# Verification of Erlang Programs using Abstract Interpretation and Model Checking

Frank Huch

Lehrstuhl für Informatik II, RWTH Aachen,
52056 Aachen, Germany
huch@informatik.rwth-aachen.de

**Abstract**

We present an approach for the verification of Erlang programs using abstract interpretation and model checking. In general model checking for temporal logics like LTL and Erlang programs is undecidable. Therefore we define a framework for abstract interpretations for a core fragment of Erlang. In this framework it is guaranteed, that the abstract operational semantics preserves all paths of the standard operational semantics. We consider properties that have to hold on all paths of a system, like properties in LTL. If these properties can be proved for the abstract operational semantics, they also hold for the Erlang program. They can be proved with model checking if the abstract operational semantics is a finite transition system. Therefore we introduce a example abstract interpretation, which has this property. We have implemented this approach as a prototype and were able to prove properties like mutual exclusion or the absence of deadlocks and lifelocks for some Erlang programs.

Keywords: abstract interpretation, verification, model checking, Erlang, distributed system

## 1 Introduction

Growing requirements of industry and society impose greater complexity of software development. Consequently understandability, maintenance and reliability can not be warranted. The best proof for this is given by the problems resulting of the year 2000.

Things get even harder whenever leaving the sequential territory and developing distributed systems. Here many programs run concurrently and interact via communication. This can e.g. yield problems like deadlocks or lifelocks. For the verification of concurrent systems model checking is a useful technique: A concurrent system is specified in a specification formalism, like CCS and the properties a system should have are specified in a formula, usually in temporal or modal logic. Under the assumption, that the operational semantics of the CCS specification builds a finite transition system the properties can be verified. In the next steps the specification gets more and more refined without changing the concurrent behaviour until you get the program.

In practice formal methods are not used in this strict kind. First the system is a little bit specified and then people implement what they understand of the specification by hand and during the implementation the specifications are even changed again. Another problem is, that specification formalisms like UML or SDL are used, which do have too much expressive power and model checking gets undecidable. They usually do not even have a formalized operational semantics and it is not clear how to build a Kripke structure the formula can be checked on.

We want to extend model checking to programs written in real programming languages. This leads to the problem that the model checking problem in general is undecidable for system implementations using programming languages and properties described in interesting logics. Hence we need abstraction [CC77, JN94, SS98]. We have found a good abstract interpretation, with the property, that the transition system defined by the abstract operational semantics includes all paths of the standard operational semantics (*sop*) and is, under some assumptions to the program, finite state. Because the abstraction can sometimes have more paths than the *sop*, it is only possible to prove properties that have to be fulfilled on all paths, like in linear time logic (LTL). If the abstraction fulfills a property expressed in LTL, also the program fulfills it, but not vice versa. Because the abstraction has a finite transition system, model checking is decidable here [LP85, Var96].

In industry the programming language Erlang [AWV93] is used for the implementation of distributed systems. Erlang is a strict functional language with additional features for the creation of processes and the communication between processes independently of a process being executed on the same computer or somewhere else in the Internet. We present an abstract interpretation tailored for Erlang.

In Section 2 we define the syntax for a core fragment of Erlang and formalize an operational semantics in Section 3. A framework for abstract operational semantics over abstract interpretations for Erlang programs is defined in Section 4 and two example abstractions are presented in Section 5. The decidability and an optimization is discussed in Section 6 and an example is verified in Section 7. Finally we conclude and discuss future work in Section 8.

1999 ACM SIGPLAN International Conference on Functional Programming (ICFP '99)

## 2 Core Erlang – Syntax

Let $\Sigma$ be a signature for the predefined function symbols with arity, e.g. a set of them. As an example $\texttt{+}/2 \in \Sigma$. Let $Var = \{X, Y, Z, \ldots\}$ be a set of variables and $Atoms$ a set of atoms, e.g. $\{1, 2, \texttt{fail}, \texttt{value}, \ldots\}$. Let $CF$ be a set of constructor functions with arity. In Erlang the set of constructors is fixed to

$$CF = \{[.|.]/2, []/0\} \cup \{\{\ldots\}/n \mid n \in \mathbb{N}\} \cup \{a/0 \mid a \in Atoms\}, \quad (1)$$

a constructor for building lists, a constructor for the empty list, constructors for building tuples of any arity and the atoms as constructor terms with arity 0.

The syntax of Core Erlang is defined as follows:

$$\begin{array}{lcl} p & ::= & f(X_1, \ldots, X_n) \texttt{ -> } e\texttt{.} \mid p\ p \\ e & ::= & \phi(e_1, \ldots, e_n) \mid X \mid pat \texttt{ = } e \mid \texttt{self} \mid \\ & & e_1\texttt{,}e_2 \mid e_1\texttt{!}e_2 \mid \texttt{case } e \texttt{ of } m \texttt{ end} \mid \\ & & \texttt{receive } m \texttt{ end} \mid \texttt{spawn(}f\texttt{,}e\texttt{)} \\ m & ::= & pat_1\texttt{->}e_1\texttt{;} \ldots \texttt{;}pat_n\texttt{->}e_n \\ pat & ::= & c(pat_1, \ldots, pat_n) \mid X \end{array}$$

All defined functions of a program build, extended with their arity, the set $FS(p)$. $\phi$ is an abbreviation for $f/n \in FS(p)$, $F/n \in \Sigma$ and $\underline{c}/n$, where $c/n \in CF$. We distinguish the constructors for building constructor terms from those used in pattern matching, because we will later handle them in different ways ($\underline{CF} := \{\underline{c}/n \mid c/n \in CF\}$). In every Core Erlang program a main function is defined: $\texttt{main}/0 \in FS(p)$. All possible Core Erlang terms $e$ build the set $T(\emptyset)$. The set $T(S)$ is defined by adding the possibility $e \ ::= \ v \in S$ for Core Erlang expressions.

**Example 2.1** An example, we will later reuse, is the Core Erlang program $p_0$ defined as:

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

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
                 []          -> fail;
                 [{K,V}|_]   -> {succ,V};
                 [_|Xs]      -> lookup(K,Xs)
               end.
```

The program creates a database process, that represents a state in which the database information is saved. The database is represented by a list of tuples consisting of a key and the corresponding value. The interface of the database are the messages `{allocate,Key,P}` and `{lookup,Key,P}` to allocate or lookup entries. The allocation is done in two steps. First the key is received and checked. If there is no conflict, the corresponding value can then be received and saved in the database. This exchange of messages in more than one step has to guarantee mutual exclusion on the database, because otherwise it could be possible that two processes send keys and values to the database and they are saved in the wrong combination. We will later prove that the database combined with two accessing clients will fulfill this property.

## 3 Core Erlang – Semantics

The semantics of Erlang is informally described in [AWV93]. For our aim we have to formalize it. In [DaF98] a formal semantics, inspired by the structured operational semantics of CCS and the $\pi$-calculus, is defined. This semantics is complicated and asynchronous communication is modeled as synchronization between the sending process and the receiving message queue respectively the message queue and the receiving process. Here we present a more direct approach in which the asynchronous communication an the queues are modeled directly:

During the execution of an Erlang program several processes run concurrently. Every process is identified by a unique process identifier (pid) and other processes need to know this pid to send messages to it. The received messages are buffered in a message queue and can be extracted by pattern matching. To provide mobility pids are values that can be saved in data structures and can also be send to other processes. We have to model this too and allow constructor terms over pids as possible values of the calculation ($CT^{CF}_{Pid}$), where $Pid$ is an infinite set of process identifiers, distinguishable from $Atoms$. Constructor terms over a set $S$ and $CF$ are the smallest set with:

$$\begin{array}{l} S \subseteq CT^{CF}_S, \\ c/n \in CF \text{ and } t_1, \ldots, t_n \in CT^{CF}_S \\ \qquad\qquad \Longrightarrow c(t_1, \ldots, t_n) \in CT^{CF}_S \end{array}$$

For Erlang $CF$ is fixed (1) and we omit $CF$ and just write $CT_S$.

Furthermore we assume the existence of an interpretation function $\iota : \Sigma \to (CT_{Pid} \to CT_{Pid})$ for the predefined function symbols, which assigns a (partial) function over calculation terms to every predefined function symbol. Both are combined in the interpretation $\mathcal{A} = (CT_{Pid}, \iota)$ and we write $F_{\mathcal{A}}$ instead of $\iota(F)$.

All possible patterns of a program are defined as $Pat = CT_{Var}$. A state of the execution 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 term over $CT_{Pid}$, and a message queue

$$Proc := Pid \times T(CT_{Pid}) \times Que,$$

and a queue is a word over ground terms of an execution

$$Que := {CT_{Pid}}^*.$$

The operational semantics is denoted by the relation $\Longrightarrow \subset State \times Label \times State$, where $Label$ is the set of actions relevant

## Sequential reduction:

$$\frac{-}{\Pi,(p,E[v\texttt{,}e],q) \Longrightarrow \Pi,(p,E[e],q)} \qquad \frac{F \text{ a predefined function}}{\Pi,(p,E[F(v_1,\ldots,v_n)],q) \Longrightarrow \Pi,(p,E[F_{\mathcal{A}}(v_1,\ldots,v_n)],q)}$$

$$\frac{-}{\Pi,(p,E[\underline{c}(v_1,\ldots,v_n)],q) \Longrightarrow \Pi,(p,E[c(v_1,\ldots,v_n)],q)} \qquad \frac{-}{\Pi,(p,E[\texttt{self}],q) \Longrightarrow \Pi,(p,E[p],q)}$$

$$\frac{f(X_1,\ldots,X_n)\texttt{->}e\texttt{.} \in p \quad \text{and} \quad e' = \text{"}e \text{ with fresh free variables"}}{\Pi,(p,E[f(v_1,\ldots,v_m)],q) \Longrightarrow \Pi,(p,E[e'[X_1/v_1,\ldots,X_m/v_m]],q')}$$

## Matching:

$$\frac{\text{match}(pat,v)=\rho}{\Pi,(p,E[pat\texttt{=}v],q) \Longrightarrow \Pi,(p,\rho(E[v]),q)}$$

$$\frac{\text{casematch}((pat_1,\ldots,pat_m),v)=(i,\rho)}{\Pi,(p,E[\texttt{case } v \texttt{ of } pat_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}pat_m\texttt{->}e_m \texttt{ end}],q) \Longrightarrow \Pi,(p,\rho(E[e_i]),q)}$$

$$\frac{\text{queuematch}((pat_1,\ldots,pat_m),(v_1,\ldots,v_u))=(i,j,\rho)}{\begin{array}{l}\Pi,(p,E[\texttt{receive}\,pat_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}pat_m\texttt{->}e_m\,\texttt{end}],(v_1,\ldots,v_j,\ldots,v_u)) \\ \overset{?v_j}{\Longrightarrow} \Pi,(p,\rho(E[e_i]),(v_1,\ldots,v_{j-1},v_{j+1},\ldots,v_u))\end{array}}$$

## Concurrent reduction:

$$\frac{f(X_1,\ldots,X_n)\texttt{->}e\texttt{.} \in p \quad \text{and} \quad v_1,\ldots,v_n \in CT_{Pid} \quad \text{and} \quad p' \text{ a new pid}}{\begin{array}{l}\Pi,(p,E[\texttt{spawn}(f,\texttt{[}v_1,\ldots,v_n\texttt{]}^1)],q) \\ \overset{\texttt{spawn}(f)}{\Longrightarrow} \Pi,(p,E[p'],q),(p',e[X_1/v_1,\ldots,X_n/v_n],())\end{array}}$$

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

## Runtime errors:

$$\frac{v_1 \text{ is not a pid}}{\Pi,(p,E[v_1\texttt{!}v_2],q) \overset{!v_2}{\Longrightarrow} \text{error}} \qquad \frac{X \text{ is a variable}}{\Pi,(p,E[X],q) \Longrightarrow \text{error}}$$

**Figure 3.1**: Operational Semantics

for concurrency:

$$\begin{array}{rl} Label := & \{\texttt{!}v \mid v \in CT_{Pid}\} \cup \{\texttt{?}v \mid v \in CT_{Pid}\} \cup \\ & \{\texttt{spawn(}f\texttt{)} \mid f/n \in FS(p)\} \cup \{\varepsilon\} \end{array}$$

We write $s \overset{a}{\Longrightarrow} s'$ for $(s,a,s') \in \Longrightarrow$ and $s \Longrightarrow s'$ for $s \overset{\varepsilon}{\Longrightarrow} s'$. For the definition of the leftmost innermost operational semantics, we use the technique of reduction contexts [FFKD87], which specifies where the next step of computation may take place:

$$\begin{array}{rl} E ::= & [\,] \mid \phi\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_i\texttt{,}E\texttt{,}e_{i+2}\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mid E\texttt{,}e \mid pat\texttt{ = }E \\ & \texttt{spawn(}f\texttt{,}E\texttt{)} \mid E\texttt{!}e \mid v\texttt{!}E \mid \texttt{case } E \texttt{ of } m \texttt{ end} \end{array}$$

Here $v_j$ denotes an evaluated expression ($v_j \in CT_{Pid}$), $E$ the subterm the reduction context is in and $e_k/m$ the parts which cannot be reduced. $[\,]$ is called the hole and marks the point for the next computation. We shall then write $E[e]$ for the expression $E$ with the hole replaced by $e$ and the next step of the computation takes place here.

The definition of the operational semantics $\Longrightarrow$ is presented in Figure 3.1. The rules for sequential reduction are almost standard. A sequence reduces to the second argument, if the first one is a reduced value, predefined functions are reduced with respect to the interpretation and constructor functions are interpreted freely. `self` reduces 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 term which is replaced and the context, we have to rename the free variables of the

[1] $\texttt{[}v_1,\ldots,v_n\texttt{]}$ is an abbreviation for $\texttt{[}v_1\texttt{|[}\ldots\texttt{|[}v_n\texttt{|[]]}\ldots\texttt{]]}$.

inserted term.
In the rules for pattern matching, `case` and `receive` we use the functions match, casematch and queuematch to model Erlangs mechanism of pattern matching. Compared to other functional languages pattern matching in Erlang is more complicated, because of non-linear patterns with multiple occurrences of the same variable.

$$\begin{array}{l}\text{match}(X,t) = [X/t]\\ \text{match}(c(pat_1,\ldots,pat_n),c(v_1,\ldots,v_n)) =\\ \qquad\qquad \text{match}(pat_1,v_1) \uplus \ldots \uplus \text{match}(pat_n,v_n)\\ \text{match}(\_,\_) = \text{Fail, otherwise}\end{array}$$

Two derived substitutions can only be unioned $\uplus$ if the overlapping parts are identical. Otherwise the result is Fail:

$$\begin{array}{l}\text{Fail} \uplus \sigma = \text{Fail}\\ \sigma \uplus \text{Fail} = \text{Fail}\\ \sigma_1 \uplus \sigma_2 = \begin{cases} \sigma_1 \cup \sigma_2, \text{ if } \forall X \in (\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2)) : \\ \qquad\qquad \sigma_1(X) = \sigma_2(X) \\ \text{Fail} \quad \text{, otherwise} \end{cases}\end{array}$$

`case` reduces to the expression corresponding to the first pattern, that matches a given value. The function casematch yields a tuple consisting of the number of the first matching pattern and the corresponding substitution or Fail, if non of the patterns match the given value. Accordingly to this result `case` is reduced and the substitution is applied to the whole term to propagate the bindings yield by the pattern matching:

$$\begin{array}{l}\text{casematch}((pat_1,\ldots,pat_m),v)\\ \quad = \begin{cases} (i,\rho), \text{ if match}(pat_i,v) = \rho \text{ and} \\ \qquad\qquad \text{match}(pat_j,v) = \text{Fail} \quad \forall j < i \\ \text{Fail} \quad \text{, otherwise} \end{cases}\end{array}$$

`receive` reduces in the same manner, but all values in the queue have to be considered. In Erlang the chronological order is, that a pattern is successively matched against all values of the queue, before the next pattern is matched. This is formalized by the function queuematch, which yields additionally the position of the queuevalue which matches:

$$\begin{array}{l}\text{queuematch}((pat_1,\ldots,pat_m),(v_1,\ldots,v_u))\\ \quad = \begin{cases} (i,j,\rho), \text{ if match}(pat_i,v_j) = \rho \text{ and} \\ \qquad\qquad \text{match}(pat_i,v_k) = \text{ Fail } \quad \forall k < j \text{ and} \\ \qquad\qquad \text{match}(pat_l,v_h) = \text{ Fail } \quad \forall l < i, h \le u \\ \text{Fail} \quad \text{, otherwise} \end{cases}\end{array}$$

These definitions of casematch and queuematch are difficult, but this cannot be impeded, because of the Erlang semantics. The variables bound by pattern matching occur free in the Core Erlang terms. To propagate their instantiation, we must substitute them in the whole term. Therefore a substitution yielded by pattern matching cannot only be applied inside the reduction context, we have to apply it to the whole term, which is done in the rules for pattern matching, `case` and `receive`.
The rules for concurrent reduction formalize the creation of a new process and the sending action, which extends the message queue of the process $v_1$. Finally we have to consider runtime errors. They can occur when a value should be send to a non-pid, e.g. a natural number, or a non instantiated variable should be reduced. Formally, error is defined as the empty set of states.

**Remarks**

- The operational semantics $\Longrightarrow$ is not confluent.
- The operational semantics $\Longrightarrow$ is deterministic, considering only one process (including its queue). Every process can be reduced in at most one way, but this can have side effects on the behaviour of another process.

## 4 Abstraction

In the last section we have defined the operational semantics of Core Erlang as a labeled transition system over sets of processes. Usually this labeled transition system has an infinite state space, because of the following facts:

1. arbitrarily many values may appear in the process terms and the queues (e.g. natural numbers)
2. the terms can grow arbitrarily, because of the context-free structure of recursive functional programs
3. the queues 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 analyse Concurrent ML programs. They extract the behaviour of a program, an abstraction, in the type inference and prove properties for the program using this behaviour. 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 ways Amtoft, Nielson and Nielson do:

1. Arbitrarily values are represented by values of a finite domain.
2. We define a subclass of Core Erlang programs, in which programs do not have this property. This subclass covers a great part of practical applications and seems to be sufficient. In future work we will extend the abstraction to reduce the sometimes context-free state space to a regular (finite) one.
3. and 4. We do not solve this problem here. But this is a usual way for handling it. E.g. 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. But tools for the verification of CCS specifications, such as the concurrency workbench [CPS90], only work for the finite state case. We use the same basic approach and obtain a finite state transition system for Core Erlang programs that use only finite parts of the message queues and create only finitely many processes.

The main aspect is reducing the infinite set of values to a finite domain and describe how to calculate on it to simulate the real calculation. In [NN97] this finite domain is the set of types occurring in the program and the behaviour is just a Concurrent ML program calculating on types.
Using a finite domain in the abstraction, we are not able to sequel exactly the operational semantics, because some

| | | |
|---|---|---|
| (P1) | For all $\phi/n \in \Sigma \cup \underline{CF}, v_1, \ldots, v_n \in CT_{Pid}$ and $\widetilde{v}_i \sqsubseteq \alpha(v_i)$ it holds, that<br>$\phi_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n) \sqsubseteq \alpha(\phi_{\mathcal{A}}(v_1, \ldots, v_n))$. | |
| (P2) | For all $pat \in Pat$, $v \in CT_{Pid}$, $\widetilde{pat} \sqsubseteq_{pat} \alpha_{pat}(pat)$ and $\widetilde{v} \sqsubseteq \alpha(v)$ it holds, that<br>$\text{match}_{\widehat{\mathcal{A}}}(\widetilde{pat}, \widetilde{v}) \sqsubseteq \alpha(\text{match}(pat, v))$. | |
| (P3) | For all $pat_1, \ldots, pat_n \in Pat, v \in CT_{Pid}$ it holds, that:<br>if casematch$((pat_1, \ldots, pat_n), v) = (i, \rho)$,<br>then for all $\widetilde{pat}_j \sqsubseteq_{pat} \alpha_{pat}(pat_j)$ and $\widetilde{v} \sqsubseteq \alpha(v)$ there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$<br>with $(i, \widetilde{\rho}) \in \text{casematch}_{\widehat{\mathcal{A}}}((\widetilde{pat}_1, \ldots, \widetilde{pat}_n), \widetilde{v})$. | |
| (P4) | For all $pat_1, \ldots, pat_n \in Pat, v_1, \ldots, v_u \in CT_{Pid}$ it holds, that:<br>if queuematch$((pat_1, \ldots, pat_n), (v_1, \ldots, v_u)) = (i, j, \rho)$,<br>then for all $\widetilde{pat}_j \sqsubseteq_{pat} \alpha_{pat}(pat_j)$ and $\widetilde{v}_k \sqsubseteq \alpha(v_k)$ there exists $\widetilde{\rho} \sqsubseteq \alpha(\rho)$<br>with $(i, j, \widetilde{\rho}) \in \text{queuematch}_{\widehat{\mathcal{A}}}((\widetilde{pat}_1, \ldots, \widetilde{pat}_n), (\widetilde{v}_1, \ldots, \widetilde{v}_u))$. | |
| (P5) | a) For all $p \in Pid$, $\widetilde{p} \sqsubseteq \alpha(p)$ it holds, that $p \in \text{pid}_{\widehat{\mathcal{A}}}(\widetilde{p})$.<br>b) For all $v \notin Pid$, $\widetilde{v} \sqsubseteq \alpha(v)$ it holds, that $\text{err} \in \text{pid}_{\widehat{\mathcal{A}}}(\widetilde{v})$. | |

**Figure 4.1**: Properties of an Abstract Interpretation[2]

branches (e.g. `case` or `receive`) in the operational semantics are not decidable anymore. Hence we replace the deterministic branches by potentially nondeterministic ones. The degree of nondeterminism depends on the concrete abstraction. We replace the concrete operational semantics of `case` and `receive` by abstract functions, which yield a set of the numbers of patterns that possibly match and the corresponding substitutions. Also the constructor functions of the program need a different semantics. We cannot use the free interpretation anymore, because this could be incompatible with our abstract domain. Instead we use an interpretation which guarants that the result of its application on values is one of the possible values of the abstract domain.

We assume an *abstract interpretation* $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$, with $\widehat{A}$ a set of possible computation values, which can be restricted to a finite set in the concrete abstraction and $\widehat{\iota}$ an interpretation function defined on

- the predefined functions $\widehat{\iota}(F/n) : \widehat{A}^n \longrightarrow \widehat{A}$,
- the constructor terms $\widehat{\iota}(\underline{c}/n) : \widehat{A}^n \longrightarrow \widehat{A}$,
- a function for matching $\widehat{\iota}(\text{match}) : \widehat{Pat} \times \widehat{A} \longrightarrow Subst_{\widehat{A}}$,
- a function for matching a list of patterns against a value $\widehat{\iota}(\text{casematch}) : \widehat{Pat}^* \times \widehat{A} \longrightarrow \mathcal{P}(\mathbb{N} \times Subst_{\widehat{A}})$,
- a function for matching a list of patterns against a queue $\widehat{\iota}(\text{queuematch}) : \widehat{Pat}^* \times \widehat{A}^* \longrightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N} \times Subst_{\widehat{A}})$ and
- a function for the possible pids a value of $\widehat{A}$ represents $\widehat{\iota}(\text{pid}) : \widehat{A} \longrightarrow \mathcal{P}(Pid \cup \{\text{err}\})$.

Again we write $f_{\widehat{\mathcal{A}}}$ instead of $\widehat{\iota}(f)$. The abstract interpretation of values as pids is, in the sense of [CC77], a concretization, which we need here because we do not want to abstract from the process model. Communication between processes stays the same as in the *sop*.

We have to use an abstract variation on patterns $\widehat{Pat}$, because patterns are not static in Erlang. They can change at runtime. E.g. the pattern matching in the rule `f(X) -> X={3,4}.` is no matching against a variable, but against the argument, `X` is bound to in the call. Therefore some parts of a pattern are statically defined and others are added at runtime, which are values of the abstract domain:

$$\widehat{Pat} := CT_{Vars \cup \widehat{\mathcal{A}}}$$

These non-static patterns are often used in Erlang, as the patterns `{value,V,P}` and `[{K,V}|_]` in the example $p_0$. With these patterns it is possible to compare queue entries with dynamic values without extracting them from the queue. 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.

Furthermore $\sqsubseteq \subseteq \widehat{A} \times \widehat{A}$ is a partial order[3] describing which values of $\widehat{A}$ are more precise than others and $\alpha : CT_{Pid} \longrightarrow \widehat{A}$ is an abstraction function yielding the most precise representation of a calculation value in the abstract domain.

We extend $\sqsubseteq$ and $\alpha$ on patterns in the following way:

$$\begin{array}{l} X \sqsubseteq_{pat} X \ \forall X \in Var, \\ v \sqsubseteq_{pat} v' \text{ if } v \in \widehat{A} \text{ and } v \sqsubseteq v', \text{ and} \\ c(pat_1, \ldots, pat_n) \sqsubseteq_{pat} c(pat'_1, \ldots, pat'_n) \\ \qquad\qquad \text{iff } pat_i \sqsubseteq_{pat} pat'_i \ \forall 1 \leq i \leq n. \end{array}$$

$$\begin{array}{l} \alpha_{pat}(X) = X, \\ \alpha_{pat}(v) = \alpha(v) \quad \forall v \in \widehat{A} \text{ and} \\ \alpha_{pat}(c(pat_1, \ldots, pat_n)) = c(\alpha_{pat}(pat_1), \ldots, \alpha_{pat}(pat_n)) \end{array}$$

This definition extends $\sqsubseteq$ and $\alpha$ on the static parts of a pattern.

We also need an extension of $\sqsubseteq$ and $\alpha$ on substitutions:

$$\begin{array}{l} \sigma_1 \sqsubseteq \sigma_2 \text{ iff } \text{dom}(\sigma_1) = \text{dom}(\sigma_2) \text{ and} \\ \qquad\qquad \sigma_1(X) \sqsubseteq \sigma_2(X) \ \forall X \in \text{dom}(\sigma_1) \end{array}$$

$$\alpha([X_1/v_1, \ldots, X_n/v_n]) = [X_1/\alpha(v_1), \ldots, X_n/\alpha(v_n)]$$

For readability we use $\alpha$ for the abstraction of substitutions too. The concrete instance of this overloading will be clear from the context.

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

[3]$\sqsubseteq$ does not need to be a cpo, because we will not use fix-point computation.

$$\frac{\phi/n \in \Sigma \cup \underline{CF}}{\Pi, (p, E[\phi(v_1,\ldots,v_n)], q) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (p, E[\phi_{\widehat{\mathcal{A}}}(v_1,\ldots,v_n)], q)} \qquad \frac{-}{\Pi, (p, E[\texttt{self}], q) \longrightarrow_{\widehat{\mathcal{A}}} \Pi', (p, E[\alpha(p)], q)}$$

$$\frac{\text{match}_{\widehat{\mathcal{A}}}(pat, v) = \rho}{\Pi, (p, E[pat\texttt{=}v], q) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (p, \rho(E[v]), q)}$$

$$\frac{(i,\rho) \in \text{casematch}_{\widehat{\mathcal{A}}}((pat_1,\ldots,pat_m), v)}{\Pi, (p, E[\texttt{case}\ v\ \texttt{of}\ pat_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}pat_m\texttt{->}e_m\ \texttt{end}], q) \longrightarrow_{\widehat{\mathcal{A}}} \Pi, (p, \rho(E[e_i]), q)}$$

$$\frac{(i,j,\rho) \in \text{queuematch}_{\widehat{\mathcal{A}}}((pat_1,\ldots,pat_m), (v_1,\ldots,v_u))}{\begin{array}{ll} & \Pi, (p, E[\texttt{receive}\ pat_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}pat_m\texttt{->}e_m\texttt{end}], (v_1,\ldots,v_j,\ldots,v_u)) \\ \xrightarrow{?v_j}_{\widehat{\mathcal{A}}} & \Pi, (p, \rho(E[e_i]), (v_1,\ldots,v_{j-1},v_{j+1},\ldots,v_u)) \end{array}}$$

$$\frac{p' \in \text{pid}_{\widehat{\mathcal{A}}}(v_1)}{\Pi, (p, E[v_1\texttt{!}v_2], q)(p', e, q) \xrightarrow{!v_2}_{\widehat{\mathcal{A}}} \Pi, (p, E[v_2], q)(p', e, q : v_2)} \qquad \frac{\text{err} \in \text{pid}_{\widehat{\mathcal{A}}}(v_1)}{\Pi, (p, E[v_1\texttt{!}v_2], q)(p', e, q) \xrightarrow{!v_2}_{\widehat{\mathcal{A}}} \text{error}}$$

**Figure 4.2**: Abstract Operational Semantics

To guarantee correctness of the abstract interpretation with respect to $\Longrightarrow$ the properties shown in Figure 4.1 have to be fulfilled. Now we can define an abstract operational semantics for Core Erlang. This is the relation:

$$\longrightarrow_{\widehat{\mathcal{A}}} \ \subseteq \widehat{State} \times \widehat{Label} \times \widehat{State}, \text{ where}$$

$$\begin{array}{l} \widehat{State} := \mathcal{P}_{fin}(\widehat{Proc}), \\ \widehat{Proc} := Pid \times T(\widehat{A}) \times \widehat{Que} \\ \widehat{Que} := \widehat{A}^* \\ \widehat{Label} := \{\texttt{!}v \mid v \in \widehat{\mathcal{A}}\} \cup \{\texttt{?}v \mid v \in \widehat{\mathcal{A}}\} \cup \\ \qquad\qquad \{\texttt{spawn(}f\texttt{)} \mid f/n \in FS(p)\} \cup \{\varepsilon\} \end{array}$$

It is defined analogously to $\Longrightarrow$, except that the calculation may be non-deterministic at some points, defined by the abstract interpretation. In Figure 4.2 we only present the differences to $\Longrightarrow$.
We now want to find a relationship between the states of the operational semantics $\Longrightarrow$ and the states of the abstract operational semantics $\longrightarrow_{\widehat{\mathcal{A}}}$. Therefore we extend the partial order $\sqsubseteq \subseteq \widehat{A} \times \widehat{A}$ on Core Erlang terms $T(\widehat{A})$, queues, processes, states and labels. In Definition 4.1 the extension on Core Erlang terms $T(\widehat{A})$ is defined. We only present this extension, the others are handled in the same way. The idea is that a term is smaller than another term iff all occurring values are smaller. The rest has to be the same.

**Definition 4.1 (Ordering on Terms)**
The partial order $\sqsubseteq$[4] $\subseteq T(\widehat{A}) \times T(\widehat{A})$ is the smallest set fulfilling the following properties:

$$\begin{array}{ll} v_1 \sqsubseteq v_2 & \text{, iff } v_1 \sqsubseteq v_2 \\ \phi(e_1,\ldots,e_n) \sqsubseteq \phi(e'_1,\ldots,e'_n) & \text{, iff } e_i \sqsubseteq e'_i \text{ for all } 1 \le i \le n \\ X \sqsubseteq X & \\ pat\texttt{=}e \sqsubseteq pat'\texttt{=}e' & \text{, iff } pat \sqsubseteq_{pat} pat' \text{ and } e \sqsubseteq e' \\ \texttt{self} \sqsubseteq \texttt{self} & \\ e_1\texttt{,}e_2 \sqsubseteq e'_1\texttt{,}e'_2 & \text{, iff } e_1 \sqsubseteq e'_1 \text{ and } e_2 \sqsubseteq e'_2 \\ \texttt{case}\ e\ \texttt{of}\ m\ \texttt{end} \sqsubseteq \texttt{case}\ e'\ \texttt{of}\ m'\ \texttt{end} & \\ & \text{, iff } e \sqsubseteq e' \text{ and } m \sqsubseteq m' \\ \texttt{receive}\ m\ \texttt{end} \sqsubseteq \texttt{receive}\ m'\ \texttt{end} & \text{, iff } m \sqsubseteq m' \\ e_1\texttt{!}e_2 \sqsubseteq e'_1\texttt{!}e'_2 & \text{, iff } e_1 \sqsubseteq e'_1 \text{ and } e_2 \sqsubseteq e'_2 \\ \texttt{spawn(}f\texttt{,}e\texttt{)} \sqsubseteq \texttt{spawn(}f\texttt{,}e'\texttt{)} & \text{, iff } e \sqsubseteq e' \\ pat_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}pat_n\texttt{->}e_n & \text{, iff } pat_i \sqsubseteq_{pat} pat'_i, e_i \sqsubseteq e'_i \\ \quad \sqsubseteq pat'_1\texttt{->}e'_1\texttt{;}\ldots\texttt{;}pat'_n\texttt{->}e'_n & \quad \text{for all } 1 \le i \le n \end{array}$$

We also extend the abstraction function $\alpha : CT_{Pid} \longrightarrow \widehat{A}$ on Core Erlang terms $T(\widehat{A})$, queues, processes, states and labels in the same way. The abstraction of a state is the state in which all values of $CT_{Pid}$ are replaced by their best representation in $\widehat{A}$.

**Lemma 4.2 (Substitution lemma)**
*Let $e \in T(CT_{Pid})$ and $\rho$ be a finite substitution. For all $\rho' \sqsubseteq \alpha(\rho)$, $e' \sqsubseteq \alpha(e)$ it holds, that $\rho'(e') \sqsubseteq \alpha(\rho(e))$.*

**Proof:** The only parts of a term, where a substitution has an effect, are the variables. The rest of the term does not change and the lemma is fulfilled. This can be checked by an easy induction over the term structure. The only cases we have to consider are:

- $X \in dom(\rho)$ :
  $\rho(X) = v \in CT_{Pid}$. Because $\widetilde{\rho} \sqsubseteq \alpha(\rho)$, $\widetilde{\rho}(X) = \widetilde{v} \in \widehat{A}$ with $\widetilde{v} \sqsubseteq \alpha(v)$.

[4] 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.

- $X \notin dom(\rho)$ :
$\rho(X) = X$ and only $X \sqsupseteq \alpha(X)$. But for all $\widetilde{\rho} \sqsupseteq \alpha(\rho)$ also $\widetilde{\rho}(X) = X$. □

**Lemma 4.3 (Correctness of $\longrightarrow_{\widehat{\mathcal{A}}}$)**
*Let $\widehat{\mathcal{A}}$ be an abstract interpretation that fulfills the properties* (P1)–(P5). *If $s \overset{a}{\Longrightarrow} t$ then for all $s' \sqsupseteq \alpha(s)$ exists $t' \sqsupseteq \alpha(t)$ and $b \sqsupseteq \alpha(a)$ such that $s' \xrightarrow{b}_{\widehat{\mathcal{A}}} t'$.*

**Proof:** The correctness of the abstraction can be proved by induction over the structure of the Core Erlang terms of the processes [Huc99]. □

**Theorem 4.4 (Correctness of the Abstraction)**
*Let $\widehat{\mathcal{A}}$ be an abstract interpretation that fulfills the 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 a 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:** With Lemma 4.3 the proof is an easy induction over $n$. □

## 5 Example Abstractions

### 5.1 The Operational Semantics

The "best" abstract interpretation is the one which calculates exactly the operational semantics $\Longrightarrow$. Therefore we can fix the abstract interpretation to $\widehat{\mathcal{A}}_{max} = (CT_{Pid}, \widehat{\iota}, =, \mathrm{id})$ with:

$$\begin{array}{l}
\widehat{\iota}(F/n) = \iota(F/n) \\
\widehat{\iota}(\underline{c}/n)(v_1, \ldots, v_n) = c(v_1, \ldots, v_n) \text{ for all } v_i \in CT_{Pid} \\
\widehat{\iota}(\mathrm{match})(pat, v) = \mathrm{match}(pat, v) \\
\widehat{\iota}(\mathrm{casematch})(\overline{pat}, v) = \{\mathrm{casematch}(\overline{pat}, v)\} \\
\widehat{\iota}(\mathrm{queuematch})(\overline{pat}, \overline{v}) = \{\mathrm{queuematch}(\overline{pat}, \overline{v})\} \\
\widehat{\iota}(\mathrm{pid})(v) = \begin{cases} \{v\}, & \text{if } v \in Pid \\ \{\mathrm{err}\}, & \text{otherwise} \end{cases}
\end{array}$$

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

**Corollary 5.1 (Correctness of $\widehat{\mathcal{A}}_{max}$)**
*The abstract interpretation $\widehat{\mathcal{A}}_{max}$ fulfills the properties* (P1)–(P5).

**Lemma 5.2 (Soundness of $\widehat{\mathcal{A}}_{max}$)**
*For the abstract interpretation $\widehat{\mathcal{A}}_{max}$ and for all $s \in States$ it holds, that:*

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

### 5.2 A Finite Domain Abstraction

Erlang is an untyped language. Hence we cannot use types as abstract values for our calculation as in [NN97]. We have to find another finite set on which we can calculate and decide some branches of the execution. In Erlang programs the use of constructors is very convenient. For our interests the relevant uses of constructors are the patterns. Hence in the abstraction we restrict the possible values for the computation to values that appear as patterns in the program. This is a finite subset of $CT_{\{\perp\} \cup Pid}$. The occurring bottom symbols in these constructor terms represent subterms, from which we have abstracted. E.g. `{succ,`$\perp$`}` can represent `{succ,succ}`, `{succ,{1,2}}`, or `{succ,`$\gamma$`}`, where $\gamma$ is a process identifier. To characterize, what it means that a representation in $CT_{\{\perp\} \cup Pid}$ is more precise than another one we define a partial order $\preceq \subseteq CT_{\{\perp\} \cup Pid} \times CT_{\{\perp\} \cup Pid}$.

**Definition 5.3 (Ordering on $CT_{\{\perp\} \cup Pid}$)**
$\perp \preceq v$ and $v \preceq v$ for all $v \in CT_{\{\perp\} \cup Pid}$
$c(e_1, \ldots, e_n) \preceq c(e'_1, \ldots, e'_n)$ , iff $e_i \preceq e'_i$ for all $1 \leq i \leq n$

To get a finite subset of $CT_{\{\perp\} \cup Pid}$ we could restrict them to constructor terms with only depth $k$ and only a finite set of atoms. The relevant atoms can only be found in the program, which we have to scan. At this we can also extract the relevant patterns and restrict the constructor terms to this extracted finite set. The functions ppats for the extraction of patterns can be found in Appendix A. The result of the scan is a finite subset of $CT_{\{\perp, pid\}}$, where pid is a constant representing pids and $\perp$ stands for abstracted subterms.
In the example program $p_0$ the extraction of patterns yields:

$$\begin{array}{rcl}
\mathrm{ppats}(p_0) & = & \{\gamma, \texttt{allocate}, \texttt{[]}, \texttt{\{allocate,}\gamma\texttt{,}\gamma\texttt{\}}, \texttt{fail}, \\
& & \texttt{free}, \texttt{value}, \texttt{\{value,}\gamma\texttt{,}\gamma\texttt{\}}, \texttt{\{}\gamma\texttt{,}\gamma\texttt{\}}, \\
& & \texttt{[\{}\gamma\texttt{,}\gamma\texttt{\}|}\gamma\texttt{]}, \texttt{succ}, \texttt{[\{succ,}\gamma\texttt{\}|}\gamma\texttt{]}, \\
& & \texttt{allocated}, \texttt{lookup}, \texttt{\{lookup,}\gamma\texttt{,}\gamma\texttt{\}} \\
& & \mid \gamma \in \{\perp, pid\}\}
\end{array}$$

We get the domain of our abstraction of a given program $p$ by substituting pids for the constant $pid$ in the extracted terms and closing this set downwards with respect to $\preceq$ (adding all less precise representations):

$$\widehat{CT}^{p}_{\{\perp\} \cup Pid} := \{[pid/i](t) \mid t \in \mathrm{ppats}(p), i \in Pid\} \downarrow$$

In the example also `{`$\perp$`,`$\perp$`,`$\perp$`}`, `[`$\perp$`|`$\perp$`]`, $\ldots \in \widehat{CT}^{p_0}_{\{\perp\} \cup Pid}$.

**Remarks**

- For every program $p$ is $\mathrm{ppats}(p)$ a finite set.
- For every constructor term in $\mathrm{ppats}(p)$ all its subterms are part of $\mathrm{ppats}(p)$.
- $\widehat{CT}^{p}_{\{\perp\} \cup Pid}$ is an infinite set, but we will later restrict to a finite number of pids and so we only use a finite part of $\widehat{CT}^{p}_{\{\perp\} \cup Pid}$. Neglecting the pids the number of the constructor terms in $\widehat{CT}^{p}_{\{\perp\} \cup Pid}$ is finite for every program $p$.
- $\preceq$ is a complete partial order on $\widehat{CT}^{p}_{\{\perp\} \cup Pid}$, because only finite chains exist.

For simplification we write $\widehat{CT}_{\{\perp\} \cup Pid}$ instead of $\widehat{CT}^{p}_{\{\perp\} \cup Pid}$, if $p$ is clear from the context.
The terms of $CT_{Pid}$ are represented by terms in $\widehat{CT}_{\{\perp\} \cup Pid}$ which do have the same top, but the abstracted subparts are replaced by $\perp$. The abstraction function $\alpha_f : CT_{Pid} \longrightarrow \widehat{CT}_{\{\perp\} \cup Pid}$ yields the best representation of a term $t$, that means the largest top part of $t$ which occurs in $\widehat{CT}_{\{\perp\} \cup Pid}$:

$$\alpha_f(t) = \bigsqcup_{\preceq} \{\widetilde{t} \in \widehat{CT}_{\{\perp\} \cup Pid} \mid \exists p_1 \ldots p_k : \widetilde{t} = t[\perp]_{p_1, \ldots, p_k}{}^{5}\}$$

$$
\begin{array}{l}
(\mathrm{f1}, [\overline{X/v}, Z_1/u_1, \ldots, Z_k/u_k]) \uplus' (\mathrm{f2}, [\overline{Y/w}, Z_1/u'_1, \ldots, Z_k/u'_k]) \\
\qquad = \begin{cases} (\mathrm{f1} \wedge \mathrm{f2}, [\overline{X/v}, \overline{Y/w}, Z_1/u_1, \ldots, Z_k/u_k]) & \text{, if } u_l = u'_l \in CT_{Pid} \ \forall l \leq k \\ (\mathrm{Poss}, [\overline{X/v}, \overline{Y/w}, Z_1/\widehat{u}_1, \ldots, Z_k/\widehat{u}_k]) & \text{, if } \widehat{u}_l = u_l \sqcup_{\preceq} u'_l \text{ exists } \forall l \leq k \\ \mathrm{Fail} & \text{, otherwise} \end{cases} \\
\quad \text{where} \quad \{Z_1, \ldots, Z_k\} = \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) \\
\mathrm{Fail} \uplus' \_ = \mathrm{Fail} \\
\_ \uplus' \mathrm{Fail} = \mathrm{Fail}
\end{array}
$$

**Figure 5.1**: The Union of Substitutions $\uplus'$

Now we know our abstract domain $\widehat{A}$ and have also described its connection to $CT_{Pid}$. We now have to describe the abstract interpretation $\widehat{\iota}_f$. In the abstract interpretation of the predefined functions we have to consider values with $\perp$. We do not know anything about the values, represented by $\perp$. If we calculate on values, we have not abstracted from, we can yield the largest representation in $\widehat{CT}_{\{\perp\}\cup Pid}$ of the standard interpretation:

$$
\widehat{\iota}_f(F/n)(v_1, \ldots, v_n) = \begin{cases} \alpha_f(F_{\mathcal{A}}(v_1, \ldots, v_n)) & \text{, if } v_i \in CT_{Pid} \ \forall 1 \leq i \leq n \\ \perp & \text{, otherwise} \end{cases}
$$

For a constructor application we yield the best representation of the result obtained with the free constructor interpretation. In the case that the free interpretation belongs to $\widehat{CT}_{\{\perp\}\cup Pid}$ we get the free interpretation.

$$
\widehat{\iota}_f(\underline{c}/n)(v_1, \ldots, v_n) = \bigsqcup_{\preceq} \{\widetilde{t} \in \widehat{CT}_{\{\perp\}\cup Pid} \mid \exists p_1, \ldots, p_k : \widetilde{t} = c(v_1, \ldots, v_n)[\perp]_{p_1, \ldots, p_k}\}
$$

Calculating on abstract values, we cannot be sure which pattern matches an incoming message. So the easiest way to abstract the operational behaviour of `receive`/`case` would be a non-deterministic branch to the right hand-side of all possibly matching patterns. But we can be more precise and avoid too much nondeterminism. Therefore we distinguish 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 but not sure that a pattern matches $\perp$): We cannot decide if the real execution matches at this point, because of our abstraction. Hence it is possible that this pattern matches or it does not match and the subsequent patterns match, which means nondeterminism.

This can be realized by extending the substitutions with a flag that indicates if a substitution is irrefutable or only possible.

As already mentioned in Section 3 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 part they occur. Hence the pattern matching `{X,X}={3,4}` would fail and `{X,X}={3,3}` would succeed with the substitution [`X`/3]. But 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`/$\perp$]). It can also happen that two terms a variable is matched with are incomparable, but a more concrete term exists, which can match both. An example therefore is the matching `{X,X}={{3,⊥},{⊥,3}}` which can match if both times $\perp$ is the abstraction of the value 3. So a matching has to yield the substitution (Poss,[`X`/`{3,3}`]). We get the more concrete term `{3,3}` as `{⊥,3}` $\sqcup_{\preceq}$ `{3,⊥}`.

First we define constants for the two kinds of calculated substitutions.

$$
\mathrm{Irref} = \mathrm{True} \quad \text{and} \quad \mathrm{Poss} = \mathrm{False}
$$

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.

$$
\begin{array}{l}
\mathrm{match}'(\mathrm{X}, v) = (\mathrm{Irref}, [\mathrm{X}/v]) \\
\mathrm{match}'(\perp, v) = (\mathrm{Poss}, []) \\
\mathrm{match}'(c(pat_1, \ldots, pat_n), \perp) = \\
\qquad (\mathrm{Poss}, [\mathrm{Vars}(pat_1, \ldots, pat_n)/\perp]) \\
\mathrm{match}'(c(pat_1, \ldots, pat_n), c(v_1, \ldots, v_n)) = \\
\qquad \mathrm{match}'(pat_1, v_1) \uplus' \ldots \uplus' \mathrm{match}'(pat_n, v_n) \uplus' (\mathrm{Irref}, []) \\
\mathrm{match}'(e, e') = \mathrm{Fail}, \text{ otherwise}
\end{array}
$$

The modified union $\uplus'$ of two substitutions (Figure 5.1) with an added flag Irref/Poss has to consider our cpo $\preceq$ on $\widehat{CT}_{\{\perp\}\cup Pid}$. The result depends on that parts of the substitutions that overlap (in the definition this is the part of the substitution on the variables $Z_1, \ldots, Z_k$). 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.

With these auxiliary functions we can define the abstract interpretation for match, casematch and queuematch:

$$
\widehat{\iota}_f(\mathrm{match})(pat, v) = \begin{cases} \rho & \text{, if } \mathrm{match}'(pat, v) = (\_, \rho) \\ \mathrm{Fail} & \text{, otherwise} \end{cases}
$$

$$
\begin{array}{l}
\widehat{\iota}_f(\mathrm{casematch})((pat_1, \ldots, pat_m), v) = \\
\qquad \{(i, \rho) \mid \mathrm{match}'(pat_i, v) = (\_, \rho) \text{ and} \\
\qquad\qquad \mathrm{match}'(pat_j, v) \neq (\mathrm{Irref}, \_) \ \forall j < i\}
\end{array}
$$

[5] $t[\perp]_{p_1,\ldots,p_k}$ means: the subterms at the positions $p_1, \ldots, p_k$ in $t$ are replaced by the term $\perp$.
The supremum always exists, because the set is directed and $\preceq$ is a *cpo*.

$$\begin{aligned}&\widehat{\iota}_f(\text{queuematch})((pat_1,\dots,pat_m),(v_1,\dots,v_u)) = \\ &\quad \{(i,j,\rho) \mid \text{match}'(pat_i,v_j) = (\_,\rho) \text{ and} \\ &\qquad\qquad\qquad \text{match}'(pat_i,v_k) \neq (\text{Irref},\_)\ \forall k<j \text{ and} \\ &\qquad\qquad\qquad \text{match}'(pat_l,v_h) \neq (\text{Irref},\_)\ \forall l<i, 1\leq h\leq u\}\end{aligned}$$

A Pid is interpreted as itself, $\perp$ can represent any value, including all pids and other values do not have an interpretation as a pid.

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

**Lemma 5.5 (Correctness of $\widehat{\mathcal{A}}$)**
*The abstract interpretation $\widehat{\mathcal{A}}_f = (\widehat{CT}_{\{\perp\}\cup Pid}, \widehat{\iota}_f, \preceq, \alpha_f)$ fulfills the properties* (P1)–(P5).

**Proof:** The proof can be found in [Huc99]. □

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, while the behaviour analysis is fixed to type inference and cannot be easily extended to other finite representations of the calculation values.

## 6 Decidability and Optimization

In this section we want to characterize a subset of Core Erlang programs with a finite state abstract operational semantics on $\widehat{\mathcal{A}}_f$. For these programs LTL model checking is decidable and we can proof properties expressed in LTL for a program, if the abstraction fulfills these properties.
An easy example for a Core Erlang program, which has an infinite state space in the abstraction $\widehat{\mathcal{A}}$ is the following:

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

For this program $\Longrightarrow$ and $\longrightarrow_{\widehat{\mathcal{A}}}$ are identical and $(0,\texttt{f}^n\texttt{(0)},()) \longrightarrow_{\widehat{\mathcal{A}}} (0,\texttt{f}^{n+1}\texttt{(0)},())$. This infinite state space is a result of the context free structure of function calls.
The first class of Core Erlang programs, with a finite state property, we can think of, are programs with only tail recursive calls. Here the size of the terms in the processes is limited by the right hand-sides of the program. Under the assumptions that only finitely many processes are spawned and the processes only use restricted parts of their message queues the abstract operational semantics is finite state[6]. These assumptions have to be made, because they are undecidable in general and otherwise good abstractions to finite systems are hard to find.
In the implementation this can be handled by restricting them to a fixed size and warning when exceeding. This usually 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. when a sending process is much faster than the receiving process and the queue may overflow. This can especially happen when distributing processes on different architectures. A synchronization message should then be send and the queue size does not exceed anymore.
From the practical point of view the class of tail recursive programs is very restricted, because e.g no sub-calculations, such as the call of `lookup` in $p_0$, can be made. But programs with sub-calculations often also yield finite state systems (e.g. $p_0$) and are needed to achieve comfort in programming. An elegant description of these programs can be given by the notion of hierarchical programs.

**Definition 6.1 (Hierarchical Core Erlang Programs)**
A Core Erlang program is called hierarchically recursive if for every rule $f(X_1$`,...,`$X_n)$`->`$e_f$`.` of the program $f \notin \text{innercalls}(e_f)$ holds.

innercalls$(e)$ is the subset of $FS(p)$ which can be reached during the reduction of $e$ except the tail recursive call. The complete definition of innercalls can be found in Appendix B.

**Lemma 6.2 (Decidability)**
*Let $p$ be a hierarchical Core Erlang program, that spawns only a finite number of processes during its abstract operational semantics and that uses only finite parts of the message queues. The abstract operational semantics of $p$ defines a finite transition system with respect to $\alpha$-conversion.*

**Proof:** The idea of the proof [Huc99] is that in every inner function call the number of possibly used functions at the inner positions is reduced and therefore the size of the Core Erlang terms is limited by the sum of all sizes of the right hand-sides. □

From the theoretical point of view it is nice to know, that model checking is decidable, but the transition system created for the abstract operational semantics is very large and cannot be represented in a computer for real applications. The main reason therefore is the explosion of state space in the interleaving semantics.
The sequential parts of the execution are less relevant for the verification of the concurrent system and usually more than the half of all transitions are labeled with $\varepsilon$. So it would be good to eliminate these sequential transitions like an $\varepsilon$-elimination known from finite automata. 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 stay to be complete with respect to $\Longrightarrow$.
The implementation showed that $\varepsilon$-elimination reduces the state-space, but most states with an $\varepsilon$-transition also have a labeled transition, that has to be considered and cannot be eliminated.
For practical use partial order reduction [Pel94] is needed. We have not implemented this yet, because we are still working on a prototype. But thinking about it, we made the following observation:

**Lemma 6.3**
*For every state $s$ with $s \longrightarrow_{\widehat{\mathcal{A}}_f} t$ it holds, that if $s \xrightarrow{a}_{\widehat{\mathcal{A}}_f} s'$ also $t \xrightarrow{a}_{\widehat{\mathcal{A}}_f} t'$ and $s' \longrightarrow_{\widehat{\mathcal{A}}_f} t'$.*

**Proof:** An easy case analysis on $\longrightarrow_{\widehat{\mathcal{A}}_f}$. □

If our logic cannot distinguish the paths $s \longrightarrow_{\widehat{\mathcal{A}}_f} t \xrightarrow{a}_{\widehat{\mathcal{A}}_f} t'$ and $s \xrightarrow{a}_{\widehat{\mathcal{A}}_f} s' \longrightarrow_{\widehat{\mathcal{A}}_f} t'$, we do not need to save both paths.

[6] In the operational semantics we use new instantiations of the right hand-sides to avoid conflicts with the free variables. Therefore states are identical if all Core Erlang terms do only distinguish in the free variables ($\alpha$-conversion). For completely tail recursive programs we can omit the renaming in the rule for function application, because no other variables can exist at this point.

For an implementation of this state space reduction we execute sequential calculations as long as possible and only consider the states, where no sequential steps are possible anymore. Again we have to consider loops, which represent possible, infinite calculations.

## 7 Implementation

We have implemented the abstraction as a prototype. The logic we use is an extension of LTL with labeled next state operator and without properties. With this logic and model checking we are able to prove some interesting properties for small Core Erlang programs, like mutual exclusion, the absence of deadlocks or properties like the program can always perform a special sending action.

For a demonstration how properties can be proved we have added two client processes accessing the database of Example 1 (Appendix C). We want to proof that mutual exclusion is provided by the database. This can be expressed by the property:

If the database receives the message $\texttt{\{allocate,\_,}p\texttt{\}}$ from a process $p$ then it may not receive the message $\texttt{\{value,\_,}p'\texttt{\}}$ from a process $p' \neq p$ until it receives the message $\texttt{\{value,\_,}p\texttt{\}}$). In extended LTL this can be expressed by the following formula

$$\varphi = \bigwedge_{\substack{p \in Pid \\ p' \neq p}} G\ (\texttt{?\{allocate,\_,}p\texttt{\}} \rightarrow (\neg\texttt{?\{value,\_,}p'\texttt{\}}\ U\ \texttt{?\{value,\_,}p\texttt{\}}),$$

where $G$ means generally, $U$ until and $\texttt{?\{allocate,\_,}p\texttt{\}}$, that this labeled transition is performed on the path. During the execution only three processes are created and we can restrict *Pid* to three values and get a finite formula. The abstract operational semantics optimized with the maximal sequential calculation builds a transition system with 549 states[7]. Verifying the formula shows, that $\varphi$ is not fulfilled. The refuting path has a sequential cycle over $\texttt{lookup(}\bot\texttt{,}\bot\texttt{)}$. The reason therefore is that we cannot decide termination in the abstraction. Such paths occur in almost every abstraction and we have decided to eliminate these sequential non terminating paths in the tool. So a property can only be proved under the assumption that every called sequential function terminates. The same assumption has to be made for the behaviour analysis presented in [JN94] and it seems to be difficult to find an abstraction where this is not required. For the `lookup`-function this assumption holds and we can prove that for every path $\varphi$ is fulfilled and also the *sop* (an infinite transition system) fulfills $\varphi$.

Alternatively to the implementation of a prototype would have been a translation into Promela, the specification language of SPIN [Hol97]. This approach was made in the Java PathFinder [HP99] and the Bandera tool [DP98, HDL98]. Both present translations from Java (Ada) to Promela. The abstraction is a pre-process on the source code.

We have decided not to use this approach, because a translation is much more complicated than in imperative languages like Java. Erlang is a functional language and we have to simulate function calls in Promela. Also the communication model is very different in Erlang and the semantics of `receive`, which does not only access the first element of the message queue, would be hard to implement in Promela, which only allows buffered channels of a fixed size and access to only the first element of a channel. Furthermore, our abstraction does not yield a program transformation on source code and has to be combined with the translation into Promela.

Another problem is the generation of counter examples. If a formula is not fulfilled, LTL model checking yields a (possibly infinite) path refuting the formula. There are two possibilities this can be estimated. The path can also appear in $\Longrightarrow$ and is a counter example for the program not fulfilling the property. Otherwise the path is an additional path. The abstract interpretation was to raw. Translating Core Erlang into Promela it will be hard to retranslate a refuting path in Promela to the corresponding path in Erlang.

An advantage of the translation into Promela would be that in SPIN partial order reduction is implemented. But at closer inspection partial order reduction is only efficient if the independent actions are known. But the independence of two actions can better be estimated on the level of Erlang actions, e.g. two sending actions to different processes are independent. In the translated program these independences can be hidden. So optimizing LTL model checking with partial order reduction on $\longrightarrow_{\widehat{\mathcal{A}}}$ will be more efficient.

## 8 Conclusions and Future Work

We have presented a framework for abstract interpretations of Core Erlang programs that guarantees correctness with respect to the *sop*. Furthermore we have defined a example abstract interpretation, that can be used for the automatic proof of properties with model checking. The abstract operational semantics for hierarchical programs yields finite state transition systems and model checking is decidable. Many Erlang programs used in practice are hierarchical, but already the definition of `append` is not hierarchical. Hence we are looking for an abstraction that yields, instead of the context-free transition system, a regular one. In this context we also have to think about higher-order functions. It is no problem to integrate them in $\Longrightarrow$ and $\longrightarrow_{\widehat{\mathcal{A}}}$. As an example we could have defined the `lookup`-function in Example 2.1 with `foldr`. This does not change $\longrightarrow_{\widehat{\mathcal{A}}_f}$ and the property $\phi$ can also be proven. The restriction to first-order programs can be omitted, but the definition of a class like the hierarchical programs gets more difficult.

Another approach for the formal verification of Erlang programs is chosen by Dam and Fredlund in [DaF98]. They use theorem proving combined with local model checking on the *sop*, without any abstraction. It would be interesting to combine both techniques. Properties that can be proved with the abstraction are automatically proved and for the others theorem proving can be used. This could also be a technique for proving properties of programs with an infinite state space in the abstract interpretation.

We want to implement a tool that can be used in practice. Usually in LTL formulas propositions of states are used. This makes a comfortable description of system properties possible. Therefore we must find a formalism to add these properties to the programs and use them comfortably in the formulas (e.g. a conjunction over all pids). For the practical use it is also necessary to reduce the state space, using partial order reduction [Pel94], as done in SPIN [Hol97]. Furthermore we want to transfer our approach to other programming languages for concurrent or distributed systems, like Concurrent ML, Concurrent Haskell or Java.

[7] The abstract semantics without the optimization has 5063 states

**Acknowledgements**

I thank the anonymous reviewers, Olaf Chitil and Michael Hanus for their valuable comments and discussions.

**References**

[ANN98] T. Amtoft, H. Riis Nielson, and F. Nielson. Behaviour analysis and safety conditions: A case study in CML. *Lecture Notes in Computer Science*, 1382:255–269, 1998.

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[CC77] P. Cousot and R. 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.

[CPS90] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 24–37, Berlin, June 1990. Springer.

[DaF98] M. Dam and L. å. 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.

[FFKD87] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[HDL98] J. Hatcliff, M. Dwyer, and S. Laubach. Staging static analyses using abstraction-based program specialization. *Lecture Notes in Computer Science*, 1490:134–148, 1998.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[HP99] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer – STTT*, to appear, 1999.

[Huc99] Frank Huch. Verification of erlang programs using abstract interpretation and model checking – extended version. Technical Report 99–02, RWTH Aachen, 1999.

[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.

[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.

[NN97] Hanne Riis Nielson and Flemming Nielson. Communication analysis for concurrent ML. In *ML with Concurrency*, Monographs in Computer Science, pages 185–235. Springer-Verlag, 1997.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Lecture Notes in Computer Science*, 818:377–390, 1994.

[SS98] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. *Lecture Notes in Computer Science*, 1503:351–380, 1998.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science*, 1043:238ff., 1996.

# Appendix

## A Extraction of Patterns

In a first step the function ppats (Figure A.1) yields a subset of $CT_{\{\perp,pid\}}$. The leaves in these constructor terms are a constant *pid*, representing possible pids at a position, or the value $\perp$, which represents values from which we have abstracted. We distinguish values which built possible patterns by the calculation over the inductive structure of the program and patterns that are already detected, which means that they need not be extended to larger patterns any more. This is implemented by a tuple of recursive and global results of the function tpats for pattern extraction over core Erlang terms.
We also want to guarantee, that for every calculated pattern its subterms are also calculated. Therefore the calculated recursive patterns in the rule for constructor application are enlarged with the constructor in the first component and also copied to the second component of the result.

## B Inner Function Calls

The function calls : $T(\emptyset) \to \mathcal{P}(FS(p))$ yields for every Core Erlang term $e \in T(\emptyset)$ the subset of $FS(p)$ that can be called during the reduction of $f$. For more details see [Huc99].

**Definition 2.1 (Inner Function Calls)**
Let $p$ be a core Erlang program, $FS(p)$ the set of function symbols of the program and $e \in T(\emptyset)$. Define innercalls($e$)

$$
\begin{array}{lcl}
\mathrm{ppats}(p_1\ p_2) & = & \mathrm{ppats}(p_1) \cup \mathrm{ppats}(p_2) \\
\mathrm{ppats}(f(X_1,\ldots,X_n)\texttt{->}e) & = & \mathit{rec} \cup \mathit{glo} \text{ where } (\mathit{rec},\mathit{glo}) = \mathrm{tpats}(e) \\
\mathrm{tpats}(c(e_1,\ldots,e_n)) & = & (\{c(p_1,\ldots,p_n) \mid p_i \in \mathit{rec}_i\},\ \bigcup\limits_{1\le i\le n} \mathit{rec}_i \cup \mathit{glo}_i) \text{ where } (\mathit{rec}_i,\mathit{glo}_i) = \mathrm{tpats}(e_i) \\
\mathrm{tpats}(\phi(e_1,\ldots,e_n)) & = & (\{\bot,\mathit{pid}\},\ \bigcup\limits_{1\le i\le n} \mathit{rec}_i \cup \mathit{glo}_i),\ \ \phi/n \in FS(p) \cup \Sigma \\
 & & \text{where } (\mathit{rec}_i,\mathit{glo}_i) = \mathrm{tpats}(e_i) \\
\mathrm{tpats}(X) & = & (\{\bot,\mathit{pid}\},\emptyset) \\
\mathrm{tpats}(\mathit{pat}\texttt{=}e) & = & (\mathit{rec}_e, \mathit{rec}_{pat} \cup \mathit{glo}_{pat} \cup \mathit{glo}_e) \text{ where } (\mathit{rec}_{pat},\mathit{glo}_{pat}) = \mathrm{tpats}(\mathit{pat}) \\
 & & \qquad (\mathit{rec}_e,\mathit{glo}_e) = \mathrm{tpats}(e) \\
\mathrm{tpats}(\texttt{self}) & = & (\{\mathit{pid}\},\emptyset) \\
\mathrm{tpats}(\texttt{case } e \texttt{ of } m \texttt{ end}) & = & (\mathit{rec}_m, \mathit{rec}_e \cup \mathit{glo}_e \cup \mathit{glo}_m) \text{ where } (\mathit{rec}_e,\mathit{glo}_e) = \mathrm{tpats}(e) \\
 & & \qquad (\mathit{rec}_m,\mathit{glo}_m) = \mathrm{tpats}(m) \\
\mathrm{tpats}(\texttt{spawn}(f,e)) & = & (\{\mathit{pid}\}, \mathit{rec}_e \cup \mathit{glo}_e) \text{ where } (\mathit{rec}_e,\mathit{glo}_e) = \mathrm{tpats}(e) \\
\mathrm{tpats}(e_1\texttt{!}e_2) & = & (\mathit{rec}_2, \mathit{rec}_1 \cup \mathit{glo}_1 \cup \mathit{glo}_2) \text{ where } (\mathit{rec}_1,\mathit{glo}_1) = \mathrm{tpats}(e_1) \\
 & & \qquad (\mathit{rec}_2,\mathit{glo}_2) = \mathrm{tpats}(e_2) \\
\mathrm{tpats}(\texttt{receive } m \texttt{ end}) & = & \mathrm{tpats}(m) \\
\mathrm{tpats}(e_1\texttt{,}e_2) & = & (\mathit{rec}_2, \mathit{rec}_1 \cup \mathit{glo}_1 \cup \mathit{glo}_2) \text{ where } (\mathit{rec}_1,\mathit{glo}_1) = \mathrm{tpats}(e_1) \\
 & & \qquad (\mathit{rec}_2,\mathit{glo}_2) = \mathrm{tpats}(e_2) \\
\mathrm{tpats}(\mathit{pat}_1 \texttt{ -> } e_1\texttt{;}\ldots\texttt{;}\mathit{pat}_n \texttt{ -> } e_n) & = & (\bigcup\limits_{1\le i\le n} \mathit{rec}_{e_i},\ \bigcup\limits_{1\le i\le n} \mathit{rec}_{pat_i} \cup \mathit{glo}_{pat_i} \cup \mathit{glo}_{e_i}) \\
 & & \text{where } (\mathit{rec}_{pat_i},\mathit{glo}_{pat_i}) = \mathrm{tpats}(\mathit{pat}_i) \\
 & & \qquad (\mathit{rec}_{e_i},\mathit{glo}_{e_i}) = \mathrm{tpats}(e_i)
\end{array}
$$

**Figure A.1**: Extraction of Patterns

as the smallest set over $FS(p)$ that fulfills the following equations:

$$
\begin{array}{lcl}
\mathrm{innercalls}(\phi(e_1\texttt{,}\ldots\texttt{,}e_n)) & = & \bigcup\limits_{i=1}^{n} \mathrm{calls}(e_i) \\
\mathrm{innercalls}(X) & = & \emptyset \\
\mathrm{innercalls}(X\texttt{=}e) & = & \mathrm{innercalls}(e) \\
\mathrm{innercalls}(\texttt{self}) & = & \emptyset \\
\mathrm{innercalls}(e_1\texttt{,}e_2) & = & \mathrm{calls}(e_1) \cup \mathrm{innercalls}(e_2) \\
\mathrm{innercalls}(\texttt{case } e \texttt{ of } m \texttt{ end}) & = & \mathrm{calls}(e) \cup \mathrm{innercalls}(m) \\
\mathrm{innercalls}(e_1\texttt{!}e_2) & = & \mathrm{calls}(e_1) \cup \mathrm{calls}(e_2) \\
\mathrm{innercalls}(\texttt{receive } m \texttt{ end}) & = & \mathrm{innercalls}(m) \\
\mathrm{innercalls}(\texttt{spawn}(f,e)) & = & \mathrm{calls}(e) \\
\mathrm{innercalls}(\mathit{pat}_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}\mathit{pat}_n\texttt{->}e_n) & = & \bigcup\limits_{i=1}^{n} \mathrm{innercalls}(e_i)
\end{array}
$$

Let $f \in FS(p)$ and $f(X_1,\ldots,X_n)\texttt{->}e_f\texttt{.}$ is a rule of $p$, then $\mathrm{innercalls}(f) = \mathrm{innercalls}(e_f)$.

## C Client Processes for Database

We define a client process as an user interface for the access to the data base. The functions `read` and `write` are predefined functions which perform I/O as side effects. Therefore especially `read` is no function. In our abstract interpretation both are interpreted as constant functions yielding the value $\bot$. Especially for `read` this is a good representation of the behaviour, because we want to consider all possible results. The function `main` is modified to start two processes behaving as clients.

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

```
client(DB) ->
  case read('(l)ookup/(i)nsert >') of
    i -> K = read('Key >'),
         DB!{allocate,K,self()},
         receive
           free -> V = read('Value >'),
                   DB!{value,V,self()},
                   client(DB);
           allocated -> write('Key allocated'),
                        client(DB)
         end;
    l -> K = read('Key >'),
         DB!{lookup,K,self()},
         receive
           fail -> write('Key not allocated');
                   {succ,V} -> write(V)
         end,
         client(DB);
    _ -> client(DB)
  end.
```

In an Erlang program it would not be suggestive to create two I/O performing processes in concurrency, but they could be created on different Erlang nodes (computers) to realize distributed access. The behaviour does not change distributing the processes.