a}\langle b\rangle.\nu Y_s.S') + M_s$ and $R \equiv (a(x).\nu Y_r.R') + M_r$ the synchronising sender and receiver respectively; $Q \equiv \nu W Y_s Y_r.(S' \parallel R'[\,b/x\,] \parallel C)$. In what follows, let $W' = W Y_s Y_r$, $C = \prod_{h\in H} C_h$, $S' = \prod_{i\in I} S'_i$ and $R' = \prod_{j\in J} R'_j$, all normal forms.

For annotated terms, the type system is syntax directed: there can be only one proof derivation for each typable term. By item 10.7.c, from the hypothesis $\Gamma \vdash_{\mathcal{T}} P$ we can deduce $\Gamma \vdash_{\mathcal{T}} \nu W.(S \parallel R \parallel C)$. The proof derivation for this typing judgment can only be of the following shape:

$$\frac{\Gamma W \vdash_{\mathcal{T}} S \quad \Gamma W \vdash_{\mathcal{T}} R \quad \forall h \in H.\, \Gamma W \vdash_{\mathcal{T}} C_h \quad \Psi}{\Gamma \vdash_{\mathcal{T}} \nu W.(S \parallel R \parallel C)} \tag{10.8}$$

where Ψ represents the rest of the conditions of the Par rule.[1] The fact that P is typable implies that each of these premises must be provable. The derivation proving $\Gamma, W \vdash_{\mathcal{T}} S$ must be of the form

$$\dfrac{\dfrac{a : t_a[\tau_b] \in \Gamma W \quad b : \tau_b \in \Gamma W \quad \Gamma W \vdash_{\mathcal{T}} \nu Y_s.S'}{\Gamma W \vdash_{\mathcal{T}} \overline{a}\langle b\rangle.\nu Y_s.S'} \quad \Psi_{M_s}}{\Gamma \vdash_{\mathcal{T}} \overline{a}\langle b\rangle.\nu Y_s.S' + M_s} \tag{10.9}$$

where $\Gamma W \vdash_{\mathcal{T}} \nu Y_s.S'$ is proved by an inference of the shape

$$\dfrac{\forall i \in I.\, \Gamma W Y_s \vdash_{\mathcal{T}} S'_i \qquad \forall i \in I.\, \Psi_{S'_i}}{\Gamma W \vdash_{\mathcal{T}} \nu Y_s.S'} \tag{10.10}$$

Analogously, $\Gamma W \vdash_{\mathcal{T}} R$ must be proved by an inference with the following shape

$$\dfrac{\dfrac{a : t_a[\tau_x] \in \Gamma W \quad \Gamma W, x : \tau_x \vdash_{\mathcal{T}} \nu Y_r.R' \quad \Psi_{R'}}{\Gamma W \vdash_{\mathcal{T}} a(x).\nu Y_r.R'} \quad \Psi_{M_r}}{\Gamma W \vdash_{\mathcal{T}} a(x).\nu Y_r.R' + M_r} \tag{10.11}$$

and to prove $\Gamma W, x : \tau_x \vdash_{\mathcal{T}} \nu Y_r.R'$

$$\dfrac{\forall j \in J.\, \Gamma W, x : \tau_x, Y_r \vdash_{\mathcal{T}} R'_j \qquad \forall j \in J.\, \Psi_{R'_j}}{\Gamma W, x : \tau_x \vdash_{\mathcal{T}} \nu Y_r.R'} \tag{10.12}$$

We have to show that from this hypothesis we can infer that $\Gamma \vdash_{\mathcal{T}} Q$ or, equivalently (by item 10.7.c), that $\Gamma \vdash_{\mathcal{T}} Q'$ where $Q' = \nu W Y_s Y_r.(S' \parallel R'[b/x] \parallel C)$. The derivation of this judgment can only end with an application of Par:

$$\dfrac{\forall i \in I.\, \Gamma W' \vdash_{\mathcal{T}} S'_i \quad \forall j \in J.\, \Gamma W' \vdash_{\mathcal{T}} R'_j[b/x] \quad \forall h \in H.\, \Gamma W' \vdash_{\mathcal{T}} C_h \quad \Psi'}{\Gamma \vdash_{\mathcal{T}} \nu W'.(S' \parallel R'[b/x] \parallel C)}$$

In what follows we show how we can infer these premises are provable as a consequence of the provability of the premises of the proof of $\Gamma \vdash_{\mathcal{T}} \nu W.(S \parallel R \parallel C)$.

From item 10.7.b and Name Uniqueness, $\Gamma W Y_s \vdash_{\mathcal{T}} S'_i$ from (10.10) implies $\Gamma W' \vdash_{\mathcal{T}} S'_i$ for each $i \in I$.

Let $\Gamma_r = \Gamma W, x : \tau_x$. We observe that by (10.9) and (10.11), $\tau_x = \tau_b$. From (10.11) we know that $\Gamma_r Y_r \vdash_{\mathcal{T}} R'_j$ which, by Lemma 10.8, implies $\Gamma_r Y_r \vdash_{\mathcal{T}} R'_j[b/x]$. By item 10.7.b we can infer $\Gamma_r Y_r Y_s \vdash_{\mathcal{T}} R'_j[b/x]$ and by applying the same lemma again using $\mathrm{fn}(R'_j[b/x]) \subseteq \mathrm{dom}(\Gamma W Y_r Y_s)$ and Name Uniqueness we obtain $\Gamma W' \vdash_{\mathcal{T}} R'_j[b/x]$.

Again applying item 10.7.b and Name Uniqueness, we have that $\Gamma W \vdash_{\mathcal{T}} C_h$ implies $\Gamma W' \vdash_{\mathcal{T}} C_h$ for each $h \in H$.

To complete the proof we only need to prove that for each $A \in \{S'_i \mid i \in I\} \cup \{R'_j \mid j \in J\} \cup \{C_h \mid h \in H\}$, $\Psi' = \forall (x : \tau_x) \in W'.\, x$ tied to A in $Q' \implies \mathrm{base}(\Gamma(\mathrm{fn}(A))) < \mathrm{base}(\tau_x)$ holds. This is trivially true by the hypothesis that Γ is P-safe. □

[1] Note that Ψ is trivially true by P-safety of Γ.

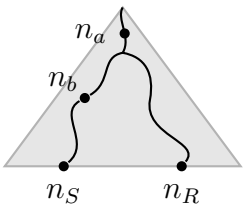
Theorem 10.10 (Invariance of $\mathcal{T}$-shapedness)**.** *Let P and Q be terms in $\mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$ such that $P \to Q$ and Γ be a P-safe environment such that $\Gamma \vdash_{\mathcal{T}} P$. Then, if P is $\mathcal{T}$-shaped then Q is $\mathcal{T}$-shaped.*

Proof. We will consider the input output synchronisation case as the τ action one is similar and simpler. We will further assume that the sending action $\overline{a}\langle b\rangle$ is such that $\nu(a:\tau_a)$ and $\nu(b:\tau_b)$ are both active restrictions of P, i.e. $(a:\tau_a) \in W$, $(b:\tau_b) \in W$ with $P \equiv \nu W.(S \parallel R \parallel C)$. The case when any of these two names is a free name of P can be easily handled with the aid of the assumption that Γ is P-safe.

As in the proof of Theorem 10.9, the derivation of $\Gamma \vdash_{\mathcal{T}} P$ must follow the shape of (10.8).

From $\mathcal{T}$-shapedness of P we can conclude that both $\nu Y_s.S'$ and $\nu Y_r.R'$ are $\mathcal{T}$-shaped. We note that substitutions do not affect $\mathcal{T}$-compatibility since they do not alter the set of bound names and their type annotations. Therefore, we can infer that $\nu Y_r.R'[\,b/a\,]$ is $\mathcal{T}$-shaped. By Lemma 10.5 we know that $\varphi = \Phi(\nu W.(S \parallel R \parallel C)) \in \mathcal{F}[\![P]\!]$, $\varphi_r = \Phi(\nu Y_r.R'[\,b/a\,]) \in \mathcal{F}[\![\nu Y_r.R'[\,b/x\,]]\!]$ and $\varphi_s = \Phi(\nu Y_s.S') \in \mathcal{F}[\![\nu Y_s.S']\!]$. Let $\varphi_r = \varphi_{\text{mig}} \uplus \varphi_{\neg\text{mig}}$ where only φ_{mig} contains a leaf labelled with a term with b as a free name. These leaves will correspond to the continuations R'_j that migrate in $a(x).\nu Y_r.R'$, after the application of the substitution $[\,b/x\,]$. By assumption, inside P both S and R are in the scope of the restriction bounding a and S must also be in the scope of the restriction bounding b. Let $t_a = \text{base}(\tau_a)$ and $t_b = \text{base}(\tau_b)$, φ will contain two leaves n_S and n_R labelled with S and R respectively, having a common ancestor n_a labelled with (a, t_a); n_S will have an ancestor n_b labelled with (b, t_b). Let p_a, p_S and p_R be the paths in φ leading from a root to n_a, n_S and n_R respectively. By $\mathcal{T}$-compatibility of φ, we are left with only two possible cases: either 1) $t_a < t_b$ or 2) $t_b < t_a$.

Let us consider case 1) first. The tree in φ to which the nodes n_S and n_R belong, would have the following shape:



Now, we want to transform φ, by manipulating this tree, into a forest φ' that is $\mathcal{T}$-compatible by construction and such that there exists a term $Q' \equiv Q$ with $\text{forest}(Q') = \varphi'$, so that we can conclude Q is $\mathcal{T}$-shaped.

To do so, we introduce the following function, taking a labelled forest φ, a path p in φ and a labelled forest ρ and returning a labelled forest:

$$\text{ins}(\varphi, p, \rho) := (N_\varphi \uplus N_\rho, \prec_\varphi \uplus \prec_\rho \uplus \prec_{\text{ins}}, \ell_\varphi \uplus \ell_\rho)$$

where $n \prec_{\text{ins}} n'$ if $n' \in \min_{\prec_\rho}(N_\rho)$ and

$$\begin{aligned} \ell_\rho(n') = (y, t_y) &\implies n \in \max_{\prec_\varphi} \{m \in p \mid \ell_\varphi(m) = (x, t_x), t_x < t_y\} \\ \ell_\rho(n') = A &\implies n \in \max_{\prec_\varphi} \{m \in p \mid \ell_\varphi(m) = (x, t_x), x \in \text{fn}(A)\} \end{aligned}$$

Note that for each n', since p is a path, there can be at most one n such that $n \prec_{\text{ins}} n'$.

To obtain the desired φ', we first need to remove the leaves n_S and n_R from φ, as they represent the sequential processes which reacted, obtaining a forest φ_C. We argue that the φ' we need is indeed

$$\begin{aligned} \varphi' &= \text{ins}(\varphi_1, p_S, \varphi_{\text{mig}}) \\ \varphi_1 &= \text{ins}(\varphi_2, p_R, \varphi_{\neg\text{mig}}) \\ \varphi_2 &= \text{ins}(\varphi_C, p_S, \varphi_s) \end{aligned}$$

It is easy to see that, by definition of ins, φ' is $\mathcal{T}$-compatible: φ_C, φ_s, $\varphi_{\neg\text{mig}}$ and φ_{mig} are $\mathcal{T}$-compatible by hypothesis, ins adds parent-edges only when they do not break $\mathcal{T}$-compatibility.

To prove the claim we need to show that φ' is the forest of a term congruent to $\nu W Y_s Y_r.(S' \parallel R'[\,b/x\,] \parallel C)$. Let $R' = \prod_{j \in J} R'_j$, $J_{\text{mig}} = \{j \in J \mid x \lhd_{\nu Y_r.R'} j\}$, $J_{\neg\text{mig}} = J \setminus J_{\text{mig}}$ and $Y'_r = \{(x : \tau) \in Y_r \mid x \in \text{fn}(R'_j), j \in J_{\neg\text{mig}}\}$. We know that no R'_j with $j \in J_{\neg\text{mig}}$ can contain x as a free name so $R'_j[\,b/x\,] = R'_j$. Now suppose we are able to prove that conditions 9.1.i, 9.1.ii and 9.1.iii of Lemma 9.1 hold for φ_C, φ_1, φ_2 and φ'. Then we could use Lemma 9.1 to prove

a) $\varphi_C = \text{forest}(Q_C), Q_C \equiv Q_{\varphi_C} = \nu W.C$,

b) $\varphi_2 = \text{forest}(Q_2), Q_2 \equiv Q_{\varphi_2} = \nu W Y_s.(S' \parallel C)$,

c) $\varphi_1 = \text{forest}(Q_1), Q_1 \equiv Q_{\varphi_1} = \nu W Y_s Y'_r.(S' \parallel \prod_{j \in J_{\neg\text{mig}}} R'_j \parallel C)$,

d) $\varphi' = \text{forest}(Q'), Q' \equiv Q_{\varphi'} = \nu W Y_s Y_r.(S' \parallel R'[\,b/x\,] \parallel C) \equiv Q$

(it is straightforward to check that $\varphi_C, \varphi_2, \varphi_1$ and φ' have the right sets of nodes and labels to give rise to the right terms). We then proceed to check for each of the forests above that they satisfy conditions 9.1.i, 9.1.ii and 9.1.iii, thus proving the theorem.

Condition 9.1.i requires that only leaves are labelled with sequential processes, condition that is easily satisfied by all of the above forests since none of the operations involved in their definition alters this property and the forests φ, φ_s and φ_r satisfy it by construction.

Similarly, since $\nu W.(S \parallel R \parallel C)$ is a normal form it satisfies Name Uniqueness, condition 9.1.ii is satisfied as we never use the same name more than once.

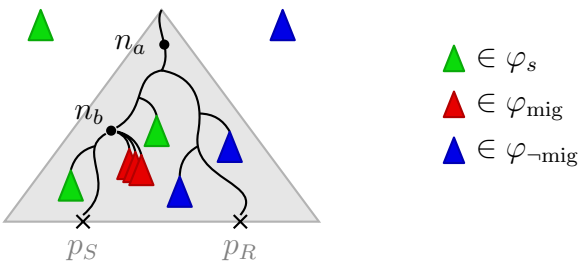
Condition 9.1.iii holds on φ and hence it holds on φ_C since the latter contains all the nodes of φ labelled with names.

Now consider φ_s: in the proof of Theorem 10.9 we established that $\Gamma \vdash_{\mathcal{T}} P$ implies that the premises $\Psi_{S_i'}$ from (10.10) hold, that is $\mathrm{base}(\Gamma W(\mathrm{fn}(S_i'))) < \mathrm{base}(\tau_x)$ holds for all S_i' for $i \in I$ and all $(x:\tau_x) \in Y_s$ such that $x \lhd_{\nu Y_s.S'} i$. Since $\mathrm{fn}(S_i') \cap W \subseteq \mathrm{fn}(S')$ we know that every name $(w:\tau_w) \in W$ such that $w \in \mathrm{fn}(S_i')$ will appear as a label $(w, \mathrm{base}(\tau_w))$ of a node n_w in p_S. Therefore, by definition of ins, we have that for each $n \in N_{\varphi_C}$, $n_w <_{\varphi_2} n$; in other words, in φ_2, every leaf in N_{φ_s} labelled with S_i' is a descendent of a node labelled with $(w, \mathrm{base}(\tau_w))$ for each $(w:\tau_w) \in W$ with $w \in \mathrm{fn}(S_i')$. This verifies condition 9.1.iii on φ_2.

Similarly, by (10.12) the following premise must hold: $\mathrm{base}(\Gamma W(\mathrm{fn}(R_j'))) < \mathrm{base}(\tau_x)$ for all R_j' for $j \in J$ and all $(y:\tau_y) \in Y_r$ such that $y \lhd_{\nu Y_r.R'} j$. We can then apply the same argument we applied to φ_2 to show that condition 9.1.iii holds on φ_1.

From (10.11) and the assumption $t_a < t_b$, we can conclude that the following premise must hold: $\mathrm{base}(\Gamma W(\mathrm{fn}(R_j') \setminus \{a\})) < t_a$ for each $j \in J$ such that R_j' is migratable in $a(x).\nu Y_r.R'$, i.e $j \in J_{\mathrm{mig}}$. From this we can conclude that for every name $(w:\tau_w) \in W$ such that $w \in \mathrm{fn}(R_j'[\,b/x\,])$ with $j \in J_{\mathrm{mig}}$ there must be a node in p_a (and hence in p_S) labelled with $(w, \mathrm{base}(\tau_w))$. Now, some of the leaves in φ_{mig} will be labelled with terms having b as a free name; we show that in fact every node in φ_{mig} labelled with a (y, t_y) is indeed such that $t_y < t_b$. From the proof of Theorem 10.9 and Lemma 10.8 we know that from the hypothesis we can infer that $\Gamma W \vdash_{\mathcal{T}} \nu Y_r.R'[\,b/x\,]$ and hence that for each $j \in J_{\mathrm{mig}}$ and each $(y:\tau_y) \in Y_r$, if y is tied to $R_j'[\,b/x\,]$ in $\nu Y_r.R'[\,b/x\,]$ then $\mathrm{base}(\Gamma W(R_j'[\,b/x\,])) < \mathrm{base}(\tau_y)$. By Lemma 10.6 we know that every root of φ_{mig} is labelled with a name (y, t_y) which is tied to each of the leaves in its tree. Therefore each such t_y satisfies $\mathrm{base}(\Gamma W(R_j'[\,b/x\,])) < t_y$. By construction, there exists at least one $j \in J_{\mathrm{mig}}$ such that $x \in \mathrm{fn}(R_j')$ and consequently such that $b \in \mathrm{fn}(R_j'[\,b/x\,])$. From this and $b \in W$ we can conclude $t_b < t_y$ for t_y labelling a root in φ_{mig}. We can then conclude that $\{n_b\} = \max_{\prec_{\varphi_2}} \{m \in p_S \mid \ell_\varphi(m) = (z, t_z), t_z < t_y\}$ for each t_y labelling a root of φ_{mig}, which means that each tree of φ_{mig} is placed as a subtree of n_b in φ'. This verifies condition 9.1.iii for φ' completing the proof.

Pictorially, the tree containing n_S and n_R in φ is now transformed in the following tree in φ':



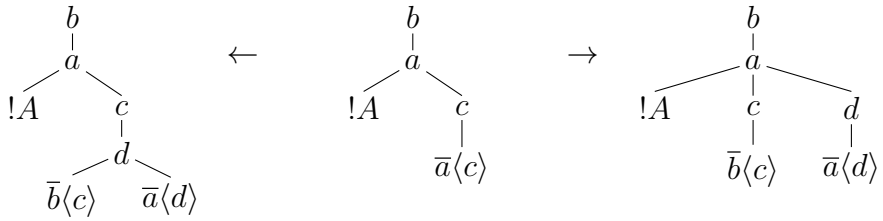
Case 2) — where $t_b < t_a$ — is simpler as the migrating continuations can be treated just as the non-migrating ones. □

To illustrate the role of φ_{mig}, $\varphi_{\neg\mathrm{mig}}$ and the ins operation in the above proof, we show an example that would not be typable if we choose a simpler "migration" transformation.

Example 10.7. Consider the normal form $P = \nu a\, b\, c.(!A \parallel \overline{a}\langle c\rangle)$ where

$$A = a(x).\nu d.\big(\overline{a}\langle d\rangle \parallel \overline{b}\langle x\rangle\big).$$

To make types consistent we need annotations satisfying $a : t_a[t]$, $b : t_b[t]$, $c : t$ and $d : t$. Any $\mathcal{T}$ satisfying the constraints $t_b < t_a < t$ would allow us to prove $\emptyset \vdash_{\mathcal{T}} P$; let then $\mathcal{T}$ be the forest with $\mathsf{b} \prec \mathsf{a} \prec \mathsf{t}$ with $t_a = \mathsf{a}$, $t_b = \mathsf{b}$ and $t = \mathsf{t}$. Let $P' = \nu abcd.(!A \parallel \overline{a}\langle d\rangle \parallel \overline{b}\langle c\rangle)$ be the (only) successor of P. The following picture shows $\Phi(P)$ in the middle, on the left a forest in $\mathcal{F}[\![P']\!]$ extracted by just putting the continuation of A under the message, on the right the forest obtained by using ins on the non-migrating continuations of A:



Clearly, the tree on the left is not $\mathcal{T}$-compatible since c and d have the same base type t. Instead, the tree on the right can be obtained because ins inserts the non-migrating continuation as close to the root as possible.

Definition 10.5 (Typably Hierarchical term)**.** A normal form P is *typably hierarchical* if P is $\mathcal{T}$-shaped and $\Gamma \vdash_{\mathcal{T}} P$ for some finite forest $\mathcal{T}$ and P-safe environment Γ. A general π-term P is typably hierarchical if its normal form $\mathrm{nf}(P)$ is.

Theorem 10.11 (Depth-boundedness)**.** *Every typably hierarchical term is depth-bounded.*

Proof. By Theorem 10.9 and Theorem 10.10 every term reachable from a typably hierarchical term P is $\mathcal{T}$-shaped. Then by Proposition 10.4 P is depth-bounded. □

10.5 Type Inference

In this section we will show that it is possible to take any non-annotated normal form P and derive a forest $\mathcal{T}$ and an annotated version of P that can be typed under $\mathcal{T}$.

Inference for simple types has already been proved decidable in [Gay93; VH93]. In our case, since our types are not recursive, the algorithm concerned purely with the constraints imposed by the type system of the form $\tau_x = t[\tau_y]$ is even simpler. The main difficulty is in inferring the structure of $\mathcal{T}$.

Let us first be more specific on assigning simple types. The number of ways a term P can be annotated with types are infinite, simply from the fact that types allow an arbitrary nesting as in t, $t[t]$, $t[t[t]]$ and so on. We observe that, however, there is no use annotating a restriction with a type with nesting deeper than the size of the program: the type system cannot inspect more deeply nested types. Thanks to this observation we can restrict ourselves to annotations with bounded nesting in the types structure. This also gives a bound on the number of base types that need to appear in the annotated term. Therefore, there are only finitely many possible annotations and possible forests under which P can be proved typably hierarchical. A naïve inference algorithm can then enumerate all of them and type check each.

Theorem 10.12 (Decidability of Inference)**.** *Given a normal form $P \in \mathcal{P}_{\mathsf{nf}}$, it is decidable if there exists a finite forest $\mathcal{T}$, a $\mathcal{T}$-annotated version $P' \in \mathcal{P}^{\mathcal{T}}$ of P and a P'-safe environment Γ such that P' is $\mathcal{T}$-shaped and $\Gamma \vdash_{\mathcal{T}} P'$.*

While enumerating all the relevant forests, annotations and environments is impractical, more clever strategies for inference exist.

We start by annotating the term with type variables: each name x gets typed with a type variable τ_x. Then we start the type derivation, collecting all the constraints on types along the way. If we can find a $\mathcal{T}$ and type expressions to associate to each type variable, so that these constraints are satisfied, the process can be typed under $\mathcal{T}$.

By inspecting rules Par and In we observe that all the "tied-to" and "migratable" predicates do not depend on $\mathcal{T}$ so for any given P, the type constraints can be expressed simply by conjuctions and disjuctions of two kinds of basic predicates:

1. *data-flow constraints* of the form $\tau_x = t_x[\tau_y]$ where t_x is a base type variable;
2. *base type constraints* of the form $\mathrm{base}(\tau_x) < \mathrm{base}(\tau_y)$ which correspond to constraints over the corresponding base type variables, e.g. $t_x < t_y$.

Note that the P-safety condition on Γ translates to constraints of the second kind. The first kind of constraints can be solved using unification in linear time. If no solution exists, the process cannot be typed. This is the case of processes that cannot be *simply typed*. If unification is successful we get a set of equations over base type variables. Any assignment of those variables to nodes in a suitable forest that satisfies the constraints of the second kind would be a witness of typability.

Example 10.8. Take the term $\nu s\, c.P$ of Example 9.2 recalling the definition (renaming some variables to disambiguate)

$$P = !S \parallel !C \parallel !M \qquad S = s(z).\nu d.\overline{z}\langle d\rangle$$
$$C = c(x).(\overline{s}\langle x\rangle \parallel x(y).\overline{c}\langle x\rangle) \qquad M = \boldsymbol{\tau}.\nu m.\overline{c}\langle m\rangle$$

We start by annotating each restriction νa with a fresh type variable $\nu(a : \tau_a)$. Then we perform a type derivation as in Example 10.5, obtaining the following data-flow constraints:

$$\tau_s = t_s[\tau_z] \qquad \tau_c = t_c[\tau_x] \qquad \tau_x = t_x[\tau_y] \qquad \tau_z = \tau_x = \tau_m = t_z[\tau_d]$$

from which we learn that:

- τ_d is unconstrained; we use the base type variable t_d for $\mathrm{base}(\tau_d)$;
- $\tau_y = \tau_d$;
- $t_x = t_z$ and $\mathrm{base}(\tau_m) = t_x$.

We can therefore completely specify the types just by associating t_s, t_c, t_x and t_d to nodes in a forest: all the types would be determined as a consequence of the data-flow constraints, apart from τ_d to which we can safely assign the type t_d.

During the type derivation we also collected the following base type constraints:

$$\begin{gathered}\mathrm{base}(\tau_z) < \mathrm{base}(\tau_d) \\ \mathrm{base}(\tau_c) < \mathrm{base}(\tau_m) \\ \mathrm{base}(\tau_c) < \mathrm{base}(\tau_x) \\ \mathrm{base}(\tau_x) < \mathrm{base}(\tau_c) \vee \mathrm{base}(\tau_s) < \mathrm{base}(\tau_c)\end{gathered}$$

These can be simplified and normalised using the equations on types seen above obtaining the set

$$\mathcal{C}_{\nu s\, c.P} = \{t_x < t_c \vee t_s < t_c, t_c < t_x, t_x < t_d\}$$

Hence any choice of $\mathcal{T} \supseteq \{t_x, t_c, t_s, t_d\}$ such that $t_s <_{\mathcal{T}} t_c <_{\mathcal{T}} t_x <_{\mathcal{T}} t_d$ would make the typing succeed.

Henceforth we refer to $\mathcal{C}_P$ as the set of base types constraints of the type derivation for P, modulo the equalities extracted from the data-flow equations.

We sketch a few approaches that can be followed to solve the base type constraints.

A Direct algorithm

Suppose we are presented with a set $\mathcal{C}$ of constraints of the form $t < t'$ (no disjuctions). If the constraints are acyclic, i.e. it is not possible to derive $t < t$ by transitivity, then there exists a finite forest satisfying the constraints, having as nodes the base type variables. To construct such forest, we can first represent the constraints as a graph with the base type variables as vertices and an edge between t and t' just when $t < t' \in \mathcal{C}$. Then we can check the graph for acyclicity. If the test fails, the constraints are unsatisfiable. Otherwise, any topological sort of the graph will represent a forest satisfying $\mathcal{C}$.

We can modify this simple procedure to support constraints including disjuctions by using backtracking on the disjuncts. Every time we arrive at an acyclic assigment, we can check for $\mathcal{T}$-shapedness (which takes linear time) and in case the check fails we can backtrack again.

Reduction to CLP(FD)

As we noted, if there exists a solution, there exists a solution where $\mathcal{T}$ is a linear chain of nodes. We can exploit this fact to give an encoding of the problem as a *Constraint Satisfaction Problem* over a finite domain, or CLP(FD).

Suppose $\mathcal{C}_P$ has n variables. We can assign to each variable t_x the finite domain of integers $\{1, \ldots, n\}$, and interpret the $t_x < t_y$ constraints in $\mathcal{C}_P$ as integer constraints. A consistent assignment of integers to the variables would correspond to letting $\mathcal{T} = \{1, \ldots, n\}$ and annotating P so that all the constraints are satisfied.

To complete the procedure, each valid assignment can be checked for $\mathcal{T}$-shapedness.

Reduction to SAT

A third strategy to solve the constraints in $\mathcal{C}_P$ is to encode the problem as a SAT instance. We have at most n base type variables where n is the number of names occurring in P. There are at most $\frac{n(n-1)}{2}$ distinct independent constraints of the form $t_x < t_y$, which can be treated as uninterpreted propositions.

The encoding can be organised in several ways. One possibility is to encode the full problem, satisfiability of $\mathcal{C}_P$ *and* $\mathcal{T}$-shapedness of P. To ensure that the constraints encode a tree, one needs to add transitivity of $<$ and acyclicity can be expressed by asserting $\neg(t_x < t_x)$.

Another option is to solve only a sub-problem using SAT techniques and deal with the rest directly. For example, one can observe that the constraints in $\mathcal{C}_P$ forms a 2-CNF formula with $O(n^2)$ boolean variables. Since 2-CNF satisfiability is linear in the number of variables [APT79], we obtain a $O(n^2)$ bound on satisfiability of these base type constraints. Once we prove satisfiability of these constraints, to prove P is typably hierarchical, it remains to prove that there exists a model $\mathcal{T}$ of the constraints so that P is $\mathcal{T}$-shaped. If a precise bound on the depth is needed, one can perform a search for the shallowest forest which is a model of the base type constraints such that P is $\mathcal{T}$-shaped. Otherwise, the search can be restricted to total orders. The solutions to a 2-SAT problem with c constraints can be enumerated using $O(c)$ preprocessing time, $O(d)$ time per solution (where d is the degree of the problem, i.e. the maximum number of constraints involving the same proposition) and $O(c)$ space [Fed94]. Additional 2-CNF constraints that can be added to the problem instance to prune the search space are the ones stating that $t_x < t_y \vee t_y < t_x$ when $x, y \in \mathrm{fn}(A)$ for any sequential subterm A of P, which are necessary conditions for $\mathcal{T}$-shapedness.

In addition to the decision problem 'is P typably hierarchical?', there is an associated optimisation problem asking for the smallest height of the forests $\mathcal{T}$ which witness that P is typably hierarchical. The precise complexity bounds of both problems are open.

10.6 Polyadic and Global Channels

The type system of Section 10.3 is defined for unary channels but extending it for the more general case of polyadic channels is very natural. A polyadic channel is a name on which one can send a tuple of names instead of a single one as a message. Input and output actions become $a(x_1, \ldots, x_n)$ and $\overline{a}\langle b_1, \ldots, b_n\rangle$ respectively, in which case we say that they have arity n. In the special case when $n = 0$, there is no exchange of messages but just a CCS-style synchronisation.

Types take the generalised form

$$\tau ::= t[\tau_1, \ldots, \tau_n]$$

where $\mathrm{base}(t[\tau_1, \ldots, \tau_n]) = t$. The rules of the type system affected by the change are Out and In. The conditions on types become the expected ones, checking that the number of arguments of actions match the type of the channel. The constraints on base types in rule In are simply checked for each of the input's arguments and the notion of migratable is changed so that a sequential subterm is migratable if it is tied to any of the input's arguments.

Since in the rest of the dissertation we will make occasional use of zero-arity channels, let us specify formally how these can be handled. A zero-arity channel can be encoded using unary channels by communicating a dummy message that is discarded by the receiver: assuming $x \notin \mathrm{fn}(P)$, $a.P = a(x).P$ and $\overline{a}.P = \nu x.(\overline{a}\langle x\rangle.P)$. In Figure 10.3 we give a specialisation of rules In and Out on zero-arity channel prefixes. Since no message is exchanged, none of the continuations of the receiver is migratable which is why rule In-0 becomes trivial.

$$\frac{a : \tau \in \Gamma \qquad \Gamma \vdash_{\mathcal{T}} Q}{\Gamma \vdash_{\mathcal{T}} \overline{a}.Q}\ \textsc{Out-0} \qquad\qquad \frac{a : \tau \in \Gamma \qquad \Gamma \vdash_{\mathcal{T}} \nu X.\prod_{i\in I} A_i}{\Gamma \vdash_{\mathcal{T}} a.\nu X.\prod_{i\in I} A_i}\ \textsc{In-0}$$

Figure 10.3 – Zero-arity specialisation of prefix rules.

Theorems 10.9 and 10.10 make the assumption that Γ is P-safe, assumption that is exploited in Lemma 10.8. It is important to note that only the environment giving types to globally free names is required to be P-safe: the environments used inside a typability

proof need not be safe for the subterms involved in each step. The P-safe assumption is a mainly technical device used to side-step the fact that an important part of the check for consistency of types is done in the Par rule, which would not apply to names that are free at the top-level. The solution offered by P-safety conceptually coincides with performing type-checking/inference on $\nu\,\mathrm{fn}(P).P$.

A more expressive, but more fiddly, solution is to include a 'global' base type $\mathsf{gl} \in \mathcal{T}$ (with no parent nor children). The rules of the type system are changed to only require a condition $t < t'$ if t and t' are both not equal to gl- Naturally, no restricted name can have gl as base type, so the Par rule needs to include the condition $\mathsf{gl} \notin \mathrm{base}(X)$. The rationale behind this solution is that the $<_{\mathcal{T}}$ relation is mainly used to make sure certain names are in the scope when migrating terms during a reaction; globally free names are however always available and, crucially, cannot be replicated, therefore whenever a term is migrated, any reference to a global name is unambiguous.

To see why using gl is a better solution than P-safety, consider the term

$$P = a(x).b(y).(\overline{a}\langle y\rangle \parallel \overline{b}\langle x\rangle) \parallel \nu(c:t).\overline{a}\langle c\rangle$$

and the proof for $\Gamma \vdash_{\mathcal{T}} P$, with $\Gamma = [\, a \mapsto t_a[t],\ b \mapsto t_b[t]\,]$. P-safety of Γ requires $t_a < t$ and $t_b < t$ since c is a bound name and a and b are gloablly free names. Rule In would require $t_b < t_a$ because the only continuation is migratable and has b as free name. However, in discharging the sub-proof for $b(y).(\overline{a}\langle y\rangle \parallel \overline{b}\langle x\rangle)$, rule In would also require $t_a < t_b$ since $\overline{a}\langle y\rangle$ is migratable. Therefore the typing fails for any $\mathcal{T}$, unless we assing $t_a = t_b = \mathsf{gl}$, obtaining that the term can be safely typed.

Special handling for global names is expecially useful when used in conjunction with Remark 9.1: process names can be seen as global names and typing them with gl would avoid putting unnecessary constraints on the definition of recursive behaviour.

Despite its benefits, we avoided introducing this complication in the presentation of the type system, in favour of the P-safe solution, to unclutter the proofs of Section 10.3. For a generalisation of the 'global' base type idea, using multiplicities, see Section 12.1.

10.7 Some examples

A server with workers

We illustrate the use of the type system using the example Erlang program of Section 1.1. Below, we give a simple π-term modelling the essential features of the program. We abstract away aspects like the data stored by the database and focus on the communication protocol between the server, the workers, the client and the database. The `main` function,

which bootstraps the system, is represented by the term M:

$$
\begin{aligned}
M &= \nu DB.\nu visit.\nu ping.(D \parallel !L \parallel !S \parallel !(\boldsymbol{\tau}.\nu c.\nu reply.(\overline{c} \parallel !C))) \\
S &= ping(pong).\overline{pong} + visit(reply').(\nu ready.\nu val.W) \\
C &= c.(\overline{visit}\langle reply\rangle \parallel reply.\overline{c}) \\
W &= \overline{lock}\langle ready\rangle \parallel ready.(\overline{read}\langle val\rangle \parallel val.\overline{write}.\overline{unlock}.\overline{reply'}) \\
D &= lock(ready').(\overline{ready'} \parallel \overline{db}) \\
L &= db.(read(val').(\overline{val'} \parallel \overline{db}) + write.\overline{db} + unlock.D)
\end{aligned}
$$

where $DB = \{db, lock, read, write, unlock\}$. The terms S, C, W and D represent the server, client, worker and database processes respectively, while L represents the database in the locked state. A single instance of a database is identified by the names in DB which encode the different kinds of messages handled by that instance of the database. An output action db (resp. c) stands for a call to the tail-recursive `database` (resp. `client`) function. Similarly, $ready$ and val express the identity of a worker process, $visit$ and $ping$ the one of the server process.

The term M is in normal form and has no free names. Some names are primed to satisfy Name Uniqueness but they will be bound, during execution, to instances of the corresponding un-primed restricted names. To perform type inference, we assume each name x is annotated with a type variable τ_x with $\mathrm{base}(\tau_x) = t_x$.

By setting $\Gamma = \{(x : \tau_x) \mid x \in DB\} \cup \{(visit : \tau_{visit}), (ping : \tau_{ping})\}$

$$
\dfrac{\Gamma \vdash_{\mathcal{T}} D \qquad \dfrac{\Gamma \vdash_{\mathcal{T}} L}{\Gamma \vdash_{\mathcal{T}} !L} \qquad \dfrac{\Gamma \vdash_{\mathcal{T}} S}{\Gamma \vdash_{\mathcal{T}} !S} \qquad \dfrac{\dfrac{\Gamma \vdash_{\mathcal{T}} \nu c.\nu reply.(\overline{c} \parallel !C)}{\Gamma \vdash_{\mathcal{T}} \boldsymbol{\tau}.\nu c.\nu reply.(\overline{c} \parallel !C)}\ \textsc{Tau}}{\Gamma \vdash_{\mathcal{T}} !(\boldsymbol{\tau}.\nu c.\nu reply.(\overline{c} \parallel !C))}}{\emptyset \vdash_{\mathcal{T}} M}\ \textsc{Par}
$$

Note that since the environment is empty, the constraint on base types becomes trivial.

To prove $\Gamma \vdash_{\mathcal{T}} D$ we have

$$
\dfrac{\tau_{lock} = t_{lock}[\tau_{ready'}] \qquad \dfrac{\dfrac{\tau_{ready'} = t_{ready'} \qquad \overline{\Gamma_1 \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma_1 \vdash_{\mathcal{T}} \overline{ready'}.\mathbf{0}} \qquad \dfrac{\tau_{db} = t_{db} \qquad \overline{\Gamma_1 \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma_1 \vdash_{\mathcal{T}} \overline{db}.\mathbf{0}}}{\Gamma_1 \vdash_{\mathcal{T}} \overline{ready'} \parallel \overline{db}}}{\Gamma \vdash_{\mathcal{T}} lock(ready').(\overline{ready'} \parallel \overline{db})}\ \textsc{In}
$$

where $\Gamma_1 = \Gamma \cup \{(ready' : \tau_{ready'})\}$. Again, the conditions on base types are trivially true since only $\overline{ready'}$ is migratable and it contains no name from the environment Γ.

The proof for $\Gamma \vdash_{\mathcal{T}} L$ starts with an application of rule In-0:

$$\dfrac{\tau_{db} = t_{db} \qquad \dfrac{\Gamma \vdash_{\mathcal{T}} \mathit{read}(\mathit{val}').(\overline{\mathit{val}'} \parallel \overline{\mathit{db}}) \qquad \Gamma \vdash_{\mathcal{T}} \mathit{write}.\overline{\mathit{db}} \qquad \Gamma \vdash_{\mathcal{T}} \mathit{unlock}.D}{\Gamma \vdash_{\mathcal{T}} \mathit{read}(\mathit{val}').(\overline{\mathit{val}'} \parallel \overline{\mathit{db}}) + \mathit{write}.\overline{\mathit{db}} + \mathit{unlock}.D}}{\Gamma \vdash_{\mathcal{T}} \mathit{db}.(\mathit{read}(\mathit{val}').(\overline{\mathit{val}'} \parallel \overline{\mathit{db}}) + \mathit{write}.\overline{\mathit{db}} + \mathit{unlock}.D)}$$

which can be completed with the following proofs

$$\dfrac{\tau_{unlock} = t_{unlock} \qquad \Gamma \vdash_{\mathcal{T}} D}{\Gamma \vdash_{\mathcal{T}} \mathit{unlock}.D} \qquad\qquad \dfrac{\tau_{write} = t_{write} \qquad \dfrac{\tau_{db} = t_{db} \qquad \overline{\Gamma \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma \vdash_{\mathcal{T}} \overline{\mathit{db}}}}{\Gamma \vdash_{\mathcal{T}} \mathit{write}.\overline{\mathit{db}}}$$

where $\Gamma \vdash_{\mathcal{T}} D$ has the same proof as above, and

$$\dfrac{\tau_{read} = t_{read}[\tau_{val'}] \qquad \dfrac{\dfrac{\tau_{val'} = t_{val'} \qquad \overline{\Gamma_2 \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma_2 \vdash_{\mathcal{T}} \overline{\mathit{val}'}} \qquad \dfrac{\tau_{db} = t_{db} \qquad \overline{\Gamma_2 \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma_2 \vdash_{\mathcal{T}} \overline{\mathit{db}}}}{\Gamma_2 \vdash_{\mathcal{T}} \overline{\mathit{val}'} \parallel \overline{\mathit{db}}}}{\Gamma \vdash_{\mathcal{T}} \mathit{read}(\mathit{val}').(\overline{\mathit{val}'} \parallel \overline{\mathit{db}})}\ \textsc{In}$$

where $\Gamma_2 = \Gamma \cup \{(\mathit{val}' : \tau_{val'})\}$. As in the previous application of rule In, the base constraints are trivial because the only migrating process is $\overline{\mathit{val}'}$; this reflects the fact that the simple pattern of asynchronously acknowledging receipt of a message is not a threat for depth boundedness.

Let us turn to the proof for $\Gamma \vdash_{\mathcal{T}} S$:

$$\dfrac{\dfrac{\tau_{ping} = t_{ping}[\tau_{pong}] \qquad \dfrac{\tau_{pong} = t_{pong} \qquad \overline{\Gamma_3 \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma_3 \vdash_{\mathcal{T}} \overline{\mathit{pong}}.\mathbf{0}}}{\Gamma \vdash_{\mathcal{T}} \mathit{ping}(\mathit{pong}).\overline{\mathit{pong}}.\mathbf{0}} \qquad \Gamma \vdash_{\mathcal{T}} \mathit{visit}(\mathit{reply}').(\boldsymbol{\nu}\mathit{ready}.\boldsymbol{\nu}\mathit{val}.W)}{\Gamma \vdash_{\mathcal{T}} S}$$

where $\Gamma_3 = \Gamma \cup \{(\mathit{pong} : \tau_{pong})\}$. The second part of the premises concerns the definition of a worker which gets spawned by the server in reaction to a 'visit' message. We let $W_1 = \overline{\mathit{lock}}\langle \mathit{ready} \rangle$ and $W_2 = \mathit{ready}.\big(\overline{\mathit{read}}\langle \mathit{val} \rangle \parallel \mathit{val}.\overline{\mathit{write}}.\overline{\mathit{unlock}}.\overline{\mathit{reply}'}\big)$ so that $W = W_1 \parallel W_2$, and $\Gamma_4 = \Gamma \cup \{(\mathit{ready} : \tau_{ready}), (\mathit{val} : \tau_{val})\}$. Then, we apply rule In:

$$\dfrac{\begin{array}{c}\tau_{visit} = t_{visit}[\tau_{reply'}] \qquad \Gamma_4 \vdash_{\mathcal{T}} \boldsymbol{\nu}\mathit{ready}.\boldsymbol{\nu}\mathit{val}.(W_1 \parallel W_2) \\ t_{reply'} < t_{visit} \vee (\{t_{lock}\} < t_{visit} \wedge \{t_{read}, t_{write}, t_{unlock}\} < t_{visit})\end{array}}{\Gamma \vdash_{\mathcal{T}} \mathit{visit}(\mathit{reply}').\boldsymbol{\nu}\mathit{ready}.\boldsymbol{\nu}\mathit{val}.(W_1 \parallel W_2)}$$

This time the constraints on base types are not trivial. Note that the base types of the names $reply'$, $ready$ and val are not constrained by this rule as they do not appear in Γ.

We now have to show $\Gamma_4 \vdash_{\mathcal{T}} \nu ready.\nu val.W$ by

$$\dfrac{\dfrac{\tau_{lock} = t_{lock}[\tau_{ready}] \qquad \overline{\Gamma_4 \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma_4 \vdash_{\mathcal{T}} \overline{lock}\langle ready\rangle.\mathbf{0}} \qquad \Gamma_4 \vdash_{\mathcal{T}} W_2 \qquad \mathcal{C}_W}{\Gamma_4 \vdash_{\mathcal{T}} \nu ready.\nu val.(W_1 \parallel W_2)}$$

where

$$\begin{aligned}\mathcal{C}_W = {} & t_{lock} < t_{ready} \wedge t_{lock} < t_{val} \\ & \wedge \{t_{read}, t_{write}, t_{unlock}, t_{reply'}\} < t_{ready} \\ & \wedge \{t_{read}, t_{write}, t_{unlock}, t_{reply'}\} < t_{val}\end{aligned}$$

Note how, since $\mathrm{fn}(W_1) \cap \mathrm{fn}(W_2) \cap \{ready, val\} = \{ready\}$, W_1 and W_2 are linked, and hence tied so both the names $ready$ and val are tied to both the processes.

The proof of $\Gamma_4 \vdash_{\mathcal{T}} W_2$ takes the shape

$$\dfrac{\tau_{ready} = t_{ready} \qquad \dfrac{\tau_{read} = t_{read}[\tau_{val}] \qquad \overline{\Gamma_4 \vdash_{\mathcal{T}} \mathbf{0}}}{\Gamma_4 \vdash_{\mathcal{T}} \overline{read}\langle val\rangle.\mathbf{0}} \qquad \Gamma_4 \vdash_{\mathcal{T}} val.\overline{write}.\overline{unlock}.\overline{reply'}}{\Gamma_4 \vdash_{\mathcal{T}} ready.\big(\overline{read}\langle val\rangle \parallel val.\overline{write}.\overline{unlock}.\overline{reply'}\big)}$$

which can be completed as the other similar cases using rules In-0 and Out-0, imposing the unsurprising type constraints $\tau_x = t_x$ for $x \in \{val, write, unlock, reply'\}$. This completes the proof of $\Gamma \vdash_{\mathcal{T}} S$; we now conclude the derivation with the proof for the client:

$$\dfrac{\Gamma_5 \vdash_{\mathcal{T}} \overline{c} \qquad \dfrac{\Gamma_5 \vdash_{\mathcal{T}} c.(\overline{visit}\langle reply\rangle \parallel reply.\overline{c})}{\Gamma_5 \vdash_{\mathcal{T}} !\big(c.(\overline{visit}\langle reply\rangle \parallel reply.\overline{c})\big)} \qquad t_{visit} < t_c \wedge t_{visit} < t_{reply}}{\Gamma \vdash_{\mathcal{T}} \nu c.\nu reply.\big(\overline{c} \parallel !\big(c.(\overline{visit}\langle reply\rangle \parallel reply.\overline{c})\big)\big)}$$

where $\Gamma_5 = \Gamma \cup \{(c : \tau_c), (reply : \tau_{reply})\}$. This derivation again can be completed as before using rules In-0 and Out-0, obtaining the type constraints $\tau_c = t_c$, $\tau_{visit} = t_{visit}[\tau_{reply}]$ and $\tau_{reply} = t_{reply}$.

We can now put all the constraints together obtaining that:

$$\begin{aligned}
\tau_{db} &= t_{db} & \tau_{ping} &= t_{ping}[t_{pong}] & \tau_c &= t_c \\
\tau_{lock} &= t_{lock}[t_{ready}] & \tau_{pong} &= t_{pong} & \tau_{reply} &= \tau_{reply'} = t_{reply} \\
\tau_{read} &= t_{read}[t_{val}] & \tau_{visit} &= t_{visit}[t_{reply}] & & \\
\tau_{write} &= t_{write} & \tau_{ready} &= \tau_{ready'} = t_{ready} & & \\
\tau_{unlock} &= t_{unlock} & \tau_{val} &= \tau_{val'} = t_{val} & &
\end{aligned}$$

and

$$t_{visit} < t_c \wedge t_{visit} < t_{reply}$$
$$t_{lock} < t_{ready} \wedge t_{lock} < t_{val}$$
$$\{t_{read}, t_{write}, t_{unlock}, t_{reply}\} < t_{ready} \wedge \{t_{read}, t_{write}, t_{unlock}, t_{reply}\} < t_{val}$$
$$t_{reply} < t_{visit} \vee (\{t_{lock}\} < t_{visit} \wedge \{t_{read}, t_{write}, t_{unlock}\} < t_{visit})$$

which can be simplified to

$$\begin{array}{llll} t_{visit} < t_c & t_{visit} < t_{reply} & t_{lock} < t_{ready} & t_{lock} < t_{val} \\ t_{read} < t_{ready} & t_{write} < t_{ready} & t_{unlock} < t_{ready} & t_{reply} < t_{ready} \\ t_{read} < t_{val} & t_{write} < t_{val} & t_{unlock} < t_{val} & t_{reply} < t_{val} \\ t_{lock} < t_{visit} & t_{read} < t_{visit} & t_{write} < t_{visit} & t_{unlock} < t_{visit} \end{array}$$

which are acyclic.

Now, acyclicity ensures that there is a forest $\mathcal{T}$ such that M is typable. To prove that M is typably hierarchical we need to show that it is $\mathcal{T}$-shaped for a $\mathcal{T}$ satisfying the constraints.

A necessary condition for $\mathcal{T}$-shapedness is that, for every sequential subterm A of M, the names of $\mathrm{fn}(A)$ label a path in $\mathcal{T}$. Using this information to prune the obviously wrong choices, we end up with the tree

$$\mathcal{T} = t_{db} \prec t_{lock} \prec t_{read} \prec t_{write} \prec t_{unlock} \prec t_{visit} \prec t_{ping} \prec t_c \prec t_{reply} \prec t_{ready} \prec t_{val}.$$

The reader can check that M is indeed $\mathcal{T}$-shaped using this tree, confirming that M is typably hierarchical.

An expanding ring

In Section 1.3 we presented an example of depth-unbounded system, the `ring` program. Since it has no bound in depth the type system ought to reject it. Let us show this is the case by analysing a π-calculus version of the same program.

The π-term R initialises the ring with a single 'master' node pointing at itself as the next in the ring. The term M represents the `master` function: it waits on $self$ and reacts to a signal by creating a new slave connected with the previous next slave. A slave S simply propagates the signals on its channel to the next in the ring.

$$R = \nu master.\nu self.(M \parallel \overline{master}\langle self\rangle \parallel \overline{self})$$
$$M = !\big(master(next).self.\nu slave.(S \parallel \overline{master}\langle slave\rangle \parallel \overline{slave})\big)$$
$$S = !(slave.\overline{next})$$

The impossibility of typing R can already be seen from M: from $\overline{master}\langle slave\rangle$ we deduce that $t_{slave} = t_{next}$, which is in contradiction with the constraint that $t_{next} < t_{slave}$ required by rule Par when typing $\nu slave.(S \parallel \overline{master}\langle slave\rangle \parallel \overline{slave})$.

Abstraction and relevance to Soter

One of the possible applications of typably hierarchical systems to program analysis is as a cheap check to know if the program's abstraction, in the form of a π-term, can be analysed with depth-bounded systems model checking. The ring example leaves us with a question: suppose a π-term is extracted from an Erlang program and fails to type check, what can we do to analyse it?

A first option is to fallback to Soter's abstraction. In fact, Soter's counting abstraction is powerful enough in the ring example, to prove properties as 'all the mailboxes are bounded by 1', (i.e. there is at most one token going around the ring at all times).

A second, more refined, option is to devise an hybrid abstract model which uses ACS counters for problematic names as $slave$, but keeps the π-calculus' precision for other restrictions. To formalise this hybrid approach one can observe that:

1. the ACS' counter abstraction corresponds to making a restriction a globally free name in a π-term;

2. the detection of 'problematic names', can be done using heuristics analysing the names involved in the conflicting constraints in the type inference.

As future work, we intend to study a generalisation of Soter's abstraction targeting the π-calculus and using our type system to gracefully degrade the precision on names, until a typably hierarchical abstraction of the input program can be found.

Chapter 11

Relation with Other Models

In this chapter we explore connections between the typably hierarchical π-calculus and other known models of concurrent computation.

We first examine Nested Data Class Memory Automata and study the semantic correspondence with typably hierarchical terms in some depth. Then we consider, inspired by this correspondence, a variant of CCS that forms a proper subset of the typably hierarchical fragment. We conclude with a discussion on the vast literature on type systems and fragments of π-calculus.

11.1 Nested Data Class Memory Automata

After isolating a fragment of a process calculus, an interesting question is *can we find an automata based presentation of the same fragment?* In this section we give an answer to this question by relating the typably hierarchical fragment to a class of automata on data-words recently defined in [CMO15]: *Nested Data Class Memory Automata* (NDCMA).

The original presentation of NDCMAs sees them as language recognition devices: they can recognise sets of data-words, that is sequences of symbols in $\Sigma \times \mathbb{D}$ where Σ is a finite alphabet and $\mathbb{D}$ is an infinite set of *data values*. Notably, (weak) NDCMAs are more expressive than Petri nets, while enjoying decidability of some verification problems. While *Class Memory Automata* [BS07] do not postulate any structure on $\mathbb{D}$, NDCMAs assume that it is equipped with an infinitely branching, finite height forest structure, a new notion of nesting for data languages. Without the local acceptance condition, NDCMAs have decidable emptiness, and in the deterministic case are closed under all Boolean operations. NDCMAs have found applications in algorithmic game semantics, specifically in the design of algorithms for deciding observational equivalence of several low-order fragments of Reduced ML [CHMO15], with the nested dataset reflecting the tree-structure of the threads in the game-semantic plays. We will make use of this forest structure to represent $\mathcal{T}$-compatible π-term forests.

We are primarily interested in establishing a tight relation between the transition systems of NDCMAs and typably hierarchical terms. Therefore we do not regard NDCMAs as language recognisation devices but simply as computational models. For this reason, our definition ignores the language-related components of the original definition of [CMO15]: there is no finite alphabet Σ, no accepting control states, no accepting run. While in the language-theoretic formulation at each step in a run a letter and a data value must be read from the input string, here a transition can fire simply if *there exists* a data value satisfying the transition's precondition.

Definition 11.1 (NDCMA [CMO15])**.** We define a *nested dataset* $(\mathbb{D}, \mathrm{pred}_{\mathbb{D}})$ to be a forest of infinitely many trees of level ℓ which is *full* in the sense that for each data value d of level less than ℓ, there are infinitely many data values d' whose parent is d.

A *class memory function* (over a set A) is a function $f\colon \mathbb{D} \to A \uplus \{\mathfrak{f}\}$ such that $f(d) = \mathfrak{f}$ for all but finitely many $d \in \mathbb{D}$; $\mathfrak{f}$ is a special symbol indicating a data value is fresh, i.e. has never been used before.

Fix a nested data set of level ℓ. A *Nested Data CMA of level* ℓ is a tuple $(\mathbb{Q}, \delta, q_0, f_0)$ where $\mathbb{Q}$ is a finite set of states, $q_0 \in \mathbb{Q}$ is the initial control state, $f_0\colon \mathbb{D} \to \mathbb{Q}_{\mathfrak{f}}$ is the initial class memory function satisfying $f_0(\mathrm{pred}(d)) = \mathfrak{f} \implies f_0(d) = \mathfrak{f}$, and δ is the transition relation. δ is given by a union $\delta = \bigcup_{i=1}^{\ell} \delta_i$ where each δ_i is a relation: $\delta_i \subseteq \mathbb{Q} \times (\mathbb{Q}_{\mathfrak{f}})^i \times \mathbb{Q} \times \mathbb{Q}^i$ and $\mathbb{Q}_{\mathfrak{f}}$ is defined as $\mathbb{Q} \cup \{\mathfrak{f}\}$. A configuration is a pair (q, f) where $q \in \mathbb{Q}$, and $f\colon \mathbb{D} \to \mathbb{Q}_{\mathfrak{f}}$ is a class memory function. The initial configuration is (q_0, f_0). The automaton can transition from configuration (q, f) to configuration (q', f'), written $(q, f) \to_{\mathcal{A}} (q', f')$, just if there exists a level-i data value d such that $(q, q_1, \ldots, q_i, q', q'_1, \ldots, q'_i) \in \delta$, for all $j \in \{1, \ldots, i\}$, $f(\mathrm{pred}^{i-j}(d)) = q_j$ and

$$f' = f\big[\,\mathrm{pred}^{i-1}(d) \mapsto q'_1,\ \ldots,\ \mathrm{pred}(d) \mapsto q'_{i-1},\ d \mapsto q'_i\,\big].$$

Given a nested dataset $\mathbb{D}$ we write $\mathrm{CMF}(\mathbb{D}, \mathbb{Q})$ for the set of all class memory functions from $\mathbb{D}$ to $\mathbb{Q}_{\mathfrak{f}}$.

Theorem 11.1 (Decidability of reachability for NDCMA)**.** *The problem of determining, given an NDCMA* $(\mathbb{Q}, \delta, q_0, f_0)$ *and a configuration* (q, f)*, if* $(q_0, f_0) \to_{\mathcal{A}}^* (q, f)$*, is decidable.*

Proof. Consider the function $\|\_\|\colon \mathrm{CMF}(\mathbb{D}, \mathbb{Q}) \to \mathscr{P}(\mathbb{D})$ defined as

$$\|f\| := \{d \mid f(d) \neq \mathfrak{f}\}.$$

By definition of class memory function, $\|f\|$ is always a finite set. Once a control state has been assigned to a data value d, no transition rule can assign $\mathfrak{f}$ to it again; so we have

$$(q_1, f_1) \to_{\mathcal{A}} (q_2, f_2) \implies \|f_1\| \subseteq \|f_2\|$$

Therefore, in a reduction $(q_0, f_0) \to_{\mathcal{A}} (q_1, f_1) \to_{\mathcal{A}} \cdots \to_{\mathcal{A}} (q_n, f_n)$ for all $0 \leq i < n$ we have $\|f_i\| \subseteq \|f_n\|$. Then, to check for reachability of (q, f) we only need to check every reduction sequence between configurations in the finite set $\mathbb{Q} \times \{f' \mid \|f'\| \subseteq \|f\|\}$. □

While reachability for NDCMA had not been considered before as it does not have applications in the language theoretic interpretation, coverability was showed to be decidable in [CMO15] by using techniques very similar to the ones used for proving decidability of coverability for depth-bounded systems (see Section 9.5). More precisely, both arguments crucially rely on the rooted tree embedding well quasi ordering.

In the following sections, we want to show that NDCMAs and typably hierarchical π-terms have a quite similar semantics. To highlight similarities and differences and support reductions of verification problems from one model to the other, in the next two sections we present two encodings.

In Section 11.2 we show an encoding from typeable π-terms, and we prove that a transition system generated from the NDCMA encoding is bisimilar to the transition system generated by the reduction semantics of the π-term. This encoding shows how NDCMA transitions can simulate π-calculus' synchronisation mechanism, when mobility is restricted to the hierarchical case.

Then in Section 11.3, we describe an encoding from NDCMA to a subset of typably hierarchical terms, with a similar semantic correspondence. This encoding highlights the mismatch between the expressivity of typably hierarchical terms and NDCMA: the image of the encoding is a fragment of π-calculus showing no mobility. In Section 11.4 we will compare the image of this encoding with variants of CCS.

11.2 Encoding Typably Hierarchical Terms into NDCMA

We make a few simplifying assumptions on the term to be encoded as an NDCMA. First, we assume P is a closed normal form, i.e. $\mathrm{fn}(P) = \emptyset$, second we assume P contains no $\boldsymbol{\tau}$ action. It would be easy to support the general case but we only focus on the core case for conciseness. Fix a closed $\mathcal{T}$-shaped π-term P such that $\emptyset \vdash_{\mathcal{T}} P$, with $\ell = \mathrm{height}(\mathcal{T})$. We will construct a level-ℓ automaton $\mathcal{A}[\![P]\!]$ from P so that their transition systems are essentially bisimilar.

The intuition behind the encoding is as follows. A configuration (q, f) represents a π-term P by using f to label a finite portion of $\mathbb{D}$ so that it is isomorphic to a $\mathcal{T}$-compatible forest in $\mathcal{F}[\![P]\!]$. Our encoding proceeds in rounds. A single synchronisation step between two processes will be simulated by a predictable number of steps of the automaton. Since π-terms exhibit non-determinism, the automata in the image of the encoding need to be non-deterministic as well. We make use of the non-determinism of the automata model in a second way: in a reduction, the two synchronising processes are not in the same path in the syntax tree (they are both leaves by construction) but the automaton can only examine one path in $\mathbb{D}$ at a time; we then first guess the sender, mark the channel carrying its message, then select a receiver waiting on that channel (which will be in the

path of both processes) and then spawn their continuations in the relevant places. This requires separate steps and could lead to spurious deadlocks when no process is listening over the selected channel. These deadlocked states can be pruned from the bisimulation by restricting the relevant transition system to those configurations where the control state is a distinguished state that signals that the intermediate steps of a synchronisation have been completed. A successful round follows very closely the operations used in the proof of Theorem 10.10.

A round starts from a configuration with control state q_{ready}, then goes through a number of intermediate steps until it either deadlocks or reaches another configuration with control state q_{ready}. Only reachable configurations of $\mathcal{A}[\![P]\!]$ with q_{ready} as control state will correspond to reachable terms of P. Thus, given an automaton $\mathcal{A} = (\mathbb{Q}, \delta, q_{\text{ready}}, f_0)$, we define the transition relation $(\Rightarrow_{\text{ready}}) \subseteq \mathrm{CMF}(\mathbb{D}, \mathbb{Q})^2$ as the minimal relation such that $f \Rightarrow_{\text{ready}} f'$ if $(q_{\text{ready}}, f) \to_{\mathcal{A}} (q_1, f_1) \to_{\mathcal{A}} \cdots \to_{\mathcal{A}} (q_n, f_n) \to_{\mathcal{A}} (q_{\text{ready}}, f')$ where in the possibly empty sequence of (q_i, f_i), $q_i \neq q_{\text{ready}}$.

To encode a reachable term Q in a configuration (q_{ready}, f) we use f to represent the forest $\Phi(Q)$: roughly speaking we represent a node n of $\Phi(Q)$ labelled with l with a data value d mapped to a q_l by f. In general, due to the generation of unboundedly many names, there might be infinitely many such labels l. We therefore need to show that we can indeed use only a finite number of distinct labels to represent names and reachable sequential subterms with control states. This is achieved by using the concept of derivatives. The set of *derivatives* of a term P is the set of sequential subterms of P, both active or not active. More formally, it is the set defined by the following function

$$\begin{aligned}
\mathrm{der}(\mathbf{0}) &:= \emptyset \\
\mathrm{der}(\boldsymbol{\nu} x.P) &:= \mathrm{der}(P) \\
\mathrm{der}(P \parallel Q) &:= \mathrm{der}(P) \cup \mathrm{der}(Q) \\
\mathrm{der}(!M) &:= \{!M\} \cup \mathrm{der}(M) \\
\mathrm{der}(M + M') &:= \{M + M'\} \cup \mathrm{der}(M) \cup \mathrm{der}(M') \\
\mathrm{der}(\pi.P) &:= \{\pi.P\} \cup \mathrm{der}(P)
\end{aligned}$$

Clearly, $\mathrm{der}(P)$ is a finite set. Every active sequential subterm of a term P' reachable from P is congruent to a $Q\sigma$ for some substitution σ. When P is depth-bounded, we know from [Mey08] that, there is a finite set of substitutions such that the substitution σ above can always be drawn from this set. The assumption that P is $\mathcal{T}$-shaped and typable allows us to be even more specific. Let $X_{\mathcal{T}} = \{\chi_t \mid t \in \mathcal{T}\}$ be a finite set of names, we define $\Delta_P := \{Q\sigma \mid Q \in \mathrm{der}(P), \sigma\colon\ \mathrm{fn}(Q) \to (X_{\mathcal{T}} \cup \mathrm{fn}(P))\}$.

Lemma 11.2. *Let P be a term such that $\mathrm{forest}(P)$ is $\mathcal{T}$-compatible. Then there exists a term Q such that $\mathrm{forest}(Q)$ is $\mathcal{T}$-compatible, Q is an α-renaming of P, $\mathrm{bn}_\nu(Q) \subseteq X_{\mathcal{T}}$ and each active sequential subterm of Q is in Δ_P.*

Proof. By definition of $\mathcal{T}$-compatible forest we have that in any path of $\mathrm{forest}(P)$ no two distinct nodes will have labels (x, t) (x', t) so α-renaming each restriction $(x : \tau)$ of P' to $(\chi_{\mathrm{base}(\tau)} : \tau)$ will yield the desired Q. □

Henceforth, we will write $\Phi'(P)$ for a relabelling of the forest $\Phi(P)$ such that its labels use only names in $X_{\mathcal{T}}$, as justified by Lemma 11.2.

Corollary 11.3. *If a term P is typably hierarchical, then every $P' \in \mathrm{Reach}(P)$ is congruent to a term Q such that $\mathrm{bn}_\nu(Q) \subseteq X_{\mathcal{T}}$ and each active sequential subterm of Q is in Δ_P.*

Proof. By Theorem 10.10 and Lemma 11.2. □

The transition relation of the automaton encoding of a term P is then derived from the set Δ_P.

Before we show how to construct the transitions of the automaton from the term, we define a relation $\sim$ between terms and class memory functions. This relation formalises how we encode the term as a labelling of data values, and will have a crucial role in proving the soundness of the encoding. Let Q be a term reachable from P and (q_{ready}, f) be a configuration of an automaton $\mathcal{A}$. Let $\varphi = \Phi'(Q)$, the relation $Q \sim f$ holds if and only if there exists an injective function $\iota \colon \mathrm{nodes}(\varphi) \to \mathbb{D}$ such that for all $n \in \mathrm{nodes}(\varphi)$:

i) if $\iota(n) = d$, $n' \prec_\varphi n$ and $\iota(n') = d'$ then $d' = \mathrm{pred}(d)$;

ii) if n is labelled with (χ_i, t) then $f(\iota(n)) = \chi_i$;

iii) if n is labelled with a sequential process Q' then $f(\iota(n)) = Q'$;

iv) for each d such that $f(d) \neq \mathfrak{f}$ either there is an n such that $\iota(n) = d$ or $f(d) = q_\dagger$.

Let us now describe how we can simulate reduction steps of a π-term with transitions in a NDCMA. In encoding a π-term's semantics into the transition relation of a NDCMA, we need to overcome the differences in the primitive steps allowed in the two models. Simulating a π-calculus synchronisation requires matching two paths, leading to the two reacting sequential terms, in $\mathbb{D}$ at the same time. A step in the automata semantics can only manipulate a single path, so we will need to split the detection of a redex in two phases: finding the sender, then finding a matching receiver. Moreover, finding a redex requires detecting that the path under consideration contains a node labelled with the synchronising channel and one with the appropriate sequential term, ignoring how many and which other nodes are in between them. To succinctly represent this operation, we introduce the following notation. Fix a set $\mathbb{Q}$ including $q, q', l_1, \ldots, l_n, l'_1, \ldots, l'_n, l$. We associate to the expression $[q, l_1 \ldots l_n] \to [q', l'_1 \ldots l'_n]$ the set of transitions

$$\mathrm{tran}_{\mathbb{Q}}\big([q, l_1 \ldots l_n] \to [q', l'_1 \ldots l'_n]\big) := \Big\{(q, q_1, \ldots, q_m, q', q'_1, \ldots, q'_m) \in \mathbb{Q}^{2m+2} \mid \exists\, i_1 \ldots i_m.\ 1 \leq i_1 < \cdots < i_n \leq m, q_{i_j} = l_j, q'_{i_j} = l'_j\Big\}.$$

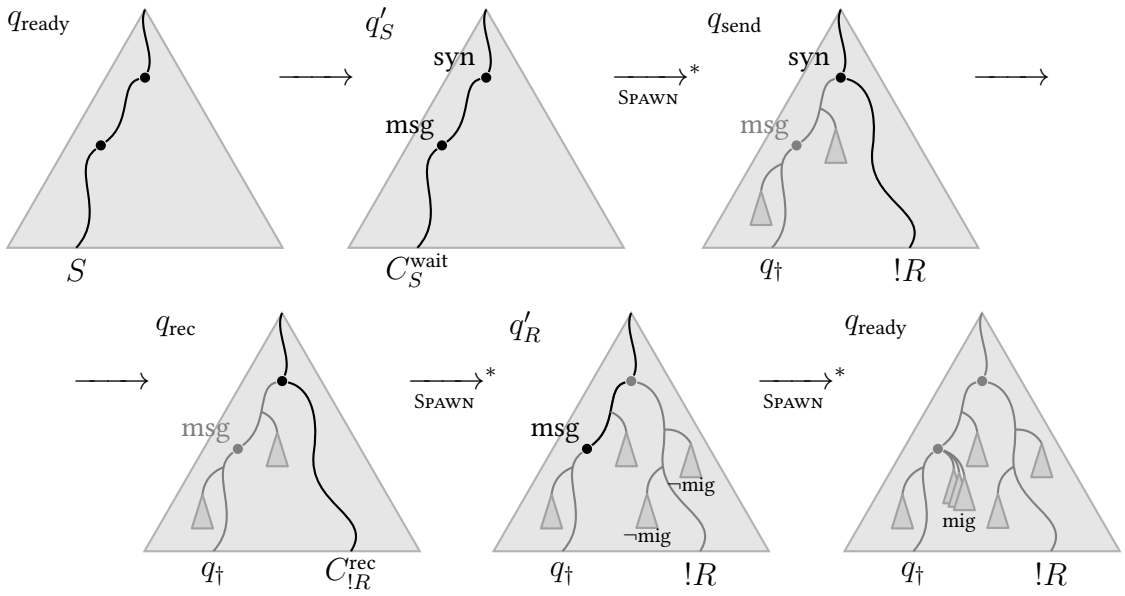


Figure 11.1 – A schema of the transitions simulating a synchronisation in the automaton encoding of a term. The trees represent the class memory functions associated with configurations in the run of the automaton. The run simulates a sender synchronising with a replicated receiver. The two displayed nodes in the path leading to S are the ones labelled with the names of, from top to bottom, the synchronisation channel and the exchanged message.

When the sequence $l_1, \ldots, l_n$ is empty, the expression simply means that the automaton may go from a configuration (q, f) to (q', f) with no condition (nor effect) on f. Similarly, we associate to the expression $[q, l_1 \ldots l_n; \mathfrak{f}] \to [q', l'_1 \ldots l'_n; l]$ the set of transitions

$$\begin{aligned}\mathrm{tran}_{\mathbb{Q}}\big([q, l_1 \ldots l_n; \mathfrak{f}] \to [q', l'_1 \ldots l'_n; l]\big) := \\ \big\{(q, q_1, \ldots, q_m, \mathfrak{f}, q', q'_1, \ldots, q'_m, l) \in \mathbb{Q}^{2m+4} \mid \\ \exists\, i_1 \ldots i_m.\ 1 \leq i_1 < \cdots < i_n = m, q_{i_j} = l_j, q'_{i_j} = l'_j\big\}.\end{aligned}$$

Note that the sequence $l_1, \ldots, l_n$ may be empty, in which case the data value labelled with $\mathfrak{f}$ is selected among the level-1 ones. The set of states mentioned in an expression is

$$\begin{aligned}\mathrm{states}\big([q, l_1 \ldots l_n] \to [q', l'_1 \ldots l'_n]\big) &:= \{q, q', l_1, \ldots, l_n, l'_1, \ldots, l'_n\} \\ \mathrm{states}\big([q, l_1 \ldots l_n; \mathfrak{f}] \to [q', l'_1 \ldots l'_n; l]\big) &:= \{q, q', l, l_1, \ldots, l_n, l'_1, \ldots, l'_n\}.\end{aligned}$$

To define the transitions of the encoding of a term, we make use of some auxiliary definitions generating sets of transition expressions.

Setup(q, q', l, l', φ) adds to the path leading to a data value labelled with l, the nodes corresponding to a forest $\varphi \in \mathcal{F}[\![Q]\!]$ for some Q. These transitions are deterministic in

the sense that a configuration (q, f) with only one data value labelled with l will transition through all the transitions dictated by $\textsc{Setup}(q, q', l, l', \varphi)$ reaching (q', f'). Formally, suppose, for some j and k, $\varphi = \{(x_1, \tau_1)[\varphi_1], \ldots, (x_j, \tau_j)[\varphi_j]\} \cup \{Q_1[], \ldots, Q_k[]\}$ where all x_i are in $X_{\mathcal{T}}$ and all $Q_i \in \Delta_P$. Then $\textsc{Setup}$ is defined as follows:

$$\begin{aligned}
\textsc{Setup}(q, q', l, l', \varphi) := {} & \{[q, l; \mathfrak{f}] \to [q_1, l; Q_1^{\text{ready}}]\} \\
& \cup \{[q_i, l; \mathfrak{f}] \to [q_{i+1}, l; Q_{i+1}^{\text{ready}}] \mid 1 \leq i \leq j\} \\
& \cup \{[q_j, l; \mathfrak{f}] \to [q'_1, l; Q_j^{\text{ready}}]\} \\
& \cup \{[q'_i, l; \mathfrak{f}] \to [q''_i, l; x_i^{\text{set}}] \mid 1 \leq i \leq k\} \\
& \cup \bigcup_{i=1}^{k} \textsc{Setup}(q''_i, q'_{i+1}, x_i^{\text{set}}, x_i, \varphi_i) \\
& \cup \{[q'_{k+1}, l] \to [q', l']\}
\end{aligned}$$

where for all $1 \leq i \leq j$ and all $1 \leq i' \leq k$, $q_i, q'_{i'}, q''_{i'}, q'_{k+1}$ are fresh intermediate control states. in the sense that they are only mentioned in the transitions generated by that specific application of $\textsc{Setup}$. We allow l to be the empty sequence, in which case l' needs to be the empty sequence as well.

Similarly, we define $\textsc{Spawn}(q, q', l, l', \varphi)$ to be the set of transitions needed to append each tree in φ to nodes in the path leading to a data value d labelled with l; the operation starts at control state q and ends at control state q' with the label for d updated to l'. Each tree is appended to the node with the lowest level such that every name mentioned in its leaves is an ancestor of such node. Since a single transition can add only one node of φ, we need a number of transitions to complete the operation; these transitions will however be deterministic in the same sense as the ones required to complete a $\textsc{Setup}$ operation. Formally, let the forest $\varphi = \Phi'(D)$ consist of trees $\theta_1, \ldots, \theta_k$, for a term $D \in \Delta_P$. We can precompute, for each θ_i, the base type $t_i := \min_{\prec_{\mathcal{T}}} \{t \mid \chi_t \in \text{fn}(A), n \in N_{\theta_i}, \ell_{\theta_i}(n) = A\}$ when defined. For each label $\chi_t \in X_{\mathcal{T}}$ we also have a label χ_t^{sp} we write $\chi(\theta_i)$ (resp. $\chi^{\text{sp}}(\theta_i)$) for χ_{t_i} (resp. $\chi_{t_i}^{\text{sp}}$) when t_i is defined, or the empty sequence when t_i is undefined (e.g. when θ_i does not have free variables). Then $\textsc{Spawn}(q, q', l, l', \varphi)$ is the set of transition expressions defined as follows:

$$\begin{aligned}
& \textsc{Spawn}(q_0, q', l, l', \varphi) := \\
& \qquad \{[q_{i-1}, \chi(\theta_i)\ l] \to [q'_{i-1}, \chi^{\text{sp}}(\theta_i)\ l] \mid 1 \leq i \leq k\} \\
& \qquad \cup \bigcup_{i=1}^{k} \textsc{Setup}(q'_{i-1}, q_i, \chi^{\text{sp}}(\theta_i), \chi(\theta_i), \theta_i)
\end{aligned}$$

where for all $1 < h \leq k$, q_h, q'_h are fresh.

We define for each $D \in \Delta_P$ the set of transition expressions $\textsc{React}(D)$ representing the steps needed to simulate in the automaton the potential reactions of D.

$$\textsc{React}(M) := \textsc{React}^{M}_{q_\dagger}(M)$$
$$\textsc{React}(!M) := \textsc{React}^{!M}_{!M}(M)$$

The set of transition expressions $\textsc{React}^{D}_{q}(M)$ collects all the potential reactions of M as a choice of D; the label q is the one that should be associated with the "consumed" term D after a reaction has been completed. The transitions simulating a replicated component will not mark, as the ones for non replicated terms, the reacted term with $q_\dagger$, which will represent "garbage" inert nodes in f. The term $\mathbf{0}$ cannot initiate any step and a choice may do any action that one of its choices can:

$$\textsc{React}^{D}_{q}(\mathbf{0}) := \emptyset$$
$$\textsc{React}^{D}_{q}(M + M') := \textsc{React}^{D}_{q}(M) \cup \textsc{React}^{D}_{q}(M')$$

Any sender can initiate a synchronisation from the ready state:

$$\begin{aligned}\textsc{React}^{D}_{q}(\overline{\chi_t}\langle\chi_{t'}\rangle.C) := \\ &\{[q_{\text{ready}}, \chi_t\chi_{t'}D] \to [q', \chi_t^{\text{syn}}\chi_{t'}^{\text{msg}}C^{\text{wait}}] \mid t < t'\} \\ &\cup \{[q_{\text{ready}}, \chi_{t'}\chi_t D] \to [q', \chi_{t'}^{\text{msg}}\chi_t^{\text{syn}}C^{\text{wait}}] \mid t > t'\} \\ &\cup \textsc{Spawn}(q', q_{\text{send}}, C^{\text{wait}}, q, \Phi'(C)).\end{aligned}$$

where q' is fresh. Here, the state q_{send} signals that we are in the middle of a synchronisation, where the sender is committed but a receiver has yet to be selected.

For the case of an input prefix $M = \chi_t(x).C$ we distinguish two cases: when the base type of χ_t is greater than the base type of x no migration occurs, otherwise part of the continuation needs to be spawned in the sender's path. In the case when the base type of χ_t is greater than the base type of x, we set

$$\begin{aligned}\textsc{React}^{D}_{q}(\chi_t(x).C) := \\ &\{[q_{\text{send}}, \chi_t^{\text{syn}}\chi_{t'}^{\text{msg}}D] \to [q_{\text{rec}}, \chi_t\chi_{t'}C^{\text{rec}}] \mid t < t' \in \mathcal{T}\} \\ &\{[q_{\text{send}}, \chi_{t'}^{\text{msg}}\chi_t^{\text{syn}}D] \to [q_{\text{rec}}, \chi_{t'}\chi_t C^{\text{rec}}] \mid t > t' \in \mathcal{T}\} \\ &\cup \textsc{Spawn}(q_{\text{rec}}, q_{\text{ready}}, C^{\text{rec}}, q, \Phi'(C)).\end{aligned}$$

In the case when the base type of χ_t is greater than the base type of x, more transitions are required. First, we precompute for each $M = \chi_t(x).C$ as above and $t < t' \in \mathcal{T}$, the two forests $\varphi_{\text{mig}}(C, t')$ and $\varphi_{\neg\text{mig}}(C)$ such that $\Phi'(C[\,\chi_{t'}/x\,]) = \varphi_{\text{mig}}(C, t') \uplus \varphi_{\neg\text{mig}}(C)$ and $\varphi_{\text{mig}}(C, t')$ contains all the nodes labelled with sequential terms tied to $\chi_{t'}$ in $C[\,\chi_{t'}/x\,]$.

As we have shown in the proof of Theorem 10.10, by virtue of Lemma 9.2, $\varphi_{\text{mig}}(C, t')$ and $\varphi_{\neg\text{mig}}(C)$ are indeed disjoint. Then we set:

$$\begin{aligned}
&\textsc{React}_q^D(\chi_t(x).C) := \\
&\quad \{[q_{\text{send}}, \chi_t^{\text{syn}} D] \to [q_{\text{rec}}, \chi_t C^{\text{rec}}]\} \\
&\quad \cup \textsc{Spawn}(q_{\text{rec}}, q', C^{\text{rec}}, q, \varphi_{\neg\text{mig}}(C)) \\
&\quad \cup \bigcup_{t' \in \mathcal{T}} \textsc{Setup}(q', q_{\text{ready}}, \chi_{t'}^{\text{msg}}, \chi_{t'}, \varphi_{\text{mig}}(C, t'))
\end{aligned}$$

where q' is a fresh intermediate control state. Figure 11.1 illustrates the steps the automaton performs when simulating a synchronisation.

Definition 11.2 (Automaton encoding)**.** The automaton encoding of a typably hierarchical term P is the NDCMA $\mathcal{A}[\![P]\!] = (\mathbb{Q}, \delta, q_{\text{ready}}, f)$ where $\textsc{Tr} = \bigcup \{\textsc{React}(D) \mid D \in \Delta_P\}$, $\mathbb{Q} = \text{states}(\textsc{Tr})$, $\delta = \text{tran}_{\mathbb{Q}}(\textsc{Tr})$ and f is an arbitrary class memory function such that $P \sim f$.

We will now show that the transition system of the semantics of P is bisimilar to the one of $\mathcal{A}$ when restricting it to configurations with control state equal to q_{ready}.

Establishing that two transition systems are bisimilar implies that a wide class of properties are preserved across bisimilar states. For our purposes, proving that the automaton encoding of a term gives rise to a bisimilar transition system has the important consequence that reachability can be reduced from one model to the other.

Theorem 11.4. *The transition system* $(\text{CMF}(\mathbb{D}, \mathbb{Q}), \Rightarrow_{\textit{ready}}, f_0)$ *induced by the automaton* $\mathcal{A}[\![P]\!] = (\mathbb{Q}, \delta, q_{\textit{ready}}, f_0)$ *obtained from a closed typably hierarchical term* P*, is bisimilar to the transition system of the reduction semantics of* P*,* $(\text{Reach}(P), \to, P)$*.*

The result is proved by showing that the relation $\sim$ defined above, is a bisimulation that relates the initial states of the two transition systems. By definition of $\mathcal{A}[\![P]\!]$ we have $P \sim f_0$. Showing that $\sim$ is indeed a bisimulation amounts to showing that if $Q \sim f$ then:

(A) for each Q' such that $Q \to Q'$ there is a f' such that $f \Rightarrow_{\text{ready}} f'$ and $Q' \sim f'$;

(B) for each f' such that $f \Rightarrow_{\text{ready}} f'$ there is a Q' such that $Q \to Q'$ and $Q' \sim f'$.

To show this holds we rely on the hypothesis that $Q \sim f$ to get a ι relating $\Phi'(Q)$ and f. The proof then closely follows the constructions in the proof of Theorem 10.10. If $Q \to Q'$ we can find two nodes n_S and n_R in $\Phi'(Q)$ labelled with the sender and receiver processes responsible for the reduction; they will share an ancestor n_a labelled (χ_t, t) corresponding to the channel on which they are synchronising. On the automaton side, we have that (q_{ready}, f) matches the rule generated from the sender by selecting the data

value $d_S = \iota(n_S)$, a data value d_b corresponding to the name being sent and $d_a = \iota(n_a)$. This leads to (q', f) where $f'(d_a) = \chi_t^{\text{syn}}$, $f'(d_b) = \chi_{t'}^{\text{msg}}$, $f'(d_S) = S'^{\text{wait}}$. From here only one of the transitions generated from Spawn of the continuation is enabled as there is only one node marked with 'wait'. The transitions are deterministic from here until a configuration (q_{send}, f') is reached with f' representing the initial forest with the continuation of the sender added and with the node of the sender updated with either $q_\dagger$ or the sender itself if it is a replicated component. At this point there is only one data value marked with 'syn' and the only transitions from q_{send} are the ones generated from a process that can receive from the marked channel. We can pick the rule that has been generated from the receiver involved in the reduction from Q to Q' and go to a configuration with control state q_{rec}. From this configuration the transitions are deterministic. The next configuration reached with control state q_{ready} is bisimilar to Q' by tracing the effects these transitions have on the class memory function. Fresh data values get assigned labels compatible with the non migrating continuations of the receiver first, and then the migrating ones as children of d_b; data values with meaningless labels get assigned the label $q_\dagger$.

To prove (B) we proceed similarly. Every reduction sequence from (q_{ready}, f) to (q_{ready}, f') must start with a transition to a configuration with control state q_{send}, which is generated by rules extracted from a sender S labelling a data value d_S; since $Q \sim f$ we know that $n_S = \iota^{-1}(d_S)$ is labelled with S in $\Phi'(Q)$, hence S is an active sequential process of Q. To complete this part of the proof we only need to follow the transitions of the automaton in the same way as done for the previous point, and note that the only way the automaton can reach a configuration with control state q_{ready} from (q_{ready}, f) is by selecting a receiver that can synchronise with the selected sender. This is important because there may be transitions from (q_{ready}, f) corresponding to selecting a sender trying to synchronise on a channel on which no receiver is listening. This transition would lead to a deadlocked configuration (one with no successors) but never going through a configuration with control state q_{ready}.

Let us know see in more detail which properties are preserved by the encoding thanks to Theorem 11.4. For reachability, the bisimulation result allows us to infer the following corollary.

Corollary 11.5. *For each class memory function $f \in \mathrm{CMF}(\mathbb{D}, \mathbb{Q})$ with $P \sim f$, a π-term Q is reachable from P if and only if there exists a $f' \sim Q$ such that $(q_{ready}, f) \to_{\mathcal{A}}^* (q_{ready}, f')$*

The existential quantification of f' makes the previous result not strong enough to inter-reduce reachability of NDCMA and reachability of typably hierarchical terms. While NDCMA reachability is decidable, as we prove in Theorem 11.1, decidability of reachability of typably hierarchical terms is open.

Corollary 11.5 does however imply that the coverability problems on the two models are equivalent.

11.3 Encoding NDCMA into Typably Hierarchical Terms

In this section we sketch how an NDCMA can be encoded into a bisimilar typably hierarchical π-term.

Similarly as the encoding in the opposite direction, the π-calculus encoding of an automaton $\mathcal{A}$ will represent a reachable configuration (q, f) using the forest of a reachable term P. A term representing a reachable configuration may need to execute several steps before reaching another term representing a successor configuration.

Fix an automaton $(\mathbb{Q}, \delta, q_0, f_0)$. For simplicity we show the case where $\forall d.\ f_0(d) = \mathfrak{f}$, the general case follows the same scheme. First we note that every transition in δ_i is of the form

$$(q_0, q_1 \dots q_j, \underbrace{\mathfrak{f}, \dots, \mathfrak{f}}_{i-j}, q'_0, q'_1 \dots q'_i)$$

for some $1 \leq j \leq i$, where $q_k \in \mathbb{Q}$ for all $0 \leq k \leq j$. Instead of using the partition $\delta = \bigcup_{i=1}^{\ell} \delta_i$ we re-partition the transition relation as $\delta = \bigcup_{j=0}^{\ell} \theta_j$ where

$$\theta_j := \bigcup_{i=j}^{\ell} \{(q_0, q_1 \dots q_j, \underbrace{\mathfrak{f}, \dots, \mathfrak{f}}_{i-j}, q'_0, q'_1 \dots q'_i) \in \delta_i\}$$

(fixing $\delta_0 = \emptyset$ for uniformity). We introduce a channel name c^i_q for each $q \in \mathbb{Q}$ and each level of the automaton i. Our encoding will show no mobility, so each such channel c will have type t_c, hence no message will be exchanged on synchronisation; we abbreviate this kind of synchronisation with $c.P$ and $\overline{c}.Q$.[1] Let $\mathcal{C}^i := \{(c^i_q : t_{c^i_q}) \mid q \in \mathbb{Q}\}$. Given a transition $tr \in \theta_j$ where $tr = (q_0, q_1 \dots q_j, \mathfrak{f}, \dots, \mathfrak{f}, q'_0, q'_1 \dots q'_i)$ we define the term A_{tr} to be

$$A_{tr} := c^0_{q_0}. \cdots .c^j_{q_j}.\nu\mathcal{C}^{j+1}. \cdots \nu\mathcal{C}^i. \left(\prod_{k=0}^{i} \overline{c}^k_{q'_k} \parallel \prod_{k=j+1}^{i} P_{\theta_k} \right)$$

where $P_{\theta_j} := \prod_{tr \in \theta_j} !(A_{tr})$ and $\prod_{k=i+1}^{i} P_{\theta_k} = \mathbf{0}$. Note that these definitions are well-defined since they are not recursive. The *π-term encoding* of the NDCMA $\mathcal{A} = (\mathbb{Q}, \delta, q_0, f_0)$ is then defined as $\mathcal{P}[\![\mathcal{A}]\!] := \nu\mathcal{C}^0.(P_{\theta_0} \parallel \overline{c}^0_{q_0})$.

Similarly to our previous result, the encoding needs more than one step to simulate a single transition of the automaton. Hence, to state the result on the correspondence between the semantics of the automaton and its encoding, we define a derived transition system on π-terms as follows. Let P and Q be two π-terms such that $P \to^+ Q$, if $P \equiv \nu\mathcal{C}^0.(\overline{c} \parallel P')$ and $Q \equiv \nu\mathcal{C}^0.(\overline{c}' \parallel Q')$ with $c, c' \in \mathcal{C}^0$, and none of the intermediate processes in the reduction from P to Q is in that form, then $P \Rightarrow_{\mathcal{C}^0} Q$. Note that even

[1] see Section 10.6 for details on how to handle zero-arity channels.

after α-renaming a term in the encoding, we would be able to pinpoint names from each $\mathcal{C}^i$ by looking at their types, as α-renaming does not affect type annotations.

Theorem 11.6. *The transition system generated by the semantics of a level-ℓ NDCMA $\mathcal{A}$ and the transition system $\Rightarrow_{\mathcal{C}^0}$ with $\mathcal{P}[\![\mathcal{A}]\!]$ as initial state, are bisimilar.*

Proof. Fix an NDCMA $\mathcal{A} = (\mathbb{Q}, \delta, q_0, f_0)$ with $\delta = \bigcup_{0 \leq j \leq \ell} \theta_j$ as before. We prove the theorem by exhibiting a bisimulation relation $(\backsim) \subseteq (\mathbb{Q} \times (\mathbb{D} \rightharpoonup \mathbb{Q}_\mathfrak{f})) \times \mathrm{Reach}(\mathcal{P}[\![\mathcal{A}]\!])$ between the two transition systems. For a class memory function $f : \mathbb{D} \to \mathbb{Q}_\mathfrak{f}$, let $f(\mathbb{D})$ be the $\mathbb{Q}$-labelled forest with the set $N = \{d \in \mathbb{D} \mid f(d) \neq \mathfrak{f}\}$ as nodes, each labelled with $f(d)$ and with $\mathrm{pred}_\mathbb{D}$ restricted to N as parent relation. We first define a hierarchy of relations $\backsim_i$ between $\mathbb{Q}$-labelled forests and π-terms, for $0 \leq i \leq \ell$, as follows: $q[\{\varphi_1, \ldots, \varphi_n\}] \backsim_i \nu\mathcal{C}^i.(P_{\theta_i} \parallel \overline{c}^i_q \parallel \prod_{1 \leq j \leq n} P_j)$ if, for all $1 \leq j \leq n$, $\varphi_j \backsim_{i+1} P_j$. Since n must be 0 for $i = \ell$, the relation is well-defined. Let $P \in \mathrm{Reach}(\mathcal{P}[\![\mathcal{A}]\!])$ and (q, f) be a reachable configuration of $\mathcal{A}$. Then $(q, f) \backsim P$ if there exists a $P' \equiv P$ such that $q_0[f(\mathbb{D})] \backsim_0 P'$. To show that $\backsim$ is indeed a bisimulation, we have to prove that if $(q, f) \backsim P$ then:

(A) for each (q', f') such that $(q, f) \to_\mathcal{A} (q', f')$ there is a P' such that $P \Rightarrow_{\mathcal{C}^0} P'$ and $(q', f') \backsim P'$;

(B) for each P' such that $P \Rightarrow_{\mathcal{C}^0} P'$ there is a (q', f') such that $(q, f) \to_\mathcal{A} (q', f')$ and $P' \backsim (q', f')$.

To prove (A) we proceed as follows; suppose $(q, f) \to_\mathcal{A} (q', f')$ is an application of a transition $t = (q, q_1 \ldots q_j, \mathfrak{f}, \ldots, \mathfrak{f}, q', q'_1 \ldots q'_i) \in \theta_j$ then the forest $q[f(\mathbb{D})]$ has a path from the root to a leaf labelled with $q, q_1, \ldots, q_j$, which, by definition of $\backsim$, implies that P is congruent to a term with the following shape:

$$\nu\mathcal{C}^0.(R_0 \parallel \overline{c}^0_q \parallel \nu\mathcal{C}^1.(R_1 \parallel \overline{c}^1_{q_1} \parallel \cdots \nu\mathcal{C}^j.(R_j \parallel \overline{c}^j_{q_j} \parallel P_{\theta_j}) \cdots).$$

By construction, $P_{\theta_j} \equiv !(A_{tr}) \parallel R$ and A_{tr} is a process inputting once from c^0_q then once from each $c^k_{q_k}$ in sequence. From the shape of P we can conclude all of these input prefixes can synchronise with the dual $\overline{c}^k_{q_k}$ processes in parallel with them, activating, in $j+1$ steps, the continuation $C = \nu\mathcal{C}^{j+1}. \cdots \nu\mathcal{C}^i.\left(\overline{c}^0_{q'} \parallel \prod_{k=1}^{\ell} \overline{c}^k_{q'_k} \parallel P_{\theta_{j+1}}\right)$, yielding the process

$$P' \equiv \nu\mathcal{C}^0.(R_0 \parallel \overline{c}^0_{q'} \parallel \nu\mathcal{C}^1.(R_1 \parallel \overline{c}^1_{q'_1} \parallel \cdots \nu\mathcal{C}^i.(R_i \parallel \overline{c}^j_{q'_j}) \cdots)$$

where for k between $j+1$ and i, $R_k = P_{\theta_k}$. Now consider the forest $q'[f'(\mathbb{D})]$: it coincides with $q[f(\mathbb{D})]$ except on the path we singled out, now labelled with $q', q'_0, \ldots, q'_j$ and continuing to a leaf with nodes labelled $q'_{j+1}, \ldots, q'_i$. It is easy to see that $q'[f'(\mathbb{D})] \backsim P'$.

To prove (B) one can proceed similarly, by observing that even if $\mathcal{P}[\![\mathcal{A}]\!]$ can perform some reductions which deadlock that do not correspond to reductions of the automaton,

these steps cannot lead to a state with $\overline{c}^0_{q'}$ as one of the active sequential processes. This claim is supported by the following easy to verify invariant: in any term P reachable from $\mathcal{P}[\![\mathcal{A}]\!]$, for each bound name c in P there is at most one active sequential subterm of P outputting on c. This is satisfied by $\mathcal{P}[\![\mathcal{A}]\!]$ and preserved by reduction. □

Theorem 11.7. *$\mathcal{P}[\![\mathcal{A}]\!]$ is typably hierarchical.*

Proof. Assume an arbitrary strict total order $<_{\mathbb{Q}}$ on the automaton's control states; let then $(\mathcal{T}, \prec)$ be the forest with nodes $\mathcal{T} = \{t_{c^i_q} \mid 0 \leq i \leq \ell, q \in \mathbb{Q}\}$ and $t_{c^i_q} \prec t_{c^i_{q'}}$ if $q <_{\mathbb{Q}} q'$, and $t_{c^i_q} \prec t_{c^{i+1}_{q'}}$ if q and q' are respectively the maximum and minimum states with respect to $<_{\mathbb{Q}}$. It can be proved that $\emptyset \vdash_{\mathcal{T}} \mathrm{nf}(\mathcal{P}[\![\mathcal{A}]\!])$: since no messages are exchanged over channels, the constraints on types are trivially satisfied; for the same reason, no sequential term under an input prefix is migratable, making all the base type constraints in rule In trivially valid. The base type inequalities of rule Par are also satisfied since in A_{tr} for $tr \in \theta_j$, every P_{θ_k} might be tied to any channel c in $\mathcal{C}^{j+1} \cup \ldots \cup \mathcal{C}^i$ but can only have as free names channels in $\mathcal{C}^h$ with $h \leq j$, which all have base types smaller than c. □

11.4 CCS!

The Calculus of Communicating Systems (CCS) introduced in the seminal work by Milner [Mil89] is one of the first process calculi. The π-calculus was conceived as a systematisation and generalisation of CCS capturing mobility, a feature which is missing from pure CCS. Since its introduction, many variants of CCS have been studied [FL10]. The syntax of CCS can be seen as the one of π-calculus where all the actions have arity zero, i.e. sends $\overline{a}.P$ and receives $a.P$ do not carry any message and simply represent synchronisation between two parallel components knowing the same name. Due to the absence of name-passing, representing recursive behaviour is a more subtle issue than it is in π-calculus [BGZ09]. The most general way is by allowing recursive process definitions. This is implemented by using process identifiers, to which a finite set of definitions associates processes, and relabelling, which corresponds to substitution of free names. In this general form, recursive definitions in conjunction with name restrictions lead to a Turing-powerful language.

However, other less general—but practically relevant—constructs for specifying infinite behaviour have been proposed. Removing relabelling and introducing replication, the same we used in the π-calculus, makes the calculus less expressive but more amenable to automated analysis. This variant of CCS is often called CCS!. In [He11] reachability is proved to be decidable for CCS! by reducing it to Petri net reachability.

To explore the relation between CCS! and typably hierarchical terms we need to relate their syntaxes. As we noted in Section 10.6, adding synchronisation on channels of arity zero in the π-calculus we consider is rather easy and all the results of Chapter 10

can be straightforwardly adapted to support this simpler form of communication. In this way, we can see CCS$^!$ as a syntactic subset of π-calculus where only zero-arity channels can be used in prefixes.[2] The semantics of CCS$^!$ can be seen as a special case of the one given in Definition 9.1, where no substitution is necessary. Refer to Figure 10.3 for the specialisation of rules In and Out on zero-arity channel prefixes.

All CCS$^!$ terms are typably hierarchical. To see this is true, we extract from a CCS$^!$ term P, a forest $\mathcal{T}_P$, and annotations for the restrictions of P, that guarantee $\mathcal{T}_P$-shapedness and typability of P. We can assume, w.l.o.g., that P is a closed normal form. Since all the names have arity zero, all the type annotations can be of the form $\nu(x : t_x)$. We construct a forest of base types $\mathcal{T}_P = (N_P, \prec_P)$ from P as follows. The the set of base types is $N_P := \{t_x \mid x \in \mathrm{bn}_\nu(P)\}$. The parent function $\prec_P$ is defined inductively on the structure of P: let $P = \nu X.\prod_{i\in I} A_i$, $X = \{(x_1 : t_{x_1}), \ldots, (x_n : t_{x_n})\}$ and $A_i = \sum_{j\in J_i} \pi_{ij}.P_{ij}$, the parent function $\prec_P$ is the smallest relation such that

1. if $t \prec_{P_{ij}} t'$ then $t \prec_P t'$ (the relation $\prec_{P_{ij}}$ is the inductive case of the definition),
2. $t_{x_1} \prec_P t_{x_2} \prec_P \cdots \prec_P t_{x_n}$ (the types in X form an arbitrary linear order),
3. $t_{x_n} \prec_P t$ for all t that are minimal elements of N with respect to any of the relations $\prec_{P_{ij}}$, i.e. $t \in \min_\prec(N)$ where $\prec = \bigcup\{\prec_{P_{ij}} \mid i \in I, j \in J_i\}$ (the roots of the inductive cases are the children of t_{x_n}).

In the base case $P = \mathbf{0}$, none of the items apply and $\prec_\mathbf{0}$ is the empty relation. When $X = \emptyset$ only item 1 applies.

Intuitively the forest of base types $\mathcal{T}_P$ is simply a forest that reflects the static nesting structure of P: a restriction νx which is nested under νy in P is mirrored by the base type t_x being a descendent of t_y in $\mathcal{T}_P$. By construction—as enforced by item 2—P is $\mathcal{T}_P$-shaped.

To see that P is typable, we can run the inference algorithm of Section 10.5. There are no constraints on types as there are no arguments in prefixes. Rules In-0 and Out-0 do not impose any constraints on base types. The only rule that constraints base types is Par: item 3 above guarantees that, in any application of the rule, any free name in a subterm A_i has a base type which is an ancestor of the base type of any restriction in X.

This shows that every CCS$^!$ term is typably hierarchical. Our type system, however, becomes trivial when mobility is absent and adds little insight on the term.

Example 11.1 (Adapted from [He11]). The normal form

$$\nu(b : \mathsf{b}).\big(!\big(b.\nu(a : \mathsf{a}).(!a.\overline{b} \parallel\ !\overline{a}.\overline{b})\big) \parallel \overline{b}\big)$$

is a CCS$^!$ term. It can be typed using base types $\mathcal{T}$ with $\mathsf{b} \prec_\mathcal{T} \mathsf{a}$.

[2] Technically [He11] uses unguarded replication and a labelled semantics. Both differences do not impact on the presented results.

11.5 Other Related Work

Type Systems for π-calculus

Simple types for (polyadic) π-calculus channels were introduced by Milner with the name of 'sorts' [Mil93]. Their main motivation was to rule out terms which used channels inconsistently with respect to their arity. For instance, the term

$$P = a(x, y).\overline{x}\langle y\rangle \parallel \big(\nu b\, c.\overline{a}\langle b, c\rangle.b(z, w)\big)$$

can reduce, by correctly synchronising on a, to $\nu b\, c.\big(\overline{b}\langle c\rangle \parallel b(z, w)\big)$ which cannot proceed with a synchronisation on b since it is used with different arities by the two sequential processes. This is considered a programming error. Simple types can statically detect this issue. The term in P inputing on a is assuming a simple type $a : \mathsf{ch}[\mathsf{ch}[\mathsf{ch}], \mathsf{ch}]$, that is a is a channel on which one can exchange messages composed of couples of channels ($\mathsf{ch}[\_, \_]$); the first channel of such a message must be a channel on which one can exchange messages consisting of a channel name. Since the type of b, as used by the second component, is $b : \mathsf{ch}[\mathsf{ch}, \mathsf{ch}]$ the process outputting on a is typing this channel as $a : \mathsf{ch}[\mathsf{ch}[\mathsf{ch}, \mathsf{ch}], \mathsf{ch}]$. This clash of assumptions means that the two sequential components of P cannot be safely composed. Note that in order to find the problem, it is necessary to realise that b can flow to x and this is done by using the nested structure of channel types.

In [Gay93] inference of simple types is proved decidable and a practical procedure is presented. A similar result for recursive simple types is shown in [VH93].

Our type system is based on (non-recursive) simple types annotated with nodes in $\mathcal{T}$. This structure is mainly used to represent data-flow information so we can reason about the use of names locally in the typing rules.

Milner's sorts were later refined into I/O types [PS93] and their variants [PS00], which allow finer constraints on the use of channels. Based on these types is a system for termination of π-terms [DS06] that uses a notion of levels, enabling the definition of a lexicographical ordering. Our type system can also be used to determine termination of π-terms in an approximate but conservative way, by composing it with a procedure for deciding termination of depth-bounded systems. Because the respective orderings between types of the two approaches are different in conception, we expect the terminating fragments isolated by the respective systems to be incomparable.

A rather different approach to typing π-terms is presented in the literature on Behavioural and Session Types [IK01; THK94; HVK98], as already mentioned in Section 7.6. These type systems are based on types that carry information about the sequence of actions that can happen on a channel. By contrast, our types do not carry information about the evolution of the system; if a system is proved depth-bounded by the type system, its evolution can be analysed quite accurately using the decision procedures for depth-bounded systems in a second stage.

Other Fragments of π-calculus

Since its conception, there have been many studies about the relative expressivity of the primitives of CCS and π-calculus [BGZ09; FL10; Pal03]. Syntactically restricting the use of primitives results in fragments with decidable properties in some cases. As we have seen in Section 11.4 restricting channels to have arity zero and using replication makes reachability decidable. Another common fragment restricts the use of replication and parallel: *finite-control* terms are π-terms where parallel is never used under replication. The finite-control fragment is the syntactic restriction enforcing that the term is *finitary*, that is, the number of parallel active sequential subterms of any reachable term is bounded by a constant.

Several other interesting fragments of the π-calculus have been proposed in the literature, such as name bounded [HMM13], mixed bounded [MG09], and structurally stationary [Mey09a]. Typically defined by a non-trivial condition on the set of reachable terms—a *semantic* property, membership becomes undecidable. Links with Petri nets via encodings of proper subsets of depth-bounded systems have been explored in [MG09]. Our type system can prove depth-boundedness for processes that are breadth and name unbounded, and which cannot be simulated by Petri nets. Amadio and Meyssonnier [AM02] consider fragments of the asynchronous π-calculus and show that coverability is decidable for the fragment with no mobility and bounded number of active sequential processes, via an encoding to Petri nets. Typably hierarchical systems can be seen as an extension of the result for a synchronous π-calculus with unbounded sequential processes and a restricted form of mobility.

Recently Hüchting et al. [RM14] proved several relative classification results between fragments of π-calculus. Using an acceleration technique based on Karp-Miller trees, they presented an algorithm to decide if an arbitrary π-term is bounded in depth by a given k. The construction is based on an (accelerated) exploration of the state space of the π-term which can be computationally expensive. By contrast, our type system uses a very different technique leading to a quicker algorithm, at the expense of precision. Our forest-structured types can also act as specifications, offering more intensional information to the user than just a bound k.

Automata on infinite alphabets

Automata that support name reasoning have been used to model the π-calculus, going back to the pioneering work of *History-Dependent Automata* [MP97]. More recently, Tzevelekos [Tze11] introduced *Fresh-Register Automata* (FRA), which operate on an infinite alphabet of names and use a finite number of registers to process fresh names; crucially it can compare incoming names with previously stored ones. He showed that *finitary π-terms* (i.e. processes that do not grow unboundedly in parallelism) are finitely representable in FRA.

Static analysis

Static analyses reasoning about the communication topologies of concurrent programs based on message passing have been studied both in the context of the π-calculus and of practical programming languages. Colby [Col95] considers the *Concurrent ML* (CML) programming language [Rep93] and presents a static analysis based on abstract interpretation answering the question: *which occurrences of 'send' can match which occurrences of 'receive'?* CML is a programming language featuring higher-order functional computations and synchronous message-passing concurrency primitives. It is therefore impossible to answer the question accurately and automatically. Colby proposes a polynomial non-uniform analysis able to distinguish between channel names created in different iterations of a recursive definition. The abstract representations of pids exploits the structure of the call-stack at the point of creation to distinguish unboundedly many pids. The obtained precision is very good when the structure of the recursive calls creating new names is closely related to the communication topology. Although mobility is supported, when the communication topology of the system evolves during time in non trivial ways, the analysis becomes very imprecise. Despite this limitation, the analysis can be very useful in practice for compiler optimisations if common patterns of functional concurrent programming.

Venet [Ven98] proposed an advanced analysis of communication topologies specifically designed for the π-calculus. Structured as an abstract interpretation of a non-standard semantics, the analysis can be instantiated with several domains abstracting relations between (dynamic) channel names. Interestingly, the kinds of topologies that can be accurately captured with these algorithms are incomparable with those captured by depth boundedness: the analysis may be imprecise for depth-bounded systems but has the ability of representing accurately some depth-unbounded systems.

Chapter 12

Extensions and Future Directions

12.1 More Expressive Types

The type system we presented in Section 10.3 is very conservative: the use of simple types, for example, renders the analysis context-insensitive. Although we have kept the system simple so as to focus on the novel aspects, a number of improvements are possible. First, the extension to the polyadic case is straightforward. Second, the type system can be made more precise by using subtyping and polymorphism to refine the analysis of control and data flow. Third, the typing rule for replication introduces a very heavy approximation: when typing a subterm, we have no information about which other parts of the term (crucially, which restrictions) may be replicated.

Handling replication

Let us explain the issue through an example. Consider the two terms $P_1 = \nu a.A$ and $P_2 = \nu a.(!A)$ where

$$A = \boldsymbol{\tau}.\nu b.\boldsymbol{\tau}.\nu c.\, !\big(\overline{a}\langle c\rangle + a(x).\overline{b}\langle x\rangle\big)$$

The typing derivations for $\emptyset \vdash_{\mathcal{T}} P_1$ and $\emptyset \vdash_{\mathcal{T}} P_2$ are almost identical and the set of constraints they impose on $\mathcal{T}$ is the same. However, while P_1 is depth-bounded, P_2 is not, and therefore the type system must reject both.

We briefly sketch a possible enhancement that is sensitive to replication. Take the depth-bounded term

$$\nu(b : t_b[t]).\nu(l : t_l[t]).\nu(r : t_r[t]).\, !\big(b(x).l(y).(\overline{r}\langle x\rangle \parallel \overline{b}\langle y\rangle)\big)$$

which acts as a 1 cell buffer between l and r. This term cannot be typed by the current type system because $l(y).(\overline{r}\langle x\rangle \parallel \overline{b}\langle x\rangle)$ is migratable for the input $b(x)$ thus requiring $t_l < t_b$, but at the same time $\overline{b}\langle y\rangle$ is migratable for $l(y)$ requiring $t_b < t_l$, leading to contradiction.

We propose to add to the structure of $\mathcal{T}$ a notion of multiplicities of base types; a base type can be marked with either 1 or ω. Suppose the forest of a term has a path p from a node n to a node n' where the trace of p consists only of base types marked with 1. This situation will represent the fact that no branching will ever occur between the two replications corresponding to n and n' and having one of the two names in the scope guarantees that the other one is in the scope too. In other words, all the restrictions represented by nodes in p can be though as a indivisible unit; when typing an input term on a name with base type t, the constraints of rule In can be relaxed to require the free variables of migratable terms to have base types smaller than the lowest t' such that the path between t and t' in $\mathcal{T}$ is formed only of base types with multiplicity 1.

In the case of buffer example, we observe that b, l and r could all be assigned base types of multiplicity 1 thus replacing the two conflicting constraints with the constraints $t_l \leq t'$ and $t_b \leq t'$ where t' is the greatest among t_l, t_r and t_b. The formalisation and validation of this extension is a topic of ongoing research.

More liberal migration

Another limitation of the current type system derives from the migration scheme supported by the proof of Theorem 10.10. This migration scheme purposely avoids rearranging the scope of restrictions which are already active and mobility is realised by activating restrictions under the appropriate scopes so that they do not need to be moved to other scopes from then on.

Relaxing this mechanism is not easy when the goal is to guarantee depth boundedness. A certain degree of replication sensitivity, as outlined above, seems necessary to determine conditions for the safe 'transplant' of subtrees on reduction.

12.2 Semantic Notion of Hierarchical System

In Section 10.1 we have structured the proofs of depth boundedness in a way that highlighted the semantic phenomenon we were trying to capture with the type system in Section 10.3. The use of $(\mu, \mathcal{T})$-compatibility leads to a notion which coincides with depth boundedness. A natural question then is: *is there an intermediate (semantic) notion of hierarchical system between depth-bounded and typably hierarchical?* In this section we give a positive answer and mention open problems stemming from it.

One of the key features of how reductions are treated in the proof of soundness of the type system, is that the already active restrictions do not get rearranged. In contrast, the use of α-renaming in conjunction with μ can reassign nodes to active names at will in order to prove compatibility of a reachable term, as shown in the proof of Proposition 10.1.

Consider the following variant of the semantics. In this section we will only consider type annotations of the form $\nu(x : t)$ with $t \in \mathcal{T}$: the nested version used in the type

system is only used to statically approximate the data-flow, but in the definition of a semantic property we have access to the full dynamic information of a name.

We parametrise the semantics with an annotating oracle Ann which is a function from the already assigned types and the newly created names to the annotations for the new names:

$$\textsc{Ann}(W, Y) = Y'$$

where W and Y are the (base) type environments of existing and to be activated restrictions respectively, and $\mathrm{dom}(Y) = \mathrm{dom}(Y')$. In other words, Ann is used to associate a base type to restrictions when they become active.

Definition 12.1 (Annotated Semantics of π-calculus)**.** The Ann-annotated operational semantics of π-calculus is defined by the transition system on annotated terms, with transitions satisfying $P \to_{\textsc{Ann}} Q$ if

(i) $P \equiv \nu W.(S \parallel R \parallel C) \in \mathcal{P}^{\mathcal{T}}_{\mathsf{nf}}$,

(ii) $S = (\overline{a}\langle b\rangle.\nu Y_s.S') + M_s$,

(iii) $R = (a(x).\nu Y_r.R') + M_r$,

(iv) $Q \equiv \nu W Y'.(S' \parallel R'[\,b/x\,] \parallel C)$,

where $Y' = \textsc{Ann}(W, Y_s Y_r)$, or if

(i) $P \equiv \nu W.(\boldsymbol{\tau}.\nu Y.P' \parallel C) \in \mathcal{P}_{\mathsf{nf}}$,

(ii) $Q \equiv \nu W Y'.(P' \parallel C)$,

where $Y' = \textsc{Ann}(W, Y)$.

We define the set $\mathrm{Reach}(\textsc{Ann}, P) := \{\, Q \mid P \to^*_{\textsc{Ann}} Q \,\}$, writing $\to^*_{\textsc{Ann}}$ to mean the reflexive, transitive closure of $\to_{\textsc{Ann}}$.

Definition 12.2 (Hierarchical term)**.** Let $\mathcal{T}$ be a finite height forest (can have infinitely many nodes). A $\mathcal{T}$-annotated term P is *hierarchical* if there exists an annotating oracle Ann such that each $Q \in \mathrm{Reach}(\textsc{Ann}, P)$ is $\mathcal{T}$-compatible.

Lemma 12.1. *Every hierarchical term is depth-bounded.*

Proof. $\mathcal{T}$-compatibility of each reachable term implies that the depth of each reachable term is bounded by $\mathrm{height}(\mathcal{T})$. □

Lemma 12.2. *Not all depth-bounded terms are hierarchical.*

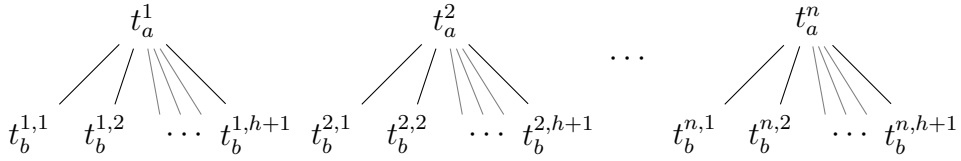
Proof. We show this by presenting a depth-bounded process which is not hierarchical. Take $P = (!A \parallel !B \parallel !(C_1 + C_2))$ where

$$A = \tau.\nu(a:\mathsf{a}).\overline{p}\langle a\rangle \qquad C_1 = p(x).!(q(y).D) \qquad D = \overline{x}\langle y\rangle$$
$$B = \tau.\nu(b:\mathsf{b}).\overline{q}\langle b\rangle \qquad C_2 = q(x).!(p(y).D)$$

then P is depth-bounded. However we can show there is no choice for Ann and $\mathcal{T}$ that can prove it hierarchical. Let h be the height of $\mathcal{T}$. From P we can reach, by reducing the τ actions of A and B, any of the terms $Q_{i,j} = P \parallel (\nu a.\overline{p}\langle a\rangle)^i \parallel (\nu b.\overline{q}\langle b\rangle)^j$ (omitting annotations) for $i, j \in \mathbb{N}$. The function Ann can potentially assign a different type in $\mathcal{T}$ to each νa and νb. Let $n, m \in \mathbb{N}$ be naturals strictly greater than $2h$ and consider the reachable term $Q_{n,nm}$; from this term we can reach a term

$$Q^{ab} = P \parallel \Big(\nu a.\Big(\big(\nu b.D[\,a/x,\ b/y\,]\big)^m \parallel !\big(q(y).D[\,a/x\,]\big)\Big)\Big)^n$$

by never selecting C_2 as part of a redex. Each occurrence of a and b will have an annotation as dictated by Ann. Each occurrence of νa in Q^{ab} has in its scope more than h occurrences of νb. Assume Ann assigned a type t_a^i to each occurrence $i \leq n$ of νa in Q^{ab} and a type $t_b^{i,j}$ to each occurrence j of νb under $\nu(a : t_a^i)$ in Q^{ab}. We cannot extrude more than h occurrences of νb because we would necessarily violate $\mathcal{T}$-compatibility by obtaining a path of length greater than h in the forest of the extruded term. Therefore, w.l.o.g., we can assume that the types $t_b^{i,1}, \ldots, t_b^{i,h+1}$ are all descendants of t_a^i, for each $i \leq n$. Pictorially, the parent relation in $\mathcal{T}$ entails the relations



where the edges represent $<_{\mathcal{T}}$. The type associations of the restrictions in Q^{ab} are already fixed in $Q_{n,nm}$. From $Q_{n,nm}$ we can however also reach any of the terms

$$Q_b^i = P \parallel \cdots \parallel \Big(\nu(b : t_b^{i,1}).\Big(\big(\nu a.D[\,a/x,\ b/y\,]\big)^n \parallel !\big(p(y).D[\,b/x\,]\big)\Big)\Big)$$

for $i \leq m$, by making C_2 and $\nu(b : t_b^{i,1}).\overline{q}\langle b\rangle$ react and then repeatedly making $!q(y).D$ react with each $\nu(a : t_a^j).\overline{p}\langle a\rangle$. Let us consider Q_b^1. As before, we cannot extrude more than h occurrences of a or we would break $\mathcal{T}$-compatibility. We must however extrude $(a : t_a^1)$ to get $\mathcal{T}$-compatibility since $t_a^1 <_{\mathcal{T}} t_b^{1,1}$. From these two facts we can infer that there must be a type associated to one of the a, let it be t_a^2, such that $t_a^1 <_{\mathcal{T}} t_b^{1,1} <_{\mathcal{T}} t_a^2$. We can apply the same argument to Q_b^2 obtaining $t_a^1 <_{\mathcal{T}} t_a^2 <_{\mathcal{T}} t_b^{2,1} <_{\mathcal{T}} t_a^3$. Since $m > 2h$ we can repeat this $h+1$ times and get $t_a^1 <_{\mathcal{T}} t_a^2 <_{\mathcal{T}} \ldots <_{\mathcal{T}} t_a^{h+1}$ which contradicts the assumption that the height of $\mathcal{T}$ is h. □

The reason why the counterexample presented in the proof above fails to be hierarchical is that (unboundedly many) names are used in fundamentally different ways regarding their relative degree of sharing, in different branches of the execution.

Being a *semantic* property, i.e. a non-trivial property of the set of reachable terms, checking if a term is hierarchical has good chances to be undecidable. In fact it can be proved undecidable using the same proof for the undecidability of depth boundedness in [Mey09b]. The result is proved by showing that the folklore encoding of 2-counters machines into π-calculus (which uses a variation of the stack process of Example 9.3 to implement counters with zero-tests) preserves termination. Then, if checking if a term is hierarchical were decidable, one could decide termination of a 2-counters machine M by

1. taking the π-calculus encoding P of M
2. deciding if P is hierarchical:
    - if it is, then it is also depth-bounded and termination can be decided;
    - if it is not, then it does not terminate.

This is clearly a contradiction since termination for 2-counters machines is undecidable.

The notion of hierarchical system can be modified by further constraining how base types are associated to names. It would be interesting to investigate the consequences of these variations on complexity of verification tasks or on expressivity.

Another direction is to relax the type associations so that some reassignment of types during execution is possible, increasing the expressivity of the fragment.

12.3 Complexity

It would be interesting to further study the above definition of hierarchical system under the lens of complexity theory: is checking coverability for hierarchical systems cheaper than it is for depth-bounded ones? Imposing different constraints on Ann can also lead to further improvements in the asymptotic complexity of verification problems.

Typably hierarchical terms have a very structured execution. This structure could lead to more efficient algorithms for their analysis.

Chapter 13

Conclusions

13.1 Summary

We investigated issues in automatic verification of concurrent message passing systems. These systems are very hard to analyse algorithmically due to the many sources of unboundedness in their semantics.

We analysed each of the problematic features and proposed some abstractions that can capture salient aspects of concurrent computation. We have defined a generic analysis for a substantial fragment of Erlang, λActor, and a way of extracting from the analysis a simulating infinite-state abstract model in the form of an ACS, which can be automatically verified for coverability: if a state of the abstract model is not coverable then the corresponding concrete states of the input λActor program are not reachable. Our constructions are parametric, thus enabling different analyses (implementing varying degrees of precision with different complexity bounds) to be easily instantiated. In particular, with a 0-CFA-like specialisation of the framework, the analysis and generation of the ACS are computable in polynomial time. Further, the dimension of the resulting ACS is polynomial in the length of the input λActor program, small enough for the verification problem to be tractable in many useful cases. The empirical results using our prototype implementation Soter are encouraging. They demonstrate that the abstraction framework can be used to prove interesting safety properties of non-trivial programs automatically.

Building on this experience, we focused our attention on a specific source of imprecision in our abstractions: the representation of the identities of dynamically created processes. To isolate the essence of the issue we turn to the π-calculus, a succinct process algebra revolving around the concept of *names*. To enable algorithmic analysis, one must put constraints on the shape of terms. We consider depth-bounded π-calculus, which is, to the best of our knowledge, the most expressive known fragment of π-calculus supporting algorithmic verification. Terms modelling Erlang programs are not always depth-bounded: it is necessary to have a procedure to determine if a term is depth-

bounded in order to apply the corresponding verification procedures. Unfortunately, depth boundedness is an undecidable property. We solve this problem by proposing a notion of *hierarchical system*, a more structured subset of depth-bounded systems. We define a type system that can algorithmically check (or infer) whether an arbitrary term is hierarchical. Part II was devoted to the definition of the principles governing this novel fragment of π-calculus.

13.2 Future Work

To conclude, we briefly discuss ideas for extensions and future directions.

Static Analysis In Chapter 5 we have defined a generic analysis for λActor, and a way of extracting from the analysis a simulating infinite-state abstract model. Our constructions are parametric on the abstractions for $Time$, $Mailbox$ and $Data$, thus enabling different analyses to be instantiated. It would be interesting to explore the design space for these abstractions tuning accuracy and performance for specific properties. Also, as mentioned in Section 7.3, the CFA phase of the analysis could be extended to support optimisations like abstract garbage collection [MS06] or stack reasoning [VS10] to bring performance and accuracy improvements to the framework. We also believe that the proposed abstraction technique can easily be adapted to accommodate other languages and other abstract models.

Soter Soter currently supports only a subset of the full Erlang language but the techniques it employs are general enough to support the missing features. Two characteristics of Erlang could prove to have a fruitful interaction with Soter. First, Erlang definitions are frequently annotated with specifications based on types; this could enable the use of Soter for intra-module verification. Second, programs using the *Open Telecoms Platform* (OTP), the standard generic library of behaviours distributed with Erlang, share a common fixed structure; specialising Soter's analysis for OTP-based software could improve performance and accuracy greatly.

Another possible extension of Soter is adding support for analysis of counter-examples generated by the model checking phase. The level of generality at which the algorithm is defined seems to support the definition of a *Counter-Example Guided Abstract Refinement* (CEGAR) loop readily. In a CEGAR loop, when the model checker fails to prove a property, it produces a counter-example, usually in the form of a trace violating the property. The trace can be analysed to detect whether it witnesses a genuine bug or a spurious behaviour. In the latter case the abstraction's precision is tuned so the trace, and other similar ones, is removed from the abstract model and the procedure is run again.

Hierarchical Systems First and foremost, we plan to implement the type system of Section 10.3 to evaluate the approach empirically. An implementation could also be integrated in Soter: Soter's analysis could be exploited to generate a π-calculus term

simulating the program, instead of an ACS. The term could be analysed using the type system. Typably hierarchical terms could then be feeded to a dedicated depth-bounded systems model checker as Picasso[1] [BKWZ12; ZWH12]. Terms that are rejected by the type system could still be analysed using the ACS abstraction.

The extraction of a π-calculus model from Erlang programs is an interesting problem in itself. The fact that the π-calculus is Turing-powerful means that it is in principle capable of encoding precisely any of Erlang's features. Keeping the π-calculus encoding 100% accurate quickly degenerates into writing an Erlang interpreter in π-calculus, which is clearly a bad choice for analysis purposes. Instead, multiple approximated π-calculus models could be extracted, each focusing on different aspects of computation by using names to represent different unbounded structures. Through each of these encodings, it would be interesting to explore the interactions between the concept of $\mathcal{T}$-compatibility and Erlang's idioms and primitives.

As detailed in Chapter 12 there are a number of directions for extending the expressivity of the type system. Special treatment for global names, multiplicity constraints, subtyping are all extensions worth exploring. Another point of extension is the semantic notion of Hierarchical system we gave in Section 12.2. Variations of this definition could be devised to characterise other patterns of communication and maybe enable feasible model checking algorithms. Different ways of characterising these semantic fragments statically by means of static analysis is also a topic of interest. Also open is the exact complexity bound of type inference.

Names can be used to give semantics to numerous programming language concepts, from heap cells addresses to process identifiers. Structuring these objects using our notion of hierarchy could enable automatic infinite-state verification. Depth-bounded systems verification can also be applied to *shape analysis* [WSR00] of concurrent programs where the communication topology of π-terms encodes the layout of the heap. The notion of hierarchical system could have interesting implications in this area.

A related research topic is analysis of *graph rewriting systems* where the concept of depth boundedness has been used to define classes of rewriting systems with decidable properties [Ber+12; BKWZ12]. In this context too, depth boundedness remains an undecidable property of rewriting systems and (generalisations of) the hierarchical topology structure we introduced could prove useful.

[1] Available at `http://pub.ist.ac.at/~zufferey/picasso`.

Bibliography

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson and Y. Tsay. 'General Decidability Theorems for Infinite-State Systems'. In: *Symposium on Logic in Computer Science*. IEEE Computer Society, 1996, pp. 313–321 (Cited on p. 18).

[Agh85] G. Agha. 'Actors: A Model of Concurrent Computation in Distributed Systems'. PhD thesis. Massachusetts Institute of Technology, 1985 (Cited on p. 25).

[AJ93] P. A. Abdulla and B. Jonsson. 'Verifying Programs with Unreliable Channels'. In: *Symposium on Logic in Computer Science*. IEEE Computer Society, 1993, pp. 160–170 (Cited on p. 36).

[AM02] R. M. Amadio and C. Meyssonnier. 'On decidability of the control reachability problem in the asynchronous π-calculus'. In: *Nordic Journal of Computing* 9.2 (2002), pp. 70–101 (Cited on p. 146).

[APT79] B. Aspvall, M. F. Plass and R. E. Tarjan. 'A linear-time algorithm for testing the truth of certain quantified boolean formulas'. In: *Information Processing Letters* 8.3 (1979), pp. 121–123 (Cited on p. 122).

[Arm10] J. Armstrong. 'Erlang'. In: *Communications of the ACM* 53.9 (2010), p. 68 (Cited on pp. 26, 28).

[AVW93] J. Armstrong, R. Virding and M. Williams. *Concurrent programming in Erlang*. Prentice Hall, 1993, pp. 1–281 (Cited on p. 26).

[Bae05] J. C. M. Baeten. 'A brief history of process algebra'. In: *Theoretical Computer Science* 335.2-3 (2005), pp. 131–146 (Cited on p. 25).

[Ber+12] N. Bertrand, G. Delzanno, B. König, A. Sangnier and J. Stückrath. 'On the Decidability Status of Reachability and Coverability in Graph Transformation Systems'. In: *Rewriting Techniques and Applications (RTA)*. Ed. by A. Tiwari. Vol. 15. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012, pp. 101–116 (Cited on p. 157).

[BGZ09] N. Busi, M. Gabbrielli and G. Zavattaro. 'On the expressive power of recursion, replication and iteration in process calculi'. In: *Mathematical Structures in Computer Science* 19.6 (2009), pp. 1191–1222 (Cited on pp. 143, 146).

[BKWZ12] K. Bansal, E. Koskinen, T. Wies and D. Zufferey. 'Structural Counter Abstraction'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'13. Rome: Springer-Verlag, 2012, pp. 62–77 (Cited on p. 157).

[BS07] H. Björklund and T. Schwentick. 'On notions of regularity for data languages'. In: *Fundamentals of Computation Theory*. 2007, pp. 88–99 (Cited on p. 131).

[Car01] R. Carlsson. 'An Introduction to Core Erlang'. In: *ACM SIGPLAN Erlang Workshop*. 2001 (Cited on pp. 29, 31).

[CH00] K. Claessen and J. Hughes. 'QuickCheck: a lightweight tool for random testing of Haskell programs'. In: *International Conference on Functional Programming (ICFP)*. 2000, pp. 268–279 (Cited on p. 78).

[CHMO15] C. Cotton-Barratt, D. Hopkins, A. S. Murawski and C.-H. L. Ong. 'Fragments of ML decidable by Nested Data Class Memory Automata'. In: *Foundations of Software Science and Computation Structures (FoSSaCS)*. To appear. 2015 (Cited on p. 131).

[Cla+09] K. Claessen et al. 'Finding race conditions in Erlang with QuickCheck and PULSE'. In: *International Conference on Functional Programming (ICFP)*. 2009, pp. 149–160 (Cited on p. 78).

[Cli81] W. D. Clinger. 'Foundations of Actor Semantics'. PhD thesis. MIT, 1981 (Cited on p. 25).

[CMO15] C. Cotton-Barratt, A. S. Murawski and C. L. Ong. 'Weak and Nested Class Memory Automata'. In: *Language and Automata Theory and Applications*. Ed. by A. H. Dediu, E. Formenti, C. Martín-Vide and B. Truthe. Vol. 8977. Lecture Notes in Computer Science. Springer, 2015, pp. 188–199 (Cited on pp. 85, 131–133).

[Col95] C. Colby. 'Analyzing the Communication Topology of Concurrent Programs'. In: *Partial Evaluation and Semantics-Based Program Manipulation*. Ed. by N. D. Jones. ACM Press, 1995, pp. 202–213 (Cited on pp. 74, 147).

[CS05] K. Claessen and H. Svensson. 'A semantics for distributed Erlang'. In: *ACM SIGPLAN Erlang Workshop*. ERLANG '05. ACM, 2005, pp. 78–87 (Cited on pp. 60, 71).

[CS10] M. Christakis and K. Sagonas. 'Static detection of race conditions in Erlang'. In: *Practical Aspects of Declarative Languages* (2010), pp. 119–133 (Cited on pp. 74, 78).

[CS11] M. Christakis and K. Sagonas. 'Detection of asynchronous message passing errors using static analysis'. In: *Practical Aspects of Declarative Languages* (2011), pp. 5–18 (Cited on pp. 74, 78).

[CSW06] R. Carlsson, K. Sagonas and J. Wilhelmsson. 'Message analysis for concurrent programs using message passing'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2006) (Cited on pp. 72, 74, 78).

[CT09] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009. ISBN: 9780596555856 (Cited on p. 26).

[DKO12] E. D'Osualdo, J. Kochems and C.-H. L. Ong. 'Soter: an Automatic Safety Verifier for Erlang'. In: *Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. AGERE!'12. Tucson, Arizona, USA: ACM, 2012, pp. 137–140 (Cited on p. 24).

[DKO13a] E. D'Osualdo, J. Kochems and C.-H. L. Ong. 'Automatic Verification of Erlang-Style Concurrency'. In: *CoRR* abs/1303.2201 (2013). Available at http://arxiv.org/abs/1303.2201 (Cited on p. 171).

[DKO13b] E. D'Osualdo, J. Kochems and C. L. Ong. 'Automatic Verification of Erlang-Style Concurrency'. In: *Static Analysis Symposium (SAS)*. Ed. by F. Logozzo and M. Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 454–476 (Cited on pp. 24, 171).

[DS06] Y. Deng and D. Sangiorgi. 'Ensuring termination by typability'. In: *Information and Computation* 204.7 (2006), pp. 1045–1082 (Cited on p. 145).

[EMH10] C. Earl, M. Might and D. V. Horn. 'Pushdown Control-Flow Analysis of Higher-Order Programs'. In: *Workshop on Scheme and Functional Programming (Scheme)*. Montreal, Quebec, Canada, Aug. 2010 (Cited on p. 60).

[Fed94] T. Feder. 'Network Flow and 2-Satisfiability'. In: *Algorithmica* 11.3 (1994), pp. 291–319 (Cited on p. 122).

[Fer05] J. Feret. 'Analysis of Mobile Systems by Abstract Interpretation'. PhD thesis. École Polytechnique, 2005 (Cited on p. 74).

[FL10] Y. Fu and H. Lu. 'On the expressiveness of interaction'. In: *Theoretical Computer Science* 411.11-13 (2010), pp. 1387–1451 (Cited on pp. 143, 146).

[Fre01] L. Fredlund. 'A Framework for Reasoning about Erlang Code'. PhD thesis. Stockholm, Sweden: Royal Institute of Technology, 2001 (Cited on p. 71).

[FS01] A. Finkel and P. Schnoebelen. 'Well-structured transition systems everywhere!' In: *Theoretical Computer Science* 256.1-2 (2001), pp. 63–92 (Cited on pp. 18, 19, 38).

[FS07] L. Fredlund and H. Svensson. 'McErlang: a model checker for a distributed functional programming language'. In: *International Conference on Functional Programming (ICFP)*. 2007, pp. 125–136 (Cited on p. 75).

[Gay93] S. J. Gay. 'A Sort Inference Algorithm for the Polyadic π-Calculus'. In: *Principles of Programming Languages (POPL)*. Ed. by M. S. V. Deusen and B. Lang. ACM Press, 1993, pp. 429–438 (Cited on pp. 119, 145).

[GPT06] P. Garoche, M. Pantel and X. Thirioux. 'Static Safety for an Actor Dedicated Process Calculus by Abstract Interpretation'. In: *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. 2006, pp. 78–92 (Cited on p. 74).

[GRV06] G. Geeraerts, J. Raskin and L. Van Begin. 'Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS'. In: *Journal of Computer and system Sciences* 72.1 (2006), pp. 180–203 (Cited on p. 98).

[HBS73] C. Hewitt, P. Bishop and R. Steiger. 'A Universal Modular ACTOR Formalism for Artificial Intelligence'. In: *International Joint Conferences on Artificial Intelligence (IJCAI)*. 1973, pp. 235–245 (Cited on p. 25).

[He11] C. He. 'The Decidability of the Reachability Problem for CCS!' In: *Concurrency Theory (CONCUR)*. Ed. by J. Katoen and B. König. Vol. 6901. Lecture Notes in Computer Science. Springer, 2011, pp. 373–388 (Cited on pp. 86, 143, 144).

[Hei92] N. Heintze. 'Set Based Program Analysis'. PhD thesis. Pittsburgh: Carnegie-Mellon University, 1992 (Cited on p. 73).

[Hei94] N. Heintze. 'Set-based analysis of ML programs'. In: *ACM SIGPLAN Lisp Pointers*. Vol. 7. 1994, pp. 306–317 (Cited on p. 73).

[Hew10] C. Hewitt. 'Actor Model for Discretionary, Adaptive Concurrency'. In: *CoRR* abs/1008.1459 (2010) (Cited on p. 26).

[Hew77] C. Hewitt. 'Viewing control structures as patterns of passing messages'. In: *Artificial Intelligence* 8.3 (June 1977), pp. 323–364. ISSN: 0004-3702 (Cited on p. 25).

[HMM13] R. Hüchting, R. Majumdar and R. Meyer. 'A Theory of Name Boundedness'. In: *Concurrency Theory (CONCUR)*. 2013 (Cited on pp. 84, 97, 146).

[HO09] P. Haller and M. Odersky. 'Scala Actors: Unifying thread-based and event-based programming'. In: *Theoretical Computer Science* 410.2-3 (2009), pp. 202–220 (Cited on p. 26).

[Huc02] F. Huch. 'Model Checking Erlang Programs — Abstracting Recursive Function Calls'. In: 64 (2002), pp. 195–219 (Cited on p. 75).

[Huc99] F. Huch. 'Verification of Erlang Programs using Abstract Interpretation and Model Checking'. In: *International Conference on Functional Programming (ICFP)*. 1999, pp. 261–272 (Cited on pp. 66, 75).

[HVK98] K. Honda, V. T. Vasconcelos and M. Kubo. 'Language Primitives and Type Discipline for Structured Communication-Based Programming'. In: *European Symposium on Programming (ESOP)*. Ed. by C. Hankin. Vol. 1381. Lecture Notes in Computer Science. Springer, 1998, pp. 122–138 (Cited on pp. 77, 145).

[HW87] M. P. Herlihy and J. M. Wing. 'Axioms for concurrent objects'. In: *Principles of Programming Languages (POPL)*. 1987, pp. 13–26 (Cited on p. 78).

[IK01] A. Igarashi and N. Kobayashi. 'A generic type system for the Pi-calculus'. In: *Principles of Programming Languages (POPL)*. 2001, pp. 128–141 (Cited on pp. 77, 145).

[JA07] N. D. Jones and N. Andersen. 'Flow analysis of lazy higher-order functional programs'. In: *Theoretical Computer Science* 375 (2007), pp. 120–136 (Cited on p. 73).

[JLMH13] J. I. Johnson, N. Labich, M. Might and D. V. Horn. 'Optimizing abstract abstract machines'. In: *International Conference on Functional Programming (ICFP)*. ACM, 2013, pp. 443–454 (Cited on p. 60).

[Jon81] N. Jones. 'Flow analysis of lambda expressions'. In: *Automata, Languages and Programming* (1981), pp. 114–128 (Cited on p. 73).

[JW95] S. Jagannathan and S. Weeks. 'A unified treatment of flow analysis in higher-order languages'. In: *Principles of Programming Languages (POPL)*. 1995, pp. 393–407 (Cited on p. 73).

[KKW12] A. Kaiser, D. Kroening and T. Wahl. 'Efficient Coverability Analysis by Proof Minimization'. In: *Concurrency Theory (CONCUR)*. www.cprover.org/bfc/. 2012 (Cited on p. 64).

[KM69] R. M. Karp and R. E. Miller. 'Parallel program schemata'. In: *Journal of Computer and system Sciences* 3.2 (1969), pp. 147–195 (Cited on p. 19).

[KNY95] N. Kobayashi, M. Nakade and A. Yonezawa. 'Static Analysis of Communication for Asynchronous Concurrent Programming Languages'. In: *Static Analysis Symposium (SAS)*. Ed. by A. Mycroft. Vol. 983. Lecture Notes in Computer Science. Springer, 1995, pp. 225–242 (Cited on p. 66).

[Lee06] E. A. Lee. 'The problem with threads'. In: *Computer* 39.5 (2006), pp. 33–42 (Cited on p. 24).

[LS05] T. Lindahl and K. Sagonas. 'TypEr: a type annotator of Erlang code'. In: *ACM SIGPLAN Erlang Workshop*. 2005, pp. 17–25 (Cited on p. 77).

[LS06] T. Lindahl and K. Sagonas. 'Practical type inference based on success typings'. In: *Principles and Practice of Declarative Programming*. 2006, pp. 167–178 (Cited on pp. 74, 76, 78).

[Mey08] R. Meyer. 'On boundedness in depth in the π-calculus'. In: *IFIP International Conference on Theoretical Computer Science, IFIP TCS*. 2008, pp. 477–489 (Cited on pp. 10, 11, 83, 88, 95–98, 102, 134).

[Mey09a] R. Meyer. 'A theory of structural stationarity in the π-calculus'. In: *Acta Informatica* 46.2 (2009), pp. 87–137 (Cited on p. 146).

[Mey09b] R. Meyer. 'Structural stationarity in the π-calculus'. PhD thesis. University of Oldenburg, 2009 (Cited on pp. 11, 97, 99, 153).

[MG09] R. Meyer and R. Gorrieri. 'On the relationship between π-calculus and finite place/transition Petri nets'. In: *Concurrency Theory (CONCUR)*. 2009, pp. 463–480 (Cited on p. 146).

[MH11] M. Might and D. V. Horn. 'A Family of Abstract Interpretations for Static Analysis of Concurrent Higher-Order Programs'. In: *Static Analysis Symposium (SAS)*. Vol. 6887. Lecture Notes in Computer Science. Springer, 2011, pp. 180–197 (Cited on pp. 39, 74).

[Mid12] J. Midtgaard. 'Control-flow analysis of functional programs'. In: *ACM Computing Surveys* 44.3 (2012), p. 10 (Cited on p. 74).

[Mig10] M. Might. 'Abstract Interpreters for Free'. In: *Static Analysis Symposium (SAS)*. Ed. by R. Cousot and M. Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 407–421 (Cited on p. 39).

[Mil89] R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2 (Cited on p. 143).

[Mil92] R. Milner. 'Functions as processes'. In: *Mathematical Structures in Computer Science* 2.02 (1992), pp. 119–141 (Cited on pp. 87, 91).

[Mil93] R. Milner. 'The polyadic pi-calculus: a tutorial'. In: *Logic and Algebra of Specification*. Springer-Verlag, 1993 (Cited on pp. 88, 91, 102, 145).

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999 (Cited on pp. 88, 93).

[MJ08] J. Midtgaard and T. Jensen. 'A calculational approach to control-flow analysis by abstract interpretation'. In: *Static Analysis Symposium (SAS)* (2008), pp. 347–362 (Cited on p. 48).

[MP97] U. Montanari and M. Pistore. 'An Introduction to History Dependent Automata'. In: *Electronic Notes in Theoretical Computer Science* 10 (1997), pp. 170–188 (Cited on p. 146).

[MPW92] R. Milner, J. Parrow and D. Walker. 'A Calculus of Mobile Processes, I, II'. In: *Information and Computation* 100.1 (1992), pp. 1–77 (Cited on pp. 8, 82, 91, 93).

[MS06] M. Might and O. Shivers. ‘Improving flow analyses via ΓCFA: Abstract garbage collection and counting’. In: *ACM SIGPLAN Notices* 41.9 (2006), pp. 13–25 (Cited on pp. 73, 156).

[MW97] S. Marlow and P. Wadler. ‘A Practical Subtyping System For Erlang’. In: *International Conference on Functional Programming (ICFP)*. 1997, pp. 136–149 (Cited on pp. 76, 77).

[Nys03] S. Nyström. ‘A soft-typing system for Erlang’. In: *ACM SIGPLAN Erlang Workshop*. 2003, pp. 56–71 (Cited on pp. 73, 74, 76).

[Oa04] M. Odersky and al. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. EPFL Lausanne, Switzerland, 2004 (Cited on p. 26).

[OR11] C.-H. L. Ong and S. J. Ramsay. ‘Verifying higher-order functional programs with pattern-matching algebraic data types’. In: *Principles of Programming Languages (POPL)*. 2011, pp. 587–598 (Cited on pp. 36, 53).

[Pal03] C. Palamidessi. ‘Comparing the expressive power of the synchronous and asynchronous π-calculi’. In: *Mathematical Structures in Computer Science* 13.05 (2003), pp. 685–719 (Cited on p. 146).

[PG92] Y. G. Park and B. Goldberg. ‘Escape analysis on lists’. In: *ACM SIGPLAN Notices*. Vol. 27. 1992, pp. 116–127 (Cited on p. 78).

[PS00] B. C. Pierce and D. Sangiorgi. ‘Behavioral equivalence in the polymorphic pi-calculus’. In: *Journal of the ACM* 47.3 (2000), pp. 531–584 (Cited on p. 145).

[PS11] M. Papadakis and K. Sagonas. ‘A PropEr Integration of Types and Function Specifications with Property-Based Testing’. In: *ACM SIGPLAN Erlang Workshop*. ACM Press, Sept. 2011, pp. 39–50 (Cited on p. 78).

[PS93] B. C. Pierce and D. Sangiorgi. ‘Typing and Subtyping for Mobile Processes’. In: *Symposium on Logic in Computer Science*. 1993, pp. 376–385 (Cited on p. 145).

[Rac78] C. Rackoff. ‘The Covering and Boundedness Problems for Vector Addition Systems’. In: *Theoretical Computer Science* 6 (1978), pp. 223–231 (Cited on pp. 19, 38).

[Rep93] J. H. Reppy. ‘Concurrent ML: Design, Application and Semantics’. In: *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991–1992, McMaster University, Hamilton, Ontario, Canada*. Ed. by P. E. Lauer. Vol. 693. Lecture Notes in Computer Science. Springer, 1993, pp. 165–198 (Cited on p. 147).

[RM14] R. M. Reiner Hüchting and R. Meyer. ‘Bounds on Mobility’. In: *Concurrency Theory (CONCUR)*. 2014, pp. 357–371 (Cited on pp. 84, 100, 146).

[RX07] J. H. Reppy and Y. Xiao. 'Specialization of CML message-passing primitives'. In: *Principles of Programming Languages (POPL)*. Ed. by M. Hofmann and M. Felleisen. ACM, 2007, pp. 315–326 (Cited on p. 74).

[SF07] H. Svensson and L. Fredlund. 'A more accurate semantics for distributed Erlang'. In: *ACM SIGPLAN Erlang Workshop*. 2007, pp. 43–54 (Cited on p. 71).

[Shi88] O. Shivers. 'Control flow analysis in Scheme'. In: *ACM SIGPLAN Notices* 23.7 (1988), pp. 164–174 (Cited on p. 73).

[Shi91] O. Shivers. 'Control-Flow Analysis of Higher-Order Languages'. PhD thesis. Carnegie Mellon University, 1991 (Cited on pp. 42, 73).

[THK94] K. Takeuchi, K. Honda and M. Kubo. 'An Interaction-based Language and its Typing System'. In: *Parallel Architectures and Languages Europe (PARLE)*. Ed. by C. Halatsis, D. G. Maritsas, G. Philokyprou and S. Theodoridis. Vol. 817. Lecture Notes in Computer Science. Springer, 1994, pp. 398–413 (Cited on pp. 77, 145).

[Tze11] N. Tzevelekos. 'Fresh-register automata'. In: *Principles of Programming Languages (POPL)*. 2011, pp. 295–306 (Cited on p. 146).

[Ven98] A. Venet. 'Automatic Determination of Communication Topologies in Mobile Systems'. In: *Static Analysis Symposium (SAS)*. Ed. by G. Levi. Vol. 1503. Lecture Notes in Computer Science. Springer, 1998, pp. 152–167 (Cited on pp. 74, 147).

[VH93] V. T. Vasconcelos and K. Honda. 'Principal Typing Schemes in a Polyadic π-Calculus'. In: *Concurrency Theory (CONCUR)*. Ed. by E. Best. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 524–538 (Cited on pp. 119, 145).

[VM10] D. Van Horn and M. Might. 'Abstracting abstract machines'. In: *International Conference on Functional Programming (ICFP)*. 2010, pp. 51–62 (Cited on pp. 40, 52, 58, 73).

[VS10] D. Vardoulakis and O. Shivers. 'CFA2: A Context-Free Approach to Control-Flow Analysis'. In: *European Symposium on Programming (ESOP)*. Ed. by A. D. Gordon. Vol. 6012. Lecture Notes in Computer Science. Springer, 2010, pp. 570–589 (Cited on pp. 74, 156).

[WSR00] R. Wilhelm, M. Sagiv and T. Reps. 'Shape analysis'. In: *Compiler Construction*. 2000, pp. 1–17 (Cited on p. 157).

[WZH10] T. Wies, D. Zufferey and T. Henzinger. 'Forward analysis of depth-bounded processes'. In: *Foundations of Software Science and Computation Structures (FoSSaCS)*. 2010, pp. 94–108 (Cited on p. 98).

[ZWH12] D. Zufferey, T. Wies and T. Henzinger. 'Ideal abstractions for well-structured transition systems'. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2012, pp. 445–460 (Cited on pp. 98, 99, 157).

Appendices

Appendix A

Proof of Soundness of the CFA

The material presented in this appendix appeared in [DKO13b; DKO13a] and is joint work with Jonathan Kochems and Luke Ong.

A.1 Abstract Domains, Orders, Abstraction Functions and Abstract Auxiliary Functions

In Chapter 5 we simplified notation by overloading the abstraction function symbol α. Here we give the explicit definitions of the abstract domains for completeness. In what follows we write $f + g := \{(x, x') \mid (x, x') \in f \text{ or } (x, x') \in g\}$. Also recall from Section 5.2 that when B is a flat domain, the abstraction of a partial map $C = A \rightharpoonup B$ to $\widehat{C} = \widehat{A} \to \mathscr{P}(\widehat{B})$ is defined as

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

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

Let us first spell out the underlying orders and abstraction functions of each abstract domain.

$$\begin{aligned}
\widehat{Pid} &:= ProgLoc \times \widehat{Time} \\
\leq_{\text{pid}} &:= = \times \leq_{\text{t}} \\
\alpha_{\text{pid}} &:= \text{id} \times \alpha_{\text{t}} \\
\widehat{VAddr} &:= \widehat{Pid} \times Var \times \widehat{Data} \times \widehat{Time} \\
\leq_{\text{va}} &:= \leq_{\text{pid}} \times = \times \leq_{\text{d}} \times \leq_{\text{t}} \\
\alpha_{\text{va}} &:= \alpha_{\text{pid}} \times \text{id} \times \alpha_{\text{d}} \times \alpha_{\text{t}} \\
\widehat{Env} &:= Var \rightharpoonup \widehat{VAddr}
\end{aligned}$$

$$\hat{\rho} \leq_{\text{env}} \hat{\rho}' \iff \forall x \in \mathit{Var} \,.\, \hat{\rho}(x) \leq_{\text{va}} \hat{\rho}'(x)$$
$$\alpha_{\text{env}}(\rho)(x) := \alpha_{\text{va}}(\rho(x))$$
$$\widehat{\mathit{KAddr}} := \widehat{\mathit{Pid}} \times \mathit{ProgLoc} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Time}}$$
$$\leq_{\text{ka}} := \leq_{\text{pid}} \times = \times \leq_{\text{env}} \times \leq_{\text{t}}$$
$$\alpha_{\text{ka}} := \alpha_{\text{pid}} \times \mathsf{id} \times \alpha_{\text{env}} \times \alpha_{\text{t}}$$
$$\widehat{\mathit{Closure}} := \mathit{ProgLoc} \times \widehat{\mathit{Env}}$$
$$\leq_{\text{cl}} := = \times \leq_{\text{env}}$$
$$\alpha_{\text{cl}} := \mathsf{id} \times \alpha_{\text{env}}$$
$$\widehat{\mathit{Value}} := \widehat{\mathit{Closure}} \uplus \widehat{\mathit{Pid}}$$
$$\leq_{\text{val}} := \leq_{\text{cl}} + \leq_{\text{pid}}$$
$$\alpha_{\text{val}} := \alpha_{\text{cl}} + \alpha_{\text{pid}}$$
$$\widehat{\mathit{ProcState}} := (\mathit{ProgLoc} \uplus \widehat{\mathit{Pid}}) \times \widehat{\mathit{Env}} \times \widehat{\mathit{KAddr}} \times \mathit{Time}$$
$$\leq_{\text{ps}} := (= + \leq_{\text{pid}}) \times \leq_{\text{env}} \times \leq_{\text{ka}} \times \leq_{\text{t}}$$
$$\alpha_{\text{ps}} := (\mathsf{id} + \alpha_{\text{pid}}) \times \alpha_{\text{env}} \times \alpha_{\text{ka}} \times \alpha_{\text{t}}$$
$$\widehat{\mathit{Procs}} := \widehat{\mathit{Pid}} \to \mathscr{P}(\widehat{\mathit{ProcState}})$$
$$\hat{\pi} \leq_{\text{proc}} \hat{\pi}' \iff \forall \hat{\imath} \in \widehat{\mathit{Pid}} \,.\, \hat{\pi}(\hat{\imath}) \subseteq \hat{\pi}'(\hat{\imath})$$
$$\alpha_{\text{procs}}(\pi)(\hat{\imath}) := \{\alpha_{\text{ps}}(\pi(\iota)) \mid \alpha_{\text{pid}}(\iota) = \hat{\imath}\}$$
$$\widehat{\mathit{Mailboxes}} := \widehat{\mathit{Pid}} \to \widehat{\mathit{Mailbox}}$$
$$\hat{\mu} \leq_{\text{ms}} \hat{\mu}' \iff \forall \hat{\imath} \in \widehat{\mathit{Pid}} \,.\, \hat{\mu}(\hat{\imath}) \leq_{\text{m}} \hat{\mu}'(\hat{\imath})$$
$$\alpha_{\text{ms}}(\mu)(\hat{\imath}) := \bigsqcup \{\alpha_{\text{m}}(\mu(\iota)) \mid \alpha_{\text{pid}}(\iota) = \hat{\imath}\}$$
$$\widehat{\mathit{Store}} := (\widehat{\mathit{VAddr}} \to \mathscr{P}(\widehat{\mathit{Value}})) \times (\widehat{\mathit{KAddr}} \to \mathscr{P}(\widehat{\mathit{Kont}}))$$
$$\hat{\sigma} \leq_{\text{st}} \hat{\sigma}' \iff \forall \hat{b} \in \widehat{\mathit{VAddr}} \,.\, \hat{\sigma}(\hat{b}) \subseteq \hat{\sigma}'(\hat{b})$$
$$\forall \hat{a} \in \widehat{\mathit{KAddr}} \,.\, \hat{\sigma}(\hat{a}) \subseteq \hat{\sigma}'(\hat{a})$$
$$\alpha_{\text{st}}(\sigma)(\hat{b}) := \{\alpha_{\text{val}}(\sigma(b)) \mid \alpha_{\text{va}}(b) = \hat{b}\}, \hat{b} \in \widehat{\mathit{VAddr}}$$
$$\alpha_{\text{st}}(\sigma)(\hat{a}) := \{\alpha_{\text{kont}}(\sigma(a)) \mid \alpha_{\text{ka}}(a) = \hat{a}\}, \hat{a} \in \widehat{\mathit{KAddr}}$$
$$\widehat{\mathit{State}} := \widehat{\mathit{Procs}} \times \widehat{\mathit{Mailboxes}} \times \widehat{\mathit{Store}}$$
$$\leq := \leq_{\text{procs}} \times \leq_{\text{ms}} \times \leq_{\text{st}}$$
$$\alpha_{\text{cfa}} := (\mathsf{id} + \alpha_{\text{pid}}) \times \alpha_{\text{env}}$$

The abstract counterparts of the auxiliary functions $\text{new}_{\text{kpush}}$, new_{kpop}, new_{va} and

$\mathrm{new}_{\mathrm{pid}}$ are straightforward liftings of the concrete functions:

$$\widehat{\mathrm{new}_{\mathrm{kpush}}} : \widehat{Pid} \times \widehat{ProcState} \to \widehat{KAddr}$$
$$\widehat{\mathrm{new}_{\mathrm{kpush}}}(\hat{\imath}, (\ell, \hat{\rho}, \_, \hat{t}\,)) := (\hat{\imath}, \ell.\mathsf{arg}_0, \hat{\rho}, \hat{t}\,)$$

$$\widehat{\mathrm{new}_{\mathrm{kpop}}} : \widehat{Pid} \times \widehat{Kont} \times \widehat{ProcState} \to \widehat{KAddr}$$
$$\widehat{\mathrm{new}_{\mathrm{kpop}}}(\hat{\imath}, \hat{\kappa}, \langle \_, \_, \_, \hat{t}\,\rangle) := (\hat{\imath}, \ell.\mathsf{arg}_{i+1}, \hat{\rho}, \hat{t}\,)$$
$$\text{where } \kappa = \mathsf{Arg}_i\langle \ell, \ldots, \hat{\rho}, \_\rangle$$

$$\widehat{\mathrm{new}_{\mathrm{va}}} : \widehat{Pid} \times Var \times \widehat{Data} \times \widehat{ProcState} \to \widehat{VAddr}$$
$$\widehat{\mathrm{new}_{\mathrm{va}}}(\hat{\imath}, x, \hat{\delta}, \langle \_, \_, \_, \hat{t}\,\rangle) := (\hat{\imath}, x, \hat{\delta}, \hat{t}\,)$$
$$\widehat{\mathrm{new}_{\mathrm{pid}}} : \widehat{Pid} \times ProgLoc \times \widehat{Time} \to \widehat{Pid}$$
$$\widehat{\mathrm{new}_{\mathrm{pid}}}((\ell', \hat{t}'), \ell, \hat{t}\,) := (\ell, \widehat{\mathrm{tick}}^*(\hat{t}, \widehat{\mathrm{tick}}(\ell', \hat{t}')))$$

Concrete and Abstract Match Function

Given two substitutions θ and θ', we define the associative and commutative operator

$$\theta \otimes \theta' = \begin{cases} \bot & \text{if } \exists x.\ \theta(x) \neq \theta'(x) \\ \theta \cup \theta' & \text{otherwise} \end{cases}$$

extending it to finite sets of substitutions: $\bigotimes \{\theta_1, \ldots, \theta_n\} = \theta_1 \otimes \theta_2 \cdots \otimes \theta_n$ and $\bigotimes \emptyset = [\,]$. The way Erlang (and λAcTor) matches data against patterns is formalised below with the function match.

$$\mathrm{match}_{\rho,\sigma}(p_i, (x, \rho')) = \mathrm{match}_{\rho,\sigma}(p_i, \sigma(\rho'(x)))$$
$$\mathrm{match}_{\rho,\sigma}(x, d) = [\,x \mapsto d\,] \text{ if } x \notin \mathrm{dom}(\rho)$$
$$\mathrm{match}_{\rho,\sigma}(x, d) = [\,x \mapsto d\,] \text{ if } \mathrm{match}_{\rho',\sigma}(p', d) \neq \bot$$
$$\text{where } (p', \rho') = \sigma(\rho(x))$$
$$\mathrm{match}_{\rho,\sigma}(p, (t, \rho')) = \bigotimes \{\mathrm{match}_{\rho,\sigma}(p_i, (t_i, \rho')) \mid 1 \leq i \leq n\}$$
$$\text{where } \quad p = \mathtt{c}(p_1, \ldots, p_n)$$
$$t = \mathtt{c}(t_1, \ldots, t_n)$$
$$\mathrm{match}_{\rho,\sigma}(p, d) = \bot \qquad \text{otherwise}$$

A sound abstract version of match is defined as follows

$$\widehat{\mathrm{match}}_{\hat{\rho},\hat{\sigma}}(p_i,(x,\hat{\rho}')) = \bigcup_{\hat{d}\in\hat{\sigma}(\hat{\rho}'(x))} \widehat{\mathrm{match}}_{\hat{\rho},\hat{\sigma}}(p_i,\hat{d}\,)$$

$$\widehat{\mathrm{match}}_{\hat{\rho},\hat{\sigma}}(x,d) = \{[\,x \mapsto \hat{d}\,]\}$$

$$\widehat{\mathrm{match}}_{\hat{\rho},\hat{\sigma}}(p,(t,\hat{\rho}')) = \widehat{\bigotimes}\,\{\widehat{\mathrm{match}}_{\hat{\rho},\hat{\sigma}}(p_i,(t_i,\hat{\rho}')) \mid 1\le i\le n\}$$

if $p = \mathtt{c}(p_1,\ldots,p_n)$ and
$t = \mathtt{c}(t_1,\ldots,t_n)$

$$\widehat{\mathrm{match}}_{\hat{\rho},\hat{\sigma}}(p,d) = \emptyset \qquad \text{otherwise}$$

where

$$\widehat{\bigotimes}(\{\Theta_i \mid 1\le i\le n\}) = \left\{\,\theta \;\middle|\; \theta = \bigotimes\{\theta_i \mid 1\le i\le n\}, \theta\neq\bot, \theta_i\in\Theta_i, 1\le i\le n\,\right\}$$

A.2 Proof of Theorem 5.1

We start with some simple technical lemmata.

Lemma A.1. *Suppose the concrete domain $C = A \rightharpoonup B$ of partial functions has abstract domain $\hat{C} = \hat{A} \to \mathscr{P}(\hat{B})$ with the induced order $\le$ and abstraction function $\alpha_C : C \to \hat{C}$, then for all $f \in C$ and for all $\alpha_C(f) \le \hat{f}$*

$$\forall a \in \mathrm{dom}(f)\,.\,\alpha_B(f(a)) \in \hat{f}(\alpha_A(a)). \tag{A.1}$$

Further suppose $f, f' \in C$ such that $f' = f[a_1 \mapsto b_1,\ldots,a_n\mapsto b_n]$ and let $\hat{f},\hat{f}'\in\hat{C}$ such that $\hat{f}' = \hat{f} \sqcup [\hat{a}_1\mapsto\hat{b}_1,\ldots,\hat{a}_n\mapsto\hat{b}_n]$ with $\alpha_C(f)\le\hat{f}$ and $\alpha_A(a_i)=\hat{a}_i$, $\alpha_B(b_i)=\hat{b}_i$ for $i=1,\ldots,n$ then

$$\alpha_C(f') \le \hat{f}'. \tag{A.2}$$

Proof. Let $f\in C$ and $\hat{f}\in\hat{C}$ such that $\alpha_{\mathrm{C}}(f)\le\hat{f}$. The definition of $\le$ implies that for all $\hat{a}\in\hat{A}$

$$\alpha_{\mathrm{C}}(f)(\hat{a}) \subseteq \hat{f}(\hat{a}).$$

Take $a\in A$ and fix $\hat{a}=\alpha_{\mathrm{A}}(a)$ then we obtain

$$\alpha_{\mathrm{C}}(f)(\alpha_{\mathrm{A}}(a)) \subseteq \hat{f}(\alpha_{\mathrm{A}}(a)).$$

Expanding the definition of α_{C} yields

$$\{\alpha_{\mathrm{B}}(b_0) \mid (a_0,b_0)\in f, \alpha_{\mathrm{A}}(a_0)=\alpha_{\mathrm{A}}(a))\} \subseteq \hat{f}(\alpha_{\mathrm{A}}(a)).$$

In particular $\alpha_{\mathrm{B}}(f(a)) \in \{\alpha_{\mathrm{B}}(b_0) \mid (a_0, b_0) \in f, \alpha_{\mathrm{A}}(a_0) = \alpha_{\mathrm{A}}(a))\}$ which yields what we set out to prove

$$\forall a \in \mathrm{dom}(f) \,.\, \alpha_{\mathrm{B}}(f(a)) \in \hat{f}(\alpha_{\mathrm{A}}(a)).$$

Turning to equation A.2 we want to show $\alpha_{\mathrm{C}}(f') \leq \hat{f'}$. Let $\hat{a} \in \hat{A}$ then there are several cases to consider

1. $\alpha_{\mathrm{C}}(f')(\hat{a}) = \alpha_{\mathrm{C}}(f)(\hat{a})$. Then since $\alpha_{\mathrm{C}}(f) \leq \hat{f} \leq \hat{f'}$ we have

$$\alpha_{\mathrm{C}}(f)(\hat{a}) \subseteq \hat{f}(\hat{a}) \subseteq \hat{f'}(\hat{a}).$$

2. $\hat{a} = \alpha_{\mathrm{A}}(a_i)$ for some $1 \leq i \leq n$. Then $\hat{a} = \hat{a}_i$ and thus

$$\begin{aligned}\alpha_{\mathrm{C}}(f')(\hat{a}) &= \{\alpha_{\mathrm{B}}(b_i)\} \cup \{\alpha_{\mathrm{B}}(f(a)) | \alpha_{\mathrm{A}}(a) = \hat{a}, a \neq a_i\} \\ &\subseteq \{\hat{b}_i\} \cup \alpha_{\mathrm{C}}(f)(\hat{a}) \\ &\subseteq \{\hat{b}_i\} \cup \hat{f}(\hat{a}) \subseteq \hat{f'}(\hat{a})\end{aligned}$$

3. otherwise there does *not* exist $(a, b) \in f'$ such that $\alpha_{\mathrm{A}}(a) = \hat{a}$, hence $\alpha_{\mathrm{C}}(f')(a) = \emptyset$ which makes our claim trivially true.

We can thus conclude that $\alpha_{\mathrm{C}}(f') \leq \hat{f'}$. □

Corollary A.2. *Let $\pi \in Procs$ and $\hat{\pi} \in \widehat{Procs}$ such that $\alpha_{proc}(\pi) \leq \hat{\pi}$, let $\sigma \in Store$ and $\hat{\sigma} \in \widehat{Store}$ such that $\alpha_{st}(\sigma) \leq \hat{\sigma}$ and Let $\mu \in Mailboxes$ and $\hat{\mu} \in \widehat{Mailboxes}$ such that $\alpha_{ms}(\mu) \leq \hat{\mu}$ then*

1. $\forall \iota \in Pid \,.\, \alpha_{ps}(\pi(\iota)) \in \hat{\pi}(\alpha_{pid}(\iota))$
2. $\forall b \in VAddr \,.\, \alpha_{val}(\sigma(b)) \in \hat{\sigma}(\alpha_{va}(b))$
3. $\forall a \in KAddr \,.\, \alpha_{kont}(\sigma(a)) \in \hat{\sigma}(\alpha_{ka}(a))$
4. $\forall \iota \in Pid \,.\, \alpha_{m}(\mu(\iota)) \leq \hat{\mu}(\alpha_{pid}(\iota))$
5. $\forall \iota \in Pid \,.\, \forall x \in Var \,.\, \forall \delta \in Data \,.\, \forall q \in ProcState \,.$

$$\alpha_{va}(\mathrm{new}_{va}(\iota, x, \delta, q)) = \widehat{\mathrm{new}_{va}}(\alpha_{pid}(\iota), x, \alpha_{d}(\delta), \alpha_{ps}(q))$$

Proof. Cases 1 - 3 follow directly from Lemma A.1; it remains to show the claims of 4 and 5.

(iv) By assumption $\alpha_{\text{ms}}(\mu) \leq \hat{\mu}$ which implies that

$$\alpha_{\text{ms}}(\mu)(\alpha_{\text{pid}}(\iota)) \leq \hat{\mu}(\alpha_{\text{pid}}(\iota)) = \hat{\mu}(\hat{\iota}).$$

Expanding α_{ms} then gives us that $\alpha_{\text{m}}(\mu(\iota)) \leq \alpha_{\text{ms}}(\mu)(\alpha_{\text{pid}}(\iota))$, since $\alpha_{\text{ms}}(\mu) = \lambda\hat{\iota}.\bigsqcup\{\alpha_{\text{m}}(\mu(\iota)) \mid \alpha_{\text{pid}}(\iota) = \hat{\iota}\}$, which allows us to conclude

$$\alpha_{\text{m}}(\mu(\iota)) \leq \hat{\mu}(\hat{\iota}).$$

(v) The claim follows straightforwardly from expanding new_{va} and $\widehat{\text{new}_{\text{va}}}$:

$$\begin{aligned}\alpha_{\text{va}}(\text{new}_{\text{va}}(\iota, x, \delta, q)) = (&\alpha_{\text{pid}}(\iota), x, \alpha_{\text{d}}(\delta), \alpha_{\text{t}}(t)) \\ &\widehat{\text{new}_{\text{va}}}(\alpha_{\text{pid}}(\iota), x, \alpha_{\text{d}}(\delta), \alpha_{\text{ps}}(q))\end{aligned}$$

where $q = \langle e, \rho, a, t\rangle$.

□

We now prove Theorem 5.1.

Proof of Theorem 5.1. The proof is by a case analysis of the rule that defines the concrete transition $s \to s'$. For each rule, the transition in the concrete system can be replicated in the abstract transition system using the abstract version of the rule, with the appropriate choice of abstract pid, the continuation from the abstract store, the message from the abstract mailbox, etc.

Let $s = \langle \pi, \mu, \sigma\rangle \to \langle \pi', \mu', \sigma'\rangle = s'$ and $u = \langle \hat{\pi}, \hat{\mu}, \hat{\sigma}\rangle$ such that $\alpha_{\text{proc}}(\pi) \leq \hat{\pi}$, $\alpha_{\text{mail}}(\mu) \leq \hat{\mu}$, and $\alpha_{\text{st}}(\sigma) \leq \hat{\sigma}$. We consider a number of rules for illustration.

Case: (**Send**). We know that $s \to s'$ using rule **Send**; we can thus assume

$$\begin{aligned}\pi(\iota) &= \langle v, \rho, a, t\rangle \\ \sigma(a) &= \mathsf{Arg}_2\langle \ell, d, \iota', \_, c\rangle \\ d &= (\texttt{send}, \_)\end{aligned}$$

and for s'

$$\begin{aligned}\pi' &= \pi[\iota \mapsto \langle v, \rho, c, t\rangle] \\ \mu' &= \mu[\iota' \mapsto \text{enq}((v, \rho), \mu(\iota'))] \\ \sigma' &= \sigma.\end{aligned}$$

As a first step we will examine u and show that $u \rightsquigarrow u'$ for some u'. For $\hat{\pi}$ and $\hat{\sigma}$, writing $\hat{\iota} := \alpha_{\text{pid}}(\iota)$, Corollary A.2 gives us

$$\begin{aligned}&\langle v, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \in \hat{\pi}(\hat{\iota}) \\ &\mathsf{Arg}_2\langle \ell, \hat{d}, \hat{\iota}', \_, \hat{c}\rangle \in \hat{\sigma}(\hat{a})\end{aligned}$$

where $\alpha_{\text{env}}(\rho) = \hat{\rho}$, $\alpha_{\text{ak}}(a) = \hat{a}$, $\hat{d} = \alpha_{\text{val}}(d)$, $\hat{t} = \alpha_{\text{t}}(t)$, $\hat{\imath}' = \alpha_{\text{pid}}(\iota')$ and $\hat{c} = \alpha_{\text{ka}}(c)$. Rule **AbsSend** is now applicable and we can set

$$\begin{aligned}
\hat{\pi}' &:= \hat{\pi} \sqcup [\hat{\imath} \mapsto \hat{q}] \\
\hat{q} &:= \langle v, \hat{\rho}, \hat{c}, \hat{t} \rangle \\
\hat{\mu}' &:= \hat{\mu}[\hat{\imath}' \mapsto \widehat{\text{enq}}((v, \hat{\rho}), \hat{\mu}(\hat{\imath}'))] \\
u' &:= \langle \hat{\pi}', \hat{\mu}', \hat{\sigma} \rangle.
\end{aligned}$$

It follows from rule (**AbsSend**) that $u \rightsquigarrow u'$. It remains to show that $\alpha_{\text{cfa}}(s') \leq u'$ which follows directly from 1. $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$ and 2. $\alpha_{\text{ms}}(\mu') \leq \hat{\mu}'$.

1. $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$ follows immediately from Lemma A.1 since $\hat{\imath} = \alpha_{\text{pid}}(\iota)$ and $\alpha_{\text{ps}}(q) = \hat{q}$.

2. $\alpha_{\text{ms}}(\mu') \leq \hat{\mu}'$. It is sufficient to show that $\alpha_{\text{pid}}(\iota') = \hat{\imath}'$, which is immediate, and $\alpha_{\text{m}}(\mu'(\iota')) \leq \hat{\mu}(\hat{\imath})$. For the latter, since $\alpha_{\text{env}}(\rho) = \hat{\rho}$, a sound basic domain abstraction gives us

$$\begin{aligned}
\alpha_{\text{m}}(\mu'(\iota')) &= \alpha_{\text{m}}(\text{enq}((v, \rho), \mu(\iota'))) \\
&\leq \widehat{\text{enq}}((v, \hat{\rho}), \hat{\mu}(\hat{\imath}')) = \hat{\mu}(\hat{\imath})
\end{aligned}$$

   provided we can show $\alpha_{\text{m}}(\mu(\iota')) \leq \hat{\mu}(\hat{\imath}')$; the latter inequality follows Corollary A.2. Hence we can conclude $\alpha_{\text{ms}}(\mu') \leq \hat{\mu}'$ which completes the proof of this case.

Case: (**Receive**). In the concrete $s \to s'$ using the **Receive**, hence we can make the following assumptions

$$\begin{aligned}
\pi(\iota) &= \langle \texttt{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \texttt{end}, \rho, a, t \rangle =: q \\
(i, \theta, \mathfrak{m}) &= \text{mmatch}(p_1 \ldots p_n, \mu(\iota), \rho, \sigma) \\
\theta &= [x_1 \mapsto d_1 \ldots x_k \mapsto d_k] \\
b_j &= \text{new}_{\text{va}}(\iota, x_j, \delta_j, q) \\
\delta_j &= \text{res}(\sigma, d_j)
\end{aligned}$$

and for state s'

$$\begin{aligned}
\pi' &= \pi[\iota \mapsto q'] \\
q' &= \langle e_i, \rho', a, t \rangle \\
\rho' &= \rho[x_1 \mapsto b_1 \ldots x_k \mapsto b_k] \\
\mu' &= \mu[\iota \mapsto \mathfrak{m}] \\
\sigma' &= \sigma[b_1 \mapsto d_1 \ldots b_k \mapsto d_k].
\end{aligned}$$

As a first step we will look at u to prove there there exists a u' such that $u \rightsquigarrow u'$ using rule **AbsReceive**. We can invoke Corollary A.2, since $\alpha_{\text{proc}}(\pi) \leq \hat{\pi}$, to obtain

$$\hat{q} := \langle \texttt{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \texttt{end}, \hat{\rho}, \hat{a}, \hat{t}\ \rangle \in \hat{\pi}(\hat{\iota})$$

where we write $\hat{\rho} := \alpha_{\text{env}}(\rho)$, $\hat{a} := \alpha_{\text{ka}}(a)$, $\hat{t} = \alpha_{\text{t}}(t)$ and $\hat{\iota} := \alpha_{\text{pid}}(\iota)$. Moreover Corollary A.2 gives us

$$\alpha_{\text{m}}(\mu(\iota)) \leq \hat{\mu}(\hat{\iota}).$$

as $\alpha_{\text{ms}}(\mu) \leq \hat{\mu}$ and $\hat{\iota} = \alpha_{\text{pid}}(\iota)$. Since the instantiation of the basic domains is sound and $\alpha_{\text{st}}(\sigma) \leq \hat{\sigma}$ we then know that

$$(i, \hat{\theta}, \hat{\mathfrak{m}}) \in \widehat{\text{mmatch}}(\vec{p}, \hat{\mu}(\hat{\iota}), \hat{\rho}, \hat{\sigma})$$

such that $\hat{\theta} = \alpha_{\text{sub}}(\theta)$ and $\hat{\mathfrak{m}} \geq \alpha_{\text{m}}(\mathfrak{m})$. Turning to the substitution $\hat{\theta}$ we can see that

$$\hat{\theta} = [x_1 \mapsto \hat{d}_1 \ldots x_k \mapsto \hat{d}_k]$$

where $\hat{d}_i = \alpha(d_i)$ for $1 \leq i \leq k$. Appealing to the sound basic domain instantiation once more, noting that $\alpha_{\text{st}}(\sigma) \leq \hat{\sigma}$, yields that for $j = 1, \ldots, k$ we have $\hat{\delta}_j := \alpha_{\text{d}}(\delta_j) \in \text{res}(\hat{\sigma}, \hat{d}_j)$; to obtain new abstract variable addresses we can now set $\hat{b}_j := \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_j, \hat{\delta}_j, \hat{q}\ )$. Rule **AbsReceive** is applicable now; we make the following definitions

$$\begin{aligned}
\hat{\pi}' &:= \hat{\pi} \sqcup [\hat{\iota} \mapsto \hat{q}'] \\
\hat{q}' &:= \langle e_i, \hat{\rho}', \hat{a}, \hat{t}\ \rangle \\
\hat{\rho}' &:= \hat{\rho}[x_1 \mapsto \hat{b}_1 \ldots x_k \mapsto \hat{b}_k] \\
\hat{\mu}' &:= \hat{\mu}[\hat{\iota} \mapsto \hat{\mathfrak{m}}] \\
\hat{\sigma}' &:= \hat{\sigma} \sqcup [\hat{b}_1 \mapsto \hat{d}_1 \ldots \hat{b}_k \mapsto \hat{d}_k] \\
\hat{u}' &:= \langle \hat{\pi}', \hat{\mu}', \hat{\sigma}', \hat{\vartheta} \rangle
\end{aligned}$$

and observe that $u \rightsquigarrow u'$. It remains to show $\alpha_{\text{cfa}}(s') \leq u'$ which follows directly if we can prove 1. $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$, 2. $\alpha_{\text{ms}}(\mu') \leq \hat{\mu}'$ and 3. $\alpha_{\text{st}}(\sigma') \leq \hat{\sigma}'$.

1. $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$. We note that by Corollary A.2 we know $\hat{b}_i = \alpha_{\text{va}}(b_i)$ as $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, $\hat{\delta}_i = \alpha_{\text{d}}(\delta_i)$ and $\alpha_{\text{proc}}(q) = \hat{q}$ for $1 \leq i \leq n$. It follows that $\hat{\rho}' = \alpha_{\text{env}}(\rho')$ and hence $\hat{q}' = \alpha_{\text{ps}}(q')$. Lemma A.1 is now applicable, since $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, to give $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$.

2. $\alpha_{\text{ms}}(\mu') \leq \hat{\mu}'$. It is sufficient to show that $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, which is immediate, and $\alpha_{\text{m}}(\mathfrak{m}) \leq \hat{\mathfrak{m}}$ which we have already established above; hence we can conclude $\alpha_{\text{ms}}(\mu') \leq \hat{\mu}'$.

3. $\alpha_{\text{st}}(\sigma') \leq \hat{\sigma}'$. The observation that $\hat{b}_i = \alpha_{\text{va}}(b_i)$ and $\hat{d}_i = \alpha_{\text{val}}(d_i)$ allows the application of Lemma A.1 which gives $\alpha_{\text{st}}(\sigma') \leq \hat{\sigma}'$ as desired.

This completes the proof of this case.

Case: (**Apply**). Since $s \to s'$ using rule **Apply** we can assume that

$$\begin{aligned}
\pi(\iota) &= \langle v, \rho, a, t\rangle =: q \\
\sigma(a) &= \mathsf{Arg}_n\langle \ell, d_0 \ldots d_{n-1}, \rho', c\rangle := \kappa \\
\text{arity}(\ell) &= n \\
d_0 &= (\texttt{fun}(x_1 \ldots x_n) \to e, \rho_0) \\
d_n &= (v, \rho)
\end{aligned}$$

and for $i = 1, \ldots, n$

$$\begin{aligned}
\delta_i &= \text{res}(\sigma, d_i) \\
b_i &= \text{new}_{\text{va}}(\iota, x_i, \delta_i, q)
\end{aligned}$$

additionally for the successor state s'

$$\begin{aligned}
\pi' &= \pi[\iota \mapsto q'] \\
&\quad \text{where } q' := \langle e, \rho'[x_1 \to b_1 \ldots x_n \to b_n], c, \text{tick}(\ell, t)\rangle \\
\sigma' &= \sigma[b_1 \mapsto d_1 \ldots b_n \mapsto d_n] \\
\mu' &= \mu
\end{aligned}$$

As a first step we will examine u and show that there exists a u' such that $u \rightsquigarrow u'$ using rule **AbsApply**. From Corollary A.2, since $\alpha_{\text{proc}}(\pi) \leq \hat{\pi}$, it follows that

$$\hat{q} := \langle v, \alpha_{\text{env}}(\rho), \alpha_{\text{ka}}(a), \alpha_{\text{t}}(t)\rangle \in \hat{\pi}(\alpha_{\text{pid}}(\iota)).$$

Letting $\hat{\rho} := \alpha_{\text{env}}(\rho)$, $\hat{a} := \alpha_{\text{ka}}(a)$, $\hat{t} := \alpha_{\text{t}}(t)$ and $\hat{\iota} := \alpha_{\text{pid}}(\iota)$ we can appeal to Corollary A.2 again, as $\alpha_{\text{st}}(\sigma) \leq \hat{\sigma}$, to obtain

$$\mathsf{Arg}_n\langle \ell, \hat{d}_0 \ldots \hat{d}_{n-1}, \hat{\rho}', \hat{c}\rangle \in \hat{\sigma}(\hat{a})$$

where we write $\hat{d}_i := \alpha_{\text{val}}(d_i)$ for $0 \leq i < n$, $\hat{\rho}' := \alpha_{\text{env}}(\rho')$ and $\hat{c} := \alpha_{\text{ka}}(c)$. Expanding α_{val} yields

$$\hat{d}_0 = (\texttt{fun}(x_1 \ldots x_n) \to e, \hat{\rho}_0)$$

where we write $\hat{\rho}_0 := \alpha_{\text{env}}(\rho_0)$. Taking $\hat{d}_n := (v, \hat{\rho})$ we obtain from our sound basic domain abstraction

$$\hat{\delta}_i := \alpha_{\text{d}}(\delta_i) \in \widehat{\text{res}}(\hat{\sigma}, \hat{d}_i) \text{ for } i = 1, \ldots, n$$

as $\alpha_{\text{st}}(\sigma) \leq \hat{\sigma}$ and $\hat{d}_i = \alpha_{\text{val}}(d_i)$. Turning to the abstract variable addresses we define

$$\hat{b}_i := \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_i, \hat{\delta}_i, \hat{q}\,) \text{ for } 1 \leq i \leq n.$$

Rule **AbsApply** is now applicable and we define

$$\begin{aligned}
\hat{\pi}' &:= \hat{\pi} \sqcup [\hat{\iota} \mapsto \hat{q}'] \\
\hat{q}' &:= \langle e, \hat{\rho}'[x_1 \to \hat{b}_1 \ldots x_n \to \hat{b}_n], \hat{c}, \widehat{\text{tick}}(\ell, \hat{t}\,)\rangle \\
\hat{\sigma}' &:= \hat{\sigma} \sqcup [\,\hat{b}_1 \mapsto \hat{d}_1 \ldots \hat{b}_n \mapsto \hat{d}_n] \\
u' &:= \langle \hat{\pi}', \hat{\mu}, \hat{\sigma}'\rangle.
\end{aligned}$$

It is clear from rule **AbsApply** that $u \rightsquigarrow u'$; it remains to show that $\alpha_{\text{cfa}}(s') \leq u'$ to prove this case. The latter follows if we can justify 1. $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$ and 2. $\alpha_{\text{st}}(\sigma') \leq \hat{\sigma}'$.

1. $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$. We can appeal to Lemma A.1 provided we can show that $\hat{\iota} = \alpha_{\text{pid}}(\iota)$ and $\alpha_{\text{ps}}(q') = \hat{q}'$ where the former is immediate. For the latter, first observe that since we have a sound basic domain abstraction we know $\alpha_{\text{t}}(\text{tick}(\ell, t)) \leq \widehat{\text{tick}}(\ell, \hat{t})$; however as Time is a flat domain so the above inequality is in fact an equality

   $$\alpha_{\text{t}}(\text{tick}(\ell, t)) = \widehat{\text{tick}}(\ell, \hat{t}\,).$$

   Moreover $\hat{b}_i = \alpha_{\text{va}}(b_i)$ for $1 \leq i \leq n$ by Corollary A.2, hence

   $$\alpha_{\text{env}}(\rho'[x_1 \to b_1 \ldots x_n \to b_n]) = \hat{\rho}'[x_1 \to \hat{b}_1 \ldots x_n \to \hat{b}_n]$$

   as $\alpha_{\text{env}}(\rho') = \hat{\rho}'$; in combination with $\hat{c} = \alpha_{\text{ka}}(c)$ we obtain the desired $\alpha_{\text{ps}}(q') = \hat{q}'$. Thus we conclude that $\alpha_{\text{proc}}(\pi') \leq \hat{\pi}'$.

2. $\alpha_{\text{st}}(\sigma') \leq \hat{\sigma}'$. Since $\alpha_{\text{va}}(b_i) = \hat{b}_i$ and $\alpha_{\text{val}}(d_i) = \hat{d}_i$ for $1 \leq i \leq n$ Lemma A.1 is applicable once more and gives us $\alpha_{\text{st}}(\sigma') \leq \hat{\sigma}'$ and completes the proof of this case.

□

Appendix B

Proof of Soundness of the Generated ACS

Terminology Analogously to our remark in Section 5.3 on active components for rules **AbsR** in the abstract operational semantics it is possible to identify a similar pattern in the concrete operational semantics. Henceforth we will speak of the concrete active component (ι, q, q') of a rule **R** of the concrete operational semantics and we will say the abstract active component $(\hat{\iota}, \hat{q}, \hat{q}')$ of a rule **AbsR** of the abstract operational semantics where **AbsR** is the abstract counterpart of **R**. We will omit the adjectives abstract and concrete when there is no confusion.

Lemma B.1. *Suppose $s \to s'$ using the concrete rule **R** with concrete active component (ι, q, q') and $\hat{s} \geq \alpha_{cfa}(s)$. Then $\hat{s} \rightsquigarrow \hat{s}'$ with $\hat{s}' \geq \alpha_{cfa}(s')$ using rule **AbsR** with abstract active component $(\alpha_{pid}(\iota), \alpha_{ps}(q), \alpha_{ps}(q'))$.*

Proof. The claim follows from inspection of the proof of Theorem 5.1. □

Proof of Theorem 5.3. Suppose $s \to s'$ using rule **R** of the concrete operational semantics with active component (ι, q, q'). We will prove our claim by case analysis on **R**.

- **R** = **FunEval**, **ArgEval**, **Apply** or **Vars**. Take $\hat{s} = \alpha_{\text{cfa}}(s)$; Lemma B.1 gives us that $\hat{s} \rightsquigarrow \hat{s}'$ using abstract rule **AbsR** = **AbsFunEval**, **AbsArgEval**, **AbsApply** or **Vars** respectively with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, $\hat{q} = \alpha_{\text{ps}}(q)$ and $\hat{q}' = \alpha_{\text{ps}}(q')$. It follows that

$$\mathbf{r} := \hat{\iota} \colon \hat{q} \xrightarrow{\tau} \hat{q}' \in R.$$

  Since $s \to s'$ with active component (ι, q, q') it follows that

  1. $\alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) \geq 1$,
  2. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1$ and

3. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1$

as $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, $\hat{q} = \alpha_{\text{ps}}(q)$ and $\hat{q}' = \alpha_{\text{ps}}(q')$. We know $\alpha_{\text{acs}}(s) \leq \mathbf{v}$ and thus $\mathbf{v}(\hat{\iota}, \hat{q}) \geq 1$. If we define

$$\mathbf{v}' := \mathbf{v}[(\hat{\iota}, \hat{q}) \mapsto \mathbf{v}(\hat{\iota}, \hat{q}) - 1, (\hat{\iota}, \hat{q}') \mapsto \mathbf{v}(\hat{\iota}, \hat{q}') + 1],$$

then it is clear that $\mathbf{v} \to_{\text{acs}} \mathbf{v}'$ using rule $\mathbf{r} \in R$ and the inequalities

$$\begin{aligned}\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}) - 1 = \mathbf{v}'(\hat{\iota}, \hat{q}) \\ \alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}') + 1 = \mathbf{v}'(\hat{\iota}, \hat{q}');\end{aligned}$$

the consequence of the latter two is that $\alpha_{\text{acs}}(s') \leq \mathbf{v}'$, since $\alpha_{\text{acs}}(s) \leq \mathbf{v}$, which completes the proof of this case.

- **R** = **Receive**. Letting $\hat{s} = \alpha_{\text{cfa}}(s)$ Lemma B.1 yields that $\hat{s} \rightsquigarrow \hat{s}'$ using abstract rule **AbsReceive** with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, $\hat{q} = \alpha_{\text{ps}}(q)$ and $\hat{q}' = \alpha_{\text{ps}}(q')$. We note that $s = \langle \pi, \sigma, \mu \rangle$ and $\hat{s} = \langle \hat{\pi}, \hat{\sigma}, \hat{\mu} \rangle$ where $\hat{\pi} = \alpha_{\text{proc}}(\pi)$, $\hat{\sigma} = \alpha_{\text{st}}(\sigma)$ and $\hat{\mu} = \alpha_{\text{ms}}(\mu)$. Let the message matched by mmatch and extracted from $\mu(\iota)$ be $d = (p_i, \rho')$ then inspecting rule **AbsReceive** we can assume that during $\hat{s} \rightsquigarrow \hat{s}'$ message $\hat{d} = (p_i, \hat{\rho}')$, where $\hat{\rho}' = \alpha_{\text{env}}(\rho')$, is matched by $\widehat{\text{mmatch}}$. Since the message abstraction is a sound data abstraction we know that

$$\hat{m} := \alpha_{\text{msg}}(\text{res}(\sigma, d)) \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d}\,)$$

and hence we have

$$\mathbf{r} := \hat{\iota} \colon \hat{q} \xrightarrow{?\hat{m}} \hat{q}' \in R$$

Additionally we know

1. $\alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) \geq 1$,
2. $\alpha_{\text{acs}}(s)(\hat{\iota}, \hat{m}) \geq 1$,
3. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1$,
4. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1$ and
5. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{m}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{m}) - 1$

since d is the message extracted from $\mu(\iota)$ and $\hat{m} = \alpha_{\text{msg}}(\text{res}(\sigma, d))$. By assumption we know $\alpha_{\text{acs}}(s) \leq \mathbf{v}$ which implies $\mathbf{v}(\hat{\iota}, \hat{q}) \geq 1$ and $\mathbf{v}(\hat{\iota}, \hat{m}) \geq 1$, and so we can define

$$\mathbf{v}' := \mathbf{v}\begin{bmatrix}(\hat{\iota}, \hat{q}) \mapsto \mathbf{v}(\hat{\iota}, \hat{q}\,) - 1, \\ (\hat{\iota}, \hat{m}) \mapsto \mathbf{v}(\hat{\iota}, \hat{m}) - 1, \\ (\hat{\iota}, \hat{q}') \mapsto \mathbf{v}(\hat{\iota}, \hat{q}') + 1\end{bmatrix};$$

it is then clear that, using rule $\mathbf{r} \in R$, $\mathbf{v} \to_{\text{acs}} \mathbf{v}'$ and

$$\begin{aligned}
\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}) - 1 = \mathbf{v}'(\hat{\iota}, \hat{q}) \\
\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{m}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{m}) - 1 \leq \mathbf{v}(\hat{\iota}, \hat{m}) - 1 = \mathbf{v}'(\hat{\iota}, \hat{m}) \\
\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}') + 1 = \mathbf{v}'(\hat{\iota}, \hat{q}').
\end{aligned}$$

Hence, since $\alpha_{\text{acs}}(s) \leq \mathbf{v}$, we can conclude $\alpha_{\text{acs}}(s') \leq \mathbf{v}'$ as desired.

- **R** = **Send**. Using Lemma B.1, with $\hat{s} = \alpha_{\text{cfa}}(s)$, gives $\hat{s} \rightsquigarrow \hat{s}'$ with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ for the abstract rule **AbsSend** where $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, $\hat{q} = \alpha_{\text{ps}}(q)$ and $\hat{q}' = \alpha_{\text{ps}}(q')$. Examining the concrete and abstract states we see $s = \langle \pi, \sigma, \mu \rangle$ and $\hat{s} = \langle \hat{\pi}, \hat{\sigma}, \hat{\mu} \rangle$ where $\hat{\pi} = \alpha_{\text{proc}}(\pi)$, $\hat{\sigma} = \alpha_{\text{st}}(\sigma)$ and $\hat{\mu} = \alpha_{\text{ms}}(\mu)$. Let the pid of the recipient be ι' and let d be the value enqueued to ι''s mailbox $\mu(\iota')$; inspecting the proof of Theorem 5.1 the pid of the abstract recipient is $\hat{\iota}' := \alpha_{\text{pid}}(\iota')$ and the sent abstract value is $\hat{d} = \alpha_{\text{val}}(d)$. Appealing to the soundness of the message abstraction we obtain

$$\hat{m} := \alpha_{\text{msg}}(\text{res}(\sigma, d)) \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d}\,)$$

and hence we have

$$\mathbf{r} := \hat{\iota} \colon \hat{q} \xrightarrow{{}'\hat{\iota}'!\hat{m}} \hat{q}' \in R$$

Additionally we know

1. $\alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) \geq 1$,
2. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1$,
3. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1$ and
4. $\alpha_{\text{acs}}(s')(\hat{\iota}', \hat{m}) = \alpha_{\text{acs}}(s)(\hat{\iota}', \hat{m}) + 1$

since d is the message enqueued to $\mu(\iota')$ and $\hat{m} = \alpha_{\text{msg}}(\text{res}(\sigma, d))$. From our assumption we know $\alpha_{\text{acs}}(s) \leq \mathbf{v}$ and thus $\mathbf{v}(\hat{\iota}, \hat{q}) \geq 1$; making the definition

$$\mathbf{v}' := \mathbf{v} \left[\begin{array}{r} (\hat{\iota}, \hat{q}) \mapsto \mathbf{v}(\hat{\iota}, \hat{q}\,) - 1, \\ (\hat{\iota}, \hat{q}') \mapsto \mathbf{v}(\hat{\iota}, \hat{q}') + 1, \\ (\hat{\iota}', \hat{m}) \mapsto \mathbf{v}(\hat{\iota}', \hat{m}) + 1 \end{array} \right];$$

we observe that we are able to use rule $\mathbf{r} \in R$ to make the step $\mathbf{v} \to_{\text{acs}} \mathbf{v}'$. Further the inequalities

$$\begin{aligned}
\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}) - 1 = \mathbf{v}'(\hat{\iota}, \hat{q}) \\
\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}') + 1 = \mathbf{v}'(\hat{\iota}, \hat{q}') \\
\alpha_{\text{acs}}(s')(\hat{\iota}', \hat{m}) = \alpha_{\text{acs}}(s)(\hat{\iota}', \hat{m}) - 1 \leq \mathbf{v}(\hat{\iota}', \hat{m}) - 1 = \mathbf{v}'(\hat{\iota}', \hat{m}).
\end{aligned}$$

imply, since $\alpha_{\text{acs}}(s) \leq \mathbf{v}$, that $\alpha_{\text{acs}}(s') \leq \mathbf{v}'$ which concludes the proof of this case.

- **R** = **Spawn**. Take $\hat{s} = \alpha_{\text{cfa}}(s)$; Lemma B.1 gives us that $\hat{s} \rightsquigarrow \hat{s}'$ using abstract rule **AbsSpawn** with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota} = \alpha_{\text{pid}}(\iota)$, $\hat{q} = \alpha_{\text{ps}}(q)$ and $\hat{q}' = \alpha_{\text{ps}}(q')$. We note that $s = \langle \pi, \sigma, \mu \rangle$, $s' = \langle \pi', \sigma', \mu' \rangle$ and $\hat{s} = \langle \hat{\pi}, \hat{\sigma}, \hat{\mu} \rangle$ where $\hat{\pi} = \alpha_{\text{proc}}(\pi)$, $\hat{\sigma} = \alpha_{\text{st}}(\sigma)$ and $\hat{\mu} = \alpha_{\text{ms}}(\mu)$. Further we can assume

$$\begin{aligned}
\pi(\iota) &= \langle \texttt{fun()} \to e, \rho, a, t \rangle \\
\sigma(a) &= \mathsf{Arg}_1 \langle \ell, d, \rho', c \rangle \\
d &= (\texttt{spawn}, _) \\
\iota' &:= \text{new}_{\text{pid}}(\iota, \ell, \vartheta) \\
\pi'(\iota) &= \langle \iota', \rho', c, t \rangle = q' \\
\pi'(\iota') &= \langle e, \rho, *, t_0 \rangle =: q''
\end{aligned}$$

Noting that we are replicating the step $s \to s'$ in the abstract $\hat{s} \rightsquigarrow \hat{s}'$, $\alpha_{\text{cfa}}(s) = \hat{s}$ and $\alpha_{\text{pid}} \circ \text{new}_{\text{pid}} = \widehat{\text{new}_{\text{pid}}} \circ \alpha$ we can see that the new abstract pid created is $\hat{\iota}' = \alpha_{\text{pid}}(\iota')$ together with its process state $\hat{q}'' = \alpha_{\text{ps}}(q'')$. Hence we can conclude that

$$\mathbf{r} := \hat{\iota} \colon \hat{q} \xrightarrow{\nu' \hat{\iota}' . \hat{q}''} \hat{q}' \in R$$

and we observe that

1. $\alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) \geq 1$,
2. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1$,
3. $\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1$ and
4. $\alpha_{\text{acs}}(s')(\hat{\iota}', \hat{q}'') = \alpha_{\text{acs}}(s)(\hat{\iota}', \hat{q}'') + 1$.

Now the assumption $\alpha_{\text{acs}}(s) \leq \mathbf{v}$ allows us to conclude $\mathbf{v}(\hat{\iota}, \hat{q}) \geq 1$, so that we can define

$$\mathbf{v}' := \mathbf{v} \begin{bmatrix} (\hat{\iota}, \hat{q}) \mapsto \mathbf{v}(\hat{\iota}, \hat{q}\,) - 1, \\ (\hat{\iota}, \hat{q}') \mapsto \mathbf{v}(\hat{\iota}, \hat{q}') + 1, \\ (\hat{\iota}', \hat{q}'') \mapsto \mathbf{v}(\hat{\iota}', \hat{q}'') + 1 \end{bmatrix};$$

and use rule $\mathbf{r} \in R$ to make the step $\mathbf{v} \to_{\text{acs}} \mathbf{v}'$. Further with the inequalities

$$\begin{aligned}
\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}) = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}) - 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}) - 1 = \mathbf{v}'(\hat{\iota}, \hat{q}) \\
\alpha_{\text{acs}}(s')(\hat{\iota}, \hat{q}') = \alpha_{\text{acs}}(s)(\hat{\iota}, \hat{q}') + 1 \leq \mathbf{v}(\hat{\iota}, \hat{q}') + 1 = \mathbf{v}'(\hat{\iota}, \hat{q}') \\
\alpha_{\text{acs}}(s')(\hat{\iota}', \hat{q}'') = \alpha_{\text{acs}}(s)(\hat{\iota}', \hat{q}'') - 1 \leq \mathbf{v}(\hat{\iota}', \hat{q}'') - 1 = \mathbf{v}'(\hat{\iota}', \hat{q}'').
\end{aligned}$$

and our assumption $\alpha_{\text{acs}}(s) \leq \mathbf{v}$ we see that $\alpha_{\text{acs}}(s') \leq \mathbf{v}'$ which completes the proof of this case and the theorem.

□

Index

Symbols

A

B

C

D

E

F

T

U

V

W