Toward Parametric Verification of Open Distributed Systems*

Mads Dam, Lars-åke Fredlund, and Dilian Gurov

Swedish Institute of Computer Science**

Abstract. A logic and proof system is introduced for specifying and proving properties of open distributed systems. Key problems that are addressed include the verification of process networks with a changing interconnection structure, and where new processes can be continuously spawned. To demonstrate the results in a realistic setting we consider a core fragment of the Erlang programming language. Roughly this amounts to a first-order actor language with data types, buffered asynchronous communication, and dynamic process spawning. Our aim is to verify quite general properties of programs in this fragment. The specification logic extends the first-order μ-calculus with Erlang-specific primitives. For verification we use an approach which combines local model checking with facilities for compositional verification. We give a specification and verification example based on a billing agent which controls and charges for user access to a given resource.

1 Introduction

A central feature of open distributed systems as opposed to concurrent systems in general is their reliance on modularity. Open distributed systems must accommodate addition of new components, modification of interconnection structure, and replacement of existing components without affecting overall system behaviour adversely. To this effect it is important that component interfaces are clearly defined, and that systems can be dynamically put together relying only on component behaviour along these interfaces. That is, behaviour specification of open distributed systems, and hence also their verification, cannot be based on a fixed systems structure but needs to be parametric on the behaviour of components. Almost all prevailing approaches to verification of such systems rely on an assumption that process networks are static, or can safely be approximated as such, as this assumption opens up for the possibility of bounding the space of global system states. Clearly such assumptions square poorly with the dynamic and parametric nature of open distributed systems.

* Work partially supported by the Computer Science Laboratory of Ericsson Telecom AB, Stockholm, the Swedish National Board for Technical and Industrial Development (NUTEK) through the ASTEC competence centre, and a Swedish Foundation for Strategic Research Junior Individual Grant.

** Address: SICS, Box 1263, S-164 28 Kista, Sweden. Email: {mfd,fred,dilian}@sics.se.

Core Erlang Our aim in this paper is to demonstrate an approach to system specification and verification that has the potentiality of addressing open distributed systems in general. We study the issue in terms of a core fragment of Ericsson's Erlang programming language [AVWW96] which we call Core Erlang. Core Erlang is essentially a first-order actor language (cf. [AMST97]). The language has primitives for local computation: data types, first-order abstraction and pattern matching, and sequential composition. In addition to this Core Erlang has a collection of primitives for component (process) coordination: sending and receiving values between named components by means of ordered message queues, and for dynamically creating new components.

Specification Language We use a temporal logic based on a first-order extension of the modal μ-calculus for the specification of component behaviour. In this logic it is possible to describe a wide range of important system properties, ranging from type-like assertions to complex interdependent safety and liveness properties. The development of this logic is actually fairly uncontroversial: To adequately describe component behaviour it is certainly needed to express potentialities of actions across interfaces and the necessary and contingent effects of these actions, to express properties of data types and properties of components depending on values, to access component names, and to express properties of messages in transit.

Challenges The real challenge is to develop techniques that allow such temporal properties to be verified in a parametric fashion in face of the following basic difficulties:

1. Components can dynamically create other components.
2. Component names can be bound dynamically, thus dynamically changing component interconnection structure (similar to the case of the π-calculus [MPW92]).
3. Components are connected through unbounded message queues.
4. Through use of non-tail recursion components can give rise to local state spaces of unbounded size.
5. Basic data types such as natural numbers and lists are also unbounded.

We would expect some sort of uniformity in the answers to these difficulties. For instance, techniques for handling dynamic process creation are likely to be adaptable to non-tail recursive constructions quite generally, and similarly message queues is just another unbounded data type.

Approach In [Dam98] an answer to the question of dynamic process creation was suggested, cast in terms of CCS. Instead of closed correctness assertions of the shape $s : \phi$ (s is a system, ϕ its specification) which are the typical objects of state exploration based techniques, the paper considered more general *open correctness assertions* of the shape $\Gamma \vdash s : \phi$ where Γ expresses assumptions $S : \psi$ on components S of s. Thus the behaviour of s is specified parametrically upon the behaviour of its component S. To address verification, a sound and weakly complete proof system was presented, consisting of proof rules to reduce complex proof goals to (hopefully) simpler ones, including a process cut rule by which properties of composite processes can be proved in terms of properties of its constituent parts. The key, however, is a loop detection mechanism, namely a *rule of discharge*, that can under certain circumstances be applied to discharge proof goals that are instances of proof goals that have already been encountered during proof construction. The combination of the rule of discharge with the process cut rule provides the power required to deal successfully with features such as dynamic process creation.

Our contribution in the present paper is to show how the approach of [Dam98] can be extended to address the difficulties enumerated above for a fragment of a real programming language, and to show the utility of our approach on a concrete example exhibiting some of those difficulties. In particular we have resolved a number of shortcomings of [Dam98] with respect to the rule of discharge.

Example We use a running example based on the following scenario: A user wants to access a resource paying for this using a given account. She therefore issues a request to a resource manager which responds by dynamically creating a billing agent process to act as an intermediary between the user, the resource, and the user's account. We view this scenario as quite typical of many security-critical mobile agent applications.

The user is clearly taking a risk by exposing her account to the resource manager and the billing agent. One of these parties might violate the trust put in him eg. by charging for services not provided, or by passing information to third parties that should be kept confidential. Equally the resource manager need to trust the billing agent (and to some minor extent the user). We show how the system can be represented in Core Erlang, how some critical properties can be expressed, and outline a proof of the desirable property of the billing agent that the number of transfers from the user account does not exceed the number of requests to use the resource.

Organisation The paper is organised as follows. Section 2 introduces the fragment of Erlang treated in the paper and presents an operational semantics for the language. The following section focuses on the variant of the μ-calculus used as the specification logic, providing examples as well as a formal semantics. Section 4 describes the local part of a proof system for verifying that an Erlang system satisfies a specification formalised in the μ-calculus, and contains proofs of soundness for some proof rules introduced in the section. The rule of discharge is motivated in terms of two simple examples in section 5, and it is formally stated and proved sound in section 6. In section 7 we put the proof system to work on the billing example, outlining parts of a correctness proof. Finally, the paper ends with a discussion in section 8 on directions for further work, and some concluding remarks.

2 Core Erlang

We introduce a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as natural numbers, lists, tuples, or process identifiers (pid's), using asynchronous, first-order call-by-value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as communication guards, exception handling, modules, distribution extensions, and a host of built-in functions.

Processes A Core Erlang system consists of a number of processes computing in parallel. Each process is named by a unique process identifier *pid* of which we assume an infinite supply. Pid's are created along with the processes themselves. Associated with a process is an Erlang *expression* e, i.e. the expression being evaluated, and a mailbox, or input *queue* q, assumed to be of unbounded capacity. Messages are sent by addressing a data value to a receiving process, identified through its pid.

Definition 1 (Processes, System States). *An Erlang* process *is a triple* $\langle e, pid, q\rangle$*, where* e *is an Erlang expression,* pid *is a process identifier, and* q *is a message queue. An Erlang* system state s *is a set of processes such that* $\langle e, pid, q\rangle, \langle e', pid', q'\rangle \in s$ *and* $\langle e, pid, q\rangle \neq \langle e', pid', q'\rangle$ *implies* $pid \neq pid'$*.* $\mathcal{S}$ *is the set of system states.*

We normally write system states using the grammar:

$$s ::= \langle e, pid, q\rangle \mid s \parallel s$$

understanding $\parallel$ as set union.

As the amount of different syntactical categories involved in the operational semantics and in the specification logic is quite large, the following notational convention is useful.

Convention 2. Corresponding small and capital letters are used to range over values, resp. variables over a given syntactical domain.

Thus, as e.g. e is used to range over Erlang expressions, E is used to range over variables taking Erlang expressions as values.

Erlang Expressions Besides expressions we operate with the syntactical categories of *matches* m, *patterns* p, and *values* v. The abstract syntax of Core Erlang expressions is summarised as follows:

$$\begin{aligned} e ::= {} & V \mid \texttt{self} \mid op(e_1, \ldots, e_n) \mid \\ & e_1\, e_2 \mid e_1, e_2 \mid \texttt{case}\ e\ \texttt{of}\ m \mid \texttt{spawn}(e_1, e_2) \mid \\ & \texttt{receive}\ m\ \texttt{end} \mid e_1 ! e_2 \\ m ::= {} & p_1 \rightarrow e_1; \cdots ; p_n \rightarrow e_n \\ p ::= {} & op(p_1, ..., p_n) \mid V \\ v ::= {} & op(v_1, ..., v_n) \end{aligned}$$

Here op ranges over a set of primitive constants and operations including zero 0, successor $e+1$, tupling $\{e_1, e_2\}$, the empty list $[\,]$, list prefix $[e_1|e_2]$, pid constants ranged over by pid, and atom constants ranged over by a, f, and g. In addition we need constants and operations for message queues: A queue is a sequence of values $q = v_1 \cdot v_2 \cdot \ldots \cdot v_n$ where ϵ is the empty queue and $q_1 \cdot q_2$ is queue concatenation, assumed to be associative.

Atoms are used to name functions. We reserve f and g for this purpose. Expressions are interpreted relative to an environment of function definitions $f(V_1, \ldots, V_n) \to e$, syntactic sugar for $f \stackrel{\Delta}{=} \{V_1, \ldots, V_n\} \to e$. Each function atom f is assumed to be defined at most once in this environment.

Intuitive Semantics The intuitive meaning of the Erlang operators, given in the context of a pid pid and a queue q, should be not too surprising:

- **self** evaluates to the pid pid of the process.
- op is a data type constructor: To evaluate $op(e_1, ..., e_n)$, e_1 to e_n are evaluated in left-to-right order.
- $e_1\ e_2$ is application: First e_1 is evaluated to a function atom[1] f, then e_2 is evaluated to a value v, and finally the function definition of f is looked up and matched to v.
- e_1, e_2 is sequential composition: First e_1 is evaluated (for its side-effect only), and then evaluation proceeds with e_2 whose value is actually returned.
- **case** e **of** m is evaluated by first evaluating e to a value v, then matching v using m. If several patterns in m match, the first one is chosen. Matching a pattern p_i of m against v can cause unbound variables to become bound in e_i[2]. In a function definition all free variables are considered as unbound.
- **spawn**(e_1, e_2) is the language construct for creating new processes. First e_1 is evaluated to a function atom f, then e_2 to a value v, a new pid pid' is generated, and a process $\langle (f\ v), pid', \epsilon \rangle$ with that pid and an initially empty queue is spawned evaluating $f\ v$. The value of the spawn expression itself is the pid pid' of the newly spawned process.
- **receive** m **end** inspects the process mailbox q and retrieves (and removes) the first element in q that matches any pattern of m. Once such an element v has been found, evaluation proceeds analogously to **case** v **of** m.
- $e_1!e_2$ is sending: e_1 is evaluated to a pid pid', then e_2 to a value v, then v is sent to pid', resulting in v as the value of the send expression.

Example: Billing Agents In the introduction we gave a scenario for accessing private resources. As an example Core Erlang program, a function for managing such accesses (a resource manager) is shown below. Erlang variables are upper-case, while atoms are lower-case. Atoms are used to name functions (*rm*, *lookup* and *billagent*), but also as constant values for identifying the "type" of a particular message (*contract*, *contract_ok*, etc.), or for other synchronisation purposes (*lookup_ok* and *lookup_nok*).

---

[1] There is no construct for lambda-abstraction in Erlang.

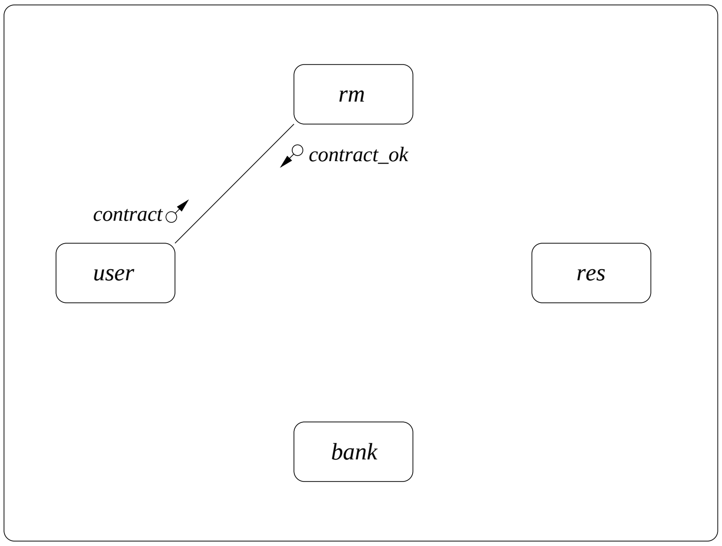
[2] This is not quite the binding the convention of Erlang proper: There the first occurrence of V in (case e_1 of $V \to e_2$), V can bind the second.

```
rm(ResList, BankPid, RAcc) ->
  receive
    {contract, {Pu, UAcc}, UserPid} ->
      case lookup(Pu, ResList)) of
        {lookup_ok, Pr} ->
          UserPid!{contract_ok,spawn(billagent, (Pr, BankPid, RAcc, UAcc))};
        lookup_nok ->
          UserPid!contract_nok
      end
  end,
  rm(ResList, BankPid, RAcc).
```

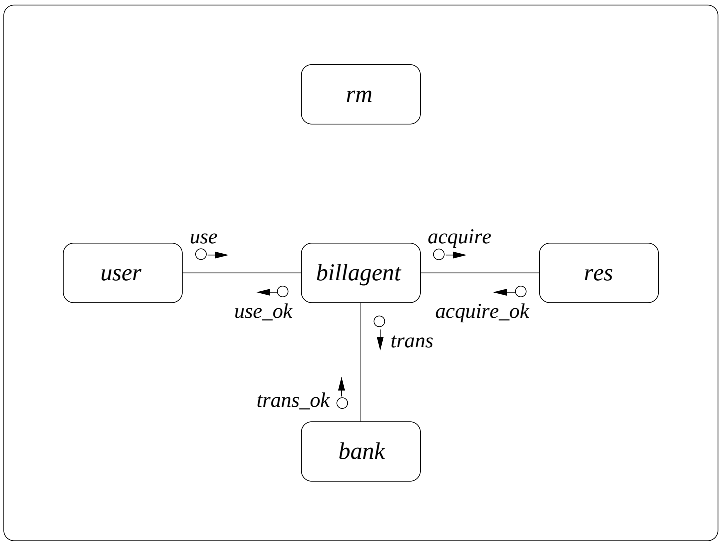
The resource manager *rm* accepts as arguments a resource list, the pid of a trusted bank agent, and a private account. The resource list uses pairs to implement a map from public to private resource "names"; given a public name Pu, a function $lookup(Pu, ResList)$ is used to extract the corresponding private name Pr. The resource manager, after receiving a contract offer (identifying the paying account $UAcc$), tries to obtain the private name of the requested resource, and if this succeeds, a billing agent is spawned to mediate between the user, the bank and the resource, and the name (i.e. pid) of the billing agent is made known to the user. Figure 1 shows the system configuration before and after the creation of the billing agent.

```
billagent(ResPid, BankPid, RAcc, UAcc) ->
  receive
    {use, UserPid} ->
      Res!{acquire, self},
      receive
        {acquire_ok, Value} ->
          BankPid!{trans, {UAcc, RAcc}, self},
          receive
            {trans_ok, {UAcc, RAcc}} -> UserPid!{use_ok, Value};
            {trans_nok, {UAcc, RAcc}} -> UserPid!use_nok
          end;
        acquire_nok -> UserPid!use_nok
      end
  end,
  billagent(ResPid, BankPid, RAcc, UAcc).
```

The billing agent coordinates access to the resource with withdrawals from the account. Upon receiving a request for the resource $\{use, UserPid\}$, it attempts to acquire the resource, and if this succeeds (resulting in a response $Value$ being received from the resource), it attempts to transfer money from the user account to the resource manager account, and then sends the value to the user.



(a)



(b)

Fig. 1. The system configuration: (a) before, and (b) after spawning the billing agent

Matching The definition of the operational semantics requires an ancillary, entirely standard, definition to be made concerning pattern matching between a value v and a pattern p. A *term environment* is a partial function τ of variables V to values v. The totally undefined term environment is $[\,]$. The *most general unifier*, $mgu(v, p, \tau)$, of v and p in term environment τ is a term environment defined as follows:

$$
\begin{aligned}
mgu(v, V, \tau) &\triangleq \tau[V \mapsto v] \text{ if } V \notin dom(\tau) \\
mgu(v, V, \tau) &\triangleq \tau \qquad\qquad \text{if } V \in dom(\tau) \text{ and } \tau(V) = v \\
mgu(op(v_1, ..., v_n)&, op(p_1, ..., p_n), \tau) \\
&\triangleq mgu(v_1, p_1, mgu(v_2, p_2, ..., mgu(v_n, p_n, \tau)...))
\end{aligned}
$$

The equations should be understood as Kleene equations: One side of the equation is defined if and only if the other side is, and if they are defined, the two sides are equal. It is not hard to show that the definition is independent of order of unification in the third clause of this definition. Term environments are lifted from functions on variables to functions on arbitrary terms in the usual way. $\tau(e)$ is τ applied to the term e. Now define the relation v *matches* p to hold just in case $mgu(v, p, [\,])$ is defined. If $\{V_1, \ldots, V_n\}$ are the free variables in p then v *matches* p can be expressed as the first-order formula $\exists V_1, \ldots, V_n . p = v$.

Given a value v, a queue $q = v_1 \cdot v_2 \cdot \ldots \cdot v_m$ and a match $m = p_1 \rightarrow e_1 \,;\, \cdots \,;\, p_n \rightarrow e_n$, we define two matching operations to be used below:

$$
\begin{aligned}
&\textit{val-match}(v, m) \triangleq mgu(v, p_i, [\,])(e_i) \\
&\qquad\qquad \text{if } v \textit{ matches } p_i \text{ and } \forall j < i.\ not(v \textit{ matches } p_j) \\
&\textit{queue-match}(q, m) \triangleq mgu(v_m, p_i, [\,])(e_i) \\
&\qquad\qquad \text{if } v_m \textit{ matches } p_i \text{ and } \forall j < m.\ \forall k.\ not(v_j \textit{ matches } p_k) \\
&\qquad\qquad \text{and } \forall k < i.\ not(v_m \textit{ matches } p_k).
\end{aligned}
$$

Operational Semantics A *reduction context* $r[\cdot]$ is an Erlang expression with a "hole" in it. The reduction context defines the evaluation order of subexpressions in language constructs; here from left to right[3]. The result of placing e in (the hole of) a context $r[\cdot]$ is denoted $r[e]$. Contexts are defined by the grammar:

$$
\begin{aligned}
r[\cdot] ::= \quad & \cdot \\
\mid\ & op(v_1, \ldots, v_{i-1}, r[\cdot], e_{i+1}, \ldots, e_n) \qquad \text{for all } i : 1 \le i \le n \\
\mid\ & r[\cdot]\,\texttt{,}\,e \\
\mid\ & r[\cdot]\ e \mid f\ r[\cdot] \\
\mid\ & \texttt{case}\ r[\cdot]\ \texttt{of}\ m\ \texttt{end} \\
\mid\ & r[\cdot]!e \mid pid!r[\cdot] \\
\mid\ & \texttt{spawn}(r[\cdot], e) \mid \texttt{spawn}(f, r[\cdot])
\end{aligned}
$$

[3] An arbitrary simplification. The full Erlang language does not specify any evaluation ordering.

Definition 3 (Operational semantics). *The operational semantics of Core Erlang is defined by table 1.*

The rules defining the operational semantics of Core Erlang are separated into three classes: the *local rules* concern classical, side-effect free, computation steps of an Erlang expression, the *process rules* define the actions of a single process (an Erlang expression with an associated pid and queue), and the *system rules* give the semantics of the parallel composition operator. In the rules "*pid* fresh" requires *pid* to be a new pid, and $foreign(pid)(s)$ states that no process in the system state s has *pid* as its pid.

3 The Property Specification Logic

In this section we introduce a specification logic for Core Erlang. The logic is based on a first-order μ-calculus, corresponding, roughly, to Park's μ-calculus [Par76], extended with Erlang-specific features. Thus the logic is based on the first-order language of equality, extended with modalities reflecting the transition capabilities of processes and process configurations, least and greatest fixpoints, along with a few additional primitives.

Syntax Abstract formula syntax is determined by the following grammar where Z ranges over predicate variables parametrised by value vectors.

$$\begin{aligned}\phi ::= {} & p = p \mid p \neq p \mid \texttt{term}(p) = p \mid \texttt{queue}(p) = p \mid \\ & \texttt{local}(p) \mid \texttt{foreign}(p) \mid \texttt{atom}(p) \mid \texttt{unevaluated}(p) \mid \\ & \phi \wedge \phi \mid \phi \vee \phi \mid \forall V.\phi \mid \exists V.\phi \mid \\ & <>\phi \mid <p?p>\phi \mid <p!p>\phi \mid [\,]\phi \mid [p?p]\phi \mid [p!p]\phi \mid \\ & Z(p_1, \ldots, p_n) \mid \\ & (\mu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n) \mid (\nu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n)\end{aligned}$$

It is important to note that patterns p and value variables V in the case of formulas range not only over Erlang values, but also over message queues.

Intuitive Semantics The intuitive meaning of formulas is given as follows:

- Equality, inequality, the Boolean connectives and the quantifiers take their usual meanings.
- The purposes of $\texttt{term}(p_1) = p_2$ and $\texttt{queue}(p_1) = p_2$ are to pick up the values of terms and queues associated with a given pid p_1. $\texttt{term}(p_1) = p_2$ requires p_1 to equal the pid of a process which is part of the system state being predicated, and the Erlang expression associated with that pid to be identical to p_2. Similarly $\texttt{queue}(p_1) = p_2$ holds if the queue associated with p_1 is equal to p_2.
- $\texttt{atom}(p)$ holds if p is equal to an atom.

Local rules

SEQ : $v\,,\,e \longrightarrow e$

CASE : $\texttt{case}\ v\ \texttt{of}\ m\ \texttt{end} \longrightarrow \mathit{val\text{-}match}(v,m)$ if $\mathit{val\text{-}match}(v,m)$ is defined.

FUN : $f\ v \longrightarrow \texttt{case}\ v\ \texttt{of}\ m\ \texttt{end}$ if $f \stackrel{\triangle}{=} m$.

Process rules

LOCAL : $\langle r[e], pid, q\rangle \longrightarrow \left\langle r[e'], pid, q\right\rangle$ if $e \longrightarrow e'$.

SELF : $\langle r[\texttt{self}], pid, q\rangle \longrightarrow \langle r[pid], pid, q\rangle$

SPAWN : $\langle r[\texttt{spawn}(f,v)], pid, q\rangle \longrightarrow \left\langle r[pid'], pid, q\right\rangle \parallel \left\langle f\ v, pid', \epsilon\right\rangle$ if pid' fresh.

SEND : $\langle r[pid'!v], pid, q\rangle \xrightarrow{pid'!v} \langle r[v], pid, q\rangle$ if $pid \neq pid'$.

SELF–SEND : $\langle r[pid!v], pid, q\rangle \longrightarrow \langle r[v], pid, q\cdot v\rangle$

RECEIVE : $\langle r[\texttt{receive}\ m\ \texttt{end}], pid, q'\cdot v\cdot q''\rangle \longrightarrow \left\langle r[\mathit{queue\text{-}match}(q'\cdot v, m)], pid, q'\cdot q''\right\rangle$
if $\mathit{queue\text{-}match}(q'\cdot v, m)$ is defined.

INPUT : $\langle e, pid, q\rangle \xrightarrow{pid?v} \langle e, pid, q\cdot v\rangle$ for all values v.

System rules

COM : $s_1 \parallel s_2 \longrightarrow s_1' \parallel s_2'$ if $s_1 \xrightarrow{pid!v} s_1'$ and $s_2 \xrightarrow{pid?v} s_2'$.

INTERLEAVE1 : $s_1 \parallel s_2 \longrightarrow s_1' \parallel s_2$ if $s_1 \longrightarrow s_1'$.

INTERLEAVE2 : $s_1 \parallel s_2 \xrightarrow{pid?v} s_1' \parallel s_2$ if $s_1 \xrightarrow{pid?v} s_1'$.

INTERLEAVE3 : $s_1 \parallel s_2 \xrightarrow{pid!v} s_1' \parallel s_2$ if $s_1 \xrightarrow{pid!v} s_1'$ and $\mathit{foreign}(pid)(s_2)$.

Table 1. Operational semantics of Core Erlang

- $\texttt{local}(p)$ holds if p is equal to the pid of a process in the system state being predicated, and analogously $\texttt{foreign}(p)$ holds if p is equal to a pid and there is no process with pid p in the predicated system state.
- $\texttt{unevaluated}(p)$ holds if $\texttt{local}(p)$ does and the Erlang expression associated with p is not a ground value.
- $<>\phi$ holds if an internal transition is enabled to a system state satisfying ϕ. $[\,]\phi$ is the dual of $<>\phi$ (i.e. all states following an internal transition satisfy ϕ). $<p_1!p_2>\phi$ holds if an output transition with appropriate parameters is enabled to a state satisfying ϕ, and $<p_1?p_2>\phi$ is used similarly for input transitions.
- $\mu Z(V_1, \ldots, V_n).\phi$ is the least inclusive predicate Z satisfying the equation $\phi = Z(V_1, \ldots, V_n)$, while $\nu Z(V_1, \ldots, V_n).\phi$ is the most inclusive such predicate.

As is by now well known, monotone recursive definitions in a complete Boolean lattice have least and greatest solutions. This is what motivates the existence of predicates Z above. Greatest solutions are used, typically, for safety (i.e. invariant) properties, while least solutions are used for liveness (i.e. eventuality) properties. For readability we often prefer the notation $f(V_1, \ldots, V_n) \Leftarrow \phi$ for defining explicitly a formula atom f which is to be applied to pattern vectors $(p_1, \ldots, p_n)$, instead of the standard notation for least fixpoints, and similarly $f(V_1, \ldots, V_n) \Rightarrow \phi$ for greatest ones. In such definitions all value variables occurring freely in ϕ have to be in $V_1, \ldots, V_n$.

We use standard abbreviations like *true*, *false* and $\forall V_1, \ldots, V_n.\phi$. It is possible to define a de Morganised negation **not**, by the standard clauses along with clauses for the Erlang specific atomic propositions $\texttt{term}(p_1) = p_2$, $\texttt{queue}(p_1) = p_2$, $\texttt{local}(p)$, $\texttt{foreign}(p)$, or $\texttt{unevaluated}(p)$. This is an easy exercise given the formal semantics in table 2 below.

Definition 4 (Boolean Formula). *A formula ϕ is* boolean *if it has no occurrences of modal operators, neither occurrences of any of the Erlang-specific atomic propositions* $\texttt{term}(p_1) = p_2$, $\texttt{queue}(p_1) = p_2$, $\texttt{local}(p)$, $\texttt{foreign}(p)$, *or* $\texttt{unevaluated}(p)$.

Boolean formulas are important as they either hold globally or not at all.

The formal semantics of formulas is given as a set $\| \phi \| \eta \subseteq \mathcal{S}$, where η is a *valuation* providing interpretations for variables (value variables, or predicate variables). We use the standard notation $\eta\{v/V\}$ for *updating* η so that the new environment maps V to v and acts otherwise as η. Predicate maps $f : (v_1, \ldots, v_n) \mapsto A \subseteq \mathcal{S}$ are ordered by $\leq$ defined as subset containment lifted pointwise.

Definition 5 (Formula semantics). *The semantics of formulas is defined by table 2, where in the last two defining equations,* $M(f)(v_1, \ldots, v_n) \triangleq \| \phi \| \eta\{f/Z, v_1/V_1, \ldots, v_n/V_n\}$.

$$
\begin{array}{rcl}
\| p_1 = p_2 \| \eta & \triangleq & \{ s \mid p_1\eta = p_2\eta \} \\
\| p_1 \neq p_2 \| \eta & \triangleq & \{ s \mid p_1\eta \neq p_2\eta \} \\
\| \mathtt{term}(p_1) = p_2 \| \eta & \triangleq & \{ \langle p_2\eta, p_1\eta, q \rangle \| s \mid p_1\eta \text{ is a pid} \} \\
\| \mathtt{queue}(p_1) = p_2 \| \eta & \triangleq & \{ \langle e, p_1\eta, p_2\eta \rangle \| s \mid p_1\eta \text{ is a pid} \} \\
\| \mathtt{local}(p) \| \eta & \triangleq & \{ \langle e, p\eta, q \rangle \| s \mid p\eta \text{ is a pid} \} \\
\| \mathtt{foreign}(p) \| \eta & \triangleq & \{ s \mid p\eta \text{ is a pid that does not belong to a process in } s \} \\
\| \mathtt{atom}(p) \| \eta & \triangleq & \{ s \mid p\eta \text{ is an atom} \} \\
\| \mathtt{unevaluated}(p) \| \eta & \triangleq & \{ \langle e, p\eta, q \rangle \| s \mid e \text{ is not a ground value, } p\eta \text{ is a pid} \} \\
\| \phi_1 \wedge \phi_2 \| \eta & \triangleq & \| \phi_1 \| \eta \cap \| \phi_2 \| \eta \\
\| \phi_1 \vee \phi_2 \| \eta & \triangleq & \| \phi_1 \| \eta \cup \| \phi_2 \| \eta \\
\| \forall V.\phi \| \eta & \triangleq & \bigcap \{ \| \phi \| \eta\{v/V\} \mid v \text{ a value} \} \\
\| \exists V.\phi \| \eta & \triangleq & \bigcup \{ \| \phi \| \eta\{v/V\} \mid v \text{ a value} \} \\
\| <>\phi \| \eta & \triangleq & \{ s \mid \exists\, s' \in \| \phi \| \eta.\ s \longrightarrow s' \} \\
\| <p_1?p_2>\phi \| \eta & \triangleq & \{ s \mid \exists\, s' \in \| \phi \| \eta.\ s \xrightarrow{p_1\eta?p_2\eta} s' \} \\
\| <p_1!p_2>\phi \| \eta & \triangleq & \{ s \mid \exists\, s' \in \| \phi \| \eta.\ s \xrightarrow{p_1\eta!p_2\eta} s' \} \\
\| []\phi \| \eta & \triangleq & \{ s \mid \forall\, s'.\ s \longrightarrow s' \text{ implies } s' \in \| \phi \| \eta \} \\
\| [p_1?p_2]\phi \| \eta & \triangleq & \{ s \mid \forall\, s'.\ s \xrightarrow{p_1\eta?p_2\eta} s' \text{ implies } s' \in \| \phi \| \eta \} \\
\| [p_1!p_2]\phi \| \eta & \triangleq & \{ s \mid \forall\, s'.\ s \xrightarrow{p_1\eta!p_2\eta} s' \text{ implies } s' \in \| \phi \| \eta \} \\
\| Z(p_1, \ldots, p_n) \| \eta & \triangleq & (Z\eta)(p_1\eta, \ldots, p_n\eta) \\
\multicolumn{3}{l}{\| (\mu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n) \| \eta} \\
 & \triangleq & \left(\bigcap \{ f \mid M(f) \leq f \}\right)(p_1\eta, \ldots, p_n\eta) \\
\multicolumn{3}{l}{\| (\nu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n) \| \eta} \\
 & \triangleq & \left(\bigcup \{ f \mid f \leq M(f) \}\right)(p_1\eta, \ldots, p_n\eta)
\end{array}
$$

Table 2. Formula Semantics

Example Formulas The combination of recursive definitions with data types makes the logic very expressive. For instance, the type of natural numbers is the least set containing zero and closed under successor. The property of being a natural number can hence be defined as a least fixpoint:

$$nat(P) \Leftarrow P = 0 \vee \exists V.(nat(V) \wedge P = V + 1) \tag{1}$$

Using this idea quite general flat data types can be defined. One can also define "weak" modalities that are insensitive to the specific number of internal transitions in the following style:

$$\begin{array}{ll} [[\,]]\phi \Rightarrow \phi \wedge [\,][[\,]]\phi & [[V!V']]\phi \;\stackrel{\Delta}{=}\; [[\,]][V!V'][[\,]]\phi \\ \ll\gg\phi \Leftarrow \phi \vee <>\ll\gg\phi & \ll V!V'\gg\phi \;\stackrel{\Delta}{=}\; \ll\gg<V!V'>\ll\gg\phi \end{array}$$

Observe the use of formula parameters, and the use of $\stackrel{\Delta}{=}$ for non-recursive definitions. A temporal property like *always* is also easily defined:

$$\begin{array}{l} always(\phi) \Rightarrow \\ \quad \phi \wedge [\,]always(\phi) \wedge \forall V, V'.[V?V']always(\phi) \wedge \forall V, V'.[V!V']always(\phi) \end{array} \tag{2}$$

The definition of *always* illustrates the typical shape of behavioural properties: one mixes state properties (ϕ) with properties concerning program transitions, separated into distinct cases for internal transitions (i.e. $[\,]always(\phi)$), input transitions and output transitions.

Eventuality operators are more delicate as progress is in general only made when internal or output transitions are taken. This can be handled, though, by nesting of minimal and maximal definitions.

Billing Agents: Specification We enumerate some desired properties of the billing agent system introduced in the previous section.

Disallowing spontaneous account withdrawals. The first correctness requirement forbids spontaneous withdrawals from the user account by the billing agent. This implies the invariant property that the number of attempts for transfers from the user account should be less than or equal to the number of requests for using the resource:

$$\begin{array}{l} safe(Ag, BankPid, UAcc, N) \;\;\Rightarrow \\ [\,]safe(Ag, BankPid, UAcc, N) \\ \wedge\; \forall P, V.[P?V] \\ \quad \left(\begin{array}{l} \;\; isuse(P, V, Ag) \wedge safe(Ag, BankPid, UAcc, N+1) \\ \vee\; \mathtt{not}(isuse(P, V, Ag)) \wedge safe(Ag, BankPid, UAcc, N) \\ \vee\; contains(V, UAcc) \end{array}\right) \\ \wedge\; \forall P, V.[P!V] \\ \quad \left(\begin{array}{l} \;\; istrans(P, V, BankPid, UAcc) \wedge N > 0 \wedge safe(Ag, BankPid, UAcc, N-1) \\ \vee\; \mathtt{not}(istrans(P, V, BankPid, UAcc)) \wedge safe(Ag, BankPid, UAcc, N) \end{array}\right) \end{array}$$

where the predicates *isuse* and *istrans* recognise resource requests and money transfers:

$$isuse(P, V, Ag) \triangleq P = Ag \wedge \exists Pid. V = \{use, Pid\}$$

$$istrans(P, V, BankPid, UAcc) \triangleq$$
$$P = BankPid \wedge (\exists Pid, Acc.\ V = \{trans, \{UAcc, Acc\}, Pid)$$

So, a billing agent with pid Ag, pid of a trusted bank agent $BankPid$, and user account $UAcc$ is defined to be *safe* if the difference N between the number of requests for using the resource (messages of type $\{use, Pid\}$ received in the process mailbox) and the number of attempts for transfers from the user account (messages of type $\{trans, \{UAcc, Acc\}, pid\}$ sent to $BankPid$) is always non-negative. Since this difference is initially equal to zero, we expect $billagent(ResPid, BankPid, RAcc, UAcc)$ to satisfy $safe(Ag, BankPid, UAcc, 0)$. The predicate $contains(v, v')$ is defined via structural induction over an Erlang value (or queue) v and holds if v' is a component of v (such as v being a tuple $v = \{v_1, v_2\}$ and either $v = v'$ or $contains(v_1, v')$ or $contains(v_2, v')$). We omit the easy definition.

Expected Service is Received. Other interesting properties concern facts like: Denial of service responses correspond to failed money transfers, and returning the resource to the proper user. These sorts of properties are not hard to formalise in a style similar to the first example.

Preventing Abuse by a Third Party. The payment scheme presented here depends crucially on the non-communication of private names. For instance, even if we can prove that a resource manager or billing agent does not make illegal withdrawals nothing stops the resource manager from communicating the user account key to a third party, that can then access the account in non-approved ways.

Thus we need to prove at least that the system communicates neither the user account key nor the agent process identifier. Perhaps the service user also requests that her identity not be known outside of the system, in such a case the return process identifiers may not be communicated either. As an example, the property that the system does not communicate the user account key is captured by $notrans(UAcc)$ given the definition below.

$$\begin{aligned} notrans(A) \Rightarrow\ & [\,]\, notrans(A) \\ & \wedge\ \forall V, V'.[V?V']\,(contains(V', A) \vee notrans(A)) \\ & \wedge\ \forall V, V'.[V!V']\,(\mathtt{not}(contains(V', A)) \wedge notrans(A)) \end{aligned}$$

4 Proof System, Local Rules

In this section we give the formal semantics of sequents, present the most important rules of the proof system and establish the soundness of the proof rules. Consideration of fixed points and discharge is delayed until later.

Sequents We start by introducing the syntax and semantics of sequents.

Definition 6 (Sequent, Sequent Semantics).

1. *An* assertion *is a pair* $s : \phi$. *An* assumption *is an assertion of the shape either* $S : \phi$ *or* $\langle E, P, Q \rangle : \phi$.
2. *A valuation* η *is said to* validate *an assertion* $s : \phi$ *when* $s\eta \in \| \phi \| \eta$.
3. *A* sequent *is an expression of the shape* $\Gamma \vdash \Delta$ *where* Γ *is a multiset of assumptions, and* Δ *is a multiset of assertions.*
4. *The sequent* $\Gamma \vdash \Delta$ *is* valid, *written* $\Gamma \models \Delta$, *provided for all valuations* η, *if all assumptions in* Γ *are validated by* η, *then some assertion in* Δ *is validated by* η.

We use standard notation for sequents. For instance we use comma for multiset union, and identify singleton sets with their member. Thus, for instance, $\Gamma, S : \phi$ is the same multiset as $\Gamma \cup \{S : \phi\}$.

Boolean formulas are independent of the system state being predicated. Thus, when ϕ is boolean we often abbreviate an assertion like $s : \phi$ as just ϕ.

Proof Rules The Gentzen-type proof system comes in a number of installments.

1. The *structural rules* govern the introduction, elimination, and use of assertions.
2. The *logical rules* introduce the logical connectives to the left and to the right of the turnstile.
3. The *equality* rules account for equations and inequations.
4. The *atomic formula rules* control the Erlang-specific atomic formulas such as the term and queue extractors, predicates such as `local` and `foreign`, etc.
5. Finally the *modal rules* account for the modal operators.

It should be noted that, as we for now lack completeness results, the selection of rules which we present is to some extent arbitrary. Indeed some rules have deliberately been left out of this presentation for reasons of space. This is the case, in particular, for the groups (4) and (5). We comment more on this below. Moreover, because of the lack of completeness or syntactical cut-elimination results, we know little about admissibility of several rules such as Cut below.

Structural Rules

$$\mathsf{Id}\ \frac{}{\Gamma, s : \phi \vdash s : \phi, \Delta} \qquad \mathsf{Bool\ Id}\ \frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \phi\ \textit{Boolean}$$

$$\mathsf{WeakL}\ \frac{\Gamma \vdash \Delta}{\Gamma, s : \phi \vdash \Delta} \qquad \mathsf{WeakR}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash s : \phi, \Delta}$$

$$\mathsf{ContrL}\ \frac{\Gamma, s : \phi, s : \phi \vdash \Delta}{\Gamma, s : \phi \vdash \Delta} \qquad \mathsf{ContrR}\ \frac{\Gamma \vdash s : \phi, s : \phi, \Delta}{\Gamma \vdash s : \phi, \Delta}$$

$$\mathsf{Cut}\ \frac{\Gamma \vdash s:\phi,\Delta \qquad \Gamma, s:\phi \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\mathsf{ProcCut}\ \frac{\Gamma \vdash s_1:\psi,\Delta \qquad \Gamma, S:\psi \vdash s_2:\phi,\Delta}{\Gamma \vdash s_2\{s_1/S\}:\phi,\Delta}\ S\ \mathit{fresh}$$

Observe that the standard cut rule is not derivable from the process cut rule as in Cut the system state s might appear in Γ or Δ.

Logical Rules

$$\mathsf{AndL}\ \frac{\Gamma, s:\phi_1, s:\phi_2 \vdash \Delta}{\Gamma, s:\phi_1 \wedge \phi_2 \vdash \Delta} \quad \mathsf{AndR}\ \frac{\Gamma \vdash s:\phi_1,\Delta \qquad \Gamma \vdash s:\phi_2,\Delta}{\Gamma \vdash s:\phi_1 \wedge \phi_2,\Delta}$$

$$\mathsf{OrL}\ \frac{\Gamma, s:\phi_1 \vdash \Delta \qquad \Gamma, s:\phi_2 \vdash \Delta}{\Gamma, s:\phi_1 \vee \phi_2 \vdash \Delta} \quad \mathsf{OrR}\ \frac{\Gamma \vdash s:\phi_1, s:\phi_2,\Delta}{\Gamma \vdash s:\phi_1 \vee \phi_2,\Delta}$$

$$\mathsf{AllL}\ \frac{\Gamma, s:\phi\{v/V\} \vdash \Delta}{\Gamma, s:\forall V.\phi \vdash \Delta} \quad \mathsf{AllR}\ \frac{\Gamma \vdash s:\phi,\Delta}{\Gamma \vdash s:\forall V.\phi,\Delta}\ V\ \mathit{fresh}$$

$$\mathsf{ExL}\ \frac{\Gamma, s:\phi \vdash \Delta}{\Gamma, s:\exists V.\phi \vdash \Delta}\ V\ \mathit{fresh} \quad \mathsf{ExR}\ \frac{\Gamma \vdash s:\phi\{v/V\},\Delta}{\Gamma \vdash s:\exists V.\phi,\Delta}$$

We claim that for the full proof system the left and right `not` introduction rules

$$\mathsf{NotL}\ \frac{\Gamma, s:\mathtt{not}\ \phi \vdash \Delta}{\Gamma \vdash s:\phi,\Delta} \quad \mathsf{NotR}\ \frac{\Gamma \vdash s:\mathtt{not}\ \phi,\Delta}{\Gamma, s:\phi \vdash \Delta}$$

are derivable. Given this a number of other useful rules such as the rule of contradiction

$$\mathsf{Contrad}\ \frac{\Gamma \vdash s:\phi,\Delta \qquad \Gamma \vdash s:\mathtt{not}\,\phi,\Delta}{\Gamma \vdash \Delta}$$

become easily derivable as well.

Equality Rules For the rules which follow recall that op ranges over data type constructors like zero, successor, unit, binary tupling, etc.

$$\mathsf{Refl}\ \frac{}{\Gamma \vdash p = p,\Delta}$$

$$\mathsf{Subst}\ \frac{\Gamma\{p_1/V\} \vdash \Delta\{p_1/V\}}{\Gamma\{p_2/V\}, p_1 = p_2 \vdash \Delta\{p_2/V\}}$$

$$\mathsf{ConstrIneq}\ \frac{op \neq op'}{\Gamma, op(p_1,\ldots,p_n) = op'(p'_1,\ldots,p'_{n'}) \vdash \Delta}$$

$$\mathsf{ConstrEqL}\ \frac{\Gamma, p_i = p'_i \vdash \Delta}{\Gamma, op(p_1,\ldots,p_n) = op(p'_1,\ldots,p'_n) \vdash \Delta}$$

$$\mathsf{ConstrEqR}\ \frac{\Gamma \vdash p_1 = p'_1,\Delta \quad \cdots \quad \Gamma \vdash p_n = p'_n,\Delta}{\Gamma \vdash op(p_1,\ldots,p_n) = op(p'_1,\ldots,p'_n),\Delta}$$

Atomic Formula Rules For the Erlang-specific primitives we give only a sample here, of rules governing the term and queue extraction constructs.

$$\mathsf{TermL}\ \frac{\Gamma, E = p \vdash \Delta}{\Gamma, \langle E, Pid, Q\rangle : \mathtt{term}(Pid) = p \vdash \Delta}$$

$$\mathsf{TermR}\ \frac{\Gamma \vdash p_1 = p_2, \Delta}{\Gamma \vdash \langle p_1, pid, q\rangle : \mathtt{term}(pid) = p_2, \Delta}$$

$$\mathsf{ParTerm}\ \frac{\Gamma \vdash s_1 : \mathtt{term}(p_1) = p_2, \Delta}{\Gamma \vdash s_1 \parallel s_2 : \mathtt{term}(p_1) = p_2, \Delta}$$

$$\mathsf{QueueL}\ \frac{\Gamma, Q = p \vdash \Delta}{\Gamma, \langle E, Pid, Q\rangle : \mathtt{queue}(Pid) = p \vdash \Delta}$$

$$\mathsf{QueueR}\ \frac{\Gamma \vdash p_2 = p_3, \Delta}{\Gamma \vdash \langle p_1, pid, p_2\rangle : \mathtt{queue}(pid) = p_3, \Delta}$$

$$\mathsf{ParQueue}\ \frac{\Gamma \vdash s_1 : \mathtt{queue}(p_1) = p_2, \Delta}{\Gamma \vdash s_1 \parallel s_2 : \mathtt{queue}(p_1) = p_2, \Delta}$$

Modal Rules The rules governing the modal operators come in three flavours: *Monotonicity rules* that capture the monotonicity of the modal operators as unary functions on sets, *transition rules* ("model checking-like rules") that prove modal properties for closed system states by exploring the operational semantics transition relations, and *compositional rules* that decompose modal properties of composite system states in terms of modal properties of their components.

Monotonicity Rules The monotonicity rules are similar to well-known rules from standard Gentzen-type accounts of modal logic. Let α stand for either no label or for a label of the form $p_1?p_2$ or $p_1!p_2$.

$$\mathsf{Mon1}\ \frac{\Gamma, S : \phi, S : \phi_1, \ldots, S : \phi_m \vdash S : \psi_1, \ldots, S : \psi_n, \Delta}{\Gamma, s : <\alpha>\phi, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m \vdash s : <\alpha>\psi_1, \ldots, s : <\alpha>\psi_n, \Delta}\ S\ \mathit{fresh}$$

$$\mathsf{Mon2}\ \frac{\Gamma, S : \phi_1, \ldots, S : \phi_m \vdash S : \psi, S : \psi_1, \ldots, S : \psi_n, \Delta}{\Gamma, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m \vdash s : [\alpha]\psi, s : <\alpha>\psi_1, \ldots, s : <\alpha>\psi_n, \Delta}\ S\ \mathit{fresh}$$

Transition Rules The transition rules explore symbolic versions of the operational semantics transition relations. These relations have the shape

$$s \xrightarrow{\mathit{pre}, \alpha, \mathit{post}} s'$$

where *pre* is a (first-order) precondition for firing the transition from s to s', α (as above) is the transition label, and *post* is the resulting (first-order) postcondition in terms of e.g. variable bindings. Since the transformation of the semantics in table 1 to a symbolic one is straightforward, we will only give a single example of the transformation where the original rule

$$\text{SEND} : \langle r[pid'!v], pid, q\rangle \xrightarrow{pid'!v} \langle r[v], pid, q\rangle \text{ if } pid \neq pid'$$

becomes the symbolic rule

$$\text{SEND}_s : \langle r[pid'!v], pid, q\rangle \xrightarrow{pid \neq pid',\, pid'!v,\, true} \langle r[v], pid, q\rangle\,.$$

The transition rules now become the following two:

$$\mathsf{Diamond}\ \frac{\Gamma, pre, post \vdash s' : \phi \quad \Gamma \vdash pre \quad s \xrightarrow{pre, \alpha, post} s'}{\Gamma \vdash s : <\alpha>\phi}$$

$$\mathsf{Box}\ \frac{\{\Gamma, \alpha = \alpha', pre, post \vdash s' : \phi \mid s \xrightarrow{pre, \alpha', post} s'\}}{\Gamma \vdash s : [\alpha]\phi}$$

In the rule Box we use $\alpha = \alpha'$ as an abbreviation. If α and α' have different sorts (e.g. α is empty and α' is not) then $\alpha = \alpha'$ abbreviates *false*. If α and α' are both empty, $\alpha = \alpha'$ abbreviates *true*. If α and α' are both send actions, e.g. $\alpha = p_1!p_2$ and $\alpha' = p_1'!p_2'$ then $\alpha = \alpha'$ abbreviates the conjunction $p_1 = p_1' \wedge p_2 = p_2'$. A similar condition holds if α and α' are both input actions.

Compositional Rules We give rules for inferring modal properties of composite system states $s_1 \parallel s_2$ in terms of modal properties of the parts s_1 and s_2.

$$\mathsf{DiaPar1}\ \frac{\Gamma, S_1 : \phi, S_2 : \psi \ \vdash\ S_1 \parallel S_2 : \theta, \Delta}{\Gamma, s_1 : <p_1!p_2>\phi, s_2 : <p_1?p_2>\psi \ \vdash\ s_1 \parallel s_2 : <>\theta, \Delta}\quad S_1, S_2\ \mathit{fresh}$$

$$\mathsf{DiaPar2}\ \frac{\Gamma, S : \phi \ \vdash\ S \parallel s_2 : \psi, \Delta}{\Gamma, s_1 : <>\phi \ \vdash\ s_1 \parallel s_2 : <>\psi, \Delta}\quad S\ \mathit{fresh}$$

$$\mathsf{DiaPar3}\ \frac{\Gamma, S : \phi \ \vdash\ S \parallel s_2 : \psi, \Delta}{\Gamma, s_1 : <p_1?p_2>\phi \ \vdash\ s_1 \parallel s_2 : <p_1?p_2>\psi, \Delta}\quad S\ \mathit{fresh}$$

$$\mathsf{DiaPar4}\ \frac{\Gamma, S : \phi \ \vdash\ S \parallel s_2 : \psi, \Delta \qquad \Gamma \ \vdash\ s_2 : \mathtt{foreign}(p_1)}{\Gamma, s_1 : <p_1!p_2>\phi \ \vdash\ s_1 \parallel s_2 : <p_1!p_2>\psi, \Delta}\quad S\ \mathit{fresh}$$

$$\text{BoxPar1}\ \frac{\begin{array}{c}\Gamma, S_1 : \phi_1 \vdash S_1 : [\,]\,\psi_{[\,]}, \Delta \\ \Gamma, S_1 : \psi_{[\,]}, s_2 : \phi_2 \vdash S_1 \parallel s_2 : \phi \\ \Gamma, S_2 : \phi_2 \vdash S_2 : [\,]\,\theta_{[\,]}, \Delta \\ \Gamma, s_1 : \phi_1, S_2 : \theta_{[\,]} \vdash s_1 \parallel S_2 : \phi \\ \Gamma, S_1 : \phi_1 \vdash S_1 : [V!V']\,\psi_{[V!V']}, \Delta \\ \Gamma, S_2 : \phi_2 \vdash S_2 : [V?V']\,\theta_{[V?V']}, \Delta \\ \Gamma, S_1 : \psi_{[V!V']}, S_2 : \theta_{[V?V']} \vdash S_1 \parallel S_2 : \phi \\ \Gamma, S_1 : \phi_1 \vdash S_1 : [V?V']\,\psi_{[V?V']}, \Delta \\ \Gamma, S_2 : \phi_2 \vdash S_2 : [V!V']\,\theta_{[V!V']}, \Delta \\ \Gamma, S_1 : \psi_{[V?V']}, S_2 : \theta_{[V!V']} \vdash S_1 \parallel S_2 : \phi\end{array}}{\Gamma, s_1 : \phi_1, s_2 : \phi_2 \vdash s_1 \parallel s_2 : [\,]\,\phi, \Delta}\ S_1, S_2, V, V'\ \mathit{fresh}$$

$$\text{BoxPar2}\ \frac{\begin{array}{c}\Gamma, S_1 : \phi_1 \vdash S_1 : [p_1?!p_2]\,\psi_{[p_1?!p_2]}, \Delta \\ \Gamma, S_1 : \psi_{[p_1?!p_2]}, s_2 : \phi_2 \vdash S_1 \parallel s_2 : \phi \\ \Gamma, S_2 : \phi_2 \vdash S_2 : [p_1?!p_2]\,\theta_{[p_1?!p_2]}, \Delta \\ \Gamma, s_1 : \phi_1, S_2 : \theta_{[p_1?!p_2]} \vdash s_1 \parallel S_2 : \phi\end{array}}{\Gamma, s_1 : \phi_1, s_2 : \phi_2 \vdash s_1 \parallel s_2 : [p_1?!p_2]\,\phi, \Delta}\ S_1, S_2\ \mathit{fresh}$$

The last two rules are less complex than they appear. They just represent the obvious case analyses needed to infer the α-indexed necessity properties stated in their conclusions. For instance in the case where α is empty it is clearly required to analyse all cases where either

- s_1 performs an internal transition and s_2 does not,
- s_2 performs an internal transition and s_1 does not,
- s_1 performs a send action and s_2 performs a corresponding receive action, and
- s_2 performs a send action and s_1 performs a corresponding receive action.

In fact, the last two rules are simplified versions of more general rules having multiple assumptions about s_1 and s_2 in the conclusion. However, a formal statement of these rules, while quite trivial, becomes graphically rather unwieldy.

Observe that these rules deal only with composite system states of the form $s_1 \parallel s_2$. Related compositional rules are needed also for singleton processes $\langle e, pid, q\rangle$, to decompose properties of e in terms of properties of its constituent parts. For closed processes, however, and for tail-recursive programs e (as is the case in the main example considered later), the transition rules are quite adequate, and so this collection of compositional proof rules for sequential processes is not considered further in the present paper.

Theorem 1 (Local Soundness). *Each of the above rules is sound, i.e. the conclusion is a valid sequent whenever all premises are so and all side conditions hold.*

Proof. Here we show soundness of the more interesting rules only; the proofs of the remaining rules are either standard or similar to these.

Rule ProcCut. Assume the premises of the rule are valid sequents. Assume all assumptions in Γ are validated by some arbitrary valuation η. Then validity of the first premise implies that also some assertion in $s_1 : \psi, \Delta$ is validated by η. Since S is fresh (i.e. not free in Γ or Δ) and since $s_1\eta$ equals $S\eta\{s_1\eta/S\}$, some assertion in $S : \psi, \Delta$ is validated by $\eta\{s_1\eta/S\}$. Then either some assertion in Δ is validated by $\eta\{s_1\eta/S\}$, or all assertions in $\Gamma, S : \psi$ are validated by $\eta\{s_1\eta/S\}$. In the latter case, the validity of the second premise implies that also some assertion in $s_2 : \phi, \Delta$ is validated by $\eta\{s_1\eta/S\}$. Since S is fresh and since $s_2\eta\{s_1\eta/S\}$ equals $s_2\{s_1/S\}\eta$, some assertion in $s_2\{s_1/S\} : \phi, \Delta$ is validated by η. Hence the conclusion of the rule is also a valid sequent.

Rule Mon1. Assume the premise of the rule is a valid sequent. Assume all assumptions in $\Gamma, s : <\alpha>\phi, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m$ are validated by η. Then, according to the semantics of the "diamond" and "box" modalities, there is a closed system state s' such that $s\eta \xrightarrow{\alpha\eta} s'$ and s' satisfies all formulas in $\phi, \phi_1, \ldots, \phi_m$ under η. Since s' equals $S\eta\{s'/S\}$, and since S is fresh, all assumptions in $\Gamma, S : \phi, S : \phi_1, \ldots, S : \phi_m$ are validated by $\eta\{s'/S\}$. By validity of the premise, some assertion in $S : \psi_1, \ldots, S : \psi_n, \Delta$ is also validated by $\eta\{s'/S\}$. This means that either some assertion in Δ is validated by η, or s' satisfies some formula in $\psi_1, \ldots, \psi_n$ under η, and consequently some assertion in $s : <\alpha>\psi_1, \ldots, s : <\alpha>\psi_n, \Delta$ is validated by η. Hence the conclusion of the rule is also a valid sequent.

Rule DiaPar1. Assume the premise of the rule is valid. Assume all assumptions in $\Gamma, s_1 : <p_1!p_2>\phi, s_2 : <p_1?p_2>\psi$ are validated by η. Then there are closed system states s' and s'' such that $s_1\eta \xrightarrow{p_1\eta!p_2\eta} s' \in\parallel \phi \parallel \eta$ and $s_2\eta \xrightarrow{p_1\eta?p_2\eta} s'' \in\parallel \psi \parallel \eta$. Since s' equals $S_1\eta\{s'/S_1, s''/S_2\}$ and s'' equals $S_2\eta\{s'/S_1, s''/S_2\}$, and since S_1 and S_2 are fresh, all assumptions in $\Gamma, S_1 : \phi, S_2 : \psi$ are validated by $\eta\{s'/S_1, s''/S_2\}$. Then, by validity of the premise, some assertion in $S_1 \parallel S_2 : \theta, \Delta$ is also validated by $\eta\{s'/S_1, s''/S_2\}$. As a consequence, either some assertion in Δ is validated by η, or $s' \parallel s'' \in\parallel \theta \parallel \eta$. In the latter case we have $s_1\eta \parallel s_2\eta \longrightarrow s' \parallel s'' \in\parallel \theta \parallel \eta$, and therefore some assertion in $s_1 \parallel s_2 : <>\theta, \Delta$ is validated by η. Hence the conclusion of the rule is also valid.

Rule BoxPar1. The proof is along the lines of the preceding ones and shall only be sketched here. Assume the premises to the rule are valid sequents. Assume all assumptions in $\Gamma, s_1 : \phi_1, s_2 : \phi_2$ are validated by η. Then, from the first two premises it follows that either some assertion in Δ is validated by η, or it is the case that for every closed system state s such that $s_1\eta \longrightarrow s$, process $s \parallel s_2\eta$ satisfies ϕ under η. Similarly, the next two assumptions imply that $s_1\eta \parallel s$ satisfies ϕ under η whenever $s_2\eta \longrightarrow s$. From the next group of three assumptions we obtain that $s' \parallel s''$ satisfies ϕ under η whenever $s_1\eta \xrightarrow{v'!v''} s'$ and $s_2\eta \xrightarrow{v'?v''} s''$ for some values v' and v''. The last three premises imply that $s' \parallel s''$ satisfies ϕ under η whenever $s_1\eta \xrightarrow{v'?v''} s'$ and $s_2\eta \xrightarrow{v'!v''} s''$ for some values v' and v''. As a consequence of these relationships, either some assertion in Δ is validated by η, or every closed system state s such that $s_1\eta \parallel s_2\eta \longrightarrow s$

satisfies ϕ under η. Hence, some assertion in $s_1 \parallel s_2 : [\,]\,\phi, \Delta$ is validated by η, and therefore the conclusion of the rule is valid. $\square$

5 Inductive and Coinductive Reasoning

To handle recursively defined formulas some mechanisms are needed for successfully terminating proof construction when this is deemed to be safe. We discuss the ideas on the basis of two examples.

Example 1 (Coinduction). Consider the following Core Erlang function:

$$stream(N, Out) \rightarrow Out!N, stream(N+1, Out).$$

which outputs the increasing stream N, $N+1$, $N+2$, ... along Out. The specification of the program could be that it can always output some value along Out:

$$stream\_spec(Out) = always(\exists X.\ll Out!X \gg true)$$

The goal sequent takes the shape

$$Out \neq P \vdash \langle stream(N, Out), P, Q\rangle : stream\_spec(Out). \tag{3}$$

That is, assuming $Out \neq P$ (since otherwise output will go to the stream process itself), and started with pid P and any input queue Q, the property *stream_spec*(Out) will hold. The first step is to unfold the formula definition. This results in a proof goal of the shape

$$Out \neq P \vdash \langle stream(N, Out), P, Q\rangle : always(\exists X.\ll Out!X \gg true). \tag{4}$$

Using the proof rules (4) is easily reduced to the following subgoals:

$$Out \neq P \vdash \langle stream(N, Out), P, Q\rangle : \ll Out!N \gg true \tag{5}$$
$$Out \neq P \vdash \langle stream((N+1), Out), P, Q\rangle : always(\exists X.\ll Out!X \gg true) \tag{6}$$
$$Out \neq P \vdash \langle stream(N, Out), P, Q \cdot V'\rangle : always(\exists X.\ll Out!X \gg true). \tag{7}$$

Proving (5) is straightforward using the local proof rules and fixed point unfolding. For (6) and (7) we see that these goals are both instances of a proof goal, namely (4), which has already been unfolded.

Continuing proof construction in example 1 beyond (6) and (7) is clearly futile: The further information contained in these sequents compared with (4) does not bring about any new potentiality for proof. So we would like the nodes (6) and (7) to be discharged, as this happens to be safe. This is not hard to see: The fixed point

$$\phi = always(\exists V.\ll Out!V \gg true)$$

can only appear at its unique position in the sequent (6) because it did so in the sequent (4). We can say that ϕ is regenerated along the path from (4) to (6).

Moreover, *always* constructs a greatest fixed point formula. It turns out that the sequent (6) can be discharged for these reasons. In general, however, fixed point unfoldings are not at all as easily analysed. Alternation is a well-known complication. The basic intuition, however, is that in this case sequents can be discharged for one of two reasons:

1. Coinductively, because a member of Δ is regenerated through a greatest fixed point formula, as in example 1.
2. Inductively, because a member of Γ is regenerated through a least fixed point formula.

Intuitively the assumption of a least fixed point property can be used to determine a kind of progress measure ensuring that loops of certain sorts must eventually be exited. This significantly increases the utility of the proof system. For instance it allows for datatype induction to be performed.

Example 2 (Induction). Consider the following function:

$$stream2(N+1, Out) \to Out!N, stream2(N, Out)$$

which outputs a decreasing stream of numbers along Out. If N is a natural number, *stream2* has the property that, provided it does not deadlock, it will eventually output zero along Out. This property can be formalised as follows:

$$evzero(Out) \Leftarrow [\,]evzero(Out) \wedge \forall V.[Out!V](V = 0 \vee evzero(Out))$$

The goal sequent is

$$Out \neq P, nat(N) \vdash \langle stream2(N+1, Out), P, Q\rangle : evzero(Out) \tag{8}$$

where *nat* is defined in (1). The least fixed point appearing in the definition of *nat* will be crucial for discharge later in the proof. By unfolding the function according to its definition we obtain the sequent:

$$Out \neq P, nat(N) \vdash \langle (Out!N, stream2(N, Out)), P, Q\rangle : evzero(Out)$$

Now the formula has to be unfolded, resulting in a conjunction and hence in two sub-goals. The first of these is proved trivially since the system state cannot perform any internal transition when $Out \neq P$. The second sub-goal, after handling the universal quantifier, becomes:

$$\begin{aligned}&Out \neq P, nat(N)\\ &\vdash \langle (Out!N, stream2(N, Out)), P, Q\rangle : [Out!V](V = 0 \vee evzero(Out))\end{aligned} \tag{9}$$

By following the output transition enabled at this state we come a step closer to showing that zero is eventually output along Out:

$$Out \neq P, nat(N), N = V \vdash \langle stream2(N, Out), P, Q\rangle : V = 0, evzero(Out) \tag{10}$$

In the next step we perform a case analysis on N by unfolding $nat(N)$. This results in a disjunction on the left amounting to whether N is zero or not, and yields the two sub-goals:

$$Out \neq P, N = 0, N = V \vdash \langle stream2(N, Out), P, Q\rangle : V = 0 \tag{11}$$

$$\begin{aligned} &Out \neq P, nat(N'), N = N' + 1, N = V \\ &\quad \vdash \langle stream2(N, Out), P, Q\rangle : evzero(Out) \end{aligned} \tag{12}$$

The first of these is proved trivially. The second can be simplified to:

$$Out \neq P, nat(N') \vdash \langle stream2(N' + 1, Out), P, Q\rangle : evzero(Out) \tag{13}$$

This sequent is an instance of the initial goal sequent (8). Furthermore, it was obtained by regenerating the least fixpoint formula $nat(N)$ on the left. This provides the progress required to discharge (13).

Finitary data types in general can be specified using least fixpoint formulas. This allows for termination or eventuality properties of programs to be proven along the lines of the above example. In a similar way we can handle program properties that depend on inductive properties of message queues.

6 Proof Rules for Recursive Formulas

The approach we use to handle fixed points, and capture the critical notions of "regeneration", "progress", and "discharge" is, essentially, formalised well-founded induction. When some fixed points are unfolded, notably least fixed points to the left of the turnstile, and greatest fixed points to the right of the turnstile, it is possible to pin down suitable *approximation ordinals* providing, for least fixed points, a progress measure toward satisfaction and, for greatest fixed points, a progress measure toward refutation. We introduce explicit ordinal variables which are maintained, and suitably decremented, as proofs are elaborated. This provides a simple method for dealing with a variety of complications such as alternation of fixpoints and the various complications related to duplication and interference between fixed points that are dealt with using the much more indirect approach of [Dam98].

We first pin down some terminology concerning proofs. A *proof structure* is a finite, rooted, sequent-labelled tree which respects the proof rules in the sense that if π is a proof node labelled by the sequent δ, and if $\pi_1, \ldots, \pi_n$ are the children of π in left to right order labelled by $\delta_1, \ldots, \delta_n$ then

$$\frac{\delta_1 \quad \cdots \quad \delta_n}{\delta}$$

is a substitution instance of one of the proof rules. A node π is *elaborated* if its label is the conclusion of a rule instance as above. A *proof* is a proof structure for which all nodes are elaborated. In the context of a given proof structure we write $\pi_1 \longrightarrow \pi_2$ if π_2 is a child of π_1. A *path* is a finite sequence $\Pi = \pi_1, \ldots, \pi_n$ for which $\pi_i \longrightarrow \pi_{i+1}$ for all $i : 1 \leq i < n$. Generally we use the term "sequent occurrence" as synonymous with node. However, when the intention is clear from the context we sometimes confuse sequents with sequent occurrences and write eg. $\delta_1 \longrightarrow \delta_2$ in place of $\pi_1 \longrightarrow \pi_2$ when the sequents label their corresponding nodes.

Ordinal Approximations In general, soundness of fixed point induction relies on the well-known iterative characterisation where least and greatest fixed points are "computed" as iterative limits of their ordinal approximations. This also happens in the present case. Let κ range over ordinal variables. Valuations and substitutions are extended to map ordinal variables to ordinals. Let U, V range over fixed point formula abstractions of the form $\sigma Z(V_1, \ldots, V_n).\phi$. We introduce new formulas of the shape U^κ and $\kappa < \kappa'$. Ordinal inequalities have their obvious semantics, and $\kappa \leq \kappa'$ abbreviates $\kappa < \kappa' \vee \kappa = \kappa'$ as usual. For approximated fixed point abstractions suppose first that $U = \sigma Z(V_1, \ldots, V_k).\phi$ and $\sigma = \nu$. Then

$$\| U^\alpha(P_1, \ldots, P_k) \| \eta = \begin{cases} \mathcal{S}, & \textit{if } \alpha = 0 \\ \| \phi\{U^{\alpha'}/Z\} \| \eta\{P_1/V_1, \ldots, P_k/V_k\}, & \textit{if } \alpha = \alpha' + 1 \\ \bigcap\{\| U^{\alpha'}(P_1, \ldots, P_k) \| \eta \mid \alpha' < \alpha\}, & \textit{if } \alpha \textit{ limit ord.} \end{cases}$$

Dually, if $\sigma = \mu$:

$$\| U^\alpha(P_1, \ldots, P_k) \| \eta = \begin{cases} \emptyset, & \textit{if } \alpha = 0 \\ \| \phi\{U^{\alpha'}/Z\} \| \eta\{P_1/V_1, \ldots, P_k/V_k\}, & \textit{if } \alpha = \alpha' + 1 \\ \bigcup\{\| U^{\alpha'}(P_1, \ldots, P_k) \| \eta \mid \alpha' < \alpha\}, & \textit{if } \alpha \textit{ limit ord.} \end{cases}$$

We get the following basic monotonicity properties of ordinal approximations:

Proposition 1. *Suppose that $\alpha \leq \alpha'$.*

1. *If U is a greatest fixed point abstraction then*

$$\| U^{\alpha'}(P_1, \ldots, P_n) \| \eta \subseteq \| U^\alpha(P_1, \ldots, P_n) \| \eta$$

2. *If U is a least fixed point abstraction then*

$$\| U^\alpha(P_1, \ldots, P_n) \| \eta \subseteq \| U^{\alpha'}(P_1, \ldots, P_n) \| \eta$$

Proof. By wellfounded induction. □

Moreover, and most importantly, we get the following straightforward application of the well-known Knaster-Tarski fixed point theorem.

Theorem 2 (Knaster-Tarski). *Suppose that $U = \sigma Z(V_1, \ldots, V_k).\phi$. Then*

$$\| U(P_1, \ldots, P_n) \| \eta = \begin{cases} \cap\{\| U^\alpha(P_1, \ldots, P_n) \| \eta \mid \alpha \textit{ an ordinal}\}, & \textit{if } \sigma = \nu \\ \cup\{\| U^\alpha(P_1, \ldots, P_n) \| \eta \mid \alpha \textit{ an ordinal}\}, & \textit{if } \sigma = \mu \end{cases}$$

As the intended model is countable the quantification in theorem 2 can be restricted to countable ordinals.

The main rules to reason locally about fixed point formulas are the unfolding rules. These come in four flavours, according to whether the fixed point abstraction concerned has already been approximated or not, and to the nature and position of the fixed point relative to the turnstile.

$$\mathsf{ApprxL}\ \frac{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash \Delta}{\Gamma, s : U(P_1, \ldots, P_n) \vdash \Delta}\ U \text{ lfp},\ \kappa \text{ fresh}$$

$$\mathsf{ApprxR}\ \frac{\Gamma \vdash s : U^{\kappa}(P_1, \ldots, P_n), \Delta}{\Gamma \vdash s : U(P_1, \ldots, P_n), \Delta}\ U \text{ gfp},\ \kappa \text{ fresh}$$

$$\mathsf{UnfL1}\ \frac{\Gamma, s : \phi\{U/Z, P_1/V_1, \ldots, P_n/V_n\} \vdash \Delta}{\Gamma, s : U(P_1, \ldots, P_n) \vdash \Delta}\ U = \sigma Z(V_1, \ldots, V_n).\phi$$

$$\mathsf{UnfR1}\ \frac{\Gamma \vdash s : \phi\{U/Z, P_1/V_1, \ldots, P_n/V_n\}, \Delta}{\Gamma \vdash s : U(P_1, \ldots, P_n), \Delta}\ U = \sigma Z(V_1, \ldots, V_n).\phi$$

$$\mathsf{UnfL2}\ \frac{\Gamma, s : \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \kappa_1 < \kappa \vdash \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash \Delta}\ U \text{ lfp}, \kappa_1 \text{ fresh}$$

$$\mathsf{UnfR2}\ \frac{\Gamma, \kappa_1 < \kappa \vdash s : \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \Delta}{\Gamma \vdash s : U^{\kappa}(P_1, \ldots, P_n), \Delta}\ U \text{ gfp}, \kappa_1 \text{ fresh}$$

$$\mathsf{UnfL3}\ \frac{\Gamma, s : \kappa_1 < \kappa \supset \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\} \vdash \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash \Delta}\ U \text{ gfp}$$

$$\mathsf{UnfR3}\ \frac{\Gamma \vdash s : \kappa_1 < \kappa \wedge \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \Delta}{\Gamma \vdash s : U^{\kappa}(P_1, \ldots, P_n), \Delta}\ U \text{ lfp}$$

Normally we would expect only least fixed point formula abstractions to appear in approximated form to the left of the turnstile (and dually for greatest fixed points). However, ordinal variables can "migrate" from one side of the turnstile to the other through one of the cut rules. Consider for instance the following application of the process cut rule:

$$\frac{\Gamma \vdash s_2 : U^{\kappa} \quad \Gamma, S : U^{\kappa} \vdash s_1 : U^{\kappa}}{\Gamma \vdash s_1\{s_2/S\} : U^{\kappa}}$$

In this example U may be a greatest fixed point formula which, through some earlier application of ApprxR has been assigned the ordinal variable κ. The second antecedent has U^{κ} occurring to the left of the turnstile.

In addition to the above 8 rules it is useful also to add versions of the identity rules reflecting the monotonicity properties of ordinal approximations, prop. 1:

$$\mathsf{IdMon1}\ \frac{\Gamma \vdash \kappa \leq \kappa', \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash s : U^{\kappa'}(P_1, \ldots, P_n), \Delta}\ U \text{ lfp}$$

$$\text{IdMon2}\ \frac{\Gamma \vdash \kappa \geq \kappa', \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash s : U^{\kappa'}(P_1, \ldots, P_n), \Delta}\ U \text{ gfp}$$

Additionally a set of elementary rules are needed to support reasoning about well-orderings, including transitivity and irreflexivity of $<$. These rules are left out of the presentation.

For the above rules we obtain the following basic soundness result:

Theorem 3. *The rules* ApprxL*,* ApprxR*,* UnfL1*,* UnfR1*,* UnfL2 *and* UnfR2 *are sound.*

Proof. Rules ApprxL and ApprxR. For ApprxL assume $\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \models_{K,\kappa} \Delta$, and that κ is fresh. Assume also that Γ and $s : U(P_1, \ldots, P_n)$ holds, up to some valuation. Then, for this valuation, so does $s : U^{\alpha}(P_1, \ldots, P_n)$ for some ordinal α. But then we find that some assertion in Δ is true as well, completing the case. For ApprxR the dual argument applies.
Rules UnfL1 and UnfL2. The soundness of these rules follows directly from the fact that $\sigma Z(V_1, \ldots, V_n).\phi$ is a parametrised fixed point of ϕ.
Rules UnfL2 and UnfR2. We consider UnfL2. Assume that

$$\Gamma, s : \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \kappa_1 < \kappa \models \Delta.$$

Assume also that U is a least fixed point abstraction, and that κ_1 is fresh. Assume furthermore that a valuation is given, making Γ and $U^{\alpha}(P_1, \ldots, P_n)$ true. Either α is 0, or $\alpha = \alpha_1 + 1$, or α is a limit ordinal. The first case is contradictory. For the second case we get the κ_1 we are looking for directly, and some assertion in Δ is established as desired. For the third case we find some $\alpha_1' < \alpha$ such that $U^{\alpha_1'}(P_1, \ldots, P_n)$ is true. We can assume that α_1' is a successor ordinal. But then the previous subcase applies, and we are done. Again UnfR2 is proved by a symmetric argument.
Rules UnfL3 and UnfR3. We consider UnfL3. Assume that

$$\Gamma, s : \kappa_1 < \kappa \supset \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\} \models \Delta.$$

Assume also that a valuation is given such that Γ and $s : U^{\alpha}(P_1, \ldots, P_n)$ is true. Then whenever $\alpha_1 < \alpha$, $s : \phi\{U^{\alpha_1}/Z, P_1/V_1, \ldots, P_n/V_n\}$ is true as well. If $\alpha = 0$ this is trivially so. If α is a successor ordinal it follows by prop. 1, and if α is a limit ordinal we know that whenever $\alpha_1' < \alpha$ then $s : U^{\alpha_1'+1}(P_1, \ldots, P_n)$, so $s : \phi\{U^{\alpha_1}/Z, P_1/V_1, \ldots, P_n/V_n\}$. In any case we can conclude that some assertion in Δ must be true, finishing the argument. Again UnfR3 is symmetric.
Rules IdMon1 and IdMon2 are trivial, given 1. □

Discharge: Some Intuition The fundamental problem in arriving at a sound, yet powerful, rule of discharge, is to control the way fixed points may interfere as proofs are elaborated. We illustrate the problem by two examples.

Example 3. Consider the proof goal

$$S : \nu Z_1.\mu Z_2.[\,]Z_1 \wedge \forall P, V.[P!V]Z_2 \ \vdash\ S : \mu Z_3.\nu Z_4.[\,]Z_4 \wedge \forall P, V.[P!V]Z_3 \quad (14)$$

The assumption states that any infinite sequence of internal or send transitions can only contain a finite number of consecutive send transitions, while the assertion states that any infinite sequence of internal or send transitions can only contain a finite number of send transitions. Thus (14) is false.

Let us introduce the following abbreviations:

$$\begin{aligned}
U_1 &= \nu Z_1.\mu Z_2.[\,]Z_1 \wedge \forall P, V.[P!V]Z_2 \\
U_2 &= \mu Z_2.[\,]U_1 \wedge \forall P, V.[P!V]Z_2 \\
U_3 &= \mu Z_3.\nu Z_4.[\,]Z_4 \wedge \forall P, V.[P!V]Z_3 \\
U_4 &= \nu Z_4.[\,]Z_4 \wedge \forall P, V.[P!V]U_3
\end{aligned}$$

We start by refining (14) to the subgoal

$$S : U_2^{\kappa_2} \vdash S : U_4^{\kappa_4} \tag{15}$$

using the rules UnfL1, UnfR1, ApprxL and ApprxR. Continuing a few steps further (by unfolding the fixed point formulas and treating the conjunctions on the left and on the right) we obtain the two subgoals

$$S : [\,]U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S : [\,]U_4^{\kappa_4'} \tag{16}$$

$$S : [\,]U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S : \forall P, V.[P!V]U_3 \tag{17}$$

Subgoal 16 is refined via rule Mon2 to

$$S' : U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S' : U_4^{\kappa_4'} \tag{18}$$

and after unfolding U_1 using UnfL1 we arrive at

$$S' : U_2, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S' : U_4^{\kappa_4'} \tag{19}$$

which sequent one might expect to be able to discharge against (15) by coinduction in κ_4. By the same token when we refine (17) to

$$S : [\,]U_1, S' : U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S' : U_4 \tag{20}$$

we would expect to be able to discharge against (15) inductively in κ_2. This does not work, however, since derivation of (19) from (15) fails to preserve the induction variable κ_2 needed for (20), and vice versa, κ_4 is not preserved along the path from (15) to (20). Therefore, the infinite proof structure resulting from an infinite repetition of the above steps contains paths in which neither of the two variables is actually being preserved and decremented infinitely many times, and hence the attempted ordinal induction fails. It would still have been sound to discharge if at least one of the two ordinal variables had been preserved in the corresponding other branch; then there would have been no such paths.

Example 4. Consider the (reversed) proof goal

$$S : \mu Z_1.\nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]Z_1 \;\vdash\; S : \nu Z_3.\mu Z_4.[\,]Z_3 \wedge \forall P, V.[P!V]Z_4 \quad (21)$$

stating that if all infinite sequences of internal or send transitions of a process can only contain a finite number of send transitions, then these infinite sequences of internal or send transitions can only contain finite sequences of consecutive send transitions. This goal is obviously valid.

The abbreviations we shall use are:

$$\begin{aligned}
U_1 &= \mu Z_1.\nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]Z_1 \\
U_2 &= \nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]U_1^{\kappa_1'} \\
U_3 &= \nu Z_3.\mu Z_4.[\,]Z_3 \wedge \forall P, V.[P!V]Z_4 \\
U_4 &= \mu Z_4.[\,]U_3^{\kappa_3'} \wedge \forall P, V.[P!V]Z_4
\end{aligned}$$

First we apply rules ApprxL, ApprxR, UnfL2 and UnfR2 to reduce (21) to the subgoal

$$S : U_2, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \;\vdash\; S : U_4 \quad (22)$$

Continuing in much the same way as in the preceding example we arrive at the two subgoals

$$S' : U_2, S : \forall P, V.[P!V]U_1^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \;\vdash\; S' : U_3^{\kappa_3'} \quad (23)$$

$$S : [\,]U_2, S' : U_1^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \;\vdash\; S' : U_4 \quad (24)$$

These subgoals are refined, using UnfR2 and UnfL2 respectively, to

$$\begin{aligned}
&S' : U_2, S : \forall P, V.[P!V]U_1^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3, \kappa_3'' < \kappa_3' \\
&\vdash S' : \mu Z_4.[\,]U_3^{\kappa_3''} \wedge \forall P, V.[P!V]Z_4 \quad (25) \\
&S : [\,]U_2, S' : \nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]U_1^{\kappa_1''}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3, \kappa_1'' < \kappa_1' \\
&\vdash S' : U_4 \quad (26)
\end{aligned}$$

These sequents can be discharged against (22) inductively in κ_3, and coinductively in κ_1, respectively. In contrast with the previous example, here every ordinal variable which is used for induction (or coinduction) in one of the two leaves is preserved throughout the path to the other leaf.

The Rule of Discharge We now arrive at the formal definition of the rule of discharge.

Convention 7. From this point onwards proof elaboration takes place in the context of some fixed, but arbitrary linear ordering $<$ on fixed point formula abstractions U.

Assuming one fixed linear ordering can be too restrictive when recursive proof structures are independent. Below we briefly discuss ways of relaxing the construction to allow the linear ordering to be built incrementally.

Below we define the critical notions of regeneration, progress, and discharge. Discharge is applied when facing a proof goal π_n which is unelaborated, such that, below π_n we find some already elaborated node π_1 such that π_n is in a sense an instance of π_1. This requires variables present in π_1 to be interpreted as terms in π_n. This is what the substitution ρ of the following definition serves to achieve.

Definition 8 (Regeneration, Progress, Discharge). *Let $\Pi = \pi_1, \ldots, \pi_n$ be a path such that π_n is not elaborated. Suppose that π_i is labelled by $\Gamma_i \vdash \Delta_i$ for all $i : 1 \leq i \leq n$.*

1. *The path Π is* regenerative *for U and the substitution ρ, if whenever there is a κ_i such that U^{κ_i} is a subformula of Γ_i (Δ_i) then there also are κ_1, …, κ_{i-1}, κ_{i+1}, …, κ_n such that for all $j : 1 < j \leq n$, U^{κ_j} is a subformula of Γ_j (Δ_j), and $\Gamma_j \vdash \kappa_j \leq \kappa_{j-1}$. Moreover we require that $\rho(\kappa_1) = \kappa_n$.*
2. *The path Π is* progressive *for U and ρ if we can find $\kappa_1, \ldots, \kappa_n$ such that:*
   (a) *For all $i : 1 < i \leq n$, U^{κ_i} is a subformula of Γ_i (Δ_i), and $\Gamma_i \vdash \kappa_i \leq \kappa_{i-1}$.*
   (b) *$\rho(\kappa_1) = \kappa_n$.*
   (c) *For some $i : 1 < i \leq n$, $\Gamma_i \vdash \kappa_i < \kappa_{i-1}$.*
3. *The node π_n can be* discharged *against the node π_1 if we can find some U and substitution ρ such that:*
   (a) *Π is regenerative for all $U' < U$ and ρ.*
   (b) *Π is progressive for U and ρ.*
   (c) *For all assumptions $s : \phi$ in Γ_1, $\Gamma_n \vdash s\rho : \phi\rho$, and all assertions $s : \phi$ in Δ_1 then $s\rho : \phi\rho \vdash \Delta_n$.*

   In this case we term π_n a discharge node *and π_1 its* companion node.

In this definition we are being slightly sloppy with our use of U's: Really we are identifying fixed point formula abstractions up to ordinal approximations except where they are explicitly stated.

It is quite easy to verify that for Example 3 no linearisation of the fixed point formulas can be devised such that the nodes (18) and (19) can be discharged. On the other hand, for Example 4, any linear ordering which (up to approximation ordinals) has $U_4 < U_2$ will do.

Observe that the linear ordering on fixed point formula abstractions can be chosen quite freely. One might expect some correlation between position in the linear ordering and depth of alternation, viz. example 4 above. In practice this is in fact a good guide to choosing a suitable linear ordering. However, as we show, we do not need to require such a correlation a priori. Moreover one can construct examples, using cut's, of proofs for which the above rule of thumb does not work.

Now, the *full proof system* is obtained by adding the proof rules for fixed points, including the rule of discharge, to the local rules of section 4.

Theorem 4 (Soundness, Recursive Formulas). *The full proof system is sound.*

Proof. The proof is by induction on the size of proof trees. Assume a proof of root sequent $\Gamma_0 \vdash \Delta_0$, and assume soundness for all proof trees of a strictly smaller size. Assume for a contradiction that $\Gamma_0 \not\models \Delta_0$. We then find a valuation η_0 which invalidates $\Gamma_0 \not\models \Delta_0$, i.e. which makes all assumptions in Γ_0 valid, and all assertions in Δ_0 invalid. We use this assumption to construct an infinite *rejection sequence* Σ of sequent-valuation pairs $(\Gamma_0 \vdash \Delta_0, \eta_0)(\Gamma_1 \vdash \Delta_1, \eta_1)\cdots$ such that for all i, η_i invalidates $\Gamma_i \vdash \Delta_i$, and, considering nodes of discharge to be followed by their respective companion nodes, the sequent sequence $(\Gamma_0 \vdash \Delta_0)(\Gamma_1 \vdash \Delta_1)\cdots$ forms a run through the proof tree. Subsequently we use this sequence to derive a contradiction.

The sequence Σ is constructed inductively. Assuming that the construction has reached the i'th element we show how to construct the $(i+1)$'th element depending on the rule by which $\Gamma_i \vdash \Delta_i$ was elaborated in the proof. For all rules except discharge the construction is trivial, by the "local" soundness results, Theorems 1 and 3. So assume that $\Gamma_i \vdash \Delta_i$ was discharged against $\Gamma_i' \vdash \Delta_i'$ using substitution ρ as specified by the definition of the discharge rule. We then define the $(i+1)$'st element of the sequence as $(\Gamma_i' \vdash \Delta_i', \eta_i \circ \rho)$ and show that $\eta_{i+1} = \eta_i \circ \rho$ invalidates $\Gamma_i' \vdash \Delta_i'$. Let $s : \phi$ be any assertion in Γ_i'. By the induction hypothesis and condition 8.3.(c) we see that since all assumptions in Γ_i are validated under η_i then so is $s : \phi$ under η_{i+1}. Secondly let $s : \phi$ be any assertion in Δ_i'. We need to show that $s\eta_{i+1} : \phi\eta_{i+1}$ is false. But if it were not, by the induction hypothesis and condition 8.3.(c) we would obtain some assertion in Δ_i which is valid under η_i, and this is an impossibility since η_i invalidates $\Gamma_i \vdash \Delta_i$. The construction is thus complete.

Infinitely often along Σ the discharge rule is applied. The proof being finite, the number of distinct fixed point abstractions that can appear in the proof is finite too. As a consequence we must be able to find a smallest U under $<$ which is appealed to infinitely often (in 8.3) in applications of discharge along Σ. Let i be such that $\Gamma_i \vdash \Delta_i$ is elaborated infinitely often through the rule of discharge by appealing to U, and that, for no $j \geq i$, is $\Gamma_j \vdash \Delta_j$ discharged with reference to a U' which is strictly smaller than U. For some κ_i we find an occurrence of U^{κ_i} in the corresponding sequent $\Gamma_i \vdash \Delta_i$, say that U^{κ_i} is a subformula of Γ_i. We then see that for each $j > i$ we can find an ordinal variable κ_j such that U^{κ_j} occurs as a subformula of Γ_j. We shall sketch an argument that the subformulas U^{κ_j} can be chosen so that the values assigned to κ_j by η_j form a sequence which is non-increasing and in fact infinitely often decreasing. But this is not possible, since ordinals are well-founded, and we hence shall arrive at a contradiction.

Consider an arbitrary interval $\Sigma(j_1, j_m)$ of Σ such that the first sequent $\Gamma_{j_1} \vdash \Delta_{j_1}$ is equal to the last $\Gamma_{j_m} \vdash \Delta_{j_m}$ and does not occur inbetween. Then there must be an element in the interval whose sequent is a discharge node, and whose companion node is either $\Gamma_{j_m} \vdash \Delta_{j_m}$ or is some sequent higher in the proof tree (i.e. closer to the root sequent). We shall call the earliest such element the *characterising element* of the interval. The interval itself might contain other intervals of the same shape. Moreover, one can choose these intervals in such a way that the elements not occurring in any of the intervals, the characterizing element being among them, form a simple run (i.e. a run not visiting any sequent more than once) through the loop defined by the discharge path for the characterising sequent. Given an initial partitioning of Σ into such intervals, one can iteratively apply this decomposition scheme until no interval can be further decomposed. Given an index j, we shall call the *active* interval the least interval of the above type containing both the j'th and the $j+1$'th element of Σ.

We perform an initial partitioning of Σ in intervals $\Sigma(j_1, j_m)$ so that $\Gamma_{j_1} \vdash \Delta_{j_1}$ is the companion node of $\Gamma_i \vdash \Delta_i$, and continue the decomposition process as described above. Starting from index $j = i$, we shall choose κ_j according to the discharge condition for the path having as a discharge node the sequent from the characterising element of the active interval. If the current interval is characterised by $\Gamma_i \vdash \Delta_i$ (which can only happen in outermost intervals) we choose κ_j as for progression (cf. 8.2), in all other cases we choose κ_j for regeneration (cf. 8.1). The discharge condition and the induction hypothesis guarantee that the values assigned to κ_j by η_j form a sequence which is non-increasing (in regenerative intervals) and infinitely often decreasing (in progressive intervals), thus yielding a contradiction.

So, no such rejection sequence Σ can exist, and the assumption $\Gamma_0 \not\models \Delta_0$ must have been false.

$\square$

7 Verifying the Resource Manager

In this section the proof system is demonstrated by outlining a proof that the resource manager function introduced in section 2 satisfies the *safe* specification defined in section 3. The proof will be kept informal. For instance we will write out neither ordinal variables nor the linear ordering on fixed point formula abstractions, since they can easily be added to the proof. Adding ordinal annotations to the proof and taking them into account presents no real difficulty since the fixed point definitions in the example are flat, i.e., they never refer to other fixed point definitions.

For simplicity it is assumed that the manager knows of only one resource, with public name P_u and private P_r. The corresponding list $[\{P_u, P_r\}]$ is referred to as R_L, and R_P denotes the process identifier of the resource manager process.

Since the definition of *safe* is parametrised on a billing agent and a user account the formula must be preceded by an initialisation phase (notice the use of the weak modality $[[\alpha]]$ introduced in section 3):

$$\begin{array}{l} \forall PubRes, UAcc, UserPid, Agent. \\ \quad [R_P?\{contract, \{PubRes, UAcc\}, UserPid\}] \\ \quad\quad [[UserPid!\{contract\_ok, Agent\}]]safe(Agent, BankPid, UAcc, 0) \end{array}$$

So we set out to prove the following sequent:

$$\Gamma \vdash \langle rm(R_L, BankPid, RAcc), R_P, \epsilon \rangle$$

$$: \forall PubRes, UAcc, UserPid, Agent.\ [R_P?\{contract, \{PubRes, UAcc\}, UserPid\}] \ldots \tag{27}$$

The needed inequations on process identifiers (e.g., $R_P \neq P_r$) are collected in Γ. By application of simple proof steps – four applications of AllR and then repeated applications of the rules for unfolding, elimination of conjunctions, and the rule for the box modality – the following proof state is reached:

$$\begin{aligned} \Gamma' \vdash\ & \langle rm(R_L, BankPid, RAcc), R_P, \epsilon\rangle \\ & || \langle billagent(P_r, BankPid, RAcc, UAcc), B_P, \epsilon\rangle \\ & : safe(B_P, BankPid, UAcc, 0) \end{aligned} \tag{28}$$

where Γ' is Γ extended with the fact that B_P is a fresh process identifier. This is a critical proof state, where we must come up with properties of the resource manager and the billing agent, that are sufficiently strong to prove that their parallel composition satisfies the *safe* property. In general such a proof step may be very difficult, but here the choice is relatively simple:

ϕ_a: The billing agent satisfies the *safe* property, i.e., $safe(B_P, BankPid, UAcc, 0)$.
ϕ_b: The billing agent communicates the user account to no process except the bank, unless some other process first sends it the account.
ϕ_c: The resource manager does not communicate the user account, unless some other process first sends it the account. This property can be formulated as $notrans(UAcc)$ given the definition of the *notrans* property at the end of section 3.
ϕ_d: The resource manager does not send a tuple containing the atom *use* in the first position (a usage request to a billing agent).
ϕ_e: The resource manager cannot receive messages sent to the bank process, nor can it receive messages sent to the billing agent.

Properties ϕ_b, ϕ_d and ϕ_e can easily be formulated in a manner similar to ϕ_c. Essentially these conditions guarantee that bank transfers are the result of user requests, rather than incorrectly programmed billing agents or resource managers that exchange information with each other.

The result of applying the ProcCut rule twice, after generalising the proof goals, is the following proof obligations:

$$\begin{aligned} \Gamma', \mathtt{not}(contains(B_Q, UAcc)), countuse(B_Q, M), M \leq N \vdash \\ \langle billagent(P_r, BankPid, RAcc, UAcc), B_P, B_Q\rangle \\ : safe(B_P, BankPid, UAcc, N) \wedge \phi_b \end{aligned} \tag{29}$$

$$\begin{aligned} \Gamma', \mathtt{not}(contains(R_Q, UAcc)) \vdash \langle rm(R_L, BankPid, RAcc), R_P, R_Q\rangle \\ : notrans(UAcc) \wedge \phi_d \wedge \phi_e \end{aligned} \tag{30}$$

$$\begin{aligned} \Gamma', S_1 : safe(B_P, BankPid, UAcc, N) \wedge \phi_b, S_2 : notrans(UAcc) \wedge \phi_d \wedge \phi_e \\ : S_1 || S_2 : safe(B_P, BankPid, UAcc, N) \end{aligned} \tag{31}$$

To prove the leftmost conjunct in the goal (29) one has to show that the number of valid usage requests in the input queue (the parameter M in $countuse(B_Q, M)$) is always less than or equal to the number of transfer requests that are possible (the parameter N). This proof involves well-known techniques for proving correctness of sequential programs, and the proof of ϕ_b is even less involved (proofs omitted). Instead we concentrate on the leftmost conjunct of (30), i.e., that rm satisfies $notrans(UAcc)$ as long as no element in its input queue contains $UAcc$. The proofs of properties ϕ_d and ϕ_e follow the same pattern (details omitted). To prove (30) we first unfold the definition of $notrans$ and eliminate the conjunctions. In case of an input step $[V?V']$ either we are done immediately (if $contains(V', UAcc)$). Otherwise the resulting proof state is

$$\Gamma', \texttt{not}(contains(R_Q, UAcc)), \texttt{not}(contains(V', UAcc)) \vdash \\ \langle rm(R_L, BankPid, RAcc), R_P, R_Q \cdot V' \rangle : notrans(UAcc) \tag{32}$$

which can be rewritten into (by referring to the definition of $contains$)

$$\Gamma', \texttt{not}(contains(R_Q \cdot V', UAcc)) \vdash \\ \langle rm(R_L, BankPid, RAcc), R_P, R_Q \cdot V' \rangle : notrans(UAcc) \tag{33}$$

which can be discharged against the leftmost conjunct of (30). The rm process can clearly not perform any output step so that part of the conjunction is trivially true. Thus only the internal step remains, and such a step must correspond to unfolding the application $rm(R_L, BankPid, RAcc)$. The resulting proof state is:

$$\Gamma', \texttt{not}(contains(R_Q, UAcc)) \vdash \\ \langle \texttt{case}\ \{R_L, BankPid, RAcc\}\ \texttt{of}\ \ldots, R_P, R_Q \rangle : notrans(UAcc) \tag{34}$$

By repeating the above steps, i.e., handling input, output and internal steps eventually one reaches the goal:

$$\begin{aligned}&\Gamma'', \texttt{not}(contains(R_Q{}', UAcc)) \vdash \\ &\big\langle UserPid!\{contract\_ok, B_p'\}, rm(R_L, BankPid, RAcc) \ldots, R_P, R_Q{}' \big\rangle \\ &\quad \| \big\langle billagent(P_r, BankPid, RAcc, UAcc'), B_P{}', \epsilon \big\rangle \\ &: \ notrans(UAcc)\end{aligned} \tag{35}$$

where Γ'' is Γ' together with inequations involving the fresh process identifier $B_P{}'$, and the fact that $UAcc' \neq UAcc$. This goal is handled by applying ProcCut to the parallel composition using $notrans(UAcc)$ as the cut formula both to the left and to the right. The resulting goals are:

$$\begin{aligned}&\Gamma'', \texttt{not}(contains(R_Q{}', UAcc)) \vdash \\ &\big\langle UserPid!\{contract\_ok, B_P{}'\}, rm(R_L, BankPid, RAcc) \ldots, R_P, R_Q{}' \big\rangle \\ &\quad : notrans(UAcc)\end{aligned} \tag{36}$$

$$\Gamma'' \vdash \big\langle billagent(P_r, BankPid, RAcc, UAcc'), B_P{}', \epsilon \big\rangle : notrans(UAcc) \tag{37}$$

$$\Gamma'', S_3 : notrans(UAcc), S_4 : notrans(UAcc) \vdash S_3 \| S_4 : notrans(UAcc) \tag{38}$$

Goal (37) is easy to prove, since no new processes are created (proof sketch omitted). For goal (36) we have to show $\mathtt{not}(\mathit{contains}(\{\mathit{contract\_ok}, B_p'\}, \mathit{UAcc}))$, since this is the value the resource manager will send to pid *UserPid*. The property is clearly true since B_P' is a fresh pid. The resulting goal, after a simple step where the resulting sequence is reduced, becomes

$$\begin{aligned} \Gamma'', \mathtt{not}(\mathit{contains}({R_Q}', \mathit{UAcc})) \vdash \langle & \mathit{rm}(R_L, \mathit{BankPid}, \mathit{RAcc}), R_P, {R_Q}' \rangle \\ & : \mathit{notrans}(\mathit{UAcc}) \end{aligned} \tag{39}$$

This goal can be discharged against the leftmost conjunct of (30). Thus only goals (31) and (38) remain. These types of goals are handled in a uniform and regular way, using applications of BoxPar1 and BoxPar2, repeated use of boolean reasoning, the UnfR and UnfL rules, and discharging against previously seen goals (details omitted).

8 Concluding Remarks

We have introduced a specification logic and proof system for the verification of programs in a core fragment of Erlang, and illustrated its application on a small, but quite delicate, agent-based example. Our approach is quite general both regarding the kinds of languages and models that can be addressed, and the kinds of assertions that can be formulated. For instance we are not restricted, as in many other approaches to compositional verification, to linear-time logic, neither does the proof system rely on auxiliary features like history or prophecy variables. In addition our approach permits the treatment of programming language constructs such as dynamic process creation, non-tail recursion and inductive data type definitions in a uniform way, via a powerful rule of discharge.

An important feature of our approach is the use of fixed points to describe recursively the fine structure of computation trees, and to use these recursive descriptions to decompose properties according to system structure. No fixed vocabulary of temporal connectives such as those of LTL, CTL, or CTL* would permit a similarly general decomposition. The proof-theoretical setting given here represents a substantial advance on the initial work for CCS reported in [Dam98]. That work suffered from a number of shortcomings which we think have now been resolved in a satisfactory manner. This concerns:

1. The account of discharge in [Dam98] used an indirect approach, tracking and indexing fixed point unfoldings in a very syntactical and opaque manner. The present approach, using explicit ordinal annotations, is arguably far simpler, more intuitive, and semantically clearer.
2. The sequent format used in [Dam98] was more restrictive than the one used here, in effect preventing contraction, affecting proof power very severely, theoretically as well as in practice.
3. The discharge condition of [Dam98] required much more rigid relationships between the structure of discharged nodes and the internal nodes motivating their discharge. In effect it was required that all information be completely

cyclic in a pointwise manner. But many examples are extremely cumbersome, if not outright impossible, to force into such a framework.

The drawback, if any, of the approach used here is the explicit use of ordinals. However, in an implementation of this proof system, users need rarely, if ever, be directly exposed to ordinals. Ordinal annotations can be automatically synthesised, and only in very special circumstances do we envisage ordinal information being passed explicitly to users for proof debugging.

Several important lessons were learned in the process of doing proofs like the billing agent example. We have already mentioned the need for more flexible sequent formats and discharge conditions. Practical proofs tend to get very large. Without support for reducing duplication of proof nodes the proof example outlined for the billing agent has in the range of 10^5–10^6 proof tree nodes. Just by avoiding proof node duplication this figure can be brought down very substantially, for the billing agent example by roughly a factor of 15. But in fact very few steps in the proof convey information which is really interesting. These are:

1. Points where a process cut need to be applied, to initiate induction in system state structure.
2. Points at which some other symbolic or inductive argument needs to be done, to handle e.g. induction in the message queue structure.
3. Choice points which we may want to return to later, for backtracking.
4. Points which we expect to want to discharge against in the future.

One can easily envisage other proof elaboration steps being automated, and eliminated from view to a very large extent, perhaps using a selection of problem-dependent proof tactics. However, it is important to realise that, in contrast to mainstream proof editors such as HOL or PVS, in this some explicit support for managing proof node histories is essential for efficiency.

To investigate these issues, and to begin doing real application studies, we are currently building a prototype proof checking tool that can handle programs of a moderate size such as the billing agent example. Some support for automation of proof steps along the above lines already exists (e.g. for some model checking analyses), but we also need to identify other classes of sequents that can be solved algorithmically. Other ongoing work focuses on integrating the operational semantics of Erlang more tightly with the proof systems (along the lines of [Sim95]) and to improve the handling of process identifier scoping (but see [AD96] for an approach to this in the context of the π-calculus).

Acknowledgements Many thanks are due to Fredrik Orava of the department of Teleinformatics at the Royal Institute of Technology, and Thomas Arts, Tony Rogvall and Dan Sahlin of Ericsson Telecom Computer Science Laboratory.

References

[AD96] R. Amadio and M. Dam. A modal theory of types for the π-calculus. In *Proc. FTRTFT'96,* Lecture Notes in Computer Science, 1135:347–365, 1996.

[AMST97] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Functional Programming*, 7:1–72, 1997.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.

[Dam98] M. Dam. Proving properties of dynamic process networks. To appear, *Information and Computation*, 1998. Preliminary version as "Compositional Proof Systems for Model Checking Infinite State Processes", Proc. CONCUR'95, LNCS 962, pp. 12–26.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[Par76] D. Park. Finiteness is mu-Ineffable. *Theoretical Computer Science*, **3**:173–181, 1976.

[Sim95] A. Simpson. Compositionality via cut-elimination: Hennessy-Milner logic for an arbitrary GSOS. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 420–430, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.