Verification of Erlang Programs using Abstract Interpretation and Model Checking

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Frank Günter Huch

aus Neuss

Berichter:
Universitätsprofessor Dr. Klaus Indermark
Universitätsprofessor Dr. Michael Hanus

Tag der mündlichen Prüfung: 2. November 2001

To Birgit and Annika

Abstract

The functional programming language Erlang is successfully used for the development of distributed systems. We present an approach for the formal verification of Erlang programs using abstract interpretation and model checking. Therefore, we define a framework for abstract interpretations of Erlang programs. In this framework it is guaranteed that the abstract operational semantics is safe with respect to the standard semantics of Erlang. In this context, safe means that it contains all paths of the standard semantics. Applying this framework to a finite domain abstraction, many system properties (e.g. the absence of deadlocks and lifelocks or mutual exclusion) can automatically be proven by model checking. Since the framework guarantees safeness of the abstraction the properties proven for the abstraction also hold for the real execution of the Erlang program.

In this theses, we develop the framework for abstract interpretations of Erlang programs for formal verification by model checking. We show how the framework can be used in model checking for Linear Time Logic. For Erlang programs containing non-tail recursive function calls data abstraction covered by our framework is not sufficient. Therefore, we extend the framework by abstraction of the control-flow. All presented techniques are prototypically implemented in a verification tool. We demonstrate the practical usability of our approach and the verification tool by the verification of a database server.

Zusammenfassung

Die Funktionale Programmiersprache Erlang wird erfolgreich zur Entwicklung Verteilter Systeme eingesetzt. Wir präsentieren einen Ansatz zur formalen Verifikation von Erlang Programmen mittels abstrakter Interpretation und Model Checking. Im allgemeinen ist Model Checking für Temporallogiken und Erlangprogramme unentscheidbar. Deshalb definieren wir ein Rahmenwerk für abstrakte Interpretationen von Erlangprogrammen. Dieses Rahmenwerk garantiert, daß die abstrakte operationelle Semantik sicher bezüglich der Standardsemantik von Erlang ist. In diesem Zusammenhang bedeutet sicher, daß alle Pfade der Standardsemantik auch in der abstrakten operationellen Semantik enthalten sind. Wird dieses Rahmenwerk auf eine abstrakte Interpretation mit endlichem Wertebereich angewendet, so können viele Systemeigenschaften (wie z.B. Verklemmungsfreiheit, Lebendigkeit oder wechselseitiger Ausschluß) mittels Model Checking automatisch bewiesen werden. Da das Rahmenwerk eine sichere Abstraktion garantiert, gelten bewiesene Eigenschaften auch für die tats"achliche Ausführung des Erlangprogramms.

In dieser Arbeit entwickeln wir das Rahmenwerk für Abstrakte Interpretationen von Erlang Programmen zur formalen Verifikation mittels Model Checking. Wir zeigen, wie das Rahmenwerk zum Model Checking für die Linearzeitlogik (Linear Time Logic) verwendet werden kann. Bei Erlangprogrammen mit nicht-endrekursiven Funktionsaufrufen, reicht die Datenabstraktion des Rahmenwerks nicht aus. Deshalb erweitern wir das Rahmenwerk um eine Abstraktion des Kontrollflusses. Alle präsentierten Techiken sind prototypisch in einem Verifikationstool implementiert. Wir zeigen die praktische Verwendbarkeit unseres Ansatzes und des Verifikationstools anhand der Verifikation eines Datenbankservers.

Contents

Chapter 1

Introduction

Growing requirements of industry and society impose greater complexity of software development. Consequently, understandability, maintenance, and reliability of software systems becomes harder. Nowadays an increasing amount of applications are distributed which is not only a result of the boom of the Internet but also of the requirements of our information society. Especially for the development of these applications, we want to contribute to the quality of software development.

In distributed systems several programs are executed in a network of computers. Synchronization and the exchange of information between programs is provided by communication over the network. Therefore, communication protocols are usually defined by describing the chronological order of the messages exchanged. An example for a distributed system is (mobile) telephony: Telephones and relay stations form a distributed network. Components of this network must communicate with each other to provide services like phone calls. Other examples are cash dispensers in combination with a bank server, chat rooms, or web browsers.

In contrast to parallel programming (e.g., by the Message Passing Interface (*MPI*) [Pac97]), the objective of distributed programming is not gaining speed up of a computation. The application itself is distributed and needs a distributed implementation.

The software crisis, which started in the seventies and still is an actual problem in software development, shows the problems of ensuring maintenance and reliability of large software applications. This already holds for large sequential systems. In distributed applications additional problems occur: A sequential system is executed deterministically for a given input. In contrast, a distributed system can behave non-deterministically even for identical inputs. In the concrete Erlang implementation this non-determinism is solved by a scheduler. Aside from the users inputs, it depends on the kind of scheduler (e.g., round-robin scheduling [Tan92]), the number of processes and the network connectivities used. Adding processes to the system or changing the used hardware can modify this scheduling and new errors can occur. Avoiding this

can only be guaranteed, if all possible executions (schedules) behave correct.

The theory of formal methods proposes the following approach to this task:

- Define the protocol for the communication of the programs in a specification formalism. Possible formalisms are the Calculus on Communicating Systems *(CCS)* [Mil80], LOTOS [Hog89] (an extension of CCS), or the Specification and Description Language *(SDL)* [Bro91].
- Specify the properties a system should have in a mathematical logic, usually in temporal or modal logic. We will discuss such logics later.
- Verify the properties by model checking. This is an algorithm which checks whether a formula (describing a property) is satisfied in a model (representing the semantics of the specified protocol).
- Refine the protocol without modifying the communication structure until this refinement yields the executable program.

However, this approach cannot be used in practice because of the following problems:

1. The specification formalisms do not fulfill the software engineering requirements for the development of the sequential parts of the application. Here, expressive formalisms like the Unified Modeling Language *(UML)* [BJR99] are used. Most of these formalisms do not even have a formalized semantics and it is not clear how to check properties for these specifications.
2. For the specification of many protocols it is necessary to specify data dependencies. Simple specification formalisms like CCS do not support this. As a solution, more expressive formalisms like LOTOS and SDL are developed. However, the semantics of these formalisms usually yields an infinite model in this case and in general, model checking is undecidable for these infinite models.
3. This strict distinction between specification of the protocol and its refinement is not possible in practice. Programmers implement the specifications by hand. There is no guarantee that the communication behavior of the implementation respects the specified protocol. Furthermore, during the implementation deficiencies in the protocol specification can be detected. Hence, the specifications might be changed so that all properties must be checked again for the modified specification.

To solve these problems, several approaches are possible. The first problem can be solved by the development of new specification formalisms, including a formal semantics. These formalisms should provide both sequential and protocol specifications. As a solution of the second problem, a combination of abstraction and model checking for protocols by means of data dependencies is proposed [CGL94b, KP98]. The idea is the construction of a finite model which abstracts the infinite model of the semantics. If a formula holds in the finite model, then it also holds in the infinite model. However, for the third problem no satisfactory solution exists yet. A particular reason for this is

that many people are involved in the development of a system. Software development is a very complex process. Automatic code generation from the specification could help but these approaches lack in expressiveness. The existing tools are more or less graphical programming environments without support for formal verification.

We propose another approach for the formal verification of distributed systems. By exploiting the idea of abstraction, we extend model checking to programs written in real programming languages. This is the only way to guarantee that a system really satisfies a property.

Related to distributed programming is the notion of concurrency. Already on one computer it can be useful to execute several concurrent processes. For instance, without concurrency a reactive program (e.g., a graphical user interface) must iterate over all possible inputs (e.g., buttons) to react on inputs (e.g., clicks), which is known as "busy waiting". Concurrency allows the creation of multiple processes which are executed "in parallel". For every possible source of input a process can be created which suspends on this possible input. Each concrete input event awakes the corresponding, suspended process. The execution of the non-suspended processes is performed by a scheduler.

The use of concurrency is also necessary in distributed systems. Distributed programs must react on messages from different other programs. These messages are different possible inputs as in the concurrent case, discussed above. Furthermore, it is possible that the program interacts with a user "in parallel". Again, concurrency is useful.

Many programming languages have been extended for the development of distributed systems. For imperative languages, the technique of *remote procedure calls (RPC)* [Wei90] has been developed. The same technique is used in distributed object-oriented systems where it is called *remote method invocation (RMI)* [Gro01]. By means of both techniques it is possible to evaluate procedures on a remote computer. During its evaluation a remote procedure call has access to the resources of the remote computer. Its result is sent back through the network to the program which initiated the procedure call. The program suspends until the result is evaluated. To avoid a blocking of the whole program, multiple concurrent processes should be executed. Exactly this is a big disadvantage of RPC and RMI. Synchronization and communication between concurrent processes must be implemented by shared variables. This is a low-level mechanism and no comfortable communication abstractions are provided. Hence, concurrent programming still has a flavor of assembler programming in these languages. Another problem is the fact that a programmer has to learn two different mechanisms for communication. RPC/RMI for distributed programming and shared variables for concurrent programming. Furthermore, the developed systems do not provide scalability. If something is implemented concurrently (by communication over shared variables), then this code cannot easily be distributed to a network, e.g., to eliminate a bottleneck in the communication protocol. The communication using shared variables must be replaced by RPC/RMI.

A different approach was chosen in the functional programming language Erlang [AVWW96]. Erlang was developed by Ericsson and the Elemental Telecommuni-

cation Systems Laboratories. It has been successfully used for the development of many telecommunication applications like Ericsson's ATM switch AXD 301. Unlike imperative languages using RPC or RMI, Erlang provides processes which only communicate by message passing. They are executed concurrently on one computer or distributed in a network. This guarantees scalability of the developed systems because parts of the system can easily be distributed without any modification of the communication structure between the processes. Furthermore, programmers must only learn one mechanism for communication between concurrent and distributed processes. Another advantage of Erlang is the use of pattern matching for functions operating on algebraic data structures as well as for dispatching messages. Furthermore, Erlang provides powerful mechanisms for robust programming, which is a gist in the implementation of distributed systems.

On the other hand, there are also some disadvantages of Erlang (perhaps resulting from the fact that Erlang was developed in industry). Results of research on functional programming, like the λ-calculus [Cur63] have not been considered in its development. This leads to the absence of scoping. As an alternative, Erlang provides bind-once variables which are more difficult to formalize. Higher-order functions have not been integrated in the language from the beginning. Their syntax is very complicated and there is, in contrast to the rest of the language, a kind of scoping for anonymous function definitions. Therefore, their use in Erlang programs is uncommon. Finally, the largest disadvantage of Erlang is the absence of a type system. As a result, for example typos in the source code often do not yield an error but result in a deadlock.

There also exist extensions of other functional languages for concurrent and distributed programming. Concurrent ML [PR97] is an extension of Standard ML [MTH89] for concurrent programming. This language is further extended for distributed programming with FACILE [TLK96, TLK97]. Conceptually, FACILE is much clearer than Erlang but it does not support fault tolerant programming. It is not widespread and we think that there is more interest in the verification of Erlang than of FACILE.

We have also worked on the extension of the lazy functional programming language Haskell [J+98] for robust distributed programming. The result is Distributed Haskell [HN00], an extension of Concurrent Haskell [JGF96]. In contrast to Erlang, our approach is based on Haskell's powerful type system. Distributed Haskell contains even more powerful mechanisms for robust programming than Erlang. We think that it could be a valuable alternative to Erlang. However, at the moment its implementation is still on research level and no larger systems have been developed in Distributed Haskell.

As a quintessence, we have chosen Erlang for formal verification using abstraction and model checking. It is a stable programming language used in practice. Large applications are implemented in Erlang and our results can be valuable for a large community of Erlang developers.

Applying the techniques of abstract interpretation [CC77a, JN94, SS98] to Erlang shows the need of some extensions of these techniques. The contribution of this thesis is the development of a framework for abstract interpretations of Erlang pro-

grams. This framework guarantees that the model defined by the abstract operational semantics *(AOS)* includes all paths of the standard operational semantics *(SOS)*. Under some assumptions to the program, the AOS yields a finite transition system for finite domain abstract interpretations.

In linear time logic *(LTL)* [LP85] a formula is satisfied in a model if all paths of the model satisfy the formula. Therefore, we can conclude: if the AOS satisfies the formula, then also the SOS satisfies it. Successful proofs in the abstraction are safe for the concrete system. Model checking for LTL and finite models is decidable. The existing algorithms are efficient enough for practical verification.

Since the abstraction has sometimes more paths than the SOS it is not possible to verify existential properties, like there exists a behavior of the process which leads to a deadlock. These properties can be specified in branching time logics, e.g., CTL [Eme90]. However, in contrast to specifications, real systems implemented in programming languages behave deterministically (with the exception of interleaving). Therefore, existential properties will usually not be used in the formal verification of implemented distributed systems (in contrast to non-deterministic specifications, like using the choice operator in CCS). Furthermore, LTL is an expressive logic. Safeness, liveness and fairness properties can be specified and verified.

The presented work is implemented as a prototype, written in Haskell. For LTL model checking we implemented the automata-based algorithm of Vardi [Var96]. Using this prototype we are able to verify some small examples.

Structure of the Thesis

In Chapter 2 we introduce the main concepts of the programming language Erlang. This includes sequential, concurrent, distributed, and robust programming. In Chapter 3 we restrict Erlang to a core fragment which we will use for formal verification. We formalize its syntax and semantics. Chapter 4 contains the central contribution of this thesis. We motivate our approach for the abstraction of Erlang. This leads to the definition of a framework for abstract interpretations. The framework is used in two example abstractions and we present finiteness results for the abstract semantics. Chapter 5 puts the verification of Erlang programs by LTL model checking and abstraction in concrete terms. Therefore, we first extend Erlang programs to state propositions. Finally, we present the verification of different system properties by the example abstraction of Chapter 4 and our approach. We discuss some extensions and optimizations of the approach in Chapter 6. For the abstract semantics of `case` and `receive` we define a simplified framework and discuss the possibilities for state space reduction. The context-free structure of non-tail recursive function calls allows the verification by model checking for only a subclass of Erlang programs. In Chapter 7 we present an abstraction of the control flow of Erlang programs. Using this abstraction it is possible to verify a larger class of Erlang programs in our approach. Chapter 8 presents how the results can be used for the verification of real Erlang applications. Then we discuss other work related to our approach in Chapter 9. We conclude in Chapter 10 with the prospects of our work.

Chapter 2

The Programming Language Erlang

This chapter provides a gentle introduction to Erlang. We informally introduce the main concepts with examples. Readers familiar with Erlang can skip this chapter and proceed with Chapter 3.

Erlang [AVWW96] is a functional programming language developed by Ericsson. Its evaluation strategy is call-by-value. Erlang has no static type system. Only simple predefined types like `integer` are checked at runtime. It provides additional features for concurrent, distributed, and robust programming.

The development of Erlang started on the basis of Prolog [SS94]. The developers eliminated backtracking and replaced predicates by functions. The result is a strict functional programming language with "bind-once" variables instead of scoping.

2.1 Sequential Programming

Variables in Erlang start with a capital letter, like in Prolog. A program is a set of function definitions. As an example, the square function can be defined as follows:

```
square(X) -> X * X.
```

Branches can be defined using a `case` expression:

```
fac(X) -> case X of
            0 -> 1;
            N -> N * fac(N-1)
          end.
```

Alternatively, we can define a function by several rules and pattern matching. Two rules are separated with a semicolon and the complete definition is terminated with a dot:

```
fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

The scope of a variable is restricted to the right-hand side of a rule.

In Erlang, it is possible to define several functions with the same name. They are distinguished with respect to their arity. For example, it is possible to define a tail-recursive version of the factorial function as follows:

```
fac(0,R) -> R;
fac(N,R) -> fac(N-1,N*R).

fac(N) -> fac(N,1).
```

Here we use an accumulator as a second parameter. We write `fac/1` respectively `fac/2` to distinguish the functions. The same notation is used in Erlang's module system as we will see in Section 2.1.2.

Repeated computations of subexpressions can be avoided with new variables in the right-hand side of a rule. These variables can be bound to the result of subexpressions. Then they can be used in the following expressions. As an example the exponentiation with four can be defined as:

```
exp4(X) -> Y = X * X,
           Y * Y.
```

First the square of `X` is computed. The variable `Y` is bound to this value and the result is the square of `Y`. This use of variables resembles imperative programming but Erlang only provides so called *bind-once variables*. If a variable is bound to a value, then it cannot be overwritten in other words bound to another value. For instance, the sequence `Y=3,Y=4` yields a runtime error. This relates to `let` expressions in functional languages. Bindings of variables are valid for the whole right-hand side.

2.1.1 Data structures

Erlang provides atoms, lists, and tuples for structuring data. Atoms start with a lower case letter and can be seen as constructors with arity zero. For example, the boolean values are the atoms `true` and `false`. It is also possible to use arbitrary strings (including special characters like blanks) as atoms. In this case the atom has to be put into single quotes (e.g., `'EXIT'` or `'Model Checking Erlang !'`).

Lists

Lists are written in the same way as in Prolog. The empty list is denoted by `[]`. An element e can be added to a list l with `[`e`|`l`]`. As in Prolog it is also possible to use the comma notation `[`e_1`,...,`e_n`|`l`]` as an abbreviation for `[`e_1`|[`e_2`|...[`e_n`|`l`]...]]` and `[`e_1`,...,`e_n`]` for `[`e_1`|[`e_2`|...[`e_n`|[]]...]]`. Erlang is untyped and there is no restriction that e_i must be an element and l must be a list. For example, it is also allowed to write `[[]|4]`. However, as a convention l should always be a list. All predefined functions require this.

Lists (and all other data structures) can also be used in patterns. Patterns may contain variables. The matching of a pattern against a value succeeds if there exists a binding for the variables of the pattern such that the application of the binding to the pattern is identical to the value. Then the binding from a successful match is applied the succeeding expression. We will formalize this in the next chapter.

As an example for programming with lists, a function for the concatenation of two lists can be defined as follows:

```
append([X|Xs],Ys) -> [X|append(Xs,Ys)];
append([],Ys)     -> Ys
```

Tuples

The second kind of data structures are tuples of any arity which are enclosed in curly brackets. Firstly, tuples can be used in functions which yield more than one value. A division function which yields the integer division and the rest can be programmed as

```
division(X,Y) -> {X div Y, X rem Y}.
```

where `div` and `rem` are predefined functions for the integer division and its remainder.

Secondly, they can be used to combine a fixed number of values. For example, a dictionary can be implemented as a list of tuples representing key-value pairs. A function for searching the value of a given key can be programmed as:

```
lookup(Key,[{Key,Value}|Rest]) -> {value,Value};
lookup(Key,[Pair|Rest])        -> lookup(Key,Rest);
lookup(Key,[])                 -> fail.
```

If the key is not found in the dictionary, then `lookup` yields the atom `fail`. Otherwise, it yields the value stored previously. The function yields a tuple consisting of the flag `value` and the value. Otherwise, `fail` could not be stored as a value in the dictionary. The successful lookup of this value would yield `fail` which cannot be distinguished from a failed lookup.

This definition also shows some Erlang specific features. The definition of `lookup` has a non-linear pattern in the first rule. We use the variable `Key` twice in the pattern. This corresponds to an explicit test of equality between two keys. Furthermore, overlapping patterns as in the first two rules are commonly used. The patterns are matched one after the other and the first one matching is chosen.

Thirdly, tuples can be used to implement algebraic data types: we use nested tuples such that

- the application of a constructor to n arguments is expressed by an $n + 1$ tuple
- the constructor is represented by an atom

The convention in Erlang is that the atom which represents the constructor is placed in the first component of the tuple. Its arguments are placed in the remaining tuple components. As an example, a tree can be represented in Erlang as

- `{leaf,`*value*`}` for the tree only containing one value and
- `{node,`t_l`,`t_r`}` for a node with two subtrees t_l and t_r.

Then the size of a tree can be computed by

```
size({leaf,_}) -> 1;
size({node,Tl,Tr}) -> size(Tl)+size(Tr).
```

The wildecard (`_`) can be used as an anonymous variable.

Tuples make lists superfluous as the following tuple based implementation of the `append` function shows. The two list constructors are represented by the atoms `nil` and `cons`.

```
append(nil,Ys) -> Ys;
append({cons,X,Xs},Ys) -> {cons,X,append(Xs,Ys)).
```

For convenient programming, Erlang also provides the special list syntax. This syntax also allows abbreviations as discussed before. Another advantage of the explicit list representation is its more efficient implementation. Erlang also provides built-in functions for the conversion from lists to tuples and vice versa.

2.1.2 Modules

Erlang has a simple module system which allows dividing larger programs into a set of modules. Each module has its own name space for the defined functions. Functions can be exported by a module and imported from other modules. The name of a module and it's filename must match. The file begins with the module declaration:

`-module(`*moduleName*`).`

The exported functions are declared with

`-export([`f_1`/`n_1`,...,`f_m`/`n_m`]).`

The list contains m functions. Every function f_i with arity n_i must be defined in the module file.

There are two methods for calling functions of another module. Firstly, they can be integrated in the local name space of functions:

`-import(`*moduleName*`,[`f_1`/`n_1`,...,`f_m`/`n_k`]).`

The listed functions from the module *moduleName* are imported and can be used inside the module.

Secondly, a function of another module can be called explicitly. With

moduleName:$f(arg_1,\ldots,arg_n)$

the function f with arity n from module *moduleName* is applied to the arguments.

2.2 Concurrent Programming

In addition to the functional language features described in the previous section, Erlang provides features for concurrent programming. The Erlang runtime system allows the concurrent execution of several processes. They can communicate via message passing for which Erlang provides explicit expressions for sending and receiving values.

New processes can be created by the expression

`spawn(`m`,`f`,[`$v_1, \ldots, v_n$`])`

The evaluation of a newly created process starts with the term $m : f(v_1, \ldots, v_n)$, where f is an exported function of the module m and the arguments $v_1, \ldots, v_n$ are values. If `spawn` is called with non-evaluated arguments, then these arguments are evaluated eagerly before the new process is spawned. In reactive systems, the newly created process will often loop. In case of termination, the result of its evaluation is discarded. The functional result of the execution of `spawn` is the *process identifier (pid)* of the created process. Process identifiers are unique and can be used as any other value. For example, it is possible to store pids in data structures or to send them to other processes. A process can also access its own pid with the built-in function `self/0`.

Processes can communicate via asynchronous message passing. Every process has a mailbox, in which all incoming messages are stored[1]. A process sends a message to another process using the construct p`!`v, where p is the pid of the other process and v is the value to be sent.

For receiving values Erlang provides convenient access to mailboxes using pattern matching. With the expression

`receive`
p_1`->`e_1`;`
$\vdots$
p_n`->`e_n
`end`

a process can select messages in its mailbox. The elements of the mailbox of a process are successively matched against the patterns $p_1, \ldots, p_n$: The first element of the mailbox matching a pattern p_i with a substitution ρ is removed from the mailbox and the receive expression evaluates to $\rho(e_i)$. If no mailbox entry matches a pattern, then the process suspends until a new value is sent to the process.

This method for accessing the elements of the mailbox makes concurrent programming very simple. The messages of interest can be picked out from the mailbox. All other messages are stored in the mailbox and can be handled later. Therefore, the mailbox is not just a queue. For example, consider a process executing the following receive expression:

[1] Since no mails but messages are sent it would be better to use *message-box* instead of *mailbox*. However, *mailbox* is established in the Erlang community.

```
receive
  stop  -> e1;
  error -> e2
end
```

If its mailbox contains the messages

```
fail : [4,5] : {value,7}
```

[2]

the receiving process suspends, until one of the messages `stop` or `error` is sent to the process.
If its mailbox contains the messages

```
fail : error : {value,7} : stop
```

then the `error` message is extracted from the mailbox and the evaluation of the process continues with e_2.

If we extend the receive expression in the example by a wildcard rule `X ->` e_3 as a third pattern, then the first message of the mailbox is extracted. If this element is `stop` or `error`, then e_1 respectively e_2 is performed. Otherwise, `X` is bound to the first message of the mailbox and the evaluation continues with e_3.

To support the development of robust applications it is necessary to use timeouts. Therefore, Erlang provides an `after` expression for `receive`:

`receive` p_1`->`e_1`;` $\ldots$ `;` p_n`->`e_n `after` t`->`e `end`

The integer t specifies the time in milliseconds in which the receive expression tries to match incoming messages against the patterns. If none of the patterns matches a message during this time, then it proceeds with e. The receive expression does not suspend. If t is zero, then the patterns are matched against the values in the mailbox only once. If t is the atom `infinity`, then the `receive-after` expression behaves like `receive`.

As an example for a concurrent Erlang program we consider a database process. The entries of this database are unique keys with values. A client process can allocate a key and add the corresponding value, if the key is not yet allocated. Naturally, a client can also lookup the value of a given key. In this thesis we will often return to this example and also verify some properties of this program.

The internal state of the database is a list of pairs containing keys and values. The database keeps this state as argument during recursive calls. For the access of a value corresponding to a key we use the function `lookup` from Section 2.1.1.

```
-module(database).
-export([start/0]).

start() -> database([]).
```

[2] We separate the messages of a mailbox with colons and notate the messages in chronological order. This means `fail` is the oldest message the process has received.

```
database(L) ->
  receive
    {allocate,Key,P} -> case lookup(Key,L) of
                          fail -> P!free,
                                  receive
                                    {value,V,P} ->
                                       database([{Key,V}|L])
                                  end;
                          {succ,V} -> P!allocated,
                                      database(L)
                        end;
    {lookup,Key,P}   -> P!lookup(Key,L),
                        database(L)
  end.
```

Using this module, database processes can be started by calling or spawning the function `start`. The initial interface of a database process are messages consisting of a triple with the flags `allocate` or `lookup`, a key, and the pid of the requesting process. When an allocation message is received, the key is looked up in the database. If the key is already allocated, then the message `allocated` is returned to the requesting process and the database proceeds in its initial state. Otherwise, it answers that the requested key is free and waits for the corresponding value to be stored.

The main point of the implementation of the database is that it guarantees mutual exclusion when several clients allocate new keys. In the absence of mutual exclusion, entries could disappear or the values of wrong clients could be inserted to the database, if two clients try to allocate the same key. To guarantee mutual exclusion, we use two common programming techniques of Erlang:

1. After receiving an allocation request, we proceed in the inner receive expression. There we only consider messages with the flag `value`. We ignore other messages like `{allocate,`k`,`p`}`. These messages are stored in the mailbox of the database process and can be considered later.

2. We check if the pid sent together with the message is the same as the pid in the allocation message. Therefore, we use the dynamic character of pattern matching and the bind-once variables of Erlang. In the pattern matching against the pattern `{value,V,P}`, the variable `P` is not newly introduced. `P` is already bound to a value (a pid). The matching only succeeds if a triple is received consisting of the atom `value`, any value and the same pid as the one received before. Values with other pids in the third component are ignored.

However, the database process can only guarantee mutual exclusion, if all accessing clients respect this protocol and send respectively receive the messages in the expected order.

In a fault tolerant system we would extend this receive expression with a timeout to prevent deadlocks when clients crash after allocating a key. We would also add

synchronization messages to guarantee the execution of updates. To keep the example small, these extensions are not presented here.

A simple client defining a user interface to the database can be programmed as follows:

```
-module(client).

-export([loop/1]).

loop(DB) ->
  case io:read('(l)ookup/(i)nsert >') of
    {ok,i} -> DB!{allocate,read('Key >'),self()},
              receive
                 free      -> V = read('Value >'),
                              DB!{value,V,self()};
                 allocated -> write('Key allocated')
              end;
    {ok,l} -> DB!{lookup,read('Key >'),self()},
              receive
                 R -> write(R)
              end;
    _      -> nop
  end,
  loop(DB).
```

Just like the database, the client is programmed as a loop. As a parameter it holds the pid of the database process. The built-in function `read` of the module `io` prints its argument as a prompt and reads a string from the keyboard. This string is converted into an atom. If this succeeds, then a tuple of the atom `ok` and the atom read is the result. Otherwise, it yields a parse error message. In correspondence to the user input the client sends `allocate` respectively `lookup` messages to the database. With respect to the protocol of the database it regards corresponding answer messages. If the newly allocated key is free, then the user is asked for the corresponding value. Then this value is sent to the database.

Notice that several clients can be executed concurrently and communicate with the database. Certainly, in a concurrent system with only one terminal the clients write to and read from, a concurrent execution of these clients is senseless. However, this client is just an example for possible clients. Other clients can be defined similarly. They only have to respect the protocol structure.

2.3 Distributed Programming

For distributed programming Erlang uses the same process concept as for concurrent programming. In a distributed Erlang system several nodes are executed on several computers in the Internet. Especially, it is also possible to execute several nodes on

one computer. On an Erlang node, several processes can be executed concurrently. To distinguish nodes located on the same computer, every node gets a name when it is started. Communication between processes executed on different nodes is done using the same constructs as in concurrent programming. Therefore, the pid of a process internally also contains the node on which it is executed. The only differences to concurrent programming are the creation of processes on other nodes and the communication between two independently started Erlang processes.

2.3.1 Creating Remote Processes

Erlang provides an extended `spawn` expression:

`spawn(`n`@`$host$`,`m`,`f`,[`$v_1, \ldots, v_n$`])`

with a similar behavior as `spawn`, except that the new process is started on the node n on *host*. n is the name of the node and *host* the hostname of the computer on which the node is located.

For instance, it would be possible to start the database process on a remote node with:

```
spawn('server@io.informatik.rwth-aachen.de',database,start,[])
```

The system is distributed to a network without any further changes of the source code.

With this extension it is very easy to distribute processes, developed in a concurrent environment, for example to provide scalability of an application or to guarantee fault tolerance to the failure of one node.

2.3.2 Open Systems

Creating processes on different nodes is an expressive mechanism for distributed programming. However, many applications are inherently distributed and a (hierarchical) distribution with `spawn/4` is not possible. Examples are telephony, chats, name-servers, or cash dispensers. In these *open systems* processes are started independently and communication between the processes has to be initiated at runtime. In Erlang this can be realized with a global registration of processes on a node. With the expression

`register(`*name*`,`p`)`

the pid p is globally registered as the atom *name* on the local node. If another pid is already registered with the same name, then the execution of `register` yields a runtime error. An arbitrary process executed on the same node can delete a registration with:

`unregister(`*name*`)`

For the communication between two independently started processes on different nodes we can use the extended send operation:

{*name*,*node*}!*v*

The message v is sent to the process registered as *name* on *node*. For flexibility, this kind of communication is usually chosen for the first contact only. Then the pids are exchanged and all further messages are sent to pids, as in concurrent programming. With this technique scalable systems can easily be implemented. For example, a server can administrate several sub-servers. Only this server is globally registered and clients can contact it with its registered name. Then the server answers with a pid of a sub-server which may be executed on a different computer. For subsequent communication with the server, the pid of the sub-server is used. Hence, all further interaction between client and server is redirected to the sub-server. Scalability can easily be provided, by adding new computers, on which more sub-servers can be executed. Communication with pids yields more flexibility than communication with registered names.

As an example for an open system we modify our database example to a distributed version. When a database process is started, it registers itself globally on its node. In addition, it gets another interface for the connection of remote processes and answers with its own pid:

```
-module(database).
-export([start/0]).

start() -> register(database,self()),
           database([]).

database(L) ->
  receive
    {allocate,Key,P} ->
       ...
    {lookup,Key,P} ->
       ...
    {connect,P} -> P!{connect,self()},
                   database(L)
  end.
```

A corresponding client can connect to the database in its initial part and then proceed as before, without any changes of the code.

```
-module(client).
-export([start/1]).

start(Node) -> {database,Node}!{connect,self()},
               receive
                  {connect,DB} -> loop(DB)
               end.
```

2.4 Robust Programming

Modern programming languages must provide exception mechanisms allowing the programmer to catch exceptions and to throw own exceptions. For concurrent and distributed programming this does not suffice: Mechanism for the controlling or monitoring of processes, especially if they crash, are needed.

2.4.1 Exception Handling

In Erlang exceptions can be thrown for several reasons:

- A match operation fails.
- None of the patterns in a function definition or a `case` expression matches a given value.
- A built-in function is called with an incorrect argument (e.g., `[]+3`).

The expressions `catch` and `throw` provide a mechanism for controlling the evaluation of an expression. With `catch` e it is possible to protect the expression e from errors, as the following example shows:

```
demo(X) -> catch X+39.
```

If the function `demo` is called with a number, e.g., `demo(3)`, then the result is `42`. No exception is thrown and hence `catch` does not effect the result. On the other hand, in the call `demo(a)` the application of `+` yields an exception. This exception is caught and the result of the function is the value:

```
{'EXIT',{badarith,{erl_eval,eval_op,['+',a,39]}}}
```

The atom `'EXIT'` indicates that the exception is a built-in exception which is specified in more detail by the second component of the tuple.

The programmer can also generate his own exceptions with the function `throw`/1:

```
fourtyTwo(X) -> case X of
                  42 -> 42;
                  _  -> throw({myError, notAnswerToEverything})
                end.
```

User defined exceptions can also be caught with `catch`. To distinguish our exception from predefined exceptions, we use the flag `myError` in the first tuple component instead of `'EXIT'`. However, this is just a convention. Internally, other exceptions are also generated with `throw({'EXIT',...})`.

2.5 Process Linking

For concurrent and distributed programming, `catch` and `throw` do not provide programmers in the development of robust applications. However, especially in a distributed setting robustness is needed. In a network, some computers can crash or some components can loose their connection to the network. Nevertheless, a crash of the entire system should be avoided in this case. For example, it should be possible to use a mobile phone, although one of the nodes in the telephone network has crashed. The phone call can be redirected via other nodes.

In Section 2.2 we have already introduced programming with timeouts. The `receive` expression can be extended with a timeout (`after`). It is possible to wait for a message for only a restricted time. However, this is not sufficient for convenient programming. For more convenience, Erlang also provides a powerful linking mechanism. With

`link(`p`)`

it is possible to establish a (bidirectional) link between the process executing this function and the process with pid p. If one of the linked processes terminates or crashes, then the other one crashes, too. For example, processes of a graphical user interface need not be terminated explicitly if the control process terminates. They can be terminated automatically with linking.

In many other cases this behavior is impractical. With the built-in function `process_flag/2` it is possible to change this behavior. After performing

```
process_flag(trap_exit,true)
```

a process does not crash, if another linked process crashes. Instead a message of the form `{'EXIT',`p`,`*reason*`}` is sent to the process. p is the pid of the process which terminated or crashed. For example, *reason* can be `normal` if the other process terminated. If the linked process crashes with an exception, then *reason* is this exception.

This mechanism works independently of the location of the processes. If a remote computer crashes and hence a linked process crashes, then this is also observed by linking.

Chapter 3

Core Erlang – A Fragment of Erlang

In the last chapter we have presented the programming language Erlang in an informal way. However, for the verification of Erlang programs we need a formal syntax and semantics. To make the task of defining a formal semantics feasible, we restrict Erlang to a core fragment without the module system and the aspects of distribution and linking. We will later discuss how these aspects can be taken into consideration.

3.1 Syntax

Erlang is an untyped functional programming language. To distinguish the different arities of function symbols, we introduce signatures.

Defintion 3.1 (Signature)
A *signature* $\Sigma = (\Sigma^{(0)}, \Sigma^{(1)}, \ldots)$ is a family of sets of function symbols with their arity. We write $f/n \in \Sigma$ instead of $f \in \Sigma^{(n)}$. $\lhd$

In the rest of this thesis we use two fixed signatures:

- $\mathcal{F}$ is the signature of all predefined Core Erlang function symbols. For instance $+/2 \in \mathcal{F}$.
- $\mathcal{C}$ is the signature of constructors. It contains atoms as constructors of arity zero. The set *Atoms* contains all constants which may occur in a program including numbers. E.g. **1, 2, fail, succ** $\in$ *Atoms*.

Defintion 3.2 (Core Erlang Constructors)
The signature of *Core Erlang constructors* is defined as:

$$\mathcal{C} := \{[\cdot|\cdot]/2, [\,]/0\} \cup \{\{\ldots\}/n \mid n \in \mathbb{N}\} \ \cup \ \{a/0 \mid a \in \mathit{Atoms}\}$$

These are the constructors for building lists, constructors for building tuples of any arity, and the atoms as constructors with arity zero. ◁

Defintion 3.3 (Σ-Terms)
The set of Σ-*terms over a set* S is defined as the smallest set $T_\Sigma(S)$ with

$$S \subseteq T_\Sigma(S),$$
$$c/n \in \Sigma \text{ and } t_1, \ldots, t_n \in T_\Sigma(S) \Longrightarrow c(t_1, \ldots, t_n) \in T_\Sigma(S)$$

For $n = 0$ we will usually omit the brackets and simply write c instead of $c()$. ◁

In the following we consider constructor terms $T_\mathcal{C}(S)$. The constructor terms for lists and tuples will usually be written in mixfix notation. We use the comma notation $[e_1, \ldots, e_n]$ as an abbreviation for the constructor term $[e_1|[e_2|\ldots[e_n|[\,]\,]\ldots]]$ and $[e_1, \ldots, e_n|\,l\,]$ for $[e_1|[e_2|\ldots[e_n|\,l\,]\ldots]]$.

We will use these constructor terms as values in the operational semantics but we can also use them for the definition of patterns in a Core Erlang Program. We distinguish the constructors for building terms from those used as values in the semantics because a different treatment of them will be necessary in the abstraction. Therefore, we use an underlined variation of constructors

$$\underline{\mathcal{C}} := \{\underline{c}/n \mid c/n \in \mathcal{C}\}.$$

In the programs we will use the type-writer font for the constructors of $\underline{\mathcal{C}}$ and the bold roman font for the constructors of $\mathcal{C}$.

Patterns in Core Erlang can also contain variables $Var := \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \ldots\}$.

Defintion 3.4 (Program Patterns)
The set of *program patterns* is defined as

$$PPat := T_{\underline{\mathcal{C}}}(Var).$$ ◁

With the definition of program patterns we can now define Core Erlang programs and expressions.

Defintion 3.5 (Core Erlang Programs)
A *Core Erlang program* is a non-empty finite set of *definitions* of the form

$$f\,\mathtt{(}X_1\mathtt{,}\ldots\mathtt{,}X_n\mathtt{)}\ \mathtt{->}\ e\,\mathtt{.}$$

where for all $X_1, \ldots, X_n \in Var$ holds $X_i \neq X_j$ for $i \neq j$ and $e \in \mathcal{E}(\emptyset)$ a Core Erlang expression which will be defined in the next definition.

All defined functions of a program p are collected in the family

$$\mathcal{F}(p) = (\mathcal{F}^{(0)}(p), \ldots, \mathcal{F}^{(n)}(p))$$

where $0, \ldots, n$ are the arities of the defined functions. In every Core Erlang program a main function is defined: $\mathtt{main}/0 \in \mathcal{F}(p)$. ◁

The Core Erlang expressions are defined over an arbitrary set S which is empty in the syntax. In the semantics S will be the set of possible values of the evaluation.

Defintion 3.6 (Core Erlang Expressions)
The set of *Core Erlang Expressions* $\mathcal{E}(S)$ over a set S is the smallest set with:

- $S \subseteq \mathcal{E}(S)$ | values
- $Var \subseteq \mathcal{E}(S)$ | variables
- `self` $\in \mathcal{E}(S)$ | the own pid
- $\phi/n \in \Sigma \cup \underline{\mathcal{C}} \cup \mathcal{F}(p), f/n \in \mathcal{F}(p)$, $e_1, \ldots, e_n \in \mathcal{E}(S), p_1, \ldots, p_n \in PPat$
  $\Longrightarrow$
  - ϕ`(`e_1`,` $\ldots$ `,`e_n`)` $\in \mathcal{E}(S)$ | application
  - p_1 `=` $e_1 \in \mathcal{E}(S)$ | pattern matching
  - e_1`,`$e_2 \in \mathcal{E}(S)$ | sequence
  - e_1`!`$e_2 \in \mathcal{E}(S)$ | send
  - `case` e `of` p_1`->`e_1`;` $\ldots$ p_n`->`$e_n \in \mathcal{E}(S)$ | branch
  - `receive` p_1`->`e_1`;` $\ldots$ p_n`->`$e_n \in \mathcal{E}(S)$ | receive
  - `spawn(`f`,`e_1`)` $\in \mathcal{E}(S)$ | spawn ◁

Note that variables occurring in the right-hand side of a function definition are not restricted to the variables of the left hand side: Erlang has no scoping (see Section 2.1). New variables can be introduced everywhere. We will often refer to the variables occurring in an expression. This is possible by means of the function

$$vars : \mathcal{E}(S) \longrightarrow Var$$

yielding all functions of an expression.

The defined core fragment is still complex. We do not want to restrict it to a smaller language because we want to work as close as possible to a real programming language. The main concepts of Erlang are represented in Core Erlang.

Example 3.7
As a first example we define a Core Erlang program for the database developed in Section 2.2. The Core Erlang program p_{db} is defined as:

```
main() -> DB = spawn(dataBase,[[]]),
          spawn(client,[DB]),
          client(DB).

dataBase(L) -> receive
                  {allocate,Key,P} ->
                     case lookup(Key,L) of
                        fail -> P!free,
                                receive
                                   {value,V,P} ->
                                       dataBase([{Key,V}|L])
                                end;
```

```
                    {succ,V} -> P!allocated,
                                dataBase(L)
                end;
            {lookup,Key,P} -> P!lookup(Key,L),
                              dataBase(L)
        end.

lookup(K,L) -> case L of
            [{K,V}|R] -> {value,V};
            [Pair|R]  -> lookup(K,R);
            []        -> fail
        end.
```

A problem is the translation of the pattern matching in the function definitions but with the use of `case` this can be solved easily. We will later prove some properties of this database combined with two accessing clients. The client can be translated accordingly.

3.2 Semantics

The semantics of Erlang is informally described in [AVWW96]. For our aims we have to formalize it.

In the theory of semantics of programming languages mainly two kinds of semantics (denotational, operational) are considered [NN92, Win93]. The denotational semantics is defined as the least fixed point of a system of equations induced by the program. Some approaches define a denotational semantics for concurrent systems [BZ82, Ros84]. The work closest to our's is a denotational semantics for Concurrent ML [DB97]. However, the big disadvantage of all these approaches is their technical effort. Unlike sequential programs, concurrent systems can behave nondeterministically. Therefore, the denotational semantics is usually defined over sets of possible results. Another problem is nontermination. Usually, denotational semantics uses $\perp$ to express nontermination. Different nonterminating evaluations cannot be distinguished because they do not yield a result. Unfortunately, many concurrent and distributed systems are reactive systems and nontermination is required. The side-effects in the loops influence the system. In order to model this, the denotational semantics can be extended to sets of possible runs, in other words infinite words of actions which is an extravagant expense. Furthermore, it is just a simulation of the operational semantics.

Therefore, we chose an operational interleaving semantics for Core Erlang. An example for the advantage of this semantics can be seen in the operational semantics of Concurrent ML [PR97] which is much clearer than the corresponding denotational semantics. The operational semantics defines a non-deterministic relation representing all possible behaviors of a system. The non-determinism in the parallel composition of processes is simulated with interleaving: all possible execution sequences of the

parallel processes are considered. A comparison of the two approaches for concurrent systems can be found in [BW91].

The only disadvantage of an operational semantics is the lack of compositionality. The semantics of two expressions e_1 and e_2 cannot be combined to the semantics of their sequence e_1, e_2 or parallel composition $e_1 \| e_2$. An approach to solve this problem are natural semantics [Ast91]. Instead of a small step operational semantics the natural semantics defines a big step semantics, i.e., the result is produced in one step. Sub-evaluations are combined to one evaluation step which make the evaluation more compositional. As with the denotational technique it is very difficult to define the semantics of non-terminating programs.

Other possible models for concurrency are partial order based techniques. Their aim is to prevent the state space explosion which is caused by the interleaving approach. Instead of using interleaving these approaches describe true concurrency. The main approaches are event structures [BC89] and Mazurkiewicz traces [Maz84]. In both approaches independent actions may be executed in parallel. The behavior of a concurrent system is described with partial orders in which the possible schedules of independent actions are not distinguished. This leads too much smaller models in many cases.

In this thesis, the aim of the definition of a formal semantics is the formal verification with model checking. Applying model checking to these partial order approaches is more difficult than applying model checking to the transition systems resulting from the interleaving semantics [Che97, BL01]. The price for the smaller models are less efficient algorithms. Furthermore, many of the existing algorithms break down the partial order structure and prove the properties with interleaving configurations of the partial order models. Another point are the logics for partial order models which are used for the verification of system properties. The dependency respectively independency of actions influences the semantics of the formulas. Specifying properties in these logics is not very intuitive.

We decided to define an interleaving semantics for Core Erlang. Later on, the partial order techniques can be applied to this semantics as partial order reduction. This technique is a successful optimization technique for model checkers which uses the ideas of partial order semantics. In Section 6.2 we will present the application of this technique for an optimization of our verification tool.

In [DF98] a formal semantics for Erlang, inspired by the structured operational semantics of CCS and the π-calculus, is defined. A rewriting logic implementation of this semantics can be found in [Nol01]. This semantics is complicated and asynchronous communication is modelled as two synchronizations

- between the sending process and the receiving mailbox and
- the mailbox and the receiving process.

This semantics is used for the verification of Erlang programs with theorem proving. In this context an inductive structure of an action can be helpful because proofs can be performed by induction. Here we present a more direct approach in which the

asynchronous communication and the mailbox are modelled directly. We describe a sending action is one single step instead of dividing it into sub-steps.

During the execution of an Erlang program several processes run concurrently. Every process is identified by a unique process identifier (*pid*). Pids can be stored in data structures or sent to other processes. Therefore, we allow constructor terms over pids as possible values of the evaluation ($T_{\mathcal{C}}(Pid)$) where Pid is an infinite set of process identifiers, distinguishable from *Atoms*. Pids only occur in the semantics of Erlang. Therefore, their internal representation is not relevant. However, in some examples where we will discuss the semantics pids will explicitly occur. Therefore, we define a representation of pids as:

$$Pid := \{@n \mid n \in \mathbb{N}\}$$

The semantics of the predefined functions is defined by an interpretation:

Defintion 3.8 (Σ-Interpretation)
Let A be a set of values, called *domain*, and ι a family of *interpretation functions* $\iota = (\iota^{(n)} : \Sigma^{(n)} \to (A^n \dashrightarrow A) \mid n \in \mathbb{N}^+)$ for the functions of Σ. Then $\mathcal{A} = (A, \iota)$ is called a Σ-*Interpretation*.
We write $\iota(F/n)$ instead of $\iota^{(n)}(F/n)$ and $F_{\mathcal{A}}$, if n is clear from the context. $\lhd$

The interpretation fuinction ι applied to a function symbol yields a partial function $(A^n \dashrightarrow A)$ because not every function can be interpreted on the whole domain. An example for such a partial function is the square root ($\sqrt{}$). In programming languages, the domain of $\sqrt{}$ usually is `Float`, although it is defined only for positive numbers. In the untyped language Erlang this is even more important: We only have one domain for all values and a predefined function is usually defined only on a subset of this domain. For example, `+/2` is not defined on atoms but on numbers.

In the interpretation for Core Erlang we use the constructor terms over Pid as domain. Hence, we have $\mathcal{A} = (T_{\mathcal{C}}(Pid), \iota)$ with $\iota = (\iota^{(n)} : \mathcal{F}^{(n)} \to (T_{\mathcal{C}}(Pid)^n \dashrightarrow T_{\mathcal{C}}(Pid)))$ the interpretation functions for the predefined functions.

Example 3.9
The function `+/2` $\in \mathcal{F}$ is interpreted as the function for adding two values and it is defined only on the atoms representing numbers. For instance,

$$\iota^{(2)}(\texttt{+}/2)\ (\mathbf{17}, \mathbf{25}) = \mathbf{42} \quad \text{and} \quad \iota^{(2)}(\texttt{+}/2)\ (\texttt{-}\mathbf{7}, \mathbf{0}) = \texttt{-}\mathbf{7}$$

On the other hand, $\iota^{(2)}(\texttt{+}/2)\ (\mathbf{17}, \{\mathbf{2}, \mathbf{fail}\})$ is undefined.

Defintion 3.10 (Mailbox, Process, State, and Label)
A state of the evaluation of a Core Erlang program is a finite set of processes

$$State := \mathcal{P}_{fin}(Proc),$$

where a process consist of a pid, a Core Erlang expression over $T_{\mathcal{C}}(Pid)$ and a mailbox

$$Proc := Pid \times \mathcal{E}(T_{\mathcal{C}}(Pid)) \times Mb,$$

and a mailbox is a word over values of the evaluation

$$Mb := T_{\mathcal{C}}(Pid)^*.$$

States $\{\pi_1, \ldots, \pi_n\} \in State$ are usually not written in set notation. Instead, we write $\pi_1 \| \ldots \| \pi_n$. Furthermore, we write Π, π for the disjoint union $\Pi \uplus \{\pi\}$.

We will define a labelled transition system for operational semantics. The set of possible labels is defined as:

$$\begin{aligned} Label := &\{\texttt{!}v \mid v \in T_{\mathcal{C}}(Pid)\} \cup \{\texttt{?}v \mid v \in T_{\mathcal{C}}(Pid)\} \\ &\cup \{\texttt{spawn(}f\texttt{)} \mid f/n \in \mathcal{F}(p)\} \cup \{\varepsilon\} \end{aligned}$$ ◁

In Erlang patterns are not static. For instance, the pattern matching in the function definition

```
f(X) -> {X,Y}={3,4},
        Y.
```

is no matching against a tuple of variables. `X` is not a free variable in the right-hand side of the rule. From the call of `f` it is bound to a value. This has to be considered in matching. The matching only succeeds if `X` is bound to 3. Otherwise, we get a runtime error. Hence, some parts of a pattern are statically defined and others are added at runtime. Therefore, in the semantics it is not sufficient to consider program patterns *PPat*. We must consider patterns which can also contain constructor terms:

Defintion 3.11 (Patterns)
The set of *patterns* is defined as

$$Pat := T_{\underline{\mathcal{C}}}(Var \cup T_{\mathcal{C}}(Pid)).$$ ◁

Beside the two different kinds of constructors, these patterns can also contain pids which are not allowed in program patterns.

For the definition of the leftmost innermost operational semantics of Erlang, we use the technique of an evaluation contexts [FFKD87] which specifies where the next step of an evaluation may take place.

Defintion 3.12 (Evaluation Contexts)
The set of *evaluation contexts* $\mathcal{EC}(S)$ for Core Erlang is defined as the smallest set such that:

- $[\,] \in \mathcal{EC}(S)$
- For all $\phi \in \Sigma \cup \underline{C} \cup \mathcal{F}(p)$, $v, v_1, \ldots, v_i \in S$, $e, e_1, \ldots, e_n \in \mathcal{E}(S)$, $f \in \mathcal{F}(p)$, $p, p_1, \ldots, p_n \in Pat$, and $E \in \mathcal{EC}(S)$:

- $\phi(v_1, \ldots, v_i, E, e_{i+2}, \ldots, e_n) \in \mathcal{EC}(S)$
- $E, e \in \mathcal{EC}(S)$
- $p = E \in \mathcal{EC}(S)$
- $\texttt{spawn}(f, E) \in \mathcal{EC}(S)$
- $E!e \in \mathcal{EC}(S)$
- $v!E \in \mathcal{EC}(S)$
- $\texttt{case}\ E\ \texttt{of}\ p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_n\texttt{->}e_n\ \texttt{end} \in \mathcal{EC}(S)$ ◁

The idea of an evaluation context is that the hole [] marks that position in an expression where the next evaluation will take place. Here, we have to consider the leftmost innermost evaluation strategy. Therefore, in the function application, all expressions on the left side of the evaluation context must be values. Only on the right side unevaluated expressions are allowed. In a sequence, the next evaluation is located in the first expression. Matching and `spawn` first evaluate the expression. The two rules for sending values define in which order the two arguments of `!` are evaluated: first the destination, then the value. Finally, in the case expression the next evaluation is located in the case expression.

The hole marks the point of the next evaluation. An evaluation context E matches a Core Erlang expression e if $e = E[e']$. $E[e']$ represents the Core Erlang expression, in which the hole in the context E is substituted with e'. The semantics is then determined by the different cases for e'.

In the standard operational semantics *(SOS)* we use $S = T_{\mathcal{C}}(Pid)$ and expressions and context over these values ($\mathcal{E}(T_{\mathcal{C}}(Pid))$ and $\mathcal{EC}(T_{\mathcal{C}}(Pid))$). Later we will use other values, too.

Defintion 3.13 (Operational Semantics)
The *operational semantics* is defined as a relation $\Longrightarrow \subset State \times Label \times State$. We write $s \overset{a}{\Longrightarrow} s'$ for $(s, a, s') \in \Longrightarrow$ and $s \Longrightarrow s'$ for $s \overset{\varepsilon}{\Longrightarrow} s'$. The definition is presented in Figures 3.1–3.4. ◁

The rules for sequential evaluation are standard. A sequence evaluates to the second argument, if the first one is a value. Predefined functions are evaluated with respect to their interpretation. Constructor functions are interpreted freely. The expression `self` evaluates to the corresponding pid and the application of defined functions yields the corresponding right-hand side of the definition. To avoid conflicts between the variables in the applied function and the context, we rename the free variables of the inserted expression. The free variables of e are all occurring variables, except $X_1, \ldots, X_n$. In contrast to other functional programming languages Erlang does not provide scoping. Hence, we do not have bound variables which can be renamed with α-conversion. Instead we rename free variables in the application of a defined function. Exactly these variables correspond to the bound variables in other languages. It would be sufficient to rename only the variables which occur in E and in $e[X_1/v_1, \ldots, X_n/v_n]$. However, for sake of simplicity we rename all variables in $e[X_1/v_1, \ldots, X_n/v_n]$.

$$\frac{-}{\Pi, (\pi, E[v\,,e], \mu) \Longrightarrow \Pi, (\pi, E[e], \mu)}$$

$$\frac{F \text{ a predefined function}}{\Pi, (\pi, E[F(v_1, \ldots, v_n)], \mu) \Longrightarrow \Pi, (\pi, E[F_{\mathcal{A}}(v_1, \ldots, v_n)], \mu)}$$

$$\frac{-}{\Pi, (\pi, E[\underline{c}(v_1, \ldots, v_n)], \mu) \Longrightarrow \Pi, (\pi, E[c(v_1, \ldots, v_n)], \mu)}$$

$$\frac{-}{\Pi, (\pi, E[\texttt{self}], \mu) \Longrightarrow \Pi, (\pi, E[\pi], \mu)}$$

$$\frac{\begin{array}{c} f(X_1, \ldots, X_n)\texttt{->}e\,. \in p \quad \text{and} \quad \{Y_1, \ldots, Y_m\} = vars(e) \setminus \{X_1, \ldots, X_n\} \quad \text{and} \\ Z_1, \ldots, Z_m \notin vars(E)\ (Z_i \neq Z_j \text{ for } i \neq j) \quad \text{and} \quad e' = e[Y_1/Z_1, \ldots, Y_m/Z_m] \end{array}}{\Pi, (\pi, E[f(v_1, \ldots, v_m)], \mu) \Longrightarrow \Pi, (\pi, E[e'[X_1/v_1, \ldots, X_n/v_n]], \mu')}$$

Figure 3.1: Operational Semantics — Sequential Evaluation

In the rules for pattern matching, `case` and `receive` (Figure 3.2) we use the functions match, casematch and mbmatch to model Erlang's mechanism of pattern matching. Compared to other functional languages pattern matching in Erlang is more complicated because non-linear patterns with multiple occurrences of the same variables are allowed (see Chapter 2). These functions are defined in the following definitions.

The result of the successful matching of a pattern against a value is a substitution:

Defintion 3.14 (Substitution)
A mapping $\sigma : Var \longrightarrow T_\Sigma(S)$ such that $\sigma(X) \neq X$ for only finitely many $X \in Var$ is called $T_\Sigma(S)$-*substitution* or simply *substitution* if the set of terms is irrelevant or clear from the context.
The (finite) set of variables changed by σ is called the *domain* of σ: $\mathcal{D}om(\sigma) := \{X \in Var \mid \sigma(X) \neq X\}$. If $\mathcal{D}om(\sigma) = \{X_1, \ldots, X_n\}$, then we may write σ as $\sigma = [X_n/\sigma(X_1), \ldots, X_n/\sigma(X_n)]$.
The set of all $T_\Sigma(S)$-substitutions will be denoted as $Subst(S)$, or simply $Subst$.
A substitution σ is canonically extended to Core Erlang expressions, Core Erlang contexts, patterns, and constructor terms: in the application of σ to these structures all variables X are replaced by $\sigma(X)$. $\triangleleft$

$$\frac{\mathsf{match}(p, v) = \rho}{\Pi, (\pi, E[p\texttt{=}v], \mu) \Longrightarrow \Pi, (\pi, \rho(E[v]), \mu)}$$

$$\frac{\mathsf{casematch}((p_1, \ldots, p_n), v) = (i, \rho)}{\Pi, (\pi, E[\texttt{case } v \texttt{ of } p_1\texttt{->}e_1\texttt{; } \ldots \texttt{;} p_n\texttt{->}e_n \texttt{ end}], \mu) \Longrightarrow \Pi, (\pi, \rho(E[e_i]), \mu)}$$

$$\frac{\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_m)) = (i, j, \rho)}{\begin{array}{l}\Pi, (\pi, E[\texttt{receive}p_1\texttt{->}e_1\texttt{; } \ldots \texttt{;} p_n\texttt{->}e_n\texttt{end}], (v_1, \ldots, v_j, \ldots, v_m)) \\ \overset{?v_j}{\Longrightarrow} \Pi, (\pi, \rho(E[e_i]), (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_m))\end{array}}$$

Figure 3.2: Operational Semantics — Matching

Usually, for instance in term rewriting, the codomain of substitutions can also contain variables ($\sigma : Var \longrightarrow T_{\mathcal{C}}(Var)$) [BN98]. However, in functional programming we are only interested in bindings for the variables. Therefore, we restrict substitutions in this way.

With the notion of substitutions, we can now define the function match for matching a pattern against a value. Formalizing pattern matching, it is necessary to join ($\uplus$) two substitutions. Two substitutions can only be joined if the overlapping parts are identical. We must consider this due to of the non-linear patterns of Erlang. Otherwise, the result is Fail:

Defintion 3.15 (Join of Substitutions)

$$\uplus : (Subst(T_{\mathcal{C}}(Pid)) \cup \{\mathsf{Fail}\})^2 \longrightarrow Subst(T_{\mathcal{C}}(Pid)) \cup \{\mathsf{Fail}\}$$

$$\begin{array}{rcl} \mathsf{Fail} \uplus \sigma & = & \mathsf{Fail} \\ \sigma \uplus \mathsf{Fail} & = & \mathsf{Fail} \\ \sigma_1 \uplus \sigma_2 & = & \begin{cases} \sigma_1 \cup \sigma_2, \text{ if } \forall X \in (\mathcal{D}om(\sigma_1) \cap \mathcal{D}om(\sigma_2)) : \sigma_1(X) = \sigma_2(X) \\ \mathsf{Fail} \quad\quad , \text{otherwise} \end{cases} \end{array}$$

◁

Using this function, we can define the function match for matching a pattern against a value:

Defintion 3.16 (Matching)

$$\mathsf{match} : Pat \times T_{\mathcal{C}}(Pid) \longrightarrow Subst(T_{\mathcal{C}}(Pid)) \cup \{\mathsf{Fail}\}$$
$$\mathsf{match}(X, t) = [X/t]$$

$$\mathsf{match}(\underline{c}(p_1,\ldots,p_n), c(v_1,\ldots,v_n)) = \mathsf{match}(p_1,v_1) \uplus \ldots \uplus \mathsf{match}(p_n,v_n)$$
$$\mathsf{match}(c(p_1,\ldots,p_n), c(v_1,\ldots,v_n)) = \mathsf{match}(p_1,v_1) \uplus \ldots \uplus \mathsf{match}(p_n,v_n)$$
$$\mathsf{match}(\pi,\pi) = []$$
$$\mathsf{match}(\_,\_) = \mathsf{Fail}, \text{ otherwise}$$ ◁

In the definition of match we must consider the two different kinds of constructors and the pids which can occur in the patterns. Constructors $\underline{c}/n \in \underline{\mathcal{C}}$ from the source code of the program and constructors $c/n \in \mathcal{C}$ which result from bindings applied to the pattern are both handled in the same way. They only match the corresponding constructors of the value, if also all sub-terms match. Later, in the abstraction we will have to distinguish these two kinds of constructors. A pid π in the pattern only matches the value π. In all other cases, the matching yields Fail.

In the operational semantics presented in Figure 3.2, the substitution which is the result of a successful match is propagated to the whole expression. This is necessary since Erlang has no scoping. It is also applied to the context where usually variables of $\mathcal{D}om(\sigma)$ occur again.

`case` evaluates to the expression corresponding to the first pattern that matches a given value. The function casematch yields a pair consisting of the number of the first matching pattern and the corresponding substitution. If none of the patterns match the given value, then it yields Fail:

Defintion 3.17 (casematch)

$$\mathsf{casematch} : Pat^* \times T_{\mathcal{C}}(Pid) \longrightarrow (\mathbb{N} \times Subst) \cup \{\mathsf{Fail}\}$$
$$\mathsf{casematch}((p_1,\ldots,p_n), v) = \begin{cases} (i,\sigma), \text{ if } \mathsf{match}(p_i,v) = \sigma \text{ and} \\ \qquad\qquad \mathsf{match}(p_j,v) = \mathsf{Fail} \quad \forall j < i \\ \mathsf{Fail} \quad , \text{ otherwise} \end{cases}$$ ◁

According to the result of casematch, the `case` expression evaluates to the corresponding expression. Again, the substitution is applied to the whole expression to propagate the bindings from pattern matching.

`receive` evaluates in the same manner but all values in the mailbox have to be considered. In Erlang the chronological order is:

- a pattern is successively matched against all values of the mailbox
- then the next pattern is matched

This is formalized by the function mbmatch which additionally yields the first matching value of the mailbox:

Defintion 3.18 (mbmatch)

$$\mathsf{mbmatch} : Pat^* \times T_{\mathcal{C}}(Pid)^* \longrightarrow (\mathbb{N} \times \mathbb{N} \times Subst) \cup \{\mathsf{Fail}\}$$
$$\mathsf{mbmatch}((p_1,\ldots,p_n),(v_1,\ldots,v_m)) = \begin{cases} (i,j,\sigma), \text{ if } \mathsf{match}(p_i,v_j) = \sigma \text{ and} \\ \qquad\qquad \mathsf{match}(p_i,v_k) = \mathsf{Fail} \quad \forall k < j \text{ and} \\ \qquad\qquad \mathsf{match}(p_l,v_h) = \mathsf{Fail} \quad \forall l < i, h \leq m \\ \mathsf{Fail} \quad , \text{ otherwise} \end{cases}$$ ◁

$$\frac{f(X_1,\ldots,X_n)\texttt{->}e\,.\ \in p \quad \text{and} \quad v_1,\ldots,v_n \in CT_{Pid} \quad \text{and} \quad \pi' \text{ a new pid}}{\Pi,(\pi,E[\texttt{spawn}(f,[v_1,\ldots,v_n])],\mu) \overset{\texttt{spawn}(f)}{\Longrightarrow} \Pi,(\pi,E[p'],\mu),(\pi',e[X_1/v_1,\ldots,X_n/v_n],())}$$

$$\frac{v_1 = \pi' \in Pid}{\Pi,(\pi,E[v_1\,!\,v_2],\mu),(\pi',e,\mu') \overset{!\,v_2}{\Longrightarrow} \Pi,(\pi,E[v_2],\mu),(\pi',e,\mu' : v_2)}$$

Figure 3.3: Operational Semantics — Concurrent Evaluation

The complexity of the definitions of casematch and mbmatch is caused by Erlang's semantics.

The rules for concurrent evaluation (Figure 3.3) formalize the creation of a new process and the sending action which extends the mailbox of the process $\pi' = v_1$.

Finally, we have to consider runtime errors (Figure 3.4). They can occur, if a value is sent to a non-pid, e.g. a natural number, a non-instantiated variable should be evaluated, or a matching fails. We have no rule for a receive expression, in which none of the patterns matches a value of the mailbox. This is the case if the corresponding mbmatch yields Fail. In this case the process does not crash but suspends. This is expressed in the formal semantics by the fact that such a process does not have any successor. If another process sends a value to this process, then this value is added to its mailbox. The process is awoken, if this new message matches one of the patterns. Formally, error is defined as the empty set of states.

As every interleaving semantics, our semantics is locally non-deterministic. For instance, the order in which two processes perform a send action is not determined. In combination with communication this local non-determinism can also result in a global non-determinism which is usually called confluence in the context of term rewriting systems [BN98]: if an expression (in our context a state) evaluates to two different expressions, then these two different expression can be reduced to the same irreducible expression (all processes are terminated).

Lemma 3.19
The operational semantics $\Longrightarrow$ is not confluent.

Proof: Consider the following program

```
main() -> spawn(sender,[42,self]),
          spawn(sender,[43,self]),
          receive
             X -> X
          end.
```

$$\frac{F \text{ a predefined function and } F_{\mathcal{A}}(v_1, \ldots, v_n) \text{ is not defined}}{\Pi, (\pi, E[F(v_1, \ldots, v_n)], \mu) \Longrightarrow \mathsf{error}}$$

$$\frac{v_1 \notin Pid}{\Pi, (\pi, E[v_1 \,!\, v_2], \mu) \overset{!\, v_2}{\Longrightarrow} \mathsf{error}} \qquad \frac{X \text{ is a variable}}{\Pi, (\pi, E[X], \mu) \Longrightarrow \mathsf{error}}$$

$$\frac{\mathsf{match}(p, v) = \mathsf{Fail}}{\Pi, (\pi, E[p{=}v], \mu) \Longrightarrow \mathsf{error}} \qquad \frac{\mathsf{casematch}((p_1, \ldots, p_n), v) = \mathsf{Fail}}{\Pi, (\pi, E[\texttt{case}\ v\ \texttt{of}\ m\ \texttt{end}], \mu) \Longrightarrow \mathsf{error}}$$

Figure 3.4: Operational Semantics — Runtime Errors

```
sender(V,P) -> P!V.
```

Two sender processes are spawned. The functional result of the `main` function is the value of the sender which sends the value first. Due to interleaving this can be any sender. Hence, we have two terminating evaluations

$$(@\mathbf{0}, \texttt{main}(), ()) \Longrightarrow^* (@\mathbf{0}, \mathbf{42}, (\mathbf{43}))\|(@\mathbf{1}, \mathbf{42}, ())\|(@\mathbf{2}, \mathbf{43}, ())$$

and

$$(@\mathbf{0}, \texttt{main}(), ()) \Longrightarrow^* (@\mathbf{0}, \mathbf{43}, (\mathbf{42}))\|(@\mathbf{1}, \mathbf{42}, ())\|(@\mathbf{2}, \mathbf{43}, ()) \qquad \square$$

This is a usual behavior of concurrent systems. If only one process (including its mailbox) is concideres the operational semantics $\Longrightarrow$ is deterministic:

Lemma 3.20 (Determinism of processes)
For all states $\Pi, \pi \in State$ *there exists at most one* $\pi' \in Proc$, $a \in Label$ *such that* $\Pi, \pi \overset{a}{\Longrightarrow} \Pi, \pi'$.

Proof: A simple induction over the structure of the Core Erlang contexts respectively expressions in π. $\square$

Chapter 4

Abstraction

In the last section we have defined an operational semantics for Core Erlang as a labelled transition relation over sets of processes. For the formal verification of a concrete system we are only interested in the part of this transition relation which is reachable from the initial state. We will use this part as the transition system for the verification by model checking. Unfortunately, this labelled transition system has an infinite state space because of the following facts:

1. arbitrarily many values may appear in the process expressions and the mailboxes (e.g., natural numbers)

2. the expressions can grow arbitrarily because of non-tail recursive function calls

3. the mailboxes can hold arbitrarily many values

4. arbitrarily many processes can be spawned

In the abstraction we want to reduce the state space to finite size with the condition that every path of the operational semantics is also represented in the abstraction. In [NN97, ANN98] Amtoft, Nielson and Nielson present a method to analyze Concurrent ML programs. They extract the behavior of a program, an abstraction, by type inference and prove properties of the program using this behavior. Erlang is an untyped language and we cannot transfer this technique to Erlang. We have to find another approach, but we handle the possible reasons for the infinite state space in the same way as Amtoft, Nielson and Nielson:

1. The posible values are represented by values of a finite domain. Therefore, we use the technique of abstract interpretation [CC77a, JN94, SS98].

2. We define a subclass of Core Erlang programs in which programs do not have this property. This subclass covers a large part of practical applications. Additionally, in Chapter 7 we define an abstraction of the control flow which abstracts the non-tail recursive function calls to a finite transition system.

3. and 4. We do not solve this problem here. However, other verification tools have similar problems. For instance, a CCS specification can also have an infinite state space (arbitrarily many processes can be created) and the model checking problem for CCS and LTL is undecidable in general. Tools for the verification of CCS specifications, such as the concurrency workbench [CPS90] only work for the finite state case. If the CCS specification defines an infinite transition system, then model checking does not terminate[1]. We use the same approach and obtain a finite state transition system for Core Erlang programs that use only finite parts of the mailboxes and create only finitely many processes.

In this section we focus on reducing the infinite set of values to a finite domain and describe how to evaluate expressions on this abstract domain to simulate the SOS. This yields the abstract operational semantics *(AOS)*. In [NN97] this finite domain is the set of types occurring in the program and the behavior is just a Concurrent ML program evaluating on types.

4.1 The Idea of Abstraction

We consider the following Core Erlang program:

Example 4.1

```
main() -> f(21).
f(X) -> f(X*2).
```

The state space of the operational semantics spawned by this program is infinite:

$$(@\mathbf{0}, \texttt{main()}, ()) \Longrightarrow (@\mathbf{0}, \texttt{f(21)}, ()) \Longrightarrow (@\mathbf{0}, \texttt{f(}\mathbf{21}\texttt{)}, ()) \Longrightarrow (@\mathbf{0}, \texttt{f(}\mathbf{21} * \mathbf{2}\texttt{)}, ())$$
$$\Longrightarrow (@\mathbf{0}, \texttt{f(}\mathbf{21} * \mathbf{2}\texttt{)}, ()) \Longrightarrow (@\mathbf{0}, \texttt{f(}\mathbf{42}\texttt{)}, ()) \Longrightarrow (@\mathbf{0}, \texttt{f(}\mathbf{42} * \mathbf{2}\texttt{)}, ()) \Longrightarrow \ldots$$

Note that in the second and fourth reduction there is a modification. The constructors (atoms) $\texttt{21}, \texttt{2} \in T_{\underline{C}}(\emptyset)$ are evaluated to the atoms $\mathbf{21}, \mathbf{2} \in T_{\mathcal{C}}(Pid)$.

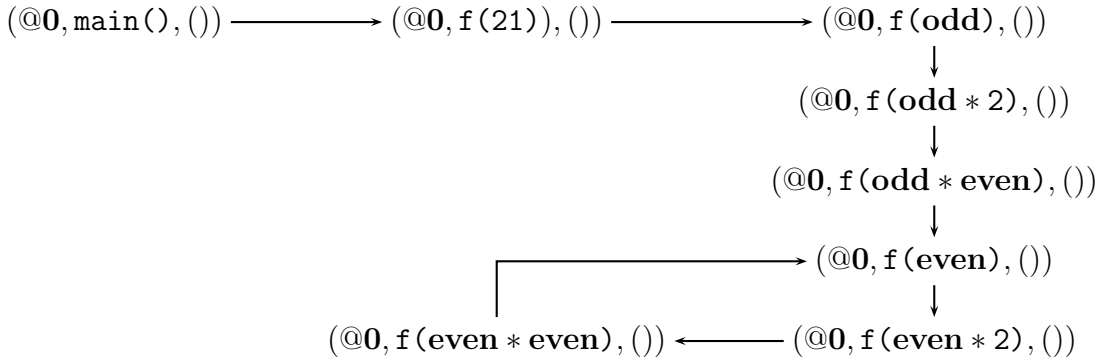
The main idea of constructing a finite model for an Erlang program is the evaluation on abstract values. We restrict the possible values to a finite domain $\widehat{A}$, for instance $\widehat{A} = \{\mathbf{even}, \mathbf{odd}\}$. The value **even** represents the values $\{2 \cdot n \mid n \in \mathbb{Z}\}$ and **odd**

[1] Using on-the-fly algorithms it can in some cases terminate, if the property can be proven on a sub-part of the transition system and by chance the model-checking algorithm verifies exactly this subpart. However, for many properties like the absence of deadlocks the whole system must be checked and model checking does not terminate.

represents the values $\{2 \cdot n + 1 \mid n \in \mathbb{Z}\}$. Similarly, we can define how to calculate on this abstract domain:

$$\widehat{\iota}^{(2)}(\texttt{*/2})(\textbf{even},\textbf{even}) = \textbf{even}$$
$$\widehat{\iota}^{(2)}(\texttt{*/2})(\textbf{odd},\ \textbf{even}) = \textbf{even}$$
$$\widehat{\iota}^{(2)}(\texttt{*/2})(\textbf{even},\textbf{odd}\ ) = \textbf{even}$$
$$\widehat{\iota}^{(2)}(\texttt{*/2})(\textbf{odd},\ \textbf{odd}\ ) = \textbf{odd}$$

The evaluation of the operational semantics by means of this abstract domain yields the finite transition system



To distinguish the abstract operational semantics from the SOS, we use the relation $\longrightarrow$ instead of $\Longrightarrow$. The infinite path in this system represents the infinite path from above but this transition system is finite and contains a cycle.

However, the abstract domain $\widehat{A} = \{\textbf{even}, \textbf{odd}\}$ causes problems. For the division of integers Erlang provides the built-in function `div` which yields the integer part of the fraction. We cannot decide, if `div`, applied to two even values, yields an even or an odd value:

$$\iota(\texttt{div/2})(\textbf{42}, \textbf{2}) = \textbf{21} \text{ and } \iota(\texttt{div/2})(\textbf{84}, \textbf{2}) = \textbf{42}$$

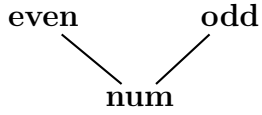
Therefore, we must add a value to the abstract domain which represents even and odd values. We call this value **num** and can define

$$\widehat{\iota}^{(2)}(\texttt{div/2})(x, y) = \textbf{num} \quad \forall x, y \in \widehat{A}$$

This new value **num** subsumes all numbers. It is less precise than the abstract values **even** and **odd**. For the formalization of this fact we use a partial order: bigger values are more precise. In our example the partial order $(\widehat{A}, \preceq)$ is defined as:

$$\preceq := \{(\textbf{even}, \textbf{even}), (\textbf{odd}, \textbf{odd}), (\textbf{num}, \textbf{num}), (\textbf{num}, \textbf{even}), (\textbf{num}, \textbf{odd})\}$$

Usually, we present partial orders as their Hasse diagrams which omit the reflexive and transitive parts of the relation:



In standard frameworks for abstract interpretations, usually complete lattices are used. The orientation of the order in these lattices is usually chosen in the other way

round (smaller values are more precise). For our approach a complete partial order instead of a complete lattice is sufficient. Partial orders are usually defined with a least element. Therefore, we use this orientation of the partial order with bigger values representing more precise values. In Section 4.4 we will compare our approach with other frameworks for abstract interpretation in more detail.

Lemma 4.2
$\langle \{\mathbf{even}, \mathbf{odd}, \mathbf{num}\}, \preceq \rangle$ *is a complete partial order.*

The proof is trivial because $\widehat{A}$ is finite. All directed sets are finite and the supremum is the greatest element of the set.

Abstract interpretations of predefined functions cannot be defined arbitrarily. They must be consistent with respect to the standard interpretation. Using the defined ordering we can formalize the safeness of abstract interpretations. For the connection between the concrete domain and the abstract domain we define an abstraction function[2] $\alpha : A \longrightarrow \widehat{A}$.

In our example, we define:

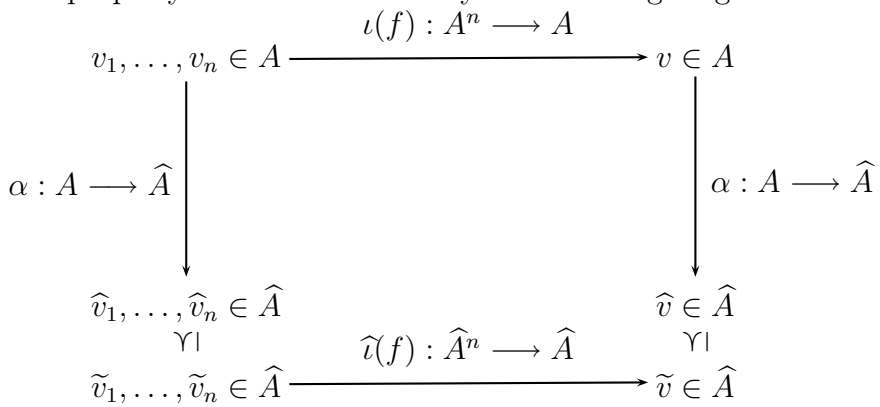
$$\alpha(v) = \begin{cases} \mathbf{even} & \text{, if } \exists n \in \mathbb{Z} \text{ with } v = 2 * n \\ \mathbf{odd} & \text{, if } \exists n \in \mathbb{Z} \text{ with } v = 2 * n + 1 \end{cases}$$

We have to guarantee that the result of the abstract interpretation of a function yields a value which represents the abstraction of the concrete interpretation of the same function. This means the result of the abstract interpretation is less precise than the abstraction of the concrete representation. Formally, this can be expressed by the following property:

(P1) For all $f/n \in \mathcal{F}, v_1, \ldots, v_n \in T_{\mathcal{C}}(Pid)$ and $\widetilde{v}_i \preceq \alpha(v_i)$ it holds that

$$\widehat{\iota}(f/n)(\widetilde{v}_1, \ldots, \widetilde{v}_n) \preceq \alpha(\iota(f/n)(v_1, \ldots, v_n))$$

The idea of this property can be illustrated by the following diagram:



[2]Often the abstraction function is defined as $\alpha : \mathcal{P}(A) \longrightarrow \widehat{A}$ combined with a concretization function $\gamma : \widehat{A} \longrightarrow \mathcal{P}$. In this case both have to build a Galois insertion. We will discuss this alternative in Section 4.4.

The main point of $\widetilde{v} \preceq \widehat{v}$ is that the two values must be comparable. Therefore, in our example it would be wrong to define

$$\widehat{\iota}(\texttt{div/2})(x, y) = \textbf{even} \quad \forall x, y \in \widehat{A}$$

because $\alpha(\iota(\texttt{div/2})(\textbf{42}, \textbf{2})) = \alpha(\textbf{21}) = \textbf{odd}$ and $\textbf{even} \not\preceq \textbf{odd}$.

On the other hand, it would not be sufficient to require that

$$\widehat{\iota}(f/n)(\alpha(v_1), \ldots, \alpha(v_n)) \preceq \alpha(\iota(f/n)(v_1, \ldots, v_n))$$

For example, it would then be possible to define

$$\widehat{\iota}(\texttt{div}/2)(\textbf{num}, \textbf{even}) = \textbf{odd}$$

for our abstract interpretation. The property is satisfied because $\forall v \in A : \alpha(v) \neq \textbf{num}$. However, during the evaluation of a Core Erlang expression this value can occur, as the following example shows:

```
main() -> X = 42 div 2,
          X div 2.
```

With the semantics:

Concrete	Abstract
$(@\textbf{0}, \texttt{main}(), ())$	$(@\textbf{0}, \texttt{main}(), ())$
$\Longrightarrow^3 (@\textbf{0}, \texttt{X} = \textbf{42 div 2}, \texttt{ X div 2}, ())$	$\longrightarrow^3 (@\textbf{0}, \texttt{X} = \textbf{even div even}, \texttt{ X div 2}, ())$
$\Longrightarrow (@\textbf{0}, \textbf{21}, \textbf{ 21 div 2}, ())$	$\longrightarrow (@\textbf{0}, \textbf{num}, \textbf{ num div 2}, ())$
$\Longrightarrow^2 (@\textbf{0}, \textbf{21 div 2}, ())$	$\longrightarrow^2 (@\textbf{0}, \textbf{num div even}, ())$
$\Longrightarrow (@\textbf{0}, \textbf{10}, ())$	$\longrightarrow (@\textbf{0}, \textbf{odd}, ())$

This is obviously wrong ($\textbf{odd} \not\preceq \alpha(\textbf{10}) = \textbf{even}$), although the property is satisfied.

4.1.1 Abstraction of Constructors

Erlang is untyped. Therefore, the domain defined above is not sufficient. We need abstract representations for arbitrary values. In other words, also for non-numerical values. Therefore, we use the value **?** which is less precise than any other value ($\textbf{?} \preceq \widetilde{v}$ for all $\widetilde{v} \in \widehat{A}$).

Erlang also provides constructor terms. These terms can be built applying constructors to other values (see Section 2.1.1). In the concrete interpretation we have interpreted these constructors freely. This is not possible in an abstract interpretation with a finite domain because constructors could be applied arbitrarily often. Hence, constructors need an abstract interpretation, too, and we have to claim an appropriate property for them:

(P1') For all $c/n \in C, v_1, \ldots, v_n \in T_{\mathcal{C}}(Pid)$ and $\widetilde{v}_i \preceq \alpha(v_i)$ it holds that

$$\widehat{\iota}(c/n)(\widetilde{v}_1, \ldots, \widetilde{v}_n) \preceq \alpha(\iota(c/n)(v_1, \ldots, v_n))$$

An example for the abstraction of constructors will be presented in Section 4.3.2.

4.1.2 Abstraction of Matching

At this point, standard frameworks for abstract interpretation [CC77a, JN94] are entirely defined. The required properties relate the concrete and the abstract domain. These frameworks add non-determinism to branching in `if-then-else`, if the condition is evaluated to an abstract value which is less precise than true and false. Unfortunately, for Core Erlang this is not sufficient. Another problem appears in the abstraction of matching, as the following program shows:

Example 4.3

```
main() -> snd({sense,42}).
snd(X) -> {Y,Z} = X,
          Z.
```

We consider the abstract domain $\widehat{A} = \{\mathbf{even}, \mathbf{odd}, \mathbf{?}\}$ with the ordering

$\preceq$: **even** **odd** **?**

To satisfy property (P1'), the atom `sense` can only be interpreted as **?** in this domain. The same holds for the tuple constructor `{}/2`:

$$\widehat{\iota}^{(2)}(\texttt{\{\}/2})(x, y) = \mathbf{?} \quad \forall x, y \in \widehat{A}$$

Again, we consider the two operational semantics of the example:

Concrete	Abstract
$(@\mathbf{0}, \texttt{main()}, ())$	$(@\mathbf{0}, \texttt{main()}, ())$
$\Longrightarrow (@\mathbf{0}, \texttt{snd(\{sense,42\})}, ())$	$\longrightarrow (@\mathbf{0}, \texttt{snd(\{sense,42\})}, ())$
$\Longrightarrow (@\mathbf{0}, \texttt{snd(\{}\mathbf{sense}\texttt{,42\})}, ())$	$\longrightarrow (@\mathbf{0}, \texttt{snd(\{?,42\})}, ())$
$\Longrightarrow (@\mathbf{0}, \texttt{snd(\{}\mathbf{sense,42}\texttt{\})}, ())$	$\longrightarrow (@\mathbf{0}, \texttt{snd(\{?,}\mathbf{even}\texttt{\})}, ())$
$\Longrightarrow (@\mathbf{0}, \texttt{snd(}\mathbf{\{}\ \mathbf{sense,42\}}\texttt{)}, ())$	$\longrightarrow (@\mathbf{0}, \texttt{snd(?)}, ())$
$\Longrightarrow (@\mathbf{0}, \texttt{\{Y,Z\}=}\mathbf{\{sense,42\}}\texttt{,Z}, ())$	$\longrightarrow (@\mathbf{0}, \texttt{\{Y,Z\}=?,Z}, ())$
$\Longrightarrow (@\mathbf{0}, \mathbf{\{sense,42\}}\texttt{,}\mathbf{42}, ())$	
$\Longrightarrow (@\mathbf{0}, \mathbf{42}, ())$	

How can this pattern matching be evaluated with respect to our abstract domain? The pattern matching in the concrete evaluation succeeds because the matching of the pattern `{Y,Z}` against the value **{sense, 42}** yields the substitution [`Y`/**sense**,`Z`/**42**]. We have modelled matching by the function match in the concrete semantics. In the abstract evaluation the tuple **{sense, 42}** is represented by the abstract value **?**. Therefore, we also need a successful pattern matching in our abstraction:

$$(@\mathbf{0}, \texttt{\{Y,Z\}=?,Z}, ()) \longrightarrow (@\mathbf{0}, \mathbf{?}, \mathbf{?}, ()) \longrightarrow (@\mathbf{0}, \mathbf{?}, ())$$

We cannot use the function match on abstract values. The value **?** also represents the value **42** and the abstract state (@**0**,`{Y,Z}=?,Z`,()) also the concrete state (@**0**,`{Y,Z}=`**42**`,Z`,()). In the concrete semantics we have

$$(@\mathbf{0}, \texttt{\{Y, Z\}=}\mathbf{42}\texttt{, Z}, ()) \Longrightarrow \mathsf{error}$$

which has to be represented in the abstract semantics, too. Therefore, matching on abstract values cannot only yield a substitution or Fail, as in the concrete case. It can yield both. To formalize this, we distinguish three cases for the abstract matching:

1. it yields a substitution: for instance, the matching `X=`**42** which will succeed for arbitrary abstractions of **42**.
2. it yields the value Fail: for instance, in the matching `42=`**odd** with the domain $\widehat{\mathcal{A}}$ from Example 4.3. We can be sure that no even value matches an odd value.
3. it can yield a substitution or Fail, as discussed before.

We represent these three cases as follows:

1. The matching is irrefutable. We indicate this by the flag Irref.
2. The matching fails, indicated by the value Fail.
3. The matching is possible but it can also fail, indicated by the flag Poss.

The match function cannot be fixed for arbitrary abstractions. It depends on the concrete abstract domain. Therefore, we add an interpretation of match to the abstraction function:

$$\widehat{\iota}(\mathsf{match}) : \widehat{Pat} \times \widehat{A} \longrightarrow (\{\mathsf{Irref}, \mathsf{Poss}\} \times \mathit{Subst}(\widehat{A})) \cup \{\mathsf{Fail}\}$$

In Erlang patterns also depend on values (e.g., pids). Therefore, we use an abstract variation of patterns $\widehat{Pat}$. E.g., pattern matching in the function definition

```
f(X) -> {X,Y}={3,4},
        Y.
```

is no matching against a tuple of variables. `X` is not a free variable in the right-hand side. From the call of `f` it is bound to a value. This has to be considered in matching. It only succeeds if `X` is bound to **3**. Otherwise we get a runtime error. Hence, some parts of a pattern are statically defined and others are added at runtime. In the abstract interpretation these are values of the abstract domain and we get:

Defintion 4.4 (Patterns)
The set of *abstract patterns* is defined as

$$\widehat{Pat}_{\widehat{A}} := T_{\underline{C}}(\mathit{Var} \cup \widehat{A})$$

In the following, the abstract domain will usually be clear from the context. In this case, we omit the index $\widehat{A}$ and write $\widehat{Pat}$. ◁

As an example, the function `f` could be applied to the abstract value **even**. Then the abstract pattern would be $\{\mathbf{even}, \texttt{Y}\} \in \widehat{Pat}$.

These non-static patterns are often used in Erlang, as the patterns `{value,V,P}` and `[{K,V}|_]` in the database process in Example 3.7. Using these patterns it is possible to compare mailbox entries containing dynamic values without extracting them from the mailbox. In the pattern `{value,V,P}` this is checking the equality of the third component of a message and the variable `P` which is bound to a pid, received before.

Again, we claim two properties for the safeness of the abstract interpretation:

(P2) For all $p \in Pat$ and $v \in A$ and for all $\widetilde{p} \preceq \alpha(p)$ and $\widetilde{v} \preceq \alpha(v)$ it holds that

a) if $\mathsf{match}(p, v) = \sigma$ then $\widehat{\iota}(\mathsf{match})(\widetilde{p}, \widetilde{v}) = (\_, \widetilde{\sigma})$ and $\widetilde{\sigma} \preceq \sigma$

b) if $\mathsf{match}(p, v) = \mathsf{Fail}$ then $\widehat{\iota}(\mathsf{match})(\widetilde{p}, \widetilde{v}) \in \{(\mathsf{Poss}, \_), \mathsf{Fail}\}$

Successful and failed matches must be represented in the abstract interpretation, to be safe with respect to the SOS.

In theses properties we compare abstract substitutions with respect to the ordering $\preceq$ on values. This natural extension is defined as:

Defintion 4.5 (Extension of Orderings to Substitutions)
Let $\preceq \subseteq \widehat{A} \times \widehat{A}$ be an ordering.

$$\sigma_1 \preceq \sigma_2 \text{ iff } \mathcal{D}om(\sigma_1) = \mathcal{D}om(\sigma_2) \text{ and } \sigma_1(X) \preceq \sigma_2(X)\ \forall X \in \mathcal{D}om(\sigma_1) \qquad \triangleleft$$

For readability we use $\preceq$ for this extension, too. The concrete instance of this overloading will be clear from the context.

In our concrete example we can define:

$$\widehat{\iota}(\mathsf{match})(\{\texttt{Y}, \texttt{Z}\}, \texttt{?}) = (\mathsf{Poss}, [\texttt{Y}/\texttt{?}, \texttt{Z}/\texttt{?}])$$

In the abstract operational semantics we have two successor states. One for the successful matching and the error state for the failed matching. We add non-determinism to be safe with respect to the concrete semantics.

4.1.3 Abstraction of Branching

In the case of branching we cannot always decide which branch has to be chosen. This is illustrated by the following example:

```
f(X) -> case X of
          0 -> 42;
          Y -> 43
        end.
```

Again, we consider the abstract **even-odd**-domain. If `f` is applied to the abstract value **odd**, then it is safe that the second branch is chosen. The abstract result is

odd. However, the abstract value **even** also represents the concrete value 0. Hence, by means of this abstract value both branches can be chosen.

In the concrete semantics we use a function

$$\mathsf{casematch} : Pat^* \times A \longrightarrow (\mathbb{N} \times Subst(A)) \cup \{\mathsf{Fail}\}$$

which defines the successive matching of patterns against a value. The result is the position of the first pattern which matches the value or Fail if none of the values matches.

As for match, we extend the abstract interpretation function to an interpretation of casematch because its behavior depends on the concrete abstraction. Not only one position and number can be chosen. We allow a finite set of possible positions and corresponding substitutions:

$$\widehat{\iota}(\mathsf{casematch}) : \widehat{Pat}^* \times \widehat{A} \longrightarrow \mathcal{P}((\mathbb{N} \times Subst(A)) \cup \{\mathsf{Fail}\})$$

It is not sufficient to represent the value Fail by the empty set of positions and substitutions. We have the same problem as for match before. If a matching is only possible but not irrefutable, then we have to yield both results: the substitution and the value Fail. Otherwise, we are not safe with respect to the SOS.

For safeness of the abstract interpretation we claim:

(P3) For all $p_1, \ldots, p_n \in Pat, v \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p}_j \sqsubseteq \alpha(p_j)$ and $\widetilde{v} \sqsubseteq \alpha(v)$ it holds that:
- a) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = (i, \rho)$, then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$ with $(i, \widetilde{\rho}) \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{v})$.
- b) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = \mathsf{Fail}$, then $\mathsf{Fail} \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{v})$.

Again, the abstract interpretation must preserve all possible results of casematch.

4.1.4 Abstraction of `receive`

The abstraction of mbmatch can be defined analogously:

$$\iota(\mathsf{mbmatch}) : \widehat{Pat}^* \times \widehat{A}^* \longrightarrow \mathcal{P}((\mathbb{N} \times \mathbb{N} \times Subst(A)) \cup \{\mathsf{Fail}\})$$

As in the SOS we handle Fail in a different way then for casematch. If the result of mbmatch is Fail, then the process does not yield an error. It just suspends. We have to represent this behavior in the abstraction, too. Therefore, it would be sufficient to claim a property similar to (P3a). However, as we will see in Section 4.6, we also need a property similar to (P3b) to detect deadlocks. The prove of the absence of deadlocks is one major goal in the verification of concurrent and distributed systems. Therefore, we claim the following properties for an abstract interpretation although, we will only need the first property here:

(P4) For all $p_1, \ldots, p_n \in Pat, v_1, \ldots, v_u \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p_j} \sqsubseteq \alpha(p_j)$ and $\widetilde{v_k} \sqsubseteq \alpha(v_k)$ it holds that:

a) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = (i, j, \rho)$,
then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$
with $(i, j, \widetilde{\rho}) \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p_1}, \ldots, \widetilde{p_n}), (\widetilde{v_1}, \ldots, \widetilde{v_u}))$.

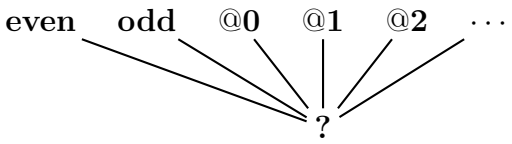
b) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = \mathsf{Fail}$,
then $\mathsf{Fail} \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p_1}, \ldots, \widetilde{p_n}), (\widetilde{v_1}, \ldots, \widetilde{v_u}))$.

4.1.5 Abstraction of pids

Finally, we have to consider the pids represented by an abstract value. As an example, we consider the sending of a value:

```
f(X) -> X!42.
```

This function sends the value **42** to the pid, the variable `X` is bound to. We consider the even-odd domain extended by pids:



If `f` is applied to the abstract values **even** or **odd**, then this should yield a runtime error. These values do not represent any pid. If `f` is applied to one of the pids in the abstract operational semantics, then this should exactly represent the concrete sending to the process with this pid. Finally, `f` can be applied to **?**. The abstract value **?** represents all values of $T_{\mathcal{C}}(Pid)$. Therefore, possible paths must contain the sending of the abstract value **even** (representing the concrete value **42**) to all existing processes. Furthermore, **?** also represents non-pid values. Therefore, the abstract semantics must also contain a runtime error.

We have not abstracted from the pids of the running processes. Due to this fact, we need a concretization of the abstract values as concrete pids. As before, we add this concretization to the abstract interpretation function:

$$\widehat{\iota}(\text{pid}) : \widehat{A} \longrightarrow \mathcal{P}(Pid \cup \mathsf{err})$$

Again, we claim a property for safeness:

(P5) a) For all $p \in Pid$ and $\widetilde{p} \preceq \alpha(p)$ it holds that $p \in \widehat{\iota}(pid)(\widetilde{p})$.
b) For all $v \notin Pat$ and $\widetilde{v} \preceq \alpha(v)$ it holds that $\mathsf{err} \in \widehat{\iota}(pid)(\widetilde{v})$.

4.2 A Framework for Abstract Interpretations

In the last section we have presented the idea of the abstract operational semantics of Core Erlang programs. In contrast to standard frameworks for abstract interpretation, it is not sufficient to relate the abstract and the concrete interpretation of the

predefined functions and add non-determinism in branching. Erlang's mechanism of pattern matching must be considered. Therefore, we have extended the abstraction by interpretations for match, casematch, and mbmatch. Finally, we have considered a concretization of values as pids. For all these abstract interpretations we have claimed properties to guarantee safeness of the abstract semantics with respect to the SOS. Now, we summarize these results in a framework for abstract interpretations of Core Erlang programs.

Defintion 4.6 (Abstract Interpretation for Core Erlang)
Let $\widehat{A}$ be a set of abstract values, called the *abstract domain*. Let $\widehat{\iota}$ be a family of *abstract interpretation functions* for

- the predefined functions $\quad \widehat{\iota}^{(n)}(F/n) : \widehat{A}^n \longrightarrow \widehat{A}$,
- the constructors $\quad \widehat{\iota}^{(n)}(\underline{c}/n) : \widehat{A}^n \longrightarrow \widehat{A}$,
- matching $\quad \widehat{\iota}(\mathsf{match}) : \widehat{Pat} \times \widehat{A} \longrightarrow (\{\mathsf{Irref}, \mathsf{Poss}\} \times \mathit{Subst}(\widehat{A})) \cup \{\mathit{Fail}\}$
- matching a list of patterns against a value
  $\widehat{\iota}(\mathsf{casematch}) : \widehat{Pat}^* \times \widehat{A} \longrightarrow \mathcal{P}((\mathbb{N} \times \mathit{Subst}(\widehat{A})) \cup \mathsf{Fail})$,
- matching a list of patterns against a mailbox
  $\widehat{\iota}(\mathsf{mbmatch}) : \widehat{Pat}^* \times \widehat{A}^* \longrightarrow \mathcal{P}((\mathbb{N} \times \mathbb{N} \times \mathit{Subst}(\widehat{A})) \cup \mathsf{Fail})$,
- and the possible pids a value of $\widehat{A}$ represents $\quad \widehat{\iota}(\mathrm{pid}) : \widehat{A} \longrightarrow \mathcal{P}(\mathit{Pid} \cup \{\mathsf{err}\})$.

Let $\sqsubseteq \subset \widehat{A} \times \widehat{A}$ be a partial order on the abstract domain and $\alpha : T_{\mathcal{C}}(\mathit{Pid}) \longrightarrow \widehat{A}$ an abstraction function. Then $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ is called an *abstract interpretation* for Core Erlang programs if it satisfies properties (P1) – (P5) of Figure 4.1.

Again, we write $\widehat{\iota}(f/n)$ instead of $\widehat{\iota}^{(n)}(f)$ and $f_{\widehat{\mathcal{A}}}$ instead of $\widehat{\iota}(f/n)$, if n is clear from the context.

If $|\widehat{A}| < \infty$, then $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ is called a *finite domain abstract interpretation*. ◁

It is sufficient that the ordering $\sqsubseteq \subseteq \widehat{A} \times \widehat{A}$, which describes which values are more precise than others, is a partial order. A complete lattice as in other frameworks is not necessary since we will not use fixed-point computations. We just evaluate the operational semantics and consider the transition system which is reachable from the initial state (@**0**,`main()`,()).

Defintion 4.7 (Abstract Operational Semantics)
The *abstract states*, *processes*, *mailboxes*, and *labels* are defined analogously to the concrete case:

$$\begin{aligned}
\widehat{\mathit{State}}_{\widehat{\mathcal{A}}} &:= \mathcal{P}_{\mathit{fin}}(\widehat{\mathit{Proc}}) \cup \{\mathsf{dead}\},\\
\widehat{\mathit{Proc}}_{\widehat{\mathcal{A}}} &:= \mathit{Pid} \times T(\widehat{A}) \times \widehat{\mathit{Mb}}_{\widehat{\mathcal{A}}}\\
\widehat{\mathit{Mb}}_{\widehat{\mathcal{A}}} &:= \widehat{A}^*\\
\widehat{\mathit{Label}}_{\widehat{\mathcal{A}}} &:= \{\mathtt{!}v \mid v \in \widehat{\mathcal{A}}\} \cup \{\mathtt{?}v \mid v \in \widehat{\mathcal{A}}\} \cup \{\mathtt{spawn(}f\mathtt{)} \mid f/n \in \mathit{FS}(p)\} \cup \{\varepsilon\}
\end{aligned}$$

(P1) For all $\phi/n \in \Sigma \cup \underline{\mathcal{C}}, v_1, \ldots, v_n \in T_{\mathcal{C}}(Pid)$ and $\widetilde{v}_i \sqsubseteq \alpha(v_i)$, it holds that $\phi_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n) \sqsubseteq \alpha(\phi_{\mathcal{A}}(v_1, \ldots, v_n))$.

(P2) For all $p \in Pat$, $v \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p} \sqsubseteq \alpha(p)$, $\widetilde{v} \sqsubseteq \alpha(v)$, it holds that
a) if $\mathsf{match}(p, v) = \rho$
then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$ with $\mathsf{match}_{\widehat{\mathcal{A}}}(\widetilde{p}, \widetilde{v}) = (\_, \widetilde{\rho})$.
b) if $\mathsf{match}(p, v) = \mathsf{Fail}$
then $\mathsf{match}_{\widehat{\mathcal{A}}}(\widetilde{p}, \widetilde{v}) \in \{(\mathsf{Poss}, \_), \mathsf{Fail}\}$.

(P3) For all $p_1, \ldots, p_n \in Pat, v \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p_j} \sqsubseteq \alpha(p_j)$ and $\widetilde{v} \sqsubseteq \alpha(v)$, it holds that:
a) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = (i, \rho)$,
then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$ with $(i, \widetilde{\rho}) \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{v})$.
b) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = \mathsf{Fail}$,
then $\mathsf{Fail} \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{v})$.

(P4) For all $p_1, \ldots, p_n \in Pat, v_1, \ldots, v_u \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p_j} \sqsubseteq \alpha(p_j)$ and $\widetilde{v}_k \sqsubseteq \alpha(v_k)$, it holds that:
a) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = (i, j, \rho)$,
then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$
with $(i, j, \widetilde{\rho}) \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), (\widetilde{v}_1, \ldots, \widetilde{v}_u))$.
b) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = \mathsf{Fail}$,
then $\mathsf{Fail} \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), (\widetilde{v}_1, \ldots, \widetilde{v}_u))$.

(P5) a) For all $p \in Pid$, $\widetilde{p} \sqsubseteq \alpha(p)$, it holds that $p \in \mathrm{pid}_{\widehat{\mathcal{A}}}(\widetilde{p})$.
b) For all $v \notin Pid$, $\widetilde{v} \sqsubseteq \alpha(v)$, it holds that $\mathsf{err} \in \mathrm{pid}_{\widehat{\mathcal{A}}}(\widetilde{v})$.

Figure 4.1: Properties of an Abstract Interpretation

The *abstract operational semantics* $\longrightarrow_{\widehat{\mathcal{A}}} \subseteq \widehat{State}_{\widehat{\mathcal{A}}} \times \widehat{Label}_{\widehat{\mathcal{A}}} \times \widehat{State}_{\widehat{\mathcal{A}}}$ is defined by the rules in Figure 4.2. It is defined analogously to $\Longrightarrow$, except that the evaluation may be non-deterministic at some points. These points result from the abstract interpretation. Figure 4.2 shows only the differences to $\Longrightarrow$. $\lhd$

In the motivation of the properties (P1) – (P5) we have argued that an abstraction is safe. Now we formalize this relationship between $\Longrightarrow$ and $\longrightarrow_{\widehat{\mathcal{A}}}$ and prove that our framework guarantees safeness. We extend the partial order $\sqsubseteq \subseteq \widehat{A} \times \widehat{A}$ on Core Erlang expressions, contexts, mailboxes, processes, states, and labels. In Definition 4.8 the extension on Core Erlang expressions $\mathcal{E}(\widehat{A})$ is defined. We only present this extension. The others are handled in the same way. An expression is smaller than another expression iff both have the same structure and all occurring values are smaller.

[2]For better readability values, substitutions and patterns over the abstract domain $\widehat{\mathcal{A}}$ are denoted with a tilde.

$$\frac{\phi/n \in \mathcal{F} \cup \underline{CF}}{\Pi, (\pi, E[\phi(v_1, \ldots, v_n)], \mu) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, E[\phi_{\widehat{\mathcal{A}}}(v_1, \ldots, v_n)], \mu)}$$

$$\frac{-}{\Pi, (\pi, E[\texttt{self}], \mu) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, E[\alpha(\pi)], \mu)}$$

$$\frac{\mathsf{match}_{\widehat{\mathcal{A}}}(p, v) = (\_, \rho)}{\Pi, (\pi, E[p\texttt{=}v], \mu) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, \rho(E[v]), \mu)}$$

$$\frac{\mathsf{match}_{\widehat{\mathcal{A}}}(p, v) = (\mathsf{Poss}, \_) \text{ or } \mathsf{match}_{\widehat{\mathcal{A}}}(p, v) = \mathsf{Fail}}{\Pi, (\pi, E[p\texttt{=}v], \mu) \longrightarrow_{\widehat{\mathcal{A}}} \mathsf{error}}$$

$$\frac{(i, \rho) \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((p_1, \ldots, p_m), v)}{\Pi, (\pi, E[\texttt{case } v \texttt{ of } p_1\texttt{->}e_1\texttt{; } \ldots \texttt{;} p_m\texttt{->}e_m \texttt{ end}], \mu) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, \rho(E[e_i]), \mu)}$$

$$\frac{\mathsf{Fail} \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((p_1, \ldots, p_m), v)}{\Pi, (\pi, E[\texttt{case } v \texttt{ of } p_1\texttt{->}e_1\texttt{; } \ldots \texttt{;} p_m\texttt{->}e_m \texttt{ end}], \mu) \longrightarrow_{\widehat{\mathcal{A}}} \mathsf{error}}$$

$$\frac{(i, j, \rho) \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((p_1, \ldots, p_m), (v_1, \ldots, v_u))}{\begin{array}{l}\Pi, (\pi, E[\texttt{receive } p_1\texttt{->}e_1\texttt{; } \ldots \texttt{;} p_m\texttt{->}e_m\texttt{end}], (v_1, \ldots, v_j, \ldots, v_u)) \\ \xrightarrow{?v_j}_{\widehat{\mathcal{A}}} \quad \Pi, (\pi, \rho(E[e_i]), (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_u))\end{array}}$$

$$\frac{\pi' \in \mathrm{pid}_{\widehat{\mathcal{A}}}(v_1)}{\Pi, (\pi, E[v_1 \,!\, v_2], \mu)(\pi', e, \mu') \xrightarrow{!\, v_2}_{\widehat{\mathcal{A}}} \Pi, (\pi, E[v_2], \mu)(\pi', e, \mu' : v_2)}$$

$$\frac{\mathsf{err} \in \mathrm{pid}_{\widehat{\mathcal{A}}}(v_1)}{\Pi, (\pi, E[v_1 \,!\, v_2], \mu) \xrightarrow{!\, v_2}_{\widehat{\mathcal{A}}} \mathsf{error}}$$

Figure 4.2: Abstract Operational Semantics (differences to SOS)

Defintion 4.8 (Ordering on Expressions)
The partial order $\sqsubseteq$[3] $\subseteq\ T(\widehat{A}) \times T(\widehat{A})$ is the smallest set with:

$v_1 \sqsubseteq v_2$	iff $v_1 \sqsubseteq v_2$
$\phi(e_1, \ldots, e_n) \sqsubseteq \phi(e'_1, \ldots, e'_n)$	iff $e_i \sqsubseteq e'_i$ for all $1 \leq i \leq n$
$X \sqsubseteq X$	
p`=`$e \sqsubseteq p'$`=`e'	iff $p \sqsubseteq p'$ and $e \sqsubseteq e'$
`self` $\sqsubseteq$ `self`	
e_1`,`$e_2 \sqsubseteq e'_1$`,`e'_2	iff $e_1 \sqsubseteq e'_1$ and $e_2 \sqsubseteq e'_2$
`case` e `of` m `end` $\sqsubseteq$ `case` e' `of` m' `end`	iff $e \sqsubseteq e'$ and $m \sqsubseteq m'$
`receive` m `end` $\sqsubseteq$ `receive` m' `end`	iff $m \sqsubseteq m'$
e_1`!`$e_2 \sqsubseteq e'_1$`!`e'_2	iff $e_1 \sqsubseteq e'_1$ and $e_2 \sqsubseteq e'_2$
`spawn(`f`,`e`)` $\sqsubseteq$ `spawn(`f`,`e'`)`	iff $e \sqsubseteq e'$
p_1`->`e_1`;` $\ldots$ `;`p_n`->`$e_n \sqsubseteq p'_1$`->`e'_1`;` $\ldots$ `;`p'_n`->`e'_n	iff $p_i \sqsubseteq p'_i, e_i \sqsubseteq e'_i\ \ \forall 1 \leq i \leq n$

◁

We also extend the abstraction function $\alpha : T_{\mathcal{C}}(Pid) \longrightarrow \widehat{A}$ on Core Erlang expressions, contexts, mailboxes, processes, states, and labels in this canonical way. The abstraction of a state is the state in which all values of $T_{\mathcal{C}}(Pid)$ are replaced by their abstract representation in $\widehat{A}$.

Lemma 4.9 (Safeness of abstract substitutions)
Let $e \in \mathcal{E}(T_{\mathcal{C}}(Pid))$ and σ be a finite substitution. If $\widetilde{\sigma} \sqsubseteq \alpha(\sigma)$ and $\widetilde{e} \sqsubseteq \alpha(e)$ then $\widetilde{\sigma}(\widetilde{e}) \sqsubseteq \alpha(\sigma(e))$.

Proof: The only parts of an expression where a substitution has an effect are the variables. The rest of the expression does not change and the lemma is satisfied. This can be checked by a simple induction over the structure of the Core Erlang expressions. The only cases we have to consider are:

- $X \in \mathcal{D}om(\sigma)$:
  $\sigma(X) = v \in T_{\mathcal{C}}(Pid)$ because $\widetilde{\sigma} \sqsubseteq \alpha(\sigma)$ $\widetilde{\sigma}(X) = \widetilde{v} \in \widehat{A}$ with $\widetilde{v} \sqsubseteq \alpha(v)$.
- $X \notin \mathcal{D}om(\sigma)$:
  $\sigma(X) = X$ and only $X \sqsubseteq \alpha(X)$. On the other hand, for all $\widetilde{\sigma} \sqsubseteq \alpha(\sigma)$ also $\widetilde{\sigma}(X) = X$. □

Lemma 4.10
Let $E \in \mathcal{EC}(T_{\mathcal{C}}(Pid))$ be a context and $e \in \mathcal{E}(T_{\mathcal{C}}(Pid))$ be a Core Erlang expression. If $\widetilde{E} \sqsubseteq \alpha(E)$ and $\widetilde{e} \sqsubseteq \alpha(e)$ then $\widetilde{E}[\widetilde{e}] \sqsubseteq \alpha(E[e])$.

Proof: A simple induction over the structure of E. □

Theorem 4.11 (Safeness of $\longrightarrow_{\widehat{\mathcal{A}}}$)
Let $\widehat{\mathcal{A}}$ be an abstract interpretation that satisfies properties (P1) – (P5).
If $s \overset{a}{\Longrightarrow} t$ then, for all $\widetilde{s} \sqsubseteq \alpha(s)$, there exists $\widetilde{t} \sqsubseteq \alpha(t)$ and $\widetilde{b} \sqsubseteq \alpha(a)$ such that $\widetilde{s} \overset{\widetilde{b}}{\longrightarrow}_{\widehat{\mathcal{A}}} \widetilde{t}$.

[3] We use the same symbol as for the ordering on substitutions to avoid too many different symbols for the extensions of $\sqsubseteq$. The concrete instance of this overloading will be clear from the context.

Proof: We prove the safeness of the abstraction by induction over the structure of the possible reducible expressions of the process p in s. These redexes are defined by the reduction contexts. For the concluding step we always use Lemma 4.10

- $s = \Pi, (\pi, E[X], \mu) \Longrightarrow \mathsf{error}$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[X], \widetilde{\mu})$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$ and $\widetilde{\mu} \sqsubseteq \alpha(\mu)$.
  Then also $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \mathsf{error}$.

- $s = \Pi, (\pi, E[v\,,e], \mu) \Longrightarrow \Pi, (\pi, E[e], \mu) = t$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[\widetilde{v}\,,\widetilde{e}], \widetilde{\mu})$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$, $\widetilde{v} \sqsubseteq \alpha(v)$, $\widetilde{e} \sqsubseteq \alpha(e)$, and $\widetilde{\mu} \sqsubseteq \alpha(\mu)$. Then also $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \widetilde{\Pi}, (\pi, \widetilde{E}[\widetilde{e}], \widetilde{\mu}) \sqsubseteq \alpha(t)$.

- $s = \Pi, (\pi, E[F(v_1, \ldots, v_n)], \mu) \Longrightarrow \Pi, (\pi, E[F_{\mathcal{A}}(v_1, \ldots, v_n)], \mu) = t$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[F(\widetilde{v}_1, \ldots, \widetilde{v}_n)], \widetilde{\mu})$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$, $\widetilde{v}_i \sqsubseteq \alpha(v_i)$ $\forall 1 \leq i \leq n$ and $\widetilde{\mu} \sqsubseteq \alpha(\mu)$.
  Then $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \widetilde{\Pi}, (\pi, \widetilde{E}[F_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n)], \widetilde{\mu}) = \widetilde{t}$. With property (P1) we know that $F_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n) \sqsubseteq \alpha(F_{\mathcal{A}}(v_1, \ldots, v_n))$ and conclude $\widetilde{t} \sqsubseteq \alpha(t)$.

- $s = \Pi, (\pi, E[\underline{c}(v_1, \ldots, v_n)], \mu) \Longrightarrow \Pi, (\pi, E[c(v_1, \ldots, v_n)], \mu) = t$:
  The proof can be done in the same way as for $E[F(e_1, \ldots, e_n)]$ with the use of (P1).

- $s = \Pi, (\pi, E[\texttt{self}], \mu) \Longrightarrow \Pi, (\pi, E[p], \mu) = t$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[\texttt{self}], \widetilde{\mu})$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$ and $\widetilde{\mu} \sqsubseteq \alpha(\mu)$.
  Then also $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \widetilde{\Pi}, (\pi, \widetilde{E}[\alpha(\pi)], \widetilde{\mu}) \sqsubseteq \alpha(t)$.

- $s = \Pi, (\pi, E[f(v_1, \ldots, v_n)], \mu) \Longrightarrow \Pi, (\pi, E[e'[X_1/v_1, \ldots, X_n/v_n)], \mu) = t$ if $f(X_1, \ldots, X_n)\texttt{->}e. \in Pat$ and e' is a variant of e with fresh free variables:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[f(\widetilde{v}_1, \ldots, \widetilde{v}_n)], \widetilde{\mu})$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$, $\widetilde{v}_i \sqsubseteq \alpha(v_i)$ $\forall 1 \leq i \leq n$ and $\widetilde{\mu} \sqsubseteq \alpha(\mu)$.
  Then $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \widetilde{\Pi}, (\pi, \widetilde{E}[e'[X_1/\widetilde{v}_1, \ldots, X_n/\widetilde{v}_n)], \widetilde{\mu}) = \widetilde{t}$.
  With Lemma 4.9 we conclude $\widetilde{t} \sqsubseteq \alpha(t)$.

- $s = \Pi, (\pi, E[p\texttt{=}v], \mu)$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[\widetilde{p}\texttt{=}\widetilde{v}], \widetilde{\mu})$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$, $\widetilde{\pi} \sqsubseteq \alpha(\pi)$, $\widetilde{v} \sqsubseteq \alpha(v)$, and $\widetilde{\mu} \sqsubseteq \alpha(\mu)$.

  - $s \Longrightarrow \Pi, (p, \sigma(E[v]), \mu) = t$ iff $\mathsf{match}(p, v) = \sigma \neq \mathsf{Fail}$.
    With property (P2a) we know that $\mathsf{match}_{\widehat{\mathcal{A}}}(\widetilde{p}, \widetilde{v}) = (\_, \widetilde{\sigma})$ with $\widetilde{\sigma} \sqsubseteq \alpha(\sigma)$ and therefore $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \widetilde{\Pi}, (\pi, \widetilde{\sigma}(\widetilde{E}[\widetilde{v}]), \widetilde{\mu}) = \widetilde{t}$ and $\widetilde{t} \sqsubseteq \alpha(t)$ because of Lemma 4.9.
  - $s \Longrightarrow \mathsf{error}$ if $\mathsf{match}(p, v) = \mathsf{Fail}$.
    With property (P2b) $\mathsf{match}_{\widehat{\mathcal{A}}}(\widetilde{p}, \widetilde{v}) = \mathsf{Fail}$ and $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \mathsf{error}$.

- $s = \Pi, (\pi, E[\texttt{case}\ v\ \texttt{of}\ p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_n\texttt{->}e_n\ \texttt{end}], \mu)$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[\texttt{case}\ \widetilde{v}\ \texttt{of}\ \widetilde{p}_1\texttt{->}\widetilde{e}_1\texttt{;}\ \ldots\ \texttt{;}\widetilde{p}_n\texttt{->}\widetilde{e}_n\ \texttt{end}], \widetilde{\mu})$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$, $\widetilde{v} \sqsubseteq \alpha(v)$, $\widetilde{p}_i \sqsubseteq \alpha(p_i)$ and $\widetilde{e}_i \sqsubseteq \alpha(e_i)$ $\forall 1 \leq i \leq n$, and $\widetilde{\mu} \sqsubseteq \alpha(\mu)$.

  - $s \Longrightarrow \Pi, (p, \sigma(E[e_i]), \mu) = t$ iff $\mathsf{casematch}(p, v) = (i, \sigma)$.
    By (P3a) we know that $(i, \widetilde{\sigma}) \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{v})$ with $\widetilde{\sigma} \sqsubseteq \alpha(\sigma)$. Therefore, $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \widetilde{\Pi}, (\pi, \widetilde{\sigma}(\widetilde{E}[\widetilde{e}_i]), \widetilde{\mu}) = \widetilde{t}$ and $\widetilde{t} \sqsubseteq \alpha(t)$ because of Lemma 4.9.
  - $s \Longrightarrow \mathsf{error}$ if $\mathsf{casematch}(p, v) = \mathsf{Fail}$.
    With property (P3b) $\mathsf{Fail} \in \mathsf{casematch}_{\widehat{\mathcal{A}}}(\widetilde{p}, \widetilde{v})$ and $\widetilde{s} \longrightarrow_{\widehat{\mathcal{A}}} \mathsf{error}$.

- $s = \Pi, (\pi, E[\texttt{receive } p_1\texttt{->}e_1\texttt{; } \ldots \texttt{;} p_n\texttt{->}e_n \texttt{ end}], (v_1, \ldots, v_m))$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi, \widetilde{E}[\texttt{receive } \widetilde{p}_1\texttt{->}\widetilde{e}_1\texttt{; } \ldots \texttt{;} \widetilde{p}_n\texttt{->}\widetilde{e}_n \texttt{ end}], (\widetilde{v}_1, \ldots, \widetilde{v}_m))$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{E} \sqsubseteq \alpha(E)$, $\widetilde{p}_i \sqsubseteq \alpha(p_i)$ and $\widetilde{e}_i \sqsubseteq \alpha(e_i)$ $\forall 1 \leq i \leq n$, and $\widetilde{v}_i \sqsubseteq \alpha(v_i)$ $\forall 1 \leq i \leq m$.
  $s \overset{?v_j}{\Longrightarrow} \Pi, (p, \sigma(E[e_i]), (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_m)) = t$ iff $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_m)) = (i, j, \sigma)$.
  By (P4) we know that $(i, j, \widetilde{\sigma}) \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), (\widetilde{v}_1, \ldots, \widetilde{v}_m))$ with $\widetilde{\sigma} \sqsubseteq \alpha(\sigma)$. Therefore, $\widetilde{s} \xrightarrow{?\widetilde{v}_j}_{\widehat{\mathcal{A}}} \widetilde{\Pi}, (\pi, \widetilde{\sigma}(\widetilde{E}[\widetilde{e}_i]), (\widetilde{v}_1, \ldots, \widetilde{v}_{j-1}, \widetilde{v}_{j+1}, \ldots, \widetilde{v}_m)) \sqsubseteq \alpha(t)$ because of Lemma 4.9.

- $s = \Pi, (\pi, E[\texttt{spawn}(f\texttt{, [}e_1\texttt{, } \ldots \texttt{, }e_n\texttt{])}], \mu)$:
  The proof can be done in the same way as for the function call $f(e_1, \ldots, e_n)$.

- $s = \Pi, (\pi_1, e, q_1), (\pi_2, E[v_1\,!\,v_2], q_2)$:
  $\widetilde{s} \sqsubseteq \alpha(s)$ implies $\widetilde{s} = \widetilde{\Pi}, (\pi_1, \widetilde{e}, \widetilde{q}_1)(\pi_2, \widetilde{E}[\widetilde{v}_1\,!\,\widetilde{v}_2, \widetilde{q}_2)$ with $\widetilde{\Pi} \sqsubseteq \alpha(\Pi)$, $\widetilde{e} \sqsubseteq \alpha(e)$, $\widetilde{E} \sqsubseteq \alpha(E)$, $\widetilde{q}_{1/2} \sqsubseteq \alpha(q_{1/2})$, and $\widetilde{v}_{1/2} \sqsubseteq \alpha(v_{1/2})$.
  We distinguish two cases:

  - $v_1 = \pi_1 \in \mathit{Pid}$ and $s \overset{!v_2}{\Longrightarrow} \Pi, (\pi_1, e, q_1 : v), (\pi_2, E[v_2], q_2) = t$.
    By (P5a) we know that $\pi_1 \in \mathit{pid}_{\widehat{\mathcal{A}}}(\widetilde{v}_1)$ and we get $\widetilde{s} \xrightarrow{!\widetilde{v}_2} \widetilde{\Pi}, (\pi_1, \widetilde{e}, \widetilde{q}_1 : \widetilde{v}), (\pi_2, \widetilde{v}, \widetilde{q}_2) \sqsubseteq \alpha(t)$.

  - $v_1 \notin \mathit{Pid}$ and $s \overset{!v_2}{\Longrightarrow} \mathsf{error}$.
    By (P5b) we know that $\mathsf{err} \in \mathrm{pid}_{\widehat{\mathcal{A}}}(\widetilde{v}_1)$ and also $\widetilde{s} \xrightarrow{!\widetilde{v}_2} \mathsf{error} \sqsubseteq \mathsf{error}$. □

Theorem 4.12 (Safeness of the Abstraction)
Let $\widehat{\mathcal{A}}$ be an abstract interpretation that satisfies properties (P1) – (P5) *and* $s_0 = (\texttt{main()}, ())$. *For every path* $s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} s_n$ *of the operational semantics, there exists an abstract path* $\widetilde{s}_0 \xrightarrow{\widetilde{a}_1}_{\widehat{\mathcal{A}}} \widetilde{s}_1 \xrightarrow{\widetilde{a}_2}_{\widehat{\mathcal{A}}} \ldots \xrightarrow{\widetilde{a}_n}_{\widehat{\mathcal{A}}} \widetilde{s}_n$ *with* $\widetilde{s}_0 = \alpha(s_0) = s_0$ *and* $\widetilde{s}_j \sqsubseteq \alpha(s_j), \widetilde{a}_j \sqsubseteq \alpha(a_j)$ *for all* $1 \leq j \leq n$.

Proof: Using Theorem 4.11 the proof is a simple induction over n. □

4.3 Example Abstractions

In this section, we define two example abstractions: The first abstraction shows the universal applicability of our framework. It is equivalent to the SOS. The second abstraction is a finite domain abstraction. It is particularly designed for the verification of Erlang programs with model checking and can be used to show many properties of the communication behavior of Core Erlang programs. We will show this in Chapter 5 where this abstraction is used for the verification of the database example.

4.3.1 The Operational Semantics

The most precise abstract interpretation is the one which evaluates exactly the operational semantics $\Longrightarrow$:

Defintion 4.13 (The Most Precise Abstract Interpretation $\widehat{\mathcal{A}}_{max}$)
The abstract interpretation $\widehat{\mathcal{A}}_{max}$ is defined as

$$\widehat{\mathcal{A}}_{max} = (T_{\mathcal{C}}(Pid), \widehat{\iota}, =, \mathsf{id})$$

with

$$\widehat{\iota}^{(n)}(F/n) = \iota(F/n)$$

$$\widehat{\iota}^{(n)}(\underline{c}/n)(v_1, \ldots, v_n) = c(v_1, \ldots, v_n) \text{ for all } v_i \in T_{\mathcal{C}}(Pid)$$

$$\widehat{\iota}(\mathsf{match})(p, v) = \begin{cases} (\mathsf{Irref}, \sigma), & \text{if } \mathsf{match}(p, v) = \sigma \\ \mathsf{Fail} & \text{, if } \mathsf{match}(p, v) = \mathsf{Fail} \end{cases}$$

$$\widehat{\iota}(\mathsf{casematch})(\overline{p}, v) = \{\mathsf{casematch}(\overline{p}, v)\}$$

$$\widehat{\iota}(\mathsf{mbmatch})(\overline{p}, \overline{v}) = \{\mathsf{mbmatch}(\overline{p}, \overline{v})\}$$

$$\widehat{\iota}(\mathrm{pid})(v) = \begin{cases} \{v\}, & \text{if } v \in Pid \\ \{\mathsf{err}\}, & \text{otherwise} \end{cases}$$

and equality as partial order and the identity as abstraction function. ◁

Corollary 4.14 (Safeness of $\widehat{\mathcal{A}}_{max}$)
The abstract interpretation $\widehat{\mathcal{A}}_{max}$ satisfies properties (P1) – (P5).

Lemma 4.15 (Soundness of $\widehat{\mathcal{A}}_{max}$)
For the abstract interpretation $\widehat{\mathcal{A}}_{max}$ and for all $s \in States$ it holds that:

$$s \overset{a}{\Longrightarrow} t \quad \textit{iff} \quad s \overset{a}{\longrightarrow}_{\widehat{\mathcal{A}}_{max}} t$$

Proof: Trivial, since $\widehat{\mathcal{A}}_{max}$ defines no non-determinism. □

4.3.2 A Finite Domain Abstraction

Erlang is an untyped language. Hence, we cannot use types as abstract values for our evaluation as in [NN97]. We have to find another finite set on which we can evaluate and decide some branches of the evaluation. In Erlang programs the use of constructors is very convenient. For our interests the relevant uses of constructors are patterns. These patterns only consider a finite part of the possible values. Hence, in this abstraction we restrict the possible values for the computation to a finite set of possible tops of the constructor terms. This can be done with respect to a parameter $k \in \mathbb{N}$ for there depth. The restricted domain is a subset of $T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid)$. The occurring question marks in these constructor terms represent subterms, from which we have abstracted. E.g., **{succ,?}** can represent **{succ,succ}**, **{succ,{1,2}}**, or **{succ,@0}**. To characterize what it means that a representation in $T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid)$ is more precise than another one, we define a partial order $\preceq \subseteq T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid) \times T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid)$.

Defintion 4.16 (Ordering on $T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid)$)

$$\begin{array}{l} \mathbf{?} \preceq v \text{ and } v \preceq v \text{ for all } v \in T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid) \\ c(e_1, \ldots, e_n) \preceq c(e'_1, \ldots, e'_n) \text{ iff } e_i \preceq e'_i \text{ for all } 1 \leq i \leq n \end{array} \qquad \lhd$$

We define a finite subset of $T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid)$ by restricting the constructor terms to depth $k \in \mathbb{N}$ and only a finite set of atoms. The depth of a constructor term is defined as:

$$\begin{array}{lcl} \mathsf{depth}(\mathbf{?}) & = & 0 \\ \mathsf{depth}(c(v_1, \ldots, v_n)) & = & \max\limits_{1 \leq i \leq n} (\mathsf{depth}(v_i)) + 1 \end{array}$$

with $\mathsf{max}(\emptyset) = 0$. Hence, for atoms we get $\mathsf{depth}(a/0) = 1$. Depth restricted constructor terms are defined as:

$$\widehat{T}^k_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid) := \{\widetilde{v} \in T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid) \mid \mathsf{depth}(\widetilde{v}) \leq k\}$$

This is still an infinite set. There is an arbitrary number of atoms, in particular containing all natural numbers. Therefore, we must restrict the atoms to a finite set. In Core Erlang branching is only possible in `case` and `receive` expressions. To decide as many branches as possible, the values corresponding to the patterns should be considered. In our database example relevant patterns are `{allocate,Key,P}` and `{lookup,Key,P}`. The atoms `allocate` and `lookup` are used as flags to distinguish the received messages. Hence, the abstract domain should contain these atoms. We scan the program with the function patoms from Figure 4.3 and restrict the possible atoms to the resulting set. All other atoms are abstracted to **?**.

In the example program p_{db} the extraction of atoms yields:

$$\mathsf{patoms}(p_{db}) \;=\; \{\texttt{allocate}, \texttt{fail}, \texttt{free}, \texttt{value}, \texttt{succ}, \texttt{allocated}, \texttt{lookup}\}$$

For constructing abstract constructor terms we only use these extracted atoms. We obtain the abstract domain $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid) \subseteq \widehat{T}^k_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid)$ with

$$\widehat{\mathcal{C}}_p = \{[.|.]/2, [\,]/0\} \;\cup\; \{\{\ldots\}/n \mid n \leq K\} \;\cup\; \mathsf{patoms}(p)$$

$$
\begin{array}{lcl}
\mathsf{patoms}(p_1\ p_2) & = & \mathsf{patoms}(p_1) \cup \mathsf{patoms}(p_2) \\
\mathsf{patoms}(f(X_1,\ldots,X_n)\texttt{->}e\,.) & = & \mathsf{eatoms}(e) \\
 & & \\
\mathsf{eatoms}(a/0) & = & \{a\} \\
\mathsf{eatoms}(\phi(e_1,\ldots,e_n)) & = & \bigcup_{1\le i\le n} \mathsf{eatoms}(e_i) \\
\mathsf{eatoms}(X) & = & \emptyset \\
\mathsf{eatoms}(p\texttt{=}e) & = & \mathsf{eatoms}(p) \cup \mathsf{eatoms}(e) \\
\mathsf{eatoms}(\texttt{self}) & = & \emptyset \\
\mathsf{eatoms}(\texttt{case}\ e\ \texttt{of}\ m\ \texttt{end}) & = & \mathsf{eatoms}(e) \cup \mathsf{eatoms}(m) \\
\mathsf{eatoms}(\texttt{spawn}(f,e)) & = & \mathsf{eatoms}(e) \\
\mathsf{eatoms}(e_1\texttt{!}e_2) & = & \mathsf{eatoms}(e_1) \cup \mathsf{eatoms}(e_2) \\
\mathsf{eatoms}(\texttt{receive}\ m\ \texttt{end}) & = & \mathsf{eatoms}(m) \\
\mathsf{eatoms}(e_1\texttt{,}e_2) & = & \mathsf{eatoms}(e_1) \cup \mathsf{eatoms}(e_2) \\
\mathsf{eatoms}(p_1\ \texttt{->}\ e_1\texttt{;}\ldots\texttt{;}p_n\ \texttt{->}\ e_n) & = & \bigcup_{1\le i\le n} (\mathsf{eatoms}(p_i) \cup \mathsf{eatoms}(e_i))
\end{array}
$$

Figure 4.3: Extraction of Atoms

We use a second constant $K \in \mathbb{N}$ for the maximal size of tuples. K will depend on the arity of the tuples occurring in the program, while k is a parameter for the granularity of the abstraction. Usually, the maximal number of tuple components will be fixed for the SOS of a program. K can be set to this maximal size. Larger values for K do not yield more precision in the abstraction. We interpret K as a global constant. Therefore, it is not added as another index to $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid)$.

Remarks

- For every program p is $\mathsf{patoms}(p)$ a finite set.
- $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid)$ is an infinite set for every $k \in \mathbb{N}$ but we will later restrict to a finite number of pids and so we only use a finite part of $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid)$. Neglecting the pids, the number of the constructor terms in $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\})$ is finite for every program p.
- $\preceq$ is a complete partial order on $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid)$ because only finite chains exist and **?** is the smallest element.

For simplification, we write $\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid)$ instead of $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid)$, if p is clear from the context.

The terms of $T_{\mathcal{C}}(Pid)$ are represented by terms in $\widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid)$ which do have the same top but the abstracted subparts are replaced by **?**. The abstraction function

$\alpha_k : T_{\mathcal{C}}(Pid) \longrightarrow \widehat{T}^k_{\widehat{\mathcal{C}}_p}(\{\mathbf{?}\} \cup Pid)$ yields the best representation of a term v. That means the top part of v restricted to depth k:

$$\begin{array}{lcl} \alpha_0(v) & = & \mathbf{?} \\ \alpha_{k+1}(a/0) & = & \begin{cases} a & \text{, if } a \in \mathsf{patoms}(p) \\ \mathbf{?} & \text{, otherwise} \end{cases} \\ \alpha_{k+1}(c(v_1, \ldots, v_n)) & = & c(\alpha_k(v_1), \ldots, \alpha_k(v_n)) \text{ for } n > 0 \end{array}$$

We will also need a depth-k-restriction on our abstract values. Hence, we define an extension of this abstraction function to abstract values:

$$\begin{array}{lcl} \widehat{\alpha}_k : T_{\mathcal{C}}(\{\mathbf{?}\} \cup Pid) & \longrightarrow & \widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid) \\ \widehat{\alpha}_0(v) & = & \mathbf{?} \\ \widehat{\alpha}_{k+1}(c(v_1, \ldots, v_n)) & = & c(\widehat{\alpha}_k(v_1), \ldots, \widehat{\alpha}_k(v_n)) \\ \widehat{\alpha}_{k+1}(\mathbf{?}) & = & \mathbf{?} \end{array}$$

Now we have defined the abstract domain and have described its connection to $T_{\mathcal{C}}(Pid)$. In the abstract interpretation $\widehat{\iota}_k$ of the predefined functions we have to consider values with **?**. Hence, the application of predefined functions to **?** can only yield **?** as well. Only for abstract values which are not abstracted (they also belong to $T_{\mathcal{C}}(Pid)$) the abstract interpretation of a predefined function yields a value which is more precise than **?**. The result is the largest representation in $\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid)$ of the standard interpretation.

$$\widehat{\iota}_k(F/n)(v_1, \ldots, v_n) = \begin{cases} \alpha_k(F_{\mathcal{A}}(v_1, \ldots, v_n)) & \text{, if } v_i \in T_{\mathcal{C}}(Pid)\ \forall 1 \leq i \leq n \\ \mathbf{?} & \text{, otherwise} \end{cases}$$

The application of a constructor yields the best representation of the result obtained with the free constructor interpretation. In the case that the free interpretation belongs to $\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid)$ we get the free interpretation.

$$\widehat{\iota}_k(\underline{c}/n)(v_1, \ldots, v_n) = \widehat{\alpha}_k(c(v_1, \ldots, v_n))$$

In the abstract interpretation of matching we have to consider the occurences of **?** as well. **?** can occur in the values as well as in the patterns because of the dynamic character of patterns in Erlang. We can define the abstract interpretation of match with structural induction. In the case of **?** our abstract interpretation does not fail because one of its concrete representations could be a value which matches. However, it yields only a possible value. In the case that **?** occurs in the abstract pattern the structural induction handles a dynamic part which has no variables. Hence, no variables must be bound and the abstract match yields the empty substitution. The other way round, if **?** occurs in the value, then all variables of the pattern must be substituted. Their only possible binding is **?**.

In the formalization we first define constants for the two kinds of evaluated substitutions.

$$\mathsf{Irref} = \mathsf{true} \quad \text{and} \quad \mathsf{Poss} = \mathsf{false}$$

The result of matching a pattern against a constructor term can either fail or, if it succeeds, be either Irref or Poss in dependence of the cases discussed above.

$$
\begin{array}{lcl}
\widehat{\iota_k}(\mathsf{match}) : \widehat{Pat} \times \widehat{A} & \longrightarrow & (\{\mathsf{Irref}, \mathsf{Poss}\} \times \mathit{Subst}(\widehat{A})) \cup \{\mathit{Fail}\} \\
\widehat{\iota_k}(\mathsf{match})(X, v) & = & (\mathsf{Irref}, [X/v]) \\
\widehat{\iota_k}(\mathsf{match})(\mathtt{?}, v) & = & (\mathsf{Poss}, []) \\
\widehat{\iota_k}(\mathsf{match})(c(p_1, \ldots, p_n), \mathtt{?}) & = & (\mathsf{Poss}, [\mathit{Var}(p_1, \ldots, p_n)/\mathtt{?}]) \\
\widehat{\iota_k}(\mathsf{match})(c(p_1, \ldots, p_n), c(v_1, \ldots, v_n)) & = & \biguplus'_{1 \leq i \leq n} \widehat{\iota_k}(\mathsf{match})(p_i, v_i) \\
\widehat{\iota_k}(\mathsf{match})'(e, e') & = & \mathsf{Fail}, \text{ otherwise}
\end{array}
$$

As already mentioned in Section 3.2 we are allowed to use non-linear patterns in Erlang. This means that a variable may occur more than ones in a pattern. The pattern matching only succeeds if these variables are bound to the same expression in every position they occur. Hence, the pattern matching `{X,X}`=**{3,4}** would fail and `{X,X}`=**{3,3}** would succeed with the substitution [`X`/**3**]. This leads to a problem in the abstraction because it could happen that we have abstracted from one of both values but the other one is still known. Then `{X,X}`=**{3,?}** could match with the substitution [`X`/**3**] but this is not irrefutable. We get the substitution (Poss, [`X`/**3**]) although the subterms yield irrefutable matches. Even if both values are not known (`{X,X}`=**{?,?}**) we only get the match (Poss, [`X`/**?**]). It is also possible that two terms a variable is matched with are incomparable but a more precise term exists which can match both. An example is the matching `{X,X}`=**{{3,?},{?,3}}** which can match if both times **?** is the abstraction of the value **3**. So a matching has to yield the substitution (Poss,[`X`/**{3,3}**]). We get the more precise term **{3,3}** as **{?,3}** $\sqcup$ **{3,?}**.

In the standard interpretation we use the function $\uplus$ for the union of substitutions. To model the abstract interpretation discussed in the last paragraph, we use the modified union $\uplus'$ (see Figure 4.4) in the definition of $\widehat{\iota_k}(\mathsf{match})$. For an empty set of substitutions (tagged with Irref/Poss) $\biguplus'$ is defined as (Irref, []). $\uplus'$ applied to two substitutions respects the partial order $\preceq$ on $\widehat{T_{\mathcal{C}}}(\{\mathtt{?}\} \cup \mathit{Pid})$. The result depends on that parts of the substitutions that overlap (in the definition this is the part of the substitution on the variables $Z_1, \ldots, Z_k$). Here, the result is the least upper bound of the overlapping bindings. The rests of the substitutions ($[\overline{X/v}] := [X_1/v_1, \ldots, X_i/v_i]$ and $[\overline{Y/w}] := [Y_1/v_1, \ldots, Y_j/v_j]$) are the non-overlapping parts. They are just copied to the joined substitution.

The abstract interpretation of casematch and mbmatch yields all possible matches of a pattern with a value. It is not sufficient to consider the first match as in the concrete semantics because the first match could only be possible in the abstraction

$$\uplus' : (\{\mathsf{Irref}, \mathsf{Poss}\} \times \mathit{Subst}(\widehat{A})) \cup \{\mathit{Fail}\} \longrightarrow (\{\mathsf{Irref}, \mathsf{Poss}\} \times \mathit{Subst}(\widehat{A})) \cup \{\mathit{Fail}\}$$

$$(f_1, \sigma_1) \uplus' (f_2, \sigma_2) =$$

$$\begin{cases} (f_1 \wedge f_2, [\overline{X/v}, \overline{Y/w}, Z_1/u_1, \ldots, Z_k/u_k]), & \text{if } u_l = u'_l \in T_{\mathcal{C}}(\mathit{Pid})\ \forall l \leq k \\ (\mathsf{Poss}, [\overline{X/v}, \overline{Y/w}, Z_1/\widehat{u}_1, \ldots, Z_k/\widehat{u}_k]) & \text{, if } \widehat{u}_l = u_l \sqcup u'_l \text{ exists } \forall l \leq k \\ \mathsf{Fail} & \text{, otherwise} \end{cases}$$

$$\text{where} \quad \begin{array}{l} \{Z_1, \ldots, Z_k\} = \mathcal{D}om(\sigma_1) \cap \mathcal{D}om(\sigma_2) \\ [\overline{X/v}, Z_1/u_1, \ldots, Z_k/u_k] = \sigma_1 \\ [\overline{Y/w}, Z_1/u'_1, \ldots, Z_k/u'_k] = \sigma_2 \end{array}$$

$$\mathsf{Fail} \uplus' \_ = \mathsf{Fail}$$
$$\_ \uplus' \mathsf{Fail} = \mathsf{Fail}$$

Figure 4.4: Union of Abstract Substitutions

but fail in a concretization.

$$\widehat{\iota}_k(\mathsf{casematch})((\widehat{p}_1, \ldots, \widehat{p}_n), \widehat{v}) = \{(i, \sigma) \mid \widehat{\iota}_k(\mathsf{match})(\widehat{p}_i, \widehat{v}) = (\_, \sigma)\} \cup \begin{cases} \{\mathsf{Fail}\} & \text{, if } \widehat{\iota}_k(\mathsf{match})(\widehat{p}_i, \widehat{v}) \neq (\mathsf{Irref}, \_)\ \ \forall 1 \leq i \leq n \\ \emptyset & \text{, otherwise} \end{cases}$$

$$\widehat{\iota}_k(\mathsf{mbmatch})((\widehat{p}_1, \ldots, \widehat{p}_n), (\widehat{v}_1, \ldots, \widehat{v}_m) = \{(i, j, \sigma) \mid \widehat{\iota}_k(\mathsf{match})(\widehat{p}_i, \widehat{v}_j) = (\_, \sigma)\} \cup \begin{cases} \{\mathsf{Fail}\} & \text{, if } \widehat{\iota}_k(\mathsf{match})(\widehat{p}_i, \widehat{v}_j) \neq (\mathsf{Irref}, \_)\ \ \forall 1 \leq i \leq n, 1 \leq j \leq m \\ \emptyset & \text{, otherwise} \end{cases}$$

A Pid is interpreted as itself. **?** can represent any value, including all pids. Other values do not have an interpretation as a pid:

$$\widehat{\iota}_k(\text{pid})(v) = \begin{cases} \{v\} & \text{, if } v \in \mathit{Pid} \\ \mathit{Pid} \cup \{\mathsf{err}\} & \text{, if } v = \mathbf{?} \\ \{\mathsf{err}\} & \text{, otherwise} \end{cases}$$

We will use this abstract interpretation for the formal verification of the database example. Hence, we must prove that the abstract interpretations $\widehat{\mathcal{A}}_k$ fulfills the properties (P1) – (P5).

Lemma 4.17 ($\widehat{\mathcal{A}}_k$ satisfies (P1))
The abstract interpretation $\widehat{\mathcal{A}}_k = (\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup \mathit{Pid}), \widehat{\iota}_k, \preceq, \alpha_k)$ *satisfies property* (P1) for all $k \in \mathbb{N}$:

For all $\phi/n \in \mathcal{F} \cup \underline{\mathcal{C}}, v_1, \ldots, v_n \in T_{\mathcal{C}}(\mathit{Pid})$ and $\widetilde{v}_i \preceq \alpha(v_i)$ it holds that $\phi_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n) \preceq \alpha(\phi_{\mathcal{A}}(v_1, \ldots, v_n))$.

Proof:

- $\phi = F/n \in \mathcal{F}$.
  We distinguish two cases:
  - $\widetilde{v}_1, \ldots, \widetilde{v}_n \in T_{\mathcal{C}}(Pid)$:
    This means $\alpha_k(v_i) = v_i$ for all $1 \leq i \leq n$.
    $F_{\widehat{\mathcal{A}}_k}(\widetilde{v}_1, \ldots, \widetilde{v}_n) = \alpha_k(F_{\mathcal{A}}(\widetilde{v}_1, \ldots, \widetilde{v}_n)) = \alpha_k(F_{\mathcal{A}}(v_1, \ldots, v_n))$
  - At least one $v_i \notin T_{\mathcal{C}}(Pid)$:
    Hence, $F_{\widehat{\mathcal{A}}_k}(\widetilde{v}_1, \ldots, \widetilde{v}_n) = \mathbf{?}$ and trivially $\mathbf{?} \preceq \alpha_k(F_{\mathcal{A}}(v_1, \ldots, v_n))$
- $\phi = \underline{c}/n \in \mathcal{C}$.
  (P1) is trivially satisfied because it is the definition of $\widehat{\iota}_k(\underline{c}/n)$. □

Before we can prove that $\widehat{\mathcal{A}}_k$ satisfies (P2), we have to show corresponding properties for the functions $\sqcup$ and $\uplus'$ which are used in the definition of match′.

Lemma 4.18
Let $v \in T_{\mathcal{C}}(Pid)$. For all $\widetilde{v}_1, \widetilde{v}_2 \in \widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid)$ with $\widetilde{v}_1 \preceq \alpha_k(v)$ and $\widetilde{v}_2 \preceq \alpha_k(v)$ it holds that $\widetilde{v} = \widetilde{v}_1 \sqcup \widetilde{v}_2$ exists and $\widetilde{v} \preceq \alpha_k(v)$.

Proof: We prove this fact by a combined induction on the structure of $\widetilde{v}_1$ and k.

- $\widetilde{v}_1 = \mathbf{?}$:
  Then by Definition 4.16: $\widetilde{v}_1 \sqcup \widetilde{v}_2 = \widetilde{v}_2 \preceq \alpha_k(v)$.
- $\widetilde{v}_1 = c(\widetilde{w}_1, \ldots, \widetilde{w}_n)$:
  Then $v = c(w_1, \ldots, w_n)$ with $\widetilde{w}_i \preceq \alpha_{k-1}(w_i)$ $\forall 1 \leq i \leq n$ and we have two different cases for $\widetilde{v}_2$:
  - $\widetilde{v}_2 = \mathbf{?}$:
    Analogously as $\widetilde{v}_1 = \mathbf{?}$.
  - $\widetilde{v}_2 = c(\widetilde{w}'_1, \ldots, \widetilde{w}'_n)$ with $\widetilde{w}'_i \preceq \alpha_{k-1}(w_i)$ $\forall 1 \leq i \leq n$:
    By induction hypothesis we know that $\widetilde{w}_i \sqcup \widetilde{w}'_i \preceq \alpha_{k-1}(w_i)$ exists $\forall 1 \leq i \leq n$.
    Hence, $\widetilde{v}_1 \sqcup \widetilde{v}_2 = c(\widetilde{w}_1 \sqcup \widetilde{w}'_1, \ldots, \widetilde{w}_n \sqcup \widetilde{w}'_n) = \widetilde{v}$ exists and $\widetilde{v} \preceq \alpha_k(v)$. □

Lemma 4.19
Let $\sigma_1, \sigma_2 \in Subst(T_{\mathcal{C}}(Pid))$ with $\sigma_1 \uplus \sigma_2 = \sigma \neq \mathsf{Fail}$. For all $\widetilde{\sigma}_1 \preceq \alpha_k(\sigma_1), \widetilde{\sigma}_2 \preceq \alpha_k(\sigma_2)$ and all $f_1, f_2 \in \{\mathsf{Irref}, \mathsf{Poss}\}$ it holds that $(f_1, \widetilde{\sigma}_1) \uplus' (f_2, \widetilde{\sigma}_2) = (\_, \widetilde{\sigma}) \neq \mathsf{Fail}$ with $\widetilde{\sigma} \preceq \alpha_k(\sigma)$.

Proof: Follows directly from Lemma 4.18 and the fact that for all $i \in \{1, 2\}$:
$\mathcal{D}om(\widetilde{\sigma}_i) = \mathcal{D}om(\sigma_i)$ and $\widetilde{\sigma}_i(X) \preceq \alpha_k(\sigma_i)(X)$ $\forall X \in \mathcal{D}om(\sigma_i)$. □

Lemma 4.20
Let $\sigma_1, \sigma_2 \in Subst(T_{\mathcal{C}}(Pid))$ with $\sigma_1 \uplus \sigma_2 = \mathsf{Fail}$. For all $\widetilde{\sigma}_1 \preceq \alpha_k(\sigma_1), \widetilde{\sigma}_2 \preceq \alpha_k(\sigma_2)$ and all $f_1, f_2 \in \{\mathsf{Irref}, \mathsf{Poss}\}$ it holds that $(f_1, \widetilde{\sigma}_1) \uplus' (f_2, \widetilde{\sigma}_2) \in \{(\mathsf{Poss}, \_), \mathsf{Fail}\}$.

Proof: $(f_1, \widetilde{\sigma}_1) \uplus' (f_2, \widetilde{\sigma}_2) = (\mathsf{Irref}, \widetilde{\sigma})$ iff $f_1 = f_2 = \mathsf{Irref}$, the overlapping parts of $\widetilde{\sigma}_1$ and $\widetilde{\sigma}_2$ are identical and belong to $T_{\mathcal{C}}(Pid)$. In this case each of them represent only one single value which is identical to its abstract value. Then $\sigma_1 \uplus \sigma_2 \neq \mathsf{Fail}$ which is a contradiction to the assumption. □

Lemma 4.21 ($\widehat{\mathcal{A}}_k$ satisfies (P2))
The abstract interpretation $\widehat{\mathcal{A}}_k = (\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid), \widehat{\iota}_k, \preceq, \alpha_k)$ *satisfies property* (P2) for all $k \in \mathbb{N}$:

(P2) For all $p \in Pat$, $v \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p} \sqsubseteq \alpha(p)$, $\widetilde{v} \sqsubseteq \alpha(v)$ it holds that

a) if $\mathsf{match}(p, v) = \sigma$
then there exists $\widetilde{\sigma} \sqsubseteq \alpha(\sigma)$ with $\mathsf{match}_{\widehat{\mathcal{A}}}(\widetilde{p}, \widetilde{v}) = (\_, \widetilde{\sigma})$.

b) if $\mathsf{match}(p, v) = \mathsf{Fail}$
then $\mathsf{match}_{\widehat{\mathcal{A}}}(\widetilde{p}, \widetilde{v}) \in \{(\mathsf{Poss}, \_), \mathsf{Fail}\}$.

Proof:

a) Let $\mathsf{match}(p, v) = \sigma$. We prove the property by induction on the structure of p:

- $p = X$:
  Hence, $\widetilde{p} = X$ and $\widetilde{\sigma} = [X/\widetilde{v}] \preceq \alpha_k(\sigma) = [X/v]$.
- $p = c(p_1, \ldots, p_n)$:
  Hence, $\widetilde{p} = \mathbf{?}$ or $\widetilde{p} = c(\widetilde{p}_1, \ldots, \widetilde{p}_n)$:
  - $\widetilde{p} = \mathbf{?}$
    Then p does not contain any variables and σ must be the empty substitution. On the other hand, $\mathsf{match}_{\widehat{\mathcal{A}}_k}(\widetilde{p}, \widetilde{v}) = (\mathsf{Poss}, [])$.
  - $\widetilde{p} = c(\widetilde{p}_1, \ldots, \widetilde{p}_n)$ with $\widetilde{p}_i \preceq \alpha_k(p_i)$ $\forall 1 \leq i \leq n$:
    By induction hypothesis we know that $\mathsf{match}(p, v) = \sigma_i$ and $\mathsf{match}_{\widehat{\mathcal{A}}_k}(\widetilde{p}_i, \widetilde{v}) = (\_, \widetilde{\sigma}_i)$ with $\widetilde{\sigma}_i \preceq \alpha_k(\sigma_i)$.
    From the definitions of match and $\mathsf{match}_{\widehat{\mathcal{A}}_k}$ we know that $\sigma = \biguplus_{1 \leq i \leq n} \sigma_i$ and $\widetilde{\sigma} = \biguplus'_{1 \leq i \leq n} \widetilde{\sigma}_i$. Hence, with Lemma 4.19 it holds that $\widetilde{\sigma} \preceq_k (\sigma)$.

b) Let $\mathsf{match}(p, v) = \mathsf{Fail}$. Again, we prove the property by induction on p:

- $p = X$:
  Hence, $\mathsf{match}(p, v) = [X/v] \neq \mathsf{Fail}$.
- $p = c(p_1, \ldots, p_n)$:
  Hence, $\widetilde{p} = \mathbf{?}$ or $\widetilde{p} = c(\widetilde{p}_1, \ldots, \widetilde{p}_n)$:
  - $\widetilde{p} = \mathbf{?}$:
    Hence, $\mathsf{match}_{\widehat{\mathcal{A}}_k}(\widetilde{p}, \widetilde{v}) = (\mathsf{Poss}, [])$ and the lemma is satisfied.
  - $\widetilde{p} = c(\widetilde{p}_1, \ldots, \widetilde{p}_n)$ with $\widetilde{p}_i \preceq \alpha_k(p_i)$ $\forall 1 \leq i \leq n$:
    There are two reasons, why $\mathsf{match}(p, v) = \mathsf{Fail}$:
    * $v = d(v_1, \ldots, v_m)$ Hence, $\widetilde{v} = \mathbf{?}$ or $\widetilde{v} = d(\widetilde{v}_1, \ldots, \widetilde{v}_m)$ with $\widetilde{v}_i \preceq \alpha_k(v_i)$ and we have $\mathsf{match}(\widetilde{p}, \widetilde{v}) = (\mathsf{Poss}, [Vars(\widetilde{p}_1, \ldots, \widetilde{p}_n)/\mathbf{?}])$ respectively $\mathsf{match}(\widetilde{p}, \widetilde{v}) = \mathsf{Fail}$.

∗ $v = c(v_1, \ldots, v_n)$ Then there exists $1 \leq i \leq n$ with $\mathsf{match}(p_i, v_i) = \mathsf{Fail}$. In the abstraction we have $\widetilde{v} = \mathbf{?}$ or $\widetilde{v} = c(\widetilde{v}_1, \ldots, \widetilde{v}_n)$ with $\widetilde{v}_i \preceq \alpha_k(v_i)$. The first case is the same as before. In the second case we know by induction hypothesis that $\mathsf{match}_{\widehat{\mathcal{A}}_k}(\widetilde{p}_i, \widetilde{v}_i) \in \{(\mathsf{Poss}, \_), \mathsf{Fail}\}$.
With Lemma 4.20 we conclude: $\bigsqcup'_{1 \leq j \leq n} \widetilde{\sigma}_j \in \{(\mathsf{Poss}, \_), \mathsf{Fail}\}$. □

Lemma 4.22 ($\widehat{\mathcal{A}}_k$ satisfies (P3))
The abstract interpretation $\widehat{\mathcal{A}}_k = (\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid), \widehat{\iota}_k, \preceq, \alpha_k)$ *satisfies property* (P3) for all $k \in \mathbb{N}$:

(P3) For all $p_1, \ldots, p_n \in Pat, v \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p}_j \sqsubseteq \alpha(p_j)$ and $\widetilde{v} \sqsubseteq \alpha(v)$ it holds that:

a) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = (i, \rho)$,
then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$ with $(i, \widetilde{\rho}) \in \mathrm{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{v})$.
b) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = \mathsf{Fail}$,
then $\mathsf{Fail} \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{v})$.

Proof: This is trivial because casematch yields the union of all possible matches. The successful match of the concrete semantics must be included. Furthermore, we add Fail, if one of the matchings is not irrefutable. □

Lemma 4.23 ($\widehat{\mathcal{A}}_k$ satisfies (P4))
The abstract interpretation $\widehat{\mathcal{A}}_k = (\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid), \widehat{\iota}_k, \preceq, \alpha_k)$ *satisfies property* (P4):

(P4) For all $p_1, \ldots, p_n \in Pat, v_1, \ldots, v_u \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p}_j \sqsubseteq \alpha(p_j)$ and $\widetilde{v}_k \sqsubseteq \alpha(v_k)$ it holds that:

a) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = (i, j, \rho)$,
then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$
with $(i, j, \widetilde{\rho}) \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), (\widetilde{v}_1, \ldots, \widetilde{v}_u))$.
b) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = \mathsf{Fail}$,
then $\mathsf{Fail} \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p}_1, \ldots, \widetilde{p}_n), (\widetilde{v}_1, \ldots, \widetilde{v}_u))$.

Proof: The proof is as trivial as for (P3). □

Lemma 4.24 ($\widehat{\mathcal{A}}_k$ satisfies (P5))
The abstract interpretation $\widehat{\mathcal{A}}_k = (\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid), \widehat{\iota}_k, \preceq, \alpha_k)$ *satisfies property* (P5):

a) For all $\pi \in Pid$, $\widetilde{p} \sqsubseteq \alpha(p)$ it holds that $\pi \in \mathrm{pid}_{\widehat{\mathcal{A}}}(\widetilde{p})$.
b) For all $v \notin Pid$, $\widetilde{v} \sqsubseteq \alpha(v)$ it holds that $\mathsf{err} \in \mathrm{pid}_{\widehat{\mathcal{A}}}(\widetilde{v})$.

Proof: The proof is a trivial case analysis. □

Theorem 4.25 (Safeness of $\widehat{\mathcal{A}}_k$)
The abstract interpretation $\widehat{\mathcal{A}}_k = (\widehat{T}^k_{\widehat{\mathcal{C}}}(\{\mathbf{?}\} \cup Pid), \widehat{\iota}_k, \preceq, \alpha_k)$ *satisfies properties* (P1) – (P5) *for all* $k \in \mathbb{N}$.

Compared to [NN97] our approach is more general because we are not restricted to one abstract interpretation, like types. If special values are relevant for the verification we can also consider these values and add them to our abstract domain. The behavior analysis in[NN97] is fixed to type inference and cannot easily be extended to other abstract domains.

4.4 Galois Insertions

As already mentioned in the motivation of our framework, our approach differs from the approach taken by Cousot and Cousot in [CC77a, CC77b]. They introduce the notion of a *Galois insertion* which is a special case of an adjunction in the sense of category theory. In their framework an abstract interpretation consists of a pair α, γ of abstraction and concretization functions with types

$$\begin{aligned} \alpha: &\quad Conc \longrightarrow \widehat{A} \\ \gamma: &\quad \widehat{A} \longrightarrow Conc \end{aligned}$$

where *Conc* is the power set of all concrete values. In our context this would be $Conc = \mathcal{P}(T_{\mathcal{C}}(Pid))$. This domain induces the subset relation as a complete lattice on *Conc*. In contrast to our framework the Cousot framework additionally requires a complete lattice on $\widehat{A}$ (instead of a partial order). Furthermore, Cousot and Cousot impose the following conditions for a Galois insertion:

1. α and γ are monotonic
2. $\forall a \in \widehat{A}: \; a = \alpha(\gamma(a))$
3. $\forall c \in Conc: \; c \subseteq \gamma(\alpha(c))$

Then property (P1) of our framework could be modified to:

(P1*) For all $f/n \in \Sigma, V_1, \ldots, V_n \in \mathcal{P}(TC(Pid))$ it holds that

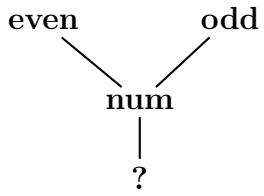
$$f_{\widehat{\mathcal{A}}}(\alpha(V_1), \ldots, \alpha(V_n)) \sqsubseteq \alpha(\{f_{\mathcal{A}}(v_1, \ldots, v_n) \mid v_i \in V_i, 1 \leq i \leq n\})$$

Similar modifications can be defined for the remaining properties. Alternatively, it is also possible to define equivalent properties using the concretization function.

Actually, the properties in the Cousot framework are not simpler. For instance, in property (P1*) we do not have to prove the connection between abstract and concrete

interpretation for all $\widetilde{v}_i \sqsubseteq \alpha(v_i)$. Instead, we have to prove the connection for arbitrary sets of values. This is at least as expensive as the prove of (P1). The same holds for the corresponding definition of the properties using the concretization function γ.

Furthermore, in the Cousot framework we would have to show the properties 1–3 of the Galois insertion which is an additional effort. Therefore, we think that our approach without the definition of a Galois insertion is simpler to use in practice. Finally, we think that it is more natural to define an abstraction function $\alpha : A \longrightarrow \widehat{A}$ instead of $\alpha : \mathcal{P}(A) \longrightarrow \widehat{A}$. In the design of the abstract domain it is clearer to define the abstract representation of a concrete value, instead of arbitrary sets of concrete values. As an example we once again consider a variation of the even-odd domain:



In our framework we can define

$$\alpha(v) = \begin{cases} \textbf{even} & \text{, if } \exists n \in \mathbb{Z} : v = 2 * n \\ \textbf{odd} & \text{, if } \exists n \in \mathbb{Z} : v = 2 * n + 1 \\ \textbf{?} & \text{, otherwise} \end{cases}$$

In the Cousot framework we would get:

$$\alpha(V) = \begin{cases} \textbf{even} & \text{, if } \forall v \in V\ \exists n \in \mathbb{Z} : v = 2 * n \\ \textbf{odd} & \text{, if } \forall v \in V\ \exists n \in \mathbb{Z} : v = 2 * n + 1 \\ \textbf{num} & \text{, otherwise and if } V \subseteq \mathbb{Z} \\ \textbf{?} & \text{, otherwise} \end{cases}$$

In our framework the third case can be omitted because all integers are either even or odd. There exists no value for which the best abstract representation yields **num**. A Galois connection must contain this case. In larger abstractions this can be even more difficult and much more cases must be considered.

The framework in the Cousot style is more pleasant from the theoretical point of view. For example, it can be shown that the definition of α can be reduced to the definition of γ and vice versa. Since we do not need these results, we have decided to define the framework in the presented style.

4.5 Finiteness of Abstract Semantics

In this section we want to characterize a subset of Core Erlang programs, for which the abstract operational semantics for finite domain abstractions yields finite transition system.

The abstract semantics $\longrightarrow_{\widehat{\mathcal{A}}}$ is defined for arbitrary abstract states. For every abstraction $\widehat{\mathcal{A}}$ the set of all states $\widehat{State}_{\widehat{\mathcal{A}}}$ is an infinite set and also the relation $\longrightarrow_{\widehat{\mathcal{A}}}$

defines an infinite transition system. For the verification, we only consider the part of this transition system which is reachable from the initial state (@**0**,`main()`,()).

In the formal semantics we used renaming of variables, to avoid name conflicts. This leads to the problem that variables can be renamed arbitrarily in the application of a function. We define an equivalence relation for the renaming of variables. Thereby, the transition system can be constructed with respect to the corresponding equivalence classes.

4.5.1 Renaming of variables

In the construction of the AOS we want to detect cycles, to get a finite transition system. Therefore, we need to formalize when two states are equivalent. Inspecting the rules for the AOS, we see that in the rule for the application of a defined function variables are renamed. It is not sufficient to define two states as equivalent if they are syntactically identical. States are also equivalent if they are identical except for the names of the free variables. This is meaningful since equivalent states can perform the same actions and also their successors are identical, except for the names of the occurring variables. For example, the two Core Erlang expressions

`Y=1,X=3+Y,X+5` and `X=1,Y=3+X,Y+5` should be considered as equivalent.

On the other hand, neither `Y=1,Y=3+Y,Y+5` nor `X=1,Y=3+X,X+5` are equivalent to one of them. In the λ-calculus there exists a similar equivalence, called α-equivalence in which only bound variables can be renamed. Erlang has no scoping and bound variables do not exist. We define Core Erlang expressions as equivalent if they only differ in the names of the variables. However, we also call it α-equivalence.

Defintion 4.26 (Renaming)
A mapping $\rho : \mathit{Var} \longrightarrow \mathit{Var}$ is called a *renaming*. $\lhd$

Two Core Erlang expressions are equivalent, if both of them can be converted into the other by renaming.

Defintion 4.27 (α-Equivalence)
Two Core Erlang expressions $e_1, e_2 \in \mathcal{E}(S)$ are *α-Equivalent* ($\sim_\alpha$) iff there exist two renamings $\rho_1, \rho_2 : \mathit{Var} \longrightarrow \mathit{Var}$ with

$$\rho_1(e_1) = e_2 \quad \text{and} \quad e_1 = \rho_2(e_2). \qquad \lhd$$

It is not sufficient, to require only a renaming in one direction. For instance, we would get:

$$[\texttt{X/Y,Y/Y}](\texttt{Y=1,X=3+Y,X+5}) = \texttt{Y=1,Y=3+Y,Y+5}$$

which are not equivalent. The reverse renaming is not possible.

Remark 4.28
It can easily be shown that $\sim_\alpha$ is an equivalence relation.

The definition of $\sim_\alpha$ is very simple but for the use in the AOS we must be able to decide if two Core Erlang expressions are equivalent or not. The definition of $\sim_\alpha$ is not constructive. However, we can define an algorithm which decides $\sim_\alpha$. We use a sequence of variables which do not occur in both Core Erlang expressions. Then the variables in the two expressions are successively renamed to the variables from this sequence. Thereafter, both expressions must be equal. This algorithm is obviously correct.

4.5.2 Hierarchical Core Erlang

For verification we can now construct a transition system over the α-equivalence classes of the states. Furthermore, we restrict to that part of the semantics which is reachable from the initial state. For simplicity we will omit the α-equivalence and the restriction to the spawned transition system in the following. If we use the terms "finite operational semantics" or "finite transition system resulting from the AOS", then we mean the system of equivalence classes reachable from the initial state.

Unfortunately, also this transition system can be infinite for finite domain abstractions, as the following example shows:

```
main() -> f(0).
f(X) -> f(f(X)).
```

The smallest possible abstract domain only contains the element **?** which represents all possible values. Using this abstract domain the abstract semantics of the program only contains the infinite path:

$$(@\mathbf{0}, \texttt{main()}, ()) \longrightarrow (@\mathbf{0}, \texttt{f(42)}, ()) \longrightarrow (@\mathbf{0}, \texttt{f(?)}, ())$$
$$\longrightarrow (@\mathbf{0}, \texttt{f(f(?))}, ()) \longrightarrow \ldots \longrightarrow (@\mathbf{0}, \texttt{f}^{\texttt{n}}\texttt{(?)}, ())$$
$$\longrightarrow (@\mathbf{0}, \texttt{f}^{\texttt{n+1}}\texttt{(?)}, ()) \ldots$$

which contains infinitely many different states. The proplem is the non-tail recursive call `f(X)`.

We want to find a large subclass of Core Erlang programs, for which it is guaranteed that the resulting transition system is finite. First, we can think of Core Erlang programs in which calls are only allowed in tail positions. Here the size of the expressions in the processes is limited by the sizes of the right-hand sides of the program.

As already mentioned at the beginning of this chapter, there are two other reasons why the AOS can yield an infinite transition system. An arbitrary number of processes can be spawned and mailboxes of processes can grow to arbitrary size. We have not abstracted from these aspects of a system state. We think that here an abstraction will damage too much of the behavior of the system. For example, an abstraction of an arbitrary number of messages to a finite number can never yield the information that a mailbox is empty. This kind of abstraction would add too much nondeterminism to the AOS and interesting properties cannot be proven. Another point is the abstraction of an arbitrary number of processes. Again, one process would have to behave like several processes and an abstraction would in general be much too coarse.

Under the assumptions that only finitely many processes are spawned and the processes only use restricted parts of their mailboxes the AOS of Core Erlang with only tail calls is finite state for finite domain abstract interpretations. In general, these assumptions are not decidable. Already for CCS it is undecidable, if an arbitrary number of processes is spawned [Tau89].

In the implementation of the AOS we handle this by restricting them to fixed sizes and warnings when exceeding these restrictions. An overflow of the mailboxes often happens on unfair scheduled paths and the warning shows a problem in the system design which can be a hint to mistakes in the program. E.g., if a process sends messages much faster than another process can receive them, then its mailbox can overflow. This can especially happen when distributing processes on different architectures. Especially for overflows of mailboxes the restriction gives a valuable hint to the programmer. Synchronization messages can be added which guarantee that only finite parts of the mailboxes are used. Then the AOS should yield a finite transition system.

From the practical point of view, the class of Core Erlang programs in which function calls are only allowed in tail positions is very restricted. Sub-evaluations as the call of `lookup` in p_{db} are not possible. On the other hand, many programs performing sub-evaluations also yield finite state systems (e.g., p_{db}) and sub-evaluations are needed to achieve convenient programming. An elegant description of these programs can be given by the notion of hierarchical programs. Therefore, we first define the function $\mathsf{calls} : \mathcal{E}(\emptyset) \to \mathcal{P}(FS(p))$ which yields the set of functions which can be called during the evaluation of a Core Erlang expression $e \in \mathcal{E}(\emptyset)$.

Defintion 4.29 (Function Calls)
Let p be a core Erlang program and $e \in \mathcal{E}(\emptyset)$. $\mathsf{calls}(e)$ is defined as the smallest set over $FS(p)$ which satisfies the following equations:

$$
\begin{array}{lcl}
\mathsf{calls}(f(e_1, \ldots, e_n)) & = & \bigcup_{i=1}^{n} \mathsf{calls}(e_i) \cup \{f\} \\
\mathsf{calls}(\phi(e_1, \ldots, e_n)) & = & \bigcup_{i=1}^{n} \mathsf{calls}(e_i) \\
\mathsf{calls}(X) & = & \emptyset \\
\mathsf{calls}(X\texttt{=}e) & = & \mathsf{calls}(e) \\
\mathsf{calls}(\texttt{self}) & = & \emptyset \\
\mathsf{calls}(e_1\texttt{,}e_2) & = & \mathsf{calls}(e_1) \cup \mathsf{calls}(e_2) \\
\mathsf{calls}(\texttt{case}\ e\ \texttt{of}\ m\ \texttt{end}) & = & \mathsf{calls}(e) \cup \mathsf{calls}(m) \\
\mathsf{calls}(e_1\texttt{!}e_2) & = & \mathsf{calls}(e_1) \cup \mathsf{calls}(e_2) \\
\mathsf{calls}(\texttt{receive}\ m\ \texttt{end}) & = & \mathsf{calls}(m) \\
\mathsf{calls}(\texttt{spawn}(f, e)) & = & \mathsf{calls}(e) \\
\mathsf{calls}(p_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}e_n) & = & \bigcup_{i=1}^{n} \mathsf{calls}(e_i)
\end{array}
$$

We extend the function calls to the defined functions:

$$
\begin{array}{l}
\mathsf{calls} : FS(p) \longrightarrow \mathcal{P}(FS(p)) \\
\mathsf{calls}(f) = \mathsf{calls}(e_f) \text{ for } f(X_1, \ldots, X_n)\texttt{->}e_f\texttt{.} \text{ in } p.
\end{array}
$$

◁

This definition of the function calls cannot directly be used as a (functional) implementation. For recursive programs, the recursive calls of the function calls will not terminate. However, the smallest set of functions fulfilling the equations for calls can be computed by a fixed-point iteration or an accumulation of already called functions. The same holds for the function innercalls which only yields the possible non-tail recursive calls of an expression.

Defintion 4.30 (Inner Function Calls)
Let p be a core Erlang program and $e \in \mathcal{E}(\emptyset)$. $\mathsf{innercalls}(e)$ is defined as the smallest set over $FS(p)$ which satisfies the following equations:

$$
\begin{array}{lcl}
\mathsf{innercalls}(f(e_1, \ldots, e_n)) & = & \bigcup_{i=1}^{n} \mathsf{calls}(e_i) \\
\mathsf{innercalls}(\phi(e_1, \ldots, e_n)) & = & \bigcup_{i=1}^{n} \mathsf{calls}(e_i) \\
\mathsf{innercalls}(X) & = & \emptyset \\
\mathsf{innercalls}(X\texttt{=}e) & = & \mathsf{innercalls}(e) \\
\mathsf{innercalls}(\texttt{self}) & = & \emptyset \\
\mathsf{innercalls}(e_1\texttt{,}e_2) & = & \mathsf{calls}(e_1) \cup \mathsf{innercalls}(e_2) \\
\mathsf{innercalls}(\texttt{case } e \texttt{ of } m \texttt{ end}) & = & \mathsf{calls}(e) \cup \mathsf{innercalls}(m) \\
\mathsf{innercalls}(e_1\texttt{!}e_2) & = & \mathsf{calls}(e_1) \cup \mathsf{calls}(e_2) \\
\mathsf{innercalls}(\texttt{receive } m \texttt{ end}) & = & \mathsf{innercalls}(m) \\
\mathsf{innercalls}(\texttt{spawn}(f, e)) & = & \mathsf{calls}(e) \\
\mathsf{innercalls}(p_1\texttt{->}e_1\texttt{;} \ldots \texttt{;}p_n\texttt{->}e_n) & = & \bigcup_{i=1}^{n} \mathsf{innercalls}(e_i)
\end{array}
$$

We also extend the function innercalls to the defined functions:

$$
\begin{array}{l}
\mathsf{innercalls} : FS(p) \longrightarrow \mathcal{P}(FS(p)) \\
\mathsf{innercalls}(f) = \mathsf{innercalls}(e_f) \text{ for } f(X_1, \ldots, X_n)\texttt{->}e_f\texttt{.} \text{ in } p.
\end{array}
$$
◁

The main difference to calls is that we do not add the function f to the set of innercalls in the first rule. In the other rules we pick up all calls by the function calls except the ones which are calls in tail positions.

Remark More than one function can be called in tail positions because of branching.

Lemma 4.31
Let p be a Core Erlang program. For all $e \in \mathcal{E}(\emptyset)$ holds

$$\mathsf{innercalls}(e) \subseteq \mathsf{calls}(e)$$

Proof: By simple induction on the structure of e. □

Defintion 4.32 (Hierarchical Core Erlang Programs)
A Core Erlang program p is called *hierarchically* iff $f \notin \mathsf{innercalls}(f)$ for all defined functions $f \in FS(p)$. ◁

For this restricted class we get the following finiteness results:

Lemma 4.33 (Finiteness of Hierarchical Core Erlang Programs)
Let p be a hierarchical Core Erlang Program and $\widehat{\mathcal{A}}$ a finite domain abstract interpretation. For the set of all Core Erlang expressions in the AOS holds

$$|\{[e]_\alpha \mid (@\mathbf{0}, \texttt{main()}, ()) \longrightarrow_{\widehat{\mathcal{A}}}^{*} \Pi, (\pi, e, q)\}| < \infty$$

Theorem 4.34 (Finiteness of Hierarchical Core Erlang Programs)
Let p be a hierarchical Core Erlang program and $\widehat{\mathcal{A}}$ a finite domain abstract interpretation. If p spawns only a finite number of processes in its abstract operational semantics and p uses only finite parts of the mailboxes:

$$\begin{aligned} \exists k \in \mathbb{N}: \quad & \forall s \text{ with } (@\mathbf{0}, \texttt{main()}, ()) \longrightarrow_{\widehat{\mathcal{A}}}^{*} s : |s| \leq k \text{ and} \\ & \forall q \text{ with } (@\mathbf{0}, \texttt{main()}, ()) \longrightarrow_{\widehat{\mathcal{A}}}^{*} \Pi, (\pi, e, q) : |q| \leq k \end{aligned}$$

then for the abstract operational semantics of p holds:

$$|\{[s]_\alpha \mid (@\mathbf{0}, \texttt{main()}, ()) \longrightarrow_{\widehat{\mathcal{A}}}^{*} s\}| < \infty$$

Intuitively, these results are clear. Infinite recursion with non-empty contexts are not possible. The proofs can be found in [Huc99a, Huc99b]. We restrict to this intuitive explanation because we will pick up the problem in Chapter 7. There we define an abstraction of the recursive calls in non-tail positions and get a safe abstraction of non-hierarchical Core Erlang programs.

The class of hierarchical Core Erlang Programs is a true sub-class of all Core Erlang programs which yield a finite transition system for finite domain abstractions. There also exist non-hierarchical Core Erlang programs which also yield a finite abstract operational semantics for finite domain abstractions: for example, the functions which call themselves as inner calls, can be dead code. The AOS can still be finite. However, these cases are artificial for real programs. The class of hierarchical Core Erlang programs matches the programs which have a finite AOS for finite domain abstractions very well.

The class of hierarchical programs is an expressive subclass of Core Erlang. However, there exist many programs which are not hierarchical. For instance, already the use of the functions `append` or `length` yields non-hierarchical programs.

Using this result it is possible to use model checking for the formal verification of hierarchical Core Erlang programs. For finite transition systems and temporal logics like linear time logic (*LTL*), the model checking problem is decidable. We will discuss this in Chapter 5. Afore, we discuss another problem which occurs in the abstraction of `receive` by means of the abstract interpretation function $\widehat{\iota}(\mathsf{mbmatch})$ and the claimed property (P4) b).

4.6 Deadlocks

In the verification of concurrent and distributed systems, a major aspect is the absence of deadlocks. A system is in a deadlock if no process can perform an action since all processes are waiting for a message and no process is able to send a message.

A system can also be partially in a deadlock if a subset of processes exists in which all processes wait for a message from one of these processes. The other processes can still perform actions and communicate with each other. In Erlang they can even send messages to the processes in the deadlock. These messages do not match the patterns of the `receive` expression, the dead processes have suspended in.

A partial deadlock is more difficult to find because not all processes suspend. These deadlock states of the system still have successors. In the verification the absence of partial deadlocks can be proven by the liveness of special processes. A process is alive, if it can reach a special state. As an example, we will prove the deadlock freeness of the database process in Chapter 5.

Here we want to discuss the case that the whole system is in a deadlock. No process can perform any action.

For the verification we will use LTL model checking which is usually only defined on infinite paths. In a deadlock state we only have a finite path. Usually two different techniques can be used to make this finite path infinite. The first technique is to add a cycle for all states which do not have a successor:

$$s \longrightarrow s \text{ for all } s \not\longrightarrow s'$$

Furthermore, we add a proposition dead to these states. This proposition indicates that the system is in a deadlock. Then we can verify that the system never reaches a state containing this proposition. Unfortunately, in combination with our abstraction this is not possible. A state can have successors but they are only possible successors. There exists a concretization of the values in which an action can be performed and another concretization in which no action can be performed. This situation can only occur in the `receive` expression. We consider the following example:

Example 4.35

```
main() -> self!2*20,
          receive
             42 -> 42
          end.
```

For this example, Figure 4.5 shows the SOS and the AOS using the even-odd abstract interpretation. The finite concrete path is represented in the abstraction. However, we cannot recognize that it ends in the abstract state (@**0**,`receive 42->42 end`,(**even**)). In this state the only successor is (@**0**, 42, ()). The matching of the pattern 42 and the abstract value **even** is only possible and not irrefutable.

A dead state in the SOS can be characterized as follows: all processes have terminated or their redex is a `receive` expression in which mbmatch yields Fail. If there still exists a process which can perform another action, then this process can send a

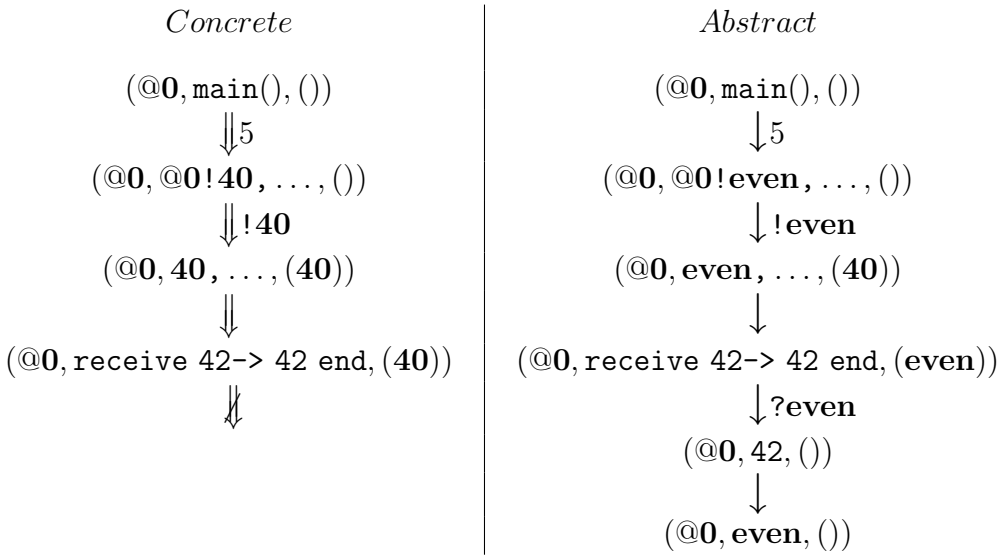


Figure 4.5: SOS and AOS of Example 4.35

message to a suspended process. This message can match one of the patterns of the `receive` expression. The process is not dead.

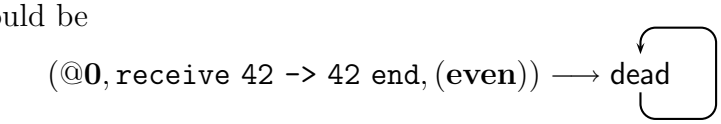
Accordingly, a state in the AOS is a (potentially) dead state if for all processes $(\pi, \widetilde{e}, \widetilde{\mu}) \in \Pi$ holds

- $\widetilde{e} \in \widehat{A}$ or
- $\widetilde{e} = \widetilde{E}[\texttt{receive}\ \widetilde{p}_1\texttt{->}\widetilde{e}_1\texttt{;}\ \ldots\widetilde{p}_n\texttt{->}\widetilde{e}_n\ \texttt{end}]$ and $\mathsf{Fail} \in \mathsf{mbmatch}((\widetilde{p}_1, \ldots, \widetilde{p}_n), \widetilde{\mu})$

Therefore, using the idea of adding a self-loop to dead states we would add a self-loop to the state $(@\mathbf{0}, \texttt{receive 42 -> 42 end}, (\mathbf{even}))$. Unfortunately, we add too many paths with this self-loop. The potentially dead state must not be a sink. In our example, the path in Figure 4.6 would be possible. However, we want to express that if a process is in a deadlock, then it stays in the deadlock forever. Hence, we add a special state dead to our abstract states:

$$\widehat{State}_{\widehat{\mathcal{A}}} := \mathcal{P}_{fin}(\widehat{Proc}_{\widehat{\mathcal{A}}}) \cup \{\mathsf{dead}\}$$

and add a transition into the state dead, for all states which can be dead. In our example this would be



The state dead is a sink indicating that the execution terminates at this point. To guarantee infinite paths of the transition system we add a self-loop to it. This self-loop cannot effect impossible paths because it is the only transition of this state.

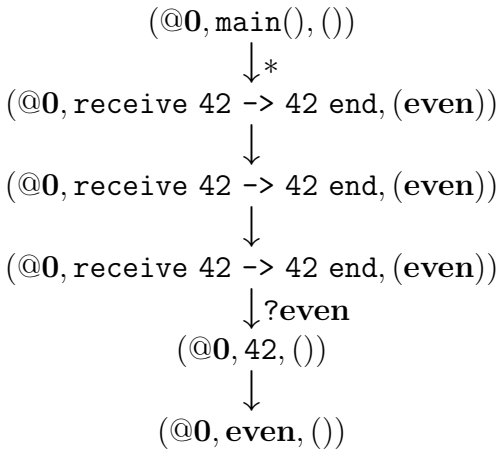


Figure 4.6: Possible path of Example 4.35

Lemma 4.36 (Terminating paths)
Let $\widehat{\mathcal{A}}$ be an abstract interpretation that satisfies properties (P1) – (P5). *For every deadlock state $s \in States$ with $s \not\Rightarrow t$ it holds that for all $\widetilde{s} \sqsubseteq \alpha(s)$: $\widetilde{s} \not\rightarrow \widetilde{t}$ or $\widetilde{s} \longrightarrow \mathsf{dead}$.*

Proof: There are only two cases in which a state does not have a successor state:

- All Core Erlang expressions are values.
  Then also the corresponding abstract Core Erlang expressions are values because of the ordering on Core Erlang expressions and $\widetilde{s} \not\rightarrow \widetilde{t}$ holds trivially.
- Some processes suspend in a `receive` expression and the Core Erlang expressions of the others are ground. All functions $\mathsf{mbmatch}$ applied to the patterns and the mailboxes yield Fail. Then with property (P4) b) we know that Fail is in the abstract interpretation $\widehat{\iota}(\mathsf{mbmatch})$ applied to arbitrary abstractions of these values. Exactly for this case we have defined $\widetilde{s} \longrightarrow \mathsf{dead}$. □

The paths on which partial deadlocks exist are adequately represented. There always exists an "unfair" path on which these processes do not perform any action and their deadlock can be detected by the missing absence of liveness for these processes. This can be expressed in LTL.

Chapter 5

Verification of Core Erlang Programs

In the last chapter we have defined a framework for the abstract interpretation of Core Erlang programs. Now we want to use this framework for the formal verification by model checking.

5.1 Core Erlang with Propositions

The AOS defines a labelled transition system. We want to prove properties of the system by model checking. It would be possible to specify properties using labels. On the other hand, it is more convenient to add propositions to the states of this transition system. With these propositions, properties can be expressed more easily in a temporal logic. As names for the propositions we use arbitrary Core Erlang constructor terms which is very natural for Erlang programmers. This is an infinite set but with respect to the abstract interpretation these values are usually restricted to a finite abstract domain. Therefore, we get a finite set of abstract propositions. This is necessary to get a finite transition system in which model checking is decidable.

We extend the predefined Core Erlang functions by the function `prop/1`. This function is only considered to introduce proposition and not to change the semantics of programs. Therefore, its interpreted as the identity function in operational semantics.

$$\iota(\texttt{prop})(v) = v \quad \text{for all } v \in T_{\mathcal{C}}(Pid)$$

Additionally, the state in which $\texttt{prop}(v)$ is evaluated has the proposition v. To indicate this in the transition system we add a label prop to the transition of its evaluation:

$$\frac{-}{\Pi, (\pi, E[\texttt{prop}(v)], \mu) \xrightarrow{\mathsf{prop}}_{\hat{\mathcal{A}}} \Pi, (\pi, E[v], \mu)}$$

The valid propositions of a process can be evaluated using the function prop.

Defintion 5.1 (Proposition of a Process)
The proposition of a process is defined by the function:

$$\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{Proc}_{\widehat{\mathcal{A}}} \longrightarrow \{\mathsf{Nothing}\} \cup \widehat{A}$$
$$\mathsf{prop}_{\widehat{\mathcal{A}}}((\pi, E[e], \mu)) := \begin{cases} \widehat{v} & \text{, if } e = \mathsf{prop}(\widehat{v}) \text{ and } \widehat{v} \in \widehat{A} \\ \mathsf{Nothing} & \text{, otherwise} \end{cases}$$ ◁

The propositions of a state are all propositions of the processes of the state. We extend the function prop to $\widehat{State}_{\widehat{\mathcal{A}}}$.

Defintion 5.2 (Propositions of a State)
The propositions of a state are defined by the function:

$$\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{State}_{\widehat{\mathcal{A}}} \longrightarrow \mathcal{P}(\widehat{A})$$
$$\mathsf{prop}_{\widehat{\mathcal{A}}}(\emptyset) = \emptyset$$
$$\mathsf{prop}_{\widehat{\mathcal{A}}}(\Pi, \pi) = \begin{cases} \mathsf{prop}_{\widehat{\mathcal{A}}}(\Pi) & \text{, if } \mathsf{prop}_{\widehat{\mathcal{A}}}(\pi) = \mathsf{Nothing} \\ \mathsf{prop}_{\widehat{\mathcal{A}}}(\Pi) \cup \mathsf{prop}_{\widehat{\mathcal{A}}}(\pi) & \text{, otherwise} \end{cases}$$ ◁

For both functions we use the name $\mathsf{prop}_{\widehat{\mathcal{A}}}$. The concrete instance of this overloading will be clear from the application of $\mathsf{prop}_{\widehat{\mathcal{A}}}$. We will also omit abstract interpretation as an index if it is clear from the context.

Example 5.3
We add the following propositions to the Core Erlang program p_{db} of Example 3.7:

```
dataBase(L) -> prop(top),
               receive
                  {allocate,Key,P} ->
                     prop({allocate,P}),
                     case lookup(Key,L) of
                        fail -> P!free,
                                receive
                                   {value,V,P} ->
                                      prop({value,P}),
                                      dataBase(insert(Key,V,L))
                                end;
                        {succ,V} -> P!prop(allocated),
                                    dataBase(L)
                     end;
                  {lookup,Key,P} -> prop(lookup),
                                    P!lookup(Key,L),
                                    dataBase(L)
               end.
```

In most cases propositions will be added in a sequence, as for example, the proposition `top`. However, it is also possible to mark existing (sub-)expressions as propositions. As an example we use the atom `allocated` which is sent to a requesting process, as a proposition.

We have added four propositions to the definition of the database. They have the following meanings:

`top`	marks the main state of the database process
`{allocate,P}`	marks that the process with the pid `P` tries to allocate a key
`{value,P}`	marks that the process with the pid `P` enters a value into the database
`lookup`	marks a reading access to the database

We will use these propositions to verify the database.

5.2 Linear Time Temporal Logic

The abstract operational semantics defines a transition system. We want to prove properties of this transition system using model checking. The properties are described in a temporal logic. We use *linear time temporal logic* (*LTL*) [GPSS80] in which properties have to hold on every path of a given transition system. From Theorem 4.12 we know that the abstract semantics has more paths than the standard operational semantics and we get: if a property φ holds in the AOS, then φ also holds for the SOS of the program.

Defintion 5.4 (Syntax of Linear Time Logic (LTL))
Let *Props* be a set of state propositions. The set of *LTL-formulas* is defined as the smallest set with:

- $Props \subseteq \text{LTL}$ — state propositions
- $\varphi, \psi \in \text{LTL} \Longrightarrow$
  - $\neg\varphi \in \text{LTL}$ — negation
  - $\varphi \wedge \psi \in \text{LTL}$ — conjunction
  - $X\varphi \in \text{LTL}$ — in the next state φ holds
  - $\varphi\ U\ \psi \in \text{LTL}$ — φ holds until ψ holds

$\triangleleft$

An LTL-formula is interpreted with respect to an infinite path. The propositional formulas are satisfied, if the first state of a path fulfills them. The next modality $X\varphi$ holds if φ holds in the continuation of the path. Finally, LTL contains a strong until: If φ holds until ψ holds and ψ eventually holds, then $\varphi\ U\ \psi$ holds: Formally, the semantics is defined as follows:

Defintion 5.5 (Path Semantics of LTL)
An infinite word over sets of propositions $\pi = p_0p_1p_2 \ldots \in \mathcal{P}(Props)^\omega$ is called a path. A path π satisfies an LTL-formula φ ($\pi \models \varphi$) in the following cases:

$$
\begin{array}{rcll}
p_0\pi & \models & P & \text{iff } P \in p_0 \\
\pi & \models & \neg\varphi & \text{iff } \pi \not\models \varphi \\
\pi & \models & \varphi \wedge \psi & \text{iff } \pi \models \varphi \text{ and } \pi \models \psi \\
p_0\pi & \models & X\varphi & \text{iff } \pi \models \varphi \\
p_0p_1\ldots & \models & \varphi\ U\ \psi & \text{iff } \exists i \in \mathbb{N} : p_ip_{i+1}\ldots \models \psi \text{ and } \forall j < i : p_jp_{j+1}\ldots \models \varphi
\end{array}
\qquad \lhd
$$

The semantics of $\varphi U \psi$ is defined by indices directly on the path. An equivalent definition of the semantics is the unwinding of the fix-point:

$$\pi \models \varphi\ U\ \psi \quad \text{iff } \pi \models \psi \vee (\varphi \wedge X(\varphi\ U\ \psi))$$

Where $\vee$ is the disjunction of two formulas. It will be defined in Definition 5.8.

Formulas are not only interpreted with respect to a single path. Their semantics is extended to Kripke Structures.

Defintion 5.6 (Kripke Structure)
$\mathcal{K} = (S, Props, \longrightarrow, \tau, s_0)$ where

- S a set of states,
- $Props$ a set of propositions,
- $\longrightarrow \subseteq S \times S$ the transition relation,
- $\tau : S \longrightarrow \mathcal{P}(Props)$ a labeling function for the states, and
- $s_0 \in S$ the initial state

is called a *Kripke Structure*. Instead of $(s, s') \in \longrightarrow$ we usually write $s \longrightarrow s'$.

A *state path of* $\mathcal{K}$ is an infinite word $s_0s_1\ldots \in S^\omega$ where $s_i \longrightarrow s_{i+1}$ and s_0 the initial state of $\mathcal{K}$. If $s_0s_1\ldots$ is a state path of $\mathcal{K}$ and $p_i = \tau(s_i)$ for all $i \in \mathbb{N}$, then the infinite word $p_0p_1\ldots \in \mathcal{P}(Props)^\omega$ is a *path of* $\mathcal{K}$. $\lhd$

Defintion 5.7 (Kripke-Structure-Semantics of LTL)
Let $\mathcal{K} = (S, \rightarrow, \tau, s_0)$ be a Kripke structure. It satisfies an LTL-formula φ ($\mathcal{K} \models \varphi$) iff for all paths π of $\mathcal{K}$: $\pi \models \varphi$. $\lhd$

The technique of *model checking* automatically decides whether or not a given Kripke structure satisfies a given formula. For finite Kripke structures and the logic LTL model checking is decidable [LP85]. Many optimizations of this first model checking algorithm have been proposed [GPVW95, Var96, Hol96, Wal98, BRS99]. Unfortunately, the complexity of all these optimizations is still exponential in the size of the formula and linear in the size of the Kripke structure. Also for some classes of infinite state Kripke structures model checking is decidable. These systems are in some sense context-free. On top of their finite representation model checking algorithms can be defined [BE97, BS92, Esp97].

For convenient specification of properties in LTL, we define the following abbreviations:

Defintion 5.8 (Abbreviations in LTL)

ff	$:=$	$\neg P \wedge P$ [1]	the boolean value true
tt	$:=$	$\neg ff$	the boolean value false
$\varphi \vee \psi$	$:=$	$\neg(\neg\varphi \wedge \neg\psi)$	disjunction
$\varphi \rightarrow \psi$	$:=$	$\neg\varphi \vee \psi$	implication
$F\ \varphi$	$:=$	$tt\ U\ \varphi$	finally φ holds
$G\ \varphi$	$:=$	$\neg F \neg\varphi$	globally φ holds
$F^{\infty}\varphi$	$:=$	$G\ F\ \varphi$	infinitely often φ holds
$G^{\infty}\varphi$	$:=$	$F\ G\ \varphi$	only finally often φ does not hold
$\varphi\ W\ \psi$	$:=$	$(\varphi\ U\ \psi) \vee (G\ \varphi)$	weak until

◁

The propositional abbreviations are standard. $F\varphi$ is satisfied if there exists a position in the path where φ holds. If in every position of the path φ holds, then $G\varphi$ is satisfied. The formulas φ which have to be satisfied in these positions of the path are not restricted to propositional formulas. They can express properties of the whole remaining path. This fact is used in the definition of $F^{\infty}\varphi$ and $G^{\infty}\varphi$. The weaker property $F^{\infty}\varphi$ postulates that φ holds infinitely often on a path. Whereas $G^{\infty}\varphi$ is satisfied if φ is satisfied with only finitely many exceptions. In other words there exists a position from which φ holds forever. Finally, the definition of a weak until $\varphi\ W\ \psi$ can be useful. It is even satisfied if φ holds forever.

For the verification of Core Erlang programs we use the AOS of a Core Erlang program as a Kripke structure. We use the transition system over α-equivalence classes which is spawned from the initial state (@**0**,`main()`,()). For labeling of the states we use the function prop from the previous section. By means of the technique presented in Section 4.6 we have already guaranteed infinite paths for states which represent a deadlock. There is one more state which does not have any successor. This is the state error. To ensure that all paths are infinite, we add a self-loop to this state.

5.2.1 Abstraction of Propositions

We want to verify Core Erlang programs by model checking. We have defined a framework for abstract interpretations of Core Erlang programs. This abstraction guarantees that every path of the SOS is also represented in the AOS. If the resulting AOS is finite, then we can use simple model checking algorithms to check if the property φ expressed in LTL holds. If φ is holds in the AOS, then φ also holds in the SOS. In the other case model checking yields a counter example which is a path in the transition system, on which φ does not hold. Due to the fact that the AOS contains more paths than the SOS, the counter example must not be a counter example for the SOS. The counter path can be a valid path in the abstraction but not in the SOS. Therefore, in this case it only yields a hint that the chosen abstraction is too coarse and must be refined.

[1] We assume that the set of propositions *Props* is not empty. This assumption is needed anyway. Otherwise, the set of LTL-formulas would be empty as well. P is an arbitrary element of *Props*.

The application of model checking seems to be simple. However, when proving state propositions some problems appear as the following example shows:

Example 5.9

```
main() -> prop(42).
```

A possible property of the program could be F **42**. To prove this property we use the AOS for an abstract interpretation, for instance, the even-odd interpretation. Using this abstraction the AOS yields the following transition system:

$$(@\mathbf{0}, \texttt{main()}, ()) \longrightarrow (@\mathbf{0}, \texttt{prop(42)}, ()) \longrightarrow (@\mathbf{0}, \texttt{prop(}\mathbf{even}\texttt{)}, ()) \overset{\mathsf{prop}}{\longrightarrow} (@\mathbf{0}, \mathbf{even}, ())$$

Only the state $(@\mathbf{0}, \texttt{prop(}\mathbf{even}\texttt{)}, ())$ has a property, namely **even**. **42** is an even number, but it is not the only even number. For safeness, this property cannot be proven. Otherwise, we could also prove the property F **40** which does not hold.

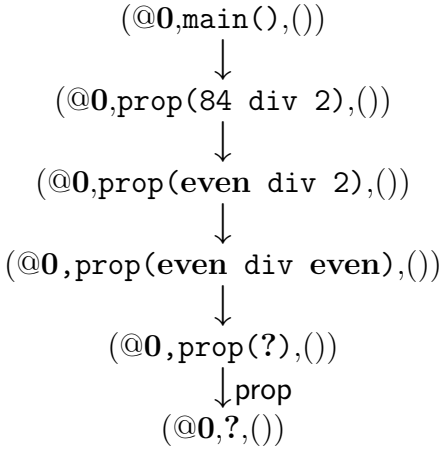
It is only possible to prove properties for which the corresponding abstract value exclusively represents this value. However, it does not make much sense to abstract from special values and express properties for these concrete values afterwards. Therefore, we only use propositions of the abstract domain, like F **even**. In the AOS the state (@**0**,`prop(`**even**`)`,()) has the property **even**. Therefore, the program this property holds in the program.

Now we consider a more complicated example:

Example 5.10

```
main() -> prop(84 div 2).
```

The property holds in this system as well because $(84 \div 2) = 42$. In the even-odd abstraction we only get:



where prop((@**0**,`prop(?)`,()))= {**?**} and $\emptyset$ as propositions of the other states. In a safe abstraction we cannot be sure that the property F **even** holds. Hence, model checking must yield that it does not hold. For instance, for the program

```
main() -> prop(42 div 2).
```

the AOS is similar but the property does not hold ($42 \div 2 = 21$). Therefore, a propositional property holds in a state if the proposition of the state is at least as precise as the expected proposition which can be formalized with the following conjecture:

$$p_0 p_1 \ldots \models \widetilde{v} \text{ iff } \exists \widetilde{v}' \in p_0 \text{ where } \widetilde{v} \sqsubseteq \widetilde{v}'$$

Unfortunately, this is not correct in all cases, as the following example shows. We want to prove that the property

$$\psi = G\neg\mathbf{even}$$

holds in the program. Therefore, one point is to check that the state (@**0**,`prop(?)`,()) models ¬**even**. Using our conjecture we can conclude

$$(@\mathbf{0},\texttt{prop(?)},()) \not\models \mathbf{even}$$

and hence,

$$(@\mathbf{0},\texttt{prop(?)},()) \models \neg\mathbf{even}.$$

Unfortunately, this is wrong because in Example 5.10 the property does not hold. The SOS has the property 42 which is an even value. The problem is the non-monotonic operation ¬. Considering abstraction, the equivalence

$$\pi \models \neg\varphi \text{ iff } \pi \not\models \varphi$$

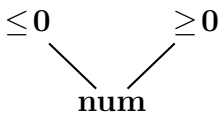
does not hold! $\pi \not\models \varphi$ only means that $\pi \models \varphi$ is not safe. In other words, there can be a concretization which satisfies φ, but we cannot be sure that it holds for all concretizations. Therefore, negation has to be handled carefully.

Which value of our abstract domain would fulfill the negated proposition ¬**even**? Only the proposition **odd** does. The values **even** and **odd** are incomparable and no value exists which is more precise than these two abstract values. This connection can be generalized as follows:

$$p_0 p_1 \ldots \models \neg\widetilde{v} \text{ if } \forall \widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist}$$

Note that this is no equivalence anymore. The non-existence of $\widetilde{v} \sqcup \widetilde{v}'$ does only imply that $p_0 p_1 \ldots \models \neg\widetilde{v}$. It does not give any information for the negation $p_0 p_1 \ldots \models \widetilde{v}$. This negation holds if $\exists \widetilde{v}' \in p_0$ where $\widetilde{v} \sqsubseteq \widetilde{v}'$.

On a first sight refuting a proposition seems not to be correct for arbitrary abstract interpretations. Consider the abstract domain



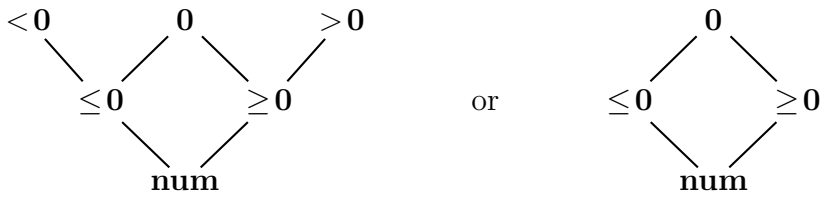
where the abstract values represent the following concrete values

abstract value	represented concrete values
$\leq \mathbf{0}$	$\{\mathbf{0}, -\mathbf{1}, -\mathbf{2}, \ldots\}$
$\geq \mathbf{0}$	$\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$
num	$\mathbb{Z}$

The represented concrete values of $\leq \mathbf{0}$ and $\geq \mathbf{0}$ overlap. Both represent the value $\mathbf{0}$. Therefore, it would be incorrect that a state containing the proposition $\leq \mathbf{0}$ satisfies the formula $\neg \geq \mathbf{0}$.

However, this abstract domain is not possible. The abstraction function $\alpha : A \longrightarrow \widehat{A}$ can only yield one abstract representation for a concrete value. Without loss of generality let $\alpha(\mathbf{0}) = \geq \mathbf{0}$. Abstract values which represent the concrete value $\mathbf{0}$ can only be the result of the use of the abstract interpretation function $\widehat{\iota}$. All these results $\widetilde{v}$ must be less precise: $\widetilde{v} \sqsubseteq \alpha(\mathbf{0}) = \geq \mathbf{0}$ because of the properties (P1) – (P5). Hence, this abstract domain can be defined, but the value $\leq \mathbf{0}$ does only represent the values $\{-\mathbf{1}, -\mathbf{2}, \ldots\}$. The name of the abstract value is not relevant, but for understandability it should be renamed to $< \mathbf{0}$.

Alternatively the abstract domain can be refinement. The two overlapping abstract values can be distinguished in a more precise abstract value:



In both cases we must define $\alpha(\mathbf{0}) = \mathbf{0}$. Otherwise, we have the same situation as before. The concrete value $\mathbf{0}$ is not represented by both abstract values $\leq \mathbf{0}$ and $\geq \mathbf{0}$.

5.3 Semantics of Propositions

Using the considerations of the previous section, we can now formalize whether a propositional formula can be proven or refuted respectively. Similar results have been found in by Clark, Grumberg, and Long [CGL94b] and Knesten and Pnueli [KP98]. The result of Knesten and Pnueli introduces a solution to the problem informally without any formalization. The paper of Clark et. al. formalizes a solution, but their framework differs from ours and the result cannot easily be transfered to our framework.

First we define the concretization of an abstract value. This is the set of all concrete values which are abstracted to this value or a more precise abstract value.

Defintion 5.11 (Concretization of Abstract Values)
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation. The concretization function $\gamma : \widehat{A} \longrightarrow \mathcal{P}(T_{\mathcal{C}}(Pid))$ is defined as

$$\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}.$$

$\lhd$

For the last example we get the following concretizations:

$$\begin{array}{lcl} \gamma(\mathbf{0}) & = & \{\mathbf{0}\} \\ \gamma(\leq\mathbf{0}) & = & \{\mathbf{0}, -\mathbf{1}, -\mathbf{2}, \ldots\} \\ \gamma(\geq\mathbf{0}) & = & \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\} \\ \gamma(\mathbf{num}) & = & \mathbb{Z} \end{array}$$

The following connections between the abstraction and the concretization function hold.

Lemma 5.12 (Connections between γ and α)
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation and γ the corresponding concretization function. Then the following properties hold:

1. $\forall v \in \gamma(\widetilde{v}) : \widetilde{v} \sqsubseteq \alpha(v)$
2. $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \widetilde{v}$

Proof:

1. $v \in \gamma(\widetilde{v})$ iff $v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}$ iff $\widetilde{v} \sqsubseteq \alpha(v)$
2. $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \bigsqcap\{\alpha(v) \mid v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}\} = \bigsqcap\{\alpha(v) \mid \widetilde{v} \sqsubseteq \alpha(v)\} = \widetilde{v}$

□

Using the concretization function we can now define whether a proposition of a state satisfies a proposition in the formula or refutes it.

Defintion 5.13 (Semantics of a Proposition)
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:

$$\begin{array}{ll} p \models \widetilde{v} & \text{if } \exists\widetilde{v}' \in p \text{ where } \gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v}) \\ p \not\models \widetilde{v} & \text{if } \forall\widetilde{v}' \in p \text{ holds } \gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset \end{array}$$ ◁

With these definitions for the concretization we can now formalize corresponding implications for abstract values. For finite domain abstractions they can be decided automatically.

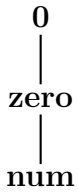
Lemma 5.14 (Deciding Propositions in the abstract domain)
Let $\widehat{\mathcal{A}} = (\widehat{A}, \iota, \sqsubseteq, \alpha)$ be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:

$$\begin{array}{ll} p \models \widetilde{v} & \textit{if } \exists\widetilde{v}' \in p \textit{ where } \widetilde{v} \sqsubseteq \widetilde{v}' \\ p \not\models \widetilde{v} & \textit{if } \forall\widetilde{v}' \in p \textit{ holds } \widetilde{v} \sqcup \widetilde{v}' \textit{ does not exist} \end{array}$$

Proof: We show: $\widetilde{v} \sqsubseteq \widetilde{v}'$ implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$ and the non-existence of $\widetilde{v}' \sqcup \widetilde{v}'$ implies $\gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset$:

- $\widetilde{v} \sqsubseteq \widetilde{v}'$
  $\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}$ and $\gamma(\widetilde{v}') = \{v \mid \widetilde{v}' \sqsubseteq \alpha(v)\}$.
  $(\widehat{A}, \sqsubseteq)$ is a partial order. Hence, it is transitive. This implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$
- $\widetilde{v} \sqcup \widetilde{v}'$ does not exist $\Longrightarrow \gamma(\widetilde{v} \sqcup \widetilde{v}') = \emptyset \Longrightarrow \{v \mid (\widetilde{v} \sqcup \widetilde{v}') \sqsubseteq \alpha(v)\} = \emptyset$
  $\Longrightarrow \{v \mid \widetilde{v}' \sqsubseteq \alpha(v) \text{ and } \widetilde{v} \sqsubseteq \alpha(v)\} = \emptyset \Longrightarrow \gamma(\widetilde{v}') \cap \gamma(\widetilde{v}) = \emptyset$ □

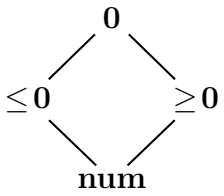
Note that we only show an implication. We can define unnatural abstract domains in which a property holds or is refuted with respect to Definition 5.13. Only using the abstract domain we cannot show this. We consider the following abstract domain:



$$\text{where } \alpha(v) = \begin{cases} \mathbf{0} & \text{if } v = \mathbf{0} \\ \mathbf{num} & \text{otherwise} \end{cases}$$

The abstract value zero is superfluous because it represents exactly the same values, as the abstract value **0**. However, this abstract domain is valid. Using the definition of the semantics of a proposition from Definition 5.13 we can show that $\{\mathbf{zero}\} \models \mathbf{0}$ because $\gamma(\mathbf{zero}) = \gamma(\mathbf{0}) = \{\mathbf{0}\}$. On the other hand, $\mathbf{zero} \sqsubseteq \mathbf{0}$ but we cannot show that $\{\mathbf{zero}\} \models \mathbf{0}$ just using the abstract domain.

The same holds for refuting a proposition, as the following example shows:



$$\text{where } \alpha(v) = \begin{cases} \geq\mathbf{0} & \text{if } v \geq \mathbf{0} \\ \leq\mathbf{0} & \text{otherwise} \end{cases}$$

In this domain the abstract value **0** is superfluous. Its concretization is empty. Hence, $\gamma(\leq\mathbf{0}) = \{-\mathbf{1}, -\mathbf{2}, \ldots\}$ and $\gamma(\geq\mathbf{0}) = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$. $\gamma(\leq\mathbf{0}) \cap \gamma(\geq\mathbf{0}) = \emptyset$ and $\leq\mathbf{0} \models \neg\geq\mathbf{0}$. In this abstract domain, this proposition cannot be refuted since $\leq\mathbf{0} \sqcup \geq\mathbf{0} = \mathbf{0}$ exists.

These examples are unnatural because the domains contain superfluous abstract values. Nobody will define domains like these. Usually, the concretization of an abstract value is nonempty and differs from the concretizations of all other abstract values. In this case deciding propositions in the abstract domain is complete with respect to the semantics of propositions. Although it is not complete in general, it is safe. If we can prove a property for abstract values, then it is also correct for its concretizations.

5.4 Proving LTL Formulas

In the last section we have discussed whether a propositional formula can be proven or refuted. LTL allows negation not only in front of propositions. Arbitrary sub-formulas can be negated. To solve this problem two different approaches are possible:

- All negations can be pushed inside the formula, until they only occur in front of the propositions. Therefore, we must add a release modality ($\varphi R \psi$) to LTL because there exists no equivalent representation of $\neg(\varphi U \psi)$ which uses negation only in front of φ and ψ. Release is the dual modality of until:

  $$\neg(\varphi U \psi) \sim \neg\varphi R \neg\psi$$

  Therefore, its semantics is defined as

  $$p_0 p_1 \ldots \models \varphi R \psi \text{ iff } \forall i \in \mathbb{N} : p_i p_{i+1} \ldots \models \psi \text{ or } \exists j < i : p_j p_{j+1} \ldots \models \varphi$$

  There is no intuitive semantics of release except that it can be used for negation of until. However, it can also be unwinded:

  $$\pi \models \varphi R \psi \quad \text{iff} \quad \pi \models \neg\psi \wedge (\neg\varphi \vee X \neg(\varphi U \psi))$$

  Therefore, it can also be automatically verified in model checking.

  Furthermore, we must add $\vee$ to LTL and use the following equivalences:

  $$\begin{aligned} \neg\neg\varphi &\sim \varphi \\ \neg(\varphi \wedge \psi) &\sim \neg\varphi \vee \neg\psi \\ \neg(X\varphi) &\sim X\neg\varphi \end{aligned}$$

  Using these equivalences all negations can be pushed into a formula. The result is an equivalent formula in which negations only occur directly in front of propositions. Then these formulas can be used for model checking. Positive and negative propositions can be checked with respect to Lemma 5.14.

- Standard model checking algorithms use a similar idea though they do not need the release modality. For example, in [Var96] an alternating automaton is constructed that represents the maximal model which satisfies a formula. The states correspond to the possible sub-formulas and their negations. For every negation in the formula the automaton switches to the corresponding state which represents the positive respectively negative sub-formula. Using this alternation the negations are pushed into the automaton representing the formula, like in the first approach. A proposition has to be valued as a positive proposition if it is used after an even number of negations. In the other case it is valued as a negative proposition. It has to be refuted.

  We can use the same idea and distinguish two different kinds of propositions. The number of negations in front of a proposition are counted. In dependency of an even or an odd number of negations the propositional formula must be proven or refuted. It is possible that the same property occurs more than ones in a formula. The different occurrences must be considered separately because there can be different numbers of negations in front of them.

  The advantage of this approach is that the non-intuitive release modality is not needed. The definition of LTL can be left unchanged. The semantics of propositions only depends on the number of negations in front of them. In the following, we will define a simple algorithm for marking all propositions in a formula with $+$ or $-$.

The number of negations in front of a proposition can easily be computed by the following algorithm. We define a function mark which accumulates if the number of negations in front of the actual sub-formula is even (+) or odd (−). If mark reaches a proposition, then this proposition is annotated with the actual accumulated sign. If a negation occurs the algorithm flips + and − by the function $\overline{\ }: \{+,-\} \longrightarrow \{+,-\}$:

$$\overline{+} = - \qquad \text{and} \qquad \overline{-} = +$$

All other operators in the formula are just copied without any modification. The marked formulas $\mathrm{LTL}^{\{+,-\}}$ are defined as LTL-formulas except that we use signed propositions ($Props \times \{+,-\}$). We write ${}^{s}P$ instead of (P,s).

$$\begin{array}{lcl}
\mathsf{mark} : (\{+,-\} \times \mathrm{LTL}) \longrightarrow \mathrm{LTL}^{\{+,-\}} & & \\
\mathsf{mark}(s,P) & = & {}^{s}P \\
\mathsf{mark}(s,\neg\varphi) & = & \neg\mathsf{mark}(\overline{s},\varphi) \\
\mathsf{mark}(s,\varphi\wedge\psi) & = & \mathsf{mark}(s,\varphi)\wedge\mathsf{mark}(s,\psi) \\
\mathsf{mark}(s,X\varphi) & = & X\mathsf{mark}(s,\varphi) \\
\mathsf{mark}(s,\varphi U\psi) & = & \mathsf{mark}(s,\varphi)U\mathsf{mark}(s,\psi)
\end{array}$$

With respect to the definitions of the abbreviations we also get:

$$\begin{array}{lcl}
\mathsf{mark}(s,tt) & = & tt \\
\mathsf{mark}(s,ff) & = & ff \\
\mathsf{mark}(s,\varphi\vee\psi) & = & \mathsf{mark}(s,\varphi)\vee\mathsf{mark}(s,\psi) \\
\mathsf{mark}(s,\varphi\longrightarrow\psi) & = & \mathsf{mark}(\overline{s},\varphi)\longrightarrow\mathsf{mark}(s,\psi) \\
\mathsf{mark}(s,F\varphi) & = & F\mathsf{mark}(s,\varphi) \\
\mathsf{mark}(s,G\varphi) & = & G\mathsf{mark}(s,\varphi) \\
\mathsf{mark}(s,F^{\infty}\varphi) & = & F^{\infty}\mathsf{mark}(s,\varphi) \\
\mathsf{mark}(s,G^{\infty}\varphi) & = & G^{\infty}\mathsf{mark}(s,\varphi)
\end{array}$$

In the first call of mark no negations must be considered. Therefore, it is initially called with the sign +:

$$\begin{array}{l}
\mathsf{mark} : \mathrm{LTL} \longrightarrow \mathrm{LTL}^{\{+,-\}} \\
\mathsf{mark}(\varphi) = \mathsf{mark}(+,\varphi)
\end{array}$$

For the two kinds of propositions we can now define

$$\begin{array}{l}
p_0p_1\ldots \models^{+} \widetilde{v} \ \text{ if } \ \exists\widetilde{v}' \in p_0 \text{ where } \widetilde{v} \sqsubseteq \widetilde{v}' \\
p_0p_1\ldots \not\models^{-} \widetilde{v} \ \text{ if } \ \forall\widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist}
\end{array}$$

5.5 Verification of the Database

In Example 5.3 we have added propositions to our database example. Now we want to prove some properties of this system. We consider a system which consists of a database and two client processes. It is started by the following `main` function:

```
main() -> spawn(client,[self]),
          spawn(client,[self]),
          database([]).
```

For the verification we use the abstract interpretation $\widehat{\mathcal{A}}_k$ in which only constructor terms of depth $\leq k$ are considered. Abstracted sub-terms are represented by **?**.

5.5.1 Liveness of the database

First we want to prove that for every request the database again reaches its initial state in which it can handle new requests. Therefore, we have added the proposition `top`. We postulate:

> If the database receives an allocation or lookup request, then finally it reaches the main state in which the proposition `top` holds.

In LTL this can be specified as

$$\mathit{alwaysTop} = G((\mathbf{\{allocate, ?\}} \vee \mathbf{lookup}) \longrightarrow F\ \mathbf{top})$$

We use the abstract value **{allocate,?}** to express that an arbitrary process sends a request for the allocation of a new key. This property is a liveness property (finally a special state can be reached). On the other hand, Erlang is a programming language and the processes behave deterministically (see Lemma 3.20). Therefore, this property also shows the absence of deadlocks in the execution of the code between the specified propositions.

Applying the mark function to this formula we get:

$$\mathsf{mark}(\mathit{alwaysTop}) = G((^{-}\mathbf{\{allocate, ?\}} \vee\ ^{-}\mathbf{lookup}) \longrightarrow F\ ^{+}\mathbf{top})$$

The propositions **{allocate,?}** and **lookup** in the requirement of the implication are marked with $-$. Therefore, only states with incomparable propositions refute them. For instance, a state having the proposition **{allocate, @0}** does not refute the property $^{-}$**{allocate, ?}**. Hence, the right side of the implication F **top** has to be satisfied.

The abstract value **lookup** represents exactly one concrete value (γ(**lookup**) = {**lookup**}). Only **lookup** satisfies $^{+}$**lookup**, while $^{-}$**lookup** is not refuted by the propositions **lookup** and **?**. Finally, **top** is only satisfied by the abstract state proposition **top**.

We want to prove this formula using our example abstraction $\widehat{\mathcal{A}}_k$. The abstract interpretation $\widehat{\mathcal{A}}_1$ is too coarse as the path of the AOS in Figure 5.1 shows. State 1 shows the system after the creation of the two client processes (π_2 and π_3 which we do not represent in more detail). Furthermore, these two processes have already sent requests to the database. All flags of these messages are abstracted. Different messages can only be distinguished with respect to their arity. In State 2 the first message in the mailbox is interpreted as an **allocate** message and this branch is chosen. The proposition **{allocate, ?}** holds in State3 but no successor state of this path has the proposition **top**. The reason is that the matching of the pattern **{value,V,?}** against the message **{?,?,?}** is not irrefutable. Due to this fact the deadlock sink dead is a successor of this state.

After sending in State 4, the process π_2 has changed to π_2' because the message **free** is sent to this process. In the abstract domain $\widehat{\mathcal{A}}_1$ also the pid of the requesting process is abstracted to **?**. So, the message **free** can in this abstraction also be sent to the other client or the database. Furthermore, it can also yield an error because **?** can also represent a non-pid. Hence, State 4 also has error as successor and the formula is not satisfied on this path either.

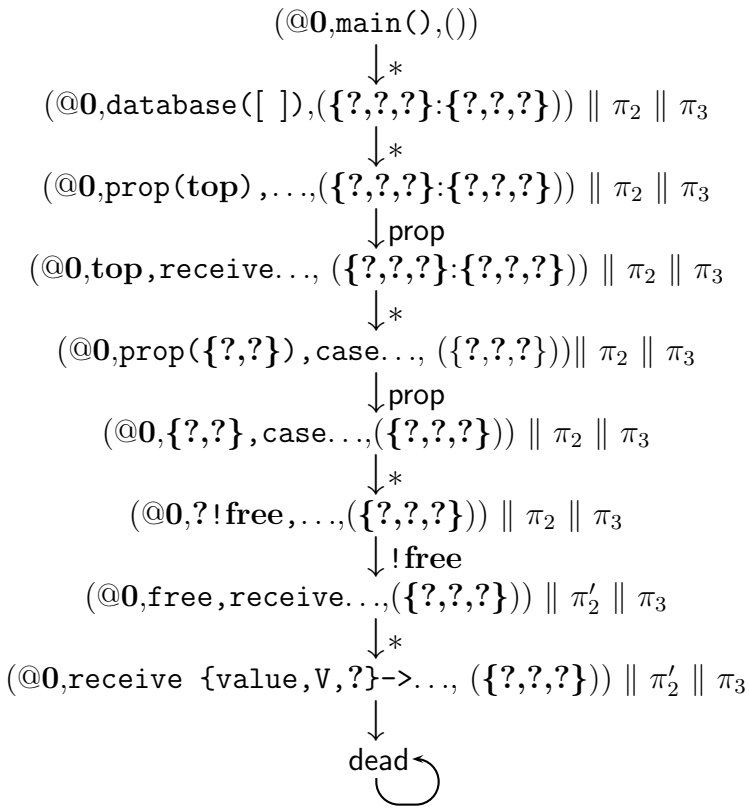


Figure 5.1: Path of the database example in the abstract interpretation $\widehat{\mathcal{A}}_1$

ε-Loops

We need more precision in the abstraction. Figure 5.2 shows the same path in the abstraction $\widehat{\mathcal{A}}_2$. We see that all branches which are relevant for the communication can be decided here. The system does not run into the deadlock state and after the proposition **{allocate,?,@1}** finally the proposition **top** holds. There exists another path (see Figure 5.3) in which the property does not hold. State 1 in Figure 5.3 does not falsify the proposition lookup. Therefore, the proposition **top** should finally hold. However, this is not true for the presented path. The path contains an ε-loop resulting from the evaluation of `lookup(?,?)` in State 2. The reason is that our abstraction is not designed to decide the termination of a recursive function call. Other techniques of abstraction exist which can decide termination of recursive function calls. An integration into our abstraction would be difficult. The definition of an abstraction which is designed for the verification of protocol properties and the termination of recursive function definitions would be much more difficult.

We decided to eliminate non-terminating sequential calculations from the transition system used as a Kripke Structure for model checking. This can be implemented, by ignoring ε-loops. In our example, this leads non-deterministically to the results **{value,?}** and **fail** for the function application `lookup(?,?)`. Both results are sent

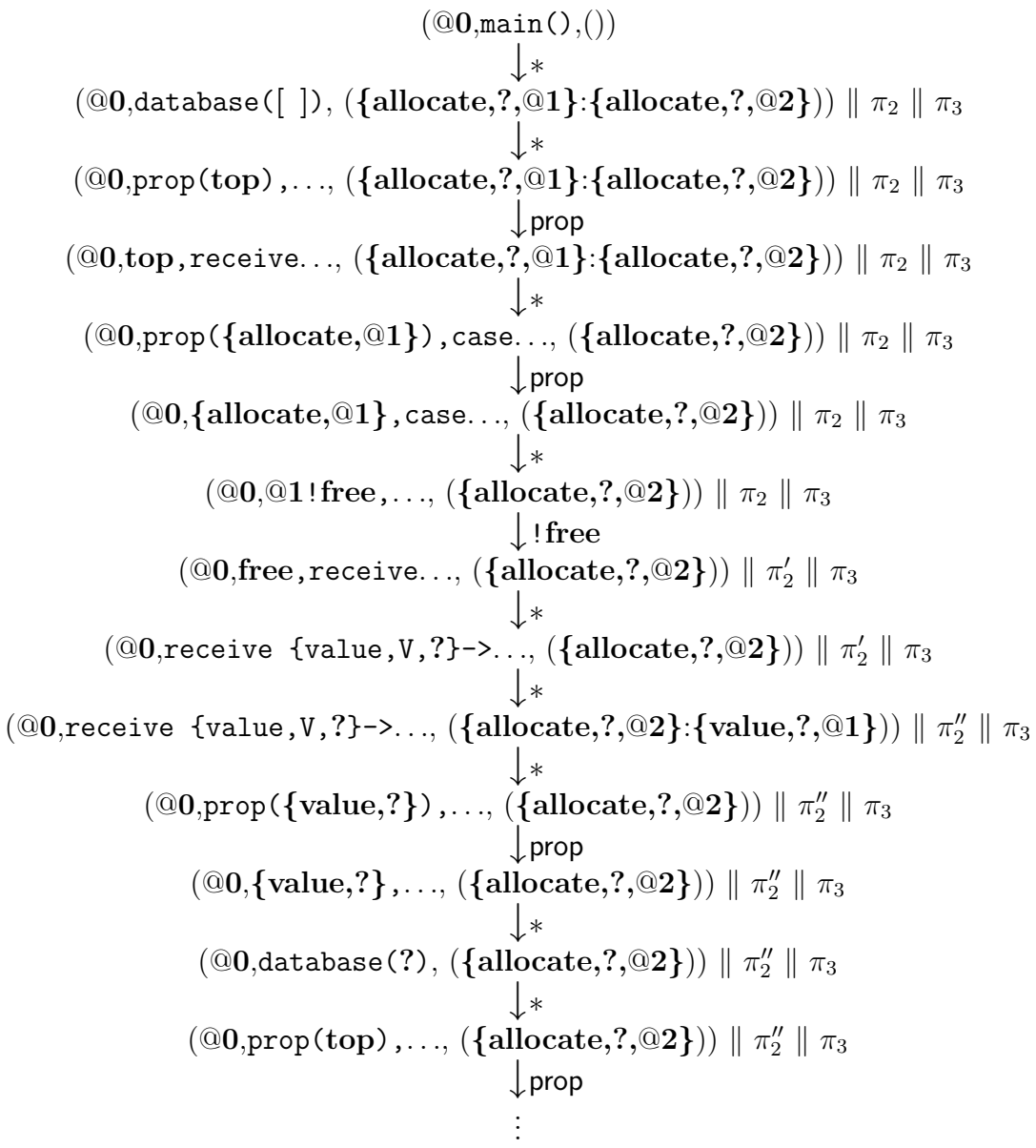


Figure 5.2: Path of the database example in the abstract interpretation $\widehat{\mathcal{A}}_2$

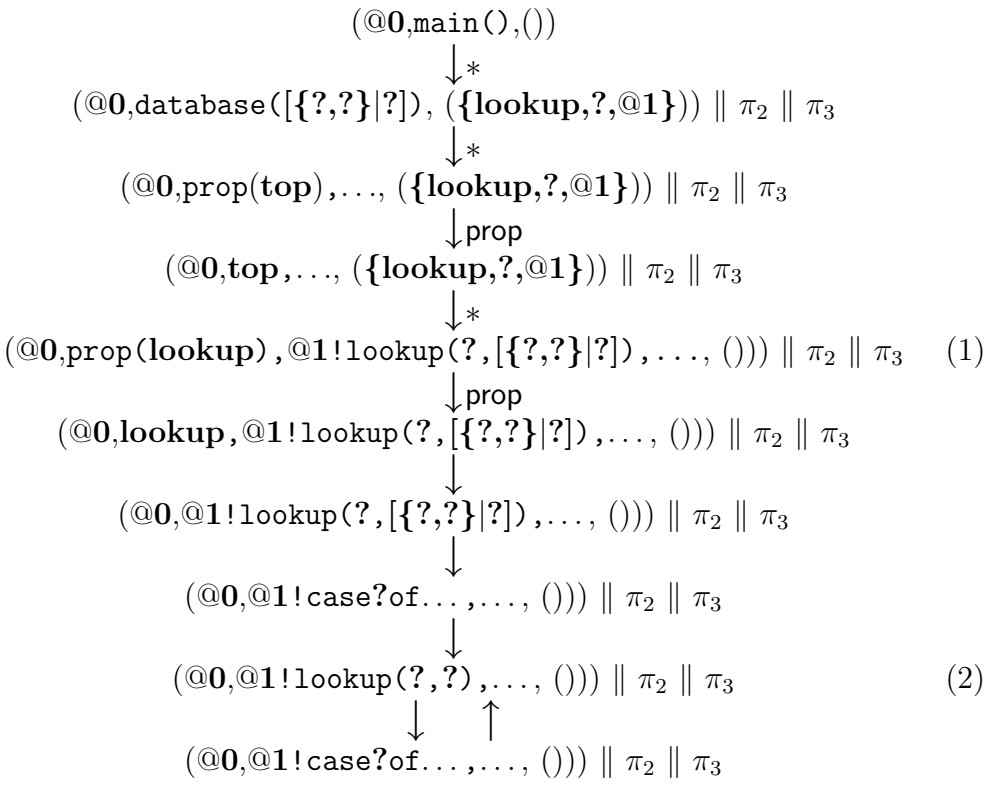


Figure 5.3: ε-loop in the database example in the abstract interpretation $\widehat{\mathcal{A}}_2$

to the corresponding client and the database finally reaches its **top** state. Using this technique we are able to prove the formula by simple finite state model checking. Our proof is only correct under the assumption that all sequential calculations terminate. The termination can be proven by different techniques, like interpretation methods [BN98], reduction orders [BN98], or critical pairs [AG00]. In practice the termination of side-effect free functions will usually be omitted because programmers always assume that functions like `append` or `lookup` terminate without any proof.

As already mentioned before, this proof also shows the absence of deadlocks in the database. In particular, there exists no deadlock between the states specified by the propositions in the formula. The absence of deadlocks for the whole database process can be specified by:

$$noDeadlock = G(\mathbf{top} \longrightarrow XF\mathbf{top})$$

From the **top** state, the database always finally reaches the **top** state again. As for liveness, we can prove this property using the abstraction $\widehat{\mathcal{A}}_2$.

5.5.2 Mutual exclusion

Another property which can automatically be verified using the abstract interpretation $\widehat{\mathcal{A}}_2$ is mutual exclusion. We want to guarantee that the process which allocates

a key also sets the corresponding value:

> If a process π allocates a key, then no other process π' sets a value before π sets a value, or the key is already allocated.

For the client processes with pids **@1** and **@2** this can be expressed as the LTL-formula

$$G(\{\mathbf{allocate}, @\mathbf{1}\} \longrightarrow (\neg\{\mathbf{value}, @\mathbf{2}\})\ U\ (\{\mathbf{value}, @\mathbf{1}\} \vee \mathbf{allocated}))$$

In our system only a finite number of pids occur. Therefore, we can express this property in general for all occurring pids:

$$\bigwedge_{\substack{\pi \in Pid \\ \pi' \neq \pi}} G\ (\{\mathbf{allocate}, \pi\} \longrightarrow (\neg\{\mathbf{value}, \pi'\})\ U\ (\{\mathbf{value}, \pi\} \vee \mathbf{allocated}))$$

This formula can be translated into a pure LTL-formula as a conjunction of all possible permutations of possible pids which satisfy the condition. This is

$$(\pi, \pi') \in \{(@\mathbf{0}, @\mathbf{1}), (@\mathbf{0}, @\mathbf{2}), (@\mathbf{1}, @\mathbf{2})\}.$$

Again, this property can automatically be proven by model checking and the finite domain abstraction $\widehat{\mathcal{A}}_2$.

5.5.3 More precise Abstractions

In the verification of distributed systems by model checking usually properties like the absence of deadlocks, live-locks or mutual exclusion are specified. In our approach it is also possible to verify more detailed properties. For example, the correct storage behavior of the database.

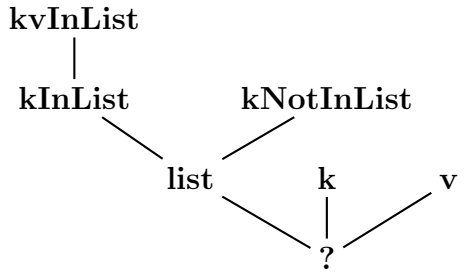
We want to prove the following property:

> If a special key-value pair `{`k`,`v`}` is inserted, then the lookup of k yields v.

The database can grow arbitrarily. Therefore, we cannot prove this property by means of a depth-k-abstraction. Instead, we define an abstraction which distinguishes lists containing the elements `{`k`,`v`}` and list which do not contain these elements. We use the abstract values:

abstract value	concrete representations
kvInList	all lists containing the pair `{`k`,`v`}`
kInList	all lists containing a pair with the key k
kNotInList	all lists containing no pair with the key k
list	all lists

the abstract values are ordered as follows:



Furthermore, we extend the abstract domain to the values of $\widehat{\mathcal{A}}_2$ to distinguish the different `allocate`, `lookup`, and `value` messages which can be sent to the database process. A presentation of the whole abstract interpretation would go beyond the scope of this thesis. Therefore, we only sketch the idea of the abstraction.

For the abstract representation of lists we define the semantics of the list constructors as:

$$\widehat{\iota}(\texttt{[]}) = \mathbf{kNotInlist}$$

$$\widehat{\iota}(\texttt{[.|.]})(\widehat{e},\widehat{l}) = \begin{cases} \mathbf{kvInList} & \text{if } \widehat{e} = \{\mathbf{k},\mathbf{v}\} \text{ and } \mathbf{list} \sqsubseteq \widehat{l} \\ \mathbf{kInList} & \text{if } \widehat{e} = \{\mathbf{k},\widehat{v}'\} \text{ where } \widehat{v}' \neq \mathbf{v} \text{ and } \mathbf{list} \sqsubseteq \widehat{l} \\ \mathbf{list} & \text{otherwise and } \mathbf{list} \sqsubseteq \widehat{l} \\ \mathbf{?} & \text{otherwise} \end{cases}$$

By means of this interpretation of constructors we extract exactly the information of the lists which is relevant for the given problem. Now we must be able to recover this information if the key k is looked up in the database. Therefore, we treat the function `lookup` as a predefined function and interpret it as follows:

$$\begin{aligned} \widehat{\iota}(\texttt{lookup})(\mathbf{k},\mathbf{kvInList}) &= \{\mathbf{succ},\mathbf{v}\} \\ \widehat{\iota}(\texttt{lookup})(\mathbf{k},\mathbf{kInList}) &= \{\mathbf{succ},\mathbf{?}\} \\ \widehat{\iota}(\texttt{lookup})(\mathbf{k},\mathbf{kNotInList}) &= \mathbf{fail} \\ \widehat{\iota}(\texttt{lookup})(\mathbf{k},\mathbf{list}) &= \mathbf{?} \\ \widehat{\iota}(\texttt{lookup})(\widehat{k},\widehat{l}) &= \mathbf{?} \quad \text{, for all } \widehat{k} \neq \mathbf{k} \end{aligned}$$

For this interpretation we have to prove property (P1). This is not difficult but technical: a structural induction over lists. Proofs like this often occur in the definition of an abstract interpretation. Therefore, our framework defines a good possibility for the integration of software verification by theorem proving and model checking. Theorem provers like Isabelle/HOL [Pau89, Pau93] are very powerful in proving properties of (structural) inductive functions, like `lookup`. Especially for these functions automatisation of proofs is a succesful field of research [Lys94, GG88]. The proposed properties of our framework define a good abstraction layer. They are proven by a theorem prover. Then the power of model checking is the verification of the concurrent behavior of the system. The interaction of the processes with respect to interleaving can be verified automatically by model checking. For these concurrent parts of the system model checking is more powerful than theorem proving because of the state space explosion problem. The interleaving semantics yields very large state spaces which make verification by theorem proving expensive. An automatization is difficult.

The combination with theorem provers does not belong to the core of this thesis. Therefore, we just present the idea. The investigation of their integration is possible future work.

Using the defined abstract interpretation it is now possible to prove the property. Therefore, we use more precise propositions of the program: `value` propositions are extended to actual keys and values and `lookup` propositions to actual keys. Furthermore, we add a proposition to the client which represents the value received from

the database. These propositions make it possible to prove that after allocating the key **k** and adding the value **v** a lookup of **k** yields the value **v**. In LTL this can be formalized as

$$G\ (\{\mathbf{value}, \mathbf{k}, \mathbf{v}, \mathbf{?}\} \longrightarrow G\ (\{\mathbf{lookup}, \mathbf{k}, \mathbf{v}\} \longrightarrow F\ \{\mathbf{succ}, \mathbf{v}\}))$$

Although this proof is possible, our approach was intentionally not designed to prove properties like this one. Usually, one will specify properties about the concurrent behavior of a system, like the absence of deadlocks. However, this example shows that our approach can also handle these more detailed properties.

5.6 Fairness Properties

So far, we have only considered formulas for simple liveness and safeness properties. Nevertheless, LTL has more expressive power. It is also possible to express more complicated properties like fairness in our approach. We motivate the practical use of fairness in a simpler example. Then we sketch how fairness can be used in our database example.

Example 5.15
Three processes are executed concurrently. They are connected to each other in a ring and every process receives a message (a number) and forwards the incremented message to its successor.

```
main() -> P1=spawn(loop,[self]),
          P2=spawn(loop,[P1]),
          P2!42,
          loop(P2).

loop(P) -> prop({main,self}),
           receive
             X -> P!X+1,
                  loop(p)
           end.
```

We want to prove that the flow of messages in this system guarantees a fair execution: No process is overtaken infinitely often. To prove this, the function `loop` contains the proposition `{main,self}`. It indicates that a process reaches its main state. To distinguish which process reaches its main state, the proposition also contains the pid of the processes. The system behaves fairly if for all processes this main proposition is valid infinitely often. This can be easily expressed by the LTL formula F^∞:

$$\bigwedge_{\pi \in \{@\mathbf{0}, @\mathbf{1}, @\mathbf{2}\}} F^\infty\ \{\mathbf{main}, \pi\}$$

Again we can prove this property using the finite abstract domain $\widehat{\mathcal{A}}_2$. However, this specification of fairness is not fully correct. It could be possible that a process stays in

its main state for ever. Then the formula would be satisfied but this is not a fair path because another process overtakes this process infinitely often. Therefore, we must refine our specification. We claim that also infinitely often the proposition **{main,**π**}** does not hold.

$$\mathit{fair} := \bigwedge_{\pi \in \{@\mathbf{0},@\mathbf{1},@\mathbf{2}\}} (F^{\infty}\ \mathbf{\{main,}\pi\mathbf{\}}) \wedge (F^{\infty}\ \neg\mathbf{\{main,}\pi\mathbf{\}})$$

This formula expresses the fairness of the system and for Example 5.15 it can be proven using the finite domain abstract $\widehat{\mathcal{A}}_2$.

Usually, we do not prove the fairness of a system. Instead we often assume a fair scheduling. Under this assumption, we want to prove other properties. The unfair paths of a system which occur in the formal semantics shall not be considered. They are irrelevant for the execution of the system because all schedulers of the Erlang runtime systems are fair.

This fairness assumption can easily be expressed in LTL by means of the implication.

$$\mathit{fair} \longrightarrow \varphi$$

For unfair paths the formula is trivially satisfied. Only for fair paths φ must be checked. For fairness conditions there also exist more efficient model checking algorithms in which the fairness assumption is already integrated in the model checker. No fairness condition is needed in the formula. It is even possible to use CTL model checking extended with fairness for the verification of arbitrary LTL formulas [CGH94, CGH97].

As an example for the use of this fairness assumption consider a system consisting of two databases and a client for each database. The operational semantics of this system contains paths in which only one database and its client are evaluated. On this path properties like *alwaysTop* do not hold for the other database.

For two databases we must modify the proposition `top` to distinguish the two databases. The easiest way is to add the pid of the database to the proposition (`prop({top,self})`). Then we also distinguish the property *alwaysTop* with respect to the pids of the databases. Adding the fairness assumption we only consider the fair paths. We can prove the property *alwaysTop* for both databases.

In this section we have defined *unconditional* fairness. There exist further, more difficult fairness properties like weak or strong fairness [Eme90]. These fairness properties can also be integrated in our approach.

Chapter 6

Extensions and Optimizations

6.1 A Simplified Framework

In Chapter 4 we have defined a framework for abstract interpretations of Core Erlang programs which can be used for the automated verification by model checking. The abstract interpretation requires five properties which relate the abstract interpretation to the concrete interpretation. This can make the definition of an abstract interpretation expensive. The simplest way to fulfill the properties for casematch and mbmatch is full non-determinism. All possible matches are collected as in the example abstraction $\widehat{\mathcal{A}}_k$ in Section 4.3.2. However, we can be more precise in the abstraction and avoid too much nondeterminism. Consider the following program and the example abstraction $\widehat{\mathcal{A}}_k$.

Example 6.1

```
f(X) -> case X of
          {succ,V} -> V;
          Y -> fail
        end.
```

We assume that this function is applied to the abstract value **{succ,?}**. It is irrefutable that the first matching succeeds, although we have abstracted from the second component of the tuple. Hence, it is superfluous to perform the second matching in the abstract semantics, as it is the case in $\widehat{\mathcal{A}}_k$.

In the abstraction of matching we have already distinguished two different kinds of matches:

- Irrefutable matches (e.g., variables match everything): It is not necessary to consider the subsequent patterns.

- Possible matches (e.g., it is possible that a pattern matches **?**): We cannot decide if the concrete evaluation matches at this point due to our abstraction. Hence, it is possible that this pattern matches or it does not match and the subsequent patterns match as well.

Using this information from matching, we can define very precise abstractions for casematch and mbmatch. If the matching of a pattern against a value is irrefutable, then all subsequent patterns need not be considered.

Defintion 6.2 (Simplified Abstract Interpretation)
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ be an abstract interpretation where $\widehat{\iota}$ is a family of interpretation functions for $\mathcal{F}$, $\mathcal{C}$, matching, and the pid representation of a value which fulfills the properties (P1), (P2), and (P5). Simplified interpretations for casematch and mbmatch are defined as:

$$\begin{aligned}\widehat{\iota}(\mathsf{casematch})((p_1,\ldots,p_n),v) = \{(i,\sigma) \mid\ & \widehat{\iota}(\mathsf{match})(p_i,v) = (\_,\sigma) \text{ and} \\ & \widehat{\iota}(\mathsf{match})(p_j,v) \neq (\mathsf{Irref},\_)\ \forall j<i\} \\ \cup \begin{cases}\{\mathsf{Fail}\} & \text{, if } \widehat{\iota}(\mathsf{match})(\widehat{p}_i,\widehat{v}) \neq (\mathsf{Irref},\_)\ \ \forall 1 \leq i \leq n \\ \emptyset & \text{, otherwise}\end{cases}\end{aligned}$$

$$\begin{aligned}&\widehat{\iota}(\mathsf{mbmatch})((p_1,\ldots,p_n),(v_1,\ldots,v_m)) = \\ &\quad \{(i,j,\sigma) \mid \widehat{\iota}(\mathsf{match})(p_i,v_j) = (\_,\sigma) \text{ and} \\ &\qquad \widehat{\iota}(\mathsf{match})(p_k,v_l) \neq (\mathsf{Irref},\_)\ \forall 1 \leq k \leq n, l < j \text{ and} \\ &\qquad \widehat{\iota}(\mathsf{match})(p_k,v_j) \neq (\mathsf{Irref},\_)\ \forall k < i\} \\ &\cup \begin{cases}\{\mathsf{Fail}\} & \text{, if } \widehat{\iota}(\mathsf{match})(\widehat{p}_i,\widehat{v}_j) \neq (\mathsf{Irref},\_)\ \ \forall 1 \leq i \leq n, 1 \leq j \leq m \\ \emptyset & \text{, otherwise}\end{cases}\end{aligned}$$ ◁

These simplified abstractions are safe with respect to the SOS. To see this, we have to prove that they fulfill the properties (P3) and (P4). The safeness with respect to the SOS follows by Theorem 4.11.

Lemma 6.3
Let $\widehat{\mathcal{A}}$ be an abstract interpretation and $\widehat{\iota}(\mathsf{match})$ fulfills property (P2)*. Then $\widehat{\mathcal{A}}$ also fulfills the property:*

(P2') For all $\widetilde{p} \in \widehat{Pat}, \widetilde{v} \in \widehat{A}$ it holds that:
if $\mathsf{match}(\widetilde{p},\widetilde{v}) = (\mathsf{Irref},\sigma)$,
then for all $p \in Pat$ where $\widetilde{p} \sqsubseteq \alpha(p)$ and for all $v \in T_{\mathcal{C}}(Pid)$ where $\widetilde{v} \sqsubseteq \alpha(v)$ is $\mathsf{match}(p,v) \neq \mathsf{Fail}$.

Proof: Assume $\mathsf{match}(\widetilde{p},\widetilde{v}) = (\mathsf{Irref},\sigma)$ and $\exists p \in Pat$ with $\widetilde{p} \sqsubseteq \alpha(p)$ or $\exists v \in T_{\mathcal{C}}(Pid)$ with $\widetilde{v} \sqsubseteq \alpha(v)$ and $\mathsf{match}(p,v) = \mathsf{Fail}$. This is a contradiction to (P2b). □

Lemma 6.4 (The Simplified Abstraction of casematch fulfills (P3))
A simplified abstract interpretation $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ fulfills the property (P3):

(P3) For all $p_1, \ldots, p_n \in Pat, v \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p_j} \sqsubseteq \alpha(p_j)$ and $\widetilde{v} \sqsubseteq \alpha(v)$ it holds that:
  a) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = (i, \rho)$,
  then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$ with $(i, \widetilde{\rho}) \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p_1}, \ldots, \widetilde{p_n}), \widetilde{v})$.
  b) if $\mathsf{casematch}((p_1, \ldots, p_n), v) = \mathsf{Fail}$,
  then $\mathsf{Fail} \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((\widetilde{p_1}, \ldots, \widetilde{p_n}), \widetilde{v})$.

Proof:

a) A simple induction over the number of patterns n.

b) Follows directly from Lemma 6.3 and the definition of $\mathsf{casematch}_{\widehat{\mathcal{A}}}$. □

Analogously we can conclude the corresponding lemma for $\mathsf{mbmatch}_{\widehat{\mathcal{A}}}$ and (P4).

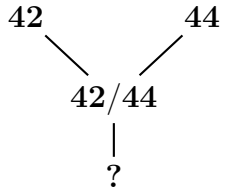
Lemma 6.5 (The Simplified Abstraction of mbmatch fulfills (P4))
A simplified abstract interpretation $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ *fulfills the property* (P4):

(P4) For all $p_1, \ldots, p_n \in Pat, v_1, \ldots, v_u \in T_{\mathcal{C}}(Pid)$ and for all $\widetilde{p_j} \sqsubseteq \alpha(p_j)$ and $\widetilde{v_k} \sqsubseteq \alpha(v_k)$ it holds that:
  a) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = (i, j, \rho)$,
  then there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$
  with $(i, j, \widetilde{\rho}) \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p_1}, \ldots, \widetilde{p_n}), (\widetilde{v_1}, \ldots, \widetilde{v_u}))$.
  b) if $\mathsf{mbmatch}((p_1, \ldots, p_n), (v_1, \ldots, v_u)) = \mathsf{Fail}$,
  then $\mathsf{Fail} \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((\widetilde{p_1}, \ldots, \widetilde{p_n}), (\widetilde{v_1}, \ldots, \widetilde{v_u}))$.

For many practical applications this simplified framework can be used. On top of $\widehat{\iota}(\mathsf{match})$ good and safe abstractions for casematch and mbmatch are predefined. Therefore, the following question raises: Why didn't we directly use the abstraction in our framework? The reason is that there exist situations in which the general framework can be more precise than this simplified framework. As an example we consider the following program:

```
f(X) -> case X of
          42 -> 21;
          44 -> 22
        end.
```

For the AOS we use the abstract domain:



The abstract value **42/44** represents exactly the two concrete values **42** and **44**. We apply the function `f` to this abstract value. Both matches in the `case` expression are

only possible, because the abstract value represents both numbers. Therefore, in the simplified abstraction this `case` expression can also fail. On the other hand, in the concrete semantics one of both patterns matches. The `case` expression cannot fail.

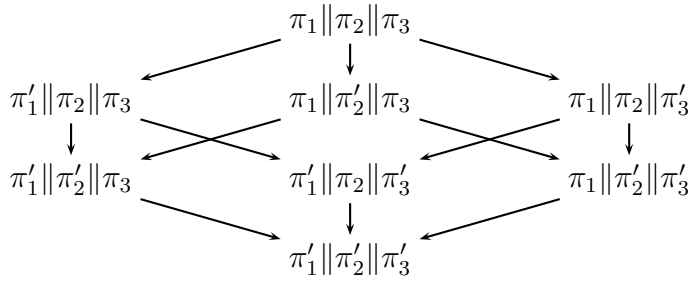
Using the general framework, it is possible to define an abstract interpretation which fulfills the property (P3), but does not yield Fail for this example. To be as expressive as possible, we have defined the general framework. However, for many abstractions the simplified framework can be used.

6.2 Reducing the State Space

From the theoretical point of view it is nice to know that model checking is decidable. Unfortunately, the transition system created for the abstract operational semantics can be very large. For real applications computers with enormous memory are needed. The main reason is the explosion of state space in the interleaving semantics. In addition to the need of much memory, runtime for the construction of the transition system or the verification of a formula increases. The state space must be constructed and the costs for the detection of cycles correspond to the absolute number of states.

The sequential parts of the evaluation are less relevant for the verification of the concurrent system and usually more than half of all transitions are labelled by ε. This is also shown in the database example. We are not interested in every state of the evaluation, like stepping through a database while looking up a key. We are only interested in the communication.

Inspecting our interleaving semantics yields the following observations: We do not only have the ε-transitions of one process. We have diamonds of ε-transitions of all processes. As an example we consider three processes which can all perform one ε-transition.



The resulting transition system has eight states which will usually not be distinguished in properties expressed in an LTL-formula. If we consider n processes which can perform m ε-transitions we get $(m+1)^n$ different states. Each of the n process occurs in $m+1$ different ways. To get an impression of the state explosion, we present the following numbers. For three processes and two ε-transitions we get 27 states and for eight processes with five ε-transitions per process we get 1679616 states. Unfortunatly, this degree of non-determinism commonly occurs in the AOS of Core Erlang programs. Furthermore, these numbers do not represent the whole state space of a system. They only represent the size of one diamond in the interleaving semantics of one evaluation. Such diamonds occur in many different places of the AOS. Therefore, a reduction of the state space is needed.

The results of the evaluations can have effects on the communication, because of branching, but the single steps in the evaluation are irrelevant. So it would be good to eliminate these sequential transitions like an ε-elimination known from automata theory [HU79]. We can use this technique here too, but we have to consider the nonterminating cycles, because they are possible paths of the system and have to be kept to be safe with respect to the SOS.

The ε-closure of a state can be defined as the smallest set of states fulfilling

$$\widehat{\varepsilon}(s) = \{s\} \cup \bigcup_{s \longrightarrow s'} \widehat{\varepsilon}(s')$$

Then the more compact transition system can be defined on the ε-closures of the states. This reduces the state space, but the states of the transition system are sets of system states now. These sets of states must be stored to detect cycles. Therefore, memory does not decrease essentially. All states of the diamond are part of the ε-closure. Only the ε-transitions in the diamonds can be dropped. We need a more compact representation of this equivalence class.

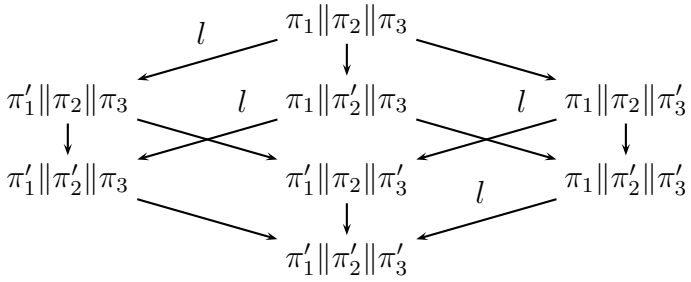
The idea is to use only the states at the end of the diamond. We do not need to store the states in between. Also for the detection of cycles this is not a problem. If a cycle yields back to one of the eliminated states, then also on the transition system constructed in the cycle this state is eliminated. We get (maybe several) cycles to the last states of the ε-closure. In our example, the whole diamond would be reduced to the state $\pi_1' \| \pi_2' \| \pi_3'$.

As an alternative, we could also represent the diamond by its first state. However, this does not match with the state propositions, as we will see later.

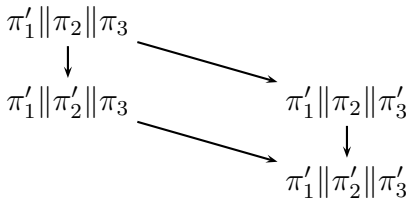
Instead of the whole ε-closure we only store the states which can perform side-effects. These are all states which have a non-ε-transition. This approach yields a reduction of the state space but it does not yield the supposed reduction:

Example 6.6

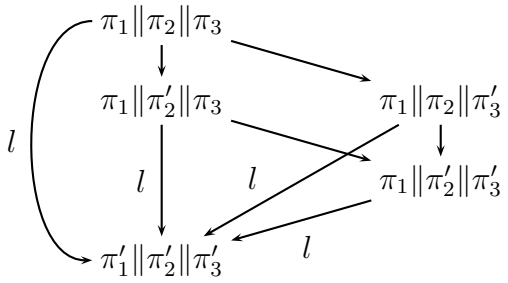
Again, we consider three parallel processes. The first process π_1 can perform a send action l and the others only one ε-transition. The AOS contains the following diamond:



Four states can perform a non-ε-transition. They cannot be eliminated. So only the small sub-diamond



can be eliminated and we get the reduced transition system:



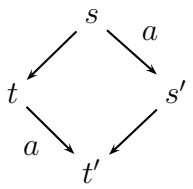
The state space is not reduced as much, as we supposed. Again, in logical specifications these states will usually not be distinguished. We would like to represent this transition system as

$$\pi_1\|\pi_2\|\pi_3 \xrightarrow{l} \pi_1'\|\pi_2'\|\pi_3'$$

This demand is meaningful considering the following lemma.

Lemma 6.7

For every abstract interpretation $\widehat{\mathcal{A}}$ *and for every state* $s \in \widehat{State}_{\widehat{\mathcal{A}}}$ *where* $s \longrightarrow_{\widehat{\mathcal{A}}} t$ *it holds that if* $s \xrightarrow{a}_{\widehat{\mathcal{A}}} s'$*, then there exists* $t' \in \widehat{State}_{\widehat{\mathcal{A}}}$ *such that*



Proof: A simple case analysis on $\longrightarrow_{\widehat{\mathcal{A}}}$. □

The AOS is locally confluent for ε-transitions. The only reason for the branching is interleaving. We need not distinguish the two paths. The idea of an optimization is the following: All possible sequential steps of the system are performed as long as possible. Only states which do not have any ε-successor are considered in our final transition system. The only exception are ε-loops which need not be eliminated. To be safe with respect to the SOS, we keep them to represent this infinite path by pure evaluations. Using this technique, we can reduce the state space of Example 6.6 and get

$$\pi_1\|\pi_2\|\pi_3 \xrightarrow{l} \pi_1'\|\pi_2'\|\pi_3'$$

We get an enormous reduction of the state space. All diamonds which contain ε-transitions are eliminated. In our example program p_{db} consisting of a database and two clients the state space is for example reduced from 5063 to 549 states.

However, we can still do better. In partial order reduction [Pel94, CGMP98], this approach is generalized. The diamonds resulting from the interleaving semantics can be eliminated for arbitrary independent actions. If we inspect the possible labels of the semantics in more detail, then we see that only send transitions have side-effects. For all other labels Lemma 6.7 holds analogously. All the other non-determinism results

from the interleaving semantics. It can be eliminated using the same technique. By this optimization the state space can be further reduced. We only preserve some of the send transitions and some loops. All other transitions are eliminated.

Furthermore, we must save the states containing different propositions. These propositions will usually occur in the formula. Hence, they distinguish relevant states for the verification. These states should not be eliminated. Fortunately, this is no problem because we can detect these states by means of prop-transitions following them. The states of the constructed transition system only have send and prop transitions. If a state can perform other actions, then these actions are performed until no other actions are possible.

We define the set of independent labels that can be eliminated:

$$\mathcal{I}_{\widehat{\mathcal{A}}} = \{?v \mid v \in \widehat{\mathcal{A}}\} \cup \{\texttt{spawn(}f\texttt{)} \mid f/n \in FS(p)\} \cup \{\varepsilon\}$$

An algorithm for the computation of the successor states can be defined as follows:

$$\begin{aligned}
&\mathsf{succs}_{\mathcal{I}_{\widehat{\mathcal{A}}}} : \mathcal{P}(\widehat{State}) \times \mathcal{P}(\widehat{State} \cup \{\mathsf{loop}\}) \times \mathcal{P}(\widehat{State}) \longrightarrow \mathcal{P}(\widehat{State} \cup \{\mathsf{loop}\}) \\
&\mathsf{succs}_{\mathcal{I}_{\widehat{\mathcal{A}}}}(S, Res, Acc) = \ \mathsf{if}\ S' = \emptyset\ \mathsf{then} \\
&\qquad\qquad\qquad\qquad\qquad\qquad Res' \\
&\qquad\qquad\qquad\qquad\quad \mathsf{else} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{succs}_{\mathcal{I}_{\widehat{\mathcal{A}}}}(S', Res', Acc \cup S') \\
&\qquad\qquad \mathsf{where}\ S' \quad = \bigcup\{t \mid s \in S\ \mathsf{and}\ s \xrightarrow{a} t\ \mathsf{and}\ a \in \mathcal{I}_{\widehat{\mathcal{A}}}\} \setminus Acc \\
&\qquad\qquad\qquad\qquad Res' = Res \cup \{s \mid s \in S\ \mathsf{and}\ \nexists t : s \xrightarrow{a} t, a \in \mathcal{I}_{\widehat{\mathcal{A}}}\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \cup \begin{cases} \{\mathsf{loop}\} & , \mathsf{if}\ S' \cap Acc \neq \emptyset \\ \emptyset & , \mathsf{otherwise} \end{cases}
\end{aligned}$$

The first argument is the set of states, from which we want to compute the successors. In the first call this set contains just one state. The second argument contains all states in which no more side-effect-free transitions can be performed. If all states are expanded, then this set is the result of the computation. In the third argument we accumulate all states which are constructed during the computation. This is needed to detect cycles and to guarantee the termination of the algorithm in this case. Finally, we add the constant loop to the result, if we detect a cycle.

Using this function, we can construct the reduced transition system. There are two possibilities where we can compute the successors by $\mathsf{succs}_{\mathcal{I}_{\widehat{\mathcal{A}}}}$. Before performing an action of $\overline{\mathcal{I}_{\widehat{\mathcal{A}}}} := \mathit{Label} \setminus \mathcal{I}_{\widehat{\mathcal{A}}}$ or after it. As standard in automata theory, we could compute them before the $\overline{\mathcal{I}_{\widehat{\mathcal{A}}}}$-actions. In this case the initial state of a transition system need not be handled in a special way. However, this approach does not compatible with state propositions, as the example in Figure 6.1 shows. We get $\mathsf{succs}_{\mathcal{I}_{\widehat{\mathcal{A}}}}(s_0) = \{s_2\}$ and then

$$s_0 \xrightarrow{\mathsf{prop}} s_4$$

This is not a good strategy to construct the reduced transition system. The state s_2 gets lost. This is wrong since this state contains a proposition (s_2 can perform a prop-action) which could be used for verification purposes.

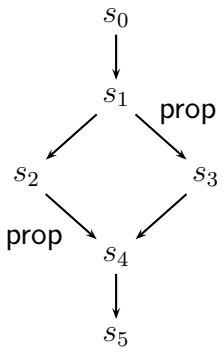


Figure 6.1: A Possible Transitionsystem

The definition of the semantics of a proposition postulates that the state from which the prop-transition can be performed fulfills a proposition. In the example these are the states s_1 and s_2. However, both states are not represented in the reduced transition system. Let the proposition P hold in these states. A property like $F\ P$ which holds in the AOS would not hold in the reduced transition system. The elimination of the state s_1 is correct because this state is the initial state of an interleaving diamond. However, we must keep the state s_2.

We can obtain this reduction of the transition system, if we compute the $\mathcal{I}_{\widehat{\mathcal{A}}}$-successors after an $\overline{\mathcal{I}_{\widehat{\mathcal{A}}}}$ transition. For the initial state we then have to compute them once and use all states of $\mathsf{succs}_{\mathcal{I}_{\widehat{\mathcal{A}}}}(s_0)$ as states for the further construction of the transition system. The $\overline{\mathcal{I}_{\widehat{\mathcal{A}}}}$ transitions then lead to the $\mathcal{I}_{\widehat{\mathcal{A}}}$-successors of the original successors. In our example we get

$$s_0 \longrightarrow s_2 \overset{\mathsf{prop}}{\longrightarrow} s_5$$

The state s_2 is represented in the reduced transition system and the property $F\ P$ is fulfilled. The reduced transition system can be constructed by means of the following function:

$$\begin{aligned}
&\mathsf{nextstates}_{\mathcal{I}_{\widehat{\mathcal{A}}}} : (\widehat{State} \cup \{\mathsf{loop}\}) \times \widehat{Label} \longrightarrow \mathcal{P}(\widehat{State} \cup \{\mathsf{loop}\}) \\
&\mathsf{nextstates}_{\mathcal{I}_{\widehat{\mathcal{A}}}}(\mathsf{loop}, \varepsilon) = \{\mathsf{loop}\} \\
&\mathsf{nextstates}_{\mathcal{I}_{\widehat{\mathcal{A}}}}(s, a) = \mathsf{succs}_{\mathcal{I}_{\widehat{\mathcal{A}}}}(S, \emptyset, S) \\
&\qquad \mathsf{where}\ S = \bigcup\{t \mid s \xrightarrow{a} t\}
\end{aligned}$$

For cycles without prop or send transitions we add a special sink loop. It only contains a cycle which is labelled by ε. Although there can exist other labels in the corresponding loops in the AOS, we do not represent them in the reduced transition system. They are not relevant for the verification. It is not possible to distinguish them in LTL, because labelled X-operators are not allowed. We can only distinguish states containing different propositions. These states are kept in the reduced transition system.

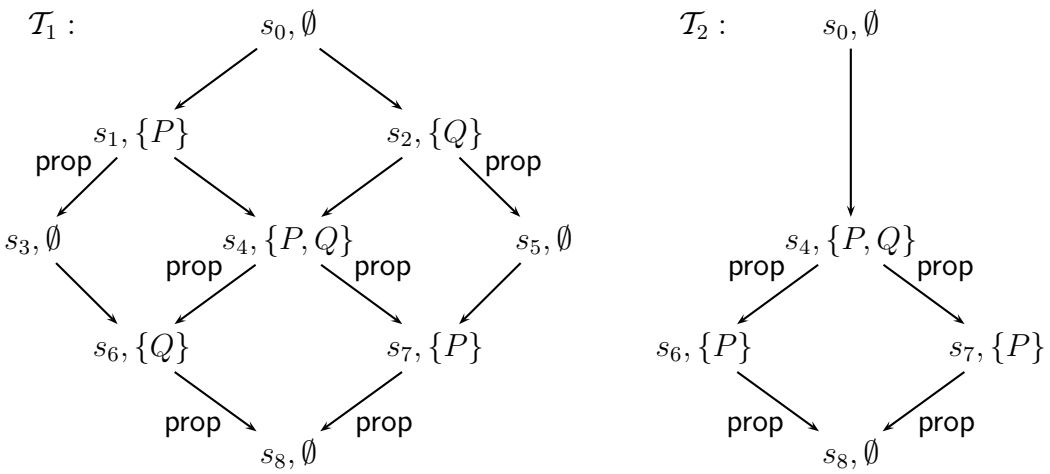


Figure 6.2: A transition system $\mathcal{T}_1$ and its reduced representation $\mathcal{T}_2$

This reduction of the AOS is not correct for arbitrary LTL-formulas as the example in Figure 6.2 shows. The transition system $\mathcal{T}_1$ presents the interleaving semantics of two processes evaluating the proposition P and Q. Its reduced representation is presented as transition system $\mathcal{T}_2$. In the figure, the set of valid propositions is added to every state.

These two transition systems do not fulfill the same LTL-formulas. The formula $X\ X\ (P \wedge Q)$ is fulfilled in $\mathcal{T}_1$ and not fulfilled in $\mathcal{T}_2$. On the other hand $X\ (P \wedge Q)$ is fulfilled in $\mathcal{T}_2$ and not in $\mathcal{T}_1$. However, in the specification of system properties the use of next is uncommon. Usually, it will be replaced by finally. Therefore, in partial order reduction LTL is used without next operator (called LTL_{-X} [CGMP98]). However, that is not sufficient. Also in LTL_{-X} we can distinguish different interleaving paths. The formula

$$(F\ (Q \wedge \neg P \wedge F\ P)) \vee (F\ (Q \wedge \neg P \wedge F\ P))$$

is fulfilled in $\mathcal{T}_1$. The sub-formula $P \wedge \neg Q$ is only fulfilled in the states s_1 and s_7. $Q \wedge \neg P$ only in s_2 and s_6. All paths of the left branch of $\mathcal{T}_1$ fulfill the first part of the disjunction and all paths of the right branch fulfill the second part. Therefore, the formula is fulfilled. On the other hand, the formula is not fulfilled in the reduced transition system. The state s_2 is eliminated. Only the state s_6 fulfills the sub-formulas $P \wedge \neg Q$. This state has no successor which fulfills Q. Hence, the first part of the disjunction is not fulfilled. Vice versa the same holds for the second part and the whole formula is not fulfilled.

This example shows that a formula which is fulfilled in the AOS can be invalid in the reduced transition system. We did not suppose this, but it is not wrong, because we would stay safe with respect to the SOS. Unfortunately, there also exist formulas which are fulfilled in the reduced transition system but not in the AOS. An example

is

$$(\neg P)\ U\ Q$$

In $\mathcal{T}_2$ all paths start with the propositions $\emptyset, \{P, Q\}$. The proposition Q is already fulfilled in the second state of the path. The only predecessor states does not contain the proposition P.

On the other hand, the transition system $\mathcal{T}_1$ contains three state paths which begin with $s_0 s_1$. In this initial part the formula is already falsified, because P is valid in s_1 and Q is not valid in the states before s_4 respectively s_6. So the reduction of the state space can yield wrong results.

There is a large theory about partial order reduction [Pel94] and its application. It is possible to decide, whether an LTL_{-X} formula is trace invariant. This is a large field of research and we do not go into details here. We perform the reduction of the state space, because this is needed to use this approach also for non-trivial programs. Most specifications of system properties expressed in LTL are trace independent. They can be checked by LTL model checking on the reduced transition system. We leave it to the user to show that the specified formula is trace invariant with respect to our reduction.

Chapter 7

Abstraction of Recursive Function Calls

In the last chapters we have developed an approach for the formal verification of Core Erlang. We have defined hierarchical Core Erlang programs, a subclass of Core Erlang programs for which a finite domain abstraction yields a finite transition system[1]. However, many Erlang programs are not hierarchical. As an example, we consider a more efficient implementation of our database.

Example 7.1
We now use an implementation of the database process by means of sorted lists. The only modification of the main loop of the database is the call of the function `insert` instead of adding the new key/value tuple at the front of the list:

```
dataBase(L) ->
  receive
    {allocate,Key,P} ->
      case lookup(Key,L) of
        fail -> P!free,
                receive
                  {value,V,P} ->
                    dataBase(insert(Key,V,L))
                end;
        {succ,V} -> P!allocated,
                    dataBase(L)
      end;
```

[1]With the restriction that only finitely many processes may be created and only finite parts of the mailboxes can be used.

```
     {lookup,Key,P} -> P!lookup(Key,L),
                       dataBase(L)
   end.
```

New elements are inserted into the database with respect to an ordering on the keys:

```
insert(KN,VN,L) ->
   case L of
      []           -> [{KN,VN}];
      [{K,V}|L1] ->
         case K<KN of
            true  -> [{K,V}|insert(KN,VN,L1)];
            false -> [{KN,VN}|L]
         end
   end.
```

Then the `lookup` function can be optimized. Only the elements which are less than or equal to the key must be compared with the key. In the averaged case, only half of the list has to be used. Every insertion of a new value also needs a lookup if the new key already exists. Therefore, together with lookup requests more lookup operations are performed on the database. The implementation by means of a sorted list is more efficient.

```
lookup(K,[{K,V}|R])  -> {value,V};
lookup(K,[(K1,_)|R]) -> case K<K1 of
                           true  -> lookup(K,R);
                           false -> fail
                        end;
lookup(Key,[])       -> fail.
```

A more efficient version could be implemented by (balanced) sorted trees but for our purpose this implementation suffices.

The problem of this implementation is that this program is not a hierarchical Core Erlang program. The recursive call in the definition of the function `insert` is not a tail recursive call. Also for finite domain abstractions the AOS yields an infinite transition system. We illustrate this by a simpler example.

Example 7.2
Consider the following Core Erlang program:

```
main() -> f(42).

f(X) -> f(f(X)).
```

The smallest possible abstract domain contains only the element **?** which represents all possible values. Using this abstract domain the abstract semantics of the program only contains the infinite path:

$$(@\mathbf{1}, \texttt{main()}, ()) \longrightarrow (@\mathbf{1}, \texttt{f(42)}, ()) \longrightarrow (@\mathbf{1}, \texttt{f(?)}, ()) \longrightarrow (@\mathbf{1}, \texttt{f(f(?))}, ())$$
$$\longrightarrow \ldots \longrightarrow (@\mathbf{1}, \texttt{f}^{n}\texttt{(?)}, ()) \longrightarrow (@\mathbf{1}, \texttt{f}^{n+1}\texttt{(?)}, ()) \longrightarrow \ldots$$

which contains infinitely many different states. This abstract semantics is safe with respect to the operational semantics in the sense that all paths of the SOS are represented. However, we cannot prove properties for this abstract semantics using simple model checking algorithms because it has an infinite state space.

Many other function like the `append`, `length`, or `insert` in the database example produce infinite transition systems for the abstract semantics over a finite domain as well. The definition of an equivalent hierarchical Core Erlang program is difficult. A tail recursive version of a function, if it exists, can be very complicated and inefficient.

The source of the problem is the context-free structure of function calls. For special classes of context-free transition systems it has been shown that model checking is decidable [BS92, BE97]. It seems that these theoretical results could be used here. Unfortunately, we do not have just one context-free transition system. We have several of them in several processes which can communicate with each other. Hence, we have several stacks which can exchange data. Using two of these stacks it is possible to show that model checking is undecidable even for very simple finite domain abstractions.

7.1 Simulation of Turing Machines

A process which behaves like a stack can be defined as:

```
stack(P) -> receive
               get -> get;
               X   -> stack(P),
                      P!X,
                      stack(P)
            end.
```

The state of the process is the pid of the process to which it sends the stored values. If the process receives the message `get`, then it returns the last element written to the stack. All other messages are stored in the stack by performing the send action after recursively behaving like a stack. The stack is implemented by non-tail recursive calls. No data structures are used to store the values of the stack.

For a convenient modification of the stack we define the operations `push` and `pop`.

```
push(St,V) -> St!V.

pop(St) -> St!get,
           receive
              X -> X
           end.
```

This stack terminates if a process tries to pop a message from the empty stack. We want to implement a Turing machine (*TM*) by means of two stacks to show that model checking is undecidable for very simple finite domain abstractions. In the semantics of a TM the tape has to be extended dynamically by blanks. This allows a finite representation of the configurations during the execution. Therefore, we extend the stack using a function which yields the atom `blank` if `get` is applied to the empty stack. We use the same technique as for the stack and define it by a non-tail recursive call.

```
blankStack(P) -> stack(P),
                 P!blank,
                 blankStack(P).
```

Using two of these stacks we can implement a TM. We add a third process for the finite control of the TM. This process holds the actually scanned symbol and shifts symbols from one stack to the other if the TM moves.

Let $\mathcal{M} = \langle Q, \Gamma, \delta, q_0, F\rangle$ be a deterministic Turing machine with the transition function $\delta : Q \setminus F \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathsf{l}, \mathsf{r}\}$. Then we can define the control process of its implementation as follows:
For all $q \in F$:

```
q(SL,SR,_) -> 42.
```

for all $q \in Q \setminus F$ where $\delta(q, a) = (p, b, \mathsf{r})$:

```
q(SL,SR,a) -> push(SL,b),
              A = pop(SR),
              p(SL,SR,A).
```

and for all $q \in Q \setminus F$ where $\delta(q, a) = (p, b, \mathsf{l})$:

```
q(SL,SR,a) -> push(SR,b),
              A = pop(SL),
              p(SL,SR,A).
```

The Turing machine is started by the following function:

```
start() -> SL=spawn(blankStack,[self()]),
           SR=spawn(blankStack,[self()]),
           writeInputToStack(SR),
           A = pop(SR),
           q0(SL,SR,A),
           prop(terminated),
           outputStack(SR).
```

The values used by this TM are the alphabet Γ of the TM, the three pids of the processes, and the atom `get`. Due to this fact we can define a finite domain abstraction which exactly consists of these values. However, for this abstraction the program still simulates the TM.

We have added the proposition `terminated` to the program. Using this proposition we can express the property "*Finally the Turing machine terminates*" in LTL as

$$F \textbf{ terminated}$$

An algorithm which decides this formula must decide the termination of the TM. This is a contradiction to the halting problem.

To decide properties for non-hierarchical Core Erlang programs we need an abstraction of the context-free structure to a finite model or a context-free model which results from only one context-free process. The second possibility seems to be very complicated for practice and it is not clear from which process the context-free structure should be kept. Therefore, we abstract a finite model. The abstraction must contain all paths of the context-free structure because we want to prove properties of the program by model checking for linear time logic (LTL).

7.2 Graph Semantics

In the semantics of Core Erlang as defined in Chapter 3 we cannot detect which parts of an Erlang term belong to which function call. If a defined function is applied, then the right-hand side vanishes in the context in which it is called. We cannot detect where it ends. We do not have an explicit call stack. To make these calls and the corresponding returns more visible we move somewhat closer to the implementation. We split an Erlang term into a stack of Erlang contexts and a term which is actually evaluated. If a function is called, then its context is stored on the stack. The corresponding right-hand side is the next term which has to be evaluated. If the actual value is ground (it cannot be evaluated anymore), then the next context is popped from the stack and the value is put in the hole. The evaluation continues for this Erlang term. These stack representations of evaluation terms are defined as

$$SR(S) := \mathcal{E}(S) \times (FS(p) \times \mathcal{EC}(S))^*$$

where S is the set of possible values. The stack also contains the name of the function which was called when this context was pushed. This is superfluous in the graph representation but we will later use this information for our abstraction.

This technique could be applied to the Erlang semantics. However, in the semantics of Core Erlang all processes act interleaved and the critical calls and returns of a process cannot be identified and modified so easily. Here we only represent the behavior of one process. This simplifies the analysis. We define a pre-compilation which transforms a Core Erlang function into a transition system which describes the behavior of a process starting as this function. The idea is that all actions are interpreted freely. The arcs in this transition system are labelled by the behavior/actions the process may perform. The states are labelled by the Erlang terms which have to be evaluated. The only difference to the SOS is that variables occur in the Core Erlang terms instead of values. In the semantics they will be bound to values. Hence, we can handle variables in our free interpretation as values, too. The position where

1. $(E[a, e], W) \xrightarrow{\varepsilon} (E[e], W)$
2. $(E[a\,\texttt{!}\,b], W) \xrightarrow{a\,\texttt{!}\,b} (E[b], W)$
3. $(E[\texttt{self}], W) \xrightarrow{Y = \texttt{self}} (E[Y], W)$ where $Y \notin Var(E)$
4. $(E[p\texttt{=}a], W) \xrightarrow{p = a} (E[a], W)$
5. $(E[\texttt{receive}\ p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_n\texttt{->}e_n\ \texttt{end}], W) \xrightarrow{(i, \texttt{?}p_i)} (E[e_i], W) \quad \forall 1 \leq i \leq n$
6. $(E[\texttt{case}\ a\ \texttt{of}\ p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_n\texttt{->}e_n\ \texttt{end}], W) \xrightarrow{(i, p_i = a)} (E[e_i], W) \quad \forall 1 \leq i \leq n$
7. $(E[\phi(a_1, \ldots, a_n)], W) \xrightarrow{Y = \phi(a_1, \ldots, a_n)} (E[Y], W)$ where $Y \notin Var(E)$
8. $(E[\texttt{spawn(f,}a)], W) \xrightarrow{Y = \texttt{spawn}(f, a)} (E[Y], W)$ where $Y \notin Var(E)$
9. $(f(\overline{a}), W) \xrightarrow{\mathsf{lc}:\overline{X} = \overline{a}} (e_f, W)$ where $f(\overline{X})\texttt{->}e_f\,\texttt{.} \in p$
10. $(E[f(\overline{a})], W) \xrightarrow{\mathsf{c}:\overline{X} = \overline{a}} (e_f, (f, E)W)$ where $f(\overline{X})\texttt{->}e_f\,\texttt{.} \in p$ and $E \neq [\,]$
11. $(a, (f, E)W) \xrightarrow{\mathsf{r}:Y = a} (E[Y], W)$ where $a \in T_{\mathcal{C}}(Vars)$ and $Y \notin Var(E)$

Figure 7.1: Graph Representation using a Stack

the next evaluation takes place is independent of the concrete variable bindings. The result is the relation

$$\longrightarrow \subseteq SR(T_{\mathcal{C}}(Var)) \times Act \times SR(T_{\mathcal{C}}(Var))$$

defined in Figure 7.1[2]. The set of all actions Act should be clear from the figure. The first eight rules just perform the free interpretation of the actions. In the rules for `receive` and `case` we have to consider branching. The correct order of the patterns is important. Therefore, we number the patterns in the corresponding arcs and preserve their order. If the result of an action can be used in subsequent states, then we introduce a new variable Y. The result of the action is bound to Y and the redex is replaced by Y. The call of a function yields a new stack frame for the context in which the function is called (10). In the SOS we also have to push the variable bindings to a runtime stack at this point and proceed using the bindings of the parameters of the

[2]For this relation we use the same symbol as for the AOS. However, they can easily be distinguished because the AOS is always indexed by the abstract interpretation: $\longrightarrow_{\widehat{\mathcal{A}}}$.

called function f. This is retained by the transition label $\mathsf{c}{:}\overline{X} = \overline{a}$ [3]. If a function is called in an empty context, then we use tail recursion optimization (9). Consider the function

```
main() -> main().
```

In the semantics without stacks, the transition system contains only one state (the initial state). This shall also be guaranteed in our graph representation using stacks. Therefore, if a function call has an empty context, then we do not extend the stack. Instead, we just jump by a last call to the corresponding right-hand side. In the SOS the old variable bindings are replaced by the new bindings of the parameters of the function to the arguments that f is applied to.

If we have no evaluation context anymore, in other words, the Core Erlang term is a constructor term over variables, then we have to return to the last context (11). We cannot simply copy the value a into the hole because a could contain variables which also occur in E. In the SOS these variables are usually bound to different values. Hence, we introduce a new variable Y which does not occur in E and bind this variable to the result of the evaluation which is a.

The semantics over this graph representation (*GOS*) can be defined as the AOS respectively SOS except that we replace the evaluation term by a state in the graph representation and a corresponding environment representing the variable bindings. This environment consists of a substitution for the actual variable bindings $Subst : Var \longrightarrow \widehat{A}$ and a stack of substitutions $Subst^*$ for the frames on the call stack.

$$\begin{aligned}
&\rightsquigarrow_{\widehat{\mathcal{A}}} \subseteq \widetilde{State} \times \widehat{Label} \times \widetilde{State} \text{ where}\\
&\widetilde{State} := \mathcal{P}_{fin}(\widetilde{Proc}),\\
&\widetilde{Proc} := Pid \times SR(T_{\mathcal{C}}(Var)) \times \widetilde{Subst} \times \widetilde{Subst}^* \times \widehat{Mb}\\
&\widehat{Mb} := \widehat{A}^*\\
&\widehat{Label} := \{!\widetilde{v} \mid \widetilde{v} \in \widehat{A}\} \cup \{?\widetilde{v} \mid \widetilde{v} \in \widehat{A}\} \cup\\
&\qquad\qquad \{\texttt{spawn(}f\texttt{)} \mid f/n \in FS(p)\} \cup \{\varepsilon\}
\end{aligned}$$

$\rightsquigarrow_{\widehat{\mathcal{A}}}$ is defined in dependence of the labelings of $\longrightarrow$. It is defined in Figures 7.2 – 7.5. The rules just bind the actions by means of the abstract interpretation. In the case of branching we have to consider all successors of s. For their evaluation we use the function $\mathsf{allSuccs} : SR(T_{\mathcal{C}}(Var)) \longrightarrow \mathcal{P}_{fin}(SR(T_{\mathcal{C}}(Var)))$:

$$\mathsf{allSuccs}(s) := \{t \mid s \xrightarrow{l} t\}$$

If a new process is spawned, then this process starts as the initial state of the graph representation of the spawned function. The function $\mathsf{init} : FS(p) \longrightarrow (SR(T_{\mathcal{C}}(Var)) \times Var^*)$ yields this state and also the variables which have to be bound in the function call:

$$\mathsf{init}(f) := ((e_f, \varepsilon), \overline{X}), \text{ if } f(\overline{X})\texttt{->}e_f\texttt{.} \in p$$

[3] We write $\overline{X}$ as an abbreviation for $X_1, \ldots, X_n$, $\overline{a}$ for $[a_1, \ldots, a_n]$, and $\overline{X} = \overline{a}$ for $X_1 = a_1, \ldots, X_n = a_n$. n will be clear from the context.

$$\frac{s \xrightarrow{\varepsilon} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma, \Sigma, \mu)}$$

$$\frac{s \xrightarrow{Y = \phi(a_1, \ldots, a_n)} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma[Y/\phi_{\mathcal{A}}(\sigma(a_1), \ldots, \sigma(a_n))], \Sigma, \mu)}$$

$$\frac{s \xrightarrow{Y = \texttt{self}} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi', (\pi, s', \sigma[Y/\pi], \Sigma, \mu)}$$

Figure 7.2: Graph Semantics – Sequential Evaluation

$$\frac{s \xrightarrow{p = a} s' \quad \text{and} \quad \text{match}_{\widehat{\mathcal{A}}}(p, \sigma(a)) = \rho}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma \uplus \rho, \Sigma, \mu)}$$

$$\frac{\begin{array}{c}\mathsf{allSuccs}(s) = \{s_1, \ldots, s_m\} \text{ and } s \xrightarrow{(1, p_1 = a_1)} s_1, \ldots, s \xrightarrow{(m, p_m = a_m)} s_m \\ \text{and } (i, \rho) \in \text{casematch}_{\widehat{\mathcal{A}}}((p_1, \ldots, p_m), v)\end{array}}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s_i, \sigma \uplus \rho, \Sigma, \mu)}$$

$$\frac{\begin{array}{c}\mathsf{allSuccs}(s) = \{s_1, \ldots, s_m\} \text{ and } s \xrightarrow{(1, ?p_1)} s_1, \ldots, s \xrightarrow{(m, ?p_m)} s_m \\ \text{and } (i, j, \rho) \in \mathsf{mbmatch}_{\widehat{\mathcal{A}}}((p_1, \ldots, p_m), (v_1, \ldots, v_u))\end{array}}{\Pi, (\pi, s, \sigma, \Sigma, (v_1, \ldots, v_j, \ldots, v_u)) \overset{?v_j}{\rightsquigarrow}_{\widehat{\mathcal{A}}} \Pi, (\pi, s_i, \sigma \uplus \rho, \Sigma, (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_u))}$$

Figure 7.3: Graph Semantics – Matching

$$\frac{s \xrightarrow{Y = \texttt{spawn}(f,a)} s' \text{ and } \mathsf{init}(f) = (s_f, (X_1, \ldots, X_n)) \text{ and } \sigma(a) = [v_1, \ldots, v_n]}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \overset{\texttt{spawn}(f)}{\rightsquigarrow}_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma[Y/\pi'], \Sigma, \mu), (\pi', s_f, [X_1/v_1, \ldots X_n/v_n], \varepsilon, ())}$$

$$\frac{s \xrightarrow{a!b} s' \quad \text{and} \quad \pi' \in \mathrm{pid}_{\widehat{\mathcal{A}}}(\sigma(a))}{\Pi, (\pi, s, \sigma, \Sigma, \mu)(\pi', t, \sigma', \Sigma', \mu') \overset{!\,\sigma(b)}{\rightsquigarrow}_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma, \Sigma, \mu)(\pi', t, \sigma', \Sigma', \mu' : \sigma(b))}$$

Figure 7.4: Graph Semantics – Concurrent Evaluation

$$\frac{s \xrightarrow{\mathsf{c}:\overline{X} = \overline{a}} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', [\overline{X}/\overline{a}], \sigma : \Sigma, \mu)}$$

$$\frac{s \xrightarrow{\mathsf{lc}:\overline{X} = \overline{a}} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', [\overline{X}/\overline{a}], \Sigma, \mu)}$$

$$\frac{s \xrightarrow{\mathsf{r}:Y = a} s'}{\Pi, (\pi, s, \sigma, (\sigma' : \Sigma), \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma'[Y/a], \Sigma, \mu)}$$

Figure 7.5: Graph Semantics – Function Calls

In the rules for function calls and returns, we push or pop the actual substitution to/from the stack. This stack has always the same size as the one we use in the graph representation. It assures the correct scoping of variables. Renaming is superfluous. We have omitted the rules for runtime errors here. Similarly to Figure 3.4, they can easily be defined.

The graph semantics and the AOS are equivalent for arbitrary Core Erlang programs. This is not surprising since our translation corresponds to standard techniques in compiler construction. We use the stacks similarly to the implementation of procedure calls. Therefore, the equivalence should be intuitively clear. This equivalence is not a major point of this thesis. Already the formalization of the equivalence is technically expensive and would require the introduction of bisimulation. The formal proof of the equivalence relates to proofs for the correctness of compilers which are expensive as well. Therefore, we omit the formalization and the proof.

For Core Erlang programs which use recursion only in tail positions the graph representation is a finite transition system. We will use this graph representation for our abstraction but we can also use it for a more efficient implementation of abstrac-

tion and model checking. In the first implementation we used Core Erlang evaluation terms to identify the states. Constructing the abstract model, it is necessary to detect cycles. Therefore, the states must be stored. For every new state in the transition system its successors are computed and compared with the stored states. Only for new states further successors must be computed. However, the storage of states needs much space and the comparison of states (although they can be stored in balanced trees) needs much time. Therefore, a compact representation of a state is desirable.

The graph representation is a transition system where the transitions represent the behavior of a process. The labels of the states have only been used for its construction but they are superfluous after that (see Figures 7.2 – 7.5). We can e.g. replace them by numbers. This can be done as a pre-compilation for all functions which can be spawned (including `main`). Then we construct the interleaving transition system using these numbers as names of the states a process is in. This is a much more compact representation of a state and allows a faster verification of even larger systems. Furthermore, we do not have to descend the evaluation context during the generation of the successors of a state. The successors can be evaluated more efficiently. Like in compiler construction the correct order of the evaluations can be defined in a pre-compilation.

For non-hierarchical Core Erlang programs this graph representation is infinite, as the following example show.

Example 7.3
Consider the following function definition:

```
f(X) -> case X of
          0 -> b;
          N -> self!a, f(X-1), self!b
        end.
```

A process executing this function sends `X` times the atom `a` to itself and after that `X` times `b`.

The resulting graph representation is sketched in Figure 7.6. For a better distinction of the commas in the Core Erlang terms and the stacks, we use | to separate the evaluation term from the stack of contexts.

7.3 Abstracting from the Context-Free Structure

As discussed in Section 7.1 we need abstraction of the context-free structure of Core Erlang programs. We use the same basic idea as for the abstract interpretation. We construct a finite abstract graph representation of the program. Its semantics is safe with respect to the SOS. Furthermore, its semantics defines a finite state transition system for Core Erlang programs containing non-tail recursive calls.

Our approach is a kind of call-string approach [SP81] on program level and was presented in [Huc01, Huc02]. The main idea of the abstraction is to replace the calls and the returns by jumps. For Example 7.3, sending n times an `a` and after that m

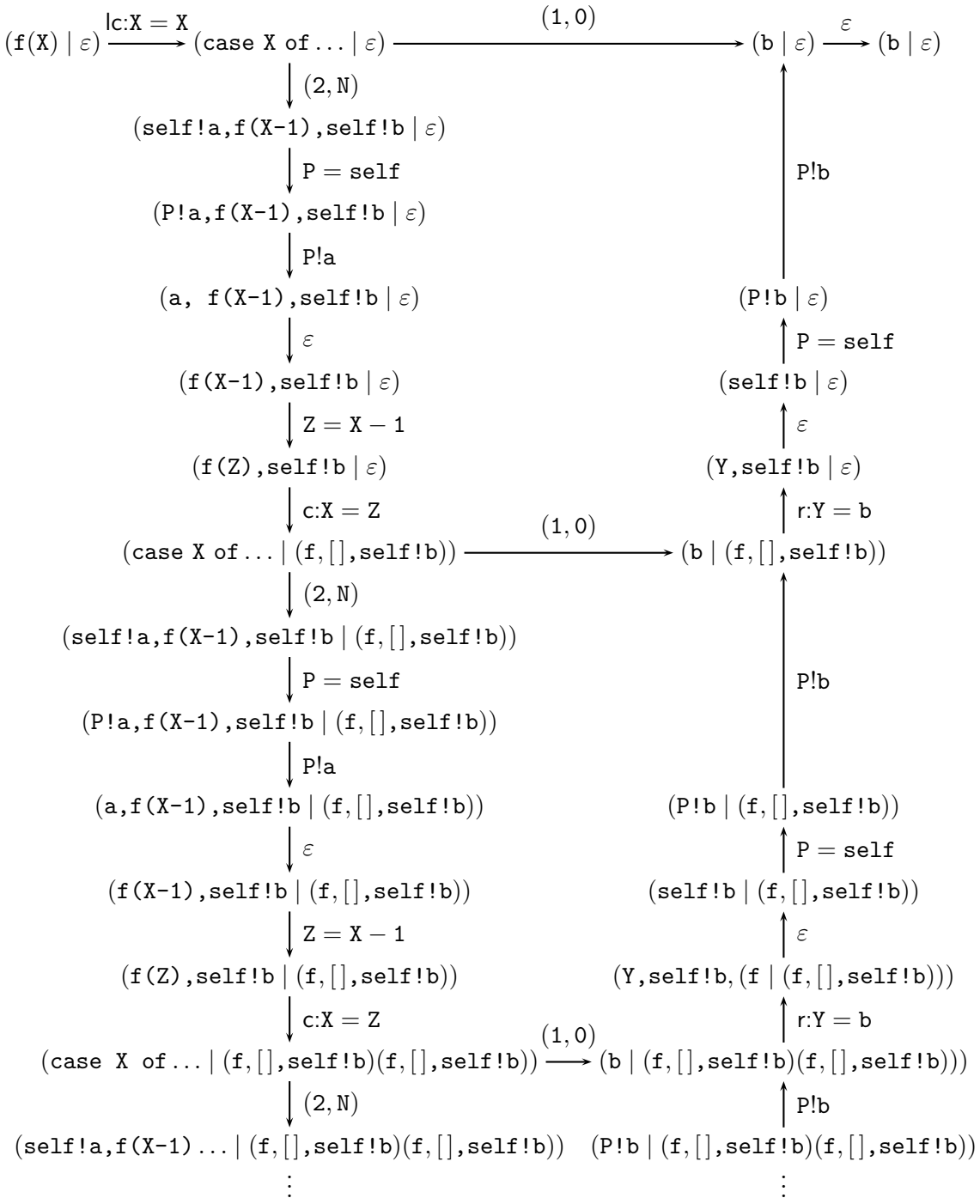


Figure 7.6: Graph Representation of Example 7.3

times a `b` is a good abstraction. A property like "no `a` is sent after a `b`" could then be proven automatically.

The idea of the abstraction is to replace the calls of `f` (see Figure 7.6) by jumps to a predecessor node where `f` was already called. Hence, we replace the second non-tail call of `f` by the following arc:

$$\begin{array}{c}(\texttt{f(Z),self!b} \mid (\texttt{f}, [\,], \texttt{self!b})) \\ \Big\downarrow \texttt{c:X=Z} \\ (\texttt{case X of} \ldots \mid (\texttt{f}, [\,], \texttt{self!b}))\end{array}$$

The states underneath $(\texttt{f(Z),self!b} \mid (\texttt{f}, [\,], \texttt{self!b}))$ in Figure 7.6 need not be considered anymore.

Now, how can we perform the corresponding return? We know the stack of the state we jumped to instead of calling `f`. Hence, the evaluation of this call will be terminated if the Core Erlang term is evaluated to a value combined the same stack as the one we jumped to instead of the call. These are all states of the form $(a \mid (\texttt{f}, [\,], \texttt{self!b}))$ where $a \in T_{\mathcal{C}}(Var)$. In our example this is only the state $(\texttt{b} \mid (\texttt{f}, [\,], \texttt{self!b}))$. The destination of this returning jump is defined by the state where the call was initiated. The result of the call is `b`:

$$(\texttt{b} \mid (\texttt{f}, [\,], \texttt{self!b})) \xrightarrow{\texttt{r:Y=b}} (\texttt{Y,self!b} \mid (\texttt{f}, [\,], \texttt{self!b}))$$

We do not pop the top-level context as usually in a return step. The context stack is not modified. The result is a finite graph representation in which n times an `a` is sent and then m times a `b`. It is presented in Figure 7.7.

Generalizing this technique some problems appear. In general, we do not have only one function which calls itself recursively. We have several functions. Therefore, we have extended the call stack to the names of the called functions. We can distinguish the different function calls. Thus, we only jump back to states which correspond to the right hand-side of the function we are calling. Another feature of this extension is that we can detect if a function was already called. If it was not called, then it does not appear in the stack. A sub-evaluation which terminates and does not recursively call something outside itself will not be abstracted. In this case the abstract graph representation is similar to the non-abstract one. No calls are converted into jumps and no additional paths are added. Only if we detect recursion in a non-tail position, then we cut off the transition system and jump back.

Example 7.4
Another problem is exposed by a modified version of Example 7.3. We send the value of the variable `X` instead of the atom `b`:

```
f(X) -> case X of
          0 -> b;
          N -> self!a, f(X-1), self!X
        end.
```

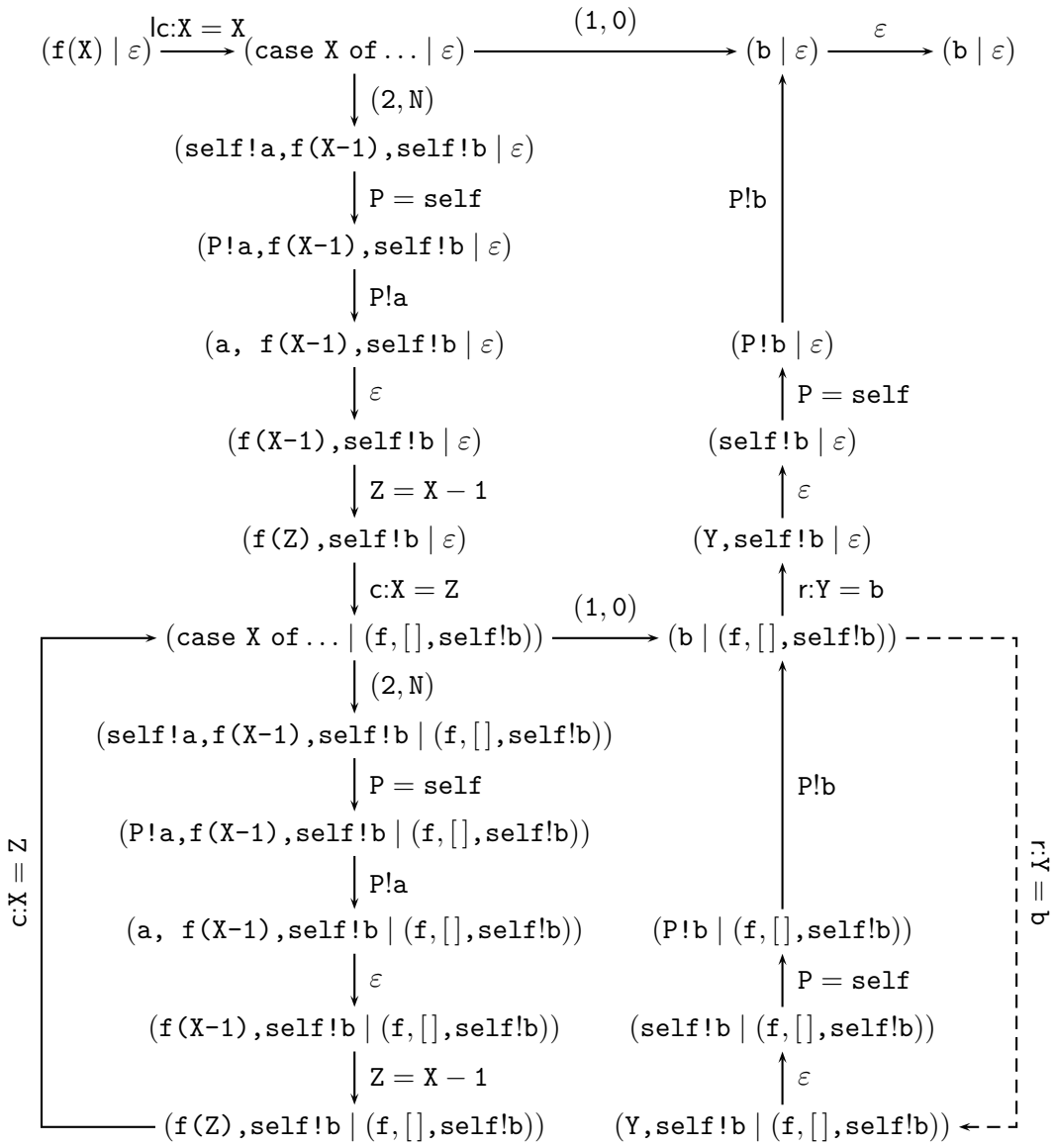


Figure 7.7: Abstract Graph Representation of Example 7.3

First, the process sends n times an `a` to itself and then it sends the numbers $1, \ldots, n$ where $n \in \mathbb{N}$ is the value `f` is applied to initially.

In the abstraction of Example 7.3 we replace the communication by sending n times `a` and m times `b`. What can we do here? In the abstract domain these values are represented by abstract values which need not be an infinite set (especially in a finite domain abstraction). Jumping back instead of calling, we cannot know to which value `X` is bound. Hence, we bind `X` to the value **?** which represents all values. We claim that such a least value exists in our abstract domain. Otherwise, we can always add **?** where $? \sqsubseteq v \; \forall v \in \widehat{A}$. We annotate the label of the return arc by this substitution:

$$(\texttt{b} \mid (\texttt{f}, [\,], \texttt{self!X})) \xrightarrow{\substack{\texttt{r:Y=b} \\ [\texttt{X}/?]}} (\texttt{Y,self!X} \mid (\texttt{f}, [\,], \texttt{self!X}))$$

In this abstract return jump we do not remove the top element of the call stack. It is even possible that we have to add more entries if the recursive call is indirect. This means that other functions have been called in between. When we return from the function call we have to reconstruct the call stack to the old stack because in the GOS these stored contexts still have to be executed. However, using the variable bindings of these contexts, we have the same problem as using variables in the Core Erlang term the evaluation returns. We lose the bindings of these contexts in the abstract call. The solution is to add bindings for the variables of these contexts to **?**.

We also have to note these changes of the call stack in the label because in the AOS we stack the substitutions in the same manner as in the graph representation. Hence, we annotate the number of stack elements which are removed instead of pushing a new block in an abstract call. Analogously, we note the number of stack elements which have to be added in the abstract return jump and add the substitutions to **?** for these stack frames. For corresponding calls and returns these numbers coincide. In our example it is zero because no functions were called in between

$$(\texttt{f(Z),self!X} \mid (\texttt{f}, [\,], \texttt{self!X})) \xrightarrow{\texttt{c(0): X=Z}} (\texttt{case X of} \ldots \mid (\texttt{f}, [\,], \texttt{self!X}))$$

and

$$(\texttt{b} \mid (\texttt{f}, [\,], \texttt{self!X})) \xrightarrow{\substack{\texttt{r(0): Y=b,} \\ [\texttt{X}/?], ()}} (\texttt{Y,self!X} \mid (\texttt{f}, [\,], \texttt{self!X}))$$

An example of indirect recursion over two functions will be presented in Section 7.4. There, stack elements removed in the abstract call and reconstructed in the corresponding abstract return jump.

So far we bind all variables to **?** in an abstract return jump. This is safe but not necessary. It is sufficient to bind only the variables to **?** which are already bound. Bindings to **?** is superfluous for the variables which will later be bound. Due to the fact that Erlang does not have scoping, we do not know if a variable occurring in

1. $(E[a, e], W) \xrightarrow{\varepsilon} (E[e], W)$
2. $(E[a\,!\,b], W) \xrightarrow{a\,!\,b} (E[b], W)$
3. $(E[\texttt{self}], W) \xrightarrow{Y = \texttt{self}} (E[Y'], W)$ where $Y \notin Var(E)$
4. $(E[p\texttt{=}a], W) \xrightarrow{p = a} (\texttt{tag}(Var(p), E[a]), W)$
5. $(E[\texttt{receive}\ p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_n\texttt{->}e_n\ \texttt{end}], W) \xrightarrow{(i, ?p_i)} (\texttt{tag}(Var(p_i), E[e_i]), W)$ for all $1 \leq i \leq n$
6. $(E[\texttt{case}\ a\ \texttt{of}\ p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_n\texttt{->}e_n\ \texttt{end}], W) \xrightarrow{(i, p_i = a)} (\texttt{tag}(Var(p_i), E[e_i]), W)$ for all $1 \leq i \leq n$
7. $(E[\phi(a_1, \ldots, a_n)], W) \xrightarrow{Y = \phi(a_1, \ldots, a_n)} (E[Y'], W)$ where $Y \notin Var(E)$
8. $(E[\texttt{spawn}(f, a)], W) \xrightarrow{Y = \texttt{spawn}(f, a)} (E[Y'], W)$ where $Y \notin Var(E)$
9. $(f(\overline{a}), W) \xrightarrow{\texttt{lc:}\overline{X} = \overline{a}} (\texttt{tag}(\{\overline{X}\}, e_f), W)$ where $f(\overline{X})\texttt{->}e_f\,\texttt{.} \in p$
10. $(E[f(\overline{a})], W) \xrightarrow{\texttt{c:}\overline{X} = \overline{a}} (\texttt{tag}(\{\overline{X}\}, e_f), (f, E)W)$ where $f(\overline{X})\texttt{->}e_f\,\texttt{.} \in p$ and $E \neq [\,]$
11. $(a, (f, E)W) \xrightarrow{\texttt{r:}Y = a} (E[Y'], W)$ where $a \in T_{\mathcal{C}}(Vars)$ and $Y \notin Var(E)$

Figure 7.8: Graph Representation using a Stack and Tagging

a sub-term is free or bound. We need an analysis which marks the variables which are already bound to values. This analysis can be combined with the construction of the abstract graph representation. Building the graph representation we can detect when a variable is bound. We mark it by a tag (′). In the abstract return jump we can bind all tagged variables to **?**. The others can be left unchanged. Consider the following example:

```
f(X) -> case X of
          0 -> b;
          N -> self!a, f(X-1), B=b, self!B
        end.
```

The variable `B` is not bound before the recursive call. We can leave it unchanged:

$$\xrightarrow{\texttt{r(0):Y=b,[],()}} \begin{array}{l} (\texttt{b} \mid (\texttt{f}, [\,], \texttt{B=b,self!B})) \\ (\texttt{Y'}, \texttt{B=b,self!B} \mid (\texttt{f}, [\,], \texttt{B} = \texttt{b}, \texttt{self!B})) \end{array}$$

7.4 Abstract Graph Representation

In the last section, we have motivated our abstraction in some examples. Now we present the formal definition and prove its safeness with respect to its use in formal verification by model checking.

First we define a function tag which tags a set of variables.

$$\mathsf{tag}(V, X) = \begin{cases} X' & \text{if } X \in V \\ X & \text{otherwise} \end{cases}$$

It is also canonically extended to Core Erlang terms and contexts. The graph representation of Core Erlang containing a stack and tagging of bound variables is defined in Figure 7.8. Every variable which is bound to a value is tagged. This tagging is just an additional information and tagged variables are treated like un-tagged ones. In the transition labels we use only the names of the variables and ignore the tags. They are superfluous for the GOS.

Recursion is abstracted by jumps back to the last call of the same function. It is detected in the call stack if the same function was already called. The destination state of this jump has a smaller call stack than the call would yield. To relate call stacks in the graph representation and their abstract representation, we define an abstraction function α. This function yields the call stack which is constructed by a stepwise execution of abstract calls. If the same function was already called, then the stack decreases.

$$\begin{aligned} \alpha(\varepsilon) &= \varepsilon \\ \alpha((f,E)W) &= \begin{cases} (f,E)\alpha(W) & \text{if } |\alpha(W)|_f = 0 \\ (f,E')V & \text{if } \alpha(W) = U(f,E')V \\ & \quad \text{where } |U|_f = |V|_f = 0 \end{cases} \end{aligned}$$

From the definition it is not directly clear that α is total. Using the following lemma, we see that always one of the two cases for $\alpha((f,E)W)$ matches. Hence, α is defined for all call stacks.

Lemma 7.5
$|\alpha(W)|_f \leq 1$ *for all call stacks* W *and all functions* $f \in FS(p)$.

Proof: A simple induction on W:

- $W = \varepsilon$. Trivial
- $W = (f,E)W'$. By induction hypothesis we know that $|\alpha(W')|_f \leq 1$. We distinguish two cases:
  - $|\alpha(W')|_f = 0$. Hence, $\alpha(W) = (f,E)\alpha(W')$ and $|\alpha(W)|_f = 1$. And for all $g \neq f$ $|\alpha(W)|_g = |\alpha(W')|_g \leq 1$ by induction hypothesis.
  - $\alpha(W') = U(f,E')V$ where $|U|_f = |V|_f = 0$. Then $\alpha(W) = (f,E')V$ and $|\alpha(W)|_f = 1$. Again $|\alpha(W)|_g = |\alpha(V)|_g \leq 1$ by induction hypothesis. □

We use this abstraction function for the analysis of a given call stack when calling a function. The abstract graph representation can be defined using this abstraction function. It is defined as the relation

$$\rightharpoonup \subseteq SR(T_{\mathcal{C}}(Var)) \times \widehat{Act} \times SR(T_{\mathcal{C}}(Var))$$

The actions $\widehat{Act}$ contain Act and the actions for abstract calls and returns. $\rightharpoonup$ is defined by the rules (1)–(9) and (11) of $\longrightarrow$. Instead of call stacks (10) we use their abstract representation:

$$(E[f(\overline{a})], \alpha(W)) \xrightarrow{\mathsf{c}(n):\overline{X} = \overline{a}} (\mathsf{tag}(\{\overline{X}\}, e_f), \alpha((f, E)W))$$

$$\begin{aligned} \text{where } & f(\overline{X})\texttt{->}e_f\texttt{.} \in p \\ & E \neq [\,] \\ & n = |\alpha(W)| - |\alpha((f, E')W)| \end{aligned}$$

If the function call is not abstracted by a jump, we get $n = -1$. This means that we can add the actual context to the call stack as we would do without abstraction. In this case we will just write c instead of $\mathsf{c}(\texttt{-1})$. Otherwise, we add a jump back. This means that we detect recursion and $|\alpha(W)|_f = 1$. For all $a \in T_{\mathcal{C}}(Var)$:

$$(a, \alpha((f, E)W)) \xrightarrow[(\sigma_1, \ldots, \sigma_n)]{\mathsf{r}(n):\, Y = a \; [\mathsf{tagged}(E)/\texttt{?}]} (E[Y'], (W_1 \ldots W_k))$$

$$\begin{aligned} \text{where } & W_{n+1} \ldots W_k = \alpha((f, E)W) \\ & W_1 \ldots W_{n+1} \ldots W_k = \alpha(W) \\ & W_i = (f_i, E_i) \\ & \sigma_i = [\mathsf{tagged}(E_i)/\texttt{?}] \quad \forall 1 \leq i \leq n \end{aligned}$$

Note that still $n = |\alpha(W)| - |\alpha((f, E')W)|$ and $n \geq 0$ always holds if $|\alpha(W)|_f = 1$. In this case $W_1 \ldots W_n$ are the blocks which have to be restored in this return jump. The bound variables in these blocks and in E cannot be known. We bind them to **?** in the evaluation. The function tagged yields all tagged variables. For these we can define substitutions which bind them to **?**. These are the substitutions $[\mathsf{tagged}(E)/\texttt{?}]$ and $(\sigma_1, \ldots, \sigma_n)$. We add them to the label.

Example 7.6
Consider the following program in which the functions `f` and `g` are defined by mutual recursion:

```
f(X) -> case X of
          0 -> b;
          N -> g(X-1), self!X
        end.

g(X) -> f(X-1), self!X.
```

The abstract graph representation of this program is presented in Figure 7.9. The arcs in this graph are only drawn as a single line but the displayed graph presents the relation $\rightharpoonup$. In the abstraction, parts of the call structure of the program are preserved. First we have an even number of function calls[4] and then also an even number of times the value `X` is send. In the AOS over this abstract graph representation `X` will be bound to **?** with the exception of the first send operation.

Lemma 7.7 (Safeness of the Abstract Graph Representation)
If $(e_1, W_1) \xrightarrow{l} (e_2, W_2)$ *then* $(e_1, \alpha(W_1)) \xrightarrow{\widetilde{l}} (e_2, \alpha(W_2))$ *where* $\widetilde{l}$ *has one of the following forms:*

if $l = \mathsf{r}{:}Y = a$ *then* $\widetilde{l} = \mathsf{r}(n){:}Y = a, \sigma, \overline{\sigma}$
or $\widetilde{l} = \mathsf{r}{:}Y = a$
if $l = \mathsf{c}{:}\overline{X} = \overline{a}$ *then* $\widetilde{l} = \mathsf{c}(n){:}\overline{X} = \overline{a}$
otherwise $\widetilde{l} = l$

Proof: We distinguish the cases 1. to 11. from the definition of $\longrightarrow$. The cases 1. to 9. are trivially fulfilled.

10. $(E[f(\overline{a})], W) \xrightarrow{\mathsf{c}:\overline{X} = \overline{a}} (e_f, (f, E) : W)$
and also
$(E[f(\overline{a})], \alpha(W)) \xrightarrow{\mathsf{c}(n):\overline{X} = \overline{a}} (e_f, \alpha((f, E)W))$.

11. $(a, (f, E)W) \xrightarrow{\mathsf{r}:Y = a} (E[Y], W)$
We distinguish two cases:

- $|\alpha(W)|_f = 0$
Then $\alpha((f, E)W)) = (f, E)\alpha(W)$ and also
$(a, (f, E)W) \xrightarrow{\mathsf{r}:Y = a} (E[Y], \alpha(W))$.
- $|\alpha(W)|_f = 1$
Then $\alpha((f, E)W)) = (f, E')V$ where $\alpha(W) = U(f, E')V$. Because W is a stack, we know that there must also be a state where

$$(E[f(\overline{a})], W) \xrightarrow{\mathsf{c}:\overline{X} = \overline{a}} (e_f, (f, E) : W) \longrightarrow^m (a, (f, E)W)$$

By simple induction over the length m of this derivation we can conclude that also

$$(E[f(\overline{a})], \alpha(W)) \xrightarrow{\mathsf{c}(n):\overline{X} = \overline{a}} (e_f, \alpha((f, E)W)) \rightharpoonup^m (a, \alpha((f, E)W))$$

and $(a, \alpha((f, E)W)) = (a, (f, E)V)$. Hence

$$(a, \alpha((f, E)W)) \xrightarrow{\mathsf{r}(n):Y = a, \sigma, \overline{\sigma}} (E[Y], \alpha(W))$$

where σ and $\overline{\sigma}$ are defined as above. □

[4] Apart from the initial tail recursive call.

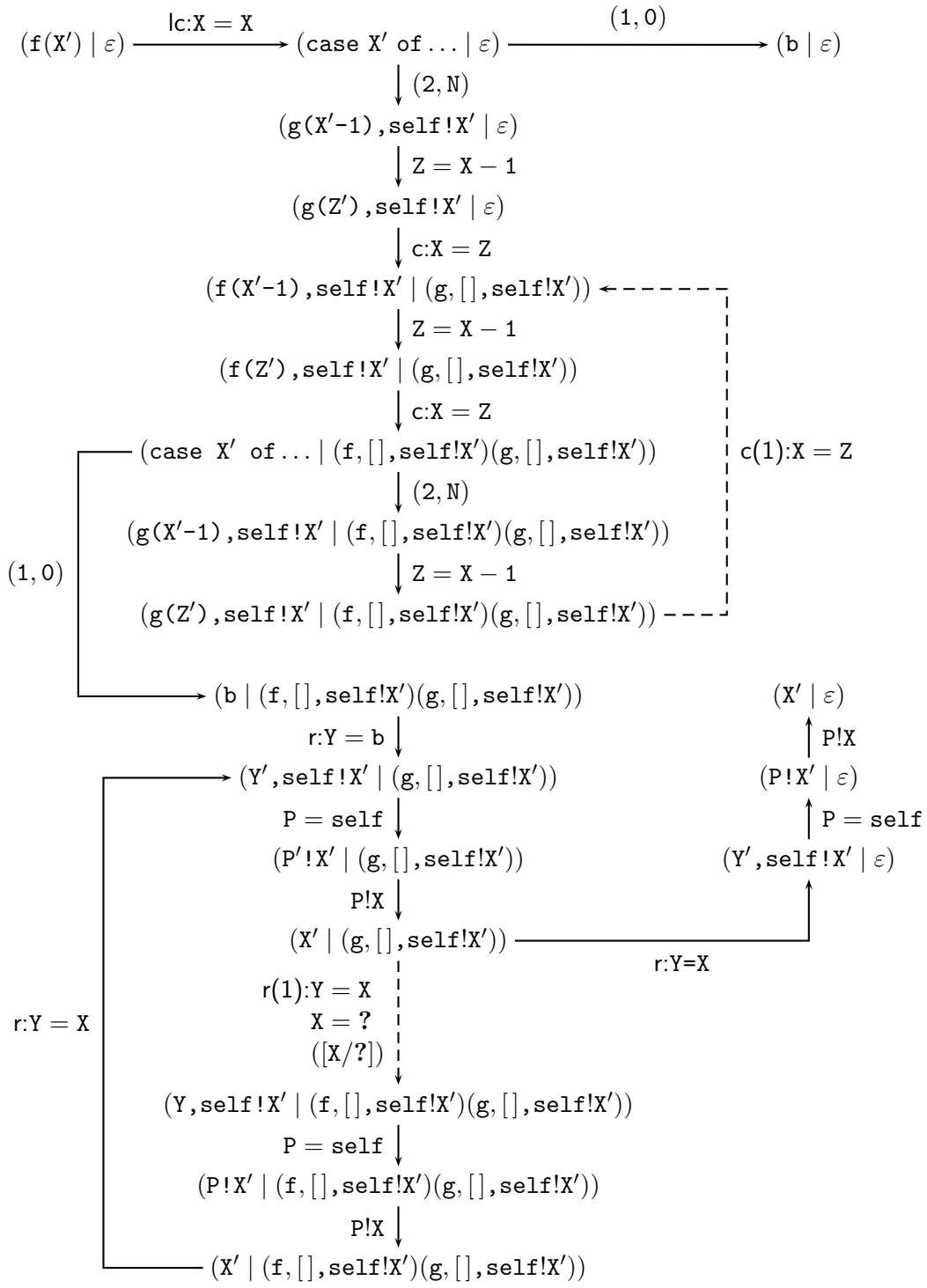


Figure 7.9: Abstract Graph Representation (⇀) of Example 7.6

For a Core Erlang program p we define the complete abstract graph representation $\mathcal{AG}_p := (S, \multimap, \mathsf{init})$ as the transition system which is the restriction of $\multimap$ to the states which can be reached from the state $(\texttt{main()}, \varepsilon)$ and the initial states of all functions which are spawned in p. $\mathsf{init} : FS(p) \longrightarrow S$ yields the initial state for the spawned functions and the variables which have to be bound in their calls. It was already defined in Section 7.2.

Lemma 7.8
The abstract graph representation $\mathcal{AG}_p := (S, \multimap, \mathsf{init})$ is finite for every Core Erlang program p.

Proof: This is easy to see because the stacks can only grow to finite depth and also only finitely many terms and contexts may occur. They are restricted in size because we only have a finite set of rules. Furthermore, only finitely many different functions can be spawned because $FS(p)$ is finite. Therefore, S is a finite set of states and also $\multimap$ can be restricted to this set. □

It is also possible to define an algorithm to compute $\mathcal{AG}_p$. States are successively added using $\multimap$ until no more states can be added. Furthermore, the algorithm memorizes which abstract calls have been performed. This yields the stacks for which abstract return jumps must be added. If the construction of $\mathcal{AG}_p$ reaches a state containing an evaluated expression, then it adds return jumps to the corresponding memorized states.

The semantics over this abstract graph representation ($\hookrightarrow$) is defined analogously to the semantics over the graph representation ($\rightsquigarrow$). We only have to add the rules for the abstract calls and returns:

$$\frac{s \xrightarrow{\mathsf{c}(n):\overline{X} = \overline{a}}_{\multimap} s' \text{ and } n \geq 0}{\begin{array}{c}\Pi, (\pi, s, \sigma, (\sigma_1 : \ldots : \sigma_k), q) \\ \hookrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', [\overline{X}/\overline{a}], (\sigma_{n+1} : \ldots : \sigma_k), q)\end{array}}$$

$$\frac{s \xrightarrow{\mathsf{r}(n):Y = a, \sigma, \overline{\sigma}}_{\multimap} s'}{\Pi, (\pi, s, \sigma', \Sigma, q) \hookrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma[\overline{Y}/\overline{a}], (\overline{\sigma} : \Sigma), q)}$$

Theorem 7.9 (Safeness of the Abstraction)
Let p be a Core Erlang program, $\mathcal{AG}_p = (S, \multimap, s_0)$ the corresponding abstract graph representation, and $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ an abstract interpretation for Core Erlang programs with least element $\mathbf{?} \in \widehat{A}$. Then for all $s \overset{a}{\rightsquigarrow}_{\widehat{\mathcal{A}}} t$ and $\widetilde{s} \preceq \alpha(s)$ there exists $\widetilde{a} \sqsubseteq \alpha(a)$ and $\widetilde{t} \sqsubseteq \alpha(t)$ such that $\widetilde{s} \overset{\widetilde{a}}{\hookrightarrow}'_{\widehat{\mathcal{A}}} \widetilde{t}$.

Proof: Using Lemma 7.7 the proof is a straightforward case analysis on $\rightsquigarrow$. In the abstract call we lose some variable bindings but in the corresponding return jump these variables are bound to $\mathbf{?}$. Therefore, a variable is either bound to the same

value as in the full graph representation or it is bound to **?**. The latter results from performing an abstracted call in which the variable is abstracted. **?** is the least element of $\widehat{A}$ and the theorem is fulfilled. □

For practical verification this abstraction is sufficient. However it is possible to precise the abstraction: instead of jumping back to the state where the same function was called for the first time we can allow $k \in \mathbb{N}$ calls of the same function in between and also accept an initial part with $n \in \mathbb{N}$ calls. We only have to modify the conditions in the definition of α:

$$\begin{aligned} \alpha(\varepsilon) &= \varepsilon \\ \alpha((f,E)W) &= \begin{cases} (f,E)\alpha(W), \text{ if } |\alpha(W)|_f < k+n \\ (f,E')V \quad\quad , \text{ if } \alpha(W) = U(f,E')V \\ \qquad\qquad\quad \text{with } |V|_f = n \text{ and } |U|_f = k \end{cases} \end{aligned}$$

The rest of the control flow abstraction can be left unchanged and we obtain more precise abstractions. The abstraction presented above is obtained by $k = 1$ and $n = 0$.

7.5 Verification

Using the presented technique we can now prove the properties of Section 5.5 for the optimized implementation of the database from the beginning of this chapter. In this optimization of the database, the non-tail recursive function `insert` is used. We inspect the presented control-flow abstraction of this function call. The function `insert` is not spawned. Therefore, its graph representation is not constructed. It occurs as a subgraph in the abstract graph representation of the `database` function which is spawned from the main function. Since the flow-abstraction of `insert` is the interesting part of this graph representation, we present it in Figure 7.10. For a smaller representation we write `i` instead of `insert`. The abstract graph representation contains two loops: The first loop represents the recursive decent through the list. Respectively, the second (return) loop represents the construction of the resulting list.

In spite of these loops, the AOS for a finite domain abstraction yields a finite transition system. The abstract interpretation of list constructors impedes that arbitrary lists are constructed in the evaluation of the loops. However, the AOS will usually contain ε-loops for both loops. We have already discussed this problem in Section 5.5. The abstractions used for the verification of the Erlang programs are usually not designed to prove the termination of sequential evaluations. Therefore, we ignore these ε-loops in the verification.

Using the presented control flow abstraction in combination with the depth-2 abstraction $\widehat{\mathcal{A}}_2$ we can prove all properties from Section 5.5 for the optimized version of the database example.

Only for the following two cases our approach yields infinite transition systems:

- an infinite number of processes is created
- the mailbox of a process grows infinitely

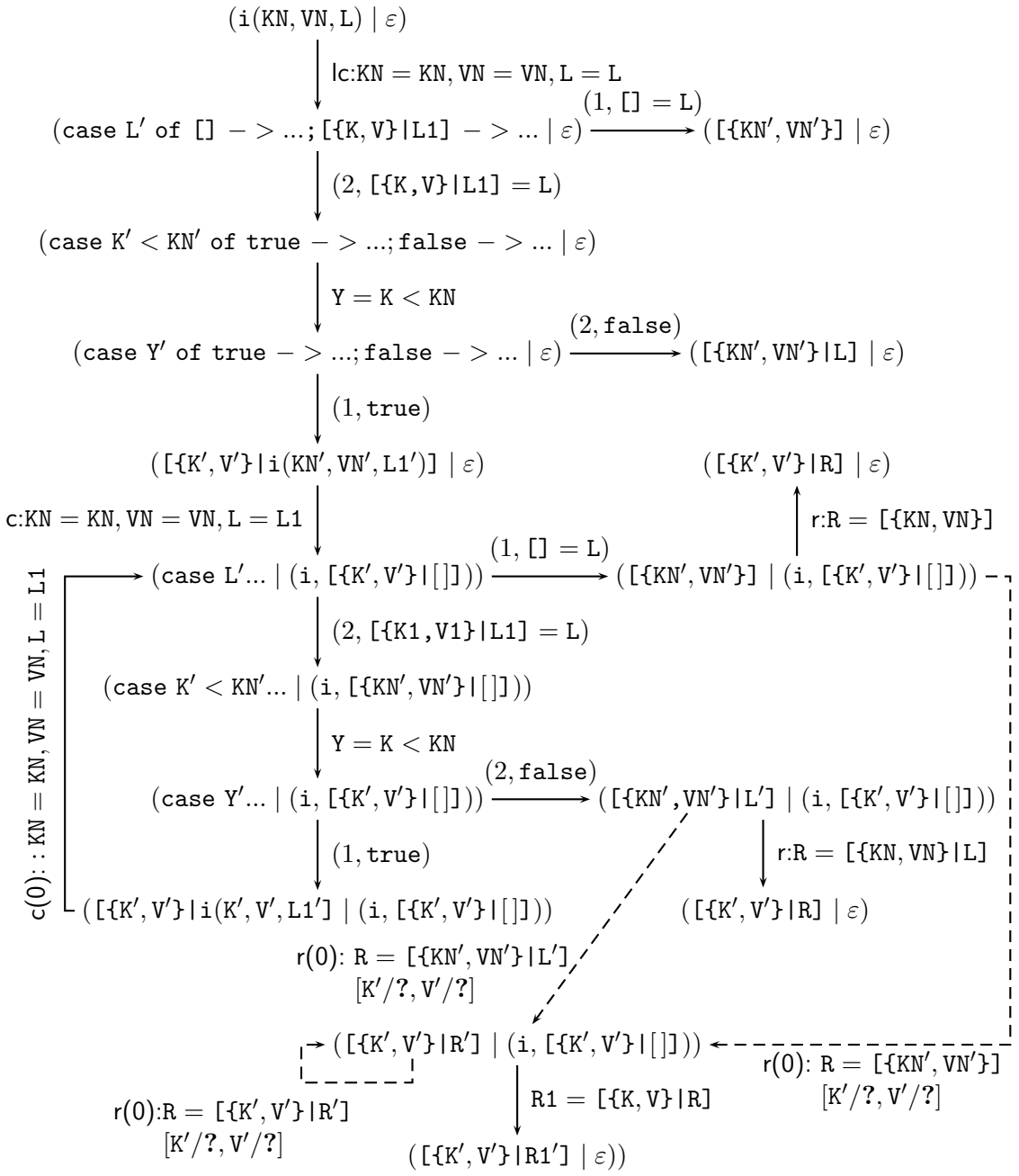


Figure 7.10: Abstract Graph Representation (⟶) of the Function `insert`

As discussed before, we do not tackle these problems.

Chapter 8

Towards the Verification of Erlang Programs

So far we have only considered a core fragment of Erlang in which however the main concepts of Erlang are considered. In this section we discuss how the other parts of real Erlang programs can be handled in formal verification.

8.1 The Module System

In Core Erlang we do not consider modules. Usually, Erlang programs consist of several modules. Functions can be exported and imported. The simplest translation to Core Erlang joins all modules of an Erlang program. Every function name is extended by the module name to avoid conflicts in the name space of the functions. In Section 2.2 we defined a module for a database process. This module can simply be translated into Core Erlang as follows:

```
database:start() -> database:database([]).

database:database(L) -> ...
```

Furthermore, we translate all calls of `spawn/3` to `spawn/2`:

$$\texttt{spawn(database,start,[])} \rightsquigarrow \texttt{spawn(database:start,[])}$$

Using this translation, it is possible to eliminate the module system in most Erlang programs. However, in Erlang it is also possible to generate names of modules or functions at runtime:

```
spawnDynamic() -> {ok,M} = io:read('Module : '),
                  {ok,F} = io:read('Function :'),
                  M:F(),
                  spawn(M,F,[]).
```

This valid Erlang program reads a module name and a function name from the keyboard. Then it evaluates the corresponding function of arity zero. Finally, it spawns a process which executes this function, again of arity zero.

This cannot be translated into Core Erlang. We strictly distinguish values from program code. Especially the function call `M:F()` is not allowed in Core Erlang. Fortunately, this kind of dynamic function calls is not often used in Erlang programs.

8.2 Branching

Branching in Erlang programs is not restricted to `case` and `receive`. The programmer can also use an `if` expression which can be translated into a sequence of `case` expressions:

```
if                                  case c1 of
  c1 -> e1;                           true -> e1;
   ⋮                                  false -> case c2 of
  cn -> en                                      ⋱
end                                               case cn of
                                                    true -> en
                                                  end
                                                  ⋰
                                              end
                                    end
```

$c_1, \ldots, c_n$ are boolean conditions. They are successively tested. The corresponding e_i of the first fulfilled condition is chosen. The `if` expression yields a runtime error, if none of the conditions is fulfilled. Hence, the last `case` in the translation has no false pattern.

Another possibility for branching in Erlang is programming by means of several rules in the function definition. This pattern matching can simply be translated into a `case` expression:

```
f(p11, ..., p1n)  -> e1;          f(X1, ..., Xn) ->
    ⋮                                 case {X1, ..., Xn} of
f(pm1, ..., pmn) -> em.                 {p11, ..., p1n}  -> e1;
                                              ⋮
                                        {pm1, ..., pmn} -> em
                                    end.
```

It is not even necessary to eliminate nested or overlapping pattern matching. The pattern matching in the `case` expression of Core Erlang is as expressive as the pattern matching in function definitions.

8.3 Higher-Order Functions

In Erlang it is also possible to use higher-order functions. As an example the broadcast of a message `M` to a list of pids can be defined by the higher-order function `map/2`.

```
map(F,L) -> case L of
              []     -> [];
              [X|Xs] -> [F(X)|map(F,L)]
            end.

broadcast(M,Pids) -> map(fun(P) -> P!M end, Pids).
```

The first argument of `map` is applied to all elements of the list in the second argument. The result is a list of the same size as the argument list.. Here it is applied to an anonymous function which takes a pid and sends the value of the variable `M` to this pid.

We can extend Core Erlang by functional abstraction (λ-abstraction) and application:

$$e ::= ... \mid \texttt{fun(}X_1\texttt{,}\ldots\texttt{,}X_n\texttt{)->}e\ \texttt{end} \mid e(e_1,\ldots,e_n)$$

For the semantics we extend the evaluation contexts to

$$E ::= ... \mid v\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_{i-1}\texttt{,}E\texttt{,}e_{i+1}\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mid E(e_1,\ldots,e_n)$$

and handle functions as values. We add a rule similar to β-reduction to the semantics:

$$\frac{\begin{array}{c}\{Y_1,\ldots,Y_m\} = vars(e) \setminus \{X_1,\ldots,X_n\} \quad \text{and} \\ Z_1,\ldots,Z_m \notin vars(E)\ (Z_i \neq Z_j \text{ for } i \neq j) \quad \text{and} \quad e' = e[Y_1/Z_1,\ldots,Y_m/Z_m]\end{array}}{\begin{array}{l}\Pi,(\pi,E[\texttt{fun(}X_1\texttt{,}\ldots\texttt{,}X_n\texttt{)->}e\ \texttt{end(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)}],\mu) \\ \longrightarrow_{\widehat{\mathcal{A}}}\ \Pi,(\pi,E[e'[X_1/v_1,\ldots,X_n/v_n]],\mu)\end{array}}$$

As in the rule for the application of a defined function, we avoid name conflicts by renaming all free variables in the body of the anonymous function. Although Erlang has no scoping in general, it uses scoping in functional abstractions. In other words, the variables $X_1,\ldots,X_n$ introduced in $\texttt{fun(}X_1\texttt{,}\ldots\texttt{,}X_n\texttt{)->}e\ \texttt{end}$ are new variables. They are not identified with variables with the same name in the context. A substitution is not applied to $X_1,\ldots,X_n$ in e and their bindings resulting from e are not exported to the context of the functional abstraction.

Using this operational semantics, we can also prove properties of Erlang programs which use higher-order functions. However, for many programs the AOS will define an infinite transition system and model checking is not possible. The reason is that functions are values. This yields an infinite domain and in many cases an infinite transition system because different functions occur. On the other hand, there also exist programs for which we get a finite transition system and can prove properties. As an example we consider the AOS of broadcast applied to the message **42** and the list only containing the own pid. Again, we consider the example abstract domain $\widehat{\mathcal{A}}_2$.

$$
\begin{array}{c}
(@\mathbf{0}, \texttt{broadcast}(42, [\texttt{self}]), ()) \\
\downarrow * \\
(@\mathbf{0}, \texttt{map(fun(P)->P!}42\ \texttt{end,}[@\mathbf{0}]), ()) \\
\downarrow \\
(@\mathbf{0}, \texttt{case}\ [@\mathbf{0}]\ \texttt{of}\ \texttt{[]->...;[X|Xs]->...}, ()) \\
\downarrow \\
(@\mathbf{0}, \texttt{[fun(P)->P!}42\ \texttt{end}(@\mathbf{0})\texttt{|map(...,[])]}, ()) \\
\downarrow \\
(@\mathbf{0}, \texttt{[}@\mathbf{0}\texttt{!}42\texttt{|map(...,[])]}, ()) \\
\downarrow \\
(@\mathbf{0}, \texttt{[}@\mathbf{0}\texttt{!}\mathbf{42}\texttt{|map(...,[])]}, ()) \\
\downarrow \texttt{!}\mathbf{42} \\
(@\mathbf{0}, \texttt{[}\mathbf{42}\texttt{|map(...,[])]}, (\mathbf{42})) \\
\downarrow * \\
(@\mathbf{0}, [\mathbf{42}|[]], (\mathbf{42}))
\end{array}
$$

The definition of a class, for which the AOS defines a finite transition system is much more difficult. For example, we could also apply the function `broadcast` to `?`. Then the termination of the recursion of `map` cannot be decided. The recursive call of `map` is not a tail call. We get an infinite transition system in the AOS.

8.4 Distributed Systems

In our formalization we have only considered concurrent systems. Most Erlang applications are distributed. However, this is no problem. The process concept of Erlang with communication via asynchronous message passing guarantees safe communication via a network. Apart from system failures, distributed communication does not distinguish from concurrent communication. The semantics of the communication is independent of the underlying hardware architecture. If we prove a property for a concurrent system, then this property also holds for any distribution of the defined processes.

If we want to prove a property of an Erlang system in which processes are spawned on different nodes, then we can just ignore the node parameters. We translate `spawn/4` into `spawn/2` for the local creation of processes. Due to this fact, hierarchically created, distributed systems can be verified using our approach[1]. On the other hand, how can our approach be extended for open systems in which independently started processes can connect at runtime? Our solution is a translation of these systems into hierarchical systems. We add a registry process which spawns the independent processes and simulates the registration mechanism. Registered pids are stored in a list of name-pid tuples. If a message is sent to a registered name, then this message is in-

[1]Here we do not mean hierarchical in the sense of Section 4.5.2. A system is hierarchically created, if one main process is the ancestor of all other processes.

stead sent to the registry process. Here the message is forwarded to the corresponding pid.

We do not have the possibility to define global constants in Core Erlang. The processes which send a message to a registered process must send this message to the registry process now. Unfortunately, the pid of the registry process is unknown in these processes. Therefore, we add another argument to all functions: the pid of the registry process.

The registry process is a kind of database. It can be defined similarly to the database process of p_{db}:

```
registry(Regs) ->
      receive
         {register,Name,Pid} ->
                      case lookup(Name,Regs) of
                         {succ,P} -> error();
                         fail     -> registry([{Name,Pid}|Regs])
                      end;
         {unregister,Name}   -> registry(remove(Name,Regs));
         {regSend,Name,Msg}  -> lookup(name,Regs)!Msg,
                                registry(Regs)
      end.

error() -> X.

remove(K,L) -> case L of
                  []         -> [];
                  [{K,V}|L1] -> remove(K,L1);
                  [E|L1]     -> [E|remove(K,L1)]
               end.
```

The function `lookup` is defined as in the database. The registration of an already registered name yields a runtime error. For example, we can create this error by evaluating `error/0` to an unbound variable. The functions `register/2` and `unregister/1` can be defined as functions with the pid of the `registry` process as additional argument.

```
register(Reg,Name,Pid) -> Reg!{register,Name,Pid}.

unregister(Reg,Name) -> Reg!{unregister,Name}.
```

We do not formalize the translation here, but as an example we present the translation of the open version of the distributed database from Section 2.3.2:

```
main() -> spawn(database:start,[self]),
          spawn(client:start,[self]),
          spawn(client:start,[self]),
          registry([]).
```

```
database:start(Reg) -> register(Reg,database,self),
                       database([]).

client:start(Reg) -> Reg!{regSend,database,{connect,self}},
                     receive
                        {connect,DB} -> client(DB)
                     end.
```

Again, this translation does not work for arbitrary Erlang programs because sending to a pid and sending to registered process cannot be distinguished at compile time. For example, the function

```
f(X) -> X!42.
```

can be applied to a pid or a registered name. We only perform the translation of sending to registered names, if they are safe. We could also add a runtime check which distinguishes the two kinds of sends. As for dynamic function calls, sending to registered names is used explicitly in most Erlang programs and the described translation is sufficient.

8.5 Timeouts

In Erlang it is also possible to program using timeouts. We have not added this feature to Core Haskell. It is also not possible to translate these programs to Core Erlang programs as before.

The verification of real-time systems by means of model checking is a growing field of research [HNSY94, Alu99]. Erlang uses asynchronous communication. In a distributed setting, this communication is performed via the Internet by TCP/IP where no times can be guaranteed. Hence, it will be difficult to prove interesting real-time properties like specifying reliability times. Therefore, we omit real-time aspects of the verification. We handle the `after` expression as a non-deterministic branch and add the following rule to the abstract operational semantics:

$$\frac{-}{\Pi, (\pi, E[\texttt{receive}\ p_1\texttt{->}e_1\texttt{;}\ \ldots\ \texttt{;}p_m\texttt{->}e_m\ \texttt{after}\ t\ \texttt{->}\ e\ \texttt{end}], \mu) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, E[e], \mu)}$$

We do not distinguish the different possibilities for t. However, this is safe. We could be a little bit more precise and branch to e only if none of the patterns matches a value of μ in the first test. However, the timeout case e will usually not be superfluous. It should be considered in the verification of the formula.

We have defined a more robust version of the database p_{db}. This database does not block forever if it waits for the value of an allocated key. Instead it continues in the main loop after some seconds. Also for this modified database we are able to prove the properties from Chapter 5 automatically using the abstraction $\widehat{\mathcal{A}}_2$ and LTL model checking.

8.6 Exception Handling

In Core Erlang we have only defined one error state. If one process produces a runtime error, then the whole system crashes in our semantics. To simulate exception handling we must add exceptions to the formal semantics. There has been a lot of work on formal semantics of exceptions [BBC86, BJ90]. However, an integration of these approaches into our formal semantics is very expensive.

8.7 Linking

One of the most powerful mechanisms of Erlang is linking. If two processes are linked with each other and one of them dies, then the other one dies as well or it receives a special message if it set process flag before. The implementation of linking is more difficult than the registration of processes and sending messages to registered processes. Therefore, we cannot define a translation into Core Erlang for linking. We have to modify the operational semantics.

As a refinement of the SOS, we do not only consider runtime errors on top level (the system state err). Instead single processes may run into a local error state. Then we extend every process by a list of linked processes considering that linking is bidirectional. The behavior of a process in the case that a linked process dies depends on the value of the process flag `trap_exit`. If this flag is set (`process_flag(trap_exit,true)`), then the process receives an exit message if the linked process dies, otherwise it dies, too. We also add this flag to the state of a process:

$$Proc := Pid \times (\mathcal{E}(T_{\mathcal{C}}(Pid)) \cup \{\mathsf{error}\}) \times Mb \times \mathcal{P}(Pid) \times \mathbb{B}$$

The process flag is modified by

$$\frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\Pi, (\pi, E[\texttt{process\_flag(trap\_exit,}v\texttt{)}], \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, E[v], \mu, \mathcal{L}, v)}$$

links are established by

$$\frac{v = \pi' \in Pid}{\begin{array}{rl} & \Pi, (\pi, E[\texttt{link(}v\texttt{)}], \mu, \mathcal{L}, f) \| (\pi', e', \mu', \mathcal{L}', f') \\ \Longrightarrow & \Pi, (\pi, E[\mathbf{true}], \mu, \mathcal{L} \cup \{v\}, f) \| (\pi', e', \mu', \mathcal{L}' \cup \{\pi\}, f') \end{array}}$$

and removed by

$$\frac{v = \pi' \in Pid}{\begin{array}{rl} & \Pi, (\pi, E[\texttt{unlink(}v\texttt{)}], \mu, \mathcal{L}, f) \| (\pi', e', \mu', \mathcal{L}', f') \\ \Longrightarrow & \Pi, (\pi, E[\mathbf{true}], \mu, \mathcal{L} \setminus \{v\}, f) \| (\pi', e', \mu', \mathcal{L}' \setminus \{\pi\}, f') \end{array}}$$

The functional result of `link` and `unlink` is the boolean value `true`. The modification of the rules for error handling are represented in Figure 8.1. Using these rules, the errors are defined locally. Two new possible errors result from linking. If `link` and `unlink` are applied to a non-pid, then this yields an error.

$$\frac{F \text{ a predefined function and } F_{\mathcal{A}}(v_1, \ldots, v_n) \text{ is not defined}}{\Pi, (\pi, E[F(v_1, \ldots, v_n)], \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

$$\frac{v_1 \notin Pid}{\Pi, (\pi, E[v_1 \,!\, v_2], \mu, \mathcal{L}, f) \stackrel{!\,v_2}{\Longrightarrow} \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

$$\frac{X \text{ is a variable}}{\Pi, (\pi, E[X], \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

$$\frac{\mathsf{match}(p, v) = \mathsf{Fail}}{\Pi, (\pi, E[p{=}v], \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

$$\frac{\mathsf{casematch}((p_1, \ldots, p_n), v) = \mathsf{Fail}}{\Pi, (\pi, E[\texttt{case}\ v\ \texttt{of}\ m\ \texttt{end}], \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

$$\frac{v \notin Pid}{\Pi, (\pi, E[\texttt{link(}v\texttt{)}], \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

$$\frac{v \notin Pid}{\Pi, (\pi, E[\texttt{unlink(}v\texttt{)}], \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

Figure 8.1: Operational Semantics – Runtime Errors and Linking

We can define the effect of linking by means of some additional transitions. If a process crashes, then all linked processes are killed or informed with a message depending on the process flag.

$$\frac{e' \neq \mathsf{error}}{\begin{array}{rl} & \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L} \uplus \{\pi'\}, f) \| (\pi', e', \mu', \mathcal{L}', \mathsf{false}) \\ \Longrightarrow & \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f) \| (\pi', \mathsf{error}, \mu', \mathcal{L}', \mathsf{false}) \end{array}}$$

$$\frac{e' \neq \mathsf{error}}{\begin{array}{rl} & \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L} \uplus \{\pi'\}, f) \| (\pi', e', \mu', \mathcal{L}', \mathsf{true}) \\ \Longrightarrow & \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f) \| (\pi', e', \mu' : \mathbf{\{'EXIT', \pi\}}, \mathcal{L}', \mathsf{true}) \end{array}}$$

In the first case, (the `trap_exit` flag is **false**) the linked process is terminated, too. Otherwise the special $'\mathsf{EXIT}'$ message is sent to the linked processes. It also contains the pid of the died process. For sake of simplicity, we ignore the reasons for the termination here. For the verification of real Erlang applications they are necessary. On the other hand the technical effort is very large and for many Erlang programs linking without considering the reason will suffice.

The informed process is eliminated from the set of linked processes. Hence, all linked processes are successively informed only once. In both cases we first check that the other process is still alive. This prevents from superfluous interaction between dead linked processes.

Using this extension, we can now verify Erlang systems with linked processes. At the moment we can only consider runtime errors which are a result of program errors. In distributed systems we are more interested in robustness against the crash of some nodes or the loose of the connection on hardware level. To model this we additionally define a set $\mathcal{D} \subseteq \mathit{Pid}$ of processes which may crash during their execution. We model the crash of one of these processes by the rule:

$$\frac{\pi \in \mathcal{D}}{\Pi, (\pi, e, \mu, \mathcal{L}, f) \Longrightarrow \Pi, (\pi, \mathsf{error}, \mu, \mathcal{L}, f)}$$

All states in the semantics are considered as possible positions for the crash of processes of $\mathcal{D}$. Using this extension, we can also prove properties about the robustness of the system.

In this section we have extended the SOS by linking. For the formal verification we need the AOS. However, an extension of the AOS by linking can be defined similarly. In some cases we must additionally apply the abstraction function to some concrete values. For instance, for the result of `link` and `unlink` or the $'\mathsf{EXIT}'$ tuple. Furthermore, we must consider values as their pid representation if we expect a pid. Again, the abstraction can yield some additional non-determinism. However, we do not need to extend our framework for abstract interpretations. Linking does not request new properties for the abstract interpretation.

An Example

As an example for a robust system we have defined a chat: A chat server holds a list of chat client pids. New clients can connect to the chat and are added to the list. If a client sends a chat message to the chat server, then the server broadcasts this message to all chat clients. Finally, chat clients can logout.

To guarantee robustness of the chat server we established a link between the chat server and every client which connects to the chat. If a client crashes, then the server removes its pid from the database.

We consider a system of one chat server and two chat clients. For this system, using the presented approach, we are able to proved that the list of processes is updated if a client dies. The global state of the database contains all pids of connected clients. We use this state as a proposition to check that after the crash of a client this client is not in the list anymore. To prove the property we cannot abstract from the list of clients. We have two clients. Therefore, a list of two elements for the state of the chat server is sufficient to check the property. This can be realized using the abstraction $\widehat{\mathcal{A}}_3$ in which lists of length two are not abstracted.

Chapter 9

Related Approaches for Software Verification

There exist some other approaches for the formal verification of concurrent and distributed systems. They can be divided in approaches which use theorem proving and approaches which use model checking in combination with abstraction.

9.1 Theorem Provers

For the formal verification of Erlang there exists only the alternative to use the Erlang Verification Tool (EVT) developed by Mads Dam et. al. [DF98, ADFG98, NFG01]. They implemented a theorem prover which is extended by an operational interleaving semantics for a core language similar to Core Erlang. They use an extension of the propositional μ-calculus [KP84] for the specification of properties. State propositions can be expressed by means of predefined predicates over system states.

EVT supports the verification of program properties by hand. The verification is guided by the tool. Only correct steps can be performed in the proof. However, there is no support in finding a proof. The whole structure of the prove has do be developed by hand. Especially for fixed point formulas like "until" in LTL, this can be difficult because an ordering has to be defined to cut off infinite proof trees by an induction rule. Hence, the use of EVT is very difficult to learn. Like our approach, EVT defines an operational interleaving semantics. Therefore, they obtain the same problems with state space explosion as we do. Furthermore, they have to prove system properties for every of these interleaving paths by hand. Due to this fact, proofs in EVT are very expensive and a practical use seems not to be feasible.

The advantage of this approach is that the proofs are not restricted to concrete systems. EVT provides variables for sub-systems, processes, mailboxes, and expres-

sions. For example, it is possible to prove properties of subcomponents of arbitrary systems or processes with arbitrary mailboxes using these variables.

For future work Mads Dam et. al. plan to integrate automatic verification like model checking in EVT. This is a possible point for the integration of the two approaches. The tool could first try to use our verification approach. If this does not succeed, then the property or the system can be simplified by theorem proving. This can be iterated until the property is fulfilled or the failed proof yields a counter example. Another alternative for the integration of theorem proving and our approach is already discussed in Section 5.5.3: for the proof that an abstract interpretation fulfills the properties (P1)–(P5) theorem proving is used. Then the correctness of the concurrent behavior of a system can automatically be proven in our approach. We think that this is more feasible in practice because the problems resulting from the state space explosion can be handled automatically in model checking (computers can handle large system better than humans). On the other hand, creativity is needed for the design of the abstract interpretation. The prove of the safeness of an abstract interpretation could be done by a theorem prover which can especially help for the frequently occurring structural inductions.

As a consequence of the described disadvantages of the theorem prover EVT, some of the developers started to use model checking, to prove properties of Erlang programs [AE01, AED02]. They use model checking for the verification of Erlang programs without any abstraction. This restricts their approach to systems which do not use arbitrary values. Furthermore, they obtain very large systems which could be reduced by abstraction. They translate the Erlang programs into the term rewriting system μ-CRL [GP95], for which a model checking component exists. Considering other examples, they realize the limitations of their approach and want to integrate our abstraction techniques into their translations.

9.2 Model Checkers for other Programming Languages

Erlang is not the only programming language for which the verification by model checking is investigated. For concurrent Java programs there are two projects for the formal verification by model checking: Java PathFinder [HP98] and the Bandera Tool [DP98, HDL98]. Both approaches translate Java source code into other Specification formalisms: Both translate into Promela, the specification language of SPIN [Hol95]. Additionally, Bandera provides translations to PVS [Rus97] and SMV [CGL94a]. In both approaches, methods can be explicitly abstracted in the source code, by redefining methods. Their work on the formalization of a framework for abstract interpretations is poor. For imperative languages abstraction seems to be simpler because only the connection (P1) of our framework must be guaranteed between the concrete and the abstract domain. This connection must be proven for every abstracted method and the resulting abstract transition system is safe with respect to the concrete semantics. In further works the authors have investigated how the resulting transition systems can be reduced and how different data structures can be

abstracted. The approach is restricted to concurrent Java Programs but an extension to distributed Java programs and communication using Remote Method Invocation is possible [SL01].

The results cannot be transferred to Erlang because the main mechanism of branching in Erlang is pattern matching. More complicated techniques like the ones presented in this thesis are needed to formalize abstraction of Erlang programs. The idea of a translation to other specification languages can also not be transfered to Erlang so easily because the functional language Erlang differs from usually imperative specification languages like Promela. A translation is much more difficult than for Java. The opportunity of such a translation is the very efficient implementation of the underlying model checkers which can handle very large state spaces. In this thesis we wanted to concentrate on the formalization and understanding of the presented approach. As future work, we will pursue translations to specification languages of other model checkers.

Although these model checkers are very well optimized, there are two disadvantages of such a translation: Firstly, the generation of counter examples is difficult. The model checker yields a counter example in its own specification formalism. This must be retranslated to an Erlang counter example. Secondly, the reduction of the state space by means of partial order techniques will probably be less efficient. Though, at first sight, partial order reduction is implemented in most of these model checkers, this partial order reduction will not be as efficient as the one we discussed in Section 6.2. For the application of partial order reduction, it is necessary to know the independent actions of the transition system. In our reduction these are all actions, except the sending actions and the prop labels. However, after a translation to another specification formalism this independence will probably vanish. Partial order reduction is less efficient. An investigation of these connections seems to be interesting and will likely be future work.

Chapter 10

Conclusions

The aim of this thesis was the formal verification of concurrent and, distributed Erlang systems by model checking. To formalize the verification problem, we first developed a formal syntax and semantics of a core fragment of Erlang. Respecting the semantics of Core Erlang, we developed a framework for abstract interpretations of Core Erlang programs. Using this framework, we defined an abstract operational semantics. The framework guarantees that the AOS is safe with respect to the SOS: in the AOS all paths of the SOS are represented. For finite domain abstract interpretations and some restrictions to Core Erlang programs, the resulting AOS defines a finite transition system. This finite transition system can be used for the formal verification by model checking. A property which is fulfilled in the AOS, is also valid in the SOS.

In Chapter 5 we discussed some problems in the application of LTL model checking to the AOS. Considering abstraction we must distinguish, if a proposition must be fulfilled or refuted. This depends on the number of negations in front of the proposition. We used our verification technique to verify different properties of the database example: we proved the absence of deadlocks, a liveness property and used fairness properties as an assumption for the verification of a system consisting of several independent databases and clients.

We discussed some extensions and optimizations in Chapter 6: Firstly, we defined a simplified framework for abstract interpretations containing good predefined abstractions for casematch and mbmatch. Secondly, we discussed the application of partial order reduction to the AOS. This yields an enormous reduction of the state space and is needed for the practical use of our approach.

Unfortunatly, the AOS of finite domain abstractions defines a finite transition system only for a subclass of Core Erlang programs. If non-tail recursive function calls are used in the program, then the AOS is infinite and verification by model checking is not possible. As a solution, we presented an abstraction of the control flow in Chapter 7. Using this abstraction our approach can be used for the verification of a larger

class of Core Erlang programs. Although Core Erlang contains the main features of Erlang, our approach should be applicable to real Erlang programs. In Chapter 8 we showed how it can be extended to the verification of real Erlang programs.

The presented approach is implemented as a prototype in the programming language Haskell. All presented properties of the database and its optimized version could be automatically be proven using this prototype. Furthermore we verified some properties of a counter, a locker, and a chat implemented in Core Erlang.

For future work we want to improve the prototype to be able to verify larger systems and add a graphical user interface. At this, we also want to investigate the relationship between formulas and partial order reduction in more detail. We hope to obtain further reductions of the state space. Thereafter, we want to investigate how our approach can be used in practice and verify larger systems which are used in larger project within Ericsson. An example is the generic server module [AN01] which provides process abstractions for client server applications and is widely used in Erlang systems. In this context it would also be interesting to translate Core Erlang programs to Promela, the specification language of SPIN [Hol95]. SPIN contains a very efficient LTL model checker and it could be possible to verify larger Erlang systems using this translation. On the other hand, Promela is an imperative language without recursion. Therefore, a translation will not be as straight forward as it was done for Java. A special problem will be the re-translation of counter examples.

Another point of future work could be the extension of our approach to higher order functions. Our approach is limited to first order programs which is sufficient for practice, because most Erlang programs do not use higher order functions. However, higher-order is possible in Erlang. In Section 8.3 we already discussed a possible extension but for arbitrary programs we need additional abstraction in this case. Although, higher order functions are less relevant for the verification in practice, an extension of our approach to higher-order functions seems interesting from the theoretical point of view.

Bibliography

[ADFG98] Thomas Arts, Mads Dam, Lars-åke Fredlund, and Dilian Gurov. System description: Verification of distributed Erlang programs. In *Proceedings of the 15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Sience*, pages 38–41, 1998.

[AE01] Thomas Arts and Clara Benac Earle. Development of a verified Erlang program for resource locking. In *Formal Methods in Industrial Critical Systems*, Paris, France, July 2001.

[AED02] Thomas Arts, Clara Benac Earle, and John Derrick. Verifying Erlang code: a resource locker case-study. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1–2):133–178, April 2000.

[Alu99] R. Alur. Timed automata. In *Proc. 11th International Computer Aided Verification Conference*, pages 8–22, 1999.

[AN01] Thomas Arts and Thomas Noll. Verifying generic Erlang client–server implementations. In *Proceedings of the 12th International Workshop on the Implementation of Functional Languages (IFL'00)*, volume 2011 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2001.

[ANN98] Torben Amtoft, Hanne Rijs Nielson, and Flenning Nielson. Behaviour analysis and safety conditions: A case study in CML. *Lecture Notes in Computer Sience*, 1382:255–269, 1998.

[Ast91] Egidio Astesiano. Inductive and operational semantics. In Erich J. Neuhold and Manfred Paul, editors, *Formal Description of Programming Concepts*, pages 51–136. Springer, 1991.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[BBC86] Gilles Bernot, Michel Bidoit, and Christine Choppy. Algebraic semantics of exception handling. In Bernard Robinet and Reinhard Wilhelm, editors, *Proceedings of the European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 173–186. Springer, 1986.

[BC89] Gerard Boudol and Ilaria Castellani. Permutation of transitions: an event structure semantics for CCS and SCCS. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *REX School and Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Sience*, pages 411–427. Springer, 1989.

[BE97] Olaf Burkart and Javier Esparza. More infinite results. *Bulletin of the European Association for Theoretical Computer Science*, 62:138–159, June 1997. Columns: Concurrency.

[BJ90] Jean-Chrysostome Bolot and Pankaj Jalote. Functional semantics of programs with exceptions. *Computer Languages*, 15(4):251–265, 1990.

[BJR99] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[BL01] Benedikt Bollig and Martin Leucker. Modelling, Specifying, and Verifying Message Passing Systems. In Claudio Bettini and Angelo Montanari, editors, *Proceedings of the Symposium on Temporal Representation and Reasoning (TIME'01)*, pages 240–248. IEEE Computer Society Press, 2001.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, 1998.

[Bro91] Manfred Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3(1):21–57, 1991.

[BRS99] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *Proc. 11th International Computer Aided Verification Conference*, pages 222–235, 1999.

[BS92] Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. In W. R. Cleaveland, editor, *Proceedings of the Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137, Stony Brook, New York, 24–27August 1992. Springer.

[BW91] Jaco W. de Bakker and H. A. Warmerdam. Four domains for concurrency. *Theoretical Computer Science*, 90(1):127–149, November 1991.

[BZ82] Jaco W. de Bakker and Jeffery I. Zucker. Denotational semantics of concurrency. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 153–158, San Francisco, California, 5–7 May 1982.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CC77b] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, pages 1–12, August 1977.

[CGH94] Edmund Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. In *Proc. 6th International Computer Aided Verification Conference*, pages 415–427, 1994.

[CGH97] Edmund Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, February 1997.

[CGL94a] E. M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings REX School/Symposium: A Decade of Concurrency,* Noordwijkerhout, The Netherlands, June 1993, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer, 1994.

[CGL94b] Edmund Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGMP98] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2, 1998.

[Che97] Allan Cheng. Petri nets, traces, and local model checking. *Theoretical Computer Science*, 183(2):229–251, September 1997.

[CPS90] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Sience*, pages 24–37. Springer, June 1990.

[Cur63] Haskell B. Curry. *Foundations of Mathematical Logic*. McGraw-Hill, New York, 1963.

[DB97] Mourad Debbabi and Dominique Bolignano. A semantic theory for ML higher-order concurrency primitives. In *ML with Concurrency: Design, Analysis, Implementation and Application*, Monographs in Computer Science, pages 145–184. Springer, 1997.

[DF98] Mads Dam and Lars-åke Fredlund. On the verification of open distributed systems. In *Proc. of the ACM Symposium on Applied Computing*, 28:532–540, June 1998.

[DP98] Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundation of Software Engineering*, November 1998.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[Esp97] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[FFKD87] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[GG88] Stephen J. Garland and John V. Guttag. Inductive methods for reasoning about abstract data types. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 219–228, New York, NY, USA, 1988. ACM Press.

[GP95] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM SIGACT and SIGPLAN, ACM Press, 1980.

[GPVW95] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[Gro01] William Grosso. *Java RMI – Designing and building distributed applications.* O'Reilly & Associates, Inc., Newton, USA, 2001.

[HDL98] John Hatcliff, Matthew B. Dwyer, and Shawn Laubach. Staging static analyses using abstraction-based program specialization. In *Proceedings of the International Symposium PLILP/ALP 98*, volume 1490 of *Lecture Notes in Computer Sience*, pages 134–148, 1998.

[HN00] Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with ports. In *Proceedings of the 12th International Workshop on the Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 107–121, 2000.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.

[Hog89] Dieter Hogrefe. *Estelle, Lotos und SDL.* Springer, Berlin, 1989.

[Hol95] Gerard J. Holzmann. Proving properties of concurrent systems with SPIN. In *Proceedings of the Sixth International Conference on Concurrency Theory*, volume 962, pages 453–455, 1995.

[Hol96] Gerard J. Holzmann. On-the-fly model checking. *ACM Computing Surveys*, 28(4es):120–120, December 1996.

[HP98] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 1998.

[HU79] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Language, and Computation.* Addison–Wesley, Reading, MA, 1979.

[Huc99a] Frank Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[Huc99b] Frank Huch. Verification of Erlang programs using abstract interpretation and model checking – extended version. Technical Report 99–02, RWTH Aachen, 1999.

[Huc01] Frank Huch. Model checking Erlang programs - abstracting the context-free structure. In Scott D. Stoller and Willem Visser, editors, *Proceedings of the Workshop on Software Model Checking*, volume 55–03 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[Huc02] Frank Huch. Model checking erlang programs - abstracting recursive function calls. In Michael Hanus, editor, *Proceedings of the International Workshop Functional and (Constraint) Logic Programming*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[J+98] Simon Peyton Jones et al. Haskell 98 report. Technical report, http://www.haskell.org, 1998.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.

[JN94] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[KP84] Dexter Kozen and Rohit Parikh. A decision procedure for the propositional μ-calculus. In E. Clarke and D. Kozen, editors, *Proceedings 2nd Workshop on Logics of Programs, CMU, Pittsburgh, PA, USA, 6–8 June 1983*, volume 164 of *Lecture Notes in Computer Science*, pages 313–325. Springer, 1984.

[KP98] Yonit Kesten and Amir Pnueli. Modularization and abstraction: The keys to practical formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS 1998)*, volume 1450 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 1998.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT-SIGPLAN, ACM Press.

[Lys94] Olav Lysne. Heuristics for completion in automatic proofs by structural induction. *Nordic Journal of Computing*, 1(1):135–156, Spring 1994.

[Maz84] Antoni Mazurkiewicz. Traces, histories, graphs: Instances of a process monoid. In M. P. Chytil et al., editors, *Proceedings of the 11th Symposium on Mathematical Foundations of Computer Science (MFCS'84)*, volume 176 of *Lecture Notes in Computer Sience*, pages 115–133, Berlin, 1984. Springer.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, Berlin, first edition, 1980.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[NFG01] Thomas Noll, Lars-åke Fredlund, and Dilian Gurov. The Erlang verification tool. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 582–585. Springer, 2001.

[NN92] Hanne Rijs Nielson and Flenning Nielson. *Semantics with Applications : A Formal Introduction*. John Wiley & Sons, Chichester, 1992.

[NN97] Hanne Riis Nielson and Flemming Nielson. Communication analysis for concurrent ML. In *ML with Concurrency*, Monographs in Computer Science, pages 185–235. Springer, 1997.

[Nol01] Thomas Noll. A rewriting logic implementation of Erlang. In Mark van den Brand and Didier Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications*, volume 44–02 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[Pac97] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, CA, 1997. http://www.usfca.edu/mpi (source programs available) and http://www.mkp.com.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.

[Pau93] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the Sixth Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Sience*, pages 377–390, 1994.

[PR97] Prakash Panangaden and John H. Reppy. The Essence of Concurrent ML. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation and Application*, Monographs in Computer Science, pages 5–29. Springer, 1997.

[Ros84] A. William Roscoe. Denotational semantics for occam. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 306–329. Springer, 1984.

[Rus97] John Rushby. Specification, proof checking, and model checking for protocols and distributed systems with PVS. Tutorial presented at {FORTE X/PSTV XVII '97}, November 1997.

[SL01] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In *Proceedings of the 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Sience*, pages 192–199. Springer, 2001.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice-Hall Software Series, pages 189–233. Prentice-Hall, Englewood Cliffs , NJ , USA, 1981.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.

[SS98] David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *Proceedings of the Fifth International Statics Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Sience*, pages 351–380, 1998.

[Tan92] Andrew S. Tannenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Tau89] Dirk Taubner. *Finite representations of CCS and TCSP programs by automata and Petri nets*, volume 369 of *Lecture Notes in Computer Science*. Springer, New York, NY, USA, 1989.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298, Pisa, Italy, 26–29 August 1996. Springer.

[TLK97] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. FACILE — From Toy to Tool. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation and Application*, Monographs in Computer Science, pages 97–144. Springer, 1997.

[Var96] Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, New York, NY, USA, 1996.

[Wal98] Frank Wallner. Model checking LTL using net unfoldings. In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 207–218, 1998.

[Wei90] William E. Weihl. Remote Procedure Call. In Sape J. Mullender, editor, *Distributed Systems*, pages 37–64. ACM Press, 1990.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.

Lebenslauf

Name:	Frank Günter Huch
geboren am:	01. August 1969
in:	Neuss
Aug. 1975 – Jul. 1979	Grundschule in Neuss-Reuschenberg
Aug. 1979 – Jun. 1988	Humboldt-Gymnasuium in Neuss Abschluss Abitur
Nov. 1988 – Jun. 1990	Zivildienst
Okt. 1990 – Jul. 1996	Studium der Informatik mit Nebenfach Medizin an der RWTH Aachen Abschluss Diplom
Sept. 1996 – Jul. 2001	Wissenschaftlicher Angestellter am Lehrstuhl für Informatik II der RWTH Aachen
seit Aug. 2001	Wissenschaftlicher Assistent am Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion der Christian-Albrechts-Universität zu Kiel