A Framework for Reasoning about ERLANG Code

Lars-Åke Fredlund

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfillment of the requirements for
the Degree of Doctor of Philosophy

August 2001

Department of Microelectronics
and Information Technology
The Royal Institute of Technology

KTH Electrum 229
SE-16440 Kista, Sweden

TRITA-IT AVH 01:04
ISSN 1403-5286
ISRN KTH/IT/AVH-01/04--SE

Swedish Institute
of Computer Science

Box 1263
SE-164 29 Kista, Sweden

SICS Dissertation Series 29
ISSN 1101-1335
ISRN SICS-D--29--SE

Dissertation for the Degree of Doctor of Philosophy
presented at the Royal Institute of Technology in 2001.

ABSTRACT

Fredlund, L.-Å. 2001: A Framework for Reasoning about ERLANG Code. TRITA-IT AVH 01:04, Department of Microelectronics and Information Technology, Stockholm. ISSN 1403-5286.

We present a framework for formal reasoning about the behaviour of software written in ERLANG, a functional programming language with prominent support for process based concurrency, message passing communication and distribution. The framework contains the following key ingredients: a specification language based on the μ-calculus and first-order predicate logic, a hierarchical small-step structural operational semantics of ERLANG, a judgement format allowing parametrised behavioural assertions, and a Gentzen style proof system for proving validity of such assertions. The proof system supports property decomposition through a cut rule and handles program recursion through well-founded induction. An implementation is available in the form of a proof assistant tool for checking the correctness of proof steps. The tool offers support for automatic proof discovery through higher–level rules tailored to ERLANG. As illustrated in several case studies this framework provides the expressive power required by the open and dynamic nature of distributed systems.

Lars-Åke Fredlund, Department of Microelectronics and Information Technology,
Royal Institute of Technology, KTH Electrum 229, SE-16440 Kista, Sweden,
E-mail: `fred@sics.se`

Acknowledgements

A thesis represents work conducted during a significant period of time – in my case far too long. As a result many people have helped and inspired me over the years, unfortunately too many to mention all by name.

First of all I would like to thank my advisor, Professor Joachim Parrow, who has always provided insightful comments and support, and most importantly has never given up on me.

To my second advisor, Professor Mads Dam, I would also like to extend my warm thanks. Apart from his role as advisor he is also the manager of the group at the Swedish Institute of Computer Science (SICS) where I work, and the chief source of ideas for the whole body of work documented in the thesis. A warm thanks is due also to Dr. Dilian Gurov, my colleague at SICS, who is the co-author of many of the papers this thesis is based on, and with whom many of the issues reported here have been resolved during long discussions.

The effort on the verification of Erlang programs, which this thesis is part of, has taken place within the ErlVer project which has received funding from the ASTEC (Advanced Software Technology) competence center and the Computer Science Laboratory of Ericsson. A number of colleagues, including the ones mentioned above, have made significant contributions to the project: Gennady Chugunov has designed and implemented the graphical user interface of the proof assistant, and has contributed many improvements to the tool itself. Dr. Thomas Arts at Ericsson has been a constant source of constructive ideas and improvements, and has provided much assistance in bridging the gap between academia and industry. Dr. Thomas Noll, now at RWTH Aachen, has been the co-author of several papers. Clara Benac Earle used an early version of the tool, and contributed many suggestions for improvements.

Further, I would like to recognise the important role Professor Bengt Jonsson has had throughout my studies. Together with Joachim Parrow he convinced me to begin my graduate studies, co-authored a number of early papers, and even after leaving SICS for Uppsala University he has always been a source of inspiration.

I would also like to mention a number of colleagues from the "early days" at SICS, with whom I collaborated on various projects: Patrik Ernberg, Hans Hansson, Fredrik Orava and Björn Victor. Later on Jan Cederquist, Pablo Giambiagi, Andrés Martinelli, and Babak Sadighi, and others, have contributed to making SICS a great place to work. Much of the credit for this atmosphere belongs also to Marianne Rosenqvist, without whom the place would have been so much duller. Thanks are due also to Christoph Sprenger and Vicki Carleson at SICS, and Jan Nyström from Uppsala University, for their help in proof reading parts of the thesis. A visit to CWI in Amsterdam organised by Frits Vaandrager provided my first exposure to theorem proving tools, and resulted in a paper co-authored with Jan Friso Groote and Henri Korver.

Heartfelt thanks goes out to my family for all the warmth and support they have provided over the years: to my father Åke and mother Greta, to my brothers Arne and Gunnar with their families: Anne, Kasper, Magnus, Meit, Andreas, Sofie.

Lastly, I would also like to acknowledge the contribution from the friends who have prompted me – in truth far too often – to sometimes step back from work in order to relax and to find joy and happiness in life.

I owe you all very much.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The production of software that functions correctly remains a truly challenging task even after at least fifty years of development in the fields of computer science and software engineering. Software projects routinely go astray, running up huge costs in terms of money and time spent and opportunity lost. The obvious question is,

> *Are there methods that can significantly improve the process of software development?*

The field of research that is known as formal methods has resulted in a collection of techniques to make the meaning of software artifacts mathematically and logically precise. There are for instance techniques which, from a usually informally worded design document, extract in mathematical logic the formal requirements that the document expresses. Often, but far from always, these formalised descriptions are analysed, sometimes with the assistance of computer support, to determine in advance any bad consequences of their deployment. An example could be checking that the specification of a safety-critical subsystem for railway traffic signalling never permits multiple trains to enter the same piece of track at the same time. In the end, though, what matters is not an idealised design specification but the properties of the software the programmers have actually implemented. In this thesis we consider the task of reasoning formally, using computer support, about computer programs written in the ERLANG programming language. This language is used at the Ericsson corporation to program a range of demanding distributed applications. However, the scope of our results clearly extend beyond this particular programming language.

Advocates of formal methods have for a long time liked to draw an analogy between software development and bridge construction. In the beginning of time, they say, bridges used to collapse, because there was no systematic knowledge how to construct them. Now that there is a proper engineering discipline of bridge building and maintenance, they simply do not fail anymore [1]. Thus our task as software engineers is

[1] Although there are by now many counter examples to this claim, e.g., the collapse of the river Douro bridge in Portugal, the stability problems of the Millennium bridge in London, or for that matter the safety concerns and cost overruns that have plagued the current "Tranebergsbron" bridge project here in Stockholm.

to go ahead and systematise an engineering discipline of programming.

And so there has been, since the 1960's at least, a strong computer science programme with this goal. Many brilliant people have spent tremendous efforts in furthering the field of program reasoning, in order to actually prove logically that programs work before they are put in use. Regrettably this programme has had little direct impact on software engineering, although there exist a number of successful applications; see Clarke and Wing [CW96] for an overview of the field. One cause for this lack of impact, which lies in the general area of research that this thesis addresses, is that checking whether a piece of software enjoys certain properties is fundamentally hard. In my opinion it remains far more difficult to prove that a program possesses certain desirable characteristics than to develop a program that with a high probability has these characteristics.

A second reason for the lack of impact is that the market accepts minor failures in software products like web browsers or word processors. It seems far more important to be able to quickly add new features to a product, or to develop a correction for a blatant software bug, than to reduce the number of defects to close to zero at the time of software release.

Many of the advocates of formal methods have also overestimated the maturity of the average software development project when calculating where the application of formal methods can provide the most benefit. For instance, to analyse specifications rather than programs is a risky activity since, as evidenced in countless large projects, after a short time any specification is unlikely to correctly reflect the implementation because of the prohibitive cost of revising it as the implementation constantly changes.

Still there is every reason not to lose faith in formal methods. We believe that trends in software engineering such as the increased reliance on software building blocks (components) will contribute to a renaissance. Given a formal model of a set of building blocks, and the ways in which these blocks can be composed, it should be possible to significantly decrease the effort required to verify an application. In addition there will always remain the systems where the cost of failure is simply too high not to warrant a very careful analysis. This can be due to the risk of loss of life (typically in the health, transport or power sectors), or when the potential for monetary gain motivates systematic attacks on security schemes and implementations. Similarly the fear of failing to fulfil contracts requiring some quality of service, or indirect monetary loss due to soiled corporate reputation, has motivated companies to submit complex designs like telecommunication switches or processor chips for formal analysis.

Now it is finally time to consider the thesis – what is its contribution?

To summarise, for a few years now the members of the formal design techniques group at the Swedish Institute of Computer Science have been building a framework for conducting formal arguments about programs written in the ERLANG [AVWW96] programming language.

The framework consists of four parts. First, a formal semantics for ERLANG has been developed: the behaviour of each language construct is described formally by indicating how the state of a program changes due to the execution of the language constructs, and what effects on the environment this has.

Next we have developed a logic for specifying the behaviour of a program. The specification logic captures some desirable properties of programs, such as the question whether a given program will always terminate its execution. The claim that an ERLANG program s has the property expressed by a formula ϕ of the specification logic is represented as the statement $s : \phi$.

The third part of the framework, the proof system, consists of a small set of rules that permit us to formally prove such statements. The claim $s : \phi$ above is represented in the proof system in the sequent $\Gamma \vdash s : \phi$, with the intuitive meaning that if all the assumptions in Γ (a sequence of statements about variables in s or ϕ) are true, then the claim $s : \phi$ can be derived in the proof system. Most of the proof rules are rather simple. Consider for instance a slightly simplified rule for introducing a conjunction:

$$\wedge_{\mathsf{R}} \frac{\Gamma \vdash \phi_1 \qquad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \wedge \phi_2}$$

The way we typically read this rule in this thesis is bottom-up: to prove that the assumptions in Γ imply the formula $\phi_1 \wedge \phi_2$ it suffices to show separately that Γ implies ϕ_1 and to show that Γ implies ϕ_2. The most complicated rules of the proof system result, as is typically the case, from the treatment of recursive program behaviour.

The proof system together with the operational semantics for ERLANG have been implemented in a software tool, a so called proof assistant. The tool helps us to check that proof steps are applied correctly, and can in many situations suggest a suitable proof rule that will make progress in the task of finding a proof. Reasoning on the level of the rule for conjunction above quickly becomes very tedious. To counter this, the tool offers a number of high-level proof rules that can collapse such tiny proof steps into larger proof steps.

Finally, though not part of the framework per se, but an important indication of its usefulness, we present a number of case studies that demonstrate the verification of properties of ERLANG programs.

1.1 Formal Reasoning about Open Distributed Systems

Here the foundation for each of the parts of the framework from the fields of software engineering or computer science is considered briefly; an extended discussion on related work is included in Chapter 7.

1.1.1 Open Distributed Systems and ERLANG

A central feature of open distributed systems as opposed to concurrent systems in general is their reliance on modularity. Large-scale open distributed systems, for instance in telecommunication applications, must accommodate complex functionality such as dynamic 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 put together relying only on component behaviour along these interfaces. That is, behaviour specification, and hence verification, needs to be parametric on subcomponents.

The programming platform considered in the thesis, i.e., ERLANG/OTP [Tor97] is a good representative of a class of concurrent languages that have adequate support for programming distributed applications; a second prominent member is Java together with its supporting libraries.

The basis of such a platform are the building blocks for concurrent behaviour, e.g., processes and threads, or a notion of concurrent objects. Such concurrent entities must be able to coordinate their activities; popular mechanisms for achieving this are semaphores, shared memory, remote method calls, or asynchronous message passing. Frequently a platform provides support for implicitly or explicitly grouping concurrently executing entities into more complex structures such as process groups, rings of processes or hypercubes. Further, in a distributed computing environment failures do happen, and applications with demands on constant availability need to take such failures into account. As a consequence, a good platform provides adequate support for detecting faults and the means to implement graceful recovery procedures. Like large software systems in general, open distributed systems are usually built from libraries of software components.

Although we focus exclusively on ERLANG in this thesis the majority of the results, excluding the language specific parts of the formal semantics, translate directly into results for other comparable platforms.

ERLANG

The ERLANG language was developed at Ericsson's Computer Science Laboratory during the 1980's [Arm97], and is at its core a conventional functional programming language, extended with a notion of processes and primitives for message passing.

Compared with other functional programming languages ERLANG lacks a few features often considered essential. There is for example no static type system – programs can fail at runtime due to trivial typing mistakes, though several type systems have been proposed [MW97, Lin96]. A second example is the lack of a proper lambda abstraction construct in early versions of ERLANG, function abstraction was by name only. However, later releases of ERLANG implementations have corrected this omission.

On the other hand, the language provides benefits seldom found in competing languages such as support for distribution, error recovery and hot code upgrade. In addition there are a number of high-quality libraries and tools available which provide support for many aspects of developing and maintaining large telecommunications applications such as a number of software patterns for programming client-server applications, a CORBA object request broker (ORB), a distributed database manager, and so on.

The language has been applied in a number of large development projects at Ericsson, with a generally very successful outcome. Experiences from the development of a state-of-the-art high-speed ATM switch [BR98] indicate that compared to an implementation in C or C++, the code size for an equivalent ERLANG implementation is at

least four times smaller [Wig01]. In addition, the number of errors is smaller with at least a factor of four. An overview of the development, use, and promotion of ERLANG inside Ericsson can be found in Däcker [Däc00].

From the point of view of proving properties about programs the ERLANG language contains features that pose problems for many verification methods based on explicitly enumerating all the states of the program under study. Problematic features include potentially unbounded data structures, processes that can be spawned at any moment, and that communicate with each other over potentially unbounded message queues.

Semantics of Open Distributed Systems

To reason in a formal fashion about the behaviour of an open distributed system a formal semantics of the design language in which the system is described is needed. This can be done in different styles, depending on the intended style of reasoning, see Chapter 7 for an overview. Our methodology is mainly tailored to operational semantics. Operational semantics are usually presented by transition rules involving labelled transitions between structured states [Plo81]. A natural approach to handling the different conceptual layers of entities in a complex language is to organise the semantics hierarchically using different sets of transition labels at each layer, and extending at each layer the structure of the state with new components as needed.

This hierarchical approach to operational semantics is adopted in our formalisation of ERLANG. There are two levels, one for evaluation of functional expressions and another for formalising the concurrency and communication aspects of the language. The evaluation of a functional expression is defined in a transition relation that does not depend on the state of the process that executes the expression. On the expression level of the semantics the main concerns are to regulate subexpression evaluation ordering (eager, left-to-right evaluation of subexpression) and pattern matching (ERLANG has a somewhat unusual binding strategy; see Chapter 3 for details). As an example, the transition $\mathit{pid} \,!\, v \xrightarrow{\mathit{pid} \,!\, v} v$ is enabled from any send expression $\mathit{pid} \,!\, v$ such that the first and second arguments of the send operator "!" have been evaluated to a process identifier *pid* and a value v respectively. The resulting expression is the value v, and the side effect of its computation, the sending of the value v to the process with process identifier *pid*, is represented by the expression action $\mathit{pid} \,!\, v$ on top of the arrow.

On the second level of the operational semantics concurrency aspects of the language such as process spawning and process communication are defined in a transition relation on ERLANG systems. Roughly, a system is a set of processes, where each process is a triple containing an expression being evaluated, a process identifier, and a message queue. However, borrowing from process algebra, we introduce a special parallel composition operator in the language instead of reasoning about sets of processes. While communication in ERLANG is asynchronous the operational semantics implements a synchronous communication scheme. Asynchronicity is recovered from the observation that a process can never block any input to the queue. A further concern on the level of systems is the assignment of process identifiers to processes, and the knowledge of these identifiers among other processes. In a sense a pid fills the

same role as a name in the π-calculus: unless a process is explicitly told about a name (pid) it should not know it or be able to retrieve it. However, because ERLANG is a real programming language with real implementation trade-offs there actually exists a built-in function that returns all the process identifiers in use. In this thesis therefore the only guarantee is that unique pids are assigned to processes.

Unusually for a programming language, if not for an operating system, ERLANG provides an explicit mechanism for recovering from abnormal process termination via the notion of bidirectional links between processes, over which process termination messages are sent. In the system semantics these links are modelled to permit reasoning about error recovery procedures.

The semantics developed for ERLANG is small step: even minute details in the computation of a functional expression or a system are considered as evaluation steps, and consecutive steps are not collapsed. The reason for this is that the treatment of side effects in such a semantics is particularly easy, whereas the drawback is that the state space in a verification can grow dramatically. Although not properly covered in this thesis, the solution is to factor out reasoning about side-effect free (e.g., non-communicating) ERLANG expressions and to treat them on the level of the proof system using the standard machinery of post- and pre-conditions [GC00].

1.1.2 The Specification Language

Reasoning about complex systems requires compositional reasoning, i.e., the capability to reduce arguments about the behaviour of a compound entity to arguments about the behaviours of its parts. To support compositional reasoning about ERLANG, a specification language should capture the labelled transitions at each layer of the transitional semantics. Further, since the behaviour of programs crucially depend on computations over data, the specification language has to be powerful enough to permit the definition of general predicates over various data domains.

As a result of these concerns our specification language is, on the syntactic and superficial level, a mix of two traditions: many-sorted first-order logic for describing data, and the modal μ-calculus[Par70, Koz83] for describing program behaviour. In their respective domains these logics have proved very successful, and there is a wealth of knowledge on how to code various correctness properties. However on a deeper level the foundation for our specification language is simply many-sorted first-order logic with equality, and with explicit fixed-point operators (the greatest fixed point $\nu X.\phi$ and the least fixed point $\mu X.\phi$). In this logic all aspects of semantics and specifications are represented.

The transition relations $s \xrightarrow{\alpha} s'$ representing the semantics of ERLANG are encoded as recursive predicates (using the least fixed-point operator) taking two program terms and an action as parameters, all encoded in the underlying term language of the logic. The box and diamond modalities of the μ-calculus are embedded by referring to the transition relations, with the meaning that a structured state s satisfies formula $\langle\alpha\rangle\Phi$ if there is an α-derivative of s (i.e. a state s' such that $s \xrightarrow{\alpha} s'$ is a valid transition) satisfying Φ, while s satisfies $[\alpha]\Phi$ if all α-derivatives of s satisfy Φ. An alternative to

this treatment which is explored in the proof assistant tool for reasons of efficiency, is to simply postulate program transitions as axiom proof rules.

Having access to the full power of first-order logic with equality in the specification language, also for conducting arguments about program terms, makes it possible to define correctness properties that consider both the actions of a program and the states encountered in a computation. For instance a number of useful state predicates are easily coded to characterise structured states.

Recursive program behaviour is described through use of fixed point operators. Roughly speaking, least fixed-point formulas $\mu X.\phi$ express eventuality properties, while greatest fixed-point formulas $\nu X.\phi$ express invariant properties. Nesting of fixed points allows complicated reactivity and fairness properties.

Some care is needed in implementing language specific reasoning principles. For instance, although the ERLANG system composed of two processes $p_1 \parallel p_2$ and the ERLANG system $p_2 \parallel p_1$ (both p_1 and p_2 are processes) have exactly the same set of future behaviours there are formulas in the logic that can tell the systems apart by considering their syntactic shape.

1.1.3 The Proof System

Verifying correctness properties of open distributed systems written in ERLANG requires reasoning about their interface behaviour relativised by assumptions about certain system parameters. Technically, this is achieved by using a Gentzen–style proof system, allowing free parameters to occur within the proof judgments. The judgments are of the form $\Gamma \vdash \Delta$ where Γ and Δ are sequences of assertions. A judgment is deemed valid if, for any interpretation of the free variables, some assertion in Δ is valid whenever all assertions in Γ are valid. Parameters are simply variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgement $X : \Psi \vdash p(X) : \Phi$ states that object p has property Φ provided the parameter X of p satisfies property Ψ.

Apart from the treatment of recursion the proof system represents a rather standard account of first-order logic. Below the two key properties, compositionality and the treatment of recursive behaviour, are explained in further detail.

Compositionality

Suppose we want to show that $r : \phi$ where r has a component q, i.e., $r = p\{q/X\}$ where X is a parameter of p. The proof of $r : \phi$ can then be split into two parts by introducing an assumption ψ on q, and proving separately that $q : \psi$ and that if we may assume $X : \psi$ then $p : \phi$ follows. Technically we achieve this through a term-cut proof rule of the shape:

$$\frac{\Gamma \vdash q{:}\Psi, \Delta \qquad \Gamma, X{:}\Psi \vdash p{:}\Phi, \Delta}{\Gamma \vdash p\{q/X\}{:}\Phi, \Delta}$$

Technically this rule is derivable from a standard cut-rule, motivating the slogan “Compositionality through cut introduction” as we argue, in contrast to Simp-

son [Sim95], that it is precisely the introduction, and not elimination of cuts from proofs, that allow compositional reasonings.

The term-cut proof rule can be used to introduce compositional reasoning on different levels of ERLANG programs. Consider for instance the function level. Let the predicate $e : eval(v)$ mean that the expression e can evaluate to the value v without causing side effects. A useful compositional proof rule derived from two applications of the term-cut rule permits the replacement of the two parts of a cons cell with the values they compute to.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : eval(v_1), \Delta \\ \Gamma \vdash e_2 : eval(v_2), \Delta \\ \Gamma \vdash [v_1 \mid v_2] : eval(v), \Delta\end{array}}{\Gamma \vdash [e_1 \mid e_2] : eval(v), \Delta}$$

On the process level the decomposition of a parallel composition is frequently essential to treat process spawning. The decomposition step can be accomplished with another derived variant of the term-cut rule (parallel composition cut):

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \psi_1, \Delta \\ \Gamma \vdash s_2 : \psi_2, \Delta \\ \Gamma, X : \psi_1, Y : \psi_2 \vdash X \parallel Y : \phi, \Delta\end{array}}{\Gamma \vdash s_1 \parallel s_2 : \phi, \Delta}$$

where the proof obligation to establish that the system $s_1 \parallel s_2$ satisfies ϕ is replaced by the obligations to establish that s_1 satisfy ψ_1, and s_2 satisfy ψ_2 respectively, and that any systems X and Y that satisfy ψ_1 and ψ_2 satisfy ϕ when they are composed. The essential difficulty when applying such a proof rule is to come up with good formulas ψ_1 and ψ_2 that suffice to establish ϕ. We will see quite a few examples of such reasoning in Chapter 6.

Since the box and diamond modalities of the logic are derived constructs, implemented in terms of the transition relations, there is no need to include them on the basic level of the proof system rules for treating combinations of program construct and modalities, in contrast to many other approaches to compositional verification [Sti85, Win90, ASW94]. For instance the rule below for input under parallel composition, on the system level of the semantics, is derivable:

$$\frac{\Gamma, S_1 : \phi_1 \vdash S_1 \parallel s_2 : \phi, \Delta}{\Gamma, s_1 : \langle pid?v \rangle \phi_1 \vdash s_1 \parallel s_2 : \langle pid?v \rangle \phi, \Delta}$$

The rule expresses that if a component system in a parallel composition can take an input step, then so can the parallel composition.

Recursion

The means of arguing about recursive program behaviour in the proof system is, technically, through a scheme for well-founded induction on ordinals.

Before considering the details of the scheme itself, let us examine a few instances of recursive program behaviour and typical correctness properties. Clearly any invocation of the ERLANG function `loop` below will never terminate

```
loop() -> loop().
```

On the other hand, the function `isProperList` below always terminates since ERLANG has no circular lists.

```
isProperList([]) -> true;
isProperList([Head|Tail]) -> isProperList(Tail).
```

We can capture their respective behaviours with the fixed point formula

$$\phi \equiv \nu X.\langle\tau\rangle X$$

expressing non-terminating behaviour and

$$\psi \equiv \mu X.[\tau]X$$

expressing terminating behaviour. Here τ is the expression computation step action. Intuitively the first property expresses that the possibility to perform a computation step is always true. The second property expresses that any program can only take a finite number of consecutive computation steps.

Suppose we set out to prove $\vdash$ `loop()` $: \phi$, after deriving one computation step, the original proof goal is again encountered. It is clear that these proof steps can be repeated indefinitely. A number of conditions for ensuring safe termination of proofs at this point, since a greatest fixed point has been unfolded, have been proposed: a constant scheme in Stirling and Walker [SW91] and a tagging scheme in Winskel [Win91].

In this thesis, and earlier in Dam et al. [DFG98b], an explicit fixed point induction scheme is used for handling a greatest fixed point on the right-hand side of the turnstile. First, the fixed point is approximated; we commit to proving the formula for an unknown ordinal in the new proof state

$$\vdash \texttt{loop()} : \phi^{\kappa}$$

where ϕ^{κ} means approximating ϕ κ times. Unfolding the approximated fixed point gives rise to a new ordinal variable κ' and an inequation $\kappa' < \kappa$ on the left-hand side. Then, eventually, the proof state

$$\kappa' < \kappa \vdash \texttt{loop()} : \phi^{\kappa'}$$

is reached. Thus $\vdash$ `loop()` $: \phi^{\kappa}$ is proved if $\kappa' < \kappa \vdash$ `loop()` $: \phi^{\kappa'}$ can be. Clearly these proof steps can be repeated an infinite number of times, causing the chain of decreasing ordinals begun by the inequation $\kappa' < \kappa$ to also grow infinitely long. However, since there exists no such infinite decreasing chain of ordinals, eventually the sequent

$$\Gamma \vdash \texttt{loop()} : \phi^{0}$$

must be reached, and this sequent is trivially true because ϕ^0 is true for any greatest fixed point formula ϕ.

The proof structure indicated above corresponds to a co-inductive proof scheme [MT91], which is generally needed when reasoning about entities of a non-well-founded nature such as non-terminating processes or infinite streams.

In Andersen [And94] least fixed points are handled through an infinitary rule that moves part of the reasoning outside the proof system itself. In this thesis, however, the dual nature of greatest and least fixed points is explored leading to a rule for handling a least fixed to the left of the turnstile that is completely analogous to the rule for dealing with greatest fixed points to the right. Consider the example with `isProperList` above, and the proof goal

$$\vdash \texttt{isProperList}(L) : \psi$$

where L is a proper list. The obvious reason why any application of the function will terminate is that its argument list will decrease in structural complexity, motivating structural induction as the obvious proof technique. In our proof system the structural induction argument is mimicked by making explicit the structure of a proper list through the introduction of a parametric least fixed point definition as an assumption

$$\gamma \;\equiv\; \mu X.(\lambda L.(L = \texttt{[]}) \vee (\exists H, T.L = \texttt{[}H \texttt{|} T\texttt{]} \wedge (X\ T)))$$

which expresses that a proper list is either empty or it consists of a head and tail, and the tail is itself a proper list. Furthermore, due to the use of the least fixed point, there can be only a finite number of tail cells. The new proof goal becomes

$$(\gamma\ L) \vdash \texttt{isProperList}(L) : \psi$$

After approximating γ, and unfolding γ and ψ, eventually the proof goal

$$\kappa' < \kappa, \left(\gamma^{\kappa'}\ T\right) \vdash \texttt{isProperList}(T) : \psi$$

is reached. The same reasoning that motivated discharge of the greatest fixed point to the right of the turnstile is valid here also. Clearly the chain of decreasing ordinals will grow only until $\gamma^0\ T'$ for some tail T' is reached. But since the ordinal 0 decorates a least-fixed point it cannot be valid, and thus the assumption is wrong, and the sequent has been proved. This proof structure corresponds to an inductive proof scheme. Complications arise in these proof schemes because of conflicting fixed points; details are elaborated in Section 4.5.3.

1.1.4 The Proof Assistant

The proof assistant tool called the "ERLANG verification tool", abbreviated EVT, implements the first-order specification logic, embeds ERLANG syntax and semantics into the first-order logic component, and provides a core set of proof rules. The underlying proof structure is a graph; to implement the fixed point induction rule it is necessary to remember the history of proof nodes in a proof.

The proof assistant implements a number of standard features of other proof assistants, more information can be found in the documentation of COQ [DFH+93], PVS [ORR+96] or Isabelle [Pau94]: lemmas, a subsumption rule, tactics and tacticals implemented in Standard ML for high-level proof rules. A number of such high-level rules are available; one example is a set of tactics for deriving the next states from a transition relation.

An experimental graphical user interface is available that implements a number of useful features such as the ability to suggest the next proof rule to try. Proof graphs can be visualized using the DaVinci graph editor [FW94].

1.2 Overview of the Thesis

Chapter 2 defines the specification logic that is used to formalise correctness properties of ERLANG programs and to encode the operational semantics of ERLANG. Chapter 3 starts with an informal overview of the ERLANG language. Then a formal semantics for ERLANG is developed, and a number of results about the semantics are established. The following chapter represents the core of the framework, presenting the formal proof rules that are used to reason about ERLANG code, and proving that these rules represent sound reasoning principles. In addition the chapter covers the embedding of the ERLANG semantics into the proof system. Next, in the short Chapter 5, the proof assistant tool for reasoning about ERLANG code is described. To evaluate the framework a number of case studies are reported in Chapter 6; these range from small examples mainly intended to illustrate the framework such as the verification of a quicksort algorithm to more ambitious studies such as the compositional verification of a typical client-server application. Chapter 7 contains a short survey of related approaches. Finally, the results are summarised in Chapter 8, which also contains a discussion on a number of topics remaining for future investigation.

Parts of this thesis are based on material from earlier articles. These are, in chronological order of writing:

- Paper 1: Formal verification of a leader election protocol in process algebra by Lars-Åke Fredlund, Jan Friso Groote and Henri Korver in *Theoretical Computer Science*, volume 177(2), 1995.

  *This paper describes the verification of a leader election protocol in the process algebra μ*CRL*, using equational reasoning. The proof has not been formalised in a theorem proving tool, although support exists in the COQ proof assistant tool for formalising such proofs. This paper has been included in the thesis mainly to serve as a comparison with a related effort in our framework.*

- Paper 2: Towards Parametric Verification of Open Distributed Systems by Mads Dam, Lars-Åke Fredlund and Dilian Gurov in *Compositionality: The Significant Difference*, LNCS 1536, Springer Verlag 1998.

  This is the first paper that accurately describes our approach to the verification of ERLANG *code, and the underlying proof system.*

- Paper 3: System Description: Verification of Distributed ERLANG Programs by Thomas Arts, Mads Dam, Lars-Åke Fredlund and Dilian Gurov in *Proceedings of CADE'98* published in LNAI 1421, Springer Verlag 1998.

  This is a short paper that describes a previous version of the proof assistant tool.

- Paper 4: A Framework for Formal Reasoning about Open Distributed Systems by Lars-Åke Fredlund and Dilian Gurov in *Proceedings of ASIAN'99* published in LNCS 1742, Springer Verlag 1998.

  This report is, in a sense, a broad overview of the framework effort. The set example studied in Section 6.5 was first covered in this paper.

- Paper 5: A Tool for Verifying Software Written in ERLANG by Thomas Arts, Gennady Chugunov, Mads Dam, Lars-Åke Fredlund, Dilian Gurov and Thomas Noll submitted to the journal on Software Tools for Technology Transfer (STTT), 2000.

  This paper describes the current version of the proof assistant tool.

1.3 Contributions

The contributions of the overall approach documented in the thesis can be summarised as follows.

The development of an operational semantics for a complex concurrent functional programming language which cleanly separates the concerns of function evaluation from issues of concurrency and communication represents a non-trivial achievement. Further the design of a rich program logic for describing the behaviours and data part of ERLANG code, on detailed levels, is a contribution to work on program logics.

The development of the proof system is crucial. Although it follows in the tradition of earlier works the fixed point rules are considerably simplified, leading to a natural statement of least fixed point arguments. Modal proof rules refer to the transition relation, which provide the sole source of defining the meaning of language constructs. This separation of concerns is shown to lead to a natural treatment of compositional reasoning using standard proof rules. The combination of compositional reasoning with a co-inductive proof scheme solves the problem of reasoning about possibly unbounded process spawning. Although the result is not surprising, the resulting proof arguments are compact and clean.

A further achievement is the design and implementation of a prototype proof assistant tool for the verification of non-trivial ERLANG code.

The set of non-trivial examples covered in the thesis illustrate the applicability of the framework to ERLANG code of varying nature in a uniform and general manner: in verification of classical functional programs, where the majority of reasoning is about data, and in the verification of dynamic and partially open process networks of different shapes, where the ability to perform compositional reasoning is crucial.

In summary, we have taken a real programming language and developed an intuitive operational semantics together with a specification language and a proof system. Further the whole framework has been embedded in a proof assistant tool, and ample support has been provided for higher-level reasoning about ERLANG programs, something not previously achieved for concurrent programming languages. The final argument is demonstrated by the case studies: it really is possible to use the framework for reasoning about industrially relevant code.

1.4 Personal Contributions

Since much of the work reported in this thesis is part of a collaboration between researchers primarily located at the Swedish Institute of Computer, a clarification of my role in the effort is needed.

Below each chapter will be considered in turn, starting with the specification logic (Chapter 2). The logic has been developed jointly by Mads Dam, Dilian Gurov and me during the course of a number of papers. However, only in this thesis is the break with the modal μ-calculus made explicit via the encoding of modalities in an underlying logic.

In formulating the present ERLANG semantics, and in embedding the language theory in our proof system and proof assistant, I have been largely responsible. Earlier semantic accounts were jointly developed with Mads Dam and Dilian Gurov.

The handling of fixed points in the proof system is due to ideas originally put forward by Mads Dam and later refined the paper [DFG98b] by Mads Dam, Dilian Gurov and me. Contributions in that paper include significant simplifications of the conditions for the discharge rule, and an improved proof, all achieved together with Dilian Gurov. Other parts of the proof system were jointly developed. For instance, the correspondence between modalities and transition relations due to an idea by Simpson [Sim95], was first implemented for ERLANG in the EVT proof assistant, and later described for CCS [DG00a]. In the present thesis the condition under which discharge is sound has been made more explicit, these improvements are solely due to me.

The present EVT proof assistant was jointly developed by Dilian Gurov and me, except for the graphical user interface which is due to Gennady Chugunov.

In each of the case studies reported in the thesis I have played a major part: the billing agent reported in Section 6.4 (single contributor), the study on leader election protocols in ERLANG reported in Section 6.7 (single contributor), leader election in μCRL reported in Section 6.6 (shared work between co-authors), as well as numerous smaller studies such as the Quick Sort example and the Set example (both shared with different co-authors).

Chapter 2

Foundations

This chapter defines the specification logic in which properties of ERLANG programs are formalised. Inspired by the modal μ-calculus [Par70, Koz83], it is a first-order logic with equality where the usual modalities $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ of Hennessy-Milner Logic [HM80] are encoded. Explicit greatest and least fixed point operators permit the definition of recursive predicates. Most of the material presented here is well known from other works [Koz83, Sti92] so the presentation will be brief.

As a basis the standard machinery from set theory is imported, with standard notation. For instance, let $\mathcal{P}(S)$ denote the power set of a set S.

2.1 Terms and Sorts

To begin we recall the usual machinery of many-sorted signatures, sorts, and terms.

Definition 1 (Signatures)**.** *A signature Σ is a pair (S, F) consisting of*

- *A non-empty set S of sort names*
- *A set of function symbols F disjoint from S where each $f \in F$ has a domain type $S^* \times S$.*

Function symbols with they domain type s, for some sort s, will be referred to as *constants*. Given a signature Σ with a function symbol f of domain type $s_1 \times \ldots \times s_n \times s$, let $domain(f)$ denote $s_1 \times \ldots \times s_n$ and let $range(f)$ denote s.

Definition 2 (Terms)**.** *Let Σ be a signature (S, F), and let X be an S-indexed family of sets X_s such that each set contains a countably infinite number of variables of sort $s \in S$, disjoint from functions and sorts in Σ. The set $T(\Sigma, X)_s$ of all terms of sort $s \in S$ over Σ and X is inductively defined:*

- $X_s \subseteq T(\Sigma, X)_s$
- *For all function symbols f of sort $s_1, \ldots, s_n, s$ and all terms $t_1, \ldots, t_n$ such that for all $1 \leq i \leq n$, $t_i \in T(\Sigma, X)_{s_i}$, then $f(t_1, \ldots, t_n) \in T(\Sigma, X)_s$.*

In the following we will use the expressions "sorts" and "types" interchangeably.

Conventions In the thesis a number of equality symbols will occur. Functions will be defined using definitional equality $\stackrel{\Delta}{=}$. Syntactical equality of terms will be denoted with "$\equiv$", whereas the semantical notion of equality, on the level of the proof system we develop, will be denoted with "$=$". The basic conventions about how various symbols are depicted are:

- Concrete names of sorts are written in sans serif and will always start with a lowercase letter, e.g., nat.
- Variables of a given sort will be written in italics, and have an initial uppercase letter, e.g., N.
- Meta-variables, e.g., the symbol n in the statement "for all natural numbers n" over a given sort are written in italics and commence with a lowercase letter, unless the meta-variable ranges only over variables of a particular sort. In the latter case the meta-variable will have an initial uppercase letter. An example of the variable convention is illustrated by the statement "let V range over the ERLANG variables".

Definition 3 (Free Variables)**.** *The set of free variables $fv(t)$ in a term t is defined inductively:*

- *if $t \in X_s$ for some sort s then $fv(t) \stackrel{\Delta}{=} \{t\}$*
- *if $t \equiv f(t_1, \ldots, t_n)$ then $fv(t) \stackrel{\Delta}{=} fv(t_1) \cup \ldots \cup fv(t_n)$*

A closed term *has no free variables.*

A Notation for Signatures As a large number of sorts will be used to provide a meaning to ERLANG constructs, a clean and compact syntax is needed to represent signatures. Henceforth a signature will usually be written in in a more stylised format, which will be introduced through simple examples. A signature Σ can be depicted by listing its sorts, and for each sort s show the functions that have the sort as range. The concrete format is demonstrated by an example. A sort named s such that the functions $f_1, \ldots, f_k$ are the only functions with s as range, is depicted together with its functions as shown below:

$$\texttt{type}\ s \stackrel{\Delta}{=} f_1\ \texttt{of}\ s_{11}, \ldots, s_{1n}\ \mid \ldots \mid f_n\ \texttt{of}\ s_{k1}, \ldots, s_{kn}$$

In case the domain of a function is empty then the keyword of can be omitted. A concrete example is represented by the natural numbers:

$$\texttt{type}\ \mathsf{nat} \stackrel{\Delta}{=} 0 \mid +1\ \texttt{of}\ \mathsf{nat}$$

We permit terms of well-known sorts such as the natural numbers to be written in an infix syntax. For instance, the successor of the natural number 0 can be written $0+1$.

Next the notion of a sort consisting of lists of elements, and tuples, are supported through encodings. The sort name s_{list}, with functions $[\,]_s$ and $[\,|\,]_s$ corresponding to the empty list and a cons cell, refer to the sort

$$\texttt{type}\ s_{\mathsf{list}} \triangleq [\,]_s \mid [\,|\,]_s\ \texttt{of}\ s, s_{\mathsf{list}}$$

A tuple sort $s_1 \times \ldots \times s_n$ refers to a sort:

$$\texttt{type}\ s_{s_1 \times \ldots \times s_n} \triangleq c_{s_1 \times \ldots \times s_n}$$

As a convention the empty list, for a sort s, will usually be written simply $[\,]$, and a cons cell $[\,|\,]_s(t_1, t_2)$ as $[t_1|t_2]$. A tuple constructor $c_{s_1 \times s_2}$ will typically be written $\langle t_1, t_2 \rangle$ unless the notation is ambiguous.

Predefined Sorts In the following we presuppose a number of sorts. These are the natural number data type nat from above, and a sort int $\triangleq$ *int* of nat, nat which is intended to represent the integers such that an integer is represented by the difference between its natural number components. Further assume a sort atom which contains a countable number of functions of zero arity corresponding to symbols built from characters, e.g., "a", "b", "abc".

2.2 Syntax of the Logic

Assume a set of predicate variables and let U, V range over these. Further assume a set of variables ranging over ordinals, let κ range over these variables. Let β range over the ordinals $0, 1, \ldots, \omega, \omega + 1, \ldots, \omega + \omega, \ldots$.

In defining the syntax of formulas let the meta variables ϕ, ψ range over the formulas, t range over the terms, X, Y over the term variables, s range over the sort names. To indicate that a meta variable X ranges over terms or variables of a sort s a subscript notation X_{nat} will be used.

Types of Formulas The formulas in the logic are considered to have types. Let prop be a set containing having two elements, $\{tt\}$ and the empty set $\{\}$. The formula types, ranged over by s_ϕ, are then the least set satisfying the construction rules:

- prop is a formula type
- if s is a sort and s_ϕ is a formula type, then $s \rightarrow s_\phi$ is a formula type.

A formula of type $s \rightarrow s_\phi$ for some sort s and formula type s_ϕ will be referred to as an *abstraction*. The arity of a formula type is defined in the obvious way: *arity*(prop) $=$ 0 and *arity*$(s \rightarrow s_\phi) = 1 +$ *arity*(s_ϕ).

Definition 4. *The formulas of the logic considered in this text are defined recursively as the least set satisfying the syntactic construction rules below:*

- *if t_1 and t_2 are terms then $t_1 = t_2$ is a formula*
- *if ϕ_1 and ϕ_2 are formulas then $\phi_1 \vee \phi_2$ is a formula*
- *if ϕ is a formula then $\neg\phi$ is a formula*
- *if ϕ is a formula, X is a term variable and s is a sort name then $\exists X : s.\phi$ and $\lambda X : s.\phi$ are formulas*
- *if ϕ is a formula and t is a term then $\phi\ t$ is a formula*
- *if ϕ is a formula, U is a predicate variable and β is an ordinal then $\mu U : s_\phi.\phi$ and the approximated least fixed point $(\mu U : s_\phi.\phi)^\beta$ are formulas*
- *A predicate variable U is a formula*

All the constructs are standard. Types are added to the fixed point construct to clarify the denotational semantics of formulas. The binding powers of the connectives is, from stronger to weaker, negation, disjunction, quantification and the least fixed point. That is, the least fixed point operator extend as far to the right as possible in a formula. As usual parentheses will be used to limit the scope of an operator.

Definition 5. *An occurrence of a predicate variable is* positive *if it occurs in the scope of an even number of negation symbols.*

In the standard manner a fixed point formula $\mu U : s_\phi.\phi$ can be formed only when all occurrences of U in ϕ are positive, to ensure that the semantics of ϕ is monotone in U so that a fixed point of the corresponding function exists.

Let $t\phi$ range over both the terms and the formulas, and let $X\phi$ range over both the term variables and the predicate variables. We assume standard substitution function where a term t' replaces all occurrences of a variable X in t denoted by $t\{t'/X\}$. Further we assume the standard capture avoiding substitution $\psi\{t\phi/X\phi\}$ of a term (or formula) $t\phi$ for a term variable (predicate variable) $X\phi$, in a formula ϕ.

Table 2.1 defines a number of formula shorthands. Note for instance that the greatest fixed point operator is defined in terms of the least fixed point operator. Note also that assertions $t : \phi$ simply abbreviate applications in our logic. A sequence of lambda abstractions or quantifications $X_1 : s, \ldots, X_n : s$ all over the same sort s can be abbreviated $X_1, \ldots, X_n : s$. Frequently the type of a fixed point will be omitted, e.g., $\mu U : s_\phi.\psi$ will normally be written as $\mu U : \psi$.

The binding power of the additional operators are the expected ones, i.e., conjunction ($\wedge$) has the same precedence as disjunction ($\vee$) and implication binds weaker than both but stronger than the quantifiers $\exists$ and $\forall$. The assertion $t : \phi$ is the operator with the weakest binding power.

$$
\begin{array}{rcl}
t_1 \neq t_2 & \triangleq & \neg\,(t_1 = t_2) \\
\mathit{true} & \triangleq & \phi \vee \neg\phi \text{ (for some formula } \phi \text{ of type prop)} \\
\mathit{false} & \triangleq & \neg \mathit{true} \\
\phi_1 \wedge \phi_2 & \triangleq & \neg\,(\neg\phi_1 \vee \neg\phi_2) \\
\phi_1 \Rightarrow \phi_2 & \triangleq & \neg\phi_1 \vee \phi_2 \\
\forall X : s.\phi & \triangleq & \neg\exists X : s.\neg\phi \\
\nu U : s_\phi.\phi & \triangleq & \neg\mu U : s_\phi.\neg(\phi\{\neg X/X\}) \\
t : \phi & \triangleq & \phi\ t \\
\lambda X_1 : s_1, \ldots, X_n : s_n.\phi & \triangleq & \lambda X_1 : s_1.\ \ldots\ .\lambda X_n : s_n.\phi \\
\exists X_1 : s_1, \ldots, X_n : s_n.\phi & \triangleq & \exists X_1 : s_1.\ \ldots\ .\exists X_n : s_n.\phi
\end{array}
$$

Table 2.1: Formula Abbreviations

Well-typed Formulas Not all formulas that can be constructed using Definition 4 will be considered in the thesis. In the following attention is restricted to the "well-typed" formulas. These are the formulas ψ that can be provided with a formula type s_ϕ, sorts for free value variables LV, and formula types for predicate variables RV, such that the proof judgement $RV, LV \vdash \psi : s_\phi$ is provable using the type checking rules in Definition 6. Concretely RV maps free predicate variables in ψ to formula types and LV maps free term variables in ψ to sorts. Adding a mapping $X \mapsto s$, or replacing an old mapping, in LV (or RV) is denoted with $LV[X \mapsto s]$.

Definition 6. *A formula ψ* conforms *to a type s_ϕ if $RV, LV \vdash \psi : s_\phi$ is provable using the type checking rules in Table 2.2 for some predicate and term variable mappings RV and LV.*

Note that the type checking rules refer to the type checking rules of terms, which are assumed. Further note that since term variables already have types, the mapping LV is not required. In fact the type of any formula is unique. However here we already anticipate the introduction of subtyping through type membership predicates in the logic. Often, when they have no effect, the mappings RV and LV will not be written out.

As formulas can trivially conform to different types only if these types have the same arity (if $LV \vdash \psi : s_{\phi 1}$ and $LV \vdash \psi : s_{\phi 2}$ then $arity(s_{\phi 1}) = arity(s_{\phi 2})$), we will in the following refer to the arity of a formula ψ as *arity*(ψ).

Example 1 (Formulas and Type Checking)**.** The type checking rules are illustrated in

$$\frac{LV \vdash t_1 : s \quad LV \vdash t_2 : s}{RV, LV \vdash t_1 = t_2 : \mathsf{prop}}$$

$$\frac{RV, LV \vdash \psi_1 : \mathsf{prop} \quad RV, LV \vdash \psi_2 : \mathsf{prop}}{RV, LV \vdash \psi_1 \vee \psi_2 : \mathsf{prop}}$$

$$\frac{RV, LV \vdash \psi : \mathsf{prop}}{RV, LV \vdash \neg\psi : \mathsf{prop}}$$

$$\frac{RV, LV[X \mapsto s] \vdash \psi : \mathsf{prop}}{RV, LV \vdash \exists X : s.\psi : \mathsf{prop}}$$

$$\frac{RV, LV[X \mapsto s] \vdash \psi : s_\phi}{RV, LV \vdash \lambda X : s.\psi : s \to s_\phi}$$

$$\frac{RV, LV \vdash \psi : s \to s_\phi \quad LV \vdash t : s}{RV, LV \vdash \psi\ t : s_\phi}$$

$$\frac{RV[U \mapsto s_\phi], LV \vdash \psi : s_\phi}{RV, LV \vdash \mu U : s_\phi.\psi : s_\phi}$$

$$\frac{-}{RV[U \mapsto s_\phi], LV \vdash U : s_\phi}$$

Table 2.2: Type Checking of Formulas

a small example by proving the type judgement:

$$\dfrac{\dfrac{\dfrac{-}{[X \mapsto \mathsf{nat}] \vdash X : \mathsf{nat}} \quad \dfrac{-}{[X \mapsto \mathsf{nat}] \vdash X : \mathsf{nat}}}{[X \mapsto \mathsf{nat}] \vdash X = X : \mathsf{prop}}}{\dfrac{\vdash \lambda X : \mathsf{nat}.X = X : \mathsf{nat} \to \mathsf{prop} \qquad \vdash 0 : \mathsf{nat}}{\vdash (\lambda X : \mathsf{nat}.X = X)\, 0 : \mathsf{prop}}}$$

2.3 Semantics

The semantics of a formula is defined as an element of the set prop if the formula is not an abstraction, and otherwise as a curried function that when all arguments are supplied, returns an element of the set prop. An ordering ($\sqsubseteq$) is defined on the denotations:

Definition 7. *If $B_1, B_2 \in \mathsf{prop}$ let $B_1 \sqsubseteq B_2$ denote $B_1 \subseteq B_2$. If $f_1, f_2 \in [s \to s_\phi]$ (f_1, f_2 are functions with domain s and range s_ϕ) let $f_1 \sqsubseteq f2$ denote the condition that for any element $t \in s$ the inclusion $f_1\ t \sqsubseteq f_2\ t$ holds. If $F \subseteq [s \to s_\phi]$ then define the upper ($\sqcup$) and lower bounds ($\sqcap$) of the set F as:*

$$\sqcup F \triangleq \lambda X : s. \bigsqcup_{f \in F} f\ X$$
$$\sqcap F \triangleq \lambda X : s. \bigsqcap_{f \in F} f\ X$$

Definition 8 (Semantics of Formulas, Valuations)**.** *The semantics of formulas is defined in Table 2.3, relative to a valuation ρ mapping predicate and term variables to appropriate values.*

The syntax $\rho[\phi/U]$ expresses a new valuation that coincides with ρ except that the predicate variable U is mapped to ϕ. Ordinal variables and term variables can be analogously remapped. As the intended model in this thesis is countable the quantification over ordinals in the fixed point definition clause can be restricted to countable ordinals.

To ensure that Definition 8 is meaningful a few observations are required.

Lemma 1. *The semantics of the logic constructs is monotone with respect to a predicate variable U and the valuation ρ.*

Proof. The proof is simple. We consider below the case for disjunction and negation.

Consider first the operators $=$, $\vee$ and $\exists$, application and abstraction which are all easily handled by induction over the structure of the formula. Examine for instance the case $\|\,\phi_1 \vee \phi_2\,\|_\rho$. From the induction hypothesis we know that (for $i \in \{1, 2\}$) $\|\,\phi_i\,\|_\rho \subset \|\,\phi_i\,\|_{\rho'}$ where $\rho(U) \sqsubset \rho'(U)$ and otherwise the valuations are identical. But clearly then $\|\,\phi_1 \vee \phi_2\,\|_{\rho'} \sqsupset \|\,\phi_1 \vee \phi_2\,\|_\rho$.

For negation the previously given condition that any fixed point variable U may only occur under an even number of negations (positively), is sufficient to ensure monotonicity. □

$$
\begin{array}{rcl}
\| t_1 = t_2 \|_\rho & \triangleq & \text{if } t_1\rho = t_2\rho \text{ then } \{tt\} \text{ else } \emptyset \\
\| \phi_1 \vee \phi_2 \|_\rho & \triangleq & \| \phi_1 \|_\rho \cup \| \phi_2 \|_\rho \\
\| \neg\phi \|_\rho & \triangleq & \{tt\} - \| \phi \|_\rho \\
\| \exists Y : s.\phi \|_\rho & \triangleq & \bigcup_{v \in s} \left(\| \phi \|_{\rho[v/Y]} \right) \\
\| \lambda Y : s.\phi \|_\rho & \triangleq & \lambda X : s. \| \phi \|_{\rho[X/Y]} \\
\| \phi\ t \|_\rho & \triangleq & \| \phi \|_\rho\ \| t \|_\rho \\
\| \mu U : s_\phi.\phi \|_\rho & \triangleq & \bigcup_\beta \| (\mu U : s_\phi.\phi)^\kappa \|_{\rho[\beta/\kappa]}, \quad \beta \text{ any ordinal} \\
\| (\mu U : s_\phi.\phi)^\kappa \|_\rho & \triangleq & \begin{cases}
\lambda X_1 : s_1.\ \ldots\ .\lambda X_n : s_n.\emptyset \\
\quad \text{if } \rho[\kappa] = 0 \text{ and } s_\phi = s_1 \to\ \ldots\ \to s_n \to \mathsf{prop} \\
\| \phi \|_{\rho[\| (\mu U : s_\phi.\phi)^\kappa \|_{\rho[\beta/\kappa]}/U]} \\
\quad \text{if } \rho[\kappa] = \beta + 1 \\
\bigcup_\beta \left\{ \| (\mu U : s_\phi.\phi)^\kappa \|_{\rho[\beta/\kappa]} \mid \beta < \rho[\kappa] \right\} \\
\quad \text{if } \rho[\kappa] \text{ is a limit ordinal}
\end{cases} \\
\| U \|_\rho & \triangleq & \rho(U)
\end{array}
$$

Table 2.3: The Denotation of Formulas

Since the semantics of fixed point definitions is monotone, and since the functions of a certain arity forms a complete lattice, then according to Tarski's fixed point theorem [Tar55] the least fixed point $\mu V : s_\phi \| \phi \|_{\rho[V/U]}$ must exists, as well as a greatest fixed point.

Theorem 1. $\| \mu U : s_\phi . \phi \|_\rho$ is the least fixed point of the operator $\lambda U : s_\phi . \| \phi \|_\rho$ and

$$\| \mu U : s_\phi . \phi \|_\rho = \bigsqcap \left\{ S \mid \| \phi \|_{\rho[S/X]} \sqsubseteq S \right\}$$
$$\| \nu U : s_\phi . \phi \|_\rho = \bigsqcup \left\{ S \mid S \sqsubseteq \| \phi \|_{\rho[S/X]} \right\}$$

Proof. Follows also from Knaster-Tarski's fixed point theorem. □

Proposition 1. *Suppose that $\beta \leq \beta'$, then*

$$\| (\mu U : s_\phi . \phi)^\beta \|_\rho \sqsubseteq \| (\mu U : s_\phi . \phi)^{\beta'} \|_\rho$$

and

$$\| (\nu U : s_\phi . \phi)^{\beta'} \|_\rho \sqsubseteq \| (\nu U : s_\phi . \phi)^\beta \|_\rho$$

Proof. Follows by well-founded induction. □

Definition 9. *The* closure ordinal κ *of an operator $\lambda U : s_\phi . \| \phi \|_\rho$ is the least ordinal κ such that*

$$\| (\mu U : s_\phi . \phi)^\kappa \|_\rho = \| (\mu U : s_\phi . \phi)^{\kappa+1} \|_\rho$$

Now that the meaning of a formula is clear, it is time to extend the notion of types with subtyping.

Definition 10. *The subtype s of a type s', given by an abstraction ϕ of formula type $s' \to \mathsf{prop}$ consists of the closed terms $t \in T(\Sigma, X)_{s'}$ such that under any valuation ρ, $\| \phi\ t \|_\rho \neq \emptyset$. This is written $s' \triangleq \{t : s \mid \phi\ t\}$.*

We require that any such subtype set is non-empty to ensure soundness of the proof system introduced in Chapter 4.

Example 2 (Formula Example: The Even Natural Numbers)**.** To exemplify the fixed point formula notation a few well-known properties of the natural numbers are formulated in the logic. As before a type nat is assumed with zero 0 and successor constructor $+1$.

The type of the even natural numbers can easily be expressed:

$$\mu U : s_\phi . \lambda N : \mathsf{nat} . N = 0 \vee \exists N' : \mathsf{nat} . N = N' + 2 \wedge U\ N'$$

where $N' + 2$ abbreviates $(N' + 1) + 1$. In the following we let ψ abbreviate $\lambda N : \mathsf{nat} . N = 0 \vee \exists N' : \mathsf{nat} . N = N' + 2 \wedge U\ N'$.

To illustrate the semantics we prove, in an informal fashion, that the denotation of the above fixed point is a function from the natural numbers to prop such that only the even natural numbers are not mapped to the empty set.

$$\| \mu U : s_\phi.\psi \|_\rho = \bigcup_\beta \| (\mu U : s_\phi.\psi)^\kappa \|_{\rho[\beta/\kappa]}$$

If $\beta = 0$ then by definition

$$\| (\mu U : s_\phi.\psi)^\kappa \|_{\rho[0/\kappa]} = \lambda N : \mathsf{nat}.\emptyset$$

If $\beta = n + 1$ for some natural number $n > 0$ and

$$\| (\mu U : s_\phi.\psi)^\kappa \|^{\mathsf{nat}}_{\rho[n/\kappa]} = f'$$

where f' is a total function from the natural numbers to the booleans then

$$\begin{array}{lll} & \| (\mu U : s_\phi.\psi)^\kappa \|^{\mathsf{nat}}_{\rho[n+1/\kappa]} \\ = & \lambda N : \mathsf{nat}. & \\ & \quad (\text{if } X = 0 \text{ then } \{tt\} \text{ else } \emptyset) \\ & \cup \quad \bigcup_{N' \in \mathsf{nat}} ((\text{if } N = N' + 2 \text{ then } \{tt\} \text{ else } \emptyset) \cap f'\ N') \end{array}$$

Thus if $n = 1$ then the denotation is $\lambda N : \mathsf{nat}.$if $N = 0$ then $\{tt\}$ else $\emptyset$. If $n = 2$ then the denotation is $\lambda N : \mathsf{nat}.$if $N = 0$ or $N = 2$ then $\{tt\}$ else $\emptyset$. For the case $n = n'+1$ the denotation is a function that maps all even natural numbers up to $(n-1) * 2$ to the set $\{tt\}$. For the first limit ordinal ω, the denotation is

$$\bigcup_{n \in \mathsf{nat}} \left\{ \| (\mu U : s_\phi.\mu U : s_\phi.\psi)^\kappa \|_{\rho[n/\kappa]} \mid n < \beta \right\}$$

which is the function that maps every even natural number to the set $\{tt\}$. In addition, the denotation for every ordinal $\beta' > \omega$ is identical, so here ω is the closure ordinal.

Example 3 (Summation of Two Natural Numbers)**.** The definition of summation of two natural numbers is a bit more tedious:

$$\begin{array}{lll} \mu U.\lambda X : \mathsf{nat}.\lambda Y : \mathsf{nat}.\lambda Z : \mathsf{nat}. & & \\ & X = 0 \wedge Z = Y & \\ \vee & \exists X' : \mathsf{nat}.\exists Z' : \mathsf{nat}.X = X' + 1 \wedge U\ X'\ Y\ Z' \wedge Z = Z' + 1 \end{array}$$

2.4 Logic Conventions

Next a number of notational niceties will be added to the logic.

2.4.1 Modalities

The usual necessity ("box") and possibility ("diamond") modalities of Hennessy-Milner logic [HM80] are in our proof system derived. They abbreviate a formula referring to some transition fixed point predicate $\rightarrow$ such that the type of the predicate is $s \rightarrow s' \rightarrow s \rightarrow \mathsf{prop}$ for some sorts s and s'. Examples of such predicates are given in Section 4.6. As usual applications of the transition predicate to its three arguments t, α, t' will be written in the format $t \xrightarrow{\alpha} t'$.

$$\begin{aligned} [\alpha]\phi &\triangleq \lambda X : s. \forall X' : s. X \xrightarrow{\alpha} X' \Rightarrow \phi\ X' \\ \langle\alpha\rangle\phi &\triangleq \lambda X : s. \exists X' : s. X \xrightarrow{\alpha} X' \wedge \phi\ X' \end{aligned}$$

It is required that the "action" α is in the closed terms of the sort s'. The binding power of these modalities are stronger than disjunction but weaker than negation.

2.4.2 Formula definitions

The syntax for formulas is, as we have found during case studies, not very intuitive for a person exposed to them for the first time. To counter this a number of abbreviations are introduced below. First, formulas may be named, as in the definition $name \triangleq \phi$, and referred to later with their names. In addition the following abbreviations are recognised:

- A definition on the form $n : s_\phi \Leftarrow \psi$, later referrable to by n, abbreviates the formula $\mu U : s_\phi.\psi\{U/n\}$, where U is assumed to be free in ψ and $\psi\{N/n\}$ replaces n with U in ψ.
- A definition on the form $n : s_\phi \Rightarrow \psi$, later referrable to by n, abbreviates the formula $\nu U : s_\phi.\psi\{U/n\}$. Thus the direction of the arrow, μ or ν, signifies a least or a greatest fixed point.
- Any definition on the form $n : s_\phi. \Leftarrow \lambda X_1 : s_1, \ldots, X_n : s_n.\psi$ such that the outermost operator in ψ is not a lambda abstraction, is considered to abbreviate the formula $n : s_1 \rightarrow \ldots s_n \rightarrow s_\phi. \Leftarrow \lambda X_1 : s_1, \ldots, X_n : s_n \psi$ (and vice versa for the least fixed point). That is, abstracted variables may be omitted from the type of a fixed point operator.

If the construct $name$ names a fixed point formula $\nu.\phi$ (or $\mu.\phi$) let $name^\kappa$ refer to the above fixed point approximated with κ. Further note that the order of listing such fixed point definitions is relevant to determine nesting.

2.4.3 Lifting of Abstractions

Unfortunately the introduction of the modalities gives rise to rather ugly looking formulas. For example, the classical looking formula $\langle\alpha\rangle$*true* does not typecheck since $\vdash$ *true* : $s \rightarrow \mathsf{prop}$ is not provable for any sort s. Such a formula must

instead be written $\langle\alpha\rangle\,(\lambda X : s.\mathit{true})$. Similarly $[\alpha]\,(\langle\alpha\rangle\mathit{true} \wedge \langle\alpha\rangle\langle\alpha\rangle\mathit{false})$ becomes $[\alpha]\lambda X\,:\,s.\,((\langle\alpha\rangle\lambda X : s.\mathit{true})\;\;X \wedge (\langle\alpha\rangle\langle\alpha\rangle\lambda X : s.\mathit{false})\;\;X)$. To counter such ugliness we permit abstractions to be silently lifted, or introduced, so that the syntax $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ abbreviates $\langle\alpha\rangle\mathit{lift}(\phi)$ and $[\alpha]\mathit{lift}(\phi)$ respectively, where *lift* is defined:

$$\begin{array}{rcl}
\mathit{lift}(t_1 = t_2) & = & \lambda X : s.t_1 = t_2 \\
\mathit{lift}(\psi_1 \vee \psi_2) & = & \lambda X : s.\mathit{lift}(\psi_1)\;X \vee \mathit{lift}(\psi_2)\;X \\
\mathit{lift}(\neg\psi) & = & \lambda X : s.\neg(\mathit{lift}(\psi)\;X) \\
\mathit{lift}(\exists X' : S'.\psi) & = & \lambda X : s.\exists X' : S'.\mathit{lift}(\psi)\;X \\
\mathit{lift}(\psi) & = & \psi \text{ for all other formulas } \psi
\end{array}$$

X is assumed to be fresh in $\psi, \psi_1, \psi_2, t, t_1, t_2$.

2.4.4 Formula Macros

Although the restriction to first-order will remain, in the concrete syntax formula abstractions and formula parameters are permitted via a simple macro facility. The macro mechanism is illustrated in a prototypical definition of *always* such that the predicate holds if its formula argument ϕ holds in every reachable state. The types erlangSystem and erlangSysAction are defined in Chapter 3.

$$\begin{array}{l}
\mathit{always}\;[\phi] \triangleq \\
\quad \nu X : \mathsf{erlangSystem} \rightarrow \mathsf{prop}. \\
\quad\quad (\forall A : \mathsf{erlangSysAction}.[A]X) \wedge \phi
\end{array}$$

Assuming this definition the formula $t : \mathit{always}\;[\langle\alpha\rangle\mathit{true}]$ expresses the statement that the term t has an transition labelled by α enabled in every reachable state.

2.4.5 Parametric Actions

A common extension to the modal μ-calculus is to to permit sets of actions in the modalities, e.g., $\langle K\rangle\phi$ and $[K]\phi$ where K is a set of actions.

In our logic this extension can be modelled as follows. Let K be either a finite enumeration of actions $\{\alpha_1, \ldots, \alpha_n\}$ or the complement of a finite enumeration $\backslash\{\alpha_1, \ldots, \alpha_n\}$. The mapping from the extended syntax, with action sets, to the core logic is as follows:

$$\begin{array}{rcl}
\langle\{\alpha_1, \ldots, \alpha_n\}\rangle\phi & \triangleq & \langle\alpha_1\rangle\phi \vee \ldots \vee \langle\alpha_n\rangle\phi \\
\langle\backslash\{\alpha_1, \ldots, \alpha_n\}\rangle\phi & \triangleq & \exists\alpha.\,((\alpha \neq \alpha_1 \wedge \ldots \wedge \alpha \neq \alpha_n) \wedge \langle\alpha\rangle\phi) \\
[\{\alpha_1, \ldots, \alpha_n\}]\phi & \triangleq & [\alpha_1]\phi \vee \ldots \vee [\alpha_n]\phi \\
[\backslash\{\alpha_1, \ldots, \alpha_n\}]\phi & \triangleq & \forall\alpha.\,((\alpha \neq \alpha_1 \wedge \ldots \wedge \alpha \neq \alpha_n) \Rightarrow [\alpha]\phi)
\end{array}$$

2.4.6 Weak Modalities

The weak modalities abstract away from the exact number of internal actions (actions that are, in some sense, not visible to observers of the program) necessary or possible, are well-known from Hennessy-Milner logic. Below it is assumed that a particular action τ represents all the internal actions. The weak modalities are then definable as:

$$\begin{array}{rcll} \langle\langle\ \rangle\rangle\phi & \triangleq & \mu X.\,(\langle\tau\rangle X \vee \phi) & \\ [[\ ]]\phi & \triangleq & \nu X.\,([\tau] X \wedge \phi) & \\ \langle\langle\alpha\rangle\rangle\phi & \triangleq & \langle\langle\ \rangle\rangle\langle\alpha\rangle\langle\langle\ \rangle\rangle\phi & \alpha \neq \tau \\ [[\alpha]]\phi & \triangleq & [[\ ]][\alpha][[\ ]]\phi & \alpha \neq \tau \end{array}$$

For example, "$\langle\langle\alpha\rangle\rangle\phi$", where $\alpha \neq \tau$, intuitively expresses that eventually the action α becomes possible within a finite number of steps.

Chapter 3

A Formal Semantics of ERLANG

In this chapter a small step operational semantics for the ERLANG programming language is developed. The point of this effort is to provide a semantics that is useful for reasoning about non-trivial ERLANG programs, or classes of ERLANG programs, by means of a proof system, and supported in a proof assistant tool. The aim is not to create an authoritative definition of the ERLANG language – the language is to a very large degree defined by its implementations.

First, the syntax of the ERLANG version considered in this thesis is defined in Section 3.1, and the intuitive meaning of its constructs is explained. Then, in Section 3.2, a formal semantics for the dynamic behaviour of ERLANG constructs is developed. In Section 4.6 an embedding of the semantics in a proof system is considered.

3.1 An ERLANG Subset

This section introduces the variant of the ERLANG language for which a formal semantics is presented in Section 3.2.

The differences between the formalised fragment, which henceforth will formally be referred to as ERLANG–F, and more standard versions, e.g., ERLANG 4.7, will be enumerated in Section 3.1.14. In this thesis future references to ERLANG will refer to ERLANG–F, unless otherwise indicated. With "regular ERLANG" we will refer collectively to existing implementations of ERLANG variants excluding the "Core Erlang" initiative [CGJ+00].

The embedding of the syntax and semantics of the language in a proof assistant tool is discussed in Section 4.6. In this chapter we will assume a concrete such syntactic representation (a set of types) of ERLANG constructs (expressions, processes, variables, etc.), and will assume the existence of predicates to check membership of these constructs in subtypes. These types and predicates are defined formally in Section 4.6.

sign	::=	`+` | `-`	
digit	::=	`0` | ... | `9`	
uppercase	::=	`A` | ... | `Z`	
lowercase	::=	`a` | ... | `z`	
digitletter	::=	*digit* | *uppercase* | *lowercase* | `_` | `@`	
number	::=	`[`*sign*`]` *digit*$^+$	
unquotedatom	::=	*lowercase* *digitletter**	
quotedatom	::=	`'` (*inputcharacter* | *escape*)$^+$ `'`	
atom	::=	*unquotedatom* | *quotedatom*	
var	::=	*uppercase* *digitletter**	
expression (e)	::=	*bv* | `[`e_1 `|` e_2`]` | `{`e_1`,` ... `,`e_n`}`	$n > 0$
	|	*var*	
	|	e `(`e_1`,` ... `,`e_n`)`	$n \geq 0$
	|	`case` e `of` m `end`	
	|	`exiting` e	
	|	`try` e `catch` m `end`	
	|	`receive` m [`after` e `->` e'] `end`	
	|	e_1 `!` e_2	
basicvalue (bv)	::=	*atom* | *number* | *pid* | `[]` | `{}`	
value (v)	::=	*bv* | `[`v_1 `|` v_2`]` | `{`v_1`,` ... `,`v_n`}`	$n > 0$
pattern (p)	::=	*bv* | *var* | `[`p_1 `|` p_2`]` | `{`p_1`,` ... `,`p_n`}`	$n > 0$
match (m)	::=	p_1 `when` g_1 `->` e_1`;` $\cdots$ `;`p_n `when` g_1 `->` e_n	$n > 0$
guard (g)	::=	ge_1`,` ... `,`ge_n	$n > 0$

Table 3.1: ERLANG syntax

3.1.1 ERLANG Syntax

Backus-Naur Form (BNF) will normally be used to indicate syntactic sets. Apart from the standard constructs, the construct $[f_1 \ldots f_n]$ in a production will be used to indicate that $f_1 \ldots f_n$ is an optional part of the production. Further repeated patterns will be expressed with a dot notation, e.g., $\{e_1, \ldots, e_n\}$ expresses a non-zero number of (ERLANG) expressions separated by comma symbols and enclosed in braces. Let e^+ and e^* indicate a nonzero, and an arbitrary, number of expressions respectively.

The Erlang constructs are summarised in the BNF grammar found in Table 3.1. In the definition of the syntax of the ERLANG constructs the tokens (e.g., `1`, `2`, ..., `9`) are written in a typewriter font. Meta variables, ranging over some syntactic domain, are written in italics with optional indices (e.g., e and v_1). In the production for *quotedatom* the valid input characters *inputchars* (e.g., including all digits and upper and lower case letters) and the characters escapes (e.g., `\n` for the line feed character)

escape are referred to; their concrete representation is defined in, for example, Barklund et al. [BV99].

In the following the notation $op(e_1, \ldots, e_n)$ (and analogously for values v and patterns p) will be used to represent, schematically, the set expressions (or values, or patterns) that can be constructed using the basic value constructors or cons `[·|·]` and tupling $\{\cdots\}$, with the subterms $e_1, \ldots, e_n$.

3.1.2 Values

The ERLANG Values, formally a subtype of the ERLANG expressions introduced in the next section, ranged over by $v \in \mathsf{erlangValue}$, are either basic or compound.

Basic Values The basic values of ERLANG, recognised by the nonterminal *basicvalue*, consists of the atoms ranged over by $a \in \mathsf{erlangAtom}$, integers ranged over by $i \in \mathsf{erlangInt}$, the process identifiers ranged over by *pid* $\in \mathsf{erlangPid}$ (fresh ones which are created using the built-in function `spawn`), the nil constant `[]` and the empty tuple $\{\}$.

The unquoted atoms, which always start with a lower case letter (e.g., `thesis`), are recognised by *unquotedatom*. The quoted atoms may begin with any character, e.g. `'Thesis'`, and are recognised by *quotedatom*. The integer numbers consist of a sequence of digits (*number*) optionally preceded by a sign. The boolean subtype of the atoms consists of the atoms `true` and `false`, and is ranged over by $b \in \mathsf{erlangBool}$.

To recognise subtypes of the basic values the predicates *isErlangAtom*, *isErlangInt*, *isErlangPid* and *isErlangBoolean* are assumed.

Compound Values The compound values are the non-empty tuples and conses (`[`v_1`|`v_2`]`). A list, an element of the erlangProperList subtype of the values, is either the nil constant, or a sequence of conses such that the second value (v_2) in any cons is itself a list. The syntax `[`v_1`,...,`v_n`]` is a shorthand for the list `[`v_1`|...|[`v_n`|[]]...]`.

3.1.3 Expressions, Variables, Patterns and Matches

The ERLANG expressions, recognised by the nonterminal *expression*, are ranged over by $e \in \mathsf{erlangExpression}$.

Variables are recognised by the nonterminal *var*, and we let $V \in \mathsf{erlangVar}$ range over them. Any occurrence of an anonymous variable `_` in a pattern is considered a shorthand for a fresh variable (a variable not occuring elsewhere in the pattern, or in the enclosing expression). For instance,

```
case test(2) of {_,_} -> true; _ -> false end
```

is a shorthand for the expression

```
case test(2) of {X1,X2} -> true; X3 -> false end
```

The ERLANG *patterns* $p \in \mathsf{erlangPattern}$, constructed as values but where variables may occur as place holders, are recognised by the *pattern* nonterminal and *matches* $m \in \mathsf{erlangMatch}$. A match is a sequences of *clauses* and recognised by *match*. A clause is a triple consisting of a pattern, guard, and an expression. In the following a clause p `->` e without guard will be understood to abbreviates the clause p `when true ->` e with the trivially true guard `true`. Further syntactic restrictions concerning guards will be explained in Section 3.1.6.

3.1.4 Functions

Expressions are interpreted relative to an environment of function definitions

$$\begin{array}{l} a\,\texttt{(}p_{11}\texttt{, ..., }p_{1n}\texttt{) when }\ g_1 \texttt{ -> } e_1\texttt{;} \\ \qquad\qquad \ldots \\ a\,\texttt{(}p_{k1}\texttt{, ..., }p_{kn}\texttt{) when }\ g_k \texttt{ -> } e_k\texttt{.} \end{array}$$

where the number of defining clauses is greater than zero ($k > 0$), whereas the number of function arguments in any clause may be zero (for any i, $in \geq 0$). Alternatively a function definition can be written as $a \triangleq m$, where the match m is:

$$\begin{array}{l} \{p_{11}\texttt{, ..., }p_{1n}\}\texttt{ when }\ g_1 \texttt{ -> } e_1\texttt{;} \\ \qquad\qquad \ldots \\ \{p_{k1}\texttt{, ..., }p_{kn}\}\texttt{ when }\ g_k \texttt{ -> } e_k \end{array}$$

It is required that any variable occurring in the guard (g_i) or body (e_i) part of a clause must also occur, and thus be bound, in the corresponding pattern. This condition can be formalised as the formula $fv(g_k) \cup fv(e_k) \subseteq \bigcup_i fv(p_{ki})$, where the *fv* function (see Page 43) returns the free variables in expressions and patterns.

3.1.5 Built-in Functions

In addition to the explicitly defined functions a number of built-in functions are available. For instance the `hd` and `tl` functions decompose lists, the `spawn` function spawns new processes, and the `throw` and `exit` functions are used to terminate the execution of a process. The built-in functions of ERLANG–F are enumerated in Table 3.2.

In regular ERLANG terminology a number of these built-in functions are considered to be *operators*. That is, that the application of an operator to its arguments (operands) should be written in infix notation, e.g., `1+2`. Here we do not differentiate between operators and other built-in functions, but still permit, when there is no risk for confusion, application of functions considered operators in regular ERLANG to be written using infix notation.

In the following we let f range over the subset of atoms that name an explicitly defined ERLANG function (Section 3.1.4) or a built-in function (Section 3.1.5).

Relational Operators	>, <, =:=, =/=, =<, >=, ==, /=
Logical Operators	and, not, or, xor
Arithmetic Operators	+, −, ⋆, / div, rem, bsl, bsr, band, bnot, bor, bxor
Type Recognisers	integer, pid, atom, list, tuple, constant, number
Destructors	hd, tl, element
Miscellaneous	++, --, abs, setelement, size, length
Side Effects	self, spawn, spawn_link, unlink, exit, throw, process_flag

Table 3.2: Built-in Functions

3.1.6 Guards

Guards, ranged over by $g \in$ erlangGuard, and recognised by *guard* in Table 3.1, are sequences of guard expressions (ge) that compute boolean conditions. Each guard expression may contain only function applications of certain well-behaved functions that are known to compute without relying on side effects - except possibly by raising exceptions - and such that the execution of each function application takes a bounded amount of time. The exact definition of what constitutes a side effects is found in Section 3.1.10.

More stringently, the top level expression of a guard expression ge is either the application of a relational operator,e.g. >, a logical operator, e.g. and, or a type recogniser, e.g. pid, to proper guard expression arguments. A proper guard expression argument is either a value, or the application of a function listed in Table 3.2, omitting the functions noted under **Side Effects**, e.g. spawn, to proper guard expression arguments.

3.1.7 Processes, Messages, Mailboxes and Links

An ERLANG *process* is a container for expression evaluation which is named by its unique process identifier (pid). Communication is always binary, with one process sending a message to a second process identified by its process identifier. Messages sent to a process are put in its mailbox, also known as a queue. Informally a mailbox is a sequence of values ordered by their arrival time. Mailboxes can store any number of messages. In the following q ranges over such sequences of ERLANG values.

Interestingly, processes can be linked together in order to detect and recover from abnormal process termination. Such process links are always bidirectional but the treatment of process termination notifications may differ between the two parties.

3.1.8 Systems

An ERLANG *system* is simply a collection of ERLANG processes.

3.1.9 Intuitive Semantics

The intuitive behaviour of the ERLANG expressions, in the context of a process with a pid *pid* and a queue q, is explained below.

An ERLANG expression either completes normally, returning a value as the result of its computation, or an exception is raised. Exceptions are caused by program errors such as typing violations or they are invoked explicitly in an `exiting` expression.

- To evaluate a cons `[`e_1`|`e_2`]` evaluate first e_1 and then e_2. To evaluate a tuple $\{e_1, \ldots, e_n\}$ the subexpressions e_1 to e_n are evaluated in left-to-right order.
- e`(`e_1`,...,`e_n`)` is function application. First e is evaluated, yielding a value. Then the parameter list e_1,...,e_n is evaluated in left-to-right order, and the function value is applied to the resulting list of argument values.
- `case` e `of` p_1 `when` g_1 `->` e_1`;` $\cdots$ `;` p_n `when` g_1 `->` e_n `end` is evaluated by first evaluating e to a value v, then matching v against patterns p_i, and checking whether the optional guard expressions g_i evaluate to true. If several clauses match the first one (in left-to-right order) is chosen.
- The expression `exiting` e interrupts normal processing by raising an exception with the value that e computes to.
- In the evaluation of `try` e `catch` m `end` first the expression e is evaluated. If this evaluation completes normally in a value v, then the whole `try` expression completes normally with that value. If an exception is raised during evaluation of e then the value of the exception is matched against the patterns in m, as if the expression `case` v `of` m `end` was executed.
- 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 the process corresponding to *pid*′, resulting in v as the value of the send expression.
- `receive` m `after` e_1 `->` e_2 `end` first computes the timeout value e_1, and then inspects the process mailbox q and retrieves the first element in q that matches any pattern of m. Once such an element v has been found, it is matched sequentially against the patterns in m. Alternatively, if the timeout deadline e_1 is met first, evaluation proceeds with the expression e_2.

3.1.10 Built-in Functions with Side Effects

Intuitively a side effect is any act by an expression that depends on, or alters, the process state in which the expression executes. A call to the `self` function, for instance, is considered a side effect since the result of the call depends on the process context of the

expression. In the section on formal semantics the side effect notion will be formally defined. Below the list of built-in functions with side effects are enumerated.

- `self()` returns the process identifier of the process in which the expression executes.
- `spawn`(f, v) creates a new process, with a unique pid and an initially empty mailbox, executing the function f with argument list in v. The process identifier of the newly spawned process is returned.
- `link`(*pid*) creates a bidirectional link between the process executing the function call and the process referenced through the process identifier *pid*, and *pid* is returned. Such a link is used to notify a process when its linked process has terminated its execution.
- `unlink`(*pid*) deletes a bidirectional link between the process executing the function call and the process referenced through the process identifier *pid*. The process identifier *pid* is returned.
- `spawn_link`(f, v) has the same effect as executing `link`(`spawn`(f, v)), except that it is atomic with respect to actions by other processes.
- `process_flag`(v_1, v_2), when v_1 is the atom `trap_exit` and v_2 is a boolean, modifies the handling of notification of process termination to linked processes, as made precise in the formal semantics. Intuitively, if a process terminates abnormally then any linked processes will also terminate abnormally. A call to `process_flag` by a linked process modifies this behaviour so that instead a message indicating termination is stored in the mailbox of the linked process.
- `exit`(v) terminates the execution of the process executing the function call, by raising an exception, unless the exception is handled in a surrounding `try` expression.
- `kill`(*pid*, v) terminates the execution of the process *pid* with the reason for termination specified by v.

3.1.11 Built-in Functions without Side Effects

The rest of the built-in functions in Table 3.2 have no side effects. Informally, the binary operators listed under **Relational Operators** compare their arguments under varying relations. For instance, =:= is exact equality, =/= is exact inequality, == and /= coincide with =:= and =/=. The latter relations coerce their arguments; since floats is not included in ERLANG–F they coincide with the exact relations. The ERLANG values are totally ordered, e.g., a call to $<$ with value arguments will always return either `true` or `false` as a result. The ordering is, in ascending order: numbers, atoms, process identifiers, tuples and conses. Conses are ordered pointwise by elements. Tuples are ordered by size, and then pointwise by elements.

The **Logical Operators** operate on booleans and are, of course, strict (both arguments are always evaluated). The **Arithmetic Operators** are standard. The operators `bsl`, `bsr`, `band`, `bnot`, `bor` and `bxor` are binary, and perform bitwise operations on their integer arguments.

The `hd` and `tl` are standard cons destructors, whereas `element`(v_1, v_2) returns the element numbered v_1 of the tuple v_2. Tuple elements are numbered starting from 1. `++` and `--` are list append and subtractions. `size` returns the lenght of a tuple, and `length` the length of a list. Finally `setelement`(v_1, v_2, v_3) updates the v_1 member of the tuple v_2 with the value v_3, and returns the result.

3.1.12 Shorthands

A number of language constructs are defined below in terms of other ERLANG constructs, thus not requiring a separate account in the operational semantics. A mapping $\|\, e \,\|$ of an expression e in extended ERLANG syntax to the core language is given below.

- A sequence of expressions (a body) enclosed within `begin` and `end` keywords – `begin` e_1`,` $\ldots$`,` e_n `end` ($n > 0$) – abbreviates a nested `case` statement:

$$
\begin{array}{ll}
\|\texttt{begin}\ e\ \texttt{end}\| & \triangleq \quad \|\, e \,\| \\
\|\texttt{begin}\ p = e\texttt{,}\ e_2\texttt{,}\ldots\texttt{,}\ e_n\ \texttt{end}\| & \triangleq \\
\quad \texttt{case}\ e\ \texttt{of}\ p\ \texttt{->}\ \|\texttt{begin}\ e_2\texttt{,}\ldots\texttt{,}\ e_n\ \texttt{end}\|\ \texttt{end} & \\
\|\texttt{begin}\ e_1\texttt{,}\ e_2\texttt{,}\ldots\texttt{,}\ e_n\ \texttt{end}\| & \triangleq \\
\quad \texttt{case}\ e_1\ \texttt{of}\ \_\ \texttt{->}\ \|\texttt{begin}\ e_2\texttt{,}\ldots\texttt{,}\ e_n\ \texttt{end}\|\ \texttt{end} & \\
\quad \text{if } e \text{ is not an assignment } p{=}e' &
\end{array}
$$

  Following regular ERLANG the enclosing `begin end` pair around bodies can be omitted in unambiguous contexts, e.g., in matches.

- The expression `try` e `end` is translated into `try` $\|\, e \,\|$ `catch V -> V end` where `V` is a fresh variable, i.e., a variable that does not otherwise occur in the context of the `try` construct. Similarly, the expression `catch` e is translated into the expression

```
try ‖e‖
catch
  {'THROW',V} -> V;
  {'EXIT',V} -> {'EXIT',V}
end
```

- The expression

$$\texttt{if}\ g_1\ \texttt{->}\ e_1;\ \cdots\ ;\ g_n\ \texttt{->}\ e_n\ \texttt{end}$$

is a sequential choice construct from regular ERLANG such that the evaluation proceeds with the first expression e_i for which the corresponding guard g_i evaluates to `true`, and is translated into

```
case true of
  _ when g1 -> ‖ e1 ‖;
  ⋮
  _ when gn -> ‖ en ‖;
  _ -> exiting {'EXIT',if_clause}
end
```

- The expression

  `cond` e_1 `->` e_1'; $\cdots$; e_n `->` e_n' `end`

  is a generalisation of the `if` construct in STANDARD ERLANG. Rather than allowing only guards to select conditional branches, as in the `if` statement, arbitrary expressions are permitted. The construct is translated into the expression

```
case ‖ e1 ‖ of
  true -> ‖ e1' ‖;
  false ->
    ⋮
    case ‖ en ‖ of
      true -> ‖ en' ‖;
      false -> exiting {'EXIT',cond_clause};
      _ -> exiting {'EXIT',badbool}
    end;
   ⋮
  _ -> exiting {'EXIT',badbool}
end
```

- The sequential functional assignment `let` expression is borrowed from conventional functional programming languages:

  `let` p_1=e_1, $\cdots$, p_n=e_n `in` e `end`

  is translated to

  `case` ‖ e_1 ‖ `of` p_1 `->` `...` `case` ‖ e_n ‖ `of` p_n `->` ‖ e ‖ `end` `...` `end`

3.1.13 Throw and Exit Functions

The `throw` and `exit/1` functions are normally assumed to be built-in functions. Here, in contrast, these functions are considered to be explicitly defined using the new `exiting` expression, and the new built-in function `kill`.

```
throw(V)  -> exiting {'THROW',V}.
exit(V)   -> exiting {'EXIT',V};
exit(P,V) -> kill(P,V).
```

Example 4 (RPC)**.** Below we sketch a simple RPC service in ERLANG.

```
server() ->
  receive
    {req, F, Pid} -> spawn(exec, [F,Pid]), server()
  end.

exec(F,Pid) -> Pid!F().
```

ERLANG variables always have an initial upper-case character (`F` and `Pid`) while atoms begin with a lower-case character (`req`, `server` and `spawn`). The `server` function is continuously prepared to receive tuples containing the name of a function `F` and a process identifier `Pid`. It then spawns a new process evaluating the `exec` function, which simply invokes the received function `F` and sends any result to the process addressed by the process identifier `Pid`.

Since ERLANG is an untyped language a possible outcome of sending a wrongly typed message to the server process is that the newly spawned process will terminate due to a runtime error. In ERLANG the treatment of process termination is included in the language itself. For instance, the server process can elect to also be terminated abnormally whenever one of its spawned processes terminates abnormally by modifying the line

```
{req, F, Pid} -> spawn(exec, [F,Pid]), server()
```

to read instead

```
{req, F, Pid} -> spawn_link(exec, [F,Pid]), server()
```

Instead of being terminated, the server process can choose to receive a message in its mailbox, indicating the termination of the linked process, using the `process_flag` built-in function. As an example, the revised server process below attempts to restart processes that terminated abnormally a fixed number (`NumRestarts`) of times:

```
server(NumRestarts) ->
  process_flag(trap_exit,true),
  server([],NumRestarts);

server(Clients,NumRestarts) ->
```

```
  receive
    {req,F,AnswerPid} ->
      server
      ([startClient(F,AnswerPid,NumRestarts)|Clients],
       NumRestarts);
    {'EXIT',Pid,Status} ->
      {{F,AnswerPid,RestartCount},NewClients} =
        getClient(Pid,Clients,[]),
      if
        Status =/= normal, RestartCount > 0 ->
          server
          ([startClient(F,AnswerPid,RestartCount-1)|
            NewClients], NumRestarts);
        true ->
          server(NewClients,NumRestarts)
      end
  end.

startClient(F,AnswerPid,NumRestarts) ->
  {spawn_link(exec,[F,AnswerPid]),
   {F,AnswerPid,NumRestarts}}.

getClient(Pid, [{Pid,Record}|Rest], Others) ->
  {Record, Rest++Others};
getClient(Pid, [Client|Rest], Others) ->
  getClient(Pid, Rest, [Client|Others]).
```

Note above that the `getClient` function does not address the case when its second argument is the empty list `[]`. This is not necessarily considered a case of bad programming practice in the ERLANG community. Rather than viewing program errors causing abnormal process termination as something disastrous, the attitude is that process termination does occur, for various reasons, and what is important is that there exists mechanisms (process linking) to recover gracefully from such situations. In larger software projects utilising ERLANG a typical application structure is to build so called *supervision trees* where a process on one level is responsible for supervising its child processes, possibly restarting them if they terminate abnormally. In this way failures ripple through the supervisor structure, restarting processes along the way, until some higher-level process is able to recover from the error condition.

3.1.14 A Comparison with other ERLANG Versions

The major differences between current ERLANG implementations and the fragment formalised in the thesis (ERLANG–F) are summarised in this section.

Care is taken to point out the limitations and deviations compared to more standard ERLANG versions. In particular we will frequently refer to ERLANG 4.7 and the

proposed STANDARD ERLANG version, since good natural language specification exists for both variants [Bar98, BV99]. A notable initiative to create an intermediate representation (between source code and intermediate code in compilers) of ERLANG, named "Core Erlang", is reported in [CGJ+00].

Modules Modules are not present in ERLANG–F. That is, a flat function name space is assumed. Module names cannot be specified in function calls, and built-in functions such as e.g. `spawn` do not accept a module parameter.

Nodes There are no nodes. That is, the distribution aspect of ERLANG, where processes are mapped onto nodes, is missing. On the other hand the semantics makes no guarantees about timely and deterministic arrivals of messages.

Real-Time There is no concept of real time in the semantics. Although the `after` clause in the `receive` construct is supported its semantics is non-standard, a timeout can be triggered non-deterministically at any time.

Assignments The semantics of assignments is non-standard: assignments have no effect outside the scope of the subexpression in which they occur, with one significant exception. The body $p = e, e_1, \ldots, e_n$ is considered to abbreviate a `case` expression `case` e `of` p `->` $e_1, \ldots, e_n$ `end`, thus permitting analysis of the large body of ERLANG code where the only out-of-scope effect of assignments are towards the tail of the body wherein the assignment occurs.

Fixed Evaluation Ordering In agreement with STANDARD ERLANG but in contrast to ERLANG 4.7 the formal semantics prescribes a left-to-right evaluation ordering of subexpressions. For example, arguments in function calls and tuple members are always evaluated in left-to-right order.

Values Missing ERLANG value constructors compared to ERLANG 4.7 are the floats, refs, binaries and ports. Anonymous functions, constructed using the `fun` expression (similar to a lambda expression in mainstream functional programming), is considered an extension to the semantics and is discussed in Section 3.2.7.

Compound data values not covered in the thesis but present in regular ERLANG are the characters, records and strings, although handling these data types in the formal semantics would likely present little difficulty.

Expressions Nonstandard expressions include the `try` and `exiting` expressions. `try` is fetched from the STANDARD ERLANG proposal. The `exiting` expression is new, and has been introduced to simplify the semantic treatment of the `exit` and `throw` functions. Missing compared to regular ERLANG is the `catch` expression; however its behaviour can largely be emulated using the `try` expression. Also not covered in the thesis is the list comprehension expression.

Functions Contrary to regular ERLANG a function is in this thesis not considered to have a fixed arity. This choice is, we believe, natural since function application

can then be defined in terms of matching (analogously to the semantics of the `catch` expression). As a consequence of this we have for instance renamed the regular ERLANG function `exit/2` as `kill` in ERLANG–F.

Some of the functions listed as built-in in Table 3.2 are not considered true built-in functions in Erlang version 4.7 but are permitted only inside guards. These are the type recognisers which include, for instance, `atom` and `integer`. In ERLANG–F there is no class of built-in functions only permitted in guards.

Finally of course the selection of built-in functions is rather arbitrary.

Guard Functions Missing compared with regular ERLANG is mainly the node built-in functions and the built-in function `self` which is excluded to simplify the semantic account of guards. The reason for the omission is that the semantics distinguishes between the functions which do not depend on the process context in which they compute, and the functions that depend, in some way, on the process context, like `self`.

Process State In the treatment of processes much of the internal state (static and dynamic properties) of a regular ERLANG processes has not been included in ERLANG–F. For instance:

- There are no process dictionaries associated with processes.
- A process does not have a process group leader; and processes are not grouped into process groups.
- Processes cannot be registered with a name, communication with a process is possible only when its process identifier is known.

In addition there are numerous minor syntactical differences. These include, for instance, the names of exceptions raised. For example, due to a change in function application semantics a non-standard exception is raised when a function is applied to parameters that do not match.

The chief reason for sacrificing exact compatibility in this thesis with current ERLANG implementations is to reduce the number of language constructs that need special treatment in the semantics. This reduction is crucially important when, as is the case in this work, the end goal is to use the semantics for analysis.

Our ambition, although not realised yet, is to develop a translator capable of transforming a substantial body of ERLANG programs into programs conforming to our variant. Already there are example of such ERLANG to ERLANG translators that are useful in our verification studies. For instance, the HiPE ERLANG compiler [JPS00] contains a stage where the constructs in the full ERLANG language are translated into Core Erlang, for instance transforming records and operations on records into tuples and operations on tuples. In another work Arts and Benac Earle [AB01] transform ERLANG programs containing calls to higher-order functions such as `map` into specialised first-order functions.

3.2 A Formal Semantics of ERLANG

In this section an operational semantics for the fragment of ERLANG introduced in the preceding section is developed. The main contribution of the semantics is that a clean account of a relatively complex language is achieved. It addresses topics like error-recovery, which has often been considered outside the scope of programming languages and in the realm of operating systems. As a result our semantics has become hierarchical; at one level regulating classical function evaluation, while another level focuses on interprocess concerns.

The resulting semantics clearly exposes the interfaces between these different levels. In particular it avoids speaking about contexts (of an evaluating expression, or a process) but rather attaches meanings to objects (expressions, processes) independently of the context in which they occur. This approach to a semantics is undoubtedly coloured by the proof system view: there open systems are considered with variables ranging over expressions and processes, whose behaviour is only partially known. In addition we believe that such a "context-free" approach to a programming language semantics naturally lends itself to possibilities for compositional reasoning.

The basic computing entities in the language are processes, supporting asynchronous message passing. Processes are addressed through their unique process identifiers. These identifiers can be communicated and so like in the π-calculus [MPW92] communication capabilities can migrate from one process to another. Additionally function names are first-class data objects which can also be freely communicated. However, compared to names in the π-calculus process identifiers are not very abstract objects. It is for instance possible for processes to obtain process identifiers not communicated to them, using a built-in function (not covered in the formal semantics here) that returns the process identifiers in use. However, clearly the overwhelming majority of ERLANG programs do not employ such "dirty tricks" and we could choose to present an operational semantics for the class of well-behaved programs that, like the π-calculus semantics, carefully track how the knowledge of process identifiers migrate in a system via the introduction of a restriction operator. In this thesis, however, a semantics is presented where the only requirements placed on process identifier creation is that process identifiers of distinct processes are different.

3.2.1 Preliminaries

First the notions of free variables, variables that occur in patterns, and substitution in ERLANG is considered in consecutive definitions.

Definition 11. *The function $fv(e)$, which calculates the free* ERLANG *variables of an expression, a guard, or a match, is defined in Table 3.3.*

Definition 12. *The function $pv(t)$ which computes the set of* ERLANG *variables that occur in patterns in t is defined using fv in Table 3.4.*

Next the concept of substitution of values for variables is defined. The syntax $\widetilde{v}$ is used to denote a vector of terms, here ERLANG values.

$$
\begin{array}{rcl}
fv(op(e_1, \ldots, e_n)) & \triangleq & \bigcup_{1 \leq i \leq n} fv(e_i) \\
fv(V) & \triangleq & \{V\} \\
fv(e(e_1, \ldots, e_n)) & \triangleq & fv(e) \cup \bigcup_{1 \leq i \leq n} fv(e_i) \\
fv(\texttt{case}\ e\ \texttt{of}\ m\ \texttt{end}) & \triangleq & fv(e) \cup fv(m) \\
fv(\texttt{exiting}\ e) & \triangleq & fv(e) \\
fv(\texttt{try}\ e\ \texttt{catch}\ m\ \texttt{end}) & \triangleq & fv(e) \cup fv(m) \\
fv(\texttt{receive}\ m\ \texttt{after}\ e\ \texttt{->}\ e'\ \texttt{end}) & \triangleq & fv(e) \cup fv(e') \cup fv(m) \\
fv(e_1\,!\,e_2) & \triangleq & fv(e_1) \cup fv(e_2) \\
 & & \\
fv(ge_1, \ldots, ge_n) & \triangleq & \bigcup_{1 \leq i \leq n} fv(ge_i) \\
 & & \\
fv\left(\begin{array}{c} p_1\ \texttt{when}\ \ g_1\ \texttt{->}\ e_1; \\ \ldots \\ p_n\ \texttt{when}\ \ g_n\ \texttt{->}\ e_n \end{array}\right) & \triangleq & \bigcup_{1 \leq i \leq n} (fv(e_i) \cup fv(g_i)) \setminus fv(p_i)
\end{array}
$$

Table 3.3: Free Variables in Expressions, Guards and Matches

$$
\begin{array}{rcl}
pv(op(e_1,\ldots,e_n)) & \triangleq & \bigcup_{1\leq i\leq n} pv(e_i) \\
pv(V) & \triangleq & \emptyset \\
pv(e(e_1,\ldots,e_n)) & \triangleq & pv(e) \cup \bigcup_{1\leq i\leq n} pv(e_i) \\
pv(\texttt{case}\ e\ \texttt{of}\ m\ \texttt{end}) & \triangleq & pv(e) \cup pv(m) \\
pv(\texttt{exiting}\ e) & \triangleq & pv(e) \\
pv(\texttt{try}\ e\ \texttt{catch}\ m\ \texttt{end}) & \triangleq & pv(e) \cup pv(m) \\
pv(\texttt{receive}\ m\ \texttt{after}\ e\ \texttt{->}\ e'\ \texttt{end}) & \triangleq & pv(e) \cup pv(e') \cup pv(m) \\
pv(e_1\,!\,e_2) & \triangleq & pv(e_1) \cup pv(e_2) \\
\\
pv(ge_1,\ldots,ge_n) & \triangleq & \bigcup_{1\leq i\leq n} pv(ge_i) \\
\\
pv\left(\begin{array}{c} p_1\ \texttt{when}\ g_1\ \texttt{->}\ e_1; \\ \ldots \\ p_n\ \texttt{when}\ g_n\ \texttt{->}\ e_n \end{array}\right) & \triangleq & \bigcup_{1\leq i\leq n} (pv(e_i) \cup pv(g_i)) \cup fv(p_i)
\end{array}
$$

Table 3.4: Variables in Patterns

$$
\begin{array}{rcl}
op(e_1,\ldots,e_n)\{\widetilde{v}/\widetilde{V}\} & \triangleq & op(e_1\{\widetilde{v}/\widetilde{V}\},\ldots,e_n\{\widetilde{v}/\widetilde{V}\}) \\
V\{\widetilde{v}/\widetilde{V}\} & \triangleq & v_i \text{ if } V = V_i \text{ for some } i \text{ otherwise } V \\
e(e_1,\ldots,e_n)\{\widetilde{v}/\widetilde{V}\} & \triangleq & e\{\widetilde{v}/\widetilde{V}\}(e_1\{\widetilde{v}/\widetilde{V}\},\ldots,e_n\{\widetilde{v}/\widetilde{V}\}) \\
\texttt{case}\ e\ \texttt{of}\ m\ \texttt{end}\{\widetilde{v}/\widetilde{V}\} & \triangleq & \texttt{case}\ e\{\widetilde{v}/\widetilde{V}\}\ \texttt{of}\ m\{\widetilde{v}/\widetilde{V}\}\ \texttt{end} \\
\texttt{exiting}\ e\{\widetilde{v}/\widetilde{V}\} & \triangleq & \texttt{exiting}\ (e\{\widetilde{v}/\widetilde{V}\}) \\
\texttt{try}\ e\ \texttt{catch}\ m\ \texttt{end}\{\widetilde{v}/\widetilde{V}\} & \triangleq & \texttt{try}\ e\{\widetilde{v}/\widetilde{V}\}\ \texttt{catch}\ m\{\widetilde{v}/\widetilde{V}\}\ \texttt{end} \\
\left(\begin{array}{l}\texttt{receive}\\ \quad m\\ \quad \texttt{after}\ e\ \texttt{->}\ e'\\ \texttt{end}\end{array}\right)[\widetilde{v}/\widetilde{V}] & \triangleq & \begin{array}{l}\texttt{receive}\\ \quad m\{\widetilde{v}/\widetilde{V}\}\\ \quad \texttt{after}\ e\{\widetilde{v}/\widetilde{V}\}\ \texttt{->}\ e'\{\widetilde{v}/\widetilde{V}\}\\ \texttt{end}\end{array} \\
e_1\,!\,e_2\{\widetilde{v}/\widetilde{V}\} & \triangleq & e_1\{\widetilde{v}/\widetilde{V}\}\,!\,e_2\{\widetilde{v}/\widetilde{V}\} \\
ge_1,\ldots,ge_n\{\widetilde{v}/\widetilde{V}\} & \triangleq & ge_1\{\widetilde{v}/\widetilde{V}\},\ldots,ge_n\{\widetilde{v}/\widetilde{V}\} \\
\left(\begin{array}{l}p_1\ \texttt{when}\ \ g_1\ \texttt{->}\ e_1;\\ \quad\ldots\\ p_n\ \texttt{when}\ \ g_n\ \texttt{->}\ e_n\end{array}\right)[\widetilde{v}/\widetilde{V}] & \triangleq & \begin{array}{c}p_1\{\widetilde{v}/\widetilde{V}\}\ \texttt{when}\ g_1\{\widetilde{v}/\widetilde{V}\}\ \texttt{->}\ e_1\{\widetilde{v}/\widetilde{V}\}\\ ;\ \ldots\ ;\\ p_n\{\widetilde{v}/\widetilde{V}\}\ \texttt{when}\ g_n\{\widetilde{v}/\widetilde{V}\}\ \texttt{->}\ e_n\{\widetilde{v}/\widetilde{V}\}\end{array}
\end{array}
$$

Table 3.5: Substitution in Expressions, Guards and Matches

Definition 13. *The substitution function $\{\widetilde{v}/\widetilde{V}\}$, which replaces the* ERLANG *variables in the vector $\widetilde{V}$ with the* ERLANG *values in $\widetilde{v}$ in an expression, a guard or a match, is defined in Table 3.5. It is required that the arities of $\widetilde{v}$ and $\widetilde{V}$ coincide, and that no variable occurs more than once in $\widetilde{V}$.*

Note that $\{\widetilde{v}/\widetilde{V}\}$ is a homomorphism. For a concrete vector of values $v_1,\ldots,v_n$ and variables $V_1,\ldots,V_n$ the notation $e[v_1/V_1,\ldots,v_n/V_n]$ will be used. Further, let the simultaneous substitution of expressions $e_1,\ldots,e_n$ for variables $V_1,\ldots,V_n$ in e be denoted by $e[e_1/V_1,\ldots,e_n/V_n]$.

Note that the definition of substitution is slightly nonstandard from the perspective of functional programming practise, but it is faithful to ERLANG. Since ERLANG lacks proper binding operators a substitution can never be cancelled by a binding operator. Consider for instance the ERLANG expression

```
case 2 of
  X -> case 3 of X -> X end
end
```

The execution of this code fragment will invariably terminate with an exception (`badmatch`) since the evaluation of the outermost `case` expression will bind the variable `X` also in the inner `case` expression. An exception to this binding strategy is represented by the `fun` expression. This is treated in Section 3.2.7. Second, no checks are necessary in the substitution function to prevent variable capture. The exception is again the `fun` expression.

Function Spaces Next we enumerate, but do not provide explicit definitions of, recogniser predicates for different categories of functions. In this thesis the sets of atoms defined by predicates *isProcFun*, *isDefined* and *isExprFun* are distinct.

- The predicate *isDefined* recognises atoms naming explicitly defined functions, such as `throw` and `exit`.
- The predicate *isBIF* recognises atoms corresponding to names of functions which are considered built-in. These are the atoms naming functions in Table 3.2 except `throw` and `exit`.
- The predicate *isExprFun* recognises atoms corresponding to built-in functions which neither depends on, nor modifies, the process state of an expression. These are the atoms naming functions in Table 3.2 that are not listed under the heading side effects.
- The predicate *isGuardFun* recognises the atoms naming a subset of the built-in functions which are permitted to occur in a guard expression. In this thesis *isExprFun* and *isGuardFun* are the same.
- The predicate *isProcFun* recognises atoms corresponding to functions that depend on the process state, enumerated under side effects in Table 3.2.

Queues The semantic rules for communication, both in the expression and system semantics, makes reference to a queue like data structure which is defined below.

Definition 14. *An* ERLANG queue*, ranged over by* $q \in \mathsf{erlangQueue}$*, is a finite sequence of values* $v_1 \cdot v_2 \cdot \ldots \cdot v_n$*, where "*$\epsilon$*" is the empty sequence and "*$\cdot$*" is concatenation.*

3.2.2 Dynamic Semantics

The ERLANG semantics is given as a small-step structural operational semantics [Plo81], in the form of transition rules between structured states.

The semantics matches closely the hierarchic structure of an ERLANG system. First, in Section 3.2.3, the ERLANG expressions are provided with a semantics that does not require any notion of processes. Then the behaviours of ERLANG systems is explored in Section 3.2.5, separated into two cases: (i) a single process constraining the behaviours of an ERLANG expression, and (ii) the parallel composition of two ERLANG systems into a single one.

To keep the presentation readable we will take some liberties with notation. For instance, we use the same transition relation symbol for both ERLANG expression transitions and ERLANG system transitions. In addition the τ action denotes both the ERLANG expression computation step, and the silent action of an ERLANG system.

A transition rule will be written on the format

$$\frac{t_1 \xrightarrow{\alpha_1} t_1' \quad \ldots \quad t_n \xrightarrow{\alpha_n} t_n' \qquad (\phi_1\ t_{1_1} \ldots\ t_{1_k}) \quad \ldots \quad (\phi_m\ t_{m_1} \ldots\ t_{m_k})}{t \xrightarrow{\alpha} t'}$$

where each ϕ_i is a formula of the underlying logic defined in Chapter 2 that does not refer to any transition relation. As described earlier in Section 3.1.4 an environment of function definitions is assumed such that $f \stackrel{\Delta}{=} m$ means that f defines a function with body m in that environment.

3.2.3 Dynamic Semantics of Expressions

Definition 15. *The* expression actions*, ranged over by* $\alpha \in$ erlangExprAction*, are:*

α	$::=$	τ	computation step
	$\mid$	*pid*$!v$	output
	$\mid$	*exiting*(v)	exception
	$\mid$	*read*(q, v)	reading from queue
	$\mid$	*test*(q)	checking queue contents
	$\mid$	$f(v_1, \ldots, v_n) \rightsquigarrow v$	built-in function call

Informally an internal send action written *pid*$!v$ represents the act of sending a message v to the process with process identifier *pid*. The action *exiting*$($*reason*$)$ represents an exception, which unless handled in a `try` expression will result in the termination of the process. The action *read*(q, v) indicates that an ERLANG expression is capable of performing a reduction under the condition that the queue $q \cdot v$ is a prefix of the message queue (of the process in which the expression executes). Intuitively the combination of both q and v in the action expresses that v can be read under the condition that q is a prefix indicating that no pattern in the match can match any value in q. If q were to be omitted from the action too little information would be available at the process level to be able to deduce for a given queue whether v could actually be read. Similarly, the action *test*(q) indicates a capability of performing a reduction under the condition that the incoming message queue is q. Finally, the action $f(v_1, \ldots, v_n) \rightsquigarrow v$ represents a call of a built-in function f, with arguments $v_1, \ldots, v_n$, somehow dependent upon the process state, and returning a result v. As an example, a call to the built-in function `self()` which returns the process identifier of the process in which the call is made, causes as we soon shall see an infinite number of actions of the pattern `self()` $\rightsquigarrow v$, for any value v. The point is that in the second level of the semantics, regulating transitions of ERLANG systems, only the actions with the correct value v will give rise to a transition.

$$
\begin{array}{lll}
r[\cdot] \quad ::= & \cdot & \\
| & op(\ldots, v_{k-1}, r[\cdot], e_{k+1}, \ldots) & k > 0 \\
| & r[\cdot](e_1, \ldots, e_n) & \\
| & v(\ldots, v_{k-1}, r[\cdot], e_{k+1}, \ldots) & k > 0 \\
| & \texttt{case}\ r[\cdot]\ \texttt{of}\ m\ \texttt{end} & \\
| & \texttt{receive}\ m\ \texttt{after}\ r[\cdot]\ \texttt{->}\ e'\ \texttt{end} & \\
| & r[\cdot]!e \mid v!r[\cdot] &
\end{array}
$$

Table 3.6: Reduction Contexts

Reduction Contexts The rules for expression evaluation will be given only for the case when all parameters of an expression construct have been fully evaluated. For defining the subexpression evaluation ordering, from left to right, reduction contexts [FFKD87] are used to schematically derive a set of transition rules.

Definition 16. *A* reduction context $r[\cdot]$ *is an* ERLANG *expression with a "hole" in it, generated by the grammar in Table 3.6.*

The result of placing e in (the hole of) a context $r[\cdot]$ is denoted $r[e]$. Intuitively a reduction context identifies a subexpression position in an expression such that if a subexpression is put in the position, and it can perform an action α, then the whole expression can perform the same action leaving the expression structure intact. Note that the `try` statement is absent from the definition of a reduction context since the expression structure can change due to the actions of a subexpression (rule *try*$_2$).

Definition 17 (Context Rules)**.** *The set of context operational rules is the smallest set of transition rules generated from the reduction context grammar and the meta-rule*

$$\textit{context}\ \frac{e \xrightarrow{\alpha} e'}{r[e] \xrightarrow{\alpha} r[e']}\ r[\cdot] \neq \cdot$$

For instance, in addition to the basic rules *send*$_0$ found in Table 3.7 and *send*$_1$ in Table 3.9 two additional rules can be derived from the above meta-rule:

$$\textit{send}_2\ \frac{e_1 \xrightarrow{\alpha} e_1{}'}{e_1!e_2 \xrightarrow{\alpha} e_1{}'!e_2} \qquad\qquad \textit{send}_3\ \frac{e \xrightarrow{\alpha} e'}{v!e \xrightarrow{\alpha} v!e'}$$

Note the implicit condition in the rule *send*$_3$ which requires the first argument in the send expression to have become a value v before the second argument can be evaluated.

Definition 18 (Immediate Subexpressions)**.** *Let the set of immediate subexpressions of an expression be defined in the obvious way over the structure of the* ERLANG *constructs in Table 3.1.*

For instance a send expression $e_1 \,!\, e_2$ has the immediate subexpressions e_1 and e_2.

Definition 19 (Expression Transition Relation)**.** *The* expression transition relation*,* $\rightarrow$: erlangExpression $\times$ erlangExprAction $\times$ erlangExpression*, written* $e_1 \xrightarrow{\alpha} e_2$ *when* $\langle e_1, \alpha, e_2 \rangle \in \rightarrow$*, is the least relation satisfying the transition rules in Tables 3.7,3.8,3.9,3.10, 3.11 and Definition 17.*

To express that a transition labelled by an action α exists from a term t the notation $t \xrightarrow{\alpha}$ will be used, and to express that a term t has any transition regardless of its action and target state the notation $t \rightarrow$ is used.

Table 3.7 defines the semantics of most expression constructs when subexpressions are fully evaluated and evaluation proceeds normally, Table 3.8 defines the rules for exception processing, Table 3.9 present rules for error cases for normal expression constructs, and Tables 3.10 and 3.11 give rules for normal, and exceptional evaluation of built-in functions. Finally Definition 17 regulates the order in which subexpressions are evaluated.

Semantics of Matching

In the `case` and `receive` expressions values are matched against patterns possibly containing variables. In the formal treatment of these expressions a number of auxiliary functions and predicates are defined, and a transition relation is defined for the evaluation of guards, built on top of the expression transition relation.

Definition 20. *The* guard transition relation*,* $\rightarrow_g$: erlangGuard $\times$ erlangGuard*, written* $g_1 \rightarrow_g g_n$*, is the least relation satisfying the rules in Table 3.12.*

Note that guard computations are given a natural semantics, i.e., any number of internal steps are combined into a single guard transition step. This presents no difficulties since a guard is a sequence of expressions, all of which are guaranteed to eventually evaluate to booleans, or raise an exception, since they are built from applications of well-behaved built-in functions.

The predicate *matches* v (p `when` g `->` e) determines when an ERLANG value v matches a case alternative:

Definition 21.

$$\mathit{matches}\ v\ (p\ \texttt{when}\ g\ \texttt{->}\ e) \triangleq \exists \widetilde{V}.\ \left(v = p\{\widetilde{V}/\widetilde{fv(p)}\} \wedge g\{\widetilde{V}/\widetilde{fv(p)}\} \rightarrow_g \texttt{true}\right)$$

In other words, a case alternative matches a value if we can find a substitution of values for the variables in the pattern such that the pattern and value becomes identical, and the guard evaluates to true. The notation in the definition is not precise; we let $\widetilde{fv(p)}$ represent a vector of variables in ascending order, based on their names, consisting exactly of the variables that are in the set $fv(p)$.

The *matches* predicate is extended to queues (mailboxes) in the obvious way:

$$\textit{case}_0\ \frac{\exists I.((\textit{result}\ v\ m_I\ e) \wedge \forall J.J < I \Rightarrow \neg(\textit{matches}\ v\ m_J))}{\texttt{case}\ v\ \texttt{of}\ m\ \texttt{end} \xrightarrow{\tau} e}$$

$$\textit{fun}_0\ \frac{f \triangleq m \qquad \texttt{case}\ \{v_1,\ldots,v_n\}\ \texttt{of}\ m\ \texttt{end} \xrightarrow{\alpha} e'}{f(v_1,\ldots,v_n) \xrightarrow{\alpha} e'}$$

$$\textit{fun}_1\ \frac{\textit{isProcFun}\ f}{f(v_1,\ldots,v_n) \xrightarrow{f(v_1,\ldots,v_n)\rightsquigarrow\{\texttt{result},v\}} v}$$

$$\textit{fun}_2\ \frac{\textit{isProcFun}\ f}{f(v_1,\ldots,v_n) \xrightarrow{f(v_1,\ldots,v_n)\rightsquigarrow\{\texttt{error},v\}} \texttt{exiting}\ \{\texttt{'EXIT'},v\}}$$

$$\textit{receive}\ \frac{\begin{array}{c}\forall I.\neg(\textit{qmatches}\ q\ m_I) \\ \exists I.\ ((\textit{result}\ v\ m_I\ e') \wedge \forall J.J < I \Rightarrow \neg(\textit{matches}\ v\ m_J))\end{array}}{\texttt{receive}\ m\ [\texttt{after}\ i\texttt{->}e]\ \texttt{end} \xrightarrow{\textit{read}(q,v)} e'}$$

$$\textit{timeout}\ \frac{\forall I.\neg(\textit{qmatches}\ q\ m_I)}{\texttt{receive}\ m\ \texttt{after}\ i\texttt{->}e\ \texttt{end} \xrightarrow{\textit{test}(q)} e}$$

$$\textit{send}_0\ \frac{}{\textit{pid}!v \xrightarrow{\textit{pid}!v} v}$$

Table 3.7: Normal expression evaluation

$$\textit{exiting}_0\ \frac{}{\texttt{exiting}\ v \xrightarrow{\textit{exiting}(v)} v}$$

$$\textit{try}_0\ \frac{}{\texttt{try}\ v\ \texttt{catch}\ m\ \texttt{end} \xrightarrow{\tau} v}$$

$$\textit{try}_2\ \frac{e \xrightarrow{\textit{exiting}(v)} e' \qquad \texttt{case}\ v\ \texttt{of}\ m\ \texttt{end} \xrightarrow{\alpha} e''}{\texttt{try}\ e\ \texttt{catch}\ m \xrightarrow{\alpha} e''}$$

$$\textit{try}_3\ \frac{e \xrightarrow{\alpha} e' \quad \neg\exists V : \mathsf{erlangValue}.\alpha = \textit{exiting}(V)}{\texttt{try}\ e\ \texttt{catch}\ m\ \texttt{end} \xrightarrow{\alpha} \texttt{try}\ e'\ \texttt{catch}\ m\ \texttt{end}}$$

Table 3.8: Exception handling

$$\textit{case}_1\ \frac{\forall I.\neg(\textit{matches}\ v\ m_I)}{\texttt{case}\ v\ \texttt{of}\ m\ \texttt{end} \xrightarrow{\textit{exiting}(\texttt{case\_clause})} \texttt{bottom}}$$

$$\textit{fun}_3\ \frac{\neg(\textit{isProcFun}\ v) \wedge \neg(\textit{isDefined}\ v) \wedge \neg(\textit{isBIF}\ v)}{v(v_1,\ldots,v_n) \xrightarrow{\textit{exiting}(\texttt{badfun})} \texttt{bottom}}$$

$$\textit{send}_1\ \frac{\neg(\textit{isErlangPid}\ v)}{v!v' \xrightarrow{\textit{exiting}(\texttt{badarg})} \texttt{bottom}}$$

$$\textit{receive}_1\ \frac{\neg(\textit{isErlangInt}\ v)}{\texttt{receive}\ m\ \texttt{after}\ v\ \texttt{->}\ e\ \texttt{end} \xrightarrow{\textit{exiting}(\texttt{badarg})} \texttt{bottom}}$$

Table 3.9: Exceptional expression evaluation

$$\textit{tl}_0\ \frac{\textit{tl}_{\text{ERLANG}}\ v_1\ v_2}{\texttt{tl}(v_1) \xrightarrow{\tau} v_2}$$

Table 3.10: Evaluation of built-in functions (example)

$$\textit{tl}_1\ \frac{n \neq 1}{\texttt{tl}(v_1,\ldots,v_n) \xrightarrow{\textit{exiting}(\texttt{cond\_clause})} \texttt{bottom}}$$

$$\textit{link}_0\ \frac{n \neq 1}{\texttt{link}(v_1,\ldots,v_n) \xrightarrow{\textit{exiting}(\texttt{cond\_clause})} \texttt{bottom}}$$

$$\textit{tl}_2\ \frac{\neg\exists H,T : \mathsf{erlangValue}.v = [H|T]}{\texttt{tl}(v) \xrightarrow{\textit{exiting}(\texttt{badarg})} \texttt{bottom}}$$

$$\textit{link}_1\ \frac{\neg(\textit{isErlangPid}\ v)}{\texttt{link}(v) \xrightarrow{\textit{exiting}(\texttt{badarg})} \texttt{bottom}}$$

Table 3.11: Exceptional evaluation of built-in functions (examples)

$$g_0\ \frac{ge \xrightarrow{\tau} v}{ge \rightarrow_g v} \qquad g_1\ \frac{ge \xrightarrow{exiting(v)} ge'}{ge \rightarrow_g \texttt{false}} \qquad g_2\ \frac{ge \xrightarrow{\tau} ge' \quad ge' \rightarrow_g v}{ge \rightarrow_g v}$$

$$g_3\ \frac{ge \rightarrow_g \texttt{false}}{ge, ge_1, \ldots, ge_n \rightarrow_g \texttt{false}} \qquad g_4\ \frac{ge \rightarrow_g \texttt{true} \quad ge_1, \ldots, ge_n \rightarrow_g v}{ge, ge_1, \ldots, ge_n \rightarrow_g v}$$

Table 3.12: Computation of Guard Expressions

Definition 22.

$$\begin{aligned} &\textit{qmatches } q\ (p\ \ \texttt{when}\ \ g\ \texttt{->}\ e) \triangleq \\ &\quad \exists V, Q_1, Q_2 . q = Q_1 \cdot V \cdot Q_2 \wedge \textit{matches } V\ (p\ \ \texttt{when}\ \ g\ \texttt{->}\ e) \end{aligned}$$

The *result* predicate includes a test of equality against the resulting expression:

Definition 23.

$$\begin{aligned} &\textit{result } v\ (p\ \ \texttt{when}\ \ g \texttt{->} e)\ e' \triangleq \\ &\quad \exists \widetilde{V} . \left(v = p\{\widetilde{V}/\widetilde{fv(p)}\} \wedge g\{\widetilde{V}/\widetilde{fv(p)}\} \rightarrow_g \texttt{true} \wedge e' = e\{\widetilde{V}/\widetilde{fv(p)}\} \right) \end{aligned}$$

Some results about these definitions are needed before proceeding:

Lemma 2. *If result v m e then matches v m.*

Proof. Immediate from the definitions of *result* and *matches*. □

Lemma 3. *If result v m e' and result v m e'' then $e' = e''$.*

Proof. Since the arguments e' and e'' are uniquely determined by the substitution function and the expression (possibly containing variables) e in m, this reduces to the question whether the ERLANG value vector $\widetilde{V}$ is uniquely determined, i.e.,

$$\begin{aligned} &\exists \widetilde{V} . \exists \widetilde{V'} . \widetilde{V} \neq \widetilde{V'} \wedge p[\widetilde{V}/\widetilde{fv(p)}] = p[\widetilde{V'}/\widetilde{fv(p)}] \wedge \\ &\qquad g\{\widetilde{V}/\widetilde{fv(p)}\} \rightarrow_g \texttt{true} \wedge g\{\widetilde{V'}/\widetilde{fv(p)}\} \rightarrow_g \texttt{true} \end{aligned}$$

The notation $\widetilde{V} \neq \widetilde{V'}$ expresses inequality, for some index i, of values V_i and V'_i.

First note that all values in $\widetilde{V}$ (and $\widetilde{V'}$) are relevant, since the arities of $\widetilde{V}$ and $\widetilde{fv(p)}$ coincide. Thus, if $\widetilde{V'} \neq \widetilde{V''}$ then there must trivially be two value subexpressions v_1 and v_2 in $p[\widetilde{V}/\widetilde{fv(p)}]$ and $p[\widetilde{V'}/\widetilde{fv(p)}]$ such that $v_1 \neq v_2$. Now all the ERLANG compound pattern constructors considered in this thesis, i.e., the conses and tuples, are freely generated, and thus also $p[\widetilde{V}/\widetilde{fv(p)}] \neq p[\widetilde{V'}/\widetilde{fv(p)}]$. We have derived a contradiction. □

Definition 24 (Closed Expression)**.** *A* closed expression e *is an expression* e *such that* $fv(e) = \emptyset$.

Let closedErlangExpression range over the subtype of expressions that contains all closed expressions.

Definition 25 (Labelled Transition System of Closed Expressions)**.** *Define the labelled transition system of the closed* ERLANG *expressions as the tuple*

$$\langle \mathsf{closedErlangExpression}, \mathsf{erlangExprAction}, \rightarrow \rangle$$

where $\rightarrow$ *is the transition relation of Definition 19.*

Discussion of Semantic Rules

To illustrate the expression semantics rules we will explain a few rules in more detail. First, the *receive* rule on Page 50 is considered. A transition from `receive` m `end` labelled by the action *read*(q, v) to the expression e' is enabled whenever we can find an index I in m (recall that m is a sequence of clauses $p_1 \rightarrow$ `when` g_1 e_1; $\cdots$; $p_n \rightarrow$ `when` g_n e_n) such that pattern I matches v, the guard expression g_I evaluates to true, and e_I is equal (under substitution of variables) to e'. Moreover, there is neither an earlier pattern in m that satisfies this condition nor is there any value in q which matches any pattern in m in this manner.

Note that in general there will be an infinite number of queues q and values v that satisfy this condition. Thus the transition semantics of an expression containing a `receive` statement is infinitely branching. Any difficulties caused by this are addressed in the proof system in a completely standard manner through use of quantifiers.

Similarly the fun_1 rule expresses a function call to a function having side effects, or depending upon the process state. Two examples of such functions are `spawn` and `self`. Since the return value v of such a function is not defined by the expression level, the rule holds for any choice of return value. In the process level semantics the choice of a return value will be constrained, for each function with side effects.

The rules in Table 3.9 regulate which exceptions are raised for standard expression constructs. The choice of a target expression in these rules is arbitrary since it will never be accessed; for clarity the atom `bottom` is chosen.

The single rule tl_0 in Table 3.10, which returns the tail part of a list, is meant to illustrate a whole class of rules that provide the linkage between the ERLANG built-in functions that do not depend upon side-effects and their logical characterisation. It is assumed that such functions are reducible, via one internal step, to a ground value, or that they raise an exception because of calling convention errors. The error case is represented by the rules tl_1 and tl_2 in Table 3.11. The definition of the tl_{ERLANG} predicate is found on Page 108. Note that calling convention errors are handled in Table 3.11 also for so called process state dependent functions like `link`.

It is required that each predicate which characterise a built-in function in the logic, e.g. $tl_{\mathrm{ERLANG}}\ v_1\ v_2$ in the rule tl_0, defines a function on is last argument (v_2).

Next some results about the ERLANG expression level transition relations will be established.

Lemma 4. *For no* ERLANG *value* v *does there exist an action* α *and expression* e *such that* $v \xrightarrow{\alpha} e$.

Proof. Via induction over the structure of values, all cases are obvious. The inductive argument is required to treat the tuple and the cons operators. □

Proposition 2 (Expression Derivations are Finite)**.** *For each expression* e *there is a finite bound on the depth of inferences from* e.

Proof. The only real complication here is the interplay between the guard transition relation and the expression transition relation.

First consider the (expression) transitions of a guard expression ge. Clearly, due to syntactic restrictions, a guard expression does not contain a guard. We proceed by structural induction on ge. As base cases we have the values, from which no transitions are derivable, and the application of a built-in function such that all its arguments have been evaluated. Then, either the transition can be derived from Table 3.10 or Table 3.11, and the result is a value or an exception, and no longer derivations can be derived. Consider the induction cases, either there is a data constructor (tuples or lists), or there is a function application, and the reduction context meta-rule is applied, and the proper subexpressions have a bound on inferences depth. But then clearly the meta-rule has a similar bound.

Consider then the case of a guard, and the guard transition relation. The problematic case is g_2. We have to establish that the "complexity" of e' is less than e. We prove, again by induction over the structure of a guard expression ge, that there is a bound on the depth of guard derivations. Again the base cases are the values and application of a built-in function, which are handled by rules g_0 and g_1 and are trivial. Consider a reduction context, where by the induction hypothesis all proper subexpression have a bounded guard derivation. If the guard derivation of any such subexpression ends in an exception the proof is done. If all derivations yield a value, then either the result is a function application of a built-in function or a value, and both cases are trivial.

Finally consider the case of an arbitrary expression e, and prove that there is a finite bound by induction over the structure of expressions. So assume all proper subexpressions have finitely bounded derivations. The only non-trivial cases (the context rules again are easy to discharge due to the induction hypothesis) are the rules fun_0 and try_2. For an expression `case` v of m `end` the only possibly applicable rules are case_0 or case_1 which have expression transition premises. They do have guard transition premises, but since we showed above that these have bounded derivations the result follows. □

Next it is established that if an instance of the schematic context reduction rule is applicable (Definition 17) then no other semantics rule is applicable, and vice versa.

Lemma 5. *If there exists a proper subexpression* e' *of an expression* e *such that* $e = r[e']$ *and* $e' \rightarrow$ *then there is no transition rule in Tables 3.7,3.8,3.9,3.10,3.11 such that* $e \rightarrow$ *is derivable. Vice versa, if any rule in the above tables can derive* $e \rightarrow$ *then there exists no proper subexpression* e' *of* e *such that* $e = r[e']$ *and* $e' \rightarrow$.

Proof. Recall that Lemma 4 proves that a value has no transitions. We prove the lemma by considering the reduction contexts of an arbitrary expression. Suppose e is a non-empty list or tuple. Clearly no rule in any of the tables is applicable. Suppose e is a function application. The only rules applicable in the tables trigger when all subexpression are values. However, none of these subexpression can then have a transition. The proofs for the case, receive and send constructs are identical. □

Next it is established that transitions under reduction contexts, at the same expression level, are deterministic.

Lemma 6. *Suppose an expression e has at least two distinct immediate subexpressions e_1 and e_2, and that there exists two reduction contexts $r[\cdot]$ and $r'[\cdot]$ which replace the subexpressions with holes. Then either e_1 or e_2 must be a value, and as a result at most one of the statements $e_1 \rightarrow$ or $e_2 \rightarrow$ are derivable.*

Proof. That at least one of the two immediate subexpressions is a value follows directly from the definition of the set of reduction contexts. That a value has no transitions follows from Lemma 4. □

Proposition 3 (Determinacy of Expressions)**.** *If $e \xrightarrow{\alpha} e'$ and $e \xrightarrow{\alpha} e''$ then $e' = e''$.*

Proof. By induction on the length of the derivation of a transition. We will assume that computations of guards are deterministic. So consider an arbitrary expression e and its derivations. From Lemma 5 it follows that if the schematic context rule is applicable any other rule is not, and vice versa. So consider first the schematic context rule. Since there is only one active subexpression, due to Lemma 6, the result follows after one step due to the induction hypothesis.

So consider any expression without an "active" reduction context. Here we have to show that the rules in Tables 3.7,3.8,3.9,3.10 and 3.11 are mutually exclusive. A case analysis follows, we discuss a few noteworthy examples below.

We consider first the case of transition rules for send, i.e., send_0 and send_1 below. Clearly if $e = \mathit{pid}!v$ for some process identifier *pid* then the premise $\neg(\mathit{isErlangPid}\ \mathit{pid})$ is not provable, and vice versa.

For `case` expressions there are again two applicable rules, case_0 and case_1. However, here it is less clear that even the rule case_0 is deterministic. There are two issues. First, is the least index I, corresponding to a particular match clause uniquely determined, and second, does the predicate *result* define a function in its last argument e. To show that I is uniquely determined suppose there is an index $J < I$ which also satisfies the conditions. Then $\mathit{result}\ v\ m_I\ e$, but from Lemma 2 it follows that $\mathit{matches}\ v\ m_I$, so the premises for index I are not satisfied, a contradiction. The same argument applies to simultaneous derivability of both case_0 and case_1. For the second issue, Lemma 3 proves that the expression results of matching are uniquely determined.

Next is the case of the `receive` expression. Suppose that

$$\texttt{receive ...end} \xrightarrow{\mathit{read}(q,v)} e'$$

and

$$\texttt{receive ...end} \xrightarrow{\mathit{read}(q,v)} e''$$

and $e' \neq e''$. Since, as commented on above, *result* defines a function in its last argument (Lemma 3) there must be two clauses m_k and m_l, $k < l$ such that (i) *result* v m_k e' (for m_k) and (ii) *result* v m_k e'' (for m_l). But then the premise for the clause m_l includes the premise $\neg$*matches* v m_k. So appeal again to Lemma 2, which establishes *matches* v m_k.

Further, predicates describing the effect of built-in functions, are required to be of function shape: the predicate arguments but the last one determine a unique last argument.

For the treatment of built-in functions with side effects, causing actions $\xrightarrow{f(v_1,\ldots,v_n)\rightsquigarrow v}$, note that the presence of the return value v in the action is crucial to achieve determinacy.

□

A stronger result than Proposition 3 is also provable.

Proposition 4. *If $e \xrightarrow{\alpha} e'$ and $e \xrightarrow{\alpha'} e''$ and $e' \neq e''$ then $\alpha \neq \alpha'$ and either the actions α and α' are both queue reads (read) or queue tests (test), or both are calls of built-in process state functions ($\rightsquigarrow$).*

Proof. An easy induction over the length of a derivation. □

3.2.4 Bisimilarity for Expressions

We develop a theory of substitutability for ERLANG, in this section focusing on the expressions. That is we answer the question when an expression e can replace an expression e' without any noticeable differences. Considering the answer in terms of the program logic developed in the thesis, it is clear that the logic is very sensitive, permitting syntactical analysis of terms. Thus one answer is that there are always formulas ϕ that can tell two syntactically different expressions e_1 and e_2 apart, i.e., $e \in \|\phi\|$ while $e' \notin \|\phi\|$. However, if attention is restricted to, say, the logic fragment of modalities and operators on modalities then a much richer notion of substitutability results.

In the following a natural notion of bisimilarity is presented, based on the actions of ERLANG expressions. Since transitions are deterministic with respect to actions, the notions of bisimulation equivalence and trace equivalence coincide.

Definition 26. *A binary relation S is an expression bisimulation if $(e_1, e_2) \in S$ implies, for all expression actions α,*

- *Whenever $e_1 \xrightarrow{\alpha} e_1'$ then, for some e_2', $e_2 \xrightarrow{\alpha} e_2'$, and $(e_1', e_2') \in S$, and vice versa.*
- *Whenever e_1 is an* ERLANG *value then $e_1 = e_2$, and vice versa.*

Definition 27. *The expressions e_1 and e_2 are expression bisimilar, written $e_1 \dot{\sim}_e e_2$, if $(e_1, e_2) \in \mathcal{S}$ for some expression bisimulation $\mathcal{S}$.*

Next a congruence relation is constructed from the bisimulation relation.

Definition 28 (Expression Contexts)**.** *An expression context $C[\cdot]$ is an expression with a hole indicated by $\cdot$ replacing an arbitrary proper subexpression.*

Note that $\dot{\sim}$, from Definition 27, is not a congruence, for the trivial reason that variables are not properly treated. Consider for instance the expressions `X` and `Y`. Clearly if these expressions are put in the context `case 1,2 of X,Y -> · end` (where $\cdot$ denotes the hole) the results are not expression bisimilar. Such effects are well-known from semantical accounts of other languages such as the π-calculus.

There are also some non-standard complications, due to the peculiar binding conventions of ERLANG. Consider for instance the expressions `case 1 of X -> X end` and `case 1 of Y -> Y end` which are trivially expression bisimilar. However, when put into the context `case 1,2 of X,Y -> · end` the resulting expressions are not bisimilar, since the surrounding case expression will bind also the inner variables.

First we define what is meant by a congruence relation, based at first on the very fine notion of variable observability.

Definition 29. *A relation $\overset{r}{\sim}$ (over pairs of* ERLANG *expressions) is considered a congruence if for any two terms e, e' such that $e \overset{r}{\sim} e'$, and for any expression context $C[\cdot]$, either one of the expressions $C[e]$ or $C[e']$ are not syntactically valid, or $C[e] \overset{r}{\sim} C[e']$.*

The conditions in the above definition are in place to prevent, for example, placing arbitrary expressions into guard positions.

We have a choice here, to proceed with this notion, and define a very fine congruence that will be sensitive to variable names due to the lack of a proper binding operator as shown above. An alternative is to relax the notion of a congruence relation to consider only substitutability into contexts that do not bind variables that are bound also in e and e'. Here the second option is chosen.

First a variant of the congruence notion is defined.

Definition 30. *A relation $\overset{r}{\sim}$ (over pairs of* ERLANG *expressions) is considered an expression congruence if for any two terms e, e' such that $e \overset{r}{\sim} e'$, and for any expression context $C[\cdot]$ for which the three conditions (i) the expressions $C[e]$ and $C[e']$ are syntactically valid, (ii) $(pv(e) \cup pv(e')) \cap pv(C[\cdot]) = \emptyset$, and (iii) $(fv(e) \cup fv(e')) \cap (fv(C[e]) \cup fv(C[e'])) = \emptyset$ hold, then $C[e] \overset{r}{\sim} C[e']$.*

Recall that the set $pv(e)$ consists of the variables that occur in a pattern in e, and was defined in Table 3.4 on Page 3.4. The extra conditions guarantee that (ii) no variables that occur in patterns in e or e' are bound by the context in which they are substituted, and (iii) that the context provides an interpretation for all free variables of e and e'. Now a suitable congruence instance can be defined.

Definition 31. *The expressions e_1 and e_2, are expression congruent, written $e \sim_c e'$, if for all substitutions $[\tilde{v}/(fv(e) \cup fv(e'))]$ assigning* ERLANG *values to the free variables of the expressions, $e[\tilde{v}/(fv(e) \cup fv(e'))] \dot{\sim}_e e'[\tilde{v}/(fv(e) \cup fv(e'))]$.*

Proposition 5. *$\sim_c$ is an expression congruence.*

Proof. Assume an arbitrary expression context $C[\cdot]$, and two expressions e,e', according to the characterisation of expression congruence. We will exhibit an expression bisimulation S which contains $(C[e_1], C[e_2])$. The proof idea is to trace the evolution of e_1 and e_2 through a bisimulation up to the point when the expressions e_1 and e_2 (under some substitution) must be reduced.

Assume that the free variables of e_1 and e_2 are named $X_1, \ldots, X_n$. Consider any value assigning substitution ρ defined over the free variables of e and e'. From the definition of expression congruence it follows that there exists a bisimulation S_ρ such that $(e, e') \in S_\rho$. Denote with $S_{e,e'}$ the bisimulation that is the union of all such bisimulations (over ρ), and add it to S.

Consider next any set S' such that

$$(C[\{Z, X_1, \ldots, X_n\}], C[\{Z, X_1, \ldots, X_n\}]) \in S'$$

where Z is a variable that does not occur in $C[\cdot]$, e or e', and where $(e'', e'') \in S'$ if and only if $C[\{Z, X_1, \ldots, X_n\}] \rightarrow^* e''$, where $e_1 \rightarrow^* e_2$ if $e_1 = e_2$ or there exists an expression e_3 and an action α_3 such that $e_1 \xrightarrow{\alpha_3} e_3$ and $e_3 \rightarrow^* e_2$. Clearly S' is an expression bisimulation.

Now construct the set S from S' by replacing a pair $(e'', e'') \in S'$, where a sub term $\{Z, e_1, \ldots, e_n\}$ occurs in e'' in a non-reduction context position with the pair (e_l, e_r) in S such that all occurrences of the subterm above is replaced with $e[e_1/X_1, \ldots, e_n/X_n] = e_a$ in e_l and with $e'[e_1/X_1, \ldots, e_n/X_n] = e_b$ in e_r.

We the following lemmata about about expression evaluation and expression contexts are easily established: if an expression $C[e_1]$ has a transition labelled by α to an expression e'', and e_1 is a tuple containing an ERLANG variable Z in the first tuple position and which does not occur in the expression context $C[\cdot]$, then

- There is at most one expression context $r'[[\cdot]]$ and expression ${e_1}'$ which contains the variable Z such that $e'' = r'[[{e_1}']]$. This follows easily by induction over the transition relation since a non-value subexpression is never duplicated, and the expression transition relation is deterministic.
- The variable Z is not found in $r'[[\cdot]]$. Again this is established by a trivial induction showing that a non-value tuple is never decomposed by any transition rule.
- For any expression e, the transition $C[e] \xrightarrow{\alpha} r'[[e']]$ is derivable, where e' is e except some free variables in e may have become bound.

This observation means that we can uniquely trace the original tuple with the variable Z, and that any other expression e can replace the tuple but have no effect (this

follows from determinacy of the transition relation). So clearly S is also a bisimulation up to the point where the tuple occurs in a redex position (if it does not we are done!).

So consider the pair with the tuple in the redex position and perform the same substitution as above, resulting two terms e_a and e_b occurring once as subterms in e_l and e_r. If either e_a or e_b are values then we are done, since they must be equal. This follows because there is a substitution ρ (defined by the tuple above) such that $e_a = e\rho$ and $e_b = e'\rho$ but then $(e_a, e_b) \in S_{e,e'} \subseteq S$. Now if e_a is a value then e_b must be a value too, and vice versa.

If e_a and e_b are equivalent to e_l and e_r, i.e., they occur at the top of the expressions, then there can be no variables free in either term. This fact follows from the original condition that no variables in e_1 or e_2 are free in the original expression context. But then again $(e_a, e_b) \in S_{e,e'} \subseteq S$. So we are done.

The alternative is that e_a and e_b occur in a position such that the meta-rule context is applicable, or else under a context `try` $\cdot$ `catch` ... `end`, such that the rules try_2 or try_3 is the only applicable ones.

We consider the context case, the case for `try` is analogous. There must exist a reduction context $r[e_a] = e_l$ and $r[e_b] = e_l$ (since e_a and e_b are not values). Here simply generate the set S''' containing all pairs $(r[e_3], r[e_4])$ such that $(e_3, e_4) \in S_{e_l,e_r}$. Due to the definition of the context rule the same transitions will be derivable. Finally if there is a value e'_3 such that $e_3 \rightarrow^* e'_3$ then (due to the bisimulation construction) also $e_4 \rightarrow^* e'_3$. So finally add the pair $(r[e'_3], r[e'_3])$ to S and all its derivatives; we have found the required bisimulation.

To conclude, we have exhibited a bisimulation relation S relating the initial expressions $C[e_1]$ and $C[e_2]$, and have thus proved that $\sim_c$ is an expression congruence. □

Example 5. Consider the ERLANG function `loop() -> loop()`, and the expression `{3, loop(), 1}`. Clearly, $(\{3, \texttt{loop()}, 1\}, \texttt{loop()})$ is an expression bisimulation. Since no variables occur in `loop()` then, for any expression context $C[\cdot]$, it holds that $C[\{3, \texttt{loop()}, 1\}] \sim_c C[\texttt{loop()}]$.

Next we would like to characterise a subset of the formulas of the logic such that two expressions satisfy exactly the same formulas if and only if they are expression congruent. For a value-free language such a restriction to a fragment of the logic poses few problems; it is the sublogic with modalities and without equality. In our case, however, in defining the formula fragment it is necessary to carefully trace the program terms to ensure that no tests for equality on them takes place. Of course a formula must nevertheless be allowed to test whether a program term has evaluated to a value. It should also be realised that expression congruence is too strong to motivate many useful equational laws, in particular since computation is deterministic. A more useful congruence would result from considering the weak actions, i.e., such that multiple internal computation steps are collapsed. The exploration of these issues is left for future work.

3.2.5 Dynamic Semantics of Systems

Here the notion of processes that encapsulate the ERLANG expressions, and the systems that are collections of processes, are formalised.

The ERLANG processes, ranged over by $p \in \mathsf{erlangProcess}$, are either live or dead. Process that are not live are not wholly inert; they will eventually inform linked processes about their termination and in addition they will respond to received $link$ signals. The reason to introduce dead processes in the semantics is to be able to reason about the semantics of process linking.

Definition 32. *A* live ERLANG process *(*$\mathsf{erlangLiveProcess} \subset \mathsf{erlangProcess}$*) is a quintuple* : $\mathsf{erlangExpression} \times \mathsf{erlangPid} \times \mathsf{erlangQueue} \times \mathcal{P}(\mathsf{erlangPid}) \times \mathsf{erlangBool}$*, written* $\langle e, pid, q, pl, b\rangle$*, such that*

- e *is an* ERLANG *expression,*
- *pid is the process identifier of the process,*
- q *is a message queue,*
- pl *is a set of process identifiers (a set of links with other processes),*
- b *is an* ERLANG *boolean determining how process exit notifications are handled.*

Definition 33. *A* terminated ERLANG process *(*$\mathsf{erlangDeadProcess} \subset \mathsf{erlangProcess}$*) is a tuple* : $\mathsf{erlangPid} \times \mathcal{P}(\mathsf{erlangPid} \times \mathsf{erlangValue})$*, written* $\langle pid, plm\rangle$*, where*

- *pid is the process identifier of the process,*
- plm *is a set of tuples combining process identifiers with a notification value that should be sent to the corresponding process.*

Definition 34. *An* ERLANG system *is either a singleton process or a composition of systems* s_1 *and* s_2 *written as* $s_1 \parallel s_2$.

Informally, a system is a multiset of ERLANG processes. The semantics of the "$\parallel$" construct guarantees that viewed as an operator it is commutative and associative (see Proposition 15).

For clarity, and when there is no risk for confusion, the linked processes parameter and the boolean flag parameter will sometimes be omitted from the state of live ERLANG processes, e.g., they are written on the format $\langle e, pid, q\rangle$.

Definition 35. *Let the function* $pids(s)$ *return the set of process identifiers belonging to processes in the system* s*. Further let a system be* well-formed *if its process identifiers are unique, i.e., a process identifier belongs to at most one process.*

In the following we assume the well-formedness predicate which is represented by *wf* in the logic.

Definition 36. *Let the proper* ERLANG *systems, of type* $\mathsf{erlangSystem}$ *, be the well-formed* ERLANG *systems:* $\{s \in \mathsf{erlangSystem}' \mid wf\ s\}$.

Henceforth let s range over only the well-formed ERLANG systems, and from know on the notion of a system will refer to a well-formed one only. Hence a system is from now intuitively a set rather than a multiset.

A *signal* is an item of information transmitted between a sending ERLANG process and a receiving ERLANG process.

Definition 37. *The* signals*, ranged over by* $sig \in \mathsf{erlangSignal}$*, are:*

sig	$::=$	$message(v)$	message
	$\mid$	$link(pid)$	linking with process
	$\mid$	$unlink(pid)$	unlinking process
	$\mid$	$exited(pid, v)$	passive termination signal
	$\mid$	$exit(pid, v)$	active termination signal

An ordinary message v transmitted to another process corresponds to the signal $message(v)$. As a convention this signal will normally be written simply v. For linking to and unlinking another process the signals $link(pid)$ and $unlink(pid)$ are used. A signal $exited(pid, v)$ indicates the termination of *pid* for reason v. Finally the signal $exit(pid, v)$ indicates that the process *pid* requests the termination of the receiving process for reason v. If v is `kill` then termination of the receiving process cannot be prevented.

A *system action*, committed by a system, is either a silent action, an *output action* or an *input action*.

Definition 38. *The* system actions*, ranged over by* $\alpha \in \mathsf{erlangSysAction}$*, are:*

α	$::=$	τ	silent action
	$\mid$	$pid!sig$	output action
	$\mid$	$pid?sig$	input action

Definition 39. *The* system transition relation*,* $\rightarrow$: $\mathsf{erlangSystem} \times \mathsf{erlangSysAction} \times \mathsf{erlangSystem}$*, written* $s_1 \xrightarrow{\alpha} s_2$*, is the least relation satisfying the rules in Tables 3.13, 3.14, 3.15, 3.16, 3.17 and 3.18.*

In Tables 3.13, 3.14, 3.15 and 3.16 the role of the process operator *proc* as an abstraction mechanism for expressions is explored. Table 3.17 presents rules for terminated processes whereas Table 3.18 describes the semantics of the parallel composition operator $\|$.

Discussion of Semantic Rules

To illustrate the system semantics rules we will explain the first three rules in Table 3.13. Consider first the rule *silent*. If the expression e has a transition $e \xrightarrow{\tau} e'$ (a computation step) then the process $\langle e, pid, q, pl, b\rangle$ has a transition labelled by the silent action τ to the process $\langle e', pid, q, pl, b\rangle$. Next let us consider the rules $output_0$

and *output*$_1$. If the expression e can perform a send action *pid*$'$!v then either the value is to be sent to a remote process (if *pid*$' \neq$ *pid*) and so an action *pid*$'$!*message*(v) with a signal *message*(v) parameter occurs. Alternatively, if *pid*$' =$ *pid* the value v is simply appended to the queue of the process $q \cdot v$.

Next consider the *read* rule on Page 63. A computation step transition from $\langle e, pid, q_1 \cdot v \cdot q_2, pl, b\rangle$ is enabled to the target process $\langle e', pid, q_1 \cdot q_2, pl, b\rangle$ whenever the process mailbox (queue) can be split into three parts $q_1 \cdot v \cdot q_2$, and the expression transition *receive* $e \xrightarrow{read(q_1,v)} e'$ is derivable. Thus the rules *read* and *receive* together ensure the intuitive semantics of the `receive` construct.

One of several rules handling termination of a process is *termination*, which triggers when an expression has been evaluated to a ground value v. Then the process changes state, and prepares to send termination messages (informing that the reason for process termination was `normal`) to linked processes.

Next the handling of functions that depend upon the process state is considered in Table 3.14 (see Page 35 for intuition). Considering rule *self*, for instance, it is enabled for an action `self()` $\rightsquigarrow$ *pid* in the context of a process with process identifier *pid*. The handling of the built-in function `trap_exit` addresses only a small part of its normal functionality, namely the modification of process termination reception semantics.

The handling of process spawning on the process level is specified by the rules *spawn*$_0$ and *spawn*$_1$ in Table 3.14. The rules essentially states that any choice of a new process identifier is acceptable as long as it does not coincide with the process identifier of the spawning process, to preserve uniqueness of process identifiers. The rule *interleave*$_0$ will further constrain the choice of a new process identifier, in the context of additional processes. Thus, in a hierarchical and context-free way, we ensure that the uniqueness of process identifiers are preserved in contrast with the global condition "*pid* fresh" found in some other operational semantics.

The input rules are given in Table 3.15, and are straightforward. Note that for live processes, input of any signal is always enabled.

The handling of reception of a process termination notification as defined in Table 3.16 is rather intricate, mimicking the real ERLANG implementations.

Processes are linked, bidirectionally, upon sending and reception of *link* signals (rules *link* and *linking*). Linked process may receive notification signals, indicating that one of the processes they are linked to is no longer alive, modelled in this semantics by the arrival of the *exited* signal.

The handling of the signal depends on a number of parameters (Table 3.16), notably the last parameter b (an ERLANG boolean) of the process state, modifiable by the `process_flag` built-in function (rules *trap_exit*), which determines whether exit notification are delivered to the message queue of the linked process.

- If the linked process completed normally ($v =$ `normal`) and the `trap_exit` flag has not been set ($b =$ `false`), or if the process mentioned in the signal is no longer linked (*pid*$' \notin$ *pl*), the process state is not modified (rule *exited*$_0$). Note that there is an implicit choice of either checking *pid*$' \notin$ *pl* here or not (since there can be a race between unlinking a process, and receiving a notification

$$\textit{silent}\ \frac{e \xrightarrow{\tau} e'}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle e', \mathit{pid}, q, pl, b\rangle}$$

$$\textit{output}_0\ \frac{e \xrightarrow{\mathit{pid}'!v} e' \qquad \mathit{pid}' \neq \mathit{pid}}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\mathit{pid}'!\mathit{message}(v)} \langle e', \mathit{pid}, q, pl, b\rangle}$$

$$\textit{output}_1\ \frac{e \xrightarrow{\mathit{pid}!v} e'}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle e', \mathit{pid}, q \cdot v, pl, b\rangle}$$

$$\textit{read}\ \frac{e \xrightarrow{\mathit{read}(q_1, v)} e'}{\langle e, \mathit{pid}, q_1 \cdot v \cdot q_2, pl, b\rangle \xrightarrow{\tau} \langle e', \mathit{pid}, q_1 \cdot q_2, pl, b\rangle}$$

$$\textit{test}\ \frac{e \xrightarrow{\mathit{test}(q)} e'}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle e', \mathit{pid}, q, pl, b\rangle}$$

$$\textit{termination}\ \frac{}{\langle v, \mathit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle \mathit{pid}, \{\langle \mathit{Pid}, \texttt{normal}\rangle \mid \mathit{Pid} \in pl\}\rangle}$$

$$\textit{exiting}\ \frac{e \xrightarrow{\mathit{exiting}(\{\texttt{'EXIT'},v\})} e'}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle \mathit{pid}, \{\langle \mathit{Pid}, v\rangle \mid \mathit{Pid} \in pl\}\rangle}$$

$$\textit{nocatch}\ \frac{e \xrightarrow{\mathit{exiting}(v)} e' \qquad \neg\exists V : \mathsf{erlangValue}.v = \{\texttt{'EXIT'}, V\}}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle \mathit{pid}, \{\langle \mathit{Pid}, \texttt{nocatch}\rangle \mid \mathit{Pid} \in pl\}\rangle}$$

Table 3.13: Process rules for expression evaluation

$$\textit{self}\ \frac{e \xrightarrow{\texttt{self}()\rightsquigarrow\{\texttt{result},\textit{pid}\}} e'}{\langle e, \textit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle e', \textit{pid}, q, pl, b\rangle}$$

$$\textit{link}\ \frac{e \xrightarrow{\texttt{link}(\textit{pid}')\rightsquigarrow\{\texttt{result},\texttt{true}\}} e'}{\langle e, \textit{pid}, q, pl, b\rangle \xrightarrow{\textit{pid}'!\textit{link}(\textit{pid})} \langle e', \textit{pid}, q, pl, b\rangle}$$

$$\textit{unlink}\ \frac{e \xrightarrow{\texttt{unlink}(\textit{pid}')\rightsquigarrow\{\texttt{result},\texttt{true}\}} e'}{\langle e, \textit{pid}, q, pl, b\rangle \xrightarrow{\textit{pid}'!\textit{unlink}(\textit{pid})} \langle e', \textit{pid}, q, pl, b\rangle}$$

$$\textit{kill}\ \frac{e \xrightarrow{\texttt{kill}(\textit{pid}',v)\rightsquigarrow\{\texttt{result},\texttt{true}\}} e'}{\langle e, \textit{pid}, q, pl, b\rangle \xrightarrow{\textit{pid}'!\textit{exit}(\textit{pid},v)} \langle e', \textit{pid}, q, pl, b\rangle}$$

$$\textit{trap\_exit}\ \frac{e \xrightarrow{\texttt{process\_flag}(\texttt{trap\_exit},b')\rightsquigarrow\{\texttt{result},b\}} e'}{\langle e, \textit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle e', \textit{pid}, q, pl, b'\rangle}$$

$$\textit{spawn}_0\ \frac{\textit{pid}' \neq \textit{pid} \qquad e \xrightarrow{\texttt{spawn}(f,[v_1,\ldots,v_n])\rightsquigarrow\{\texttt{result},\textit{pid}'\}} e'}{\langle e, \textit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle e', \textit{pid}, q, pl, b\rangle \parallel \langle f(v_1,\ldots,v_n), \textit{pid}', \epsilon, \emptyset, \textit{false}\rangle}$$

$$\textit{spawn}_1\ \frac{\textit{pid}' \neq \textit{pid} \qquad e \xrightarrow{\texttt{spawn\_link}(f,[v_1,\ldots,v_n])\rightsquigarrow\{\texttt{result},\textit{pid}'\}} e'}{\langle e, \textit{pid}, q, pl, b\rangle \xrightarrow{\tau} \langle e', \textit{pid}, q, pl \cup \{\textit{pid}'\}, b\rangle \parallel \langle f(v_1,\ldots,v_n), \textit{pid}', \epsilon, \{\textit{pid}\}, \textit{false}\rangle}$$

Table 3.14: Process rules for evaluation of process functions

$$\textit{input}\ \frac{}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\mathit{pid}?\mathit{message}(v)} \langle e, \mathit{pid}, q \cdot v, pl, b\rangle}$$

$$\textit{linking}\ \frac{}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\mathit{pid}?\mathit{link}(\mathit{pid}')} \langle e, \mathit{pid}, q, pl \cup \{\mathit{pid}'\}, b\rangle}$$

$$\textit{unlinking}\ \frac{}{\langle e, \mathit{pid}, q, pl, b\rangle \xrightarrow{\mathit{pid}?\mathit{unlink}(\mathit{pid}')} \langle e, \mathit{pid}, q, pl \setminus \{\mathit{pid}'\}, b\rangle}$$

Table 3.15: Process rules for external input

of its completion). We follow Barklund's text [Bar98, BV99] in additionally checking the linked condition.

- Otherwise (rule exited_1) the linked process may have requested that process termination notifications should be put in its mailbox ($b = \texttt{true}$).
- Otherwise, if process completed abnormally, and the linked process has taken no special action, then the linked process is also terminated abnormally (rule exited_2) with the same reason v.

Exit notifications can also be sent actively with the built-in function `kill` (in real ERLANG with `exit/2`). In the semantics such active notification are modelled with the signal `exit`. The semantic treatment (in Table 3.16) is analogous to the linked case, except that (i) there is no check that the process invoking `kill` was linked, and (ii) the reason `kill` causes unconditional process termination (rule kill_3) which is broadcast to linked process as the reason `killed`.

Lemma 7. *For any system (of one process) $\langle e, q, \mathit{pid}, pl, b\rangle$, and for all values v and process identifiers pid', there exists a system s such that*

$$\langle e, q, \mathit{pid}, pl, b\rangle \xrightarrow{\mathit{pid}?\mathit{exited}(\mathit{pid}', v)} s$$

and

$$\langle e, q, \mathit{pid}, pl, b\rangle \xrightarrow{\mathit{pid}?\mathit{exit}(\mathit{pid}', v)} s$$

Proof. We have to prove that the disjunction of the preconditions of the transition rules exit_0, exit_1 and exit_2 (the *exited* signal case) is a tautology (and analogous for the *exit* signal).

The *exited* case:

$$\begin{array}{ll} & (v = \texttt{normal} \wedge b = \texttt{false}) \vee \mathit{pid}' \notin pl \\ \vee & b = \texttt{true} \wedge \mathit{pid}' \in pl \\ \vee & v \neq \texttt{normal} \wedge b = \texttt{false} \wedge \mathit{pid}' \in pl \end{array}$$

$$\textit{exited}_0\ \frac{(v = \texttt{normal} \wedge b = \texttt{false}) \vee \textit{pid}' \notin \textit{pl}}{\langle e, \textit{pid}, q, \textit{pl}, b\rangle \xrightarrow{\textit{pid}?\textit{exited}(\textit{pid}',v)} \langle e, \textit{pid}, q, \textit{pl}, b\rangle}$$

$$\textit{exited}_1\ \frac{b = \texttt{true} \wedge \textit{pid}' \in \textit{pl}}{\langle e, \textit{pid}, q, \textit{pl}, b\rangle \xrightarrow{\textit{pid}?\textit{exited}(\textit{pid}',v)} \langle e, \textit{pid}, q \cdot \{\texttt{'EXIT'}, \textit{pid}', v\}, \textit{pl}, b\rangle}$$

$$\textit{exited}_2\ \frac{v \neq \texttt{normal} \wedge b = \texttt{false} \wedge \textit{pid}' \in \textit{pl}}{\langle e, \textit{pid}, q, \textit{pl}, b\rangle \xrightarrow{\textit{pid}?\textit{exited}(\textit{pid}',v)} \langle \textit{pid}, \{\langle \textit{Pid}, v\rangle \mid \textit{Pid} \in \textit{pl}\}\rangle}$$

$$\textit{kill}_0\ \frac{v = \texttt{normal} \wedge b = \texttt{false}}{\langle e, \textit{pid}, q, \textit{pl}, b\rangle \xrightarrow{\textit{pid}?\textit{exit}(\textit{pid}',v)} \langle e, \textit{pid}, q, \textit{pl}, b\rangle}$$

$$\textit{kill}_1\ \frac{v \neq \texttt{kill} \wedge b = \texttt{true}}{\langle e, \textit{pid}, q, \textit{pl}, b\rangle \xrightarrow{\textit{pid}?\textit{exit}(\textit{pid}',v)} \langle e, \textit{pid}, q \cdot \{\texttt{'EXIT'}, \textit{pid}', v\}, \textit{pl}, b\rangle}$$

$$\textit{kill}_2\ \frac{v \neq \texttt{kill} \wedge v \neq \texttt{normal} \wedge b = \texttt{false}}{\langle e, \textit{pid}, q, \textit{pl}, b\rangle \xrightarrow{\textit{pid}?\textit{exit}(\textit{pid}',v)} \langle \textit{pid}, \{\langle \textit{Pid}, v\rangle \mid \textit{Pid} \in \textit{pl}\}\rangle}$$

$$\textit{kill}_3\ \frac{v = \texttt{kill}}{\langle e, \textit{pid}, q, \textit{pl}, b\rangle \xrightarrow{\textit{pid}?\textit{exit}(\textit{pid}',v)} \langle \textit{pid}, \{\langle \textit{Pid}, \texttt{killed}\rangle \mid \textit{Pid} \in \textit{pl}\}\rangle}$$

Table 3.16: Process rules for handling exit notifications

is trivially a tautology (assuming $b \in \{\texttt{true}, \texttt{false}\}$).

The *exit* case:

$$\begin{array}{ll} & v = \texttt{normal} \wedge b = \texttt{false} \\ \vee & v \neq \texttt{kill} \wedge b = \texttt{true} \\ \vee & v \neq \texttt{kill} \wedge v \neq \texttt{normal} \wedge b = \texttt{false} \\ \vee & v = \texttt{kill} \end{array}$$

is also a tautology. □

Lemma 8. *For any system (of one process) $\langle e, q, \textit{pid}, \textit{pl}, b\rangle$, and for all values v and process identifiers $\textit{pid}'$, and all systems s, at most one of the transition rules* exited$_0$, exited$_1$ *and* exited$_2$ *(and analogously at most one of the rules* exit$_0$, exit$_1$, exit$_2$, exit$_3$ *are applicable).*

Proof. Trivially the preconditions of each of the transition rules are mutually exclusive. □

$$\textit{tlink}\ \frac{plm' = plm \cup \{\langle \textit{pid}', \texttt{noproc}\rangle\}}{\langle \textit{pid}, plm\rangle \xrightarrow{\textit{pid}?\textit{link}(\textit{pid}')} \langle \textit{pid}, plm'\rangle}$$

$$\textit{tinput}\ \frac{\neg\exists Pid' : \mathsf{erlangPid}.s = \textit{link}(Pid')}{\langle \textit{pid}, plm\rangle \xrightarrow{\textit{pid}?s} \langle \textit{pid}, plm\rangle}$$

$$\textit{tnotify}\ \frac{\langle \textit{pid}', v\rangle \in plm}{\langle \textit{pid}, plm\rangle \xrightarrow{\textit{pid}'!\textit{exited}(\textit{pid},v)} \langle \textit{pid}, plm \setminus \{\langle \textit{pid}', v\rangle\}\rangle}$$

Table 3.17: Process rules for terminated processes

Table 3.17 defines the signal input rules of terminated ERLANG processes. Note for instance the rule *tlink*. If a link request arrives at a terminated process then it will reply, eventually, with the termination notification `noproc`. The rule *tnotify* controls how termination notifications are sent. They are sent, as all other signals, through binary message passing, and notifications are sent out unordered.

Finally consider Table 3.18 which presents the rules for the parallel composition operator. The rules *interleave*$_1$, *interleave*$_2$ and *interleave*$_3$ also come in symmetric variants, where the left system performs a transition step. As usual this is an interleaving operator, except for the communication rule *com* either the left or the right system can perform a transition step, but not both.

As described in the standard ERLANG textbook [AVWW96], any ERLANG implementation must satisfy the following scheduling criteria:

- The scheduling algorithm must be *fair*, that is, any process which can be run will be run, if possible in the same order as they became runnable.
- No process will be allowed to block the machine for a long time. A process is allowed to run for a short period of time, called a *time slice*, before it is rescheduled to allow another runnable process to run.

The second criterion relates to real-time performance, which is not modelled by our semantics. The first criterion, however, does restrict the permissible execution sequences. Consider for instance the system $\langle \texttt{loop}(), q, \textit{pid}, pl, b\rangle \parallel \langle \textit{pid}''!0, q', \textit{pid}', pl', b'\rangle$, executing the function `loop() -> loop().` Clearly the execution sequence which only performs actions from the left process is not permitted by the criteria. Our semantics, however, permits such a sequence. The approach taken in this thesis, and indeed in our previous work [DFG98b], is to encode such fairness assumptions as part of the specification of correctness properties in the logic.

In the rule for process output *interleave*$_1$, the condition $\textit{pids}(s_1)' \cap \textit{pids}(s_2) = \emptyset$ ensures that any new processes in ${s_1}'$ are assigned unique process identifiers. Second the condition $\textit{pid} \notin \textit{pids}(s_2)$ in the rule *interleave*$_2$ permits outputs only to processes not in the same system, much as the rule *output*$_0$ does for the case of a singleton process.

$$\textit{com}\ \frac{s_1 \xrightarrow{\textit{pid}!s} s_1' \quad s_2 \xrightarrow{\textit{pid}?s} s_2'}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2'}$$

$$\textit{interleave}_1\ \frac{s_1 \xrightarrow{\tau} s_1' \quad \textit{pids}(s_1') \cap \textit{pids}(s_2) = \emptyset}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2}$$

$$\textit{interleave}_2\ \frac{s_1 \xrightarrow{\textit{pid}?s} s_1'}{s_1 \parallel s_2 \xrightarrow{\textit{pid}?s} s_1' \parallel s_2}$$

$$\textit{interleave}_3\ \frac{s_1 \xrightarrow{\textit{pid}!s} s_1' \quad \textit{pid} \notin \textit{pids}(s_2)}{s_1 \parallel s_2 \xrightarrow{\textit{pid}!s} s_1' \parallel s_2}$$

Table 3.18: Process communication (symmetrical rules omitted)

It may come as a surprise that the semantics for an asynchronous message passing language uses a process algebra communication scheme in the rule *com*. There are two points to be made: as explained earlier we want to obtain a context free account of processes, and thus prescribing an input action is completely natural, in particular considering the timeout clause in the `receive` which makes the arrival time of messages crucially important. Second, as Theorem 10 shows, input actions are always enabled, and we thus regain the asynchronous nature of communication found in regular ERLANG.

In the semantics messages are never lost. The semantics communication (rule *com*) is such that if two consecutive messages m_1 and m_2 are sent by process p_1 to process p_2 then message m_1 will always arrive before message m_2.

Note that terminated processes are not silent, and can be queried by sending link signals to them. A terminated process will respond to such signals by issuing an exit `noproc` reply.

Next some results about the ERLANG systems and the ERLANG system transition relation will be summarised.

Proposition 6 (System Derivations are Finite)**.** *For each system s there is a finite bound on the depth of inferences of a transition from s.*

Proof. By induction on the structure of the ERLANG systems. In the base case, the singleton process, there are no rules with a system transition as a premise, and from Proposition 2 the result immediately follows. Consider the parallel compositions of two systems s_1 and s_2, with bounded derivations. But then clearly the parallel compositions have only bounded derivations as well. □

Proposition 7 (Well-Formed Systems Are Invariantly Well-Formed)**.** *For any well-formed system s, if $s \xrightarrow{\alpha} s'$ for some s', then s' is also well-formed.*

Proof. The proposition will be proved using induction on the size of derivations. Consider all the system rules. No base case rule (the singleton process rules) changes a process identifier so only the rules introducing new processes needs to be considered. In the rules *spawn*$_0$ and *spawn*$_1$ the condition that $pid \neq pid'$ clearly suffices. For the rule *interleave*$_1$ we know by assumption that $s_1 \parallel s_2$ is well-formed. From the definition of well-formedness follows that $pids(s_1) \cap pids(s_2) = \emptyset$ and *wf* s_1 and *wf* s_2. From the induction hypothesis it follows that *wf* $s_1{}'$, and the transition condition is $pids(s_1{}') \cap pids(s_2) = \emptyset$ but then *wf* $s_1{}' \parallel s_2$ follows. For the rules *com*, *interleave*$_2$ and *interleave*$_3$ it suffices to first establish the easy lemma, not proved here, that for all s', pid, v, if $s \xrightarrow{pid?v} s'$ or $s \xrightarrow{pid!v} s'$ then *wf* s' and $pids(s) = pids(s')$. □

Proposition 8 (Process Determinacy)**.** *For any process p, and systems s, s', if $p \xrightarrow{\alpha} s$ and $p \xrightarrow{\alpha} s'$ then $s\rho = s'\rho$ where ρ is a substitution that renames process identifiers.*

Proof. A case analysis on the proof rules of the singleton processes and actions. Consider first a live process p. Recall the lemmas 3, and 4, which states that expression transitions are deterministic with respect to the same actions, and that only queue or process-state function call actions give rise to multiple expression level transitions. For the output action there is only one process rule *output*$_0$ applicable, which is derived from an expression rule.

For the input case we need to establish that input rules *input*, *linking*, *unlinking*, *exited*$_i$ and *exit*$_i$ are all mutually exclusive. The only complications are *exited*$_2$ and *exited*$_3$, but here the boolean parameter separates the cases. Similarly the *exit*$_i$ rules are all mutually exclusive.

For the silent step case we have to consider only actions "of the same kind", because of Proposition 4, since the expression level transition semantics is deterministic with respect to other action types, and the process level rules for deriving silent transitions all have premises that are mutually exclusive with respect to any action.

Both the queue and the process state transition rules clearly give rise to an infinite number of actions, leading to different expression states. For the process state action, for all rules other than *spawn*$_0$ and *spawn*$_1$, clearly the resulting expression is uniquely determined by the process function name. For the case of transitions due to spawn the results are identical up to a renaming of the process identifier of the newly spawned process (and thus the substitution ρ in the proposition above).

Next we consider the queue case. Suppose both the action $read(q_1, v_1)$ and an action $read(q_2, v_2)$ is derivable leading to different expressions e_1 and e_2. From the determinacy of the expression transition relation we get that $q_1 \neq q_2$ or $v_1 \neq v_2$. But clearly both are true, so q_2 must be a proper prefix of q_1, i.e., $\exists q_2{}''$ such that $q_1 = q_2 \cdot v_2 \cdot q_2{}''$ or vice versa if q_1 is a proper prefix of q_2, but this case is symmetrical. Now consider expression transition rule *receive*, which must have derived the transition labelled $read(q_2, v_2)$. A premise for its derivation is *result* $v_2\ m_I\ e'$ for some clause m_I and expression e'. Further, since an expression action labelled $read(q_1, v_1)$ was also derivable we have the premise (also from the definition of the *receive* rule) $\forall I.\neg(qmatches\ q\ m_I)$, and thus specifically (from the definition of *qmatches*) it follows that $\neg matches\ v_2\ m_I$. But from Lemma 2 follows that *matches* $v_2\ m_I$, which

leads to a contradiction.

The cases of both a read action $\mathit{read}(q_1, v)$ and a test action $\mathit{test}(q)$ are analogous and not shown.

Finally the transition rules of the terminated processes, trivially satisfy the conditions. □

The condition above on the substitution can naturally be sharpened to take place only under the silent transitions and only require a substitution of variables that do not occur in p.

Next we establish an important result guaranteeing that if input of any signal is enabled then all signals can be input.

Proposition 9 (Input Enabledness-1)**.** *If a system $s \xrightarrow{\mathit{pid}?sig} s'$ for some pid, signal sig and system s' then for any signal sig' there is a system s'' such that $s \xrightarrow{\mathit{pid}?sig'} s''$.*

Proof. By induction on the length of derivations. Only the input base cases are non-trivial.

For live processes, for signals which are not exit notifications or exit requests, the result is trivial since the rules have no conditions. Next we have to check that the rules exit_i and kill_i cover all such signals, and possible process states, which has been established in Lemma 7.

It remains to consider the terminated process rules, which are trivially enabled for all input values. □

Next it is proved that if a process identifier belongs to a process in a system, then that process can input any signal. This property is a requirement for regaining asynchronous communication from the synchronous scheme used by the *com* rule.

Proposition 10 (Input Enabledness-2)**.** *For all process identifiers pid and systems s, if $\mathit{pid} \in \mathit{pids}(s)$ then for all signals sig there exists a system s' such that $s \xrightarrow{\mathit{pid}?sig} s'$.*

Proof. We will show that the transition $s \xrightarrow{\mathit{pid}?0} s'$ is enabled for some s'. It then follows from Proposition 9 that for any signal sig there is a system s'' such that $s \xrightarrow{\mathit{pid}?sig} s''$.

The proof strategy is induction on the structure of the system s. If s is a singleton process then the process identifier of s must be pid. But then, trivially, either the rule *input* (for live processes) or the rule *tinput* (for terminated processes) is applicable.

Consider instead a parallel composition $s = s_1 \parallel s_2$. From the definition of $\mathit{pids}(s)$ either (or both, since well-formedness is not taken into account) $\mathit{pid} \in \mathit{pids}(s_1)$ or $\mathit{pid} \in \mathit{pids}(s_2)$. Apply the induction hypothesis and the rule $\mathit{interleave}_2$ to derive the input transition. □

The relation between process identifiers and input transitions is obvious.

Proposition 11. *$pid \in pids(s)$ if and only if there exists a signal sig and system s' such that $s \xrightarrow{pid?sig} s'$.*

Proof. An easy induction over the length of derivations. □

Let $s \rightarrow^* s'$ mean that there exists a sequence of coupled transitions from the system s and ending in s', in the obvious manner.

Proposition 12 (Input Invariance)**.** *If a system s has a transition labelled by $\xrightarrow{pid?sig}$ for some pid and signal sig then for any derivative s' such that $s \rightarrow^* s'$ there exists a system s'' and signal sig' such that $s \xrightarrow{pid?sig'} s''$.*

Proof. This observations follows from the observation that the process identifiers of a system increase monotonically over derivations (not proven here, but an easy induction over the length of a derivation), and Proposition 10. □

Proposition 13 (Input Determinacy)**.** *For any systems s, s', s'', process identifier pid and signal sig if $s \xrightarrow{pid?sig} s'$ and $s \xrightarrow{pid?sig} s''$ then $s' = s''$.*

Proof. By induction on the length of derivations. For singleton processes this follows from Proposition 8. So consider the case of a parallel composition $s_1 \parallel s_2$. Assume $s \xrightarrow{pid?sig} s'$ and $s \xrightarrow{pid?sig} s''$. From the assumption on well-formedness and Proposition 11 it follows that either s_1 or s_2 must be responsible for both transitions. But then the induction hypothesis is applicable. □

3.2.6 Bisimilarity for Systems

Here the notion of bisimilarity is extended to the systems.

Definition 40. *A binary relation S is a system bisimulation if $(s_1, s_2) \in S$ implies, for all actions a,*

- *Whenever $s_1 \xrightarrow{\alpha} s_1'$ then, for some s_2', $s_2 \xrightarrow{\alpha} s_2'$, and $(s_1', s_2') \in S$, and vice versa.*

Definition 41. *The systems s_1 and s_2 are system bisimilar, written $s_1 \sim_s s_2$, if $(s_1, s_2) \in \mathcal{S}$ for some system bisimulation $\mathcal{S}$.*

System bisimilarity is a very strong equivalence relation, and not very practically useful. For instance, two systems cannot be bisimilar if they have differently named processes, since input actions (giving away the process name) are always enabled.

Proposition 14. *$\sim_s$ is a congruence.*

Proof. We have to consider only the case of a parallel composition, so assume $s_1 \sim_s s_2$, and prove that for any system s_3, $s_1 \parallel s_3 \sim_s s_2 \parallel s_3$ (and vice versa for the right-hand side of the parallel composition but this case will be omitted).

We claim that $S = \{(s_1 \parallel s_3, s_2 \parallel s_3) \mid s_1 \sim_s s_2\}$ is a system bisimulation, which would establish the needed result. So consider the transitions of $s_1 \parallel s_3 \xrightarrow{\alpha} s$. Crucially first note that $s_1 \sim_s s_2$ implies that $\mathit{pids}(s_1) = \mathit{pids}(s_2)$, since if not, then some input steps cannot be matched.

1. $s_1 \xrightarrow{\mathit{pid}?v} s_1'$, and $s = s_1' \parallel s_3$. But then we also have $s_2 \xrightarrow{\mathit{pid}?v} s_2'$, for some s_2', and $s_1' \sim_s s_2'$. But then also $s_2 \parallel s_3 \xrightarrow{\mathit{pid}?v} s_2' \parallel s_3$, and $(s_1' \parallel s_3, s_2' \parallel s_3) \in S$.

2. $s_1 \xrightarrow{\mathit{pid}!v} s_1'$, and $s = s_1' \parallel s_3$. But then we also have $s_2 \xrightarrow{\mathit{pid}!v} s_2'$, for some s_2', and $s_1' \sim_s s_2'$. But then also $s_2 \parallel s_3 \xrightarrow{\mathit{pid}!v} s_2' \parallel s_3$, and $(s_1' \parallel s_3, s_2' \parallel s_3) \in S$.

3. $s_1 \xrightarrow{\mathit{pid}!v} s_1'$, and $s = s_1' \parallel s_3$, and $s_3 \xrightarrow{\tau} s_3'$. But then we also have $s_2 \xrightarrow{\mathit{pid}!v} s_2'$, for some s_2', and $s_1' \sim_s s_2'$, and $s_3 \xrightarrow{\mathit{pid}?v} s_3'$ But then also $s_2 \parallel s_3 \xrightarrow{\tau} s_2' \parallel s_3'$, and $(s_1' \parallel s_3', s_2' \parallel s_3') \in S$.

4. $s_1 \xrightarrow{\tau} s_1'$, and $s = s_1' \parallel s_3$. But then we also have $s_2 \xrightarrow{\tau} s_2'$, for some s_2', and $s_1' \sim_s s_2'$. Since $\mathit{pids}(s_1) = \mathit{pids}(s_2)$ (from $s_1' \sim_s s_2'$) and $\mathit{wf}\ s_1' \parallel s_3$ then obviously $\mathit{wf}\ s_2' \parallel s_3$, and thus $s_2 \parallel s_3 \xrightarrow{\tau} s_2' \parallel s_3$, and $(s_1' \parallel s_3, s_2' \parallel s_3) \in S$.

The case of s_3 performing a step is analogous and not shown here. □

First a much expected fact about the parallel composition is provable.

Proposition 15 (Parallel Composition is Associative and Commutative)**.** *For any three systems s_1, s_2 and s_3 it holds that $s_1 \parallel s_2 \sim_s s_2 \parallel s_1$ and $s_1 \parallel (s_2 \parallel s_3) \sim_s (s_1 \parallel s_2) \parallel s_3$.*

Proof. The proof is standard, we show that

- The set $\{(s_1 \parallel s_2, s_2 \parallel s_1)\}$ which contains all pairs of systems s_1 and s_2, is a system bisimulation.
- The set $\{(s_1 \parallel (s_2 \parallel s_3), (s_1 \parallel s_2) \parallel s_3)\}$ which contains all triples of systems s_1, s_2 and s_3 is a system bisimulation.

We consider the case for commutativity. Suppose $s_1 \parallel s_2 \xrightarrow{\alpha} s'$. This is either because one of the components performs a single step, or there is a communication. We consider one alternative below. Suppose $s_1 \xrightarrow{\alpha} s_1'$ and $s' \equiv s_1 \parallel s_2$ but then $s_2 \parallel s_1 \xrightarrow{\alpha} s_2 \parallel s_1'$ under the same symmetric conditions for the derivation on the left-hand side of the parallel composition, and the resulting pair is in the bisimulation. □

Even though the notion of bisimilarity is very strong we can still prove some practically useful facts about it. For instance, bisimilar expressions produce bisimilar systems:

Proposition 16. *Whenever* $e_1 \sim_e e_2$ *then for all* pid, q, pl, b *it holds that* $\langle e_1, pid, q, pl, b\rangle \sim_s \langle e_2, pid, q, pl, b\rangle$.

Proof. We claim that

$$\{(\langle e_1, pid, q, pl, b\rangle, \langle e_2, pid, q, pl, b\rangle) \mid e_1 \sim_c e_2\}$$
$$\bigcup$$
$$\{(\langle e_1, pid, q, pl, b\rangle \parallel s, \langle e_2, pid, q, pl, b\rangle \parallel s) \mid e_1 \sim_c e_2\}$$

is a system bisimulation.

We will consider actions by e, the case for e' is completely symmetrical. Consider any transition step by the expression e. Clearly the action is mimicked by e' and since the rest of the process state is identical, then the resulting systems must be identical (up to the derivatives of e and e'). Suppose e has finished its evaluation yielding a value v. Well, then so must also e' (since they are bisimilar) with an equivalent value. So consider a system action not involving an expression action. Clearly, since the process states are identical, and since no rule examines e (except to determine if the expression has finished its computation), the resulting systems are identical. □

3.2.7 Language Extension: Function Values

Regular ERLANG, since version 4.4, supports a form of lambda expression to permit the definition of anonymous functions that can be treated as values. In this section we describe the modifications of the semantics required in order to support function values.

- A function value is constructed using the syntax `fun` m `end` where, as usual, m is a match (a sequence of clauses - pattern, guard and expression triples). The syntax `fun` a/i, available in regular ERLANG, where a is an atom used to refer to a named function, and i is an arity, is not supported in this semantics. The predicate *isFunctionValue* is assumed to recognise the function values. Syntactically it is expected that the function value occur in a context that binds all the free variables in the match ($fv(m)$), which is ensured by restrictions on function definitions.
- The built-in function `function` is assumed, which recognises function values, and is permitted in guards. Further, the set of tuple values and function values are considered distinct, i.e., a function value is not represented using a tuple (in contrast with regular ERLANG), thus the built-in function `tuple` will return `false` when applied to a function value.
- To reflect the fact that an anonymous function represents a value, the definition of the Erlang Values erlangValue is considered to be extended with the function values.

- In the operational semantics one new rule *fun*$_v$ is added, and an existing rule *fun*$_3$ is modified:

$$\textit{fun}_v \ \frac{\texttt{case } \{v_1,\ldots,v_n\} \texttt{ of } m \texttt{ end} \xrightarrow{\tau} e'}{\texttt{fun } m \texttt{ end}(v_1,\ldots,v_n) \xrightarrow{\tau} e'}$$

$$\textit{fun}_3 \ \frac{\begin{array}{c}\neg(\textit{isProcFun } v) \wedge \neg(\textit{isDefined } v) \\ \wedge \quad \neg(\textit{isFunctionValue } v) \wedge \neg(\textit{isBIF } v)\end{array}}{v(v_1,\ldots,v_n) \xrightarrow{\textit{exiting}(\texttt{badfun})} \texttt{bottom}}$$

- The *fv* function that computes free variables is extended to the case of function values: $\mathit{fv}(\texttt{fun } m \texttt{ end}) = \mathit{fv}(m)$.

- Crucially the substitution function has to be extended, to take into account binding of variables in Function Values. The implementations of regular ERLANG ensure that variables occurring in the pattern of a clause in the function head are fresh (are not bound by surrounding matches), whereas variables occurring in the function expressions but not patterns must be bound by surrounding matches. Consider the function definition

```
f(X,Y,Pid) -> Pid!(fun X -> {X,Y} end).
```

  If, say, `f` is called $\texttt{f}(1, 2, \textit{Pid})$, where *Pid* is a valid process identifier, then the value `fun X -> {X,2} end` will be sent to *Pid*.

  To achieve this formally, first define a function $\textit{rs}(\widetilde{v}, \widetilde{V}, \widetilde{V_p}, e)$ (restrict-substitution) that constructs new sequences $\widetilde{v'}$ and $\widetilde{V'}$ from $\widetilde{v}$ and $\widetilde{V}$ such that $V_i \in \widetilde{V'}$ and $v_i \in \widetilde{v'}$ if $V_i \notin \widetilde{V_p}$, and then applies the resulting substitution to e as $e\{\widetilde{v'}/\widetilde{V'}\}$.

  The extension of the substitution function to Function Values then becomes

$$\texttt{function } p_1 \texttt{ when } g_1 \texttt{ -> } e_1 \texttt{;} \ldots \texttt{;} p_n \texttt{ when } g_n \texttt{ -> } e_n \texttt{ end}\{\widetilde{v}/\widetilde{V}\}$$

$$= \texttt{function } \begin{array}{c} p_1 \texttt{ when } \textit{rs}(\widetilde{v}, \widetilde{V}, \mathit{fv}(p_1), g_1) \texttt{ -> } \textit{rs}(\widetilde{v}, \widetilde{V}, \mathit{fv}(p_1), e_1) \\ \texttt{;} \ldots \texttt{;} \\ p_n \texttt{ when } \textit{rs}(\widetilde{v}, \widetilde{V}, \mathit{fv}(p_1), g_n) \texttt{ -> } \textit{rs}(\widetilde{v}, \widetilde{V}, \mathit{fv}(p_1), e_n) \end{array} \texttt{ end}$$

Chapter 4

A Proof System for Reasoning about ERLANG Code

The proof system for reasoning about ERLANG code is of Gentzen-type [Gen69] and sequent based. Sequents are of the shape $\Gamma \vdash \Delta$ where Γ and Δ are finite multisets of formulas. The multisets will sometimes be referred to as sequences when the distinction is not relevant. The formulas in Γ will often be referred to as the left-hand side, or the assumptions, and Δ will be referred to as the right-hand side. In this chapter we give the formal semantics of sequents, present the rules of the proof system, and establish their soundness.

In the following we consider only formulas that contain no free predicate variables and which satisfy the type rules of Chapter 2. That is, for a formula ϕ it is required that, assuming a mapping LV of the free term variables in ϕ to types, the type checking condition $LV \vdash \phi : \mathsf{prop}$ holds. Recall from Chapter 2 that a valuation maps term variables and predicate variables to appropriate values, i.e., values of some sort or function abstractions.

Definition 42 (Sequents and Valuations)**.**

- *A valuation ρ* respects *a type interpretation LV if ρ maps term variables in LV to values of the type specified in LV.*
- *A* sequent *is an expression of the shape $\Gamma \vdash \Delta$ where Γ and Δ are (possibly empty) multisets of formulas written $\phi_1, \ldots, \phi_n$ and $\psi_1, \ldots, \psi_m$ such that there exists a type interpretation LV so that for each ϕ_i and ψ_i the formula type checking conditions $LV \vdash \phi_i : \mathsf{prop}$ and $LV \vdash \psi_i : \mathsf{prop}$ are derivable.*
- *A valuation ρ is said to* validate *a formula ϕ when $\| \phi \|_\rho \neq \emptyset$.*
- *The sequent $\Gamma \vdash \Delta$, with an implicit type interpretation LV, is* valid*, written $\Gamma \models \Delta$, if for all valuations ρ that respect LV the following holds: if all formulas in Γ are validated by ρ, then some formula in Δ is validated by ρ.*

- *Let* $fv(\Gamma \vdash \Delta)$ *denote the free variables of the sequent* $\Gamma \vdash \Delta$*, i.e.,* $fv(\phi_1) \cup \ldots \cup fv(\phi_n) \cup fv(\psi_1) \cup \ldots \cup fv(\psi_m)$.

Further extend the notion of a substitution applied to a formula to multisets of formulas Γ with the syntax $\Gamma\{t/X\}$, when v is a term and X is a variable.

Next the notion of a proof rule is formalised.

Definition 43. *A* proof rule *is a triple consisting of a* conclusion *sequent* $\Gamma \vdash \Delta$*, a possibly empty finite set of sequents* $\{\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n\}$ *representing the* premises*, and a symbol representing the name of the rule.*

Henceforth proof rules will always be displayed graphically on the format

$$\textsc{name}\frac{\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

where NAME is the unique name of the rule. If a proof rule has no premises it will be referred to as an axiom proof rule. Consider as an example the rule for introducing disjunctions on the left hand side:

$$\vee_{\text{L}}\frac{\Gamma, \phi_1 \vdash \Delta \qquad \Gamma, \phi_2 \vdash \Delta}{\Gamma, \phi_1 \vee \phi_2 \vdash \Delta}$$

The symbols Γ and Δ in the proof rule range over any, possibly empty, multiset of formulas. Formally the rule $\vee_{\text{L}}$ here is a schematic way to define a collection of proof rules, each obtained by instantiating Γ, ϕ_1, ϕ_2 and Δ. In the following we will slightly abuse terminology and simply refer to such schemas as proof rules.

The intuition for the notion of a proof rule is that, whenever all the premisses of a rule are valid then so must the conclusion be. In this thesis most reasoning starts with the conclusion; we prove a formula by finding a proof rule that yields a set of premisses sufficient to establish its validity. In other words, typically proof rules are read bottom-up: we match the proof goal we have with the conclusion of a proof rule and derive a set of new goals which must be proved in turn.

The notation "X fresh", where X is a term variable, will be used in rules to describe the classical side condition that any variable X may be chosen as long as it does not occur free in the conclusion sequent $\Gamma \vdash \Delta$. However, in this thesis, a further restriction is placed on the choice of a fresh ordinal variable κ which must be chosen globally fresh in the proof tree. That is, there may not be any other proof node in which κ occurs free. This additional condition simplifies the treatment of ordinal induction in Section 4.4.

4.1 Proof Rules for Classical First-Order Logic

In this section the proof rules that do not involve fixed point reasoning are presented. These proof rules can be separated into four categories, and are defined in Tables 4.1 and 4.2. The majority of the rules are standard from accounts of Gentzen–style proof systems.

- The *structural rules* govern the introduction and elimination of formulas. For example, the proof rule ID allows discharging a goal whenever an identical formula occurs on both sides of the turnstile.
- The *logical rules* introduce the logical connectives to the left and to the right of the turnstile. One example is the proof rule $\vee_{\mathsf{L}}$ which governs the introduction of a disjunction on the left hand side.
- The *equality* rules account for equational reasoning. The proof rules state that equality is reflexive, symmetric and transitive, and that it is a congruence.
- The *equality rules for terms of freely generated sorts* With a sort of *freely-generated* terms we mean a sort where no equalities are postulated between syntactically different terms. An example of a freely generated sort is the lists. Two lists are equal only if their head elements are equal, and their tails are equal. An example of a sort that is not freely generated is the queues with the empty queue ϵ since the equality $q \cdot \epsilon = q$ holds where $\cdot$ is the concatenation constructor. As indicated additional proof rules admit structural decomposition of of terms of such sorts.

Consider the proof rule $\mathsf{CINEQ}_{\mathsf{L}}$ as an example. Suppose an assumption states that two terms are equivalent, and that these terms are of freely generated sorts. Moreover, the assumption states that the terms have different head constructors. Clearly this is contradictory. That is, there can be no valuation that validates the assumption, and so the sequent is trivially valid.

Theorem 2 (Local Soundness)**.** Each of the rules in Table 4.1 and Table 4.2 is sound, i.e. the conclusion is a valid sequent whenever all premises are so and all side conditions hold.

Proof. Here we prove the soundness of only a few rules; the proofs of the remaining rules are either standard or similar to these.

$\exists_{\mathsf{L}}$**:** Assume that the premise $\Gamma, \phi\{X'/X\} \vdash \Delta$ is valid, that X' is fresh, and that ρ validates the assumptions in $\Gamma, \exists X : S.\phi$. That ρ validates $\exists X : S.\phi$ means that $\bigcup_{v \in S} \left(\| \phi \|_{\rho[v/X]} \right) \neq \emptyset$, i.e., there is an element $v \in S$, such that $\| \phi \|_{\rho[v/X]} \neq \emptyset$. Consider the valuation $\rho[v/X']$, clearly it must validate $\Gamma, \phi\{X'/X\}$ since X' does not occur in Γ. But then Δ must be valid under $\rho[v/X']$, and since X' does not occur in Δ, the formulas in Δ must also be valid under ρ.

$\exists_{\mathsf{R}}$**:** Assume that the premise $\Gamma \vdash \phi\{t/X\}, \Delta$ is valid, that $t \in S$, and that ρ validates the assumptions in Γ. Hence it follows that ρ validates either $\phi\{t/X\}$ or one of the formulas in Δ. Assume that none of the formulas in Δ are validated, but $\phi\{t/X\}$ is validated, since otherwise the proof immediately succeeds. The semantics of $\exists X : S.\phi$ under ρ is $\bigcup_{v \in S} \left(\| \phi \|_{\rho[v/X]} \right) \neq \emptyset$. By choosing t for v, we can extract the term $\| \phi \|_{\rho[t/X]}$ which is identical to $\| \phi\{t/X\} \|_{\rho}$ which is

Structural Rules

$$\mathsf{ID}\frac{-}{\Gamma, \phi \vdash \phi, \Delta}$$

$$\mathsf{W_L}\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad \mathsf{W_R}\frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta}$$

$$\mathsf{CUT}\frac{\Gamma \vdash \phi, \Delta \qquad \Gamma, \phi \vdash \Delta}{\Gamma \vdash \Delta}$$

Logical Rules

$$\neg_{\mathsf{L}}\frac{\Gamma \vdash \Delta, \phi}{\Gamma, \neg\phi \vdash \Delta} \qquad \neg_{\mathsf{R}}\frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}$$

$$\vee_{\mathsf{L}}\frac{\Gamma, \phi_1 \vdash \Delta \qquad \Gamma, \phi_2 \vdash \Delta}{\Gamma, \phi_1 \vee \phi_2 \vdash \Delta} \qquad \vee_{\mathsf{R}}\frac{\Gamma \vdash \phi_1, \phi_2, \Delta}{\Gamma \vdash \phi_1 \vee \phi_2, \Delta}$$

$$\exists_{\mathsf{L}}\frac{\Gamma, \phi\{X'/X\} \vdash \Delta}{\Gamma, \exists X : S.\phi \vdash \Delta}\ X' \textit{ fresh} \qquad \exists_{\mathsf{R}}\frac{\Gamma \vdash \phi\{t'/X\}, \Delta}{\Gamma \vdash \exists X : S.\phi, \Delta}\ t' \in S$$

$$\mathsf{APPLY_L}\frac{\Gamma, \phi\{t'/X\} \vdash \Delta}{\Gamma, (\lambda X : S.\phi)\ t' \vdash \Delta} \qquad \mathsf{APPLY_R}\frac{\Gamma \vdash \phi\{t'/X\}, \Delta}{\Gamma \vdash (\lambda X : S.\phi)\ t', \Delta}$$

Equality Rules

$$\mathsf{REFL}\frac{-}{\Gamma \vdash t = t, \Delta}$$

$$\mathsf{SYMM_L}\frac{\Gamma, t_2 = t_1 \vdash \Delta}{\Gamma, t_1 = t_2 \vdash \Delta} \qquad \mathsf{SYMM_R}\frac{\Gamma \vdash t_2 = t_1, \Delta}{\Gamma \vdash t_1 = t_2, \Delta}$$

$$\mathsf{TRANS_L}\frac{\Gamma, t_1 = t_3 \vdash \Delta}{\Gamma, t_1 = t_2, t_2 = t_3 \vdash \Delta} \qquad \mathsf{TRANS_R}\frac{\Gamma \vdash t_1 = t_3, \Delta}{\Gamma \vdash t_1 = t_2, t_2 = t_3, \Delta}$$

$$\mathsf{SUBST}\frac{\Gamma\{t_2/X\} \vdash \Delta\{t_2/X\}}{\Gamma\{t_1/X\}, t_1 = t_2 \vdash \Delta\{t_1/X\}}$$

Table 4.1: Standard Proof Rules

$$\mathsf{CEQ_L}\frac{\Gamma, t_1 = t_1', \ldots, t_n = t_n' \vdash \Delta}{\Gamma, op(t_1, \ldots, t_n) = op(t_1', \ldots, t_n') \vdash \Delta}$$

$$\mathsf{CEQ_R}\frac{\Gamma \vdash t_1 = t_1', \Delta \quad \cdots \quad \Gamma \vdash t_n = t_n', \Delta}{\Gamma \vdash op(t_1, \ldots, t_n) = op(t_1', \ldots, t_n'), \Delta}$$

$$\mathsf{CINEQ_L}\frac{-}{\Gamma, op(t_1, \ldots, t_i) = op'(t_1', \ldots, t_j') \vdash \Delta}\ op \neq op'$$

Table 4.2: Rules for Terms of Freely Generated Sorts

known to be nonempty since $\phi\{t/X\}$ is valid under ρ. So $\exists X : S.\phi$ must be valid under ρ.

$\mathsf{APPLY_R}$**:** Assume that the premise of the rule is valid. Consider an arbitrary valuation ρ that validates all assumptions in Γ, and at least one formula in $\phi\{t'/T\}, \Delta$. We will show that ρ validates also the consequence of the rule. Assume further that ρ does not validate any formula in Δ (otherwise trivially ρ validates the consequence also), so it must validate $\phi\{t'/T\}$. Since $\| \lambda Y : S.\phi \|_\rho = \lambda X : S.\| \phi \|_{\rho[X/Y]}$ then $\| (\lambda Y : S.\phi)t' \|_\rho = \| \phi \|_{\rho[t'/Y]} = \| \phi\{t'/Y\} \|_\rho$, which is exactly the assumption that ρ validates $\phi\{t'/T\}$, and thus ρ validates also the consequence.

□

4.2 Pre-Proofs

Here some terminology concerning proofs, such as the notion of a substitution instance of a rule, and the notion of a pre-proof, is formalised.

Definition 44. *A* substitution instance *of a proof rule* R *of the shape*

$$\mathsf{R}\frac{\Gamma_1 \vdash \Delta_1 \qquad \ldots \qquad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

is a tree containing one root node with n direct descendants. The tree is labelled by sequents

$$\frac{\Gamma_1' \vdash \Delta_1' \qquad \ldots \qquad \Gamma_n' \vdash \Delta_n'}{\Gamma' \vdash \Delta'}$$

such that there exists a substitution ρ such that $\Gamma\rho = \Gamma'$ and for all $1 \leq i \leq n$ the multiset equalities $\Gamma_i\rho = \Gamma_i'$ and $\Delta_i\rho = \Delta_i'$ hold.

Definition 45. *A* pre-proof*, or* proof tree*, is a finite rooted tree such that all nodes of the tree are labelled by sequents, and which respects the local rules of the proof*

system in the sense that if a proof tree node π is labelled by a sequent δ and its children $\pi_1, \ldots, \pi_n$ are labelled by $\delta_1, \ldots, \delta_n$ then the subtree

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

is a substitution instance of some proof rule.

Let π range over pre-proof nodes and T range over pre-proofs. Let $sequent(\pi)$ refer to the sequent label of a node π. Further, let $children(\pi)$ refer to the possibly empty set of children of π and the parent of a node π (if it has a parent) with $parent(\pi)$.

A proof is intuitively a pre-proof such that all the leaf nodes are axioms, i.e. discharged due to proof rules such as ID or REFL, or they are instances of ancestor nodes in the proof structure, and such that the global discharge condition holds. The notion of a proof will be formally defined in Section 4.5.3.

As a convention the name of a proof rule will often be displayed on the right hand side of a proof node to indicate that the subtree rooted in the node is a substitution instance of the proof rule.

4.3 Derived Proof Rules

The derived proof rules in Table 4.3 will be used heavily, and are easily proven from the abbreviations and proof rules introduced so far. By "proven" we understand the fact that there exists a proof schema (a tree-like structure of proof rules) which can be applied to any instance of the conclusion in a derived proof rule, and through application of the proof rules in the proof schema derive its premisses.

For a trivial example consider the proof schema below which derives the proof rule $\wedge_{\mathsf{L}}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \phi_1, \phi_2 \vdash \Delta}{\Gamma \vdash \neg\phi_1, \neg\phi_2, \Delta}\ \neg_{\mathsf{R}},\neg_{\mathsf{R}}}{\Gamma \vdash \neg\phi_1 \vee \neg\phi_2, \Delta}\ \vee_{\mathsf{R}}}{\Gamma, \neg\left(\neg\phi_1 \vee \neg\phi_2\right) \vdash \Delta}\ \neg_{\mathsf{L}}}{\Gamma, \phi_1 \wedge \phi_2 \vdash \Delta}\ \text{abbrev.}$$

4.3.1 A Cut Rule for Terms

In Dam [Dam95] a rule named CUT for CCS terms was introduced to enable compositional reasoning. Similarly Dam et al. [DFG98b] uses a PROCESSCUT rule to similar effect in the treatment of parallel compositions in ERLANG. As Figure 4.1 shows an analogous rule TERMCUT is derivable using the classical CUT rule in our proof system.

$$\textsf{TERMCUT}\frac{\Gamma \vdash t' : \psi, \Delta \qquad \Gamma, X : \psi \vdash t : \phi, \Delta}{\Gamma \vdash t\{t'/X\} : \phi, \Delta}\ X \textit{ fresh}$$

$$\mathsf{FALSE_L}\frac{-}{\Gamma, \mathit{false} \vdash \Delta} \qquad \mathsf{TRUE_R}\frac{-}{\Gamma \vdash \mathit{true}, \Delta}$$

$$\mathsf{C_L}\frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad \mathsf{C_R}\frac{\Gamma \vdash \phi, \phi, \Delta}{\Gamma \vdash \phi, \Delta}$$

$$\wedge_{\mathsf{L}}\frac{\Gamma, \phi_1, \phi_2 \vdash \Delta}{\Gamma, \phi_1 \wedge \phi_2 \vdash \Delta} \qquad \wedge_{\mathsf{R}}\frac{\Gamma \vdash \phi_1, \Delta \qquad \Gamma \vdash \phi_2, \Delta}{\Gamma \vdash \phi_1 \wedge \phi_2, \Delta}$$

$$\forall_{\mathsf{L}}\frac{\Gamma, \phi\{t/X\} \vdash \Delta}{\Gamma, \forall X : S.\phi \vdash \Delta}\, t' \in S \qquad \forall_{\mathsf{R}}\frac{\Gamma \vdash \phi\{X'/X\}, \Delta}{\Gamma \vdash \forall X : S.\phi, \Delta}\, X' \textit{ fresh}$$

$$\Rightarrow_{\mathsf{L}}\frac{\Gamma \vdash \phi_1, \Delta \qquad \Gamma, \phi_2 \vdash \Delta}{\Gamma, \phi_1 \Rightarrow \phi_2 \vdash \Delta} \qquad \Rightarrow_{\mathsf{R}}\frac{\Gamma, \phi_1 \vdash \phi_2, \Delta}{\Gamma \vdash \Delta, \phi_1 \Rightarrow \phi_2}$$

Table 4.3: Derived Proof Rules

$$\pi : \frac{\dfrac{\Gamma, X : \psi \vdash t : \phi, \Delta}{\Gamma \vdash X : \psi \Rightarrow t : \phi, \Delta}\, \Rightarrow_{\mathsf{R}}}{\Gamma \vdash \forall X.\, (X : \psi \Rightarrow t : \phi)\, , \Delta}\, \forall_{\mathsf{R}}$$

$$\dfrac{\dfrac{\dfrac{-}{\Gamma, t\{t'/X\} : \phi \vdash t\{t'/X\} : \phi, \Delta}\, \mathsf{ID} \qquad \dfrac{\Gamma \vdash t' : \psi, \Delta}{\Gamma \vdash t\{t'/X\} : \phi, t' : \psi, \Delta}\, \mathsf{W_R}}{\Gamma, t' : \psi \Rightarrow t\{t'/X\} : \phi \vdash t\{t'/X\} : \phi, \Delta}\, \Rightarrow_{\mathsf{L}}}{\Gamma, \forall X.\, (X : \psi \Rightarrow t : \phi) \vdash t\{t'/X\} : \phi, \Delta}\, \forall_{\mathsf{L}} \qquad \pi}{\Gamma \vdash t\{t'/X\} : \phi, \Delta}\, \mathsf{CUT}$$

Figure 4.1: Derivation of the TERMCUT rule

Intuitively the proof rule expresses that if we can prove that (i) *the property ψ holds of the subterm t' of t*, and (ii) *t satisfies ϕ when the subterm t' has been replaced by any term satisfying ψ*, then we are allowed to conclude that the term *t satisfies ϕ*.

4.3.2 Proof Rules for Modalities

Convenient proof rules for modalities can easily be obtained, similar to the rules of Simpson [Sim95]. Recall that the modalities abbreviate the following formulas:

$$\begin{aligned} \langle\alpha\rangle\phi &\triangleq \lambda X : S.\exists X' : S.X \xrightarrow{\alpha} X' \wedge \phi\ X' \\ [\alpha]\phi &\triangleq \lambda X : S.\forall X' : S.X \xrightarrow{\alpha} X' \Rightarrow \phi\ X' \end{aligned}$$

These abbreviation yield the following set of derived proof rules:

$$\langle\alpha\rangle_{\mathsf{R}}\frac{\Gamma \vdash \exists X' : S.t \xrightarrow{\alpha} X' \wedge X' : \phi, \Delta}{\Gamma \vdash t : \langle\alpha\rangle\phi, \Delta} \qquad \langle\alpha\rangle_{\mathsf{L}}\frac{\Gamma, t \xrightarrow{\alpha} X', X' : \phi \vdash \Delta}{\Gamma, t : \langle\alpha\rangle\phi \vdash \Delta}\ X' \textit{ fresh}$$

$$[\alpha]_{\mathsf{L}}\frac{\Gamma, \forall X'.t \xrightarrow{\alpha} X' \Rightarrow X' : \phi \vdash \Delta}{\Gamma, t : [\alpha]\phi \vdash \Delta} \qquad [\alpha]_{\mathsf{R}}\frac{\Gamma, t \xrightarrow{\alpha} X' \vdash X' : \phi, \Delta}{\Gamma \vdash t : [\alpha]\phi, \Delta}\ X' \textit{ fresh}$$

Consider as an example the derivation of the proof rule $\langle\alpha\rangle_{\mathsf{L}}$:

$$\dfrac{\dfrac{\dfrac{\Gamma, t \xrightarrow{\alpha} X', X' : \phi \vdash \Delta}{\Gamma, \exists X' : S.t \xrightarrow{\alpha} X' \wedge X' : \phi \vdash \Delta}\ \exists_{\mathsf{L}},\wedge_{\mathsf{L}}}{\Gamma, t : \lambda X : S.\exists X' : S.X \xrightarrow{\alpha} X' \wedge X' : \phi \vdash \Delta}\ \mathsf{APPLY}_{\mathsf{R}}}{\Gamma, t : \langle\alpha\rangle\phi \vdash \Delta}\ abbrev.$$

As a further example the following monotonicity rules, well-known rules from standard Gentzen-type accounts of modal logic, are derivable.

$$\mathsf{MON1}\frac{\Gamma, S : \phi, S : \phi_1, \ldots, S : \phi_m \vdash S : \psi_1, \ldots, S : \psi_n, \Delta}{\Gamma, s : \langle\alpha\rangle\phi, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m \vdash s : \langle\alpha\rangle\psi_1, \ldots, s : \langle\alpha\rangle\psi_n, \Delta}\ S \textit{ fresh}$$

$$\mathsf{MON2}\frac{\Gamma, S : \phi_1, \ldots, S : \phi_m \vdash S : \psi, S : \psi_1, \ldots, S : \psi_n, \Delta}{\Gamma, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m \vdash s : [\alpha]\psi, s : \langle\alpha\rangle\psi_1, \ldots, s : \langle\alpha\rangle\psi_n, \Delta}\ S \textit{ fresh}$$

The derivation of rule MON1, for instance, proceeds by applying $\langle\alpha\rangle_{\mathsf{L}}$ obtaining a new target state S, and then reducing the other formulas using $\langle\alpha\rangle_{\mathsf{R}}$ and $[\alpha]_{\mathsf{L}}$, introducing the resulting existential and universal quantifiers using S as witness. Intuitively the rule expresses an obligation to prove that there are a set of program actions labelled by α in the program state s which are such that there are target states of these actions which satisfy $\psi_1, \ldots, \psi_n$. We are allowed to assume that s has a transition labelled by α which leads to a state satisfying ϕ, and further that any transition from s labelled by α leads to states satisfying the formulas $\phi_1, \ldots, \phi_m$. A sufficient assumption is then that such states also satisfy $\psi_1, \ldots, \psi_n$.

4.4 Inductive and Coinductive Reasoning

To handle recursively defined formulas a mechanism is needed for successfully terminating recursive proof construction. We discuss the ideas on the basis of two informal examples, where the embedding of ERLANG formalised in Section 4.6 is assumed.

Example 6 (Coinduction)**.** Consider the following Erlang function:

```
loop(Out,V) -> Out!V, loop(Out,V).
```

which outputs a constant stream of values `V` along `Out`. One property of the program is that it can always eventually output some value along `Out`:

$$\begin{aligned} &spec : \mathsf{erlangExpression} \to \mathsf{prop} \Rightarrow \\ &\quad \lambda Out : \mathsf{erlangPid}. \\ &\qquad \exists V : \mathsf{erlangValue}.\langle\langle Out!V\rangle\rangle true \\ &\qquad \wedge \quad [\tau] spec\ Out \\ &\qquad \wedge \quad \forall P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V] spec\ Out \end{aligned}$$

where erlangExpression denotes the type of ERLANG expressions and erlangPid denotes the type of ERLANG process identifiers (these notions are formalised in Section 4.6). Further, $\langle\langle\alpha\rangle\rangle\phi$ is the weak modality introduced on Page 27.

The goal sequent takes the shape

$$\vdash \mathtt{loop}(Out, V) : spec\ Out \tag{4.1}$$

where the type of V is assumed to be the ERLANG values (erlangValue).

The first proof step is to unfold the formula definition, and split the conjunction. This results in the three proof goals

$$\vdash \mathtt{loop}(Out, V) : \exists V : \mathsf{erlangValue}.\langle\langle Out!V\rangle\rangle true \tag{4.2}$$

$$\vdash \mathtt{loop}(Out, V) : [\tau] spec\ Out \tag{4.3}$$

$$\vdash \mathtt{loop}(Out, V) : \exists P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V] spec\ Out \tag{4.4}$$

Proving 4.2 is straightforward using the local proof rules and fixed point unfolding. Goal 4.4 is trivially true since no output transition is enabled according to the semantics of ERLANG. Continuing with goal 4.3 one transition is taken, and the formula is again unfolded, resulting in the new goals:

$$\vdash Out\,!\,V,\ \mathtt{loop}(Out, V) : \exists V : \mathsf{erlangValue}.\langle\langle Out!V\rangle\rangle true \tag{4.5}$$

$$\vdash Out\,!\,V,\ \mathtt{loop}(Out, V) : [\tau] spec\ Out \tag{4.6}$$

$$\vdash Out\,!\,V,\ \mathtt{loop}(Out, V) : \exists P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V] spec\ Out \tag{4.7}$$

The first goal is again straightforward. The second goal is trivially true since there are no internal transitions. Continuing with goal 4.7, following one transition step results in the goal

$$\vdash \mathtt{loop}(Out, V) : spec\ Out \tag{4.8}$$

Continuing the proof construction beyond 4.8 is clearly futile: the sequent is identical to the original sequent 4.1. We would like to discharge the goal 4.8 at this point, assuming that this is a sound proof step. For goal 4.8 the soundness of eliminating it is not hard to see: the fixed point *spec* can only appear at its unique position in the sequent 4.8 because it did so in the sequent 4.1. We can say that *spec* is regenerated along the path from 4.1 to 4.8. Moreover, *spec* is a greatest fixed point formula. It turns out that the sequent 4.8 can be discharged for these reasons. In general, however, fixed point unfoldings can be hard to analyse as is seen in examples 8 and 9 in Section 4.5.2.

The basic intuition is that sequents $\Gamma \vdash \Delta$ can be discharged for one of two reasons:

1. Coinductively, because a member of Δ is regenerated through a greatest fixed point formula, as seen in example 6.

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, permitting, for instance, a scheme similar to structural induction on datatypes.

Example 7 (Induction)**.** Consider the following function:

```
last([Hd|Tl],Out) -> Out!Hd, last(Tl,Out);
last(Tl,Out) -> Tl.
```

which returns the tail of the last cons cell of its list parameter, and outputs all other elements of the list along `Out`. The function `last` has the property that it will eventually terminate, in the sense of the following property:

$$\begin{array}{l} \mathit{terminates} : \mathsf{erlangExpression} \to \mathsf{prop} \Leftarrow \\ \quad [\tau]\mathit{terminates} \wedge \forall P : erlangPid, V : \mathsf{erlangValue}.[P!V]\mathit{terminates} \end{array}$$

This is a least fixed point, and must be satisfied without requiring the fixed point to be further unfolded after a finite number of program steps. Hence it can be satisfied only for programs that eventually reach a state in which no internal or output transition is enabled.

The goal sequent is

$$\mathit{properList}\ L \vdash \texttt{last}(L, Out) : \mathit{terminates} \tag{4.9}$$

where the list recognition predicate *properList* is defined as

$$\begin{array}{l} \mathit{properList} \Leftarrow \\ \quad \lambda L : \mathsf{erlangValue}. \\ \quad\quad ((L = \texttt{[]}) \vee (\exists H, T : \mathsf{erlangValue}. L = \texttt{[}H \texttt{|} T\texttt{]} \wedge \mathit{properList}\ T)) \end{array}$$

The least fixed point appearing in the definition of *properList* will be crucial for discharge later in the proof. By unfolding the definition of *properList* (essentially performing a case analysis on the list L using *properList*), unfolding the definition of *terminates* and performing a computation step the result is the two goals:

$$\vdash [\,] : terminates \tag{4.10}$$

$$\textit{properList } T \vdash Out!H, \textit{ last(}T,Out\textit{)} : \textit{terminates} \tag{4.11}$$

The first goal is clearly provable, since the expression `[]` has no transitions. To prove goal 4.11 unfold the definition of *terminates*, resulting in two goals:

$$\textit{properList } T \vdash Out!H, \textit{ last(}T,Out\textit{)} : [\tau]\textit{terminates} \tag{4.12}$$

$$\begin{aligned}&\textit{properList } T \vdash Out!H, \textit{ last(}T,Out\textit{)} :\\ &\quad \exists P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V]\textit{terminates}\end{aligned} \tag{4.13}$$

Goal 4.12 is trivially true (there are no internal transitions). Proceeding with goal 4.13 a computation step is taken resulting in the goal

$$\textit{properList } T \vdash \textit{last(}T,Out\textit{)} : \textit{terminates} \tag{4.14}$$

This sequent is an instance of the initial goal sequent 4.9. Moreover, it was obtained by regenerating the least fixed point formula *properList* on the left. This provides the progress measure required to discharge 4.14.

Finitary data types in general can be specified using least fixed point formulas. This allows for termination or eventuality properties of programs to be proven along the lines of Example 7. In a similar way we can handle program properties that depend on inductive properties of message queues.

4.5 Proof of Recursive Formulas

The approach we use to handle fixed points is essentially 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 conducted. This provides a simple method for dealing with a variety of complications such as alternation of fixed points.

4.5.1 Fixed Point Rules

In general, soundness of fixed point induction relies on the well-known iterative characterisation where least and greatest fixed points are computed as limits of their ordinal approximations.

$$\textsf{APPRX}_\textsf{L}\ \frac{\Gamma, ((\mu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, (\mu U : s_\phi.\phi)\ t_1\ \ldots\ t_n \vdash \Delta}\ \kappa \text{ fresh}$$

$$\textsf{UNF1}_\textsf{L}\ \frac{\Gamma, (\phi\{\mu U : s_\phi.\phi/U\})\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, (\mu U : s_\phi.\phi)\ t_1\ \ldots\ t_n \vdash \Delta}$$

$$\textsf{UNF1}_\textsf{R}\ \frac{\Gamma \vdash (\phi\{\mu U : s_\phi.\phi/U\})\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash (\mu U : s_\phi.\phi)\ t_1\ \ldots\ t_n, \Delta}$$

$$\textsf{UNF2}_\textsf{L}\ \frac{\Gamma, (\phi\{(\mu U : s_\phi.\phi)^{\kappa'}/U\})\ t_1\ \ldots\ t_n, \kappa' < \kappa \vdash \Delta}{\Gamma, ((\mu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n \vdash \Delta}\ \kappa' \text{ fresh}$$

$$\textsf{UNF3}_\textsf{R}\ \frac{\Gamma \vdash \kappa' < \kappa, \Delta \qquad \Gamma \vdash (\phi\{(\mu U : s_\phi.\phi)^{\kappa'}/U\})\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash ((\mu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}$$

Table 4.4: Least Fixed Point Proof Rules

The main rules to reason locally about fixed point formulas are the unfolding rules in Table 4.4. 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. Note that since the greatest fixed point is a derived operator, the proof rules involving this operator are derived and not part of the base set of local proof rules but shown instead in Table 4.5. Let σ range over ν and μ, and κ range over the ordinals. The models considered in the thesis will be countable and thus it suffices to consider countable ordinals.

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 application of the CUT rule. Consider for instance the following application of the TERMCUT rule:

$$\frac{\Gamma \vdash s_2 : (\nu U : s_\phi.\phi)^\kappa \qquad \Gamma, S : (\nu U : s_\phi.\phi)^\kappa \vdash s_1 : (\nu U : s_\phi.\phi)^\kappa}{\Gamma \vdash s_1\{s_2/S\} : (\nu U : s_\phi.\phi)^\kappa}\ \textsf{TERMCUT}$$

In addition to the rules above the identity rules reflecting the monotonicity properties of ordinal approximations are included in the proof system:

$$\textsf{IDMON1}\ \frac{\Gamma \vdash \kappa \leq \kappa', \Delta}{\Gamma, (\mu U : s_\phi.\phi)^\kappa\ t_1\ \ldots\ t_n \vdash (\mu U : s_\phi.\phi)^{\kappa'}\ t_1\ \ldots\ t_n, \Delta}$$

$$\textsf{IDMON2}\ \frac{\Gamma \vdash \kappa' \leq \kappa, \Delta}{\Gamma, (\nu U : s_\phi.\phi)^\kappa\ t_1\ \ldots\ t_n \vdash (\nu U : s_\phi.\phi)^{\kappa'}\ t_1\ \ldots\ t_n, \Delta}$$

$$\textsf{APPRX}_\textsf{R}\ \frac{\Gamma \vdash ((\nu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash (\nu U : s_\phi.\phi)\ t_1\ \ldots\ t_n, \Delta}\ \kappa \text{ fresh}$$

$$\textsf{UNF1}\nu_\textsf{L}\ \frac{\Gamma, (\phi\{\nu U : s_\phi.\phi/U\})\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, (\nu U : s_\phi.\phi)\ t_1\ \ldots\ t_n \vdash \Delta}$$

$$\textsf{UNF1}\nu_\textsf{R}\ \frac{\Gamma \vdash (\phi\{\nu U : s_\phi.\phi/U\})\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash (\nu U : s_\phi.\phi)\ t_1\ \ldots\ t_n, \Delta}$$

$$\textsf{UNF2}_\textsf{R}\ \frac{\Gamma, \kappa' < \kappa \vdash (\phi\{(\nu U : s_\phi.\phi)^{\kappa'}/U\})\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash ((\nu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}\ \kappa' \text{ fresh}$$

$$\textsf{UNF3}_\textsf{L}\ \frac{\Gamma \vdash \kappa' < \kappa, \Delta \qquad \Gamma, (\phi\{(\nu U : s_\phi.\phi)^{\kappa'}/U\})\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, ((\nu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n \vdash \Delta}$$

Table 4.5: Derived Greatest Fixed Point Proof Rules

The rule ORDTRANS expresses transitivity of $<$ over ordinals which is necessary to reason about well-orderings.

$$\textsf{ORDTRANS}\ \frac{\Gamma, \kappa < \kappa', \kappa' < \kappa'', \kappa < \kappa'' \vdash \Delta}{\Gamma, \kappa < \kappa', \kappa' < \kappa'' \vdash \Delta}$$

Theorem 3. The rules $\textsf{APPRX}_\textsf{L}$, $\textsf{UNF1}_\textsf{L}$, $\textsf{UNF1}_\textsf{R}$, $\textsf{UNF2}_\textsf{L}$, IDMON1, IDMON2 and ORDTRANS are sound.

Proof.

$\textsf{APPRX}_\textsf{L}$**:** For $\textsf{APPRX}_\textsf{L}$ assume $\Gamma, ((\mu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n \vdash \Delta$ is valid, and that κ is fresh. Assume further that Γ and $(\mu U : s_\phi.\phi)\ t_1\ \ldots\ t_n$ holds, up to some valuation ρ. Since the semantics is monotone with respect to application, i.e., $\|\phi\ t\|_\rho \triangleq \|\phi\|_\rho\ \|t\|_\rho$, we can focus on the fixed points only. That the unapproximated fixed point is valid means that $\bigcup_\beta \|(\mu U : s_\phi.\phi)^\kappa\|_{\rho[\beta/\kappa]} \neq \emptyset$. So there is some ordinal β such that the semantics of the fixed point approximated with β is non-empty. But then the valuation $\rho[\beta/\kappa]$ that maps κ to β must validate also the premise (since κ is fresh), and thus Δ too. But since κ is fresh then Δ must be valid under ρ too, completing the case.

$\textsf{UNF1}_\textsf{L}$**,**$\textsf{UNFR1}_\textsf{R}$**:** The soundness of these rules follows directly from the fact that $\mu Z.\phi$ is a parametrised fixed point of ϕ.

$\textsf{UNF2}_\textsf{L}$**:** Assume that

$$\Gamma, \phi\{(\nu U : s_\phi.\phi)^{\kappa_1}/U\}\ t_1\ \ldots\ t_n, \kappa_1 < \kappa \vdash \Delta$$

is valid, and that κ_1 is fresh. Assume furthermore that a valuation ρ is given, such that Γ and $(\nu U.\phi)^{\kappa}\ t_1\ \ldots\ t_n$ are valid under it. 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 as α_1, and some assertion in Δ is established as desired. For the third case we find some ${\alpha_1}' < \alpha$ such that $\phi\{(\nu U : s_\phi.\phi)^{{\alpha_1}'}/U\}\ t_1\ \ldots\ t_n$ is true. We can assume that α_1' is a successor ordinal. But then the previous subcase applies, and we are done.

UNF3$_R$**:** Assume that

$$\Gamma \vdash \kappa' < \kappa, \Delta \quad \text{and} \quad \Gamma \vdash (\phi\{(\mu U : s_\phi.\phi)^{\kappa'}/U\})\ t_1\ \ldots\ t_n, \Delta$$

are valid. Assume that Γ is valid under a valuation ρ, we have to show that one of the formulas in $(\mu U : s_\phi.\phi)^{\kappa}, \Delta$ is valid under ρ. So suppose no formula in Δ is valid. Let α be the ordinal that κ is mapped to by ρ. Suppose $\alpha = 0$, this is a contradiction. If $\alpha = \alpha' + 1$ then since $(\phi\{(\mu U : s_\phi.\phi)^{\alpha'}/U\})\ t_1\ \ldots\ t_n$ must be valid under ρ the result follows trivially. If α is a limit ordinal and $\alpha'' < \alpha$ then $(\phi\{(\mu U : s_\phi.\phi)^{\alpha''+1}/U\})\ t_1\ \ldots\ t_n$ is valid so $(\mu U : s_\phi.\phi)^{\alpha''}\ t_1\ \ldots\ t_n$ must be too, and the result follows by monotonicity of fixed points.

IDMON1**,** IDMON2**,** ORDTRANS**:** Trivial given Proposition 1, and ordinal arithmetic.

□

The following text will sometimes refer to the operation of "decreasing an ordinal" κ decorating some fixed point formula $(\sigma X.\phi)^{\kappa}$. By decreasing the ordinal κ we understand the introduction of a fresh ordinal κ' in an assumption $\kappa' < \kappa$, and the replacement of κ with κ' in the fixed point formula, typically by unfolding it yielding the new formula $\phi\{(\sigma X.\phi)^{\kappa'}/X\}$ (as in the rule UNF2$_L$).

4.5.2 Discharge: Some Intuition

The fundamental problem in arriving at a sound, yet powerful, rule of discharge, is to control the way different fixed points may interfere with each other as proofs are conducted. We illustrate the problem by two examples.

Example 8. Consider the proof goal

$$S : U_1 \vdash S : U_3 \tag{4.15}$$

using the abbreviations:

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

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 4.15 is false.

We start by refining 4.15 to the subgoal

$$S : U_2^{\kappa_2} \vdash S : U_4^{\kappa_4} \tag{4.16}$$

using the rules $\mathsf{UNF1_L}$, $\mathsf{UNF1_R}$, $\mathsf{APPRX_L}$ and $\mathsf{APPRX_R}$. 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

$$S : [\tau]U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S : [\tau]U_4^{\kappa_4'} \tag{4.17}$$

$$S : [\tau]U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S : \forall P, V.[P!V]U_3 \tag{4.18}$$

Subgoal 4.17 is refined via rule $\mathsf{MON2}$ to

$$S' : U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S' : U_4^{\kappa_4'} \tag{4.19}$$

and after unfolding U_1 using $\mathsf{UNF1_L}$ we arrive at

$$S' : U_2, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S' : U_4^{\kappa_4'} \tag{4.20}$$

which one might expect to be able to discharge against 4.16 by coinduction in κ_4. Similarly when we refine 4.18 to

$$S : [\tau]U_1, S' : U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \vdash S' : U_4 \tag{4.21}$$

we would expect to be able to discharge against 4.16 inductively in κ_2. This does not work, however, since the derivation of 4.20 from 4.16 fails to preserve the induction variable κ_2 needed for 4.21, and vice versa, κ_4 is not preserved along the path from 4.16 to 4.21. 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 decreased infinitely many times, and preserved when not decreased. 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 9. Consider the (reversed) proof goal

$$S : U_3 \vdash S : U_1 \tag{4.22}$$

with the almost identical abbreviations, except for U_2 and U_4:

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

The goal states 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.

First we apply rules $\mathsf{APPRX_L}$, $\mathsf{APPRX_R}$, $\mathsf{UNF2_L}$ and $\mathsf{UNF2_R}$ to reduce 4.22 to the subgoal

$$S : U_4, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \vdash S : U_2 \tag{4.23}$$

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

$$S' : U_4, S : \forall P, V.[P!V]U_3^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \vdash S' : U_1^{\kappa_3'} \tag{4.24}$$

$$S : [\tau]U_4, S' : U_3^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \vdash S' : U_2 \tag{4.25}$$

These subgoals are refined, using $\mathsf{UNF2_R}$ and $\mathsf{UNF2_L}$ respectively, to

$$\begin{aligned} &S' : U_4, S : \forall P, V.[P!V]U_3^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3, \kappa_3'' < \kappa_3' \\ &\quad \vdash \quad S' : \mu Z_2.[\tau]U_1^{\kappa_3''} \wedge \forall P, V.[P!V]Z_2 \end{aligned} \tag{4.26}$$

$$\begin{aligned} &S : [\tau]U_4, S' : \nu Z_4.[\tau]Z_4 \wedge \forall P, V.[P!V]U_3^{\kappa_1''}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3, \kappa_1'' < \kappa_1' \\ &\quad \vdash \quad S' : U_2 \end{aligned} \tag{4.27}$$

These sequents can be discharged against 4.23 coinductively in κ_3, and inductively 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.

In the following section we shall make the discharge condition formally precise.

4.5.3 The Global Discharge Condition

The *global discharge condition* implements a scheme for fixed point induction via well-founded induction on ordinals. The scheme is a global one, defined as a condition on an otherwise finished proof tree such that all its open goals can be discharged at once, thus completing the proof. The condition requires that each open goal is an instance of an ancestor proof node, and that for each pair of ancestor and open goal some ordinal has decreased along the path from the ancestor to the open proof goal. The condition is complicated by the observation that paths between such pairs of ancestors and proof goals can cross (as evidenced in example 8), and it is therefore necessary to consider the ordinal variables that are preserved along each path.

Definition 46 (Proof Path)**.** *A* path *from a proof node π to a proof node π' in a pre proof (Definition 45) is a sequence of proof nodes $\pi_1, \ldots, \pi_n$ such that $\pi_1 \equiv \pi, \pi_n \equiv \pi'$ or $\pi_1 \equiv \pi', \pi_n \equiv \pi$, and such that for each pair of consecutive nodes π_i, π_{i+1}, $parent(\pi_{i+1}) = \pi_i$ (π_i is the ancestor node of π_{i+1}).*

Let $path(\pi, \pi')$ refer to the unique path from node π to π' (if it exists), and let Π range over the sets of proof nodes.

Definition 47 (Ordinal Variable Preservation and Progress)**.** *Given two proof nodes π_C and π_D such that $sequent(\pi_C) = \Gamma_C \vdash \Delta_C$ and $sequent(\pi_D) = \Gamma_D \vdash \Delta_D$ in a proof tree T, and a substitution ρ,*

- *the ordinal variable $\kappa \in fv(\pi_C)$* progresses *under ρ if $\Gamma_D \vdash \kappa\rho < \kappa$ by application of the proof rules* ORDTRANS *and* ID
- *the ordinal variable $\kappa \in fv(\pi_C)$* is preserved *by ρ if $\Gamma_D \vdash \kappa\rho \leq \kappa$ by application of the proof rules* ORDTRANS *and* ID
- $progressing(\pi_D, \pi_C, \rho) \stackrel{\Delta}{=} \{\kappa : \kappa \in fv(\pi_C) \wedge \kappa$ *progresses under* $\rho\}$
- $preserved(\pi_D, \pi_C, \rho) \stackrel{\Delta}{=} \{\kappa : \kappa \in fv(\pi_C) \wedge \kappa$ *is preserved by* $\rho\}$

Recall that the proof rules that introduce ordinal variables (in Chapter 4 and particularly Section 4.5) require that fresh ordinal variables, from the application of CUT, APPRX_L, APPRX_R, UNF2_L, etc., do not coincide with ordinal variables occurring elsewhere in the proof tree.

Definition 48 (Substitution Instance)**.** *A proof node π_D such that $sequent(\pi_D) = \Gamma_D \vdash \Delta_D$* is a substitution instance *of a proof node π_C such that $sequent(\pi_C) = \Gamma_C \vdash \Delta_C$ under the substitution ρ if*

- *For each $\gamma_i \in \Gamma_C$, $\gamma_i\rho \in \Gamma_D$*
- *For each $\delta_i \in \Delta_C$, $\delta_i\rho \in \Delta_D$*

In other words, the substitution ρ maps each assumption in Γ_C to a corresponding assumption in Γ_D (and vice versa for Δ_C and Δ_D).

Note that the above formalisation is a rather syntactic characterisation of conditions for preservation, progress and substitution instances. An alternative would be to remove the restriction to derivability under ORDTRANS and ID. Then, however, the ordinal variable κ is not guaranteed to occur in the sequent $\Gamma_D \vdash \Delta_D$ (perhaps $\Gamma_D \vdash$ is valid). Another variation is to replace the clause stating inclusion of assumptions

> For each $\gamma_i \in \Gamma_C$, $\gamma_i\rho \in \Gamma_D$

by a statement of derivability

> For each $\gamma_i \in \Gamma_C$, it holds that $\Gamma_D \vdash \gamma_i\rho, \Delta_D$

Lemma 9. *Suppose that π_D is a substitution instance of π_C and the ordinal variable κ progresses (or is preserved) under ρ from π_C to π_D, then κ must occur syntactically in both π_C and π_D and hence also occur in every node between π_C and π_D.*

Proof. Consider for instance preservation, i.e., $\Gamma_D \vdash \kappa\rho \leq \kappa$. The proof that $\kappa \in fv(\pi_D)$ is by a completely trivial induction on the length of this derivation. The final step must be an application of the ID proof rule. But then κ must be present in a formula in Γ_D. Since the two rules ID and ORDTRANS never remove any formula from the left-hand side, and do not introduce any new free variables, the result immediately follows. Since the proof rules by construction are prohibited from re-introducing ordinal variables it follows that κ is present in all the nodes of the path. □

Definition 49 (Proof Structure)**.** *A* proof structure *is a pair consisting of a pre-proof T and a set of pairs $\langle \pi_D, \pi_C \rangle$, one for each open proof goal π_D in T, such that π_C is a proper ancestor node of π_D in T and π_D is a substitution instance of π_C under some substitution. Let such an open proof goal be referred to as a* discharge node*, and its ancestor node as a* companion node*.*

Henceforth, we let T range over the proof structures (as well as the pre-proofs). Further, given a proof structure T and an open proof goal π_D, let $companion(\pi_D)$ refer to its unique companion node π_C.

Definition 50 (Runs over a Proof Structure)**.** *A* run *over a proof structure is a sequence of proof nodes starting in the root node of the proof structure (the pre-proof) such that: (i) for each immediate successor π' of a proof node π of the run, either $parent(\pi') = \pi$, or π is a discharge node and π' is its companion ($companion(\pi) = \pi'$); (ii) any final node in the run was proved through the application of an axiom proof rule (a proof rule without premisses, e.g.,* ID*).*

As a consequence of the definition every finite run has as its last node an axiom rule.

Definition 51 (Discharge Nodes in Runs)**.** *Let $runs(T)$ denote the set of runs over a proof structure T, and let $discharges(T)$ denote the set of discharge nodes in T. Further, let $discharges(r)$ for a run segment $r \in runs(T)$ denote the following set of proof nodes: $\{\pi \mid \pi$ is a discharge node and $\pi \in r\}$.*

The following definitions capture the sequence of proof nodes that can repeat indefinitely in run over a proof structure.

Definition 52 (Repeating Run Segment)**.** *A* repeating run segment *over a proof structure T is a segment $\pi_1, \pi_2, \ldots, \pi_n$ of a run over T such that π_1 is a discharge node, $\pi_1 \equiv \pi_n$, and such that each discharge node occurs at most once in the segment $\pi_2, \ldots, \pi_n$. Denote with $loops(T)$ the set of repeating run segments over a proof structure T.*

Clearly, since there are a finite number of discharge nodes in T the set $loops(T)$ must be finite too.

Definition 53 ($infpaths$)**.** *Define the set of loop paths over a proof structure T, denoted with $infpaths(T)$, as the set*

$$\{discharges(s) \mid s \in loops(T)\}$$

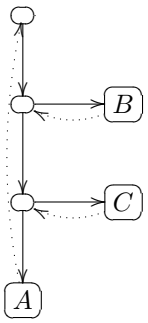

Figure 4.2: A Symbolic Pre-Proof Tree

Lemma 10. *For each element* $\Pi \in \mathit{infpaths}(T)$ *there is a run* r *over* T *such that the set of discharge nodes that occur infinitely often in* r *is* Π*. Vice versa, for any run* r *over* T *the set* Π *of discharge nodes that occur infinitely often in* r *is a member of* $\mathit{infpaths}(T)$*.*

Proof. Obvious from the definition of $\mathit{infpaths}(T)$. □

The set $\mathit{infpaths}(T)$ is finite for any proof structure T, and can easily be computed from the proof structure.

Consider the symbolic proof structure T depicted in Figure 4.2. Discharge nodes are labelled by A, B and C respectively and the paths to their respective companion nodes are indicated with dotted arrows. Now

$$\mathit{infpaths}(T) = \{\{A, B, C\}, \{A, C\}, \{A, B\}, \{A\}, \{B\}, \{C\}\}$$

Note that the set $\{B, C\}$ is missing since the paths from companion nodes to discharge nodes B and C do not intersect.

Next, for a proof structure T and a subset of its discharge nodes $\langle \pi_{D_1}, \ldots, \pi_{D_n} \rangle$ and substitutions $\langle \rho_1, \ldots, \rho_n \rangle$ such that each π_{D_i} is a substitution instance of $\mathit{companion}(\pi_{D_i})$ under ρ_i, the sets of preserved ordinal variables and progressing ones are defined:

Definition 54 (Progress and Preservation for Discharge Sets)**.** *The two functions* $\mathit{preserved}$ *and* $\mathit{progressing}$ *are defined below:*

$$\begin{aligned}
&\mathit{preserved}(\langle \pi_{D_1}, \ldots, \pi_{D_n} \rangle, \langle \rho_1, \ldots, \rho_n \rangle) \triangleq \\
&\quad \textstyle\bigcap_{1 \le i \le n} \mathit{preserved}(\pi_{D_i}, \mathit{companion}(\pi_{D_i}), \rho_i) \\
&\mathit{progressing}(\langle \pi_{D_1}, \ldots, \pi_{D_n} \rangle, \langle \rho_1, \ldots, \rho_n \rangle) \triangleq \\
&\quad \textstyle\bigcup_{1 \le i \le n} \mathit{progressing}(\pi_{D_i}, \mathit{companion}(\pi_{D_i}), \rho_i) \cap \\
&\quad \mathit{preserved}(\langle \pi_{D_1}, \ldots, \pi_{D_n} \rangle, \langle \rho_1, \ldots, \rho_n \rangle)
\end{aligned}$$

In other words, $\kappa \in \mathit{preserved}(\langle \pi_{D_1}, \ldots, \pi_{D_n} \rangle, \langle \rho_1, \ldots, \rho_n \rangle)$ if κ is preserved for each discharge node, companion node, and substitution triple. An ordinal κ progresses, i.e., $\kappa \in \mathit{progressing}(\langle \pi_{D_1}, \ldots, \pi_{D_n} \rangle, \langle \rho_1, \ldots, \rho_n \rangle)$ if there is at least one discharge node, companion node and substitution triple where it progresses, and it is preserved everywhere.

Finally we are ready to state the global discharge condition that checks whether all open proof goals can be discharged:

Definition 55 (Global Discharge Condition)**.** *Consider a proof structure T with a possibly empty set of discharges $\{\pi_{D_1}, \ldots, \pi_{D_n}\}$. The discharge condition on T is true, if, for each element $\{\pi_1, \ldots, \pi_j\} \in \mathit{infpaths}(T)$, there exists a corresponding set of substitutions $\{\rho_1, \ldots, \rho_j\}$ such that*

- *each π_i is a substitution instance of $\mathit{companion}(\pi_i)$ under ρ_i,*
- *$\mathit{progressing}(\langle \pi_1, \ldots, \pi_j \rangle, \langle \rho_1, \ldots, \rho_j \rangle) \neq \emptyset$*

Next the notion of a proof is finalised.

Definition 56 (Proof)**.** *A* proof *is a proof structure on which the global discharge condition holds.*

Before the whole proof system can be proved sound, a lemma on the validation properties of proof rules is required.

Lemma 11. *Consider any substitution instance of a proof rule with premisses*

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

Suppose a substitution ρ invalidates $\Gamma \vdash \Delta$, i.e., each $\gamma_i \in \Gamma$ is valid under ρ while no $\delta_i \in \Delta$ is valid. Then there exists a sequent $\Gamma_j \vdash \Delta_j$ among the premisses, and a substitution ρ' which coincides with ρ on any variable that occurs in the conclusion and which invalidates $\Gamma_j \vdash \Delta_j$.

Proof. We consider the proof rules.

$\mathsf{W_L}$,$\mathsf{W_R}$**:** Trivial.

CUT**:** Suppose there is no extension (a substitution that coincides with ρ on variables also in the conclusion) ρ' of ρ such that ϕ is valid, then choose any such extension ρ''. Trivially $\Gamma \vdash \phi, \Delta$ is not valid under ρ''. Suppose instead there is an extension ρ'' validating ϕ, then clearly no formula in Δ is validated by ρ'', since ρ'' is an extension of ρ.

$\neg_\mathsf{L}$,$\neg_\mathsf{R}$,$\vee_\mathsf{L}$,$\vee_\mathsf{R}$**:** Trivial.

$\exists_\mathsf{L}$**:** That ρ validates $\exists X : S.\phi$, means that due to its semantics $\bigcup_{v \in S} \left(\| \phi \|_{\rho[v/X]} \right)$, there must exist a term $v \in S$ such that $\phi\{v/X\}$ is valid under ρ. Let ϕ' be $\phi[v/X']$, clearly ϕ' validates $\phi\{X'/X\}$ whereas since X' is fresh, Δ is not validated by ϕ'.

$\exists_R$: That ρ does not validate $\exists X : S.\phi$ means, according to the semantics of the quantification construct, that there can be no term $t \in S$ such that $\phi\{t'/X\}$ is validated by ρ.

APPLY$_L$,APPLY$_R$: A trivial conclusion from the semantics of the application construct.

SYMM$_L$, SYMM$_R$, TRANS$_L$, TRANS A trivial conclusion from the semantical properties of equality.

SUBST It suffices to consider the effect of substitution on an another equality, i.e., on the right hand side suppose that $t_3\{t_1/X\} = t_4\{t_1/X\}$ is invalid under ρ, and $t_1 = t_2$ is valid under ρ but $t_3\{t_2/X\} = t_4\{t_2/X\}$ is valid under ρ. For this explicit equality the observation that equality is a congruence suffices to establish that the latter equality cannot be valid under ρ.

CEQ$_L$, CEQ$_R$ Trivial.

APPRXL$_L$: Any ρ' can be chosen which corresponds to ρ except for the fresh variable κ, and which is such that $((\mu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n$ is valid, clearly such an ordinal must exists.

UNF1$_L$,UNF3$_L$: The original substitution ρ can be used to invalidate the single new premise, due to the semantics of the least fixed point constructor.

UNF2$_L$: Choose as ρ' any valuation that coincides with ρ and which assigns the fresh ordinal variable κ' an ordinal value less than $\kappa\rho$ and which is such that $(\phi\{(\mu U : s_\phi.\phi)^{\kappa'}/U\})\ t_1\ \ldots\ t_n$ is valid. Such a value must exist, since otherwise due to the semantics of the least fixed point construct, $((\mu U : s_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n$ could not be valid.

IDMON1: Choose ρ as the new valuation; the result follows trivially, i.e., $\kappa < \kappa'$ must be invalid since otherwise $(\mu U : s_\phi.\phi)^{\kappa'}\ t_1\ \ldots\ t_n, \Delta$ must be valid (the denotation of the left hand side fixed point would be included in its denotation).

ORDTRANS: A trivial consequence of ordinal arithmetic.

□

The following lemma constructs what is to be used as a "rejection path" in the proof of soundness. The idea is to equip a run, starting in an invalid sequent, with a valuation that almost everywhere preserves the values of ordinal variables, except possibly at discharge nodes. However, as is established in the soundness proof itself, in a sufficiently small fragment of the run the coupled valuation will ensure that at least one ordinal variable will be infinitely often decreased, and never increased.

Lemma 12 (Existence of a Rejection Path)**.** *Suppose that a proof structure T is rooted in a sequent $\Gamma \vdash \Delta$, and has a non-empty set of discharge nodes. Further assume that each each discharge node $\pi_{D_1}, \ldots, \pi_{D_n}$ is a substitution instance of its companion node $companion(\pi_{D_1}), \ldots, companion(\pi_{D_n})$ under a given set of substitutions*

$\rho_1, \ldots, \rho_n$. Suppose further that the sequent $\Gamma \vdash \Delta$ is invalid, i.e., there exists a valuation ρ that validates all the assumptions in Γ but does not validate any formula in Δ. Then there exists an infinite sequence of pairs $\langle \pi, \rho \rangle$ of a proof node π and a valuation ρ such that each sequent labelling a proof node is invalidated by the coupled valuation, and such that the proof nodes of the sequence forms a run over the proof structure.

Proof. The proof will be by construction. We will trace a path through the proof structure finding pairs $\langle \pi, \rho \rangle$ of a proof node and a substitution that invalidates the sequent labelling the paired node, starting with the pair $\langle \pi_0, \rho_0 \rangle$ chosen such that $\Gamma \vdash \Delta$ labels π_0 and ρ_0 is any substitution that invalidates the sequent. Given a pair $\langle \pi_i, \rho_i \rangle$ such that $\Gamma_i \vdash \Delta_i$ labels π_i, the next pair is chosen according to the following scheme. Suppose that the proof rule that labels π is an axiom then from the soundness of the proof rules immediately a contradiction results (ρ_i cannot invalidate the sequent). Consider instead an application of any other proof rule yielding a number of premises $\Gamma_{i_1} \vdash \Delta_{i_1}, \ldots, \Gamma_{i_n} \vdash \Delta_{i_n}$. Since ρ_i invalidates $\Gamma_i \vdash \Delta_i$ then again by soundness some $\Gamma_{i_j} \vdash \Delta_{i_j}$ must be invalidated by some ρ_{i_j} also. Choose a substitution ρ_{i_j} which conforms to Lemma 11, i.e., all ordinal variables (and other variables too) preserve their values. Let the next pair be $\langle \pi_{i_j}, \rho_{i_j} \rangle$, where π_{i_j} labels $\Gamma_{i_j} \vdash \Delta_{i_j}$. Suppose instead the proof node π_i labelled by $\Gamma_i \vdash \Delta_i$ and is a discharge node. Then the proof structure contains a companion node $companion(\pi_i)$ and the lemma provides a substitution ρ_c such that the discharge node is a substitution instance of the companion node under ρ_c. Let the next pair be $\langle companion(\pi_i), \rho_c \rho_i \rangle$. Clearly $\rho_c \rho_i$ invalidates $companion(\pi_i) \equiv \Gamma_c \vdash \Delta_c$ since for every $\gamma_i \in \Gamma_c$ it holds that $\gamma_i \rho_c \in \Gamma_i$ and must thus be validated by ρ_i; similarly for every $\delta_i \in \Delta_c$ it holds $\delta_i \rho_c \in \Delta_i$ which must thus be invalidated by ρ_i.

□

Theorem 4 (Soundness of the Proof System)**.** Any sequent $\Gamma \vdash \Delta$, for which there exists a proof that is rooted in the sequent is valid.

Proof. The existence of a proof of $\Gamma \vdash \Delta$ means that there is a proof structure T labelled by $\Gamma \vdash \Delta$ on which the global discharge condition holds.

Assume that a sequent $\Gamma \vdash \Delta$ is invalid, then by Lemma 12 there exists an infinite sequence of pairs of a proof node (forming a proof run) and a valuation that invalidates its paired proof node. Given the constructed "invalidation" run we will extract an ordinal that decreases infinitely often which is impossible, thus contradicting the assumption that the initial node π in the path was not validated by ρ.

First choose a infinite suffix of the sequence such that any discharge node that occurs along it occurs infinitely often, and such that the first node of the segment is either a discharge or a companion node. This is trivial, since if a discharge node π occurs only finitely often in the run we can simply choose a suffix of the run which begins after the last occurrence of π. Let $\pi_{D_1}, \ldots, \pi_{D_n}$ be the discharge nodes that occur infinitely often in the run suffix, which according to Lemma 10 is a member of $infpaths(T)$.

It follows, due to the definition of the global discharge condition, that there exists substitutions $\rho_1, \ldots, \rho_n$ such that each discharge node π_{D_i} is an substitution instance of

its companion node $companion(\pi_{D_i})$ under the substitution ρ_i, and most importantly there exists an ordinal variable κ such that $\kappa \in progressing(\langle \pi_1, \ldots, \pi_j \rangle, \langle \rho_1, \ldots, \rho_j \rangle)$.

From the definition of the $progressing$ (and $preserved$) condition (see Definition 47 and Definition 54) it is clear that κ must occur syntactically in every discharge node among $\{\pi_1, \ldots, \pi_n\} = discharges(r) \in infpaths(T)$, and the associated companion nodes. Lemma 9 further establishes that κ must occur in every node in the run segment r_s, since the segment is characterised by its discharge nodes.

We will now establish that the coupled valuation of the run segment forces the ordinal variable κ to be infinitely often decreased, and never increased.

Consider first the application of a proof rule. Clearly the variable κ is present both in the conclusion and all the premisses of the rule. In the construction of the rejection path (Lemma 12) the constructed path explicitly preserved the values of all variables from one valuation to the following, and thus also the value of κ.

Suppose instead that the current node on the path is a discharge node π_{D_i} with a valuation ρ', i.e., $\langle \pi_{D_i}, \rho' \rangle$. The following node on the constructed path was $\langle companion(\pi_{D_i}), \rho_i \rho' \rangle$. Suppose further that the discharge node π_{D_i} is such that the ordinal variable κ progresses, i.e., $\kappa \in progress(\pi_{D_i}, companion(\pi_{D_i}, \rho_i))$.

There must be at least one such progress occurrence in a discharge node due to the global discharge condition. That κ progresses means, per Definition 47, that $\Gamma_{D_i} \vdash \kappa\rho_i < \kappa$, is derivable using (sound) proof rules. Since ρ' validates Γ_{D_i} this means that $\kappa\rho_i\rho' < \kappa\rho'$. Hence the value of κ, under the valuation $\rho_i\rho'$ in the following path element, must be strictly less than the value of κ under ρ'. Since the discharge node π_{D_i} is encountered infinitely often, this means that the ordinal variable κ decreases infinitely often under its coupled valuation.

Suppose instead that κ does not progress but rather is preserved at the discharge node π_{D_i}. Note that κ must trivially either be preserved or progress due to the assumption that it progresses in the sense of the discharge condition for the set of discharge nodes $\pi_{D_1}, \ldots, \pi_{D_n}$. Preservation means, per Definition 47, that $\Gamma_{D_i} \vdash \kappa\rho_i \leq \kappa$, is derivable using (sound) proof rules. Since ρ' validates Γ_{D_i} this means that $\kappa\rho_i\rho' \leq \kappa\rho'$. Hence the value of κ, under the valuation $\rho_i\rho'$ in the following path element, must be equal or less than the value of κ under ρ'. That is, the valuation of the ordinal variable κ never increases.

Since there is no chain of infinitely decreasing ordinals we have derived a contradiction, and thus the original assumption that ρ invalidates $\Gamma \vdash \Delta$ must have been false. □

For clarity we have chosen to present the global discharge condition as a final check on an otherwise finished proof in this section. However, in the implementation (see Chapter 5 for details) the checking of the condition is separated into a local part (find a candidate instance for a potential discharge node, and compute its candidate substitutions) which is applicable even when non-dischargeable proof goals remain, and a global part (check whether local discharges interfere) which can be applied iteratively to an increasing number of discharge node and companion node pairs. It should be noted that compared to the existing implementation this formalisation explicitly strengthens the requirement on choosing new ordinal variables, and relaxes the

requirement that among every pair of "related discharges" some ordinal must progress.

4.5.4 Fixed Point Induction via Local Proof Rules

This section introduces an explicit, local, fixed point induction scheme in the proof system and motivates its soundness.

The chief reason for adding the global discharge condition to the proof system, rather than a local proof rule equivalent, is to permit the lazy discovery of fixed point induction schemes. That is, using the global condition there is at least conceptually no need to commit to a particular induction scheme before all non-induction cases have been handled.

However, there is clearly also merit in giving an account of fixed point reasoning using local proof rules. One consideration is ease of embedding into standard proof assistants. The global discharge condition necessitates checking a condition global to a proof tree which is difficult to express in most proof assistants, if the notion of a proof tree exists at all. Furthermore the soundness of local induction schemes can be more readily established. For these reasons, and to properly motivate the global discharge condition we here give examples of alternative local proof rules for fixed point induction with similar expressive power. We show that the application of these rules, in concrete instances, can be mimicked using the global discharge condition. In the other direction we would like to demonstrate that any proof, using the global discharge condition, can be coded using local proof rules, thus providing an alternative proof of the soundness of the whole proof system. However, this part is left for future work.

Some proviso: only closed fixed point formulas $(\sigma X.\phi)\widetilde{V}$ are considered, i.e., such that $fv(\phi) = \emptyset$. Note that formulas not in this shape can easily be put in this form by abstracting free variables and extending the argument vector $\widetilde{V}$. Further note that in this section an extension of the logic is assumed which permits quantification over ordinals. The construct $\forall\kappa'.\kappa' < \kappa \Rightarrow (\nu X.\phi)^{\kappa'}$, with the valuation ρ, has the obvious semantics $\bigcap_\beta \left\{ \| (\nu U : s_\phi.\phi)^\kappa \|_{\rho[\beta/\kappa]} \mid \beta < \rho[\kappa] \right\}$. In the following rules are provided both for unfolding greatest and least fixed points, where as usual, the greatest fixed point rules are derived.

Fixed Point Induction Schemes

$$\mathsf{FP_R}\frac{\Gamma, \forall\kappa'.\kappa' < \kappa \Rightarrow (\nu X.\phi)^{\kappa'}\widetilde{V} \vdash (\nu X.\phi)^{\kappa}\widetilde{V}, \Delta}{\Gamma \vdash (\nu X.\phi)\widetilde{V}, \Delta}\ \kappa \text{ fresh}$$

$$\mathsf{FP_L}\frac{\Gamma, (\mu X.\phi)^{\kappa}\widetilde{V}, \forall\kappa'.\kappa' < \kappa \Rightarrow \neg(\mu X.\phi)^{\kappa'}\widetilde{V} \vdash \Delta}{\Gamma, (\mu X.\phi)\widetilde{V} \vdash \Delta}\ \kappa \text{ fresh}$$

Theorem 5. The rules $\mathsf{FP_R}$ and $\mathsf{FP_L}$ are sound.

Proof. Consider $\mathsf{FP_R}$ as an example. The proof will be by contradiction, i.e., assume that

$$\forall \kappa'.\kappa' < \kappa \Rightarrow (\nu X.\phi)^{\kappa'}\widetilde{V} \vdash (\nu X.\phi)^{\kappa}\widetilde{V}, \Delta$$

is validated by all substitutions, and that there exists some substitution ρ which invalidates $\Gamma \vdash (\nu X.\phi)\widetilde{V}, \Delta$. We will construct a substitution ρ' which invalidates also the premise. Since $(\nu X.\phi)\widetilde{V}$ is not valid there exists an ordinal $\kappa\rho$ such that $(\nu X.\phi)^{\kappa\rho}(\widetilde{V}\rho)$ is not valid. Further assume that there exists no $\kappa' < \kappa\rho$ for which $(\nu X.\phi)^{\kappa'}(\widetilde{V}\rho)$ is not valid. If there exists such an ordinal, less than $\kappa\rho$, than invalidates the formula, select it and repeat the search procedure. Since $(\nu X.\phi)^{0}(\widetilde{V}\rho)$ is trivially valid this procedure will terminate successfully with an ordinal κ''. Now we will show that $\rho[\kappa''/\kappa]$ invalidates also the premise. Trivially all formulas in Γ are valid (since κ is fresh and the conclusion not valid) and similarly all formulas in Δ are not valid. From the construction of κ'' follows that $\forall \kappa'.\kappa' < \kappa\rho[\kappa''/\kappa] \Rightarrow (\nu X.\phi)^{\kappa'}\widetilde{V}\rho[\kappa''/\kappa]$ is valid also. But by construction $(\nu X.\phi)^{\kappa\rho[\kappa''/\kappa]}\widetilde{V}\rho[\kappa''/\kappa]$ is invalid, and thus the premise is invalid. □

Example 10. Consider the function definition `loop() -> loop().`, and the goal

$$\vdash \texttt{loop()} : \nu X : \mathsf{erlangExpression} \rightarrow \mathsf{prop}.(\langle\tau\rangle \mathit{true} \wedge [\tau]X)$$

This sequent can be proved using $\mathsf{FP_R}$ without instances of the DISCHARGE rule.

Unfortunately the rules $\mathsf{FP_R}$ and $\mathsf{FP_L}$ are too weak to motivate the discharge condition, in particular the values in the argument vector $\widetilde{V}$ need to be relativised with respect to Γ and Δ. A more practical set of fixed point induction rules are, assuming that $\widetilde{X}$ are the free variables in Γ, Δ, $\widetilde{V}$, and omitting κ which is assumed fresh, $(\widetilde{X} = \left(fv(\Gamma) \cup fv(\Delta) \cup fv(\widetilde{V})\right) \setminus \{\kappa\})$:

$$\mathsf{FPR_R}\frac{\Gamma, \forall \widetilde{X}.\left(\bigwedge \Gamma \Rightarrow \bigvee \Delta \vee \left(\forall \kappa'.\kappa' < \kappa \Rightarrow (\nu X.\phi)^{\kappa'}\widetilde{V}\right)\right) \vdash (\nu X.\phi)^{\kappa}\widetilde{V}, \Delta}{\Gamma \vdash (\nu X.\phi)\widetilde{V}, \Delta}$$

$$\mathsf{FPR_L}\frac{\Gamma, \forall \widetilde{X}.\left(\bigwedge \Gamma \Rightarrow \bigvee \Delta \vee \left(\forall \kappa'.\kappa' < \kappa \Rightarrow \neg(\mu X.\phi)^{\kappa'}\widetilde{V}\right)\right), (\mu X.\phi)^{\kappa}\widetilde{V} \vdash \Delta}{\Gamma, (\mu X.\phi)\widetilde{V} \vdash \Delta}$$

Theorem 6. The rules $\mathsf{FPR_R}$ and $\mathsf{FPR_L}$ are sound.

Proof. We consider the soundness proof of $\mathsf{FPR_R}$. The proof will be by contradiction. We assume that the premise is valid (is validated by all substitutions), that there is a substitution ρ that invalidates the conclusion, and from this fact construct a substitution ρ' that also invalidates the premise, thus arriving at a contradiction. That ρ invalidates the conclusion implies that it invalidates $(\nu X.\phi)\widetilde{V}$, i.e., there is a $\kappa\rho$ such that $(\nu X.\phi)^{\kappa\rho}\widetilde{V}\rho$ is not valid. Consider all the ordinals κ' less than $\kappa\rho$, and all substitutions ρ'' mapping free variables in Γ, Δ, $\widetilde{V}$ to values. If there is a substitution

ρ'' and an ordinal $\kappa' < \kappa\rho$ such that $\bigwedge \Gamma\rho'' \Rightarrow \bigvee \Delta\rho'' \vee (\nu X.\phi)^{\kappa'}\widetilde{V}\rho''$ is not valid, select that ordinal and repeat the search procedure, remembering the candidate "failure" ordinal κ' and substitution ρ''. Since trivially, for any valuation ρ''', the above condition is satisfied by $(\nu X.\phi)^0\widetilde{V}\rho'''$ eventually the search will terminate with a suitable κ' and a substitution ρ'' such that $(\nu X.\phi)^{\kappa'}\widetilde{V}\rho''$ is not valid. Consider now the substitution $\rho''[\kappa'/\kappa]$, we will show that it invalidates the premise. From the construction of κ',ρ'' follows that Γ is validated by ρ'' and Δ is not. But again from the construction of κ',ρ'' follows immediately that for all substitutions ρ'', the assumption $\bigwedge \Gamma \Rightarrow \bigvee \Delta \vee \left(\forall \kappa'.\kappa' < \kappa \Rightarrow (\nu X.\phi)^{\kappa'}\widetilde{V}\right)$ is valid, and thus the whole premise is not valid under $\rho''[\kappa'/\kappa]$ and so we have arrived at a contradiction. □

Example 11. Consider a typical natural number induction scheme (N is assumed to be a variable ranging over the natural numbers, and for simplicity it is assumed that ϕ contains no free variables):

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \vdash \phi\ 0 & \phi\ N \vdash \phi\ N+1\end{array}}{\vdash \phi\ N}$$

Such a scheme can easily be mimicked using the $\mathsf{FPR_L}$ proof rule. A recursive characterisation of the natural numbers is assumed:

$$isNat \triangleq \mu X.\lambda N : \mathsf{nat}.N = 0 \vee \exists N' : \mathsf{nat}.N = N' + 1 \wedge isNat\ N'$$

The above proof scheme can then be derived as follows:

$$\frac{\begin{array}{cc}\vdots & \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{-}{\kappa' < \kappa \vdash \kappa' < \kappa} \quad \dfrac{-}{isNat^{\kappa'}\ N' \vdash isNat^{\kappa'}\ N'}}{\kappa' < \kappa \Rightarrow \neg isNat^{\kappa'}\ N', \kappa' < \kappa, isNat^{\kappa'}\ N' \vdash \phi\ N'+1}}{\forall N.\left(\phi\ N \vee \left(\forall \kappa'.\kappa' < \kappa \Rightarrow \neg isNat^{\kappa'}\ N\right)\right), \kappa' < \kappa, isNat^{\kappa'}\ N' \vdash \phi\ N'+1}}{\forall N.\left(\phi\ N \vee \left(\forall \kappa'.\kappa' < \kappa \Rightarrow \neg isNat^{\kappa'}\ N\right)\right), isNat^{\kappa}\ N \vdash \phi\ N}}{isNat\ N \vdash \phi\ N}$$

Example 12. The following example illustrates a typical proof using the rule of induction, and a corresponding proof using the explicit fixed point induction proof rules. A recursive characterisations of the natural numbers and the even natural numbers is assumed:

$$isEven \triangleq \mu X.\lambda N : \mathsf{nat}.N = 0 \vee \exists N' : \mathsf{nat}.N = N' + 2 \wedge isEven\ N'$$

The following (abbreviated) derivation uses the discharge condition to eliminate

remaining goals:

$$\dfrac{\dfrac{\vdots}{\vdash isNat\ 0} \quad \dfrac{\dfrac{\dfrac{\dfrac{-}{isEven^{\kappa'}\ N', \kappa' < \kappa \vdash isNat\ N'}\ \textsf{DISCHARGE}}{isEven^{\kappa'}\ N', \kappa' < \kappa \vdash isNat\ N' + 1}}{isEven^{\kappa'}\ N', \kappa' < \kappa \vdash isNat\ N' + 2}}{\dfrac{isEven^{\kappa}\ N \vdash isNat\ N}{isEven\ N \vdash isNat\ N}\ \textsf{APPRX}_{\textsf{R}}}$$

Using fixed point induction an alternative derivation results:

$$\dfrac{\dfrac{\vdots}{\ldots \vdash isNat\ 0} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{-}{\ldots, isNat\ N' \vdash isNat\ N'} \quad \dfrac{-}{\ldots, isEven^{\kappa'}\ N' \vdash isEven^{\kappa'}\ N', \ldots}}{\kappa' < \kappa, isEven^{\kappa'}\ N', isNat\ N' \vee \forall \kappa'.\kappa' < \kappa \Rightarrow \neg isEven^{\kappa'}\ N' \vdash isNat\ N'}}{\kappa' < \kappa, isEven^{\kappa'}\ N', \forall N. \ldots \vdash isNat\ N'}}{\kappa' < \kappa, isEven^{\kappa'}\ N', \forall N. \ldots \vdash isNat\ N' + 1}}{\kappa' < \kappa, isEven^{\kappa'}\ N', \forall N. \ldots \vdash isNat\ N' + 2}}{\dfrac{isEven^{\kappa}\ N, \forall N. isNat\ N \vee \forall \kappa'.\kappa' < \kappa \Rightarrow \neg isEven^{\kappa'}\ N \vdash isNat\ N}{isEven\ N \vdash isNat\ N}\ \textsf{FPR}_{\textsf{L}}}$$

4.6 Embedding ERLANG into the Proof System

This section sketches the embedding of the ERLANG dialect considered in this thesis into the underlying many-sorted first-order logic. Summarised briefly, the approach is to encode ERLANG language constructs as terms of the underlying logic, the transition relations as fixed point definitions, and to derive convenient proof rules from the transition relation.

The approach to the embedding of ERLANG, and the ERLANG semantics in the proof system, is similar to earlier treatments of CCS [Nes93] and the π-calculus [Mel94] in HOL [Ge93] although this thesis addresses a more complex language. Other more recent examples include the embedding of Java source and bytecode in various theorem provers such as Isabelle and COQ [vdBHJP00, Hui00, NvOP00, BDJ$^+$01].

4.6.1 Embedding Expressions and Values

In Figure 4.3 a data type corresponding to the ERLANG expression is given. The recursive definition presupposes the int, nat and atom sorts. The embedding is a deep one, in the respect that the ERLANG variables are represented as constant terms, constructed by the *erl_expr_var* constructor rather than represented as variables of the proof system. The deep embedding naturally requires us to define the notion of ERLANG substitution

$$
\begin{array}{ll}
& isErlangValue \Leftarrow \\
& \lambda E : \mathsf{erlangExpression}. \\
& \quad \exists I : \mathsf{int}. E = \textit{erl\_expr\_int}(I) \\
\vee & \quad \exists N : \mathsf{nat}. E = \textit{erl\_expr\_pid}(N) \\
\vee & \quad \exists A : \mathsf{atom}. E = \textit{erl\_expr\_atom}(A) \\
\vee & \quad E = \textit{erl\_expr\_nil} \\
\vee & \quad \exists E_1, E_2 : \mathsf{erlangExpression}. \\
& \qquad E = \textit{erl\_expr\_cons}(E_1, E_2) \wedge isErlangValue\ E_1 \wedge isErlangValue\ E_2 \\
\vee & \quad \exists E_l : \mathsf{erlangExpression}\ list. \\
& \qquad E = \textit{erl\_expr\_tuple}(E_l) \wedge isErlangValues\ E_l \\
& isErlangValues \Leftarrow \\
& \lambda E_l : \mathsf{erlangExpression}\ list. \\
& \quad E_l = [\,] \\
\vee & \quad \exists E : \mathsf{erlangValue}, {E_l}' : \mathsf{erlangValue}\ list. \\
& \qquad E_l = [E|{E_l}'] \wedge isErlangValue\ E \wedge isErlangValues\ {E_l}'
\end{array}
$$

Table 4.6: Predicates for Determining Membership among ERLANG Values

independent of the notion of substitution in the proof system. Further note that process identifiers are represented as natural numbers.

An ERLANG match erlangMatch is a list of clauses erlangClause $list$. An ERLANG clause is defined by

$$
\begin{array}{l}
\texttt{type}\ \mathsf{erlangClause} \triangleq \\
\quad \textit{erl\_clause}\ \texttt{of}\ \mathsf{erlangPattern}, \mathsf{erlangExpression}, \mathsf{erlangGuard}
\end{array}
$$

such that erlangPattern is a subtype of erlangExpression which omits all non-data constructor expressions, e.g., the send and `case` expressions. The erlangGuard data type is a sequence of ERLANG expressions erlangExpression, where the permissible functions are enumerated.

The ERLANG values is a subtype of the patterns, which omits the ERLANG variables in addition to all non-data constructors. The ERLANG values $\mathsf{erlangValue} \triangleq \{t : \mathsf{erlangPattern} \mid isErlangValue\ t\}$ are characterised by the two fixed point definitions in Table 4.6.

A number of further subtypes, the ERLANG atoms, integers, process identifiers and booleans (the atoms `true` and `false`) are provided with obvious definitions and type names as erlangAtom, erlangInt, erlangPid and erlangBool as subtypes of the values through recogniser predicates with names *isErlangAtom*, *isErlangInt*, etc., definitions omitted. Trivially the subtype erlangValue defined by the $isErlangValue$ predicate is

non-empty (consider for instance the empty list). Similarly all the other data predicates define non-empty subtypes of the ERLANG expressions. Finally the subtype of proper lists (values which are either the empty list `[]` or a cons cell such that the tail is a proper list) is recognised by the predicate

$$\begin{array}{l} \mathit{properList} \Leftarrow \\ \quad \lambda L : \mathsf{erlangValue}.L = \texttt{[]} \vee \exists H, T : \mathsf{erlangValue}.L = \texttt{[}H \texttt{|} T\texttt{]} \wedge \mathit{properList}\ T \end{array}$$

Expression Actions To embed the ERLANG expression actions a type erlangExprAction is defined:

$$\begin{array}{l} \texttt{type}\ \mathsf{erlangExprAction} = \\ \quad \mathit{erl\_eact\_tau} \\ \quad | \quad \mathit{erl\_eact\_out}\ \texttt{of}\ \mathsf{erlangValue}, \mathsf{erlangValue} \\ \quad | \quad \mathit{erl\_eact\_exiting}\ \texttt{of}\ \mathsf{erlangValue} \\ \quad | \quad \mathit{erl\_eact\_read}\ \texttt{of}\ \mathsf{erlangQueue}, \mathsf{erlangValue} \\ \quad | \quad \mathit{erl\_eact\_testq}\ \texttt{of}\ \mathsf{erlangQueue} \\ \quad | \quad \mathit{erl\_eact\_predef}\ \texttt{of}\ \mathsf{erlangValue}, \mathsf{erlangValue}\ list, \mathsf{erlangValue} \end{array}$$

ERLANG Queues The ERLANG Queues are sequences of values, built using the empty sequence constructor, the value constructor, and sequencing:

$$\begin{array}{l} \texttt{type}\ \mathsf{erlangQueue} = \\ \quad \mathit{erl\_queue\_epsilon} \\ \quad | \quad \mathit{erl\_queue\_val}\ \texttt{of}\ \mathsf{erlangValue} \\ \quad | \quad \mathit{erl\_queue\_append}\ \texttt{of}\ \mathsf{erlangQueue}, \mathsf{erlangQueue} \end{array}$$

The ERLANG queues are not freely generated and as a result a number of additional proof rules in Table 4.7 are provided to reason about queue equality. In these rules the underlying term representation is not used to display queues, rather they are shown using the intuitive constructors such as concatenation and the empty sequence ϵ. To make the rules clearer we indicate the queue containing one value t as $\langle t \rangle$. Note that the proof rules $Q_{2_{\mathsf{L}}}$ and $Q_{1_{\mathsf{L}}}$ transform queue equations involving singleton values (built from the operation *erl_queue_val*) into term equations.

Theorem 7. Queue proof rules $Q_{0_{\mathsf{L}}}$, $Q_{1_{\mathsf{L}}}$, $Q_{1_{\mathsf{R}}}$, $Q_{2_{\mathsf{L}}}$, $Q_{2_{\mathsf{R}}}$, $Q_{3_{\mathsf{L}}}$, $Q_{3_{\mathsf{R}}}$, $Q_{4_{R\,\mathsf{L}}}$, $Q_{4_{R\,\mathsf{R}}}$, $Q_{4_{L\,\mathsf{L}}}$, $Q_{4_{L\,\mathsf{R}}}$, are sound, if ϵ is interpreted as the the empty sequence and $\cdot$ is interpreted as concatenation of value sequences.

Proof. Follows directly from the theory of sequences. □

On Page 119 proof rules (or more correctly, tactics) are provided for reasoning about queue equality on a more convenient level.

```
type erlangExpression =
      erl_expr_int of int
    | erl_expr_pid of nat
    | erl_expr_atom of atom
    | erl_expr_var of atom
    | erl_expr_nil
    | erl_expr_cons of erlangExpression, erlangExpression
    | erl_expr_tuple of erlangExpression list
    | erl_expr_application of erlangExpression, erlangExpression list
    | erl_expr_case of erlangExpression, erlangMatch
    | erl_expr_exiting of erlangExpression
    | erl_expr_try of erlangExpression list, erlangMatch
    | erl_expr_receive of erlangMatch
    | erl_expr_timeout of erlangMatch, erlangExpression
    | erl_expr_send of erlangExpression, erlangExpression
```

Figure 4.3: The Embedding of ERLANG Expressions

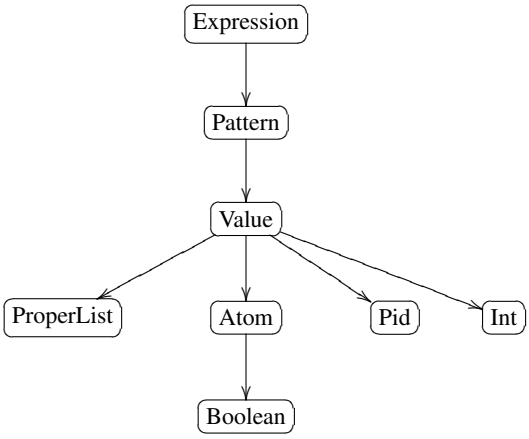

Figure 4.4: Subtypes of an ERLANG Expression

$$Q_{0_{\mathsf{L}}}\frac{}{\Gamma, t' = \epsilon \vdash \Delta}$$

$$Q_{1_{\mathsf{L}}}\frac{\Gamma, q_1 = \epsilon, q_2 = \epsilon \vdash \Delta}{\Gamma, q_1 \cdot q_2 = \epsilon \vdash \Delta} \quad Q_{1_{\mathsf{R}}}\frac{\Gamma \vdash q_1 = \epsilon, \Delta \qquad \Gamma \vdash q_2 = \epsilon, \Delta}{\Gamma \vdash q_1 \cdot q_2 = \epsilon, \Delta}$$

$$Q_{2_{\mathsf{L}}}\frac{\Gamma, t_1 = t_2 \vdash \Delta}{\Gamma, \langle t_1 \rangle = \langle t_2 \rangle \vdash \Delta} \qquad Q_{2_{\mathsf{R}}}\frac{\Gamma \vdash t_1 = t_2, \Delta}{\Gamma \vdash \langle t_1 \rangle = \langle t_2 \rangle, \Delta}$$

$$Q_{3_{\mathsf{L}}}\frac{\begin{array}{c}\Gamma, q_{11} = q_{21} \cdot Q, q_{22} = Q \cdot q_{12} \vdash \Delta \\ \Gamma, q_{12} = Q \cdot q_{22}, q_{21} = q_{11} \cdot Q \vdash \Delta \quad Q \text{ fresh}\end{array}}{\Gamma, q_{11} \cdot q_{12} = q_{21} \cdot q_{22} \vdash \Delta}$$

$$Q_{3_{\mathsf{R}}}\frac{\Gamma \vdash \begin{array}{c}\exists Q : \mathsf{erlangQueue}. q_{11} = q_{21} \cdot Q \wedge q_{22} = Q \cdot q_{12}, \\ \exists Q : \mathsf{erlangQueue}. q_{12} = Q \cdot q_{22} \wedge q_{21} = q_{11} \cdot Q\end{array}, \Delta}{\Gamma \vdash q_{11} \cdot q_{12} = q_{21} \cdot q_{22}, \Delta}$$

$$Q_{4_{R\mathsf{L}}}\frac{\Gamma, q = q \vdash \Delta}{\Gamma, q \cdot \epsilon = q' \vdash \Delta} \qquad Q_{4_{R\mathsf{R}}}\frac{\Gamma \vdash q = q', \Delta}{\Gamma \vdash q \cdot \epsilon = q', \Delta}$$

$$Q_{4_{L\mathsf{L}}}\frac{\Gamma, q = q' \vdash \Delta}{\Gamma, \epsilon \cdot q = q' \vdash \Delta} \qquad Q_{4_{L\mathsf{R}}}\frac{\Gamma \vdash q = q', \Delta}{\Gamma \vdash \epsilon \cdot q = q', \Delta}$$

Table 4.7: Basic Proof Rules for Reasoning about ERLANG Queues

Processes and Systems Next the embedding of the processes and systems is considered. An ERLANG process is either alive or dead:

$$\begin{array}{l}\texttt{type erlangProcess} = \\ \quad \textit{erl\_alive}\ \texttt{of erlangExpression, erlangPid, erlangQueue,} \\ \qquad\qquad \texttt{erlangPid}\ list, \texttt{erlangBool} \\ \quad \mid\ \textit{erl\_dead}\ \texttt{of erlangValue, erlangPid}\ list, \texttt{erlangPid}\ list\end{array}$$

ERLANG systems are defined below:

$$\begin{array}{l}\texttt{type erlangSystem} = \\ \quad \textit{erl\_proc}\ \texttt{of erlangProcess} \\ \quad \mid\ \textit{erl\_par}\ \texttt{of erlangSystem, erlangSystem}\end{array}$$

The types of ERLANG signals and system actions are given below for completeness:

$$\begin{array}{l}\texttt{type erlangSignal} = \\ \quad \textit{erl\_message}\ \texttt{of erlangValue} \\ \quad \mid\ \textit{erl\_link}\ \texttt{of erlangPid} \\ \quad \mid\ \textit{erl\_unlink}\ \texttt{of erlangPid} \\ \quad \mid\ \textit{erl\_exited}\ \texttt{of erlangPid, erlangValue} \\ \quad \mid\ \textit{erl\_exit}\ \texttt{of erlangPid, erlangValue}\end{array}$$

$$\begin{array}{l}\texttt{type erlangSysAction} = \\ \quad \textit{erl\_sact\_tau} \\ \quad \mid\ \textit{erl\_sact\_in}\ \texttt{of erlangPid, erlangValue} \\ \quad \mid\ \textit{erl\_sact\_out}\ \texttt{of erlangPid, erlangValue}\end{array}$$

4.6.2 Embedding the Transition Relations

The transition relations of the ERLANG expressions and systems are embedded in the proof system as fixed point definitions. An example excerpt, for the case of a send expression, of the definition of the ERLANG expression transition relation $\rightarrow_e$, is shown in Figure 4.5.

These four cases represents the encoding of transition rules *send*$_2$, *send*$_3$, *send*$_0$ and *send*$_1$ respectively. In the definition of the expression predicate the notion of a context is not formalised, instead the derived transition rules (e.g., *send*$_2$ and *send*$_3$) are directly asserted.

For by now obvious reasons, and when no ambiguities can arise, normal ERLANG syntax rather than the underlying term representation will be used when displaying formulas with embedded ERLANG constructs. Similarly, the formula $\rightarrow_e\ e\ \alpha\ e'$ will normally be written $e \xrightarrow{\alpha} e'$.

$$
\begin{array}{l}
\rightarrow_e \Leftarrow \\
\quad \lambda E : \mathsf{erlangExpression}, A : \mathsf{erlangExprAction}, E' : \mathsf{erlangExpression}. \\
\qquad \exists E_1, E_2, {E_1}' : \mathsf{erlangExpression}. \\
\qquad\quad E = \mathit{erl\_expr\_send}(E_1, E_2) \wedge \rightarrow_e E_1\ A\ {E_1}' \wedge E' = \mathit{erl\_expr\_send}({E_1}', E_2) \\
\quad \vee \quad \exists V_1 : \mathsf{erlangValue}, E_2, {E_1}' : \mathsf{erlangExpression}. \\
\qquad\quad E = \mathit{erl\_expr\_send}(V_1, E_2) \wedge \rightarrow_e E_2\ A\ {E_2}' \wedge E' = \mathit{erl\_expr\_send}(V_1, {E_2}') \\
\quad \vee \quad \exists \mathit{Pid}_1 : \mathsf{erlangPid}, V_2 : \mathsf{erlangExpression}. \\
\qquad\quad E = \mathit{erl\_expr\_send}(\mathit{Pid}_1, V_2) \wedge A = \mathit{erl\_eact\_out}(\mathit{Pid}_1, V_2) \wedge E' = V_2 \\
\quad \vee \quad \exists V_1 : \mathsf{erlangPid}, V_2 : \mathsf{erlangExpression}. \\
\qquad\quad E = \mathit{erl\_expr\_send}(V_1, V_2) \wedge \neg \mathit{isErlangPid}\ V_1 \wedge \\
\qquad\quad A = \mathit{erl\_eact\_exiting}(\mathit{erl\_expr\_atom}(\text{“}\mathit{badarg}\text{”})) \wedge \\
\qquad\quad E' = \mathit{erl\_expr\_atom}(\text{“}\mathit{bottom}\text{”}) \\
\quad \vdots
\end{array}
$$

Figure 4.5: Expression Transition Relation Embedded in the Proof System

The case of matching values or queues against patterns will be illustrated using the example of the *timeout* rule on Page 50. The encoding refers to a fair number of additional definitions. First, the predicate *nth* selects an ERLANG clause from a sequence of clauses:

$$
\begin{array}{l}
\mathit{nth} \Leftarrow \\
\quad \lambda N : \mathsf{nat}, M : \mathsf{erlangMatch}, C : \mathsf{erlangClause}. \\
\left(\begin{array}{ll}
 & N = 0 \wedge \exists M' : \mathsf{erlangMatch}, C' : \mathsf{erlangClause}. M = [C'|M'] \wedge C = C' \\
\vee & \exists N' : \mathsf{nat}. N = N' + 1 \wedge \\
 & \quad \exists C' : \mathsf{erlangClause}, M' : \mathsf{erlangMatch}. M = [C'|M'] \wedge \mathit{nth}\ N'\ M'\ C
\end{array} \right)
\end{array}
$$

In addition an encoding of the predicate *qmatches* introduced on Page 52 is assumed, requiring in turn the definition of predicates characterising the free variables of a pattern and substitution defined over expressions and sequences of expressions. With this, here assumed, machinery in place the clause defining the *timeout* rule becomes, utilising the new convention of writing in ERLANG syntax:

$$
\begin{array}{l}
\exists M : \mathsf{erlangMatch}, I : \mathsf{erlangInt}, Q : \mathsf{erlangQueue}. \\
\quad E = \texttt{receive}\ M\ \texttt{after}\ I\ \texttt{->}\ E'\ \texttt{end} \wedge A = \mathit{test}(Q) \wedge \\
\quad \neg (\exists N : \mathsf{nat}, C : \mathsf{erlangClause}. \mathit{nth}\ N\ M\ C \wedge \mathit{qmatches}\ Q\ C)
\end{array}
$$

The final examples of the encoding of the ERLANG expression transition relation demonstrates the handling of built-in functions. Here another clause of the coding

of the expression transition relation (see Figure 4.5) shows the transitions of the tail function `tl`:

$$\begin{array}{l}\exists L : \mathsf{erlangValue\ list}.E = \texttt{tl(}L\texttt{)} \wedge \\ \left(\begin{array}{ll} & \exists V : \mathsf{erlangValue}.L = \texttt{[}V\texttt{]} \wedge A = \tau \wedge \mathit{tl}_{\text{ERLANG}}\ V\ E' \\ \vee & \exists V : \mathsf{erlangValue}.L = \texttt{[}V\texttt{]} \wedge \\ & \quad A = \mathit{exiting}(\texttt{badarg}) \wedge \\ & \quad (\neg\exists V_1, V_2 : \mathsf{erlangValue}.V = \texttt{[}V_1 \texttt{|} V_2\texttt{]}) \wedge E' = \texttt{bottom} \\ \vee & \neg\exists V : \mathsf{erlangValue}.L = \texttt{[}V\texttt{]} \wedge \\ & \quad A = \mathit{exiting}(\texttt{cond\_clause}) \wedge E' = \texttt{bottom} \end{array}\right)\end{array}$$

Note the reference to the $\mathit{tl}_{\text{ERLANG}}$ predicate which characterises the result of applying the function to a proper argument. It is defined as follows

$$\mathit{tl}_{\text{ERLANG}} \triangleq \lambda V, V' : \mathsf{erlangValue}.\exists H : \mathsf{erlangValue}.V = [H|V']$$

The transition relations for evaluation of guards, $\rightarrow_g$, and systems, $\rightarrow_s$, is embedded in a similar fashion. Note however, that the guard transition relation $\rightarrow_g$ refers to a limited transition relation of the expressions (which in turn does not refer to the guard transition relation), to ensure the condition of positivity of predicate variables (since the guard transition occurs under a negation), to ensure monotonicity of its semantics.

Example 13. To illustrate the encoding of the semantics a simple property about the proof system and definitions will be investigated, namely whether the sequent

$$\mathit{isErlangValue}\ E, E \xrightarrow{\tau} E' \vdash$$

is provable. In other words, does the assumption that an ERLANG value can perform a transition lead to a contradiction.

To show this first approximate the $\mathit{isErlangValue}$ predicate (using the rule $\textsc{approx}_{\textsc{l}}$):

$$\mathit{isErlangValue}^{\kappa} E, E \xrightarrow{\tau} E' \vdash \tag{4.28}$$

The proof will proceed, essentially, by structural induction over the structure of the ERLANG values. Another example of this type of reasoning can be found in Section 6.2. First unfold the value definition using $\textsc{unf2l}_{\textsc{l}}$, and then eliminate the disjunctions with $\vee_{\textsc{l}}$. We consider two resulting proof goals, a base case and a goal where an inductive argument is necessary; the proofs of other goals are analogous.

$$\exists I : \mathsf{int}.E = \mathit{erl\_expr\_int}(I), E \xrightarrow{\tau} E' \vdash \tag{4.29}$$

$$\begin{array}{l}\exists E_1, E_2 : \mathsf{erlangExpression}.E = \texttt{[}E_1 \texttt{|} E_2\texttt{]} \wedge \\ \qquad \mathit{isErlangValue}^{\kappa'} E_1 \wedge \mathit{isErlangValue}^{\kappa'} E_2, E \xrightarrow{\tau} E' \vdash\end{array} \tag{4.30}$$

The proof proceeds by eliminating the existential quantifiers ($\exists_L$), eliminating the conjunctions, and applying the SUBST rule, resulting in goals:

$$\mathit{erl\_expr\_int}(I) \xrightarrow{\tau} E' \vdash \tag{4.31}$$

$$\kappa' < \kappa, \mathit{isErlangValue}^{\kappa'} E_1, \mathit{isErlangValue}^{\kappa'} E_2, [\, E_1 \mid E_2 \,] \xrightarrow{\tau} E' \vdash \tag{4.32}$$

The proof of goal 4.31 proceeds with unfolding the transition relation predicate, and eliminating the disjunctions. A large number of cases results, all of which are trivially provable since every clause starts with an equality that is trivially false because no expression E matches $\mathit{erl\_expr\_int}(I)$. Consider instead goal 4.32. Again unfold the definition of the transition predicate and proceed as for the previous goal. There are two goals that are not trivially provable:

$$\kappa' < \kappa, \mathit{isErlangValue}^{\kappa'} E_1, \mathit{isErlangValue}^{\kappa'} E_2, E_1 \xrightarrow{\tau} {E_1}', E' = [\, {E_1}' \mid E_2 \,] \vdash \tag{4.33}$$

$$\kappa' < \kappa, \mathit{isErlangValue}^{\kappa'} E_1, \mathit{isErlangValue}^{\kappa'} E_2, E_2 \xrightarrow{\tau} {E_2}', E' = [\, E_1 \mid {E_2}' \,] \vdash \tag{4.34}$$

Now both these goals can be discharged against goal 4.28 since, intuitively, there exists a substitution from variables in goal 4.28 to terms in goal 4.33, for example, such that all assertions in goal 4.28 can be found in goal 4.33. Secondly a least fixed point definition has been unfolded to the left, and an ordinal has been decreased ($\kappa' < \kappa$ is provable). Thus both goals can be solved with the DISCHARGE rule, and thus the claim holds.

4.6.3 Expression Properties

In this section a number of key properties of expressions, and expression evaluation, is defined formally in the underlying logic by referring to the expression transition relation.

Definition 57. *An* ERLANG *expression E has finished its computation and represents an* ERLANG *value $v \in$ erlangValue, if the predicate assertion E : the_term v is provable, where the_term $\triangleq \lambda V$: erlangValue.V*

From the semantics of formulas it is clear that E : *the_term* v if and only if $E = v$. A proof rule THE_TERM is derivable:

$$\frac{\Gamma\{v/E\} \vdash \Delta\{v/E\}}{\Gamma, E : \mathit{the\_term}\ v \vdash \Delta}$$

In this thesis any action but the computation step or an exception is considered a "side effect".

Definition 58. *An expression* e *is* side-effect free*, if it satisfies the predicate sef (side-effect free):*

$$\begin{array}{l} \mathit{sef} \Rightarrow \\ \quad \forall A : \mathsf{erlangExprAction}. \\ \quad\quad [A]\,((A = \tau \wedge \mathit{sef}) \vee (\exists V : \mathsf{erlangValue}. A = \mathit{exiting}(V) \wedge \mathit{sef})) \end{array}$$

In other words an expression is side effect free if it never performs any action but computational steps or exceptions.

For completeness an analogue definition is provided for the case when no exceptions are raised:

Definition 59. *An expression* e *is* functional*, if it satisfies the predicate functional:*

$$\mathit{functional} \Rightarrow \forall A : \mathsf{erlangExprAction}.[A]\,(A = \tau \wedge \mathit{functional})$$

Definition 60. *An expression* e terminates*, if it satisfies the predicate terminates:*

$$\mathit{terminates} \Leftarrow \forall A : \mathsf{erlangExprAction}.[A]\mathit{terminates}$$

Definition 61. *An expression* e strongly normalises to a value v*, if it satisfies the predicate normalizes*(v)*:*

$$\begin{array}{l} \mathit{normalizes} \Leftarrow \\ \quad \lambda V : \mathsf{erlangValue}. \forall A : \mathsf{erlangExprAction}. \\ \quad\quad [A](\mathit{normalizes}\ V) \wedge\ (\exists A : \mathsf{erlangExprAction}.\langle A\rangle \mathit{true} \vee \mathit{the\_term}\ V) \end{array}$$

Naturally these predicates can be combined to express, for instance, the property that an expression normalises without side-effects or raising an exception, to a value v:

$$\mathit{functional} \wedge (\mathit{normalizes}\ v)$$

but instead a more direct, but provably equal definition, is usually preferred:

$$\begin{array}{l} \mathit{fnormalizes} \Leftarrow \\ \quad \lambda V : \mathsf{erlangValue}. \\ \quad\quad \left(\begin{array}{ll} & \forall A : \mathsf{erlangExprAction}.[A]\,(A = \tau \wedge (\mathit{fnormalizes}\ V)) \\ \wedge & \langle\tau\rangle \mathit{true} \vee \mathit{the\_term}\ V \end{array} \right) \end{array}$$

4.6.4 System Properties

In this section a number of key properties of systems, and system evaluation, is defined formally in the underlying logic. First, a system analogue to the *the_term* v property, is defined inductively over the structure of systems:

Definition 62.

$$
\begin{array}{l}
\mathit{sthe\_term} \Leftarrow \\
\quad \lambda \mathit{Pid} : \mathsf{erlangPid}, V : \mathsf{erlangValue}, S : \mathsf{erlangSystem}. \\
\left(\begin{array}{ll}
 & \exists Q : \mathsf{erlangQueue}, Pl : \mathsf{erlangPid}\ list, B : \mathsf{erlangBool}. \\
 & \quad S = \langle V, \mathit{Pid}, Q, Pl, B \rangle \\
\vee & \exists S_1, S_2 : \mathsf{erlangSystem}. \\
 & \quad S = S_1 \parallel S_2 \wedge (S_1 : \mathit{sthe\_term\ Pid}\ V \vee S_2 : \mathit{sthe\_term\ Pid}\ V)
\end{array} \right)
\end{array}
$$

The property holds of a process, with process identifier *Pid*, if it has terminated its computation with the value V. A queue analogue is:

Definition 63.

$$
\begin{array}{l}
\mathit{sthe\_queue} \Leftarrow \\
\quad \lambda \mathit{Pid} : \mathsf{erlangPid}, Q : \mathsf{erlangQueue}, S : \mathsf{erlangSystem}. \\
\left(\begin{array}{ll}
 & \exists V : \mathsf{erlangValue}, Pl : \mathsf{erlangPid}\ list, B : \mathsf{erlangBool}. \\
 & \quad S = \langle V, \mathit{Pid}, Q, Pl, B \rangle \\
\vee & \exists S_1, S_2 : \mathsf{erlangSystem}. \\
 & \quad S = S_1 \parallel S_2 \wedge (S_1 : \mathit{sthe\_queuePid}\ Q \vee S_2 : \mathit{sthe\_queuePid}\ Q)
\end{array} \right)
\end{array}
$$

Frequently there is a need to specify that a process identifier belongs, or does not belong, to a process in a system. Clearly, a direct syntactical characterisation of this property, similar to the definition of *sthe_term* above is possible:

$$
\begin{array}{l}
\mathit{pid\_in\_sys} \Leftarrow \\
\quad \lambda P : \mathsf{erlangPid}. S : \mathsf{erlangSystem}. \\
\quad \exists E : \mathsf{erlangExpression}, Q : \mathsf{erlangQueue}. Pl : \mathsf{erlangPid}\ list, B : \mathsf{erlangBool}. \\
\quad \left(\begin{array}{ll}
 & S = \langle E, \mathit{Pid}, Q, Pl, B \rangle \\
\vee & \exists S_1, S_2 : \mathsf{erlangSystem}. \\
 & \quad S = S_1 \parallel S_2 \wedge (S_1 : \mathit{pid\_in\_sys\ Pid} \vee S_2 : \mathit{pid\_in\_sys\ Pid})
\end{array} \right)
\end{array}
$$

However, a more indirect characterisation is also possible, let us call these predicates *local* and *foreign*:

$$
\begin{array}{l}
\mathit{local} : \mathsf{erlangPid} \rightarrow \mathsf{erlangSystem} \rightarrow \mathsf{prop} \stackrel{\Delta}{=} \\
\quad \lambda P : \mathsf{erlangPid}. \lambda S : \mathsf{erlangSystem}. S : \exists V : \mathsf{erlangValue}. \langle P?V \rangle \mathit{true} \\
\mathit{foreign} : \mathsf{erlangPid} \rightarrow \mathsf{erlangSystem} \rightarrow \mathsf{prop} \stackrel{\Delta}{=} \\
\quad \lambda P : \mathsf{erlangPid}. \lambda S : \mathsf{erlangSystem}. \forall V : \mathsf{erlangValue}. [P?V] \mathit{false}
\end{array}
$$

Given these definitions, the following sequents are valid:

$$\begin{aligned} s : \mathit{pid\_in\_sys} &\vdash s : \mathit{local} \\ s : \mathit{local} &\vdash s : \mathit{pid\_in\_sys} \\ s : \mathit{foreign} &\vdash s : \neg \mathit{pid\_in\_sys} \\ s : \neg \mathit{foreign} &\vdash s : \mathit{pid\_in\_sys} \end{aligned}$$

4.6.5 Deriving Convenient Operational Semantics Rules

Previous efforts in the development of a proof system for reasoning about ERLANG code [DF98, DFG98b] did not directly encode the operational semantics rules in the proof system. Rather, various types of proof rules for syntactical constructs and modalities were given, such as the rule $\parallel \langle ? \rangle$ below.

$$\parallel \langle ? \rangle \frac{\Gamma, S : \phi \vdash S \parallel s_2 : \psi, \Delta}{\Gamma, s_1 : \langle \mathit{pid}?v \rangle \phi \vdash s_1 \parallel s_2 : \langle \mathit{pid}?v \rangle \psi, \Delta}$$

We claim that the present approach of directly encoding the semantics is much more natural, and permits additional theorems about the semantics to be stated and proved, and generally leads to less mistakes in the encoding process. The previous basic proof rules are now instead derivable, except in cases where the semantics has changed, e.g., notably for internal actions due to process spawning.

For transition formulas $s \xrightarrow{\alpha} s'$ rules can be derived that trigger on the syntactic shape of the systems s and the action α (similar rules can of course be derived for the expression case). These rules are highly useful when reasoning primarily about formulas involving modalities. As a first example, the case of expression contexts is considered.

Example 14 (Deriving Rules for Redexes)**.** To establish that reduction contexts in the semantics truly define the redexes of an expression, we can derive a proof rule like the one below

$$\mathsf{CONTEXT}[\alpha] \frac{\Gamma, E : \phi \vdash r[E] : \psi, \Delta}{\Gamma, e : [\alpha]\phi \vdash r[e] : [\alpha]\psi, \Delta}$$

permitting us to focus solely on the actions of the redex to compute actions of the whole expression. Similarly a rule for $\langle \alpha \rangle$ can be established:

$$\mathsf{CONTEXT}\langle \alpha \rangle \frac{\Gamma, E : \phi \vdash r[E] : \psi, \Delta}{\Gamma, e : \langle \alpha \rangle \phi \vdash r[e] : \langle \alpha \rangle \psi, \Delta}$$

Example 15 (Rules for Parallel Composition and Modalities)**.** For the case of a parallel composition, and an input action, the two rules $\parallel ?_{\mathsf{L}}$ and $\parallel ?_{\mathsf{R}}$ depicted in Table 4.8 are derivable.

As a further example, the proof rule $\parallel \langle ? \rangle$ given above, originally from [DFG98b], is now derivable, as seen in Figure 4.6. The example uses the derived proof rule $\parallel ?1_{\mathsf{R}}$ in Table 4.7 for reasoning about the parallel composition.

$$\|\ ?_{\mathsf{L}}\ \frac{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} S, s' = S \parallel s_2 \vdash \Delta \qquad \Gamma, s_2 \xrightarrow{\mathit{pid}?v} S, s' = s_1 \parallel S \vdash \Delta}{\Gamma, s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} s' \vdash \Delta}$$

$$\|\ ?_{\mathsf{R}}\ \frac{\Gamma \vdash \begin{array}{l} \exists S : \mathsf{erlangSystem}.s_1 \xrightarrow{\mathit{pid}?v} S \wedge s' = S \parallel s_2, \\ \exists S : \mathsf{erlangSystem}.s_2 \xrightarrow{\mathit{pid}?v} S \wedge s' = s_1 \parallel S, \\ \Delta \end{array}}{\Gamma \vdash s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} s', \Delta}$$

Table 4.8: Derived Rules for Parallel Composition and Input

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} S \vdash s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} S \parallel s_2, \Delta}{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} S \vdash s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} S \parallel s_2, \Delta}\ \|\ ?1_{\mathsf{R}}}{\Gamma' \vdash s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} S \parallel s_2, \Delta}\ \mathsf{W}_{\mathsf{L}} \qquad \dfrac{\Gamma, S : \phi \vdash S \parallel s_2 : \psi, \Delta}{\Gamma' \vdash S \parallel s_2 : \psi, \Delta}\ \mathsf{W}_{\mathsf{R}}}{\Gamma' \vdash \exists S.s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} S \wedge S : \psi, \Delta}\ \exists_{\mathsf{R}}, \wedge_{\mathsf{R}}}{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} S, S : \phi \vdash s_1 \parallel s_2 : \langle \mathit{pid}?v \rangle \psi, \Delta}\ \langle\rangle_{\mathsf{R}}}{\Gamma, s_1 : \langle \mathit{pid}?v \rangle \phi \vdash s_1 \parallel s_2 : \langle \mathit{pid}?v \rangle \psi, \Delta}\ \langle\rangle_{\mathsf{L}}$$

Figure 4.6: The Derivation of Proof Rule $\| \langle ? \rangle$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} s_1' \vdash s_1 \xrightarrow{\mathit{pid}?v} s_1', \Delta}\ \mathsf{ID} \qquad \dfrac{}{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} s_1' \vdash s_1' \parallel s_2 = s_1' \parallel s_2, \Delta}\ \mathsf{REFL}}{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} s_1' \vdash s_1 \xrightarrow{\mathit{pid}?v} s_1' \wedge s_1' \parallel s_2 = s_1' \parallel s_2, \Delta}\ \wedge_{\mathsf{R}}}{\Gamma, s_1 \xrightarrow{\mathit{pid}?v} s_1' \vdash s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} s_1' \parallel s_2, \Delta}\ \|\ ?_{\mathsf{R}}, \mathsf{W}_{\mathsf{R}}, \exists_{\mathsf{R}}$$

Figure 4.7: The Derivation of Proof Rule $\|\ ?1$

$$\|\ [?]\ \frac{\begin{array}{c}\Gamma, S_1 : \phi_1 \vdash S_1 \parallel s_2 : \phi, \Delta \\ \Gamma, S_2 : \phi_2 \vdash s_1 \parallel S_2 : \phi, \Delta\end{array}}{\Gamma, s_1 : [\mathit{pid}?v]\phi_1, s_2 : [\mathit{pid}?v]\phi_2 \vdash s_1 \parallel s_2 : [\mathit{pid}?v]\phi, \Delta}$$

$$\|\ [!]\ \frac{\begin{array}{c}\Gamma, S_1 : \phi_1, \mathit{pid} \notin \mathit{pids}(s_2) \vdash S_1 \parallel s_2 : \phi, \Delta \\ \Gamma, S_2 : \phi_2, \mathit{pid} \notin \mathit{pids}(s_1) \vdash s_1 \parallel S_2 : \phi, \Delta\end{array}}{\Gamma, s_1 : [\mathit{pid}!v]\phi_1, s_2 : [\mathit{pid}!v]\phi_2 \vdash s_1 \parallel s_2 : [\mathit{pid}!v]\phi, \Delta}$$

$$\|\ [\tau]\ \frac{\begin{array}{c}\Gamma, S_1 : \phi_1, S_2 : \phi_4 \vdash S_1 \parallel S_2 : \phi, \Delta \\ \Gamma, S_1 : \phi_3, S_2 : \phi_2 \vdash S_1 \parallel S_2 : \phi, \Delta \\ \Gamma, S_1 : \phi_5, \mathit{pids}(S_1) \cap \mathit{pids}(s_2) = \emptyset \vdash s_1 \parallel S_2 : \phi, \Delta \\ \Gamma, S_2 : \phi_6, \mathit{pids}(s_1) \cap \mathit{pids}(S_2) = \emptyset \vdash s_1 \parallel S_2 : \phi, \Delta\end{array}}{\Gamma, \left\{\begin{array}{c} s_1 : \forall P, V.[\mathit{pid}!v]\phi_1, \\ s_2 : \forall P, V.[\mathit{pid}!v]\phi_2, \\ s_1 : \forall P, V.[\mathit{pid}?v]\phi_3, \\ s_2 : \forall P, V.[\mathit{pid}?v]\phi_4, \\ s_1 : [\tau]\phi_5, s_2 : [\tau]\phi_6 \end{array}\right\} \vdash s_1 \parallel s_2 : [\tau]\phi, \Delta}$$

Table 4.9: Derived Rules for $\|\ [\alpha]$

The treatment of parallel compositions is crucial in proofs; frequently we decompose proofs by abstracting the parts of a parallel composition. For this reason we provide here example proof rules for the necessity modality in Table 4.9 and the possibility modality in Table 4.10.

Note for instance the problematical rule $\|\ \langle\tau\rangle 1$, which contains the proof obligation $\mathit{pids}(S_1) \cap \mathit{pids}(s_2) = \emptyset$ which guards against process spawning by the left process s_1. Actually some liberties are taken with notation in this example. Since the logic does not admit non-constructor functions, the side condition is not representable in the proof system,. An alternative characterisation is the following claim:

$$\begin{array}{ll} \forall \mathit{Pid} : \mathsf{erlangPid}. & \\ \quad (S_1 : \langle \mathit{Pid}?0\rangle \mathit{true} \Rightarrow S_2 : [\mathit{Pid}?0]\mathit{false}) & \\ \wedge \quad (S_2 : \langle \mathit{Pid}?0\rangle \mathit{true} \Rightarrow S_1 : [\mathit{Pid}?0]\mathit{false}) & \end{array}$$

4.6.6 A More Convenient Theory of Matching

The operation of matching a value against a sequence of patterns, or a queue of values against such a sequence, is a basic operation of ERLANG. Matching takes place during every function application, in every `case` statement, and in every `receive` statement. As such it is crucial that the handling of this operation is convenient in the proof system.

$$\parallel \langle ? \rangle \frac{\Gamma, S_1 : \phi_1 \vdash S_1 \parallel s_2 : \phi, \Delta}{\Gamma, s_1 : \langle \mathit{pid}?v \rangle \phi_1 \vdash s_1 \parallel s_2 : \langle \mathit{pid}?v \rangle \phi, \Delta}$$

$$\parallel \langle ! \rangle \frac{\Gamma, S_1 : \phi_1 \vdash S_1 \parallel s_2 : \phi, \Delta}{\Gamma, s_1 : \langle \mathit{pid}!v \rangle \phi_1, \mathit{pid} \notin \mathit{pids}(s_2) \vdash s_1 \parallel s_2 : \langle \mathit{pid}!v \rangle \phi, \Delta}$$

$$\parallel \langle ! \rangle 1 \frac{\Gamma, S_1 : \phi_1, S_2 : \phi_2 \vdash S_1 \parallel S_2 : \phi, \Delta}{\Gamma, s_1 : \langle \mathit{pid}!v \rangle \phi_1, s_2 : \langle \mathit{pid}?v \rangle \phi_2 \vdash s_1 \parallel s_2 : \langle \tau \rangle \phi, \Delta}$$

$$\parallel \langle \tau \rangle 1 \frac{\Gamma, S_1 : \phi_1 \vdash S_1 \parallel s_2 : \phi, \Delta \quad \Gamma, S_1 : \phi_1 \vdash \mathit{pids}(S_1) \cap \mathit{pids}(s_2) = \emptyset, \Delta}{\Gamma, s_1 : \langle \tau \rangle \phi_1 \vdash s_1 \parallel s_2 : \langle \tau \rangle \phi, \Delta}$$

Table 4.10: Derived Rules for $\parallel \langle \alpha \rangle$ (symmetrical rules omitted)

Here, instead of referring to the definition of *matches* an alternative and more direct characterisation is given for some special cases, as proof rules available in the proof system.

First, derived operational semantics rules, and corresponding derived proof rules, for case_{0k}, $\mathit{receive}_k$, and $\mathit{timeout}_k$ are provided, for the case when all the clauses in a match (a sequence of clauses) are enumerated, and the parameter k refers to clause k. Then the definitions of quantification over clauses can be replaced with enumeration. We show below the new rule for case_{0k}:

$$\mathit{case}_{0k} \frac{\begin{array}{c} \mathit{result}(v, p_k \texttt{ when } g_k \texttt{ -> } e_k, e_k) \\ \neg \mathit{matches}(v, p_1 \texttt{ when } g_1 \texttt{ -> } e_1) \\ \vdots \\ \neg \mathit{matches}(v, p_{k-1} \texttt{ when } g_{k-1} \texttt{ -> } e_{k-1}) \end{array}}{\texttt{case } v \texttt{ of } p_1 \texttt{ when } g_1 \texttt{ -> } e_1 \texttt{;} \ldots \texttt{;} p_n \texttt{ when } g_k \texttt{ -> } e_n \texttt{ end} \rightarrow e}$$

Further, in the case when a pattern p does not contain any (proof system) variables that range over the ERLANG variables, for example if the pattern contains only variables ranging over the ERLANG values, then the free ERLANG variables of the pattern can be computed. For this situation, four convenient proof rules are derivable (below

only the right-hand side rules are given):

$$\textsf{MATCHES}_{0_R}\frac{\Gamma \vdash \exists \tilde{V}' : \textsf{erlangValue}.\ \begin{pmatrix} v = p[\tilde{V}'/\tilde{V}] \\ \wedge \quad g[\tilde{V}'/\tilde{V}] \rightarrow_{\mathcal{G}} \texttt{true} \end{pmatrix}, \Delta}{\Gamma \vdash \mathit{matches}(v, p\ \texttt{when}\ g\ \texttt{->}\ e), \Delta}$$

$$\textsf{RESULT}_{0_R}\frac{\Gamma \vdash \exists \tilde{V}' : \textsf{erlangValue}.\ \begin{pmatrix} v = p[\tilde{V}'/\tilde{V}] \\ \wedge \quad g[\tilde{V}'/\tilde{V}] \rightarrow_{\mathcal{G}} \texttt{true} \\ \wedge \quad e[\tilde{V}'/\tilde{V}] = e' \end{pmatrix}, \Delta}{\Gamma \vdash \mathit{result}(v, p\ \texttt{when}\ g\ \texttt{->}\ e, e'), \Delta}$$

In these rules the side condition $fv(p) = \tilde{V}$ where $\tilde{V}$ are the ERLANG variables is assumed. Further the operation of substituting proof system variables $\tilde{V}'$ for ERLANG variables $\tilde{V}$ (or, put differently, substituting variables for terms) in an expression e is denoted with $e[\tilde{V}'/\tilde{V}]$. Note that the substitution operation is well-defined only because there are no (proof system) variables in p ranging over patterns.

Next, proof rules for the decomposition of queues under the *qmatches* predicate are given, here only for the left-hand side:

$$\textsf{QUEUE1}_{\textsf{L}}\frac{\Gamma, \neg\,(q = \epsilon) \vdash \Delta}{\Gamma, \mathit{qmatches}(q, \mathit{erl\_expr\_var}(a)\ \texttt{when}\ \texttt{true}\ \texttt{->}\ e) \vdash \Delta}$$

$$\textsf{QUEUE2}_{\textsf{L}}\frac{}{\Gamma, \mathit{qmatches}(\epsilon, p\ \texttt{when}\ g\ \texttt{->}\ e) \vdash \Delta}$$

$$\textsf{QUEUE3}_{\textsf{L}}\frac{\Gamma, \mathit{qmatches}(q_1, p\ \texttt{when}\ g\ \texttt{->}\ e) \vdash \Delta \quad \Gamma, \mathit{qmatches}(q_2, p\ \texttt{when}\ g\ \texttt{->}\ e) \vdash \Delta}{\Gamma, \mathit{qmatches}(q_1 \cdot q_2, p\ \texttt{when}\ g\ \texttt{->}\ e) \vdash \Delta}$$

$$\textsf{QUEUE4}_{\textsf{L}}\frac{\Gamma, \mathit{matches}(v, p\ \texttt{when}\ g\ \texttt{->}\ e) \vdash \Delta}{\Gamma, \mathit{qmatches}(v, p\ \texttt{when}\ g\ \texttt{->}\ e) \vdash \Delta}$$

Note the restrictive condition in $\textsf{QUEUE1}_{\textsf{L}}$ corresponding to the case when a queue is matched against a variable, which should always succeed unless the queue is empty.

Chapter 5

An Implementation of the Proof System

The proof system described in Chapter 4 has been implemented in a proof assistant tool, or proof checker, named the "ERLANG verification tool", abbreviated EVT. It has been tailored to the underlying proof system; rather than working with a set of open goals, the underlying data structure is an acyclic proof graph, to make it possible to check the side conditions of the global discharge condition. The main reason for developing a new tool is our desire to experiment with different implementation strategies for the discharge condition and with the underlying proof graph representation. Moreover most existing theorem provers are rather inflexible in that they offer a set of predefined induction schemes, from which the user has to choose one at the outset of the proof. This contrasts with our ambition to discover induction schemes through a lazy search procedure in the course of the proof.

Two notable versions of the proof assistant exists. The first release was reported in [ADFG98] and was an experimental prototype tailored especially to the verification of ERLANG code. An example verification conducted with this tool is reported in [AD99]. The second and current tool release is more general, permitting the embedding of theories for other languages. Apart from the support for ERLANG, an experimental embedding of a variant of the value-passing Calculus of Communicating Systems [Mil89] (CCS for short) exists.

The current tool is, like the HOL [Ge93] and Isabelle [Pau94] theorem provers written in the Standard ML [MTH97] programming language. The Standard ML top level is used to provide a command line interface for interacting with the tool. In addition there is also a graphical user interface programmed in Java, and support for proof graph visualisation using the daVinci graphical visualisation system [FW94].

Further information about the EVT tool, including a reference manual documenting its commands and proof rules, and the possibility to download the tool, is available on the web page `http://www.sics.se/fdt/VeriCode/evt.html`.

5.1 Terms, Variables, Formulas and Proofs

As its basis the tool implements the many-sorted first order logic described in Chapter 2. Among the minor differences between EVT and the presentation in this thesis is the introduction in the tool of additional logical constructs in the core logic, e.g., truth, falsity, and conjunction are basic rather than derived constructs.

Recursively defined types can be defined and equipped with type specific parsers and pretty printers to enable reading and printing of terms and formulas in native formats. The presence of subtyping in the underlying theory can, as usual, introduce proof obligations during parsing of terms and formulas. Consider for instance the predicate definition

$$isTrue \triangleq \lambda X : \mathsf{erlangBool}.X = \texttt{true}$$

which checks whether its ERLANG boolean argument is equal to the atom true. With this definition the proof goal $\Gamma \vdash V : isTrue$ does not typecheck if V is of type erlangValue. However, the assumptions in Γ may contain enough information to deduce that V is in fact an ERLANG boolean. If this cannot be established automatically, the tool will generate the proof obligation $\Gamma \vdash V : isErlangBool$ where $isErlangBool$ is the predicate that distinguishes the booleans from other ERLANG values.

For types considered to be freely-generated such as the type of the natural numbers recursive predicates can be generated automatically that permit structural induction style arguments about elements of the type.

Sequents are ordered sequences of formulas. These may contain free variables, which are of two kinds: *parameters* which are generated by rules such as $\exists_{\mathsf{L}}$ and *meta-variables*, the result of postponing the choice of a witness in a proof rule such as $\exists_{\mathsf{R}}$ defined in Table 4.1 on Page 78. To ensure that assignments to meta-variables are sound a simple scheme based on associating indices to variables from Sahlin et al. [SFH92], is used. Bound variables are represented using de Bruijn indices, to permit checking equality of formulas quickly up to α-conversion. This is important for obtaining efficient implementations of the discharge rule.

From an EVT user's point of view, proving a property of an Erlang program involves "backward", i.e., goal-directed construction of a proof graph. Each proof node in the graph is either a leaf node, meaning that it either represents an open goal or that the sequent was solved by the application of an axiom proof rule without premises, or it is a parent node that has been reduced by applying the proof rule and such that its children nodes correspond to the premises of the rule. The implementation of the global discharge condition defined in Section 4.5.3 is split into different parts. One local check, in this chapter referred to as an application of the discharge rule, is represented in the proof graph by a directed arc from the discharged node to its companion node. Arcs in the proof tree are labelled by the proof rule that caused the arc to appear.

Open proof goals may also be copydischarged, or *subsumed* in more standard terminology, when instances of the goal can be found elsewhere in the proof graph. However, there are two restrictions. First, no open proof goal can be copydischarged against an ancestor proof node. Second an acyclicity condition is enforced to prevent cyclic copydischarges, for example such that node N is copydischarged against node M which

is in turn copydischarged against node N.

The application of a proof rule can be cancelled, resulting potentially in non-local cancellation effects on the proof tree when e.g. the companion node of a copydischarge node is cancelled, naturally also causing the copydischarge to fail. Another such problematic case is when a meta-variable is assigned, and cancelled in a proof branch, but where the meta-variable is also present in a second branch. In such a situation both the assignment and the cancellation may also affect the proof steps in the second branch. To implement a sound cancellation scheme in spite of these difficulties a global ordering of sequents is introduced, based on the absolute order in which proof nodes were introduced by application of rules.

A node in a proof graph can also be discharged due to a lemma, or proof, that exists in a different proof tree. In the terminology of EVT these will be called "lemmadischarges". To prohibit mutual dependencies between lemmas a test for cyclic dependencies is performed.

A complete proof is then, in the tool, a collection of proof graphs such that all proof nodes have been discharged due to an axiom rule, or because it is subsumed by another proof node, or because of a lemma proved elsewhere, or because the node can be discharged to a fixed point argument involving the global discharge condition.

5.2 Rules, Tactics, Tacticals and Proof Scripts

The basic proof rules of the proof assistant are enumerated in Chapter 4 and are implemented in the tool as *tactics*, which are functions in the Standard ML (SML) sense from a sequent representing the current goal (the conclusion) to a pair consisting of a list of goals (the premises of the rule) and a list of assignments to meta-variables caused by the tactic. Thus, if the SML type of sequents is `seq` and meta-variables are of type `var` and terms are represented by the type `term` then the type of a tactic is

```
type tactic =
  seq -> (seq list * (var * term) list)
```

Most rules trigger on a particular assertion position in a sequent, and thus requires a positive natural number argument to determine where in the sequent the rule should applied. For instance, the tactic implementing the proof rule $\vee_R$ has the signatures `or_r: int -> tactic`.

Similar to other proof assistants like Isabelle [Pau94] and HOL [Ge93] EVT provides tactic combinators, or *tacticals*, which give the possibility to derive new sound tactics from basic tactics. Examples of such tacticals, with their signatures, are

```
t_skip:
  tactic
t_pretactic:
  (seq -> 'a option) -> ('a -> tactic) -> tactic
t_compose:
   tactic -> tactic -> tactic
```

```
t_bool:
   (seq -> bool) -> tactic -> tactic -> tactic
t_fix:
   'a -> ('a -> ('a -> tactic) -> tactic) -> tactic
```

Informally their behaviour is the following: The tactical `t_skip` succeeds returning simply its sequent argument, and causing no assignments. The tactical `t_pretactic` permits analysis of its sequent argument in a safe way. Its first argument is a predicate that checks some condition on the current sequent and returns some value if the test succeeds, and the value `NONE` otherwise. If the test succeeds the tactic provided in the second argument to `t_pretactic` is applied to the return value of the predicate. The tactical `t_compose t1 t2` applies the tactic `t1` to the current sequent, and then applies in `t2` to the resulting sequents. In the alternative construct `t_orelse t1 t2` first tactic `t1` is applied. If this fails, then tactic `t2` is applied instead. The `t_bool` tactical accepts as its first argument a function that performs a test on the current sequent. If the test succeeds evaluation proceeds with `t1`, otherwise with tactic `t2`. Finally `t_fix` can be used to write recursive tactics. Its first argument is an arbitrary initialisation value and its second argument is a function, which should implement the body of the tactic, and which accepts an arbitrary parameter and a "continuation" tactic as arguments.

The above tactics essentially perform depth-first traversal and reduction of the proof graph being built.

Example 16. Consider a simple example. First we define a new tactical, `recurse_down` that accepts one tactic argument and applies it to all the argument positions on the right hand side of a sequent, starting from the greatest position. Some explanation of the code is in order. The resulting tactical, of type `(int -> tactic) -> tactic`, has the following first behaviour. First the number of assertions on the right-hand side in `seq` is calculated by the code fragment `List.length(Sequent.get_rhs seq)`. The `t_fix` tactical loops until, as evidenced by the recursive call to `continuation` with a decreased position parameter, until the position parameter becomes zero. At every decreasing position the tactic argument, or more properly the abstraction from a position (`int`) to a tactic `tac`, is applied to the current position.

```
fun recurse_down tac =
  t_pretactic
  (fn seq => SOME(List.length(Sequent.get_rhs seq)))
  (fn pos =>
    t_fix pos
    (fn position => fn continuation =>
      t_bool
      (fn seq => pos>0)
      (t_compose
        (t_orelse (tac pos) t_skip)
        (continuation (pos-1)))
      t_skip));
```

We can use the new tactical to, for instance, deriving a new tactic for applying the the disjunction introduction rule $\vee_R$, or as it is named in the tool, `or_r`, everywhere on the right hand side of a sequent. The application `recurse_down or_r` yields such a tactic.

5.3 User Interfaces and Commands

The standard user interface to the proof assistant is the conventional command line interface of Standard ML of New Jersey to which a number of *commands* for interacting with the proof assistant have been added. Conceptually the user interface defines proof notions such as "which is the current proof graph" and "which is the current proof node". The commands of the proof assistant operate on proof graphs, possibly causing side effects. For instance, there are commands to start a new proof, to define a lemma, to navigate proof graphs, to navigate the hierarchy of proof graphs, to grow or possibly complete a proof graph by applying a tactic to its current sequent resulting in new proof goals, and to cancel a previous command. Further the local check of the global discharge condition, and discharging due to subsumption, are implemented as commands rather than tactics since they cause side effects to the graph structure.

An alternative to combining tactics using tacticals is to directly use the Standard ML programming language facilities to define functions that execute commands. This, however, has the disadvantage that all intermediate proof nodes are stored in the proof graph. In contrast, using tactical combinators, intermediate proof nodes used in the computation of the effects of a tactic are never stored in the proof graph.

A second, graphical, user interface is also available. This user interface consists of two parts: the first is programmed in Java and provides additional user assistance through the implementation of modern theorem prover features [BT98] such as "proof–by–pointing" (to suggest, based on the proof context, the next proof rule to apply), a more structured database of lemmata, proof recording and playback, etc. A screen shot of a proof session using the graphical user interface is shown in Figure 5.1. The second component of the graphical user interface is used to visualise and navigate through the proof graph, and is implemented by interfacing with the daVinci [FW94] graph visualisation system. Experiences with the graphical interface indicate that the initial training period required to become familiar with the tool is considerably shortened.

5.4 Fixed Point Rules and Checking the Discharge Condition

Of particular interest is the implementation of the global discharge condition which is the foundation for inductive and co-inductive reasoning. To summarise the formal treatment in Section 4.5.3 the goal is to check whether a proof node (the *discharge node*) can be discharged since it is an instance of an ancestor node (the *companion node*), and since appropriate fixed points have been unfolded.

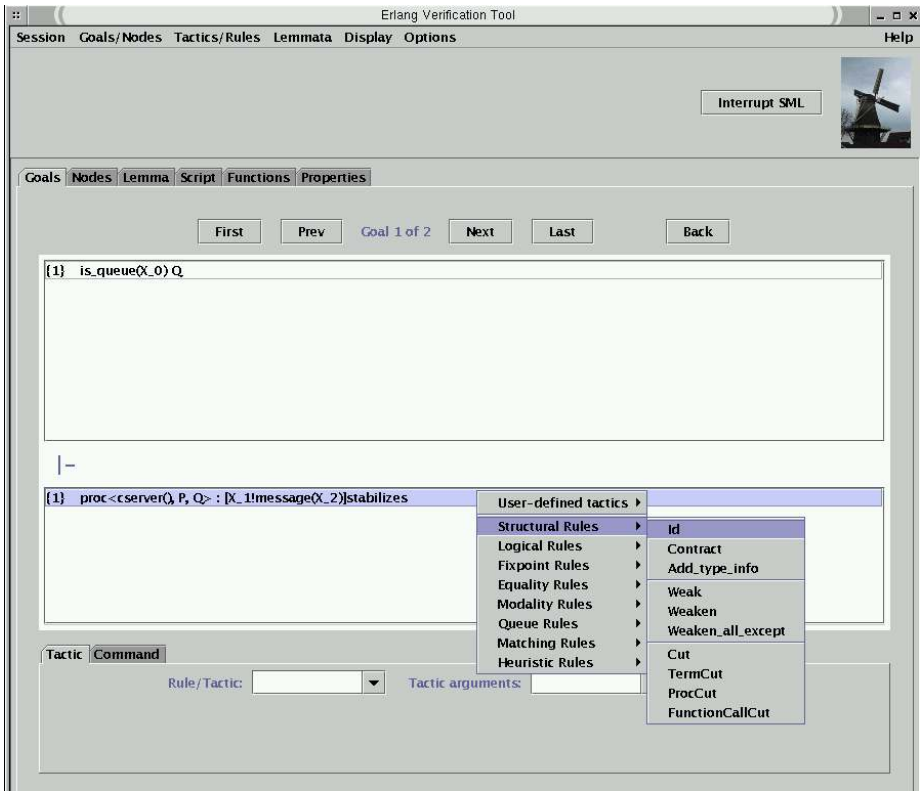

Figure 5.1: The Graphical User Interface of EVT

In fact the global discharge condition is implemented in an incremental fashion: there is no requirement that all discharges be considered at the same time, thereby freeing the user of the nearly impossible task of directly devising a complex inductive argument over a set of ordinal variables. Instead individual discharge node and companion node pairs are incrementally added to the proof graph, and the local conditions under which the partial discharge was successful are recorded in the graph. When a new discharge is added to the proof graph, the set of related discharges are re–computed, to check whether a progressing measure can still be found. Henceforth we will refer to the process of adding a new discharge and companion node to the global discharge condition through this iterative procedure as an application of the *discharge rule*.

As a further refinement to the basic fixed point scheme the implementation employs a tagging mechanism [Win91] such that the state vector under which a fixed point is unfolded is recorded in the unfolded fixed point. For example in the proof rule $\mathsf{UNF1_L}$ (Defined on Page 86) the arguments $t_1 \ldots t_n$ are recorded together with the fixed point. Then, in a proof search, a fixed point is typically unfolded only when the current state vector is not an instance of an earlier (recorded) one.

The point of employing a tagging scheme is two-fold: first automatic proof search can be efficiently guided by the tags by providing hints when to stop unfolding a fixed point and in the discovery of an induction scheme. Second the scheme can also be used to improve the diagnostic result from an unsuccessful application of the local discharge rule:

- Is there an earlier proof state with a "control state" of which the current state is an instance? Suppose there is no such state in the fixed point tag, then likely the decision to discharge here is incorrect and proof search should continue.
- Is the current proof state an instance of an earlier one? If not, perhaps an additional argument about data is required.
- Does some ordinal decrease? Otherwise perhaps an inductive argument is missing from earlier proof steps.
- Can a progress ordinal variable be found, or do the derivations of discharge nodes from companion nodes interfere with each other? If this is the case then there is risk that the whole induction argument has to be reconsidered.

5.5 The Embedding of ERLANG

ERLANG program constructs are encoded in EVT in the manner described in Section 4.6, albeit with some minor differences.

EVT contains a definition of the transition relations as recursive predicates according to the principles discussed in Section 4.6.2. In addition, to improve the speed with which new transitions are computed, a set of low-level proof rules have been provided for inferring transitions $t \xrightarrow{\alpha} t'$ that trigger on the syntactic shape of an ERLANG expression t or ERLANG system t. Examples of such rules are, for the system transition case, found in Figure 4.8 on Page 113.

5.5.1 Tactics for Deriving Transitions

Suppose we are faced with a typical proof goal, $\Gamma \vdash s : [\alpha]\phi$, which requires us to find all the α derivatives of the system state s and check whether they satisfy the property ϕ. To find manually all the τ derivatives of a complex state s by applying the the operational semantics step-by-step results in quite lengthy derivations.

As an alternative EVT offers four high-level tactics, `diasem_l`, `diasem_r`, `boxsem_l` and `boxsem_r`, for reasoning about combinations of program terms and modalities. For example, the `diasem_r` and `boxsem_r` tactics tries to achieve the result of the rules $\langle\alpha\rangle_r$ and $[\alpha]_r$ below. In these schematic rules it is assumed that the set of α derivatives of t are enumerated by the states $t_1, \ldots, t_n$.

$$\langle\alpha\rangle_r \frac{\Gamma \vdash t_1 : \phi, \ldots, t_n : \phi, \Delta}{\Gamma \vdash s : \langle\alpha\rangle\phi, \Delta}$$

$$[\alpha]_r \frac{\Gamma \vdash t_1 : \phi, \Delta \quad \ldots \quad \Gamma \vdash t_n : \phi, \Delta}{\Gamma \vdash s : [\alpha]\phi, \Delta}$$

A measure of the complexity of the implementation of these two tactics is in order. In a sense they are both quite general problem solving tactics. However, they have domain specific knowledge about the ERLANG programming language such as knowledge of the rules in Figure 4.7 on Page 105 for reducing queue equations. It should be realised that the task these tactics attempt to automate is *much harder* than simply deriving the next program state given a concrete program. Often the program s below contains variables governed by a set of assumptions Γ:

$$\Gamma \vdash s : [\alpha]\phi$$

The assumptions may for instance contain information that two symbolic process identifiers are not equivalent, that a proof system variable is of a certain type, and even that a symbolic component of the program state satisfies some abstract behavioural description.

The `boxsem_r` tactic is rather simplistic. First a derived rule like $[\alpha]_{\mathsf{R}}$ on Page 82 is applied yielding a goal $\Gamma, t : t \xrightarrow{\alpha} T' \vdash T' : \phi, \Delta$. Then a simplification tactic is applied to the new transition assumption, and recursively to all freshly generated proof goals and new assumptions until no new proof goals are generated and all assumption positions have been considered. The simplification tactic essentially attempts to apply sequentially a set of tactics, until one succeeds. These are the low-level transition tactics, equational rewriting tactics, conjunction splitting tactics, introduction of universal quantifiers, etc.

The implementation of the `diasem_r` tactic is more involved. After a few reduction steps typically formulas of the following shape are encountered on the right-hand side of the resulting sequent:

$$\exists V_1 : S_1 \ldots \exists V_n : S_n. \left(t_1 \xrightarrow{\alpha_1} t_{1'} \wedge \ldots t_j \xrightarrow{\alpha_n} t_{j'} \wedge \phi_1 \wedge \ldots \wedge \phi_k \right)$$

where a formula ϕ_i does not contain instances of transition predicates. The difficulty here is knowing how many copies of the above formula that are required due to the presence of the existential quantifiers.

5.6 Evaluation of the Proof Assistant

A number of small to medium sized examples have been completed in the tool, some of which are reported in this thesis in Chapter 6. Apart from the purchasing agent study, the most challenging verification attempted with the proof assistant concerns a protocol for protocol for distributed query evaluation in distributed database lookup manager [AD99], which was verified using an earlier version of the tool.

Experiences of using the theorem prover are mixed. Correctness properties can be modelled, without major difficulties, in the program logic. The specification of properties that require fairness assumptions requires, as is often the case in approaches based on the modal μ-calculus, the embedding of fairness assumptions in the correctness properties themselves. Similarly assumptions about the privacy of process identifiers due to process spawning must normally be encoded in correctness properties since the treatment of privacy is rather weak in the present operational semantics.

With some effort proof arguments can be conducted at a reasonably high-level. Recent verification attempts have been automated to a large extent. These include the verification of a skeletal concurrent server program [FGN$^+$], a set of tactics for discovering programs that may cause runtime exceptions demonstrated at the 2000 Erlang User Conference, and recent work on the verification of a mutual exclusion protocol using model checking techniques (unpublished work by me). A conclusion of this work is that a substantial effort is required in building up a theory (formulas, lemmas, tactics, etc.) for the problem domain. Thus answering the question of how long a typical verification takes, and what is the effort, is rather difficult. If the program to be addressed fits a behavioural pattern previously encountered then verification can be quick. On the other hand, if new theory has to be developed, as can be the case for a program that internally employs some intricate data structure, then verification is a larger undertaking.

Some concrete figures about the current implementation are in order. EVT comprises about 15000 lines of Standard ML code, excluding the graphical user interface programmed in Java. The core tool was developed intensively over a period of 2 years (1998-1999) by Dilian Gurov and myself. After this mostly incremental changes have been made such as adding high-level tactics or providing support for reasoning about side-effect free code.

5.7 A Session with the Proof Assistant

This section demonstrates the interaction with EVT by considering a small example fetched from Section 6.2. The motivation for the proof will explained later, and a reader may want to look at Section 6.2 first. Here we will concentrate on the steps involved in realising the proof in EVT.

We shall not fully explain the meaning or even the syntax of the commands used to interact with EVT; rather this section provides an impression of how a session can proceed.

The proof challenge of the example is to show that list reversal, as expressed in

```
/* List append */
append:
 erlangProperList -> erlangProperList ->
 erlangProperList -> prop <=

 \U:erlangProperList.\V:erlangProperList.
 \W:erlangProperList.
 (((U=[]) /\ (V=W)) \/
  (exists X: erlangValue.
   exists U_prime: erlangProperList.
   exists W_prime: erlangProperList.
   ((U=[X | U_prime]) /\
    (W=[X | W_prime]) /\
    (append U_prime V W_prime))))
end;

/* List reversal */
rev:
 erlangProperList -> erlangProperList -> prop <=

 \U:erlangProperList. \V:erlangProperList.
 ((U=[] /\ V=[]) \/
  (exists Hd:erlangValue.
   exists Tl:erlangProperList. U=[Hd|Tl] /\
   (exists W:erlangProperList.
     (append W [Hd] V) /\ (rev Tl W))))
end
```

Table 5.1: List Reversal and List Concatenation in EVT

the specification logic, is a reversible operation. That is, if we apply the list reversal predicate to a list L yielding a result L', then using the same predicate L' can be reversed into L. The predicates expressing list reversal `rev` and list concatenation `append` are presented using the syntax of EVT in Table 5.7.

The proof session presented here is conducted using the command-line interface rather than the graphical user interface. Input that a user provides to the tool will be written in a bold typewriter font, and responses will be written in a normal typewriter font.

To begin the proof session EVT is started and a number of auxiliary tactics are read in from a file `support.sml`, and the list reversal and concatenation properties are read from a file `apprev.pde`.

```
> cevt
Erlang Verification Tool, command line interface,
version 00 Cervantes, by SICS/VeriCode.

Standard ML of New Jersey, Version 110.0.6,
October 31, 1999
val it = true :  bool
- use "support.sml";
...
- read_formula_defs "apprev.pde";
...
```

The most important tactics defined are `reduce_r` and `reduce_l` for performing obvious reduction steps such as equational reasoning.

The main proof challenge can be stated:

```
- prove "declare L:erlangProperList,M:erlangProperList
    in rev L M |- rev M L";
```

The proof will involve induction over the definition of `rev`. We commence by approximating the left-hand side definition and then unfold it and simplify the result by applying the `reduce_l` tactic using the `by` command. As a result there are two remaining goals:

```
- by(approx_r 1); to_goal 1;
- by(reduce_l 1); print_goals(); to_goal 1;
Goal #1:  {1} X_1<X_0 |- {1} rev [] []
Goal #2:  {1} append W [Hd] M, {2} rev(X_1) Tl W,
{3} X_1<X_0 |-  {1} rev M [Hd|Tl]
```

We can see that a fresh ordinal variable has been introduced in both proof goals which is assumed to be less than the ordinal introduced by the approximation operation; this is represented by the assumption `X_1<X_0`. The inequality results from the unfolding of the definition of `rev` on the left-hand side. Note also the ordinal variable `X_1` that annotates the second assumption in the second goal.

The first goal is trivially solved by the `reduce_r` tactic. To proceed with the second goal we note that it is an instance of the (approximated) original goal if only the formula `rev W Tl` occurred on the right hand side; we introduce it using the `cut` tactic:

```
- by(cut "rev W Tl"); print_goals(); to_goal 2;
Goal #1:  {1} append W [Hd] M, {2} rev(X_1) Tl W,
{3} X_1<X_0, {4} rev W Tl |- {1} rev M [Hd|Tl]
Goal #2:  {1} append W [Hd] M, {2} rev(X_1) Tl W,
{3} X_1<X_0 |- {1} rev M [Hd|Tl], {2} rev W Tl
```

Now the second goal is an instance of the first approximated proof node so we discharge this goal.

```
- discharge1();
...
Discharge succeeded.
```

One goal remains. By inspecting this goal a few interesting facts become obvious.

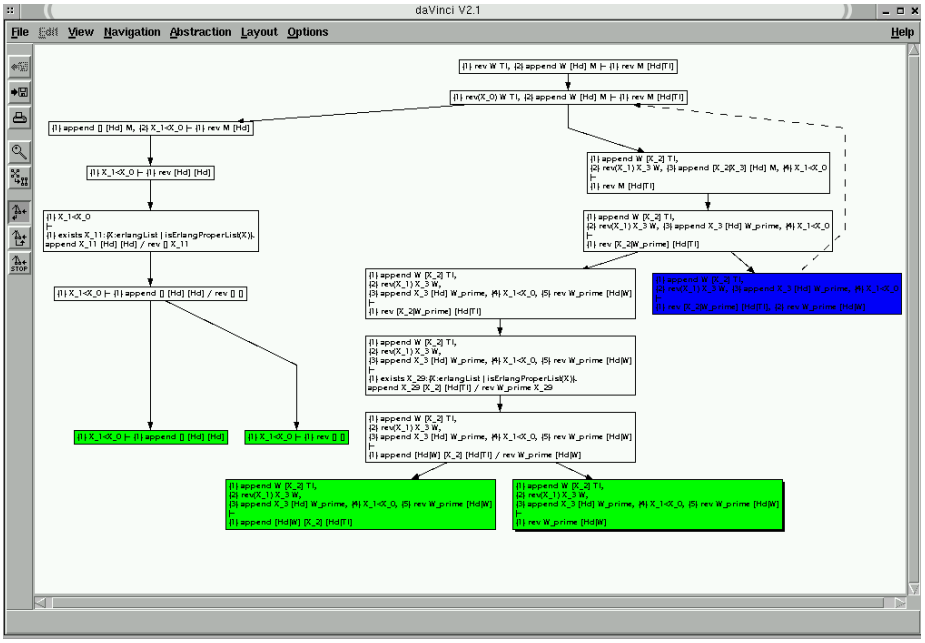

Figure 5.2: Proof of the append lemma

Clearly `W` is equal to `M` except that `M` has `Hd` at its tail. Second `W` is the reverse of `Tl`. But then `M` must indeed be the reverse of `[Hd|Tl]`, we establish this in a separate lemma:

```
- prove_lemma "appendrev" "declare M:erlangProperList,
W:erlangProperList, Hd:erlangValue, Tl:erlangProperList
in rev W Tl, append W [Hd] M |- rev M [Hd|Tl]";
...
```

The proof nodes of this lemma are indicated in Figure 5.7, which was automatically generated from a proof using the daVinci graph editor. The proof contains five leafs, one eliminated due to the application of discharge with respect to the second proof node, and the rest eliminated through local proof reasoning.

Now, it only remains to invoke the new lemma and check whether the proof is finished:

```
- lemmadischarge1 "appendrev";
Lemmadischarge succeeded.
- is_finished();
val it = true :  bool
```

Chapter 6

Examples

This chapter contains applications of the framework to examples of varying nature and complexity. The examples cover a wide range of applications and properties: from classical functional algorithms to security properties of involved process communication structures. The conclusion of these studies is that the framework can cope rather uniformly with all of these properties.

The first example in Section 6.2 demonstrates a simple proof of a property of a recursive data predicate, and shows how standard inductive proofs can be recast in our framework. The first code example in Section 6.3 considers side-effect free ERLANG programs using an implementation of the classical Quicksort algorithm in ERLANG. The example illustrates, in a simple setting, the interplay between code verification and data verification in the proof system.

After these minor examples two distributed applications or protocols are examined in detail in the Sections 6.4 and 6.5. These examples all have in common the fact that spawning of new processes is essential. In the first example a bag like structure of processes is built, while in the second example a list of processes is built. An alternative approach to the whole framework is considered in Section 6.6, where a leader election protocol for a ring network topology is verified in the framework of μCRL. Since the theory of μCRL has been embedded in the COQ theorem prover this presents a realistic alternative.

These proofs will by necessity be conducted on a relatively high level, since treating every minor proof rule would obscure interesting proof details. In particular the use of high-level tactics for handling the operational semantics, such as the ones discussed in Section 5.5.1, is crucial.

6.1 Patterns of Compositional Reasoning in our Framework

Here we will briefly summarise the role of compositional reasoning in our framework: how proofs reduce arguments about systems to arguments about their components. The

topic will be further demonstrated in the case studies that follow this section. A crucial decision is what kind of assumptions components make about the environment in which they operate. In the assumption-guarantee paradigm a component is typically characterised by (i) assumption on the environment, and (ii) a guarantee of service to the environment provided the assumptions are met. A classical problem in such decomposition schemes is how to handle recursive system structures: what if a component A depends on the component B which in turn depends on A?

In our case, the use of an assumption-guarantee paradigm is implicit in the formulas used to cut out the components of a composite term (usually the processes of a parallel composition). Consider for instance the verification of the billing agent in Section 6.4. A typical instance of an assumption-guarantee pair is the statement: "The purchasing agent communicates the credit card number to no process except the payment clearing center, unless some other process first sends it the number". This is represented by the formula *notransto* $Payment\ PCC$ given the definition

$$
\begin{array}{l}
\mathit{notransto} : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Rightarrow \\
\quad \lambda C : \mathsf{erlangValue}, PCC : \mathsf{erlangPid}. \\
\qquad [\tau]\, \mathit{notransto}\ C\ PCC \\
\quad \wedge \quad \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}. \\
\qquad\quad [P?V']\,(\mathit{contains}\ V'\ C \vee \mathit{notransto}\ C\ PCC) \\
\quad \wedge \quad \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}. \\
\qquad\quad [P!V']\,((P = PCC \vee \neg \mathit{contains}\ V'\ A) \wedge \mathit{notransto}\ C\ PCC)
\end{array}
$$

For the resource agent this assumption-guarantee pair a corresponding assumption-guarantee is made: "The resource manager does not communicate the credit card number, unless some other process first sends it the card number."

If the system under study were closed, these two assumptions would suffice to establish the fact that the credit card number would not be communicated (except to the payment clearing center). Since the system is open, the assumption that the credit card number is not received again is instead transferred to the formula to prove of the open system itself, and will decorate the input modalities.

Note also that the apparent circular reasoning (the purchasing agent relies on the resource manager which relies on the purchasing agent...) presents no difficulties in contrast with many other schemes for assumption-guarantee pairs. The key here is that the approach is semantics-based; it may be that assumptions and guarantees do not match, in which case little can be derived about the joint actions of two processes, but certainly no unsound conclusions can be drawn. Furthermore compared with synchronous schemes the asynchronous message passing scheme of ERLANG prevents circular reasoning. There is no way for a process to prevent messages from being sent to it.

$$
\begin{array}{l}
\mathit{rev} \Leftarrow \\
\quad \lambda L, L' : \mathsf{erlangProperList}. \\
\quad \left(\begin{array}{ll}
 & L = [\,] \wedge L' = [\,] \\
\vee & \exists H, T : \mathsf{erlangProperList}. L = [\,H \mid T\,] \wedge \\
 & \quad \exists L'' : \mathsf{erlangProperList}. \mathit{append}\ L''\ [\,H\,]\ L' \wedge \mathit{rev}\ T\ L''
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{append} \Leftarrow \\
\quad \lambda L_1, L_2, L_3 : \mathsf{erlangProperList}. \\
\quad \left(\begin{array}{ll}
 & L_1 = [\,] \wedge L_3 = L_2 \\
\vee & \exists V : \mathsf{erlangValue}, L_4, L_5 : \mathsf{erlangProperList}. \\
 & \quad L_1 = [V \mid L_4] \wedge \mathit{append}\ L_4\ L_2\ L_5 \wedge L_3 = [V \mid L_5]
\end{array} \right)
\end{array}
$$

Figure 6.1: Reversal and Append Predicates

6.2 A Simple Example Using Induction

This short section studies a simple predicate (fixed point definition) expressing reversal of ERLANG lists, and contains a proof that list reversal is indeed a reversible operation. The proof serves to illustrate, chiefly, how common inductive arguments can be recast in our framework while also providing a few additional predicate definition examples.

The ERLANG lists are normally understood as a sequence of "cons" cells $[\,H_1 \mid \ldots [\,H_n \mid T_n\,] \ldots]$, but without any guarantee that the tail T_n of the last cons cell is the empty list $[\,]$. In this example *proper* ERLANG lists are assumed, which are the lists where the tail is the empty list, and which are are characterised by the predicate *properList* below:

$$
\begin{array}{l}
\mathit{properList} \Leftarrow \\
\quad \lambda L : \mathsf{erlangValue}. L = [\,] \vee \exists H, T : \mathsf{erlangValue}. L = [\,H \mid T\,] \wedge \mathit{properList}\ T
\end{array}
$$

The subtype erlangProperList denotes the subtype of the ERLANG values which are proper lists (that satisfies the *properList* predicate).

In Figure 6.1 the list reversal predicate (*rev*) is defined, together with a list append predicate (*append*) referred to in its definition.

The initial proof statement (goal) is

$$\mathit{rev}\ L\ L' \vdash \mathit{rev}\ L'\ L \tag{6.1}$$

where L and L' are both proper ERLANG lists.

To prove such a proof statement normally involves referring to a structural induc-

tion proof scheme, analysing the list L:

$$\frac{\begin{array}{cc} & [\mathit{rev}\ T\ L'' \Rightarrow \mathit{rev}\ L''\ T] \\ \vdots & \vdots \\ \mathit{rev}\ [\,]\ L' \Rightarrow \mathit{rev}\ L'\ [\,] & \mathit{rev}\ [H \mid T]\ L' \Rightarrow \mathit{rev}\ L'\ [H \mid T] \end{array}}{\mathit{rev}\ L\ L' \Rightarrow \mathit{rev}\ L'\ L}$$

In our proof system such a scheme is mimicked by approximating and unfolding the *rev* predicate to the left, and introducing an analogue to the induction hypothesis to the left via a combination of CUT and DISCHARGE proof rules. An abstract sketch of the proof is:

$$\dfrac{\vdots \\ \vdash \mathit{rev}\ [\,]\ [\,] \qquad \dfrac{\dfrac{-}{\Gamma \vdash \mathit{rev}\ L''\ T, \Delta}\ \textsc{discharge} \qquad \begin{array}{c}\vdots\\ \Gamma, \mathit{rev}\ L''\ T \vdash \Delta\end{array}}{\mathit{append}\ L''\ [H]\ L', \mathit{rev}^{\kappa'}\ T\ L'', \kappa' < \kappa \vdash \mathit{rev}\ L'\ [H \mid T]}\ \textsc{cut}}{\dfrac{\mathit{rev}^{\kappa}\ L\ L' \vdash \mathit{rev}\ L'\ L}{\mathit{rev}\ L\ L' \vdash \mathit{rev}\ L'\ L}\ \textsc{approx}_{\mathsf{L}}}\ \textsc{unfold}_{\mathsf{L}},\ldots$$

The application of the DISCHARGE rule succeeds since there is a substitution $[L''/L', T/L]$ such that all assertions in the goal $\mathit{rev}^{\kappa}\ L\ L' \vdash \mathit{rev}\ L'\ L$ can be found under the substitution in the goal $\mathit{append}\ L''\ [H]\ L', \mathit{rev}^{\kappa'}\ T\ L'', \kappa' < \kappa \vdash \mathit{rev}\ L''\ T, \mathit{rev}\ L'\ [H \mid T]$. In addition a least fixed point definition *rev* to the left which was approximated with κ is now found approximated with κ' and the inequality $\kappa' < \kappa$ is provable.

The proof of the goal $\mathit{append}\ L''\ [H]\ L', \mathit{rev}^{\kappa'}\ T\ L'', \kappa' < \kappa, \mathit{rev}\ L''\ T \vdash \mathit{rev}\ L'\ [H \mid T]$ follows immediately from the lemma (not proven here) $\mathit{rev}\ L\ L', \mathit{append}\ L\ [H]$ L"$\vdash \mathit{rev}\ L''\ [H \mid L']$.

```
quick_sort([]) -> [];
quick_sort([Pivot|Rest]) ->
  {S, B} = split(Pivot, Rest),
  append(quick_sort(S), [Pivot|quick_sort(B)]).

split(Pivot, L) -> split1(Pivot, L, [], []).

split1(Pivot, [], S, B) -> {S, B};
split1(Pivot, [H|T], S, B) ->
  if
    H < Pivot -> split1(Pivot, T, [H|S], B);
    H >= Pivot -> split1(Pivot, T, S, [H|B])
  end.

append([H|L1],L2) -> [H|append(L1,L2)];
append([], L) -> L.
```

Figure 6.2: The Quick Sort Algorithm in ERLANG

6.3 The Quicksort Example

As a first gentle introduction to the verification of ERLANG code a side-effect free implementation of the classical Quicksort algorithm is considered. The example illustrates a number of key aspects of the proof system:

- Code (an implementation) is verified, not algorithms. Naturally a large part of the proof effort is required in establishing a link between the implementation and the logical characterisation of the algorithm.
- Proofs are compositional in the sense that key properties are proved of functions used in the main algorithm, which are later reusable.

This section reports on a joint verification effort with Gennady Chugunov which was partially reported in a seminar at the Nordic Workshop on Program Correctness in Autumn 1999.

The code for the implementation of the Quicksort algorithm is found in Figure 6.2. There are no major surprises. For clarity we choose to include an implementation of the `append` function rather than relying on the built-in function `++`.

The correctness properties of the example are the classical ones:

- **Partial correctness**: *If a call to the* `quick_sort` *function with an argument list L terminates it will return a sorted version of the list.*
- **Termination**: *Any call to the* `quick_sort` *function terminates.*

As a refinement of the termination condition the proof will additionally show that, under the assumption that the argument to the `quick_sort` function is well-formed

$$
\begin{array}{l}
\textit{sorted} \Leftarrow \\
\quad \lambda L : \mathsf{erlangProperList}. \\
\quad \left(\begin{array}{lll}
 & L = \texttt{[]} & \\
\vee & \exists H_1 : \mathsf{erlangValue}, T_1 : \mathsf{erlangProperList}. & \\
 & \quad L = [H_1 \mid T_1] & \\
 & \quad \wedge \left(\begin{array}{lll}
 & T_1 = \texttt{[]} & \\
\vee & \exists H_2 : \mathsf{erlangValue}, T_2 : \mathsf{erlangProperList}. & \\
 & \quad T_1 = [H_2 \mid T_2] \wedge \leq_{\text{ERLANG}} H_1\ H_2 \wedge \textit{sorted}\ T_1 &
\end{array} \right) &
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\textit{permutation} \Leftarrow \\
\quad \lambda L_1, L_2 : \mathsf{erlangProperList}. \\
\quad \left(\begin{array}{lll}
 & (L_1 = \texttt{[]} \wedge L_2 = \texttt{[]}) & \\
\vee & \exists H : \mathsf{erlangValue}, T_1, L_{21}, L_{22}, L_{23} : \mathsf{erlangProperList}. & \\
 & \quad L = [H \mid T_1] \wedge \textit{append}\ L_{21}\ [H \mid L_{22}]\ L_2 \wedge & \\
 & \quad \textit{permutation}\ T_1\ L_{23} \wedge \textit{append}\ L_{21}\ L_{22}\ L_{23} &
\end{array} \right)
\end{array}
$$

Figure 6.3: Sortedness and Permutation Predicates

(a proper list), the function call will terminate normally (not in an exception) and will produce no side-effects.

The formalisation of these correctness properties is relatively straightforward. For expressing proper termination the *fnormalizes* predicate on Page 110 is used. Secondly a returned list L' should (1) be a permutation of the original *permutation* $L\ L'$ (containing the same elements) and (2) be a sorted list *sorted* L. These properties are expressed in the program logic in Figure 6.3, together with a predicate *append* $L_1\ L_2\ L_3$ characterising the result of applying the `append` function. The definitions of these predicates are certainly not canonical, several alternatives are possible. In the definitions note the occurrence of declarations such as T : erlangProperList, where as in the previous section erlangProperList denotes the subtype of the ERLANG values (exact definition omitted), such that the tail part of a cons cell is again of type erlangProperList, rather than an arbitrary value as may be the case in ERLANG. This restriction of the value domain results in cleaner proofs since, for example, otherwise applications of the `quick_sort` function could raise exceptions, due to runtime typing errors.

6.3.1 A Proof Sketch

The proof sketch that follows will be kept informal since the main point of this example is to illustrate the interplay between program and data reasoning. For instance, the proof of goals solely relating to data will usually not be given.

The initial proof goal is

$$\begin{array}{l}\vdash \texttt{quick\_sort}(L) : \\ \quad \exists L' : \mathsf{erlangProperList}.\mathit{fnormalizes}\ L' \wedge \mathit{sorted}\ L' \wedge \mathit{permutation}\ L\ L'\end{array} \tag{6.2}$$

where L is of type erlangProperList. The predicate *fnormalizes* V was introduced on Page 110 for expressing strong normalisation of an expression to a value V. Proof goal 6.2 expresses that any call to `quick_sort` with a list parameter L must terminate with a resulting list L' which is sorted and a permutation of the original list. The proof proceeds by two applications of CUT, resulting in three goals:

$$\vdash \exists L' : \mathsf{erlangProperList}.\mathit{sorted}\ L' \wedge \mathit{permutation}\ L\ L' \tag{6.3}$$

$$\vdash \exists N : \mathsf{nat}.\mathit{length}\ L\ N\ \wedge \mathit{isNat}\ N \tag{6.4}$$

$$\begin{array}{l}\exists N : \mathsf{nat}.\mathit{length}\ L\ N \wedge \mathit{isNat}\ N, \\ \exists L' : \mathsf{erlangProperList}.\mathit{sorted}\ L' \wedge \mathit{permutation}\ L\ L' \vdash \\ \quad \texttt{quick\_sort}(L) : \exists L' : \mathsf{erlangProperList}. \\ \quad\quad \mathit{fnormalizes}\ L' \wedge \mathit{sorted}\ L' \wedge \mathit{permutation}\ L\ L'\end{array} \tag{6.5}$$

The predicate *length* $L\ N$ (shown in Figure 6.4) expresses that the length of the list L is the natural number N, and *isNat* characterises the natural numbers. Goals 6.3 and 6.4, henceforth referred to as Lemmas Q_1 and Q_2, which express mostly trivial facts about predicates will not be proved here.

The proof proceeds by eliminating the existential quantifiers in goal 6.5 to the left and right (using rules $\exists_L$,$\exists_R$), splitting conjunctions to the left and right (using rules $\wedge_L$,$\wedge_R$), eliminating resulting trivial facts (using rule ID), and approximating the least fixed point predicate *isNat* using rule APPROX_L, resulting in the sequent:

$$\begin{array}{l}\mathit{length}\ L\ N, \mathit{isNat}^{\kappa}\ N, \mathit{sorted}\ L', \mathit{permutation}\ L\ L' \vdash \\ \quad \texttt{quick\_sort}(L) : \mathit{fnormalizes}\ L'\end{array} \tag{6.6}$$

The basis for claiming termination of the algorithm will be that a sequent with a recursive call to $\texttt{quick\_sort}(L'')$ is encountered, such that the length of L'' is less than L. For the DISCHARGE rule to apply the second sequent must contain assumptions *length* $L''\ N'$ and $\mathit{isNat}^{\kappa'}\ N'$ such that $\kappa' < \kappa$.

After unfolding the definition of *length* in goal 6.6 the result is two new goals (after splitting the disjunction in *length* and unfolding *isNat*):

$$\begin{array}{l}\mathit{sorted}\ L', \mathit{permutation}(\texttt{[]}, L') \vdash \\ \quad \texttt{quick\_sort([])} : \mathit{fnormalizes}\ L'\end{array} \tag{6.7}$$

$$\begin{array}{l}\mathit{isNat}^{\kappa'}\ N', \kappa' < \kappa, \mathit{length}\ L''\ N', \mathit{sorted}\ L', \mathit{permutation}\ [V \mid L'']\ L' \vdash \\ \quad \texttt{quick\_sort([}V \mid L''\texttt{])} : \mathit{fnormalizes}\ L'\end{array} \tag{6.8}$$

The assumptions in goal 6.8 will henceforth be referred to as Γ. Goal 6.7 is easily handled since *permutation* `[]` L' requires that $L' = $ `[]` and then

$$
\begin{array}{l}
\mathit{length} \Leftarrow \\
\quad \lambda L : \mathsf{erlangProperList}, N : \mathsf{nat}. \\
\quad \left(\begin{array}{ll} & L = \texttt{[]} \wedge N = 0 \\ \vee & \exists V : \mathsf{erlangValue}, L' : \mathsf{erlangProperList}, N' : \mathsf{nat}. \\ & \quad L = [V \mid L'] \wedge N = N' + 1 \wedge \mathit{length}\ L'\ N' \end{array} \right)
\end{array}
$$

$$
\mathit{isNat} \Leftarrow \\
\quad \lambda N : \mathsf{nat}. N = 0 \vee \exists N' : \mathsf{nat}. N = N' + 1 \wedge \mathit{isNat}\ N'
$$

Figure 6.4: Definition of *length* and *isNat* Predicates

quick_sort([]) : *fnormalizes* [] is provable by a model checking tactic. For goal 6.8 we proceed by unfolding the definition of *fnormalizes* and applying the `quick_sort` function to its argument resulting in[1]

$$\Gamma \vdash \{\texttt{S,B}\} = \texttt{split}(V, L'')\texttt{,}\ \ldots : \mathit{fnormalizes}\ L' \tag{6.9}$$

At this point the subexpression `split(`V`,`L''`)` is analysed separately with the help of the TERMCUT proof rule, showing that it satisfies its specification

$$
\begin{array}{l}
\Gamma \vdash \texttt{split}(V, L'') : \\
\quad \exists S, B : \mathsf{erlangProperList}. \mathit{fnormalizes}\ \{S, B\} \wedge \mathit{split}\ V\ L''\ S\ B
\end{array} \tag{6.10}
$$

where *split* V L'' S B specifies that the list L'' is split into the two parts S and B by the pivot V such that S are all the elements less than V and B are the elements greater or equal to V (definition in Figure 6.5). The proof of goal 6.10 is omitted from this presentation, for reasons of space. The second new proof goal becomes (after eliminating the existential quantifier and the conjunction of the new assumption):

$$
\begin{array}{c}
\Gamma, X : \mathit{fnormalizes}\ \{S', B'\}, \mathit{split}\ V\ L''\ S'\ B' \vdash \\
\{\texttt{S,B}\} = X\texttt{,}\ \ldots : \mathit{fnormalizes}\ L'
\end{array} \tag{6.11}
$$

Proceeding with goal 6.11 we start by approximating and unfolding the left-hand side assumption X : *fnormalizes*$(\{S', B'\})$. There are two cases, either X represents a value $\{S', B'\}$ since the corresponding computation has terminated (goal 6.13), or there is a computation step $X \xrightarrow{\tau} X'$ for some X'. If there is a computation step then the result is, after unfolding *fnormalizes* on the right-hand side:

$$
\begin{array}{c}
\Gamma, \kappa' < \kappa, \quad X' : \mathit{fnormalizes}^{\kappa'}\ \{S', B'\}, \mathit{split}\ V\ L''\ S'\ B' \vdash \\
\{\texttt{S,B}\} = X'\texttt{,}\ \ldots : \mathit{fnormalizes}\ L'
\end{array} \tag{6.12}
$$

[1] recall that an assignment X `=` V`,` ... is an abbreviation of `case` V `of` X `->` ... `end`

$$
\begin{array}{l}
\mathit{split} \Leftarrow \\
\quad \lambda V : \mathsf{erlangValue}, L, S, B : \mathsf{erlangProperList}. \\
\left(\begin{array}{ll}
 & L = [\,] \wedge S = [\,] \wedge B = [\,] \\
\vee & \exists H : \mathsf{erlangValue}, T : \mathsf{erlangProperList}. L = [H \mid T] \wedge \\
 & \quad \left(\begin{array}{ll}
 & <_{\text{ERLANG}} H\ P \wedge \\
 & \quad \exists S' : \mathsf{erlangProperList}. \\
 & \quad\quad \mathit{append}\ S'\ [H]\ S \wedge \mathit{split}\ V\ T\ S'\ B \\
 & \geq_{\text{ERLANG}} H\ P \wedge \\
\vee & \quad \exists B' : \mathsf{erlangProperList}. \\
 & \quad\quad \mathit{append}\ B'\ [H]\ B \wedge \mathit{split}\ V\ T\ S\ B'
\end{array}\right)
\end{array}\right)
\end{array}
$$

Figure 6.5: The Definition of the *split* Predicate

Since the goal 6.12 is an instance of goal 6.11 and since a least fixed point has been unfolded to the left, this goal can be discharged. The termination case

$$
\begin{array}{l}
\Gamma, \kappa' < \kappa, \quad X : \mathit{the\_term}\ \{S', B'\}, \mathit{split}\ V\ L''\ S'\ B' \vdash \\
\qquad \{\mathtt{S},\mathtt{B}\}\ =\ X,\ \ldots : \mathit{fnormalizes}\ L'
\end{array}
\tag{6.13}
$$

becomes after applying the derived proof rule THE_TERM, and weakening out the ordinal inequation,

$$
\Gamma, \mathit{split}\ V\ L''\ S'\ B' \vdash \{\mathtt{S},\mathtt{B}\}\ =\ \{S', B'\},\ \ldots : \mathit{fnormalizes}\ L' \tag{6.14}
$$

This level of reasoning is unfortunately very tedious, it is at this point that the decision to adopt a small-step operational semantics comes back to haunt us. To recover some of the convenience of a natural semantics derived proof rules are used. For instance, the EVAL rule below is derivable (assuming e does not occur in Γ or Δ) as seen in Example 17 below, and can be used to immediately obtain goal 6.14 from goal 6.11.

$$
\text{EVAL}\frac{\Gamma \vdash r[v_1] : \mathit{fnormalizes}\ v_2, \Delta}{\Gamma, e : \mathit{fnormalizes}\ v_1 \vdash r[e] : \mathit{fnormalizes}\ v_2, \Delta}
$$

where as in Definition 16 on Page 48, $r[e]$ represents an ERLANG expression with the subexpression e in a computation enabled position. In Gurov and Chugunov [GC00] the side-effect free fragment of ERLANG is studied in further detail. Note in particular that the above definition depends crucially on the fact that e occurs in a reduction context, due to the peculiar variable binding conventions of ERLANG (compare the definition of expression congruences on Page 30).

Example 17 (Derivation of EVAL proof rule)**.** By approximating the definition of *fnormalizes* on the left-hand side of the initial proof goal, the goal below is reached

$$
\Gamma, e : \mathit{fnormalizes}^{\kappa}\ v_1 \vdash r[e] : \mathit{fnormalizes}\ v_2, \Delta \tag{6.15}
$$

First the left-hand side definition of *fnormalizes* is unfolded, and conjunctions and disjunctions are split, yielding two goals:

$$\begin{array}{l}\Gamma, e : \forall A : \mathsf{erlangExprAction}.[A]\left(A = \tau \wedge (\mathit{fnormalizes}^{\kappa'}\ v_1)\right), \\ \quad \kappa' < \kappa, e : \langle\tau\rangle \mathit{true} \vdash r[e] : \mathit{fnormalizes}\ v_2, \Delta\end{array} \tag{6.16}$$

$$\begin{array}{l}\Gamma, e : \forall A : \mathsf{erlangExprAction}.[A]\left(A = \tau \wedge (\mathit{fnormalizes}^{\kappa'}\ v_1)\right), \\ \quad \kappa' < \kappa, e : \mathit{the\_term}\ v_1 \vdash r[e] : \mathit{fnormalizes}\ v_2, \Delta\end{array} \tag{6.17}$$

By applying the THE_TERM proof rule to goal 6.17, and weakening out the first assumption we immediately get the desired premise:

$$\Gamma \vdash r[v_1] : \mathit{fnormalizes}\ v_2, \Delta \tag{6.18}$$

Continuing with goal 6.16 the right-hand side assertion is unfolded, conjunctions and disjunctions are split, and the right-hand side assertion on *the_term* v_2 is weakened out, producing two goals:

$$\begin{array}{l}\Gamma, e : \forall A : \mathsf{erlangExprAction}.[A]\left(A = \tau \wedge (\mathit{fnormalizes}^{\kappa'}\ v_1)\right), \\ \quad \kappa' < \kappa, e : \langle\tau\rangle \mathit{true} \vdash \\ \qquad r[e] : \forall A : \mathsf{erlangExprAction}.[A]\left(A = \tau \wedge (\mathit{fnormalizes}\ v_2)\right), \Delta\end{array} \tag{6.19}$$

$$\begin{array}{l}\Gamma, e : \forall A : \mathsf{erlangExprAction}.[A]\left(A = \tau \wedge (\mathit{fnormalizes}^{\kappa'}\ v_1)\right), \\ \quad \kappa' < \kappa, e : \langle\tau\rangle \mathit{true} \vdash \\ \qquad r[e] : \langle\tau\rangle \mathit{true}, \Delta\end{array} \tag{6.20}$$

Goal 6.20 is solvable using proof rule CONTEXT$\langle\rangle$ defined on Page 112 and REFL (recall that *true* abbreviates an equality). For dealing with goal 6.19 first the universal quantifiers are stripped, then the rule CONTEXT[] is applied. After some additional simple reasoning the goal below is reached

$$\Gamma, e : \mathit{fnormalizes}^{\kappa'}\ v_1, \kappa' < \kappa \vdash r[e] : \mathit{fnormalizes}\ v_2, \Delta \tag{6.21}$$

which can be discharged against goal 6.15.

Continuing with goal 6.14 of the quick sort example by unfolding *fnormalizes*, and performing the assignment, results in the proof goal

$$\begin{array}{l}\Gamma, \mathit{split}\ V\ L''\ S'\ B' \vdash \\ \quad \texttt{append(quick\_sort(}S'\texttt{), [}V\texttt{|quick\_sort(}B'\texttt{)])} : \mathit{fnormalizes}\ L'\end{array} \tag{6.22}$$

where $\Gamma' = \Gamma, \mathit{split}\ V\ L''\ S'\ B'$. Here the proof rules CUT, $\exists_{\mathsf{L}}$ and then EVAL, is applied

twice on the calls to `quick_sort` resulting in the goals

$$\Gamma' \vdash \texttt{quick\_sort}(S') : \exists L : \mathsf{erlangProperList}.\mathit{fnormalizes}\ L \wedge \mathit{sorted}\ L \wedge \mathit{permutation}\ S'\ L \tag{6.23}$$

$$\Gamma' \vdash \texttt{quick\_sort}(B') : \exists L : \mathsf{erlangProperList}.\mathit{fnormalizes}\ L \wedge \mathit{sorted}\ L \wedge \mathit{permutation}\ B'\ L \tag{6.24}$$

$$\Gamma', \mathit{permutation}\ [V \mid L'']\ L', \mathit{split}\ V\ L''\ S'\ B', \mathit{sorted}\ S'', \mathit{permutation}\ S'\ S'', \mathit{sorted}\ B'', \mathit{permutation}\ B'\ B'' \vdash \texttt{append}(S'', [V \mid B'']) : \mathit{fnormalizes}\ L' \tag{6.25}$$

Goals 6.23 and 6.24 represents, in a sense, the induction hypothesis. As an example (they are symmetrical) goal 6.23 is handled. First, from lemmas Q_1 and Q_2 follows, analogously to the treatment of goal 6.6:

$$\Gamma', \mathit{length}\ S\ N'', \mathit{isNat}\ N'', \mathit{sorted}\ S'', \mathit{permutation}\ S'\ S'' \vdash \texttt{quick\_sort}(S) : \mathit{fnormalizes}\ S'' \tag{6.26}$$

Using the DISCHARGE proof rule it is permissible to terminate such a proof branch if a number of conditions can be satisfied:

- A substitution can be found from the assertions in an earlier goal to goal 6.26, such that all assertions are covered.
- A least fixed point has unfolded on the left hand side along the path from the original goal to the present goal.

Clearly the candidate discharge goal is 6.6. Unfortunately there is a complication, in that the assumption $\mathit{isNat}^{\kappa}\ N$ cannot be mapped to any assumption in 6.26. The only approximated formula is $\mathit{isNat}^{\kappa'}\ N'$ in Γ but N must clearly be mapped to N'' instead. So after application of CUT the new proof obligation is

$$\Gamma', \mathit{length}(S'', N''), \mathit{isNat}\ N'', \mathit{sorted}\ S'', \mathit{permutation}\ S'\ S'' \vdash \mathit{isNat}^{\kappa'}(N'') \tag{6.27}$$

This goal is proven using two new lemmas, Q_3 and Q_4. Lemma Q_3 is an observation about the length of lists returned by the `split` function, and about permutations:

$$\mathit{length}\ L\ N, \mathit{split}\ V\ L\ S\ B, \mathit{permutation}\ S\ S' \vdash \exists N' : \mathsf{nat}.\mathit{length}\ S'\ N' \wedge N' \leq_{\mathsf{nat}} N$$

Lemma Q_4 states a basic fact about natural numbers and ordinal approximations:

$$\mathit{isNat}^{\kappa}\ N, N' \leq_{\mathsf{nat}} N \vdash \mathit{isNat}^{\kappa}\ N'$$

Given these lemmas and the assumptions in Γ' goal 6.29 can be proved. The discharge case remains, in goal 6.28:

$$\Gamma', \mathit{length}\ S\ N'', \mathit{isNat}\ N'', \mathit{sorted}\ S'', \mathit{permutation}\ S'\ S'', \mathit{isNat}^{\kappa'}\ N'' \vdash \texttt{quick\_sort(}S\texttt{)} : \mathit{fnormalizes}\ S'' \tag{6.28}$$

which now can be immediately discharged against goal 6.6.

It remains to prove goal 6.25. Here again a CUT and EVAL combination is applied resulting in the goals

$$\Gamma' \vdash \texttt{append(}S''\texttt{,[}V\texttt{|}B''\texttt{])} : \exists L : \mathsf{erlangProperList}.\mathit{fnormalizes}\ L \wedge \mathit{append}\ S''\ [V|B'']\ L_3 \tag{6.29}$$

$$\Gamma', \mathit{sorted}\ S'', \mathit{permutation}\ S'\ S'', \mathit{sorted}\ B'', \mathit{permutation}\ B'\ B'', \mathit{append}\ S''\ [V|B'']\ L_3 \vdash L_3 : \mathit{fnormalizes}\ L' \tag{6.30}$$

The proof of goal 6.29 is omitted from this presentation. To solve goal 6.30 first the definition of *fnormalizes* is unfolded, and then the rule THE_TERM is applied resulting in a goal

$$\begin{array}{l}\mathit{isNat}^{\kappa'}\ N', \kappa' < \kappa, \mathit{length}\ L''\ N', \mathit{sorted}\ L',\\ \mathit{permutation}\ [V \mid L'']\ L', \mathit{split}\ V\ L''\ S'\ B',\\ \mathit{sorted}\ S'', \mathit{permutation}\ S'\ S'', \mathit{sorted}\ B'', \mathit{permutation}\ B'\ B'',\\ \mathit{append}\ S''\ [V|B'']\ L_3 \vdash L' = L_3\end{array} \tag{6.31}$$

The exact steps needed to establish $L' = L_3$ will not be discussed here. Intuitively, however, from the definition of *split* follows that $\mathit{append}\ S'\ [V \mid B']\ L_4$, for any list L_4 must be a permutation of the original list $[V \mid L'']$. Since the permutation property is, naturally, preserved by *permutation* then also L_3 from $\mathit{append}\ S''\ [V \mid B'']\ L_3$ must be a permutation of the original list. Similarly *split* $V\ L''\ S'\ B'$ ensures that the pivot V lies between S' and B'. Now since S'' (and B'') is a sorted permutation of S' (and B'') then it follows that the list L_3 must also be sorted. Finally, from the Lemma Q_5

$$\begin{array}{l}\vdash \forall L_1, L_2, L_3 : \mathsf{erlangProperList}.\\ \quad \mathit{permutation}\ L_1\ L_2 \wedge \mathit{permutation}\ L_1\ L_3 \wedge\\ \quad \mathit{sorted}\ L_2\ \wedge \mathit{sorted}\ L_3 \Rightarrow L_2 = L_3\end{array}$$

follows that the lists L' and L_3 must be identical.

6.4 A Purchasing Agent

This example illustrates how typical client-server and agent applications can be verified. The key proof techniques used in the study is proof decomposition. When a new process is spawned it is abstracted by means of a formula characterising its behaviour, and it is shown that the abstraction, together with an abstraction of the spawning process, satisfies the original formula.

The example is based on the following scenario: a user wants repeated access to a resource and in return offers to pay for the service, e.g., using a credit card. A request is therefore sent to the manager of the resource which responds by creating a new agent process to act as an intermediary between the user, the resource, and a clearing centre for the user's payment. We view this scenario as quite typical of many security-critical agent applications.

The user is clearly taking a risk by exposing its confidential data (credit card number) to the resource manager and the purchasing agent. One of these parties might violate the trust put in them e.g. by charging for services not provided, or by passing information that should be kept confidential to third parties. Equally the resource manager need to trust the purchasing agent (and to some minor extent the user).

The system considered is open, there are no arbitrary limits placed on the number of processes involved, or the number of service requests received, to facilitate verification.

We show how the above scenario can be modelled in Erlang and how critical properties can be expressed in the program logic, and outline a proof of the desirable property of the purchasing agent that the number of charging requests to the payment clearing centre do not exceed the number of requests by the user for the resource.

Finally a remark to the reader is in order. This example was originally stated and verified using a previous version of the ERLANG verification tool, with a different syntax for ERLANG, and a different syntax for the logic than the one presently used.

6.4.1 Implementation as an Erlang Program

The function `rm` in Figure 6.6 implements the resource management scheme. The parameters to the function are a resource list, the process identifier of a trusted payment clearing centre (e.g., the credit card company), and a well-known identification of itself `RMname`. The resource manager implements a map from public to private resource "names" to prevent unauthorised access to resources; 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 credit card number), searches the resource list for the requested resource. If the resource is found, a purchasing agent is spawned to mediate between the user, the payment clearing centre (pcc) and the resource, and the name (i.e. pid) of the purchasing agent is made known to the user. Figure 6.7 shows the system configuration before and after the creation of the purchasing agent.

The purchasing agent coordinates accesses to the resource with charging requests to the payment clearing centre. Upon receiving a request for the resource $\{use, UserPid\}$, it attempts to acquire the resource, and if this succeeds (resulting in a response *Value*

```
rm(ResList,PayAgency,RMname) ->
  receive
    {contract,{Pu,Payment},From} ->
      case lookup(Pu,ResList) of
        {ok,Pr} ->
          From!{contract_ok,
                spawn(agent,
                      [Pr,PayAgency,
                       RMname,Payment])};
        nok ->
          From!contract_nok
      end
  end,
  rm(ResList,PayAgency,RMname).

agent(Res,PayAgency,RMname,Payment) ->
  receive
    {use,From} ->
        Res!{acquire,self()},
        receive
          {acquire_ok,Value} ->
              PayAgency!{trans,
                         {Payment,RMname},
                         self()},
              receive
                {trans_ok,{Payment,RMname}} ->
                  From!{use_ok,Value};
                {trans_nok,{Payment,RMname}} ->
                  From!use_nok
              end;
          acquire_nok -> From!use_nok
        end
  end,
  agent(Res,PayAgency,RMname,Payment).
```

Figure 6.6: Source Code for Agent Example

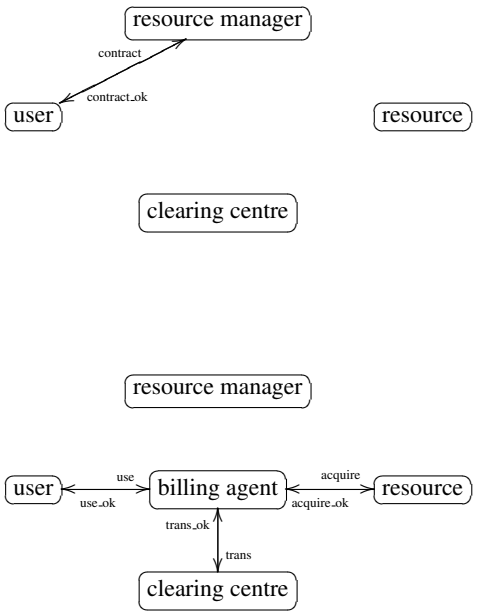

Figure 6.7: The system configuration: (above) before, and (below) after spawning the purchasing agent

offering access to the resource), it attempts to charge for the resource, transferring money from the credit card to the resource manager `RMname`. If the payment is successful finally the response from resource is sent back to the user.

6.4.2 Property Specification

Some desired properties of the purchasing agent system are discussed and formalised in the logic.

Disallowing Non-Approved Charging

The first correctness requirement considered forbids non-approved charging of the users credit card by requiring that the number of payment requests to the payment clearing centre should be less than or equal to the number of requests by the user for the resource. The size of the payment is not modelled in this verification.

$$
\begin{array}{l}
\mathit{safe} : \mathsf{erlangSystem} \to \mathsf{prop} \Rightarrow \\
\quad \lambda \mathit{Ag}, \mathit{PCC} : \mathsf{erlangPid}, \mathit{Payment}, N : \mathsf{nat}. \\
\qquad [\tau](\mathit{safe}\ \mathit{Ag}\ \mathit{PCC}\ \mathit{Payment}\ N) \\
\quad \wedge \quad \forall P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P?V] \\
\qquad \left(\begin{array}{ll} & (\mathit{isuse}\ P\ V\ \mathit{Ag}) \wedge (\mathit{safe}\ \mathit{Ag}\ \mathit{PCC}\ \mathit{Payment}\ N{+}1) \\ \vee & \neg(\mathit{isuse}\ P\ V\ \mathit{Ag}) \wedge (\mathit{safe}\ \mathit{Ag}\ \mathit{PCC}\ \mathit{Payment}\ N) \\ \vee & \mathit{contains}\ V\ \mathit{Payment} \end{array} \right) \\
\quad \wedge \quad \forall P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V] \\
\qquad \left(\begin{array}{ll} & (\mathit{istrans}\ P\ V\ \mathit{PCC}\ \mathit{Payment}) \wedge \\ & \qquad \exists N' : \mathsf{nat}. N = N'{+}1 \wedge (\mathit{safe}\ \mathit{Ag}\ \mathit{PCC}\ \mathit{Payment}\ N') \\ \vee & \neg(\mathit{istrans}\ P\ V\ \mathit{PCC}\ \mathit{Payment}) \wedge (\mathit{safe}\ \mathit{Ag}\ \mathit{PCC}\ \mathit{Payment}\ N) \end{array} \right)
\end{array}
$$

The predicates isuse and $\mathit{istrans}$ recognise resource requests and money transfers:

$$
\begin{array}{l}
\mathit{isuse} \triangleq \\
\qquad \lambda P : \mathsf{erlangPid}, V : \mathsf{erlangValue}, \mathit{Ag} : \mathsf{erlangPid}. \\
\qquad\quad P = \mathit{Ag} \wedge \exists \mathit{Pid} : \mathsf{erlangPid}. V = \{\mathtt{use}, \mathit{Pid}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{istrans} \triangleq \\
\qquad \lambda P : \mathsf{erlangPid}, V : \mathsf{erlangValue}, \mathit{PCC} : \mathsf{erlangPid}, \mathit{Payment} : \mathsf{nat}. \\
\qquad\quad \left(\begin{array}{ll} & P = \mathit{PCC} \\ \wedge & \exists \mathit{Pid} : \mathsf{erlangPid}, \mathit{Acc} : \mathsf{nat}. \\ & \qquad V = \{\mathtt{trans}, \{\mathit{Payment}, \mathit{Acc}\}, \mathit{Pid}\} \end{array} \right)
\end{array}
$$

The predicate $\mathit{contains}\ v\ v'$ 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 $\mathit{contains}\ v_1\ v'$ or $\mathit{contains}\ v_2\ v'$). We omit the easy definition.

So, a purchasing agent with pid Ag, pid of a payment clearing centre PCC, and credit card number $\mathit{Payment}$ is defined to be *safe* if the difference N between the number of requests for using the resource (messages of type $\{\mathit{use}, \mathit{Pid}\}$ received in the process mailbox) and the number of attempts for transfers from the credit card (messages of type $\{\mathtt{trans}, \{\mathit{Payment}, \mathit{Acc}\}, \mathit{pid}\}$ sent to PCC) is always non-negative. Since this difference is initially equal to zero, we expect $\mathtt{agent}(\mathit{ResPid}, \mathit{PCC}, \mathit{RMname}, \mathit{Payment})$ to satisfy $\mathit{safe}\ \mathit{Ag}\ \mathit{PCC}\ \mathit{Payment}\ 0$.

Note in the definition of safe that if a new request is received with the same credit card number $\mathit{Payment}$ is received (clause $\mathit{contains}$ V $\mathit{Payment}$), then the property is trivially true. This clause represents the simplifying assumption that no concurrent resource requests with the same credit card number is ever made to the resource manager.

Expected Service is Received

Other interesting properties are that the user receives the proper answer upon a successful resource request. 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 purchasing agent does not make illegal withdrawals nothing stops the resource manager from communicating the credit card number to a third party, which can then use the credit card number in non-approved ways.

Thus we need to prove that the system communicates neither the user credit card number nor the agent process identifier. An example of such a property specification, e.g. that the system does not communicate the user credit card number, is specified by the formula $notrans\ Payment$ given the definition below.

$$
\begin{array}{l}
notrans : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Rightarrow \\
\quad \lambda V : \mathsf{erlangValue}. \\
\qquad [\tau]\ notrans\ V \\
\quad \wedge \quad \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}.[P?V']\,(contains\ V'\ V \vee notrans\ V) \\
\quad \wedge \quad \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}.[P!V']\,(\neg contains\ V'\ V \wedge notrans\ V)
\end{array}
$$

6.4.3 Verification

Here we will demonstrate that the resource manager satisfies the *safe* specification. 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 present 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 purchasing agent and a credit card number the formula must be preceded by an initialisation phase (notice the use of the weak modality $[[\alpha]]$ introduced on Page 27):

$$
\begin{array}{c}
\forall PubRes, Payment : \mathsf{nat}, UserPid, Agent : \mathsf{erlangPid}. \\
[R_P?\{contract, \{PubRes, Payment\}, UserPid\}] \\
[[UserPid!\{\texttt{contract\_ok}, Agent\}]] \\
(safe\ Agent\ PCC\ Payment\ 0)
\end{array}
$$

So we set out to prove the following sequent:

$$\begin{array}{lll} \Gamma & \vdash & \langle \mathit{rm}(R_L, PCC, RMname), R_P, \epsilon\rangle \\ & & : \forall PubRes, Payment, UserPid, \mathit{Agent}. \\ & & \quad [R_P?\{\mathit{contract}, \{PubRes, Payment\}, UserPid\}] \ldots \end{array} \tag{6.32}$$

The necessary inequations on process identifiers (e.g., $R_P \neq P_r$) are collected in Γ. By application of simple proof steps, as realised in the tool by a "model checking" tactic which is programmed to stop as soon as a proof sequent with $safe$ as the main formula is seen, a number of essentially identical proof states results. A typical such state is

$$\begin{array}{lll} \Gamma' & \vdash & \langle \mathit{rm}(R_L, PCC, RMname), R_P, \epsilon\rangle \\ & & \quad || \langle \mathit{agent}(P_r, PCC, RMname, Payment), P_P, \epsilon\rangle \\ & & \quad : (\mathit{safe}\ P_P\ PCC\ Payment\ 0) \end{array} \tag{6.33}$$

where Γ' is Γ extended with the fact that P_P is a fresh process identifier. The other proof states correspond to different program points, relating to the number of ways how the weak modality $[[UserPid!\{\texttt{contract\_ok}, Agent\}]]\phi$ can be resolved. Such proof states become, after a few program execution steps, instances of the goal 6.33 and can thus be discharged against it.

Goal 6.33 is a critical proof state, where we must come up with properties of the resource manager and the purchasing agent, that are sufficiently strong to prove that their parallel composition satisfies the $safe$ property. The alternative would be to simply continue with a model checking like tactic, clearly this would not be successful since the capacity for process spawning remains possible.

In general such a decomposition proof step may be very difficult, but here the choice is relatively simple, and equally important, to a large extent independent of the actual programs being verified:

- The purchasing agent satisfies the $safe$ property, i.e.,

$$\mathit{safe}\ P_P\ PCC\ Payment\ 0$$

- The purchasing agent communicates the credit card number to no process except the payment clearing center, unless some other process first sends it the number, which is represented by the formula *notransto* $Payment\ PCC$ given the definition

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{notransto} : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Rightarrow} \\ & & \lambda C : \mathsf{erlangValue}, PCC : \mathsf{erlangPid}. \\ & & \quad [\tau]\, \mathit{notransto}\ C\ PCC \\ & \wedge & \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}. \\ & & \quad [P?V']\, (\mathit{contains}\ V'\ C \vee \mathit{notransto}\ C\ PCC) \\ & \wedge & \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}. \\ & & \quad [P!V']\, ((P = PCC \vee \neg \mathit{contains}\ V'\ A) \wedge \mathit{notransto}\ C\ PCC) \end{array}$$

- The resource manager does not communicate the credit card number, unless some other process first sends it the card number. This property can be formulated as $notrans\ Payment$ given the definition of $notrans$ on Page 147.

- The resource manager does not send a tuple containing the atom *use* in the first position (a usage request to a purchasing agent), which is described by $nouse$ below:

$$
\begin{array}{lll}
nouse : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Rightarrow & & \\
\quad [\tau]\ nouse & & \\
\wedge & \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}.[P?V']nouse & \\
\wedge & \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}. & \\
\quad [P!V']\,(\neg \exists V' : \mathsf{erlangValue}.V = \{\mathtt{use}, V'\} \wedge nouse) & &
\end{array}
$$

- The resource manager cannot receive messages sent to the clearing centre process, nor can it receive messages sent to the purchasing agent. This is described by the properties *norecv* PCC and *norecv* P_P given

$$
\begin{array}{lll}
\mathit{norecv} : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Rightarrow & & \\
\quad \lambda P' : \mathsf{erlangPid}. & & \\
\quad\quad [\tau]\ \mathit{norecv}\ P' & & \\
\wedge & \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}.[P?V'](\neg P = P' \wedge \mathit{norecv}\ P') & \\
\wedge & \forall P : \mathsf{erlangPid}, V' : \mathsf{erlangValue}.[P!V']\mathit{norecv}\ P' &
\end{array}
$$

Essentially these conditions guarantee that charges to the credit card are the result of user requests, rather than incorrectly programmed or malicious purchasing agents or resource managers that exchange information with each other.

This specification is a result of an iterative approach where first an initial sketch of such a decomposition property was attempted, and shown to be too weak (the composition of the properties is not sufficient to prove the desired property) or less frequently too strong (the decomposition property is not provable of the decomposed processes). After strengthening the property a second attempt at proof was attempted, and so on, until finally (i) both the individual processes satisfy the cut properties, and (ii) the cut properties together ensure the desired end property $safe$. In this proof exposition, however, only the end result of the proof process is shown.

Before decomposing the program the proof sequent is first generalised, by proving the property $\neg contains\ P_Q\ Payment$ about the resource manager and the agent input queues, and by showing that the number of resource requests in the agent queue (currently zero) is less than or equal to the last parameter of $safe$ property (the number of permitted charge requests, currently also zero). The result of applying the $\mathsf{TERMCUT_L}$ rule twice, after generalising the proof goals in this manner, is then the following proof

obligations:

$$\begin{array}{l}\Gamma', \neg \mathit{contains}\ P_Q\ \mathit{Payment}, \mathit{countuse}\ P_Q\ M, M \leq_{\mathsf{nat}} N \vdash \\ \quad \langle \mathit{agent}(P_r, PCC, RMname, \mathit{Payment}), P_P, P_Q \rangle \\ \quad : \mathit{safe}\ P_P\ PCC\ \mathit{Payment}\ N \wedge \mathit{notransto}\ \mathit{Payment}\ PCC\end{array} \tag{6.34}$$

$$\begin{array}{l}\Gamma', \neg \mathit{contains}\ R_Q\ \mathit{Payment} \vdash \\ \quad \langle \mathit{rm}(R_L, PCC, RMname), R_P, R_Q \rangle \\ \quad : \mathit{notrans}\ \mathit{Payment} \wedge \mathit{nouse} \wedge \mathit{norecv}\ PCC \wedge \mathit{norecv}\ P_P\end{array} \tag{6.35}$$

$$\begin{array}{l}\Gamma', S_1 : \mathit{safe}\ P_P\ PCC\ \mathit{Payment}\ N, S_1 : \mathit{notransto}\ \mathit{Payment}\ PCC, \\ \quad S_2 : \mathit{notrans}\ \mathit{Payment}, S_2 : \mathit{nouse}, S_2 : \mathit{norecv}\ PCC, S_2 : \mathit{norecv}\ P_P \\ \quad \vdash S_1 || S_2 : \mathit{safe}\ P_P\ PCC\ \mathit{Payment}\ N\end{array} \tag{6.36}$$

To prove the leftmost conjunct in the goal (6.34) it is necessary to show that the number of valid usage requests in the input queue (the parameter M in $\mathit{countuse}\ P_Q\ M$) 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. Similarly the proof that the billing agent satisfies $\mathit{notransto}\ \mathit{Payment}\ PCC$ is rather straightforward (both proofs omitted).

Instead we focus on the leftmost conjunct of (6.35), i.e., that the resource manager process satisfies $\mathit{notrans}\ \mathit{Payment}$ as long as no element in its input queue contains $\mathit{Payment}$. The proofs of the other conjuncts follow the same pattern so details are omitted from here. To prove (6.35) first unfold the definition of $\mathit{notrans}$ and split the conjunctions. There are three cases to consider: an input step, an output step or an internal step. In the case of an input step $[V?V']$ one possibility is that the property holds trivially (if $\mathit{contains}\ V'\ \mathit{Payment}$). Otherwise the resulting proof state is

$$\begin{array}{l}\Gamma', \neg \mathit{contains}\ R_Q\ \mathit{Payment}, \neg \mathit{contains}\ V'\ \mathit{Payment}\ \vdash \\ \quad \langle \mathit{rm}(R_L, PCC, RMname), R_P, R_Q \cdot V' \rangle : \mathit{notrans}\ \mathit{Payment}\end{array} \tag{6.37}$$

which can be rewritten into (by referring to the definition of $\mathit{contains}$)

$$\begin{array}{l}\Gamma', \neg \mathit{contains}\ R_Q \cdot V'\ \mathit{Payment}\ \vdash \\ \quad \langle \mathit{rm}(R_L, PCC, RMname), R_P, R_Q \cdot V' \rangle : \mathit{notrans}\ \mathit{Payment}\end{array} \tag{6.38}$$

which can be discharged against the leftmost conjunct of (6.35). For the output step it is clear that the resource manager process cannot perform such a step so that proof branch is trivially true. Thus only the internal step remains, and such a step must correspond to a function application of $\mathit{rm}(R_L, PCC, RMname)$. The resulting proof state is:

$$\begin{array}{l}\Gamma', \neg \mathit{contains}\ R_Q\ \mathit{Payment}\ \vdash \\ \quad \langle \texttt{receive} \ldots \texttt{end}, R_P, R_Q \rangle : \mathit{notrans}\ \mathit{Payment}\end{array} \tag{6.39}$$

By repeating the above steps, i.e., handling input, output and internal steps eventually a new process is spawned:

$$\begin{array}{l} \Gamma'', \neg \mathit{contains}\ {R_Q}'\ \mathit{Payment} \vdash \\ \quad \left\langle \mathit{UserPid}!\{\texttt{contract\_ok}, B_p'\}, \mathit{rm}(R_L, PCC, \mathit{RMname}) \ldots, R_P, {R_Q}' \right\rangle \\ \quad \| \quad \left\langle \mathit{agent}(P_r, PCC, \mathit{RMname}, \mathit{Payment}'), {P_P}', \epsilon \right\rangle \\ \quad : \quad \mathit{notrans}\ \mathit{Payment} \end{array} \tag{6.40}$$

where Γ'' is Γ' together with inequations involving the fresh process identifier ${P_P}'$, and the fact that $\mathit{Payment}' \neq \mathit{Payment}$ (since this is guaranteed by the input clause, or rather, if $\mathit{Payment}$ is input an alternative proof strategy is used). This goal is handled by applying $\textsc{termcut}_{\textsf{L}}$ to the parallel composition using $\mathit{notrans}\ \mathit{Payment}$ as the cut formula both to the left and to the right process. The resulting goals are:

$$\begin{array}{l} \Gamma'', \neg \mathit{contains}\ {R_Q}'\ \mathit{Payment} \vdash \\ \quad \left\langle \mathit{UserPid}!\{\texttt{contract\_ok}, {P_P}'\}, \mathit{rm}(R_L, PCC, \mathit{RMname}) \ldots, R_P, {R_Q}' \right\rangle \\ \quad : \quad \mathit{notrans}\ \mathit{Payment} \end{array} \tag{6.41}$$

$$\Gamma'' \vdash \left\langle \mathit{agent}(P_r, PCC, \mathit{RMname}, \mathit{Payment}'), {P_P}', \epsilon \right\rangle : \mathit{notrans}\ \mathit{Payment} \tag{6.42}$$

$$\begin{array}{l} \Gamma'', S_3 : \mathit{notrans}\ \mathit{Payment}, S_4 : \mathit{notrans}\ \mathit{Payment} \vdash \\ \quad S_3 \parallel S_4 : \mathit{notrans}\ \mathit{Payment} \end{array} \tag{6.43}$$

Goal (6.42) is easy to prove, since no new processes are created (proof sketch omitted). For the handling of goal (6.41) several arguments about data are needed. For example, the property $\neg \mathit{contains}\ \{\texttt{contract\_ok}\ B_p'\}, \mathit{Payment}$ must be established, since this is the value the resource manager will send as a confirmation. This property clearly holds since P_P' is a fresh pid. The resulting goal, after a simple step where the resulting sequence is reduced, becomes

$$\begin{array}{lll} \Gamma'', \neg \mathit{contains}\ {R_Q}'\ \mathit{Payment} & \vdash & \left\langle \mathit{rm}(R_L, PCC, \mathit{RMname}), R_P, {R_Q}' \right\rangle \\ & : & \mathit{notrans}\ \mathit{Payment} \end{array} \tag{6.44}$$

This goal can be discharged against the leftmost conjunct of (6.35).

Now only the two goals 6.36 and 6.43 that deal with compositional reasoning remain. These types of goals are handled in a uniform and regular way, by applying the derived proof rules for parallel composition, e.g., $\| \ [!]1$, found on Page 114, and by discharging against previously seen goals. In the checked proof this compositional reasoning, modulo a few lemmas involving data reasoning, was achieved using a general tactic. For completeness here we instead examine a few manual steps in the proof of goal 6.43; the proof of goal 6.36 is similar.

First the right-hand side $notrans$ formula is approximated (using rule $\mathsf{APPRX_R}$) producing the goal

$$\begin{array}{l}\Gamma'', S_3 : notrans\ Payment, S_4 : notrans\ Payment \vdash \\ \quad S_3 \parallel S_4 : notrans^{\kappa}\ Payment\end{array} \tag{6.45}$$

Then a contraction rule is applied to the left-hand side assumptions about $notrans$, after which all instances of the $notrans$ formula are unfolded, and applications are split. This results in three goals

$$\Gamma_{\parallel} \vdash S_3 \parallel S_4 : [\tau] notrans^{\kappa'}\ Payment \tag{6.46}$$

$$\Gamma_{\parallel} \vdash S_3 \parallel S_4 : \forall P, V.[P?V]\left(contains\ V\ Payment \vee notrans^{\kappa'}\ Payment\right) \tag{6.47}$$

$$\Gamma_{\parallel} \vdash S_3 \parallel S_4 : \forall P, V.[P!V]\left(\neg contains\ V\ Payment \wedge notrans^{\kappa'}\ Payment\right) \tag{6.48}$$

where $\Gamma_{\parallel}$ is

$$\begin{array}{l}\Gamma'', \kappa' < \kappa, \\ S_3 : notrans Payment, S_3 : [\tau] notrans\ Payment, S_3 : \forall \ldots, S_3 : \forall \ldots, \\ S_4 : notrans\ Payment, S_4 : [\tau] notrans\ Payment, S_4 : \forall \ldots, S_4 : \forall \ldots\end{array}$$

To solve goals 6.46, 6.47 and 6.48 the proof rules $\parallel [\tau]$, $\parallel [?]$ and $\parallel [!]$ are applied respectively. For the output case the result are the new goals (after application of the weakening rule $\mathsf{W_L}$)

$$\begin{array}{l}\Gamma'', \kappa' < \kappa, {S_3}' : notrans\ Payment, S_4 : notrans\ Payment \vdash \\ \quad S_3' \parallel S_4 : notrans^{\kappa'}\ Payment\end{array} \tag{6.49}$$

$$\begin{array}{l}\Gamma'', \kappa' < \kappa, {S_4}' : notrans\ Payment, S_3 : notrans\ Payment \vdash \\ \quad S_3 \parallel {S_4}' : notrans^{\kappa'}\ Payment\end{array} \tag{6.50}$$

$$\neg contains\ V\ Payment \vdash \neg contains\ V\ Payment \tag{6.51}$$

$$\neg contains\ V\ Payment \vdash \neg contains\ V\ Payment \tag{6.52}$$

Clearly goals 6.51 and 6.52 are true. For goals 6.49 and 6.50 note that a substitution can be found from goal 6.45, and that a greatest fixed point has been unfolded to the right, resulting in an ordinal inequation being provable. Thus, for both these goals it is safe to discharge at this point. The internal step and the input step cases are treated similarly.

6.4.4 Conclusions

Several important lessons were learned in the proof of the purchasing agent example. One problem seen in practise is that proofs tend to grow large. Without support for

reducing duplication of proof nodes the proof example outlined for the purchasing agent has 10^5–10^6 proof tree nodes. Just by avoiding proof node duplication this figure can be brought down substantially, for the purchasing agent example by roughly a factor of 15. But in fact few steps in the proof convey information which is interesting. These are:

1. Points where a process cut need to be applied, to initiate induction in system state structure.
2. Points where some other symbolic or inductive argument needs to be done, to handle e.g. induction in the message queue structure.
3. Choice points to which we may want to return later, for backtracking. For instance, these can be application of rules like $\exists_R$ where we commit to a particular witness.
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, perhaps using a selection of problem-dependent proof tactics. However, some explicit support for managing proof node histories would then be essential for efficiency.

6.5 Verifying an Active Data Structure

This example focuses on the abstraction of protocols and services through a functional interface, and illustrates how to reason modularly about functions with side effects. It also shows how the gap between expression and system level reasoning in the proof system can be bridged. Part of the proof sketched in this example has been checked using a recent version of the ERLANG verification tool.

Further, the example illustrates the module concept of ERLANG, for which no formal semantics has been given in the thesis, but which is likely to be examined in the future. The example further illustrates typical patterns of reasoning using ERLANG queues, and how their semantics affect proof arguments.

6.5.1 Active Data Structures

Active data structures, i.e., collections of processes that by coordinating their activities mimic in a concurrent way some data structure, are frequently used in telecommunication software. In a previous study [AD99] a protocol for responding to database queries, directed to the distributed database manager Mnesia, was verified. Internally the protocol built up a ring like structure of connected processes in order to answer queries efficiently. In the current example we examine a scheme for a set implementation inspired by a set-as-process example of Hoare [Hoa85]. Here the active data structure is a linked list.

6.5.2 An Implementation of a Persistent Set

As an abstract mathematical notion, a *set* is simply a collection of objects (taken out of an universe of objects), characterised by the membership relation "$\in$": if s is an object and S is a set, then the statement $s \in S$ is either true or false. Computer scientists have also another view of sets, namely as *mutable* objects: a set, when manipulated by adding or removing elements, still keeps its "identity", e.g. through an identifier.

The objects to be manipulated can be distributed in space, and if the objects themselves are large, it is conceivable, that we might want each object to be maintained by a separate process. A further reason for implementing a set as an active data structure is to permit concurrent access to multiple elements.

An implementation of a set, without the possibility to remove elements, by means of a collection of interacting processes is given below in Section 6.5.3, where a module `persistent_set_adt` is defined. The module construct of ERLANG provides a name (here `persistent_set_adt` for a collection of functions, and prohibits direct access to functions not listed in an `export` clause. Functions exported from a module are now accessible only by providing also the module name in function calls. In the present formalisation of ERLANG the module and export keywords are without observable behaviour. A module referring to them can be parsed, but they keywords themself are not interpreted by the proof assistant tool. In the following, to conform with the ERLANG dialect considered in the thesis, the `spawn` function call in the function `mk_empty` has been modified to omit the first argument.

Internally the `persistent_set_adt` module implements two functions - one for maintaining single elements, and one for the empty set. A set is identified by an Erlang process identifier. When creating a new set, it initially consists of a single process executing the `empty_set` function; it is the process identifier of this process by which the set is to be identified. When an element is added, a new process is spawned to store the element if it is not already present in the set. Internally, when a new element is added to a set, it is "pushed downwards" through the list of processes representing set elements, until it reaches the emptyset process, which spawns another `emptyset` process, and becomes itself a process maintaining the new element. So, as a result, a set is implemented as a unidirectional linked collection of processes referenced by a process identifier.

To encapsulate the set against improper use, we provide a controlled interface to the set module, consisting of a function for set creation `mk_empty`, testing for membership `is_member`, addition of elements `add_element`, etc. The set creation function, for example, spawns a process executing the `empty_set` function, and returns the process identifier of the newly spawned process. This process identifier has then to be provided as an argument to all the other interface functions. The implementation of the two set functions and the interface prevents the user of the set module from having to notice that sets are internally represented by processes, and moreover prevents direct access to any other process identifiers created internal to the linked list of processes.

Any process, given knowledge of the process identifier of a persistent set, can choose to circumvent the interface functions and directly communicate (through message passing) with the set process. As we shall see in the proof such "protocol abuse" can lead to program errors.

6.5.3 The Set Erlang Module

```
-module(persistent_set_adt).
-export([mk_empty/0, is_empty/1, is_member/2,
         add_element/2, empty_set/0]).

empty_set() ->
  receive
    {is_empty, Client} when pid(Client) ->
       Client ! {is_empty, true},
       empty_set();
    {is_member, Element, Client} when pid(Client) ->
       Client ! {is_member, Element, false},
       empty_set();
    {add_element, Element}->
       set (Element, mk_empty())
    end.
```

```
set(Element, Set) ->
  receive
    {is_empty, Client} when pid(Client) ->
       Client ! {is_empty, false},
       set(Element, Set);
    {is_member, SomeElement, Client} when pid(Client) ->
       if
         SomeElement == Element ->
            Client ! {is_member, SomeElement, true},
            set(Element, Set);
         SomeElement /= Element ->
            Set ! {is_member, SomeElement, Client},
            set(Element, Set)
       end;
    {add_element, SomeElement} ->
       if
         SomeElement == Element ->
            set(Element, Set);
         SomeElement /= Element ->
            Set ! {add_element, SomeElement},
            set(Element, Set)
       end
    end.

%% MODULE INTERFACE FUNCTIONS

mk_empty() ->
    spawn(persistent_set_adt, empty_set, []).

is_empty(Set) ->
    Set ! {is_empty, self()},
    receive
        {is_empty, Value} -> Value
    end.

is_member(Element, Set) ->
    Set ! {is_member, Element, self()},
    receive
        {is_member, Element, Value} -> Value
    end.

add_element(Element, Set) ->
    Set ! {add_element, Element}.
```

6.5.4 A Persistent Set Property

To check the correctness of a persistent set implementation, we have to specify those properties of sets which we consider paramount for correct behaviour. Ideally, one would like such a specification to be complete, i.e. a system should satisfy the specification exactly when it implements such a set. This goes beyond the scope of this thesis.

One crucial property of persistent sets is naturally that they retain any element added to them. For simplicity, we will here prove a simpler property, that once any element has been added to such a set the set will forever be non-empty. The main predicates are:

$$
\begin{array}{l}
\mathit{ag\_non\_empty} \Rightarrow \\
\quad \lambda \mathit{SetPid} : \mathsf{erlangPid}, \mathit{SetSys} : \mathsf{erlangSystem}. \\
\quad \left(\begin{array}{ll} & \mathit{SetSys} : \mathit{non\_empty}\ \mathit{SetPid} \\ \wedge & \mathit{SetSys} : \forall A : \mathsf{erlangSysAction}.[A]\mathit{ag\_non\_empty}\ SetPid \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{persistently\_non\_empty} \Rightarrow \\
\quad \lambda \mathit{SetPid} : \mathsf{erlangPid}, \mathit{SetSys} : \mathsf{erlangSystem}. \\
\quad \left(\begin{array}{ll} & \mathit{SetSys} : \mathit{non\_empty}\ \mathit{SetPid} \wedge SetSys : \mathit{ag\_non\_empty}\ \mathit{SetPid} \\ \vee & \left(\begin{array}{ll} & \mathit{SetSys} : \mathit{empty}\ \mathit{SetPid} \\ \wedge & \mathit{SetSys} : \forall A : \mathsf{erlangSysAction}. \\ & \quad [A]\mathit{persistently\_non\_empty}\ SetPid \end{array} \right) \end{array} \right)
\end{array}
$$

Intuitively the *persistently_non_empty* predicate expresses an automaton that, when applied to a process identifier *SetPid* and an Erlang system *SetSys* representing a set, checks that *empty SetPid* remains true until *non_empty SetPid* becomes true, after which *non_empty SetPid* must remain continuously true forever (definition *ag_non_empty*). Note that this is, in some respect, a challenging property since it contains both a safety part (*non-empty sets never claim to be empty*) and a liveness part (*all sets eventually answer queries whether they are empty*).

In a lemma we show that no Erlang system can at the same time satisfy both *empty* and *non_empty*, thus increasing our confidence in the above formula:

$$\forall S : \mathsf{erlangSystem}. \neg\,(S : \mathit{empty}\ \mathit{SetPid}) \vee \neg\,(S : \mathit{non\_empty}\ \mathit{SetPid})$$

We advocate an observational approach to specification, through invocation of the interface functions, as evidenced in the definition of the *empty* predicate:

$$
\begin{array}{l}
\mathit{empty} \triangleq \\
\quad \lambda \mathit{SetPid} : \mathsf{erlangPid}, \mathit{SetSys} : \mathsf{erlangSystem}. \forall \mathit{Pid} : \mathsf{erlangPid}. \\
\quad\quad \neg(Pid = SetPid) \Rightarrow \\
\quad\quad\quad \langle \mathtt{is\_empty}(\mathit{SetPid}), \mathit{Pid}, \epsilon \rangle \parallel \mathit{SetSys} : (\mathit{evaluates\_to}\ \mathit{Pid}\ \mathtt{true})
\end{array}
$$

The *empty* predicate expresses that $\langle \texttt{is\_empty}(\textit{SetPid}), \textit{Pid}, \epsilon \rangle$, an observer process, will eventually (in a finite number of steps) – the liveness part of the property – terminate with the value `true`, if executing concurrently with the observed set *SetSys*. The definition of $nonempty$ is analogous:

$$
\begin{array}{l}
nonempty \triangleq \\
\quad \lambda \textit{SetPid} : \mathsf{erlangPid}, \textit{SetSys} : \mathsf{erlangSystem}.\forall \textit{Pid} : \mathsf{erlangPid}. \\
\quad\quad \neg(Pid = SetPid) \Rightarrow \\
\quad\quad\quad \langle \texttt{is\_empty}(\textit{SetPid}), \textit{Pid}, \epsilon \rangle \parallel \textit{SetSys} : (\textit{evaluates\_to Pid } \texttt{false})
\end{array}
$$

The definition of *evaluates_to* predicate below

$$
\begin{array}{l}
\textit{evaluates\_to} : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Leftarrow \\
\quad \lambda Pid : \mathsf{erlangPid}.\lambda\, Value : \mathsf{erlangValue}. \\
\quad\quad (sthe\_term\, Pid\, Value) \vee (\langle \tau \rangle \textit{true} \wedge [\tau](\textit{evaluates\_to } Pid\ \ Value))
\end{array}
$$

expresses that within a finite number of steps the process with process identifier Pid will always evaluate to $Value$. The definition of the $sthe\_term$ predicate can be found on Page 111.

6.5.5 A Proof Sketch

The main proof obligation is then

$$\vdash \langle \texttt{empty\_set}\,\texttt{()}, P, \epsilon \rangle : \textit{persistently\_non\_empty } P$$

That is, the Erlang system $\langle \texttt{empty\_set}\,\texttt{()}, P, \epsilon \rangle$, corresponding to an initially empty set, satisfies the *persistently_non_empty* P property. In fact a slightly stronger property will be proved, and where the *persistently_non_empty* fixed point has been approximated:

$$\neg(\textit{add\_in\_queue } Q) \vdash \langle \texttt{empty\_set}\,\texttt{()}, P, Q \rangle : \textit{persistently\_non\_empty}^{\kappa} P \quad (6.53)$$

where the $\neg(\textit{add\_in\_queue } Q)$ assumption expresses that the queue Q does not contain a request to add an element to the queue:

$$
\begin{array}{l}
\textit{add\_in\_queue} \Leftarrow \\
\quad \lambda Q : \mathsf{erlangQueue}. \\
\quad \left(\begin{array}{ll}
 & \exists V : \mathsf{erlangValue}.Q = \{\texttt{add\_element}, V\} \\
\vee & \exists V : \mathsf{erlangValue}, Q' : \mathsf{erlangQueue}.\textit{add\_in\_queue } Q' \wedge Q = V \cdot Q'
\end{array} \right)
\end{array}
$$

This proof goal is reduced by unfolding the definition of the *persistently_non_empty* predicate, choosing to show that the set process will signal that it is empty when

queried, and performing a few other trivial proof steps. There are two resulting proof goals:

$$\kappa' < \kappa, \neg(\textit{add\_in\_queue}\ Q) \vdash \langle \texttt{empty\_set}\,() , P, Q\rangle : \textit{empty}\ P \tag{6.54}$$

$$\begin{array}{l}\kappa' < \kappa, \neg(\textit{add\_in\_queue}\ Q) \vdash \langle \texttt{empty\_set}\,() , P, Q\rangle : \\ \quad \forall A : \mathsf{erlangSysAction}.[A]\textit{persistently\_non\_empty}^{\kappa'}\ P\end{array} \tag{6.55}$$

Goal 6.54 reduces to (after unfolding *empty* and rewriting):

$$\begin{array}{l}\kappa' < \kappa, \neg(\textit{add\_in\_queue}\ Q), \neg P = P' \vdash \\ \quad \langle \texttt{is\_empty}(P), P', \epsilon\rangle \parallel \langle \texttt{empty\_set}\,() , P, Q\rangle : \\ \qquad \textit{evaluates\_to}\ P'\ \texttt{true}\end{array} \tag{6.56}$$

That is, an observer process calling the interface routine `is_empty` with the set process identifier P as argument will eventually (in a finite number of steps) evaluate to the value `true` (meaning that the set is considered empty). Here the proof strategy is to symbolically "execute" the two processes together with the formula, and observe that in all possible future states the observer process terminates with `true` as the result. Note however that the assumption $\neg(\textit{add\_in\_queue}\ Q)$ is crucial due to the Erlang semantics of queue handling. If the queue Q contains an `add_element` message the observer process will instead return `false` as a result, since its `is_empty` message would be stored after the `add_element` message in the queue and thus be serviced only after an element was added to the set.

The second proof goal 6.55 is reduced by eliminating the universal quantifier, and computing the next state under all possible types of actions. Since the process is unable to perform an output action there are two resulting goals, one which corresponds to the input of a message V (note the resulting queue $Q \cdot V$) and the second a computation step (applying the `empty_set` function).

$$\begin{array}{l}\kappa' < \kappa, \neg(\textit{add\_in\_queue}\ Q)\ \vdash \\ \quad \langle \texttt{empty\_set}(), P, Q \cdot V\rangle : \textit{persistently\_non\_empty}^{\kappa'}\ P\end{array} \tag{6.57}$$

$$\begin{array}{l}\kappa' < \kappa, \neg(\textit{add\_in\_queue}\ Q)\ \vdash \\ \quad \langle \texttt{receive}\ldots, P, Q\rangle : \textit{persistently\_non\_empty}^{\kappa'}\ P\end{array} \tag{6.58}$$

Proceeding with goal 6.58 either the first message to be read from the queue is `is_empty` or `is_member` (the possibility of an `add_element` message can be discarded due to the queue assumption). Handling these two new goals presents no major difficulties. Goal 6.57 is reduced by analysing the value of V. If it is not an `add_element` message then we can easily extend the assumption about the non-emptiness of Q:

$$\begin{array}{l}\kappa' < \kappa, \neg(\textit{add\_in\_queue}\ Q \cdot V)\ \vdash \\ \quad \langle \texttt{empty\_set}(), P, Q \cdot V\rangle : \textit{persistently\_non\_empty}^{\kappa'}\ P\end{array} \tag{6.59}$$

Goal 6.59 is clearly an instance of goal 6.53, i.e., we can find a substitution of variables that when applied to the original goal will result in the current proof goal (the identity substitution except that it maps the queue Q to the queue $Q \cdot V$). Since we have at the same time unfolded a greatest fixed point on the right hand side of the turnstile (the definition of *persistently_non_empty*) we are allowed to discharge the current proof goal at this point. If, on the other hand, V is an `add_element` message the next goal becomes:

$$\begin{array}{l} \kappa' < \kappa, \mathit{add\_in\_queue}\ Q \cdot V\ \vdash \\ \qquad \langle \texttt{empty\_set}(), P, Q \cdot V \rangle : \mathit{persistently\_non\_empty}^{\kappa'}\ P \end{array} \tag{6.60}$$

At this point we cannot discharge the proof goal, since there is no substitution from the original proof goal to the current one. Instead we repeat the steps of the proof of goal 6.53 but taking care to show *non_empty* P instead of *empty* P. Also, we cannot discard the possibility of receiving an `add_element` message adding an element V' to the list and the resulting goal is (after weakening out the assumptions):

$$\vdash \langle \texttt{set}(V', \texttt{mk\_empty}(...)), P, Q' \rangle : \mathit{ag\_non\_empty}\ P \tag{6.61}$$

By repeating the above pattern of reasoning with regards to goal 6.61 we eventually reach the proof state:

$$\neg P = P' \vdash \langle \texttt{set}(V', P'), P, Q'' \rangle \parallel \langle \texttt{empty\_set}(), P', \epsilon \rangle : \mathit{ag\_non\_empty}\ P \tag{6.62}$$

The ERLANG components of the proof states of the proof, up to the point of the spawning off of the new process, are illustrated in Figure 6.8.

At this point we have reached a critical point in the proof where some manual decision is required. Clearly we can repeat the above proof steps forever, never being able to discharge all proof goals, due to the possibility of spawning new processes. Instead we apply the TERMCUT proof rule, to abstract the freshly spawned processes with a formula ψ ending up with two new proof goals:

$$\neg P = P' \vdash \langle \texttt{empty\_set}, P', \epsilon \rangle : \psi\ P\ P' \tag{6.63}$$

$$\neg P = P', X : \psi\ P\ P' \vdash \langle \texttt{set}(V', P'), P, Q'' \rangle \parallel X : \mathit{ag\_non\_empty}\ P \tag{6.64}$$

How should we choose ψ? The cut formula must be expressive enough to characterise the $\langle \texttt{empty\_set}, P', \epsilon \rangle$ process, in the context of the second process and for the purpose of proving the formula *ag_non_empty* P. Here it turns out that the following

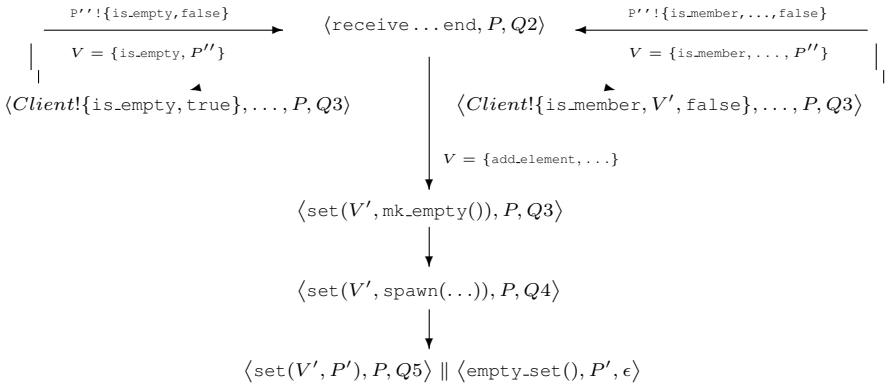

Figure 6.8: Erlang components of initial proof states

formula definitions are sufficient:

$$\psi : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Rightarrow$$
$$\lambda P, P' : \mathsf{erlangPid}.$$
$$\left(\begin{array}{ll} & \forall P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P?V](\neg(\mathit{is\_empty}\ V) \Rightarrow \psi\ P\ P') \\ \wedge & \forall P : \mathsf{erlangPid}, \forall V : \mathsf{erlangValue}.[P!V]\neg(\mathit{is\_empty}\ V) \\ \wedge & \mathit{converges}\ P\ P' \wedge \mathit{foreign}\ P \wedge \mathit{local}\ P' \end{array}\right)$$

$$\mathit{converges} : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \Leftarrow$$
$$\lambda P, P' : \mathsf{erlangPid}.$$
$$\left(\begin{array}{ll} & \psi\ P\ P' \\ \wedge & [\tau]\mathit{converges}\ P\ P' \\ \wedge & \forall P'' : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P''!V]\mathit{converges}\ P\ P'' \end{array}\right)$$

$$\mathit{is\_empty} : \mathsf{erlangValue} \rightarrow \mathsf{prop} =$$
$$\lambda V : \mathsf{erlangValue}.\exists P : \mathsf{erlangValue}.V = \{\mathtt{is\_empty}, P\}$$

Intuitively ψ expresses:

- Whenever a new message is received, and it is not an `is_empty` message then ψ continues to hold.
- An `is_empty` reply is never issued.

- The predicated system can only perform a finite number number of internal and output steps (definition of *converges* omitted).
- Process identifier P is foreign (does not belong to any process in the predicated system) and process identifier P' is local, characterised by the definitions on Page 111.

The proof of goal 6.63 is straightforward up to reaching the goal:

$$\neg P' = P'' \vdash \langle \texttt{set}(V', P''), P', Q''' \rangle \parallel \langle \texttt{empty\_set}, P'', \epsilon \rangle : \psi\ P\ P' \tag{6.65}$$

Here we once again apply the term-cut rule to obtain the following goals:

$$\neg P' = P'' \vdash \langle \texttt{empty\_set}, P'', \epsilon \rangle : \psi\ P'\ P'' \tag{6.66}$$

$$Y : \psi\ P'\ P'' \vdash \langle \texttt{set}(V', P''), P', Q''' \rangle \parallel Y : \psi\ P\ P' \tag{6.67}$$

Goal 6.66 can be discharged immediately due to the fact that it is an instance of goal 6.63. Goal 6.67 involves symbolically executing the $\langle \texttt{set}(V', P''), P', Q''' \rangle$ process together with the (abstracted) process variable Y, thus generating their combined proof state space. Since both these systems generate finite proof state spaces this construction will eventually terminate. The proof of goal 6.64 is highly similar to the proof of goal 6.67 above, but is omitted from the presentation.

6.5.6 A Discussion of the Proof

The proof itself represented a serious challenge in several respects:

- The modelled system is an open one in which at any time additional set elements can be added outside of the control of the set implementation itself. The state space of the set implementation is clearly not finite state: both the number of processes and the size of message queues can potentially grow without bound.
- The queue semantics of Erlang has some curious effects with regards to an observer interacting with the set implementation. It is for instance not sufficient to consider only the program states of the set process to determine whether an observer will recognise a set to be empty or not; also the contents of the input message queue of the set process has to be taken into account.

Although the correctness of the program may at first glance appear obvious, a closer inspection of the source code through the process of proving the implementation correct revealed a number of problems.

For instance, in an earlier version of the set module the guards `pid(Client)` in the `empty_set` and `set` functions were missing. These guards serve to ensure that any received `is_empty` or `is_member` message must contain a valid process identifier. Should these guards be removed a set process will terminate due to a runtime (typing) error if, say, a message $\{\texttt{is\_empty}, 21\}$ is sent to it.

In most languages adding such guards would not be needed since usage of the interface functions should ensure that these kinds of "typing errors" can never take place.

In Erlang, in contrast, it is perfectly possible to circumvent the interface functions and communicate directly with the set implementation.

6.6 Formal Verification of a Leader Election Protocol in Process Algebra

The language μCRL (micro Common Representation Language) [GP90] has been defined as a combination of process algebra and (equational) data types to describe and verify distributed systems. It is a very precisely defined language provided with a logical proof system [GP94]. It is primarily intended to verify statements of the form

$$Condition \rightarrow Specification = Implementation.$$

This language and proof system has been applied to verify a number of data transfer and distributed scheduling protocols of considerable complexity [BG93, GK93, GvdP93, Kor94]. It incorporates several old and new techniques [BG94, BG93]. Due to the logical nature of the proof system proofs can be verified by computer. Some sizable examples of proofs verified using the proof checker Coq [CH88] are reported in [GP94, KS94].

In this section we show its applicability on Dolev, Klawe and Rodeh's *leader election* or *extrema finding* protocol [DKR82] that has been designed for a network with a unidirectional ring topology. At the same time, Peterson published a nearly identical version of this protocol [Pet82]. This protocol is efficient, $O(n \log n)$, and highly parallel. As far as we know this is the first leader election protocol verified in a process algebraic style. In [BKKM95, BKKM96] a number of leader election protocols for carrier sense networks have been specified and some (informal) proof sketches are given in modal logic.

In Section 6.6.1 we specify Dolev, Klawe and Rodeh's leader election protocol formally in μCRL. The protocol is proven correct in Section 6.6.2 using a detailed argument. Section 6.6.4 summarises the proof theory for μCRL, and Section 6.6.5 defines the data types used in the specification and proof of the protocol.

Acknowledgements. We thank Frits Vaandrager for pointing out this protocol to us. Also Marco Pouw is thanked for his suggestions for improvement.

6.6.1 Specification and correctness of the protocol

We assume n processes in a ring topology, connected by unbounded queues. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier ident (in the following assumed to be a natural number). The task of an algorithm for solving the leader election problem is then to make sure that eventually exactly one process will become the leader.

In Dolev, Klawe and Rodeh's algorithm [DKR82] each process in the ring carries out the following task:

Active:
d:= ident
do forever
    send(d)

receive(e)
if d=e **then stop** /* *Process is the leader* */
send(e)
receive(f)
if $e > max(d, f)$ **then** d:=e **else goto** *Relay*
end

Relay:
do forever
receive(d)
send(d)
end

The intuition behind the protocol is as follows. In each round the number of electable processes decreases, if there are more than two active processes around. During each round every active process, i.e., a process in state *Active*, receives two different values. If the first value is larger than the second value and its own value, then it stays active. In this case its anti-clockwise neighbour will become a relay process. So, from every set of active neighbours, one will die in every round. Furthermore, the maximal value among the identifiers will never be lost in the ring network, it will traverse the ring in messages, or be stored in a variable in a process, until only one active process remains. If only one active process is left, i.e., not in state *Relay*, then the leader-in-spe sends its own value of d to itself, and then halts.

As the attentive reader may have noticed, there is a simpler way to elect a leader. For example, it would be sufficient for a process to receive just one value, i.e., the value (e) of its direct neighbour. In this case, only two values instead of three values have to be compared ($e > d$ instead of $e > max(d, f)$). However, this approach is not so efficient as one may need $2n^2 + 2n$ actions before a leader is selected. The protocol described earlier is faster. It is bounded by $2n \log n + 2n$ actions because in every round at least one process becomes inactive.[2] For an explanation of these complexity bounds one is referred to [DKR82].

Below we formalise the processes and their configuration in the ring as described above in μCRL.

act *leader*
$r, s : Nat \times Nat$
proc $Active(i{:}Nat, d{:}Nat, n{:}Nat) =$
$s(i,d) \sum_{e:Nat} r(i{-_n}1, e)\,(leader\,\delta \lhd eq(d,e) \rhd s(i,e)$
$\sum_{f:Nat}(r(i{-_n}1, f)\,Active(i,e,n) \lhd e > max(d,f) \rhd Relay(i,n)))$
$Relay(i{:}Nat, n{:}Nat) = \sum_{d:Nat} r(i{-_n}1, d)\; s(i,d)\; Relay(i,n)$

Here a process in the imperative description with value ident for d corresponds to $Active(i, \mathsf{ident}, n)$. Intuitively the μCRL process first sends the value of the variable d to the next process in the ring ($s(i,d)$) via a queue, which is described below. Then it

[2] By $\log n$, we mean $\log_2 n$.

reads a new value e from the queue connected to the preceding process in the ring by an action $r(i -_n 1, e)$. The notation $-_n$ stands for subtraction modulo n. Consequently, it executes a then-if-else test denoted by $\_ \lhd \_ \rhd \_$. If the variables d and e are equal, expressed by $eq(d, e)$, then the process declares itself leader by executing the action *leader*. Otherwise the value of e is sent ($s(i, e)$) and a value f is read ($r(i -_n 1, f)$). Now, if e is larger than both d and f the process repeats itself with e replacing d. Otherwise, the process becomes a relay process (denoted by $\mathit{Relay}(i, n)$). The *leader* action has been introduced in the μCRL specification of the protocol for verification purposes; it makes visible the fact that exactly one leader is elected.

The δ process after the leader action in the *Active* process is not essential. We have inserted it for technical reasons and more details on this issue are given at the end of this section.

In order to prove the correctness of the protocol we must be precise about the behaviour of the queues that connect the processes. We assume that the queues have infinite size and deliver data in a strict first in first out fashion without duplication or loss. In the queue process data is stored in a data queue q. Note that the behaviour of the queue process is straightforward; it reads data via $r(i, d)$ at process i and delivers it via $s(i +_n 1)$ at process $i +_n 1$ ($+_n$ is addition modulo n). Below, $toe(q)$ denotes the first element that was inserted in data queue q.

proc $Q(i{:}Nat, n{:}Nat, q{:}Queue) =$
$\sum_{d:Nat} r(i, d)\, Q(i, n, in(d, q)) +$
$s(i +_n 1, toe(q))\, Q(i, n, \mathit{untoe}(q)) \lhd \text{not } \mathit{empty}(q) \rhd \delta$

It remains to connect all processes together. First we state that send actions s communicate with receive actions r. Then, using the processes *Spec′* and *Spec* we combine the processes with the queues, and assign a unique number to them. The process *Spec*(n) represents a ring network of n processes interconnected by queues. The injective function $\mathit{id} : Nat \to Nat$ maps natural numbers to process identifiers, for convenience also represented as natural numbers. The process identifiers are related by the total ordering $\leq$. The abbreviation max will be used to denote the maximal identifier, with respect to the ordering $\leq$ and the number of processes n, of the set $\{\mathit{id}(x) : 0 \leq x \leq n - 1\}$.

func $\mathit{id} : Nat \to Nat$
act $c : Nat \times Nat$
comm $r|s = c$
proc $\mathit{Spec'}(m{:}Nat, n{:}Nat) =$
$(\mathit{Active}(m - 1, \mathit{id}(m - 1), n) \parallel Q(m - 1, n, q_0) \parallel \mathit{Spec'}(m - 1, n))$
$\lhd m > 0 \rhd \delta$
$\mathit{Spec}(n{:}Nat) = \tau_{\{c\}}\partial_{\{r,s\}}(\mathit{Spec'}(n, n))$

Since the protocol is supposed to select exactly one leader after some internal negotiation we formulate correctness by the following formula, where '$=$' is to be interpreted as 'behaves the same':

Theorem 6.6.1. *For all* $n : Nat$

$$n > 0 \rightarrow Spec(n) = \tau\, leader\, \delta$$

The theorem says that in a ring with at least one process exactly one leader will be elected after some internal activity.

In the specification of the *Active* process given above, we have inserted a δ process after the leader action. We introduced this δ for technical convenience in our verification. However, omitting δ does not effect the behaviour of the leader protocol, $Spec$, as a whole. In fact, if we leave out this δ the whole system $Spec$ still deadlocks after performing a leader action as stated in Theorem 6.6.1. The reason for this is that $Spec$ can only terminate if all processes in the system terminate. In particular, the *Relay* processes can not terminate and evolve in a deadlock situation when a leader is selected. So, even if the process that performs the leader action terminates successfully (which is not the case here), the full system will still end up in a deadlock.

As experience shows the correctness reasoning above is too imprecise to serve as a proof of correctness of the protocol. Many, often rather detailed arguments, are omitted. Actually, the protocol does not have to adhere to the rather synchronous execution suggested by the word 'rounds', but is highly parallel. One can even argue that given the large number of rather 'wild' executions of the protocol, the above description makes little sense. Therefore, we provide in the next sections a completely formalised proof, where we are only interested in establishing correctness of the protocol and not in proving its efficiency.

6.6.2 A proof of the protocol

The proof strategy for proving the correctness theorem consists of a number of distinct steps. First in Section 6.6.2 we define a linear representation of the specification in which the usage of the parallel composition operator in the original specification is replaced by a tabular data structure encoding the states of processes in the network, and actions with guards that check the contents of the data structure. The linearised specification is proven equivalent to the original specification in Lemma 6.6.4. Then, in Section 6.6.2, we define a (focus) condition on the tabular data structure such that if the condition holds then no internal computation is any longer possible in the protocol, i.e., no τ-steps can be made [BG93]. The focus condition is used in Lemma 6.6.11, in Section 9, to separate the proof that the linear specification can be proven equivalent to a simple process into two parts. Lemma 6.6.11 together with Lemma 6.6.4 then immediately proves the correctness theorem of the protocol, i.e., Theorem 6.6.1. The proof of Lemma 6.6.11 makes use of the Concrete Invariant Corollary (see [BG94]), i.e., a number of invariance properties are defined (in Section 6.6.2) on the tabular data structure such that regardless which execution step the linear specification performs, the properties remain true after the step if they were true before the execution of the step. These invariants are used to prove the equality between the linear specification and the simple process in Lemma 6.6.11. In order to make use of the Concrete Invariant

Corollary we have to show that the linear specification can only perform finitely many consecutive τ-steps. This is proven in Section 9.

Linearisation

As a first step the leader election protocol is described as a μCRL process in a state based style, as this is far more convenient for proving purposes. The state based style very much resembles the Unity format [Bru95, CM88] or the I/O automata format [LT89]. Following [Bru95] we call this format the Unity format or a process specification in Unity style. Inspection of the processes *Active* and *Relay* indicates that there are 7 different major states between the actions. The states in *Active* are numbered 0,1,2,3,6 and those in *Relay* get numbers 4 and 5. The processes *Active* and *Relay* can then be restated as follows:

proc $Act(i{:}Nat, d{:}Nat, e{:}Nat, n{:}Nat, s{:}Nat) =$
$s(i,d)\,Act(i,d,e,n,1) \lhd eq(s,0) \rhd \delta +$
$\sum_{e:Nat} r(i -_n 1, e)\,Act(i,d,e,n,2) \lhd eq(s,1) \rhd \delta +$
$\mathit{leader}\,Act(i,d,e,n,6) \lhd eq(d,e)$ and $eq(s,2) \rhd \delta +$
$s(i,e)\,Act(i,d,e,n,3) \lhd$ not $eq(d,e)$ and $eq(s,2) \rhd \delta +$
$\sum_{f:Nat} r(i -_n 1, f)\,Act(i,e,e,n,0) \lhd e > \mathit{max}(d,f)$ and $eq(s,3) \rhd \delta +$
$\sum_{f:Nat} r(i -_n 1, f)\,Act(i,d,e,n,4) \lhd e < \mathit{max}(d,f)$ and $eq(s,3) \rhd \delta +$
$\sum_{d:Nat} r(i -_n 1, d)\,Act(i,d,e,n,5) \lhd eq(s,4) \rhd \delta +$
$s(i,d)\,Act(i,d,e,n,4) \lhd eq(s,5) \rhd \delta$

Lemma 6.6.2. *For all i, d, e, n, we have:*

$$\mathit{Active}(i,d,n) = Act(i,d,e,n,0)$$

$$\mathit{Relay}(i,n) = Act(i,d,e,n,4)$$

Proof. The proof of this lemma is straightforward, using the Recursive Specification Principle (RSP), but note that it uses $a\,(p \lhd c \rhd q) = a\,p \lhd c \rhd a\,q$ as well as the distributivity of Σ over $+$. □

We now put the processes and queues in parallel. As we work towards the Unity style, we must encode the states of the individual processes in a data structure. For this we take a table (or indexed queue) with an entry for each process i. This entry contains values for the variables d, e, s and the contents of the queue in which process i is putting its data. Furthermore, it contains a variable of type $\mathbf{Bool}$, which plays a role in the proof. The data structure has the name *Table* and is defined in Section 6.6.5.

We put the processes and queues together in three stages. First we put all processes together, using Π_{Act} and X_{Act} below. Then we put all queues together, via Π_Q and X_Q. Finally, we combine X_{Act} and X_Q obtaining the process X which is a description in Unity style of the leader election protocol.

proc $Spec(B{:}\mathit{Table}, n{:}Nat) = \tau_{\{c\}}\partial_{\{r,s\}}(\Pi_{Act}(B,n) \,\|\, \Pi_Q(B,n))$
$\Pi_{Act}(B{:}\mathit{Table}, n{:}Nat) =$

$$
\begin{array}{l}
\quad \delta \triangleleft \mathit{empty}(B) \triangleright \\
\quad (Act(hd_i(B), get_d(hd_i(B), B), get_e(hd_i(B), B), n, get_s(hd_i(B), B)) \\
\quad \quad \| \, \Pi_{Act}(tl(B), n)) \\
\Pi_Q(B{:}\mathit{Table}, n{:}Nat) = \\
\quad \delta \triangleleft \mathit{empty}(B) \triangleright (Q(hd_i(B), n, get_q(hd_i(B), B)) \| \, \Pi_Q(tl(B), n))
\end{array}
$$

$$
\begin{array}{l}
X_{Act}(B{:}\mathit{Table}, n{:}Nat) = \\
\quad \sum_{j:Nat} s(j, get_d(j, B)) \, X_{Act}(upd_s(1, j, B), n) \\
\quad \quad \triangleleft eq(get_s(j, B), 0) \text{ and } test(j, B) \triangleright \delta + \\
\quad \sum_{j:Nat} \sum_{e:Nat} r(j -_n 1, e) \, X_{Act}(upd_e(e, j, upd_s(2, j, B)), n) \\
\quad \quad \triangleleft eq(get_s(j, B), 1) \text{ and } test(j, B) \triangleright \delta + \\
\quad \sum_{j:Nat} \mathit{leader} \, X_{Act}(upd_s(6, j, B), n) \\
\quad \quad \triangleleft eq(get_d(j, B), get_e(j, B)) \text{ and } eq(get_s(j, B), 2) \text{ and } test(j, B) \triangleright \delta + \\
\quad \sum_{j:Nat} s(j, get_e(j, B)) \, X_{Act}(upd_s(3, j, B), n) \\
\quad \quad \triangleleft \text{not } eq(get_d(j, B), get_e(j, B)) \text{ and} \\
\quad \quad \quad eq(get_s(j, B), 2) \text{ and } test(j, B) \triangleright \delta + \\
\quad \sum_{f:Nat} \sum_{j:Nat} r(j -_n 1, f) \, X_{Act}(upd_d(get_e(j, B), j, upd_s(0, j, B)), n) \\
\quad \quad \triangleleft get_e(j, B) > \mathit{max}(get_d(j, B), f) \text{ and} \\
\quad \quad \quad eq(get_s(j, B), 3) \text{ and } test(j, B) \triangleright \delta + \\
\quad \sum_{f:Nat} \sum_{j:Nat} r(j -_n 1, f) \, X_{Act}(upd_s(4, j, B), n) \\
\quad \quad \triangleleft get_e(j, B) \leq \mathit{max}(get_d(j, B), f) \text{ and} \\
\quad \quad \quad eq(get_s(j, B), 3) \text{ and } test(j, B) \triangleright \delta + \\
\quad \sum_{d:Nat} \sum_{j:Nat} r(j -_n 1, d) \, X_{Act}(upd_d(d, j, upd_s(5, j, B)), n) \\
\quad \quad \triangleleft eq(get_s(j, B), 4) \text{ and } test(j, B) \triangleright \delta + \\
\quad \sum_{j:Nat} s(j, get_d(j, B)) \, X_{Act}(upd_s(4, j, B), n) \\
\quad \quad \triangleleft eq(get_s(j, B), 5) \text{ and } test(j, B) \triangleright \delta
\end{array}
$$

$$
\begin{array}{l}
X_Q(B{:}\mathit{Table}, n{:}Nat) = \\
\quad \sum_{d:Nat} \sum_{j:Nat} r(j, d) \, X_Q(in_q(d, j, B), n) \triangleleft test(j, B) \triangleright \delta + \\
\quad \sum_{j:Nat} s(j +_n 1, toe(j, B)) \, X_Q(\mathit{untoe}(j, B), n) \\
\quad \triangleleft \text{not } \mathit{empty}(j, B) \text{ and } test(j, B) \triangleright \delta
\end{array}
$$

The leader election protocol in Unity form is given below and will be the core process of the proof. Note that in many cases verification of a protocol only starts after the process below has been written down. In the description of X most details of the description are directly reflected in corresponding behaviour of the constituents X_{Act} and X_Q. However, there is one difference. It appears that in the protocol two kinds of messages travel around, *active* and *passive* ones. The active messages contain numbers that may replace the current value of the d-variable of its receiver. The passive messages are not essential for the correctness of the protocol, but only used to improve its speed. For the correctness of the protocol it is important to know that the maximum identifier is always somewhere in an active position and that no identifier occurs in more than one active position. In order to distinguish active from passive messages, we have added a boolean b to each message in the queues, where if $b = \mathsf{t}$ the message is active, and if $b = \mathsf{f}$ the message is passive. When processes become *Relay*s then

they also act as a queue. Therefore, we have also added a boolean b to the process parameters, to indicate the status of the message that a process in state 5 is holding. The equation below is referred to by (I) in the remainder of the proof.

proc $X(B{:}\mathit{Table}, n{:}Nat) =$
$\sum_{j:Nat} \tau\, X(upd_s(1, j, in_q(get_d(j, B), \mathsf{t}, j, B)), n)$
$\quad \lhd eq(get_s(j, B), 0)$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \tau\, X(\mathit{untoe}(j -_n 1, upd_e(toe(j -_n 1, B), j, upd_s(2, j, B))), n)$
$\quad \lhd eq(get_s(j, B), 1)$ and not $\mathit{empty}(j -_n 1, B)$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \mathit{leader}\, X(upd_s(6, j, B), n)$
$\quad \lhd eq(get_s(j, B), 2)$ and $eq(get_d(j, B), get_e(j, B))$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \tau\, X(upd_s(3, j, in_q(get_e(j, B), \mathsf{f}, j, B)), n)$
$\quad \lhd eq(get_s(j, B), 2)$ and not $eq(get_d(j, B), get_e(j, B))$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \tau\, X(\mathit{untoe}(j -_n 1, upd_d(get_e(j, B), j, upd_s(0, j, B))), n)$
$\quad \lhd get_e(j, B) > \mathit{max}(get_d(j, B), toe(j -_n 1, B))$ and $eq(get_s(j, B), 3)$ and
$\qquad$ not $\mathit{empty}(j -_n 1, B)$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \tau\, X(\mathit{untoe}(j -_n 1, upd_s(4, j, upd_b(toe_b(j -_n 1, B), j, B))), n)$
$\quad \lhd get_e(j, B) \leq \mathit{max}(get_d(j, B), toe(j -_n 1, B))$ and $eq(get_s(j, B), 3)$ and
$\qquad$ not $\mathit{empty}(j -_n 1, B)$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \tau\, X(\mathit{untoe}(j -_n 1, upd_d(toe(j -_n 1, B), j, upd_s(5, j, upd_b(toe_b$
$\qquad\qquad (j -_n 1, B), j, B)))), n)$
$\quad \lhd eq(get_s(j, B), 4)$ and not $\mathit{empty}(j -_n 1, B)$ and $j < n \rhd \delta +$
$\sum_{j:Nat} \tau\, X(in_q(get_d(j, B), get_b(j, B), j, upd_s(4, j, B)), n)$
$\quad \lhd eq(get_s(j, B), 5)$ and $j < n \rhd \delta$

Definition 6.6.3. The function $\mathit{init} : Nat \rhd htarrow \mathit{Table}$, which is used for denoting the initial state of the protocol, is defined as follows:

$$\mathit{init}(n) = \mathit{if}(eq(n, 0), t_0, in(n - 1, \mathit{id}(n - 1), 0, 0, \mathsf{f}, q_0, \mathit{init}(n - 1))).$$

See also Section 6.6.5.

Lemma 6.6.4. *For all B : Table and $m, n : Nat$*

1. $\mathit{UniqueIndex}(B) \rightarrow \Pi_{Act}(B, n) = X_{Act}(B, n)$,
2. $\mathit{UniqueIndex}(B) \rightarrow \Pi_Q(B, n) = X_Q(B, n)$,
3. $\mathit{UniqueIndex}(B) \wedge test(j, B) = j < n \rightarrow \mathit{Spec}(B, n) = X(B, n)$,
4. $\mathit{Spec}'(m, n) = \Pi_{Act}(\mathit{init}(m), n) \,\|\, \Pi_Q(\mathit{init}(m), n)$,
5. $\mathit{Spec}(n) = \mathit{Spec}(\mathit{init}(n), n)$,
6. $\mathit{Spec}(n) = X(init(n), n)$.

Proof.

1. A standard expansion using induction on B (cf. [KS94]).
2. Again a straightforward expansion.
3. $Spec(B,n) = \tau_{\{c\}}\partial_{\{r,s\}}(\Pi_{Act}(B,n) \parallel \Pi_Q(B,n)) = \tau_{\{c\}}\partial_{\{r,s\}}(X_{Act}(B,n) \parallel X_Q(B,n))$. Now expand $X_{Act}(B,n) \parallel X_Q(B,n)$ and apply hiding. The equations obtained in this way match those of $X(B,n)$, except that '$j < n$' is replaced by '$test(j,B)$' or '$test(j,B)$ and $test(j -_n 1, B)$'. As X is convergent (proven in Lemma 6.6.8) it follows with the Concrete Invariant Corollary [BG94] that $Spec(B,n)$ and $X(B,n)$ are equal. The invariant '$test(j,B) = j < n$' is used and easy to show true.
4. By induction on m, using associativity and commutativity of the merge.
5. Directly from the previous case, i.e. Lemma 6.6.4.4.
6. Directly using cases 3 and 5.

□

Notation

In the sequel we will for certain property formulas $\phi(j)$ write

$$\forall_{j<n}\phi(j) \text{ for } \phi'(0,n) \text{ and } \forall_{i<j<n}\phi(j) \text{ for } \phi'(i+1,n)$$

and

$$\exists_{j<n}\phi(j) \text{ for } \phi''(0,n) \text{ and } \exists_{i<j<n}\phi(j) \text{ for } \phi''(i+1,n)$$

where $\phi'(j,n)$ and $\phi''(j,n)$ are defined by:

$$\phi'(j,n) = \mathit{if}(j \geq n, \mathsf{t}, \phi(j) \text{ and } \phi'(j+1,n)),$$
$$\phi''(j,n) = \mathit{if}(j \geq n, \mathsf{f}, \phi(j) \text{ or } \phi''(j+1,n)).$$

Summation over an arithmetic expression $\gamma(j)$ can be written

$$\textstyle\sum_{j<n}\gamma(j) \text{ for } \gamma'(0,n)$$

where

$$\gamma'(j,n) = \mathit{if}(j \geq n, 0, \gamma(j) + \gamma'(j+1,n)).$$

Note that if we can prove that

$$(j < n \text{ and } \phi(j)) \rightarrow \psi(j),$$

then we can also show that

$$\forall_{j<n}\phi(j) \rightarrow \forall_{j<n}\psi(j) \text{ and}$$
$$\exists_{j<n}\phi(j) \rightarrow \exists_{j<n}\psi(j).$$

Also note that

$$\text{not } (\forall_{j<n}\phi(j)) = \exists_{j<n}\text{not } \phi(j) \text{ and}$$
$$\text{not } (\exists_{j<n}\phi(j)) = \forall_{j<n}\text{not } \phi(j)$$

Focus Condition

The focus condition $FC : \mathit{Table} \times \mathit{Nat} \to \mathbf{Bool}$ indicates at which points the leader election protocol cannot do τ-steps. This means it can either do nothing, or do a *leader* action. The focus condition is constructed in a straightforward fashion by collecting the conditions for the τ-steps in process X.

$$
\begin{array}{ll}
FC(B,n) = & \\
\forall_{j<n} & \text{not } eq(get_s(j,B),0) \text{ and} \\
& (\text{not } eq(get_s(j,B),1) \text{ or } \mathit{empty}(j -_n 1, B)) \text{ and} \\
& (\text{not } eq(get_s(j,B),2) \text{ or } eq(get_d(j,B), get_e(j,B))) \text{ and} \\
& (\text{not } eq(get_s(j,B),3) \text{ or } \mathit{empty}(j -_n 1, B)) \text{ and} \\
& (\text{not } eq(get_s(j,B),4) \text{ or } \mathit{empty}(j -_n 1, B)) \text{ and} \\
& \text{not } eq(get_s(j,B),5)
\end{array}
$$

Some invariants of X

In this section we state four invariants ($Inv_1, \ldots, Inv_4$) of the process $X(B,n)$ that are used in Section 9 to prove the correctness of the protocol. We prove that the predicates below are indeed invariance properties in a traditional manner. First we show that they hold in the initial state of the protocol, i.e., for invariant Inv_i we show $Inv_i(\mathit{init}(n), n)$. Then for each protocol step (there are eight such steps in the linearised process X) we show that if both the precondition of the step holds and the predicate holds in the state before the protocol step, then the predicate holds also in the state that is the result of performing the step. For example, to prove that Inv_2 is an invariance property we need to establish that the first step in X preserves the property, i.e., that

$$
\begin{array}{l}
eq(get_s(j,B),0) \text{ and } j < n \text{ and } Inv_2(B,n) \to \\
\quad Inv_2(upd_s(1,j,in_q(get_d(j,B),\mathsf{t},j,B)),n)
\end{array}
$$

where B is a tabular data structure. This entails proving a large number of rather trivial lemmas, such as:

$$qsizes(upd_s(1,j,B),n) = qsizes(B,n)$$

We omit here the rather long and tedious details of these proofs. In order to establish that Inv_3 and Inv_4 are indeed invariants we first have to prove additional statements on the behaviour of the protocol, i.e., $\mathit{Inv}_5, \mathit{Inv}_7, \mathit{Inv}_6$, Inv_8 and Inv_9 in Sections 4 to 8 respectively.

Acceptable states Each process is in one of the states $0, \ldots, 6$:

$$Inv_1(B,n) = \forall_{j<n} 0 \le get_s(j,B) \le 6$$

Bound on the number of messages in queues Invariant Inv_2 expresses the property that the number of processes in state 1 or 3 is equal to the number of processes in state 5 plus the number of messages in message channels.

$$\begin{aligned}&Inv_2(B,n) = \\ &\quad eq(nproc(B,1,n) + nproc(B,3,n), nproc(B,5,n) + qsizes(B,n))\end{aligned}$$

where

$$\begin{aligned}&nproc(B,s,n) = \textstyle\sum_{j<n} if(eq(get_s(j,B),s),1,0)\\ &qsizes(B,n) = \textstyle\sum_{j<n} size(get_q(j,B))\end{aligned}$$

Termination of one process implies termination of all processes Invariant Inv_3 expresses that if a process is in state 6, then all processes are either in state 4 or state 6. It is provable using invariant Inv_9.

$$\begin{aligned}&Inv_3(B,n) = (\exists_{j<n} eq(get_s(j,B),6)) \rightarrow \\ &\qquad \forall_{j<n} eq(get_s(j,B),4) \text{ or } eq(get_s(j,B),6)\end{aligned}$$

Max is preserved In the initial state, $init(n)$, the maximal identifier in the ring is equal to max. Invariant Inv_4 expresses that this value can not be lost. The invariants Inv_5, Inv_7, Inv_6 are needed to establish Inv_4.

$$Inv_4(B,n) = \exists_{j<n} ActiveNode(\mathsf{max}, j, B) \text{ or } ActiveChan(\mathsf{max}, get_q(j,B))$$

where

$$\begin{aligned}&ActiveNode(k,j,B) = \\ &\quad (eq(get_s(j,B),0) \text{ and } eq(get_d(j,B),k)) \text{ or}\\ &\quad ((eq(get_s(j,B),2) \text{ or } eq(get_s(j,B),3) \text{ or } eq(get_s(j,B),6)) \text{ and}\\ &\quad eq(get_e(j,B),k)) \text{ or}\\ &\quad (eq(get_s(j,B),5) \text{ and } get_b(j,B) \text{ and } eq(get_d(j,B),k))\end{aligned}$$

$$\begin{aligned}&ActiveChan(k,q) = if(empty(q), \mathsf{f},\\ &\quad (hd_b(q) \text{ and } eq(k,hd(q))) \text{ or } ActiveChan(k,tl(q)))\end{aligned}$$

An identifier has not been lost if it can in the future be received by another process and replace the value of the d variable of that process. Identifiers can be stored either in a variable (*ActiveNode*) or in a channel (*ActiveChan*).

Trivial facts Inv_5 formulates two trivial protocol properties, that all identifiers are less than n (less than or equal to the maximal identifier max), and that the values of variables d and e differ when a process is in state 3.

$$\begin{aligned}&Inv_5(B,n) = \\ &\quad \forall_{j<n} Bounded_q(get_q(j,B),n) \text{ and}\\ &\qquad get_d(j,B) < n \text{ and } get_e(j,B) < n \text{ and}\\ &\qquad if(eq(get_s(j,B),3), \text{not } eq(get_d(j,B), get_e(j,B)), \mathsf{t})\end{aligned}$$

where

$$\mathit{Bounded}_q(q,n) = \mathit{if}(\mathit{empty}(q), \mathsf{t}, hd(q) < n \text{ and } \mathit{Bounded}_q(tl(q),n))$$

Active and passive messages The invariant Inv_6 characterises the relation between neighbour processes and channel contents.

$$\mathit{Inv}_6(B,n) = \forall_{j<n}\mathit{Alt}(j,B,n)$$

where

$$\begin{array}{l}
\mathit{Alt}(j,B,n) = \\
\quad \mathit{if}(eq(get_s(j,B),0) \text{ or} \\
\quad\quad eq(get_s(j,B),3) \text{ or} \\
\quad\quad (eq(get_s(j,B),4) \text{ and not } get_b(j,B)) \text{ or} \\
\quad\quad (eq(get_s(j,B),5) \text{ and } get_b(j,B)), \\
\quad\quad \mathit{secondary}(get_q(j,B),j,B,n), \\
\quad\quad \mathit{primary}(get_q(j,B),j,B,n))
\end{array}$$

$$\begin{array}{l}
\mathit{primary}(q,j,B,n) = \\
\quad \mathit{if}(\mathit{empty}(q), \\
\quad\quad eq(get_s(j +_n 1,B),2) \text{ or } eq(get_s(j +_n 1,B),3) \text{ or } eq(get_s(j +_n 1,B),6) \text{ or} \\
\quad\quad ((eq(get_s(j +_n 1,B),4) \text{ or } eq(get_s(j +_n 1,B),5)) \text{ and } get_b(j +_n 1,B)), \\
\quad\quad hd_b(q) \text{ and } \mathit{secondary}(\mathit{tl}(q),j,B,n))
\end{array}$$

$$\begin{array}{l}
\mathit{secondary}(q,j,B,n) = \\
\quad \mathit{if}(\mathit{empty}(q), \\
\quad\quad eq(get_s(j +_n 1,B),0) \text{ or } eq(get_s(j +_n 1,B),1) \text{ or} \\
\quad\quad ((eq(get_s(j +_n 1,B),4) \text{ or } eq(get_s(j +_n 1,B),5)) \text{ and not } get_b(j +_n 1,B)), \\
\quad\quad \text{not } hd_b(q) \text{ and } \mathit{primary}(\mathit{tl}(q),j,B,n))
\end{array}$$

This rather complex looking invariant captures the protocol property that there are two kinds of messages sent: *active messages* which are received by the following process as values on the e variable and which can subsequently replace the d value of the process. The *passive messages* are received as values on the f variables (state 3) and will not replace the original d value of the process.

The *Alt* property guarantees that an active message can never be received as a passive message (or vice versa), i.e., neighbour processes and channels are always kept synchronised by the protocol. Inv_6 is needed to establish the invariants Inv_8 and Inv_4, to guarantee that identifiers are neither duplicated nor lost.

In order to prove Inv_6 the following two lemmas are useful. Lemma_{61} allows to prove that $\mathit{secondary}(get_q(j,B),j,B)$ implies $\mathit{secondary}(get_q(j,B'),j,B)$, assuming that the channels $get_q(j,B)$ and $get_q(j,B')$ are identical.

$$
\begin{array}{l}
Lemma_{61}(B, B', n) = \\
\quad \forall_{j<n} eq(get_q(j, B), get_q(j, B')) \rightarrow \\
\quad\quad (\mathit{secondary}(get_q(j, B), j, B, n) \rightarrow \mathit{secondary}(get_q(j, B'), j, B', n)) \leftrightarrow \\
\quad\quad (\mathit{even}(\mathit{size}(get_q(j, B))) \rightarrow (\mathit{secondary}(q_0, j, B, n) \rightarrow \mathit{secondary}(q_0, j, B', n))) \\
\quad\quad \text{and} \\
\quad\quad (\text{not}(\mathit{even}(\mathit{size}(get_q(j, B))))) \rightarrow (\mathit{primary}(q_0, j, B, n) \rightarrow \mathit{primary}(q_0, j, B', n))))
\end{array}
$$

Similarly, $Lemma_{62}$ is convenient for proving that the transition from state 3 to state 4 preserves the invariant:

$$
\begin{array}{l}
Lemma_{62}(B, n) = \\
\quad \forall_{j<n}(\mathit{Alt}(j, B, n) \text{ and not } \mathit{empty}(j, B) \text{ and } \mathit{secondary}(get_q(j, B), j, B, n) \text{ and} \\
\quad\quad eq(get_s(j +_n 1, B), 3) \rightarrow \text{not } toe_b(j, B))
\end{array}
$$

Consecutive identifiers are distinct Inv_7 guarantees that when an identifier in an active position follows an identifier in a passive position, the identifiers are distinct. This invariant depends on Inv_5 and Inv_6.

$$Inv_7(B, n) = \forall_{j<n} \mathit{Cons}(j, B, n)$$

where

$$
\begin{array}{l}
\mathit{Cons}(j, B, n) = \\
\quad \mathit{Cons}_q(get_q(j, B), j, B, n) \text{ and} \\
\quad \mathit{if}(eq(get_s(j, B), 5) \text{ and not } get_b(j, B), \\
\quad\quad \mathit{Neq}_q(get_d(j, B), get_q(j, B), j, B, n), \\
\quad\quad \mathit{if}(eq(get_s(j, B), 1) \text{ or } eq(get_s(j, B), 2), \mathit{Eq}_q(get_d(j, B), get_q(j, B), j, B, n), \mathsf{t})) \\
\mathit{Cons}_q(q, j, B, n) = \\
\quad \mathit{if}(\mathit{empty}(q), \mathsf{t}, \mathit{Cons}_q(tl(q), j, B, n) \text{ and } \mathit{if}(\mathit{hd}_b(q), \mathsf{t}, Neq_q(\mathit{hd}(q), \mathit{tl}(q), j, B, n)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Neq}(k, j, B, n) = \\
\quad ((eq(get_s(j, B), 2) \text{ or } eq(get_s(j, B), 3)) \text{ and not } eq(get_e(j, B), k)) \text{ or} \\
\quad (eq(get_s(j, B), 4) \text{ and } \mathit{Neq}_q(k, get_q(j, B, n), j, B, n)) \text{ or} \\
\quad (eq(get_s(j, B), 5) \text{ and } get_b(j, B) \text{ and not } eq(get_d(j, B), k)) \\
\mathit{Neq}_q(k, q, j, B, n) = \\
\quad \mathit{if}(\mathit{empty}(q), \mathit{Neq}(k, j +_n 1, B, n), \mathit{hd}_b(q) \text{ and not } eq(\mathit{hd}(q), k))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Eq}(k, j, B, n) = \\
\quad ((eq(get_s(j, B), 2) \text{ or } eq(get_s(j, B), 3)) \text{ and } eq(get_e(j, B), k)) \text{ or} \\
\quad (eq(get_s(j, B), 4) \text{ and } \mathit{Eq}_q(k, get_q(j, B, n), j, B, n)) \text{ or} \\
\quad (eq(get_s(j, B), 5) \text{ and } get_b(j, B) \text{ and } eq(get_d(j, B), k)) \\
\mathit{Eq}_q(k, q, j, B, n) = \mathit{if}(\mathit{empty}(q), \mathit{Eq}(k, j +_n 1, B, n), \mathit{hd}_b(q) \text{ and } eq(\mathit{hd}(q), k))
\end{array}
$$

Uniqueness of identifiers Inv_8 expresses the fact that identifiers can occur in at most one active position in the ring of processes. It is provable with the help of Inv_6.

$$Inv_8(B,n) = \forall_{k<n} Count(B,k,n) \leq 1$$

where

$$\begin{array}{l} Count(B,k,n) = \\ \quad \sum_{j<n} if(ActiveNode(k,j,B),1,0) \\ +\sum_{j<n} ActiveChanOcc(k,get_q(j,B)) \end{array}$$

$$\begin{array}{l} ActiveChanOcc(k,q) = \\ \quad if(empty(q),0,if(hd_b(q) \text{ and } eq(k,hd(q)),1,0) + ActiveChanOcc(k,tl(q))) \end{array}$$

Intuitively, the definition of *Count* counts the number of times an identifier occurs in an active position, i.e., in a position such that the identifier can be transmitted and received by another process and later replace the d value of that process. An identifier in an active position can either occur in a variable (*ActiveNode*) or in a channel (*ActiveChanOcc*).

Identifier travel creates relay processes Inv_9 points out that if two processes contain the same identifier (k) then the processes in between are guaranteed to be in state 4 and the connecting channels all empty. It is provable using Inv_8.

$$\begin{array}{l} Inv_9(B,n) = \\ \forall_{k<n}\forall_{i<n}(eq(get_s(i,B),1) \text{ or } eq(get_s(i,B),2)) \text{ and } eq(get_d(i,B),k) \rightarrow \\ \quad (\forall_{j<n} eq(get_s(j,B),0) \rightarrow \text{not } eq(get_d(j,B),k)) \text{ and} \\ \quad (\forall_{j<n} ActiveNode(k,j,B) \rightarrow empty(i,B) \text{ and } EmptyNodes(i,j,n,B)) \text{ and} \\ \quad (\forall_{j<n} ActiveChan(k,get_q(j,B)) \rightarrow \\ \qquad eq(hd(j,B),k) \text{ and } hd_b(j,B) \text{ and} \\ \qquad if(eq(i,j),\mathsf{t},eq(get_s(j,B),4) \text{ and } empty(i,B) \text{ and } EmptyNodes(i,j,n,B))) \end{array}$$

where

$$\begin{array}{l} EmptyNode(j,B) = eq(get_s(j,B),4) \text{ and } empty(j,B) \\ EmptyNodes(i,j,n,B) = \\ \quad if(i<j,\forall_{i<l<j} EmptyNode(l,B), \\ \qquad (\forall_{l<j} EmptyNode(l,B)) \text{ and } (\forall_{i<l<n} EmptyNode(l,B))) \end{array}$$

Convergence of the protocol

In this section we prove that the linear process X is convergent, i.e., that we can find a decreasing measure on the data parameter over the τ-steps in the X process operator. This result implies that all sequences of τ-steps are finite, which is a necessary condition for applying the Concrete Invariant Corollary. We prove that the function *Meas* defined below is a decreasing measure, and thus proving convergence.

$$
\begin{array}{l}
\mathit{Meas}(B,n) = \\
\quad \sum_{j<n}[\mathit{if}(eq(get_s(j,B),0),(n-get_d(j,B)+2)\,6\,n^3, \\
\qquad \mathit{if}(eq(get_s(j,B),1 \text{ or } eq(get_s(j,B),2),(1+n-get_d(j,B))\,6\,n^3+3\,n^3, \\
\qquad \mathit{if}(eq(get_s(j,B),3),(1+n-get_d(j,B))\,6\,n^3,0))))]+ \\
\quad \sum_{j<n}\sum_{k<\mathit{size}(get_q(j,B))} \mathit{Term}(j,B,k)+ \\
\quad \sum_{j<n} \mathit{if}(eq(get_s(j,B),5),1+\mathit{Term}(j+_n 1,B,\mathit{size}(get_q(j,B))),0).
\end{array}
$$

$$
\begin{array}{l}
\mathit{Term}(j,B,st) = \\
\quad \mathit{if}([eq(st,1) \text{ and } (eq(get_s(j,B),0) \text{ or } eq(get_s(j,B),1))] \text{ or} \\
\qquad [eq(st,0) \text{ and } get_s(j,B) \leq 3], \\
\qquad\quad 1, \\
\qquad\quad 2+\mathit{Term}(j+_n 1,B,\mathit{size}(get_q(j,B)))+st+ \\
\qquad\quad \mathit{if}(eq(get_s(j,B),5),1,0)-\mathit{if}(eq(get_s(j,B),1,3),1,0)).
\end{array}
$$

We have a sequence of theorems that are useful to show that $\mathit{Meas}(B,n)$ shows that all τ-sequences in X are finite.

Lemma 6.6.5. *If* $n>0$, $0 \leq j,k<n$ *and*

$$
\begin{array}{l}
st<\sum_{i=j}^{k}[\mathit{if}(eq(get_s(i,B),1) \text{ or } eq(get_s(i,B),3),1,0)-\mathit{if}(eq(get_s(i,B),5),1,0)- \\
\qquad \mathit{size}(get_q(i-_n 1,B))]+\mathit{size}(get_q(j-_n 1,B)).
\end{array}
$$

then

1. $\mathit{Term}(j,B,st) \leq 2(k-_n j)+1$.
2. *If* $get_s(j,B)=5$ *and* $B'=in_q(d,b,j,upd_s(4,j,B))$ *then* $\mathit{Term}(i,B',st)=\mathit{Term}(i,B,st)$.
3. *If* $get_s(j,B)=4$, $B'=\mathit{untoe}(j-_n 1,upd_s(5,j,B))$ *then* $\mathit{Term}(i,B',st) \leq \mathit{Term}(i,B,st+\mathit{if}(eq(i,j),1,0))$.

Proof. All statements are proven by induction on $(k-_n j)$. □

Corollary 6.6.6.

1. *For* $n>0$ *and* $0 \leq k<n$ *we find* $\mathit{Term}(k,B,st)<2n$ *provided* $st<\mathit{size}(get_q(k,B))$.
2. *If* $get_s(j,B)=5$, $B'=in_q(d,b,j,upd_s(4,j,B))$ *and* $st<\mathit{size}(get_q(i,B))$ *then*
   $\mathit{Term}(i,B',st) \leq \mathit{Term}(i,B,st)$.
3. *If* $get_s(j,B)=4$, $st<\mathit{size}(get_q(i,B))$, $i \neq j$, $B'=\mathit{untoe}(j-_n 1,upd_s(5,j,B))$ *then*
   $\mathit{Term}(i,B',st) \leq \mathit{Term}(i,B,st)$.

Proof. Respectively, instantiate case *1* of Lemma 6.6.5 with $j = k +_n 1$; case 1 with $k = l -_n 1$ and $j = l +_n 1$; case 2 with $j = k +_n 1$ and at last case 3 with $j = k +_n 1$. □

Lemma 6.6.7.

$$
\begin{aligned}
&get_s(j,B) = 0 \rightarrow \quad \mathit{Meas}(upd_s(1,j,B),n) + 3n^3 \leq \mathit{Meas}(B,n)\\
&get_s(j,B) = 1 \rightarrow \quad \mathit{Meas}(upd_s(2,j,B),n) = \mathit{Meas}(B,n)\\
&get_s(j,B) = 2 \rightarrow \quad \mathit{Meas}(upd_s(3,j,B),n) + 3n^3 \leq \mathit{Meas}(B,n)\\
&get_s(j,B) = 3 \rightarrow \quad \mathit{Meas}(upd_s(0,j,B),n) \leq \mathit{Meas}(B,n)\\
&get_s(j,B) = 3 \rightarrow \quad \mathit{Meas}(upd_s(4,j,B),n) < \mathit{Meas}(B,n)\\
&get_s(j,B) = 0 \rightarrow \quad \mathit{Meas}(in_q(get_d(j,B),b,j,B),n) < \mathit{Meas}(T,n) + 3n^3\\
&get_s(j,B) = 2 \rightarrow \quad \mathit{Meas}(in_q(get_e(j,B),b,j,B),n) < \mathit{Meas}(B,n) + 3n^3\\
&get_s(j,T) = 1 \rightarrow \quad \mathit{Meas}(\mathit{untoe}(j -_n 1,B),n) < \mathit{Meas}(B,n)\\
&get_s(j,B) = 3 \rightarrow \quad \mathit{Meas}(\mathit{untoe}(j -_n 1,B),n) < \mathit{Meas}(B,n)\\
&get_s(j,B) = 4 \rightarrow \quad \mathit{Meas}(upd_s(5,j,B),n) < \mathit{Meas}(B,n)\\
&get_s(j,B) = 5 \rightarrow \quad \mathit{Meas}(upd_s(4,j,B),n) < \mathit{Meas}(B,n)\\
&get_s(j,B) = 4 \rightarrow \quad \mathit{Meas}(in_q(get_d(j,B),b,j,B),n) < \mathit{Meas}(B,n)\\
&get_s(j,B) = 5 \rightarrow \quad \mathit{Meas}(\mathit{untoe}(j -_n 1,B),n) < \mathit{Meas}(B,n)
\end{aligned}
$$

Theorem 6.6.8. *X is convergent.*

Proof. This follows as with the help of Lemma 6.6.7 it is straightforward to see that $\mathit{Meas}(B,n)$ is a decreasing measure. □

Remark 6.6.9. The measure *Meas* is certainly not optimal. It suggest that the algorithm requires about $6n^4(n+2)$ actions to select a leader. This is a very rough measure; looking at the far sharper bound in [DKR82] suggests that the bound can actually be improved to $4n \log_2 n + 2n$ actions. However, we did not try this yet.

Final calculations

We now prove the following crucial lemma that links the *leader* action to X. But first we provide an auxiliary function that expresses that no process $j < n$ is in state 6.

Definition 6.6.10.

$$\mathit{nonsix}(B,n) = \forall_{j<n} \text{not } eq(get_s(j,B),6).$$

Lemma 6.6.11. *The invariants $\mathit{Inv}_1(B,n),\ldots,\mathit{Inv}_4(B,n)$ imply:*

$$
\begin{aligned}
&X(B,n) =\\
&\quad (\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta) \lhd FC(B,n) \rhd \tau\,(\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta).
\end{aligned}
$$

Proof. We show assuming the invariants $\mathit{Inv}_1(B,n),\ldots,\mathit{Inv}_4(B,n)$ that

$$\lambda B{:}\mathit{Table}, n{:}Nat.(\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta) \lhd FC(B,n) \rhd \tau\ (\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta)$$

is a solution for X in (I). As (I) is convergent, the lemma follows from the Concrete Invariant Corollary (see [BG94]). First suppose $FC(B,n)$ holds. This means that we must show that

$$\begin{array}{l}\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta = \\ \sum_{j:Nat} \mathit{leader}\,(\mathit{leader}\,\delta \lhd \mathit{nonsix}(\mathit{upd}_s(6,j,B)) \rhd \delta) \\ \lhd eq(\mathit{get}_s(j,B),2) \text{ and } eq(\mathit{get}_d(j,B),\mathit{get}_e(j,B)) \text{ and } j < n \rhd \delta.\end{array} \tag{6.68}$$

Note that it follows from $FC(B,n)$ that the other summands of (I) may be omitted. As $\mathit{nonsix}(\mathit{upd}_s(6,j,B)) = \mathsf{f}$, equation (6.68) reduces to:

$$\begin{array}{l}\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta = \\ \sum_{j:Nat} \mathit{leader}\,\delta \lhd eq(\mathit{get}_s(j,B),2) \text{ and } eq(\mathit{get}_d(j,B),\mathit{get}_e(j,B)) \text{ and } j < n \rhd \delta.\end{array} \tag{6.69}$$

Now assume $\mathit{nonsix}(B,n)$. From $FC(B,n)$ and $\mathit{Inv}_1(B,n)$ is follows that

$$\forall_{j<n} 1 \leq \mathit{get}_s(j,B) \leq 4. \tag{6.70}$$

First we show that $\exists_{j<n} eq(\mathit{get}_s(j,B),2)$ and $eq(\mathit{get}_d(j,B),\mathit{get}_e(j,B))$. Now suppose

$$\exists_{j<n} eq(\mathit{get}_s(j,B),1) \text{ or } eq(\mathit{get}_s(j,B),3).$$

Hence, using $\mathit{Inv}_2(B,n)$ and $\mathit{nproc}(B,1,n)+\mathit{nproc}(B,3,n) > 0$ and (6.70), it follows that $\mathit{qsizes}(B,n) > 0$. Hence, $\exists_{j<n} \mathit{size}(j -_n 1,B) > 0$. Hence, using the focus condition and $\mathit{Inv}_1(B,n)$:

$$\exists_{j<n} eq(\mathit{get}_s(j,B),2) \text{ and } eq(\mathit{get}_d(j,B),\mathit{get}_e(j,B)).$$

Now suppose

$$\text{not } \exists_{j<n} eq(\mathit{get}_s(j,B),1) \text{ or } eq(\mathit{get}_s(j,B),3).$$

Hence, using (6.70) it follows that

$$\forall_{j<n} eq(\mathit{get}_s(j,B),2) \text{ or } eq(\mathit{get}_s(j,B),4). \tag{6.71}$$

Now assume

$$\forall_{j<n} eq(\mathit{get}_s(j,B),4).$$

But this contradicts $\mathit{Inv}_4(B,n)$ in conjunction with $\mathit{Inv}_2(B,n)$. Hence, using (6.71) it follows that

$$\exists_{j<n} eq(\mathit{get}_s(j,B),2).$$

From this and $FC(B,n)$ it follows that

$$\exists_{j<n} eq(\mathit{get}_s(j,B),2) \text{ and } eq(\mathit{get}_d(j,B),\mathit{get}_e(j,B)).$$

Hence, using SUM3 the right-hand side of (6.69) has a summand

$$\mathit{leader}\,\delta. \tag{6.72}$$

But using some straightforward calculations (6.72) has the right-hand side of (6.69) as a summand. Hence, if $\mathit{nonsix}(B,n)$ then (6.69) is equivalent to

$$\mathit{leader}\,\delta = \mathit{leader}\,\delta$$

which is clearly a tautology. Now assume not $\mathit{nonsix}(B,n)$. Hence, $\exists_{j\leq n} eq(get_s(j,B),6)$. Using $\mathit{Inv}_3(B,n)$ it follows that

$$\forall_{j<n} eq(get_s(j,B),4) \text{ or } eq(get_s(j,B),6).$$

Hence (6.69) reduces to

$$\delta = \delta$$

which is clearly true. Now suppose the focus condition does not hold, i.e., not $FC(B,n)$. We find (where we use that $n>0$ and Milner's second τ-law (T2)):

$$\begin{array}{l}
\tau\,(\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta) = \\
\sum_{j:Nat} \tau\,(\mathit{leader}\,\delta \lhd \mathit{nonsix}(B,n) \rhd \delta) \lhd j < n \rhd \delta + \\
\sum_{j:Nat} \mathit{leader}\,\delta \\
\quad \lhd \mathit{nonsix}(B,n) \text{ and } eq(get_s(j,B),2) \text{ and} \\
\quad eq(get_d(j,B),get_e(j,B)) \text{ and } j<n \rhd \delta
\end{array} \tag{6.73}$$

Now note that it follows from $\mathit{Inv}_3(B,n)$ that if $\exists_{j<n} eq(get_s(j,B),2)$, then $\mathit{nonsix}(B,n)$. So, (6.73) reduces to:

$$\begin{array}{l}
\sum_{j:Nat} \tau(\mathit{leader}\delta \lhd \mathit{nonsix}(B,n) \text{and} j<n \rhd \delta) + \\
\sum_{j:Nat} \mathit{leader}\delta \lhd eq(get_s(j,B),2) \text{and} eq(get_d(j,B),get_e(j,B)) \text{and} j<n \rhd \delta = \\
(\sum_{j:Nat} \tau\,(\mathit{leader}\delta \lhd \mathit{nonsix}(B,n) \text{and} j<n \rhd \delta) \lhd \text{not } FC(B,n) \rhd \delta) + \\
\sum_{j:Nat} \mathit{leader}\delta \lhd eq(get_s(j,B),2) \text{ and } eq(get_d(j,B),get_e(j,B)) \text{and} j<n \rhd \delta = \\
\sum_{j:Nat} \tau\,(\mathit{leader}\delta \\
\quad \lhd \mathit{nonsix}(upd_s(1,j,in_q(get_d(j,B),j,B)),n) \rhd \delta) \lhd eq(get_s(j,B),0) \text{and} j<n \rhd \delta + \\
\sum_{j:Nat} \tau\,(\mathit{leader}\delta \lhd \mathit{nonsix}(\mathit{untoe}(j -_n 1, upd_e(toe(j -_n 1,B),j,upd_s(2,j,B))),n) \rhd \delta) \\
\quad \lhd eq(get_s(j,B),1) \text{and not } \mathit{empty}(j -_n 1,B) \text{ and } j<n \rhd \delta + \\
\sum_{j:Nat} \mathit{leader}\,(\mathit{leader}\delta \lhd \mathit{nonsix}(upd_s(6,j,B),n) \rhd \delta) \\
\quad \lhd eq(get_s(j,B),2) \text{and} eq(get_d(j,B),get_e(j,B)) \text{and} j<n \rhd \delta + \\
\sum_{j:Nat} \tau\,(\mathit{leader}\delta \lhd \mathit{nonsix}(upd_s(3,j,in(get_e(j,B),j,B)),n) \rhd \delta) \\
\quad \lhd eq(get_s(j,B),2) \text{and not } eq(get_d(j,B),get_e(j,B)) \text{and} j<n \rhd \delta + \\
\sum_{j:Nat} (\tau\,(\mathit{leader}\delta \lhd \mathit{nonsix}(\mathit{untoe}(j -_n 1, upd_d(get_e(j,B),j,upd_s(0,j,B))),n) \rhd \delta) \\
\qquad \lhd get_e(j,B) > max(get_d(j,B),toe(j -_n 1,B)) \rhd \\
\qquad \tau\,(\mathit{leader}\delta \lhd \mathit{nonsix}(\mathit{untoe}(j -_n 1, upd_s(4,j,B)),n) \rhd \delta)) \\
\quad \lhd eq(get_s(j,B),3) \text{and not } \mathit{empty}(j -_n 1,B) \text{and} j<n \rhd \delta + \\
\sum_{j:Nat} \tau\,(\mathit{leader}\delta \lhd \mathit{nonsix}(\mathit{untoe}(j -_n 1, upd_d(toe(j -_n 1,B),j,upd_s(5,j,B))),n) \rhd \delta)
\end{array}$$

$$\begin{array}{l} \lhd eq(get_s(j,B),4) \,\mathrm{and}\, \mathrm{not}\, \mathit{empty}(j -_n 1, B) \,\mathrm{and}\, j < n \rhd \delta + \\ \sum_{j:Nat} \tau\,(\mathit{leader}\delta \lhd \mathit{nonsix}(in_q(get_d(j,B),j,upd_s(4,j,B)),n) \rhd \delta) \\ \lhd eq(get_s(j,B),5) \,\mathrm{and}\, j < n \rhd \delta \end{array}$$

Because $FC(B,n) = \mathsf{f}$, nearly all the summands given above are equal to δ. □

Proving Theorem 6.6.1 Finally we are ready to prove that the main theorem of the paper holds, i.e.,

$$n > 0 \rightarrow \mathit{Spec}(n) = \tau\, \mathit{leader}\, \delta$$

Proof. Using Lemma 6.6.4 we know

$$\mathit{Spec}(n) = X(\mathit{init}(n), n).$$

From Lemma 6.6.11 it then follows that

$$\begin{array}{l} \mathit{Spec}(n) = \\ \quad (\mathit{leader}\, \delta \lhd \mathit{nonsix}(\mathit{init}(n), n) \rhd \delta) \\ \quad \lhd FC(\mathit{init}(n), n) \rhd \\ \quad \tau\,(\mathit{leader}\, \delta \lhd \mathit{nonsix}(\mathit{init}(n), n) \rhd \delta). \end{array}$$

However, $FC(\mathit{init}(n), n)$ is not true if $n > 0$ while $\mathit{nonsix}(\mathit{init}(n), n)$ is true. Therefore

$$n > 0 \rightarrow \mathit{Spec}(n) = \tau\, \mathit{leader}\, \delta$$

is true. □

6.6.3 Conclusion

We have outlined a formal proof of the correctness of the leader election or extrema finding protocol of Dolev, Klawe and Rodeh in μCRL. The proof is now ready to be proof checked conform [BBG93, GvdP93, KS94, Sel94, WWN99].

It is shown that process algebra, in particular μCRL, is suited to prove correctness of non-trivial protocols. A drawback of the current verification is that it is rather complex and lengthy. A possible lead towards improvement is given by Frits Vaandrager in [Vaa93], where by using the notion of confluency (see e.g. [Mil80]) one only needs to consider one trace to establish correctness. Currently we are formalising this notion in [GS95]. We expect that using this idea our proof can be simplified significantly.

6.6.4 An overview of the proof theory for μCRL

We provide here a very short account of the axioms that have been used. We also give the Concrete Invariant Corollary for referencing purposes.

All the process algebra axioms used to prove the leader election protocol can be found in Table 6.1–6.6. We do not explain the axioms (see [BW90, BG94, GP94]) but only include them to give an exact and complete overview of the axioms that we used. Actually, the renaming axioms are superfluous, but have been included for completeness.

Besides the axioms we have used the Concrete Invariant Corollary [BG94] that says that if two processes p and q can be shown a solution of a well-founded recursive specification using an invariant, then p and q are equal, for all starting states where the invariant holds. It is convenient to use linear process operators, which are functions that transform a parameterised process into another parameterised process. If such an operator is well-founded, it has a unique solution, and henceforth defines a process. Note that if a linear process operator is applied to a process name, it becomes a process in Unity format.

Definition 6.6.12. A linear process operator Ψ is an expression of the form

$$\lambda p{:}D{\to}\mathbb{P}.\lambda d{:}D.\ \Sigma_{i\in I}\Sigma_{e_i:D_i} c_i(f_i(d,e_i))\cdot p(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta + \Sigma_{i\in I'}\Sigma_{e_i:D'_i} c'_i(f'_i(d,e_i)) \lhd b'_i(d,e_i) \rhd \delta$$

for some finite index sets I, I', actions c_i, c'_i, data types D_i, D'_i, D_{c_i} and $D_{c'_i}$, functions $f_i : D{\to}D_i{\to}D_{c_i}$, $g_i : D{\to}D_i{\to}D$, $b_i : D{\to}D'_i{\to}\mathbf{Bool}$, $f'_i : D{\to}D'_i{\to}D_{c'_i}$, $b'_i : D{\to}D'_i{\to}\mathbf{Bool}$.

Definition 6.6.13. A linear process operator (*LPO*) Ψ written in the form above is called *convergent* iff there is a well-founded ordering $<$ on D such that $g_i(d,e_i) < d$ for all $d \in D$, $i \in I$ and $e_i \in D_i$ with $c_i = \tau$ and $b_i(d,e_i)$.

Corollary 6.6.14 *(Concrete Invariant Corollary). Assume*

$$\Phi = \lambda p{:}D{\to}\mathbb{P}.\lambda d{:}D.\ \Sigma_{j\in J}\Sigma_{e_j:D_j} c_j(f_j(d,e_j))\cdot p(g_j(d,e_j)) \lhd b_j(d,e_j) \rhd \delta + \Sigma_{j\in J'}\Sigma_{e_j:D'_j} c'_j(f'_j(d,e_j)) \lhd b'_j(d,e_j) \rhd \delta$$

is a LPO. If for some predicate $I : D{\to}\mathbf{Bool}$

$$\lambda pd.\Phi pd \lhd I(d) \rhd \delta \text{ is convergent, and}$$
$$I(d) \wedge b_j(d,e_j) \to I(g_j(d,e_j)) \text{ for all } j \in J,\ d \in D \text{ and } e_j \in D_j,$$

i.e. I *is an invariant of* Φ*, and for some* $q : D{\to}\mathbb{P}$, $q' : D{\to}\mathbb{P}$ *we have*

$$I(d) \to q(d) = \Phi q d,$$
$$I(d) \to q'(d) = \Phi q' d,$$

then

$$I(d) \to q(d) = q'(d).$$

A1	$x + y = y + x$	CF	$n(\bar{t}) \mid m(\bar{t}) = \begin{cases} \gamma(n,m)(\bar{t}) & \text{if } \gamma(n,m)\downarrow \\ \delta & \text{otherwise} \end{cases}$
A2	$x + (y + z) = (x + y) + z$		
A3	$x + x = x$		
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$		
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$		
A6	$x + \delta = x$		
A7	$\delta \cdot x = \delta$	CD1	$\delta \mid x = \delta$
		CD2	$x \mid \delta = \delta$
CM1	$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$	CT1	$\tau \mid x = \delta$
CM2	$a \lfloor\!\lfloor x = a \cdot x$	CT2	$x \mid \tau = \delta$
CM3	$a \cdot x \lfloor\!\lfloor y = a \cdot (x \parallel y)$		
CM4	$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	DD	$\partial_H(\delta) = \delta$
CM5	$a \cdot x \mid b = (a \mid b) \cdot x$	DT	$\partial_H(\tau) = \tau$
CM6	$a \mid b \cdot x = (a \mid b) \cdot x$	D1	$\partial_H(n(\bar{t})) = n(\bar{t})$ if $n \notin H$
CM7	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	D2	$\partial_H(n(\bar{t})) = \delta$ if $n \in H$
CM8	$(x + y) \mid z = x \mid z + y \mid z$	D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$
CM9	$x \mid (y + z) = x \mid y + x \mid z$	D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$

Table 6.1: The axioms of ACP in μCRL.

$(x \lfloor\!\lfloor y) \lfloor\!\lfloor z = x \lfloor\!\lfloor (y \parallel z)$	$(x \mid y) \mid z = x \mid (y \mid z)$
$x \parallel \delta = x\delta$	$x \mid (ay \lfloor\!\lfloor z) = (x \mid ay) \lfloor\!\lfloor z$
$x \mid y = y \mid x$	$x \mid (y \mid z) = \delta$ Handshaking

Table 6.2: Axioms of Standard Concurrency (SC).

TID	$\tau_I(\delta) = \delta$	
TIT	$\tau_I(\tau) = \tau$	
TI1	$\tau_I(n(\bar{t})) = n(\bar{t})$	if $n \notin I$
TI2	$\tau_I(n(\bar{t})) = \tau$	if $n \in I$
TI3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	
TI4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	

Table 6.3: Axioms for abstraction.

SUM1	$\sum_{d:D}(p) = p$	if d not free in p
SUM2	$\sum_{d:D}(p) = \sum_{e:D}(p[e/d])$	if e not free in p
SUM3	$\sum_{d:D}(p) = \sum_{d:D}(p) + p$	
SUM4	$\sum_{d:D}(p_1 + p_2) = \sum_{d:D}(p_1) + \sum_{d:D}(p_2)$	
SUM5	$\sum_{d:D}(p_1 \cdot p_2) = \sum_{d:D}(p_1) \cdot p_2$	if d not free in p_2
SUM6	$\sum_{d:D}(p_1 \between p_2) = \sum_{d:D}(p_1) \between p_2$	if d not free in p_2
SUM7	$\sum_{d:D}(p_1 \mid p_2) = \sum_{d:D}(p_1) \mid p_2$	if d not free in p_2
SUM8	$\sum_{d:D}(\partial_H(p)) = \partial_H(\sum_{d:D}(p))$	
SUM9	$\sum_{d:D}(\tau_I(p)) = \tau_I(\sum_{d:D}(p))$	
SUM11	$\dfrac{\mathcal{D}}{\dfrac{p_1 = p_2}{\sum_{d:D}(p_1) = \sum_{d:D}(p_2)}}$	provided d not free in the assumptions of $\mathcal{D}$

Table 6.4: Axioms for summation.

COND1	$x \lhd \mathsf{t} \rhd y$	$=$	x
COND2	$x \lhd \mathsf{f} \rhd y$	$=$	y
BOOL1	$\neg(\mathsf{t} = \mathsf{f})$		
BOOL2	$\neg(b = \mathsf{t}) \to b = \mathsf{f}$		

Table 6.5: Axioms for the conditional construct and $\mathbf{Bool}$.

B1	$x\,\tau$	$=$	x
B2	$\tau\,x$	$=$	$\tau\,x + x$

Table 6.6: Some τ-laws.

6.6.5 Data types

Booleans

sort $\mathbf{Bool}$
cons $\mathsf{t}, \mathsf{f} :\to \mathbf{Bool}$
func $\mathrm{not} : \mathbf{Bool} \to \mathbf{Bool}$
$\mathrm{and}\,, \mathrm{or}\,, \mathit{eq} : \mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$
$\mathit{if} : \mathbf{Bool} \times \mathbf{Bool} \times \mathbf{Bool} \to \mathbf{Bool}$
var $b, b' : \mathbf{Bool}$
rew $\mathrm{not}\ \mathsf{t} = \mathsf{f}$
$\mathrm{not}\ \mathsf{f} = \mathsf{t}$
$\mathsf{t}\ \mathrm{and}\ b = b$
$\mathsf{f}\ \mathrm{and}\ b = \mathsf{f}$
$\mathsf{t}\ \mathrm{or}\ b = \mathsf{t}$
$\mathsf{f}\ \mathrm{or}\ b = b$
$\mathit{eq}(\mathsf{t}, \mathsf{t}) = \mathsf{t}$
$\mathit{eq}(\mathsf{f}, \mathsf{f}) = \mathsf{t}$
$\mathit{eq}(\mathsf{t}, \mathsf{f}) = \mathsf{f}$
$\mathit{eq}(\mathsf{f}, \mathsf{t}) = \mathsf{f}$
$\mathit{if}(\mathsf{t}, b, b') = b$
$\mathit{if}(\mathsf{f}, b, b') = b'$

Natural numbers

sort Nat
cons $0 :\to \mathit{Nat}$
$S : \mathit{Nat} \to \mathit{Nat}$

func $1, 2, 3, 4, 5, 6 :\rightarrow Nat$
$P : Nat \rightarrow Nat$
$\mathit{even} : Nat \rightarrow \mathbf{Bool}$
$+, -, *, \mathit{max} : Nat \times Nat \rightarrow Nat$
$\mathit{eq}, \geq, \leq, <, > : Nat \times Nat \rightarrow \mathbf{Bool}$
$\mathit{if} : \mathbf{Bool} \times Nat \times Nat \rightarrow Nat$
var $n, m : Nat$
rew $1 = S(0)$
$2 = S(1)$
$3 = S(2)$
$4 = S(3)$
$5 = S(4)$
$6 = S(5)$
$P(0) = 0$
$P(S(n)) = n$
$\mathit{even}(0) = \mathsf{t}$
$\mathit{even}(S(0)) = \mathsf{f}$
$\mathit{even}(S(S(n))) = \mathit{even}(n)$
$n + 0 = n$
$n + S(m) = S(n + m)$
$n - 0 = n$
$n - S(m) = P(n - m)$
$n * 0 = 0$
$n * S(m) = n + n * m$
$\mathit{max}(n, m) = \mathit{if}(n \geq m, n, m)$
$\mathit{eq}(0, 0) = \mathsf{t}$
$\mathit{eq}(0, S(n)) = \mathsf{f}$
$\mathit{eq}(S(n), 0) = \mathsf{f}$
$\mathit{eq}(S(n), S(m)) = \mathit{eq}(n, m)$
$n \geq 0 = \mathsf{t}$
$0 \geq S(n) = \mathsf{f}$
$n \geq S(m) = n \geq m$
$n \leq m = m \geq n$
$n > m = n \geq S(m)$
$n < m = S(n) \leq m$
$\mathit{if}(\mathsf{t}, n, m) = n$
$\mathit{if}(\mathsf{f}, n, m) = m$

Modulo arithmetic

func $\mathit{mod} : Nat \times Nat \rightarrow Nat$
$+, - : Nat \times Nat \times Nat \rightarrow Nat$
var $k, m, n : Nat$
rew $m \mathit{\ mod\ } 0 = m$
$m \mathit{\ mod\ } S(n) = \mathit{if}(m \geq S(n), m - S(n) \mathit{\ mod\ } S(n), m)$

$k +_n m = k + m \bmod n$
$k -_n m = \mathit{if}(k \bmod n \geq m \bmod n, k \bmod n - m \bmod n, n - m \bmod n - k \bmod n)$

Queues

We use two kind of queues which are subtlely different. The first is of sort $Queue$ with the usual operations. The second is of sort $Queue_b$ which is similar to $Queue$ except that a boolean is added for technical purposes. The specification of $Queue_b$ is given below. We do not present the data type $Queue$ here because it can be considered as a simple instance of $Queue_b$ as follows: omit the functions $\mathit{hd}_b, \mathit{toe}_b$ and remove all boolean arguments. For example, $\mathit{in} : Nat \times \mathbf{Bool} \times Queue_b \to Queue_b$ corresponds with $\mathit{in} : Nat \times Queue \to Queue$.

sort $Queue_b$
cons $q_0 :\to Queue_b$
$\mathit{in} : Nat \times \mathbf{Bool} \times Queue_b \to Queue_b$
func $\mathit{rem} : Nat \times Queue_b \to Queue_b$
$\mathit{tl}, \mathit{untoe} : Queue_b \to Queue_b$
$\mathit{con} : Queue_b \times Queue_b \to Queue_b$
$\mathit{hd}, \mathit{toe} : Queue_b \to Nat$
$\mathit{hd}_b : Queue_b \to \mathbf{Bool}$
$\mathit{toe}_b : Queue_b \to \mathbf{Bool}$
$\mathit{eq} : Queue_b \times Queue_b \to \mathbf{Bool}$
$\mathit{empty} : Queue_b \to \mathbf{Bool}$
$\mathit{test} : Nat \times Queue_b \to \mathbf{Bool}$
$\mathit{size} : Queue_b \to Nat$
$\mathit{if} : \mathbf{Bool} \times Queue_b \times Queue_b \to Queue_b$
var $d, e : Nat$
$b, c : \mathbf{Bool}$
$q, r : Queue_b$

rew $rem(d, q_0) = q_0$
$rem(d, \mathit{in}(e, b, q)) = \mathit{if}(\mathit{eq}(d, e), q, \mathit{in}(e, b, rem(d, q)))$
$tl(q_0) = q_0$
$tl(\mathit{in}(d, b, q)) = q$
$untoe(q_0) = q_0$
$untoe(\mathit{in}(d, b, q_0)) = q_0$
$untoe(\mathit{in}(d, b, \mathit{in}(e, c, q))) = \mathit{in}(d, b, untoe(\mathit{in}(e, c, q)))$
$con(q_0, q) = q$
$con(\mathit{in}(d, b, q), r) = \mathit{in}(d, b, con(q, r))$
$hd(q_0) = 0$
$hd(\mathit{in}(d, b, q)) = d$
$hd_b(q_0) = \mathsf{f}$
$hd_b(\mathit{in}(d, b, q)) = b$
$toe(q_0) = 0$

$toe(\mathit{in}(d, b, q_0)) = d$
$toe(\mathit{in}(d, b, \mathit{in}(e, c, q))) = toe(\mathit{in}(e, c, q))$
$toe_b(q_0) = \mathsf{f}$
$toe_b(\mathit{in}(d, b, q_0)) = b$
$toe_b(\mathit{in}(d, b, \mathit{in}(e, c, q))) = toe_b(\mathit{in}(e, c, q))$
$\mathit{eq}(q_0, q_0) = \mathsf{t}$
$\mathit{eq}(q_0, \mathit{in}(d, b, q)) = \mathsf{f}$
$\mathit{eq}(\mathit{in}(d, b, q), q_0) = \mathsf{f}$
$\mathit{eq}(\mathit{in}(d, b, q), \mathit{in}(e, c, r)) = \mathit{eq}(d, e)$ and $\mathit{eq}(b, c)$ and $\mathit{eq}(q, r)$
$\mathit{empty}(q) = \mathit{eq}(size(q), 0)$
$test(d, q_0) = \mathsf{f}$
$test(d, \mathit{in}(e, b, q)) = \mathit{eq}(d, e)$ or $test(d, q)$
$size(q_0) = 0$
$size(\mathit{in}(d, b, q)) = S(size(q))$
$\mathit{if}(\mathsf{t}, q, r) = q$
$\mathit{if}(\mathsf{f}, q, r) = r$

Protocol states

sort *Table*
cons $t_0 :\rightarrow \mathit{Table}$
$in : Nat \times Nat \times Nat \times Nat \times \mathbf{Bool} \times Queue_b \times \mathit{Table} \rightarrow \mathit{Table}$
func $\mathit{init} : Nat \rightarrow \mathit{Table}$
$get_d, get_e, get_s : Nat \times \mathit{Table} \rightarrow Nat$
$get_b : Nat \times \mathit{Table} \rightarrow \mathbf{Bool}$
$get_q : Nat \times \mathit{Table} \rightarrow Queue_b$
$upd_d, upd_e, upd_s : Nat \times Nat \times \mathit{Table} \rightarrow \mathit{Table}$
$upd_b : \mathbf{Bool} \times Nat \times \mathit{Table} \rightarrow \mathit{Table}$
$upd_q : Queue_b \times Nat \times \mathit{Table} \rightarrow \mathit{Table}$
$test : Nat \times \mathit{Table} \rightarrow \mathbf{Bool}$
$in_q : Nat \times \mathbf{Bool} \times Nat \times \mathit{Table} \rightarrow \mathit{Table}$
$hd : Nat \times \mathit{Table} \rightarrow Nat$
$hd_b : Nat \times \mathit{Table} \rightarrow \mathbf{Bool}$
$hd_i : \mathit{Table} \rightarrow Nat$
$toe : Nat \times \mathit{Table} \rightarrow Nat$
$toe_b : Nat \times \mathit{Table} \rightarrow \mathbf{Bool}$
$\mathit{untoe} : Nat \times \mathit{Table} \rightarrow \mathit{Table}$
$\mathit{empty} : Nat \times \mathit{Table} \rightarrow \mathbf{Bool}$
$\mathit{tl} : \mathit{Table} \rightarrow \mathit{Table}$
$\mathit{rem} : Nat \times \mathit{Table} \rightarrow \mathit{Table}$
$UniqueIndex : \mathit{Table} \rightarrow \mathbf{Bool}$
$\mathit{empty} : \mathit{Table} \rightarrow \mathbf{Bool}$
$\mathit{if} : \mathbf{Bool} \times \mathit{Table} \times \mathit{Table} \rightarrow \mathit{Table}$
var $d, e, s, v, i, j, n : Nat$
$B, B' : \mathit{Table}$

$b, b' : \mathbf{Bool}$
$q, q' : Queue_b$
rew $\mathit{init}(n) = \mathit{if}(eq(n, 0), t_0, in(n-1, \mathit{id}(n-1), 0, 0, \mathsf{f}, q_0, \mathit{init}(n-1)))$
$get_d(i, t_0) = 0$
$get_d(i, in(j, d, e, s, b, q, B)) = \mathit{if}(eq(i, j), d, get_d(i, B))$
$get_e(i, t_0) = 0$
$get_e(i, in(j, d, e, s, b, q, B)) = \mathit{if}(eq(i, j), e, get_e(i, B))$
$get_s(i, t_0) = 0$
$get_s(i, in(j, d, e, s, b, q, B)) = \mathit{if}(eq(i, j), s, get_s(i, B))$
$get_b(i, t_0) = \mathsf{f}$
$get_b(i, in(j, d, e, s, b, q, B)) = \mathit{if}(eq(i, j), b, get_b(i, B))$
$get_q(i, t_0) = q_0$
$get_q(i, in(j, d, e, s, b, q, B)) = \mathit{if}(eq(i, j), q, get_q(i, B))$
$upd_d(v, i, t_0) = in(i, v, 0, 0, \mathsf{f}, q_0, t_0)$
$upd_d(v, i, in(j, d, e, s, b, q, B)) =$
  $\mathit{if}(eq(i, j), in(j, v, e, s, b, q, B), in(j, d, e, s, b, q, upd_d(v, i, B)))$
$upd_e(v, i, t_0) = in(i, 0, v, 0, \mathsf{f}, q_0, t_0)$
$upd_e(v, i, in(j, d, e, s, b, q, B)) =$
  $\mathit{if}(eq(i, j), in(j, d, v, s, b, q, B), in(j, d, e, s, b, q, upd_e(v, i, B)))$
$upd_s(s, i, t_0) = in(i, 0, 0, s, \mathsf{f}, q_0, t_0)$
$upd_s(v, i, in(j, d, e, s, b, q, B)) =$
  $\mathit{if}(eq(i, j), in(j, d, e, v, b, q, B), in(j, d, e, s, b, q, upd_s(v, i, B)))$
$upd_b(b', i, t_0) = in(i, 0, 0, 0, b', q_0, t_0)$
$upd_b(b', i, in(j, d, e, s, b, q, B)) =$
  $\mathit{if}(eq(i, j), in(j, d, e, s, b', q, B), in(j, d, e, s, b, q, upd_b(b', i, B)))$
$upd_q(q', i, t_0) = in(i, 0, 0, 0, \mathsf{f}, q', t_0)$
$upd_q(q', i, in(j, d, e, s, b, q, B)) =$
  $\mathit{if}(eq(i, j), in(j, d, e, s, b, q', B), in(j, d, e, s, b, q, upd_q(q', i, B)))$
$test(i, t_0) = \mathsf{f}$
$test(i, in(j, d, e, s, b, q, B)) = eq(i, j)$ or $test(i, B)$
$\mathit{untoe}(i, B) = upd_q(\mathit{untoe}(get_q(i, B)), i, B)$
$hd(i, B) = hd(get_q(i, B))$
$hd_b(i, B) = hd_b(get_q(i, B))$
$hd_i(t_0) = 0$
$hd_i(in(j, d, e, s, b, q, B)) = j$
$toe(i, B) = toe(get_q(i, B))$
$toe_b(i, B) = toe_b(get_q(i, B))$
$\mathit{untoe}(i, B) = upd_q(\mathit{untoe}(get_q(i, B)), i, B)$
$\mathit{empty}(i, B) = \mathit{empty}(get_q(i, B))$
$\mathit{tl}(t_0) = t_0$
$\mathit{tl}(in(j, d, e, s, b, q, B)) = B$
$\mathit{rem}(i, t_0) = t_0$
$\mathit{rem}(i, in(j, d, e, s, b, q, B)) = \mathit{if}(eq(i, j), B, in(j, d, e, s, b, q, \mathit{rem}(i, B)))$
$UniqueIndex(t_0) = \mathsf{t}$
$UniqueIndex(in(j, d, e, s, b, q, B)) =$ not $test(j, B)$ and $UniqueIndex(B)$

$\mathit{empty}(t_0) = \mathsf{t}$
$\mathit{empty}(in(j, d, e, s, b, q, B)) = \mathsf{f}$
$\mathit{if}(\mathsf{t}, B, B') = B$
$\mathit{if}(\mathsf{f}, B, B') = B'$

6.7 Proof of Leader Election Protocols in ERLANG

This section continues the study of leader election protocols commenced with the μCRL based verification of Dolev, Klawe and Rodeh's leader election protocol in Section 6.6 and reports some preliminary conclusions. Here, in contrast, a class of these protocols, again for a unidirectional ring topology, is specified in ERLANG and logic.

We focus on the leader election algorithms again since they are a rather good benchmark of the ERLANG proof system, as the proofs involve reasoning about process spawning and ring structures of a bounded, but unknown, size. In addition these algorithm appear simple, but often possess surprisingly complex behaviour.

While the verification of the leader election protocol has not been fully completed we report on preliminary results, as the proof method is interesting.

6.7.1 Describing the Protocols in ERLANG

A few key differences between ERLANG and μCRL affect the specification of the protocols.

First, communication in ERLANG is asynchronous, with potentially unbounded channels, matching the assumptions made by the class of leader election protocols modelled here. Second, for communication in ERLANG a knowledge of the receiving process process identifier (*pid*) is necessary, whereas in μCRL communication is more abstract, taking place via synchronisation of actions decoupled from any notion of processes. In practise the process identifier based communication discipline of ERLANG makes it slightly harder to set up a ring communication structure.

Each node in the network is assumed to execute the same function, accepting three initial parameters: `Out` – the process identifier of the following process in the unidirectional network, `Leader` – the process identifier for the process to which a leader election announcement will be made, and `D` – a value representing the identity of the node. In the ERLANG formalisation the process identifier of the process executing the node function will be reused as its initial identity (`D`).

6.7.2 Setting up the Network Topology

The setting up of a ring like structure is provided by the ERLANG function `r`, shown below.

```
r(Fun,[Hd|Rst],Leader) ->
  Pnew = spawn(d,[Fun,Leader,Hd]),
  r1(Fun,Rst,Leader,Pnew,Pnew).

r1(Fun,[],Leader,Pstop,Pprev) -> Pstop!{out,Pprev};
r1(Fun,[Hd|Rst],Leader,Pstop,Pprev) ->
  Pnew = spawn(Fun,[Pprev,Leader,Hd]),
  r1(Fun,Rst,Leader,Pstop,Pnew).
```

```
d(Fun,Leader,D) ->
  receive
    {out,Out} -> Fun(Out,Leader,D)
  end.
```

The `r` function accepts three parameters, `Fun` is the name of the ERLANG function that describes the action of a ring node, *Ids* is a list of ERLANG values corresponding to identifiers for nodes (elements of the list should be unique), and `Leader` is the process identifier to which a message will be sent when a leader has been elected.

6.7.3 Defining the Network Functions

A large number of node functions for electing a leader are possible. Consider first a simple function named `tnode`:

```
tnode(Out,Leader,D) -> Out!D, tnodeB(Out,Leader,D).

tnodeB(Out,Leader,D) ->
  receive E ->
    if
      E==D -> Leader!D;
      E>D -> tnode(Out,Leader,E);
      E<D -> tnodeB(Out,Leader,D)
    end
  end.
```

Intuitively the function will filter incoming values, retransmitting only values greater than its previously stored value. A simple variant of `tnode` is to become inert once a smaller value is seen:

```
snode(Out,Leader,D) ->
  Out!D,
  receive E ->
    if
      E==D -> Leader!D;
      E>D -> snode(Out,Leader,E);
      E<D -> c(Out)
    end
  end.
```

The function `c` copies all elements read from its input queue to its process identifier argument.

```
c(Out) -> receive V -> Out!V, c(Out) end.
```

Dolev, Klawe and Rodeh's algorithm is described by the function `dnode`:

```
dnode(Out,Leader,D) ->
  Out!D,
  receive E ->
    if
      E==D -> Leader!D;
      true ->
        Out!E,
        receive F ->
          if
            E > D, E > F -> dnode(Out,Leader,E);
            true -> c(Out)
          end
        end
    end
  end.
```

6.7.4 Common Formulas

Below a number of commonly utilised constructs are defined:

$$\begin{array}{l} \mathit{alwaysEV}\ [\phi] \triangleq \\ \quad \mu X : \mathsf{erlangSystem} \rightarrow \mathsf{prop} \\ \qquad \phi \vee \left(\begin{array}{ll} & \langle\tau\rangle \mathit{true} \vee \exists P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.\langle P!V\rangle \mathit{true} \\ \wedge & [\tau](X\ [\phi]) \wedge \forall P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V](X\ [\phi]) \end{array} \right) \end{array}$$

$$\begin{array}{l} \mathit{alwaysEV}_\tau\ [\phi] \triangleq \\ \quad \nu X : \mathsf{erlangPid} \rightarrow \mathsf{erlangSystem} \rightarrow \mathsf{prop} \\ \qquad \phi \vee (\langle\tau\rangle \mathit{true} \wedge [\tau](X\ [\phi])) \end{array}$$

$$\mathit{silent} \triangleq \forall P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V]\mathit{false}$$

$$\mathit{stable} \triangleq [\tau]\mathit{false} \wedge \mathit{silent}$$

The macro formula $s : \mathit{alwaysEV}[\phi]$ expresses that along every path labelled by internal or output actions in the transition system of s there must eventually occur a state such that the predicate ϕ is satisfied. Similarly $\mathit{alwaysEV}_\tau[\phi]$ expresses a similar statement for τ labelled paths. The formula $s : \mathit{silent}$ expresses that s has no transition labelled by an output action. Finally a system s is stable if can neither perform an internal nor output action.

6.7.5 Main Correctness Property

The main correctness property (*cp*) states that, in the absence of input and after some internal negotiation, eventually one process will announce itself leader and thereafter

the ring network will become silent.

$$
\begin{array}{l}
cp : \mathsf{erlangPid} \rightarrow \mathsf{erlangSystem} \rightarrow \mathsf{prop} \triangleq \\
\quad \lambda L : \mathsf{erlangPid}. \\
\quad \mathit{alwaysEV}_{\tau} \left[\begin{array}{lll} & \exists V : \mathsf{erlangValue}.\langle L!V \rangle \mathit{true} \\ \wedge & \forall P : \mathsf{erlangPid}, V : \mathsf{erlangValue}.[P!V]\,(P = L \wedge \mathit{stable}) \\ \wedge & [\tau]\mathit{false} \end{array} \right]
\end{array}
$$

6.7.6 Proof Structure

The proof structure is the same for all protocols; the behaviour of a ring segment (but not the whole ring) is characterised, and then the composition of ring segments is considered, by showing that the composition of two segments yields another segment. This stepwise argument is the basis of an inductive argument to extend the ring up to its last element.

The next proof obligation is to show that when the ring is closed the desired end property *cp* holds. The argument here is about well-founded induction. We must find some decreasing measure to ensure that the complexity, in terms of states of processes and messages in transit, is constantly decreasing.

Before ring segments can be considered, however, the concrete ring creation procedure has to be studied. In the following proof sketch, however, we consider only the segment based induction argument.

Behaviour of Ring Segments A ring segment is described by the generic $state$ predicate:

$$
\begin{array}{l}
state \Rightarrow \\
\quad \lambda I : \mathsf{erlangPid}. O : \mathsf{erlangPid}. L : \mathsf{erlangPid}. V_l : \mathsf{erlangQueue}. O_l : \mathsf{erlangValue}. \\
\qquad [\,](state\ I\ O\ L\ V_l\ O_l) \\
\quad \wedge \quad \forall V.[I?V]\,(\forall {V_l}', O_l'.(upd\ V\ V_l\ {V_l}'\ O_l\ O_l') \Rightarrow (state\ I\ O\ L\ {V_l}'\ O_l')) \\
\quad \wedge \quad \forall O', V.[O'!V] \\
\qquad\qquad \left(\begin{array}{lll} & O' = O \wedge \exists O_l'.O_l = V \cdot O_l' \wedge (state\ I\ O\ L\ V_l\ O_l') \\ \vee & \exists V_a, V_b.O' = L \wedge V_l = V_a \cdot \{\texttt{leader}, V\} \cdot V_b \wedge \\ & \qquad (state\ I\ O\ L\ V_a \cdot V_b\ O_l) \end{array} \right) \\
\quad \wedge \quad O_l \neq \epsilon \ \Rightarrow\ \exists O', V.\mathit{alwaysEV}_{\tau}\ [\langle O'!V \rangle \mathit{true}] \\
\quad \wedge \quad \mathit{alwaysEV}\ [\mathit{stable}]
\end{array}
$$

Intuitively the $state$ predicate expresses the following facts of a ring segment: (i) a segment output values only to its designated output channels (P or L), (ii) if the output queue (O_l) is nonempty then in the absence of inputs the process always reaches a state in which some value is output, (iii) in the absence of input the process eventually halts.

The shape of the definition allows to consider input only in stable states, the idea resembles the concept of focus points in μCRL.

Characterisation of a ring segment state update The predicate that describes the reaction of a ring segment to an input message differs for each network function.

For the node function `tnode` the update predicate can be characterised:

$$
\begin{array}{l}
upd \Leftarrow \\
\quad \lambda V : \mathsf{erlangValue}.V_l : \mathsf{erlangQueue}.{V_l}' : \mathsf{erlangQueue}. \\
\quad O_l : \mathsf{erlangQueue}.{O_l}' : \mathsf{erlangQueue}. \\
\qquad V_l = \epsilon \Rightarrow {O_l}' = O_l \cdot V \wedge {V_l}' = V_l \\
\quad \wedge \quad \exists H_d, T_l.V_l = H_d \cdot T_l \wedge \\
\qquad \left(\begin{array}{l}
\neg isAtom\ H_d \Rightarrow {O_l}' = O_l \wedge {V_l}' = V_l \\
isAtom\ H_d \wedge \exists H_d, T_l.V_l = H_d \cdot T_l \wedge V = H_d \Rightarrow \\
\qquad {O_l}' = O_l \wedge {V_l}' = \{\texttt{leader}, V\} \\
isAtom\ H_d \wedge \exists H_d, T_l.V_l = H_d \cdot T_l \wedge V < H_d \Rightarrow \\
\qquad {O_l}' = O_l \wedge {V_l}' = V_l \\
isAtom\ H_d \wedge \exists H_d, T_l.V_l = H_d \cdot T_l \wedge V > H_d \wedge \\
\qquad \exists {V_l}''.(upd\ V\ T_l\ {V_l}''\ O_l\ {O_l}') \Rightarrow {V_l}' = V \cdot {V_l}''
\end{array} \right)
\end{array}
$$

In upd_l the value V is generalised to a queue Q of received values:

$$
\begin{array}{l}
upd_l \Leftarrow \\
\quad \lambda Q : \mathsf{erlangQueue}.V_l : \mathsf{erlangQueue}.{V_l}' : \mathsf{erlangQueue}. \\
\quad O_l : \mathsf{erlangQueue}.{O_l}' : \mathsf{erlangQueue}. \\
\qquad Q = \epsilon \Rightarrow {V_l}' = V_l \wedge {O_l}' = O_l \\
\quad \wedge \quad \exists H_d, T_l.Q = H_d \cdot T_l \wedge \exists {V_l}'', {O_l}''.(upd\ H_d\ V_l\ {V_l}''\ O_l\ {O_l}'') \Rightarrow \\
\qquad\quad (upd_l\ T_l\ {V_l}''\ {V_l}'\ {O_l}''\ {O_l}')
\end{array}
$$

The generalisation is needed for two purposes. First, to account for a generalisation of the queue of a single node. Second in the composition of two nodes, where there is a list of elements to output from one node to the other.

A basic fact about the upd predicates can easily be established: $upd\ V\ V_l\ {V_l}'\ O_l\ {O_l}'$ (and $upd_l\ Q\ V_l\ {V_l}'\ O_l\ {O_l}'$) is a function in the arguments V, V_l and O_l (and Q), i.e.,

$$
\begin{array}{l}
\forall V, V_l, O_l.\exists {V_l}', {O_l}'. \\
\quad upd\ V\ V_l\ {V_l}'\ O_l\ {O_l}' \wedge \\
\quad \forall {V_l}'', {O_l}''.upd\ V\ V_l\ {V_l}''\ O_l\ {O_l}'' \Rightarrow {V_l}' = {V_l}'' \wedge {O_l}' = {O_l}''
\end{array}
$$

The axioms (**L–upd–fun**), and (**L–upd_l–fun**) establishes the analogue for upd_l.

Base Case The overall proof structure will be an induction on the network structure. First we show the base case, that a process executing the node function `tnode(Out,Leader,D)` can be characterised as a ring segment:

$$O \neq L, O \neq P, L \neq P \vdash \langle \texttt{tnode(}O\texttt{,}L\texttt{,}P\texttt{)}, P, \epsilon \rangle : \mathit{state}\ P\ O\ L\ P\ P$$

Following the standard proof practise the queue ϵ must first be generalised (to later be able to discharge with respect to the greatest fixed point of upd after the input of a value):

$$\Gamma, \mathit{upd}_l\ Q\ P\ V_l\ P\ O_l \vdash \langle \texttt{tnode(}O\texttt{,}L\texttt{,}P\texttt{)}, P, Q \rangle : \mathit{state}\ P\ O\ L\ V_l\ O_l \quad (6.74)$$

where Q,D' and O_l are of the expected types (queue, a single value in a queue, and queue, respectively), and Γ are the distinctions $O \neq L, O \neq P, L \neq P$. Note that proof obligation $\mathit{upd}_l\ \epsilon\ P\ P\ P\ P$ must be proved to validate the generalisation, but this is a trivial consequence of the definition of upd_l.

Next, in the by now standard manner, the definition of state is approximated, unfolded, a few simple proof steps are applied, resulting in five goals:

$$\Gamma' \vdash \langle \texttt{begin}\ \ O\texttt{!}P\texttt{,}\ \ \texttt{tnode(}O\texttt{,}L\texttt{,}P\texttt{)}\ \ \texttt{end}, P, Q \rangle : \mathit{state}\ P\ O\ L\ V_l\ O_l \quad (6.75)$$

$$\Gamma', \mathit{upd}\ V\ V_l\ V_l{}'\ O_l\ O_l{}' \vdash \langle \texttt{tnode(}O\texttt{,}L\texttt{,}P\texttt{)}, P, Q \cdot V \rangle : \mathit{state}\ P\ O\ L\ V_l{}'\ O_l{}' \quad (6.76)$$

$$\Gamma' \vdash \langle \texttt{tnode(}O\texttt{,}L\texttt{,}P\texttt{)}, P, Q \rangle : [O'!V] \ldots \quad (6.77)$$

$$\Gamma' \vdash \langle \texttt{tnode(}O\texttt{,}L\texttt{,}P\texttt{)}, P, Q \rangle : \exists O', V. \mathit{alwaysEV}_\tau\ [\langle O'!V \rangle \mathit{true}] \quad (6.78)$$

$$\Gamma' \vdash \langle \texttt{tnode(}O\texttt{,}L\texttt{,}P\texttt{)}, P, Q \rangle : \mathit{alwaysEV}\ [\mathit{stable}] \quad (6.79)$$

where Γ' is $\Gamma, \kappa' < \kappa, \mathit{upd}_l\ Q\ P\ V_l\ P\ O_l$.

Treating the goals bottom-to-top, goal 6.79 is solved by introducing an explicit assumption on the finite size of the queue,

$$\begin{array}{l} \mathit{queue} : \mathsf{erlangQueue} \rightarrow \mathsf{prop} \Leftarrow \\ \quad \lambda Q : \mathsf{erlangQueue}. \\ \qquad Q = \epsilon \\ \quad \vee \quad \exists V : \mathsf{erlangValue}, Q1 : \mathsf{erlangQueue}, Q2 : \mathsf{erlangQueue}. \\ \qquad Q = Q1 \cdot V \cdot Q2 \wedge \mathit{queue}\ Q1 \cdot Q2 \end{array}$$

and then applying a model checking scripts. After the queue is emptied, eventually the process will halt and wait in its receive state (proof sketch omitted).

Next goal 6.78 is solved by application of the same model checking script since after an initial silent transition, P will be output to O. The third goal (6.77) is trivially true since no output transition is enabled. The fourth goal (6.76) is the input case and we expect to be able to discharge against the original sequent (6.74). For the instance check to succeed the assumption $\mathit{upd}_l\ Q \cdot V\ P\ V_l{}'\ P\ O_l{}'$ must be established. That is, the sequent

$$\mathit{upd}_l\ Q\ P\ V_l\ P\ O_l, \mathit{upd}\ V\ V_l\ V_l{}'\ O_l\ O_l{}' \vdash \mathit{upd}_l\ Q \cdot V\ P\ V_l\ P\ O_l{}'$$

(**L–*upd*–input**) must be proved. This is accomplished through a (fixed point) induction on the definition of upd_l on the left hand side, reducing Q, and matching the steps on the right hand side. Goal 6.75 remains. After unfolding the right-hand formula once again, and simplifying the goals according to the established pattern only the following sequent is not trivially proved:

$$\Gamma' \vdash \langle \texttt{tnodeB}\,(O\texttt{,}\,L\texttt{,}\,P)\,, P, Q\rangle : \exists O_l'.O_l = P \cdot O_l{}' \wedge (state\ I\ O\ L\ V_l\ O_l{}') \quad (6.80)$$

Using the lemma

$$upd_l\ Q\ P\ V_l\ P\ O_l \vdash \exists O_l{}'.O_l = P \cdot O_l{}'$$

(**L–*upd*–input**) the sequent can be rewritten into

$$\Gamma'\{P \cdot O_l{}'/O_l\} \vdash \langle \texttt{tnodeB}\,(O\texttt{,}\,L\texttt{,}\,P)\,, P, Q\rangle : state\ I\ O\ L\ V_l\ O_l{}' \quad (6.81)$$

After again unfolding the right-hand side and solving straightforward goals, two remain:

$$\Gamma'' \vdash \langle \texttt{receive}\ \ldots\ \ \texttt{end}, P, Q\rangle : state\ I\ O\ L\ V_l\ O_l{}' \quad (6.82)$$

$$\Gamma'' \vdash \langle \texttt{tnodeB}\,(O\texttt{,}\,L\texttt{,}\,P)\,, P, Q\rangle : O_l{}' \neq \epsilon\ \Rightarrow\ \exists O', V.\mathit{alwaysEV}_\tau\ [\langle O'!V\rangle \mathit{true}] \quad (6.83)$$

where $\Gamma'' = \Gamma'\{P \cdot O_l{}'/O_l\}$.

Induction Step The induction step is represented by the sequent

$$\begin{array}{l} S_1 : state\ I\ I'\ L\ V_{l_1}\ O_{l_1}, \\ S_2 : state\ I'\ O\ L\ V_{l_2}\ O_{l_2}, \\ upd_l\ O_{l_1}\ V_{l_2}\ V_{l_2}{}'\ O_{l_2}\ O_l \vdash \\ S_1 \parallel S_2 : state\ I\ O\ L\ V_{l_1} \cdot V_{l_2}{}'\ O_l \end{array} \quad (6.84)$$

The sequent expresses that, if two segments are coupled together by the output of one and the input of the other, and if the output list of the first process updates the state vector of the second in the manner prescribed by the assumption on upd_l above then the new composed segment satisfies the segment property.

Proving this goal involves the following steps of quite heavy reasoning with data: we approximate the formula $state$ on the right hand side a co-inductive argument will take place, essentially matching an action by the parallel composed system $S_1 \parallel S_2$ with a corresponding action by their components as ring segments. For the input case it is necessary to show that upd_l updates the combined state accordingly, after the input has caused a reduction in S_1, and as a result of later output, also in S_2. Output is easier as it is handled only by S_2.

Conclusion There are a number of preliminary conclusions to be made. First, the proof structure is quite regular and seems to permit elegant reasoning about a variety of leader election protocols for ring structures. The different election protocols yield different semantics states of ring segments, and different functions to update the semantic content of a ring segment.

Characterising a ring segment by the $state$ predicate seems natural. The predicate focuses on input and output behaviour and abstracts away from the internal computation required to convert input to output. These concerns are instead handled on the logical level where the abstract state of a segment is characterised. The drawback of the method is clear: a large number of results about functions that update the semantic state of a segment are needed for a reasonably high-level proof.

Chapter 7

Related Work

This chapter briefly surveys related approaches to solving the challenges encountered in this thesis. The basis for our effort is the formal semantics for ERLANG; in Section 7.1 comparable efforts in providing semantics for concurrent languages are discussed. The second section focuses on the proof system, and traces the evolution of proof systems for variants of Hennessy-Milner logic and the modal μ-calculus. Next, we consider works in which programming language semantics for concurrent languages are embedded in proof tools. Finally related developments for ERLANG are investigated. Naturally these areas overlap; for instance the embedding of a programming language in a proof tool of course requires a formal semantics.

An interesting observation is the impact of process algebras such as CSP [Hoa85], CCS [Mil89] and the π-calculus [MPW92] on all these areas; no doubt this is caused by the combination of a well-worked out theory, apparent simplicity, an algebra with few operators, and expressive power; the operators capture right operational abstractions. For instance, semantics for concurrent languages are often provided through a translation into the π-calculus; CCS is a popular choice for experimenting with proof systems for concurrency, and numerous works embed CCS and the π-calculus in various proof tools.

7.1 Formal Semantics for Concurrent Programming Languages

By now it is a straightforward activity to equip a programming language with some kind of formal semantics, and numerous textbooks are available on the topic, e.g. Winskel [Win93] and Nielson [NN92]. In this section we will briefly recall some efforts to provide formal semantics for languages that are comparable to ERLANG in the sense that they offer a notion of concurrency and support message passing. Further their primary use is as programming languages rather than for the specification of applications.

Three different approaches to provide a semantics for programming languages are widely recognised. An operational semantics provides an account of stepwise transformation of the states of a program, and the side effects a program causes [Plo81]. The second approach defines a semantics by translating the source language into a target one; in denotational semantics the target language is a mathematical formalism. A common occurrence in the study of semantics for concurrent languages is to translate into the π-calculus [MPW92]. The third type of semantics, axiomatic semantics, provides axioms and proof rules grounded in the syntactic constructs of the programming language under study, a founding example is Hoare logic [Hoa69].

The full range of semantic options was illustrated early in the case of POOL, a parallel object-oriented language, where variants of the language were treated both using a denotational semantics based on metric spaces [AdBKR89], a traditional operational semantics [ABKR86] and a translation into process algebra [Vaa90].

Facile [GMP89] and Concurrent ML [Rep93] (CML) are two early concurrent extensions of Standard ML which in a sense rival ERLANG as programming languages. Facile clearly borrows inspiration from process algebra: the basic communication mechanism is synchronous (blocking) where processes communicate over first-order typed channels. Similarly CML offers threads and synchronous communication as basic primitives. Numerous semantic treatments of Facile exist; Degano et al. [DPLT96] for instance develop a non-interleaving semantics for the language to analyse causal and locality aspects of Facile programs. Giacalone et al. [GMP90] develop an interleaving operational semantics for Facile, and consider a variant of (weak) bisimulation equivalence [Mil89]. The equivalence relation is relativised with respect to the notion of a window: the set of channels over which communication can take place, similar to the dual role of restriction in the π-calculus. Further the notion of action equality is also relativised: actions communicating higher-order objects are considered equivalent if the objects yield equivalent results, and have equivalent behaviour. Adopting a window-based scheme for our ERLANG semantics would provide one means for encoding private knowledge such as process identifiers of freshly spawned processes. However, an alternative, for future work, is to enrich the target language, i.e., ERLANG, with primitive constructs for expressing privacy like restriction in the case of the π-calculus.

Works covering aspects of formal semantics for Concurrent ML include [PR97, NN93, FH95, ANN98, FHJ96]. Consider for instance Nielson and Nielson [NN93] which extends the type system of Concurrent ML with communication behaviours and develops an operational semantics that takes these behaviours into account. The semantics is similar to ours in that the concerns of sequential evaluation, concurrent evaluation, and communication are treated separately. On the sequential level the notion of evaluation contexts, which are called reduction contexts in our work, are used to regulate the order of subexpression evaluation. In contrast to our approach primitives that cause environmental changes are not modelled on the sequential level, but are interpreted at the concurrency level to handle process spawning, channel creation and so on. The communicating behaviours can be seen as an instance of a process algebra, and the main result connects such an algebra with Concurrent ML.

Gurevich et al. have employed abstract state machines [Gur95] for specifying language semantics; treatments include an investigation into Java Concurrency [EGGP00]

and a proposed authoritative semantics for SDL-2000 (see discussion in Eschbach et al. [EGGP00]). As an example of the approach the SDL subset considered in Eschbach [EGGP00] is compiled into code for an abstract machine; this abstract machine is itself equipped with a rigorous executable semantics.

7.2 Logics and Proof Systems for Reasoning about Concurrent Systems

This section considers developments in tableau based proof systems for reasoning about concurrent systems using some modal logic. Predominantly in these works the modal logic extends Hennessy-Milner logic [HM80] with fixed point operators, resulting in the propositional μ-calculus [Koz83] (also referred to as the modal μ-calculus). The chosen specification language is frequently a variant of CCS. An excellent introduction to the area can be found in Stirling [Sti01].

Early work in the field include numerous algorithms and proof systems for the verification of finite-state systems [Sti85, Lar88, Win91, Cle90, SW91]. Larsen [Lar88] introduced the concept of local modal checking, i.e., algorithms that considers only the part of the state space reachable from an agent p which is necessary to determine whether a satisfaction $p : \phi$ holds. Another common topic is compositional proof systems, in order to naturally decompose the proof task, and as a means of dealing with the explosion in the number of states due to parallelism. Stirling [Sti85], for example, introduces proof rules that to split the task of verifying the satisfaction of a parallel composition $p_1 \parallel p_2 : \phi$ into the tasks of verifying the three goals $p_1 : \phi_1$ and $p_2 : \phi_2$ and $\phi_1 \parallel \phi_2 : \phi$, where the latter goal is to be interpreted as requiring to establish that any agents satisfying ϕ_1 and ϕ_2 by necessity satisfy also ϕ. This rule is in our proof system derivable from the TERMCUT proof rule. The main difference between the compositional proof system of Stirling [Sti85] and our work is that modalities are basic in Stirling, and thus a basic set of proof rules have to be provided for solving the goal $\phi_1 \parallel \phi_2 : \phi$ above whereas in contrast regard modalities as abbreviating operational semantics assertions. Other examples of compositional proof systems include [Win90, XL90, ASW94].

In 1991 Stirling and Walker presented a tableau system for local modal checking of the μ-calculus [SW91], based on the idea of introducing constants abbreviating fixed point formulas, and to permit unfolding of constants only once, as implemented by two rules (here written bottom-up instead of top-down):

$$\frac{s \vdash_{\Delta'} U}{s \vdash_{\Delta} \mu Z.A} \; \Delta' \text{ is } \Delta \cdot U = \mu Z.A$$

$$\frac{s \vdash_{\Delta} A[U/Z]}{s \vdash_{\Delta} U} \; \Delta(U) = \mu Z.A \text{ and no ancestor proof node is labelled by } s \vdash_{\Delta'} U$$

Sequents are decorated with sets of fixed points unfolded on a proof path. The first rule is a constant replacement rule: the least fixed point is replaced by a constant and

information about the replacement is stored in the set Δ'. The second rule unfolds a constant (e.g., the corresponding fixed point), under the condition that there is no ancestor proof node such that s was unfolded also under Δ. In other words, the rule provides a bound for the number of times a fixed point may be unfolded. In our proof system there is no rule providing such a bound. However, in the tool implementation a fixed point tagging scheme [Win91] is used as a heuristic for when further unfolding of a fixed point is futile, much to the same effect as the constant scheme of Stirling and Walker. In Stirling and Bradfield [BS92] the notion of local model checking was extended to the infinite state case by, among other changes, removing the side condition on the second rule above, and stipulating a global well-foundedness condition on the paths between nodes labelled by the least fixed point operator. An implementation of the proof method is documented in Bradfield [Bra93]. An alternative proof system developed by Andersen [And94] results from replacing the use of constants in [BS92] with the tagging approach of Winskel [Win91]. In the tagging approach instead of replacing fixed points with constants as Stirling and Walker, the states under which a fixed point was unfolded is kept in the unfolded fixed point itself.. As an example of the tagging approach, consider two rules from Andersen [And94] albeit in a different notation:

$$\nu 0 \quad \frac{U : \phi[\nu X\{V \cup U\}\phi/X]}{U : \nu X\{V\}\phi} \text{ if } U \not\subseteq V$$

$$\nu 1 \qquad \frac{}{U : \nu X\{V\}\phi} \text{ if } U \subseteq V$$

Informally U and V are sets of states, and a fixed point have associated with a set of states (the tag) under which it has been unfolded. Thus the rule $\nu 0$ expresses that further unfolding is useful if the set of states U is not included in V. Further the unfolding itself adds the set of states U to V and records it in the tag of the fixed point. In contrast, if $U \subseteq V$ then the greatest fixed point property holds of ϕ. The treatment of the least fixed point operator is more complicated, and involves an infinitary proof rule. Essentially the idea is to find an external well-founded relation on the set of states under which the least fixed point is unfolded, and to use it in the proof rule for unfolding a least fixed point resulting and as a result requiring an infinitary proof system. In contrast we employ a global induction scheme in the proof system. Further there is no external justification of a least fixed point induction scheme.

The above proof systems cater for non-value passing process algebras; an extension to a value-passing variant of CCS using a proof system based on the approach is considered in Gurov [Gur98]. Rather than basis the reasoning for concluding that a least fixed point holds on an inductive argument on the set of states under which it is unfolded, Gurov [Gur98] motivates discharge based on an inductive argument external to the proof system about the domain of values considered. Rathke [Rat97] also considers a local proof system using the tagging approach for model checking value-passing processes but here the underlying models for value-passing processes are symbolic graphs.

Dam [Dam98] considers the verification of infinite-state systems described in CCS using a compositional proof system. Apart from regular proof rules, the proof system

contains for each CCS operator a set of compositional proof rules, a cut-rule, and a global rule for discharging proof goals due to fixed point induction. The proof system scheme was later applied to ERLANG [DF98]. In Dam et al. [DFG98b] the proof system was improved by the explicit introduction of ordinal inequation, and a simplified global rule of discharge. The resulting proof system is included in this thesis, and has been also been adapted to the π-calculus in Dam [Dam01].

Simpson [Sim95] simplified and generalised the linking between operational semantics of process algebras (in the GSOS format) and Hennessy-Milner logic in typical sequent based proof system. Proof rules are provided for introducing modalities based on the transitional capabilities of agents, rather than on the syntactical shape of the agent under study. The paper inspired a number of papers by Dam et al; treating CCS [DG00a], Erlang [FG99] and in obtaining completeness results [DG00b].

7.3 Embedding Semantics of Concurrent Languages in Theorem Proving Tools

By now there are many studies where a concurrent programming or specification language has been embedded in a theorem proving tool, in order to permit formal and possibly mechanised analysis of the semantics of the language, or to provide a facility to analyse programs in the target language. Numerous proof tools (theorem provers, proof assistants, proof checkers, etc.) are by now available. Among the early efforts can be mentioned the Boyer Moore theorem prover [BM79] and the Automath system [dB68]. Several comparative studies exist that classify and compare contemporary tools [GH98, BFM99]. Here we will briefly describe some key features of a few tools that figure prominently in efforts to embed language semantics.

The Isabelle general theorem proving environment [Pau94] is founded on a fragment of typed higher-order logic. In the foundation, or meta-logic, other frameworks or object logics are represented, e.g., typed higher-order logic (Isabelle/HOL) and first-order logic (Isabelle/FOL). The principal proof method of Isabelle is through resolution. The HOL [Ge93] theorem prover is based on the LCF (Logic of Computable Functions) tradition [GMW79], representing propositions as abstract data types, and offering a few select base mechanisms (rules of definition) to soundly extend the underlying logic.

The NuPRL [CAB$^+$86], COQ [DFH$^+$93], and Automath [dB68] theorem provers are based in type theory, i.e., they regard a proposition as a type, and a proof as an inhabitant of the type. The PVS [ORR$^+$96] prover is based on classical higher-order typed logic, its type system extended with predicate subtypes and dependent types.

Apart from the underlying logical framework, many aspects influence the suitability of a particular proof system for the task of embedding a concurrent language. For instance, what level of automation is supported by a prover, are there any decision procedures (say for propositional logic), what is the exact nature of its specification logic, can user defined syntax be embedded in the prover (e.g., can user defined parsers and pretty-printers be added for ERLANG), how are theories specified, what is the imple-

mentation language and to what extent does it support convenient user defined higher-order proof procedures, and are convenient graphical or textual interfaces available that aim to provide assistance in selecting useful proof rules to apply.

Leaving this brief overview of contemporary proof tools, we turn to the task of giving an overview of previous efforts at embedding a concurrent language in a theorem prover. An early example of the embedding of a concurrent language in a theorem prover is represented by the formalisation of the Calculus of Communicating Systems (CCS) in Nuprl [CAB$^+$86] by Cleaveland and Panangaden [CP88].

The HOL [Ge93] theorem prover has been the vehicle for a number of formalisations. Camillieri [Cam90] has formalised the process algebra CSP (Communicating Sequential Processes) [Hoa85], Nesi [Nes93, Nes99] considers CCS and Melham [Mel94] the π-calculus. Taking the work on formalising value-passing CCS in HOL as an example, Nesi embeds the operational semantics of the value-free fragment of CCS as an inductively defined relation, together with various congruence notions over CCS agents. Then a mapping from the value-passing calculus to the value-free fragment is formalised. Together, the mapping and the semantics for the value-free calculus are used to prove the soundness of a direct characterisation of the value-passing semantics of CCS. The author then proceeds to derive the usual axioms for observation congruence from the embedded notion of observation congruence. Röckl [Röc00] also considers a formalisation of the π-calculus, albeit in Isabelle/HOL [Pau94]. A key difference compared to Melham (and for that matter also to the treatment of substitution in ERLANG in this thesis) is the treatment of substitution in the π-calculus : Melham explicitly defines a substitution function in HOL (a so called deep embedding of substitution) whereas Röckl reuses the underlying substitution mechanism of Isabelle in the encoding of π-calculus substitution (a shallow language embedding).

Lin developed the two general proof checking tools PAM [Lin95] and VPAM [Lin93] (for reasoning about value-passing processes) to support reasoning about process algebra specifications. The tools permit users to define their own process algebra, by specifying its syntax and the axioms that govern term equality. Recursive, possibly infinite, behaviours of processes are reasoned about using unique fixed point induction. Little, if any, support for mechanisation of proofs is available in PAM.

Another important line of investigation concerns the embedding of the μCRL [GP90] process algebra in the COQ proof assistant [DFH$^+$93]. Numerous papers and theses discuss details in the formalisation of μCRL [BBG97, Sel96], or the application of the resulting framework to the verification of μCRL specifications [BBG93, GvdP93, Kor94, KS96]. The style of proofs is equational; the proof theory of μCRL is embedded in COQ and proofs proceed by rewriting.

The previously mentioned efforts yield proofs that are largely manual or at best semi-automatic (there is little support for proof strategic decisions such as finding invariants). In contrast Monroy et al. [Mon97, MBG00] attempts to automate proofs of equality between value-passing, infinite-state and even parametric CCS specifications. The means of automation is by embedding the equational theory of CCS in the Clam proof planning tool [BvHHS90] and providing CCS specific heuristics to drive the proof process. Naturally the result is an incomplete proof method; still interesting results have been obtained.

Lately much attention has been targeted towards the formalisation of the Java programming language. Research groups at Munchen, Nijmegen, INRIA and elsewhere have developed semantics for subsets of Java and embedded them in various theorem provers [vdBHJP00, Hui00, NvOP00, BDJ+01]. Typically, these formalisations have focused on the sequential object-oriented part of the language rather than concurrent language features such as threads.

7.4 Semantics and Analysis Techniques for ERLANG

A number of approaches to defining the semantics of ERLANG exists. Barklund and Virding [BV99, Bar98] have given a semi-formal account of ERLANG 4.7.3 (and a proposed new version, referred to as STANDARD ERLANG). In careful wording the syntax, semantics and assumptions underlying the language implementations are explained. An initial, not completed, attempt to specify the semantics of ERLANG in the tradition of natural semantics was undertaken by Pettersson [Pet96]. The Formal Design Techniques group at the Swedish Institute of Computer Science has developed a number of formal semantics for various subsets of ERLANG. Initial attempts (see for example [DFG98a] and [DFG98b]) are seen in hindsight to be inelegant compared to the current formalisation; the separation between functional and concurrent behaviour is incomplete. For instance, in [DFG98b] no meaning is ascribed to a send expression except in the context of a process. An alternative approach to defining a semantics for (a smaller subset of) ERLANG is found in Huch [Huc99]. His semantics, in contrast to ours, relies heavily on contextual information. For example, there are no rules that describe the effect of executing a send expression outside the context of a receiving process. Thus, in a sense, the equality of processes can be compared only when considered as explicit parts of a closed system.

A number of works have applied model checking techniques to the verification of ERLANG code. Wiklander [Wik99] implemented a translation from a finite-state subset of ERLANG to Promela, the specification language of SPIN [Hol91, Hol97]. The major difficulties experienced were the translation of the ERLANG dynamic data types, i.e., the lists and tuples, into Promela constructs. A second difficulty lay in achieving an efficient implementation of the ERLANG receive statement using the rather different message passing constructs of Promela. Arts and Benac Earle [AB01] have implemented a tool for translating from ERLANG to μCRL [GP90], and applied it to the verification of a series of mutual exclusion protocols. The translation tool performs a number of non-trivial transformations. For example, higher-order functions are transformed into first-order ERLANG functions, and asynchronous calls using the generic server design pattern of ERLANG/OTP are translated into synchronous communication in μCRL, mimicking the concept of the design pattern rather than its implementation in ERLANG. Correctness properties are specified in the alternation free modal μ-calculus, and checked using the CÆSAR/ALDÉBARAN tool set [FGK+96] against the state spaces that are generated by the μCRL tool set [Wou01].n A difficulty with the approach to translate ERLANG into μCRL is the handling of fairness. The ERLANG language specification require process fairness, which μCRL (with regards to the parallel composition

operator) does not provide. As a result fairness assumptions must be explicitly stated in the correctness properties. Since the model checking tool of CÆSAR/ALDÉBARAN handles only the alternation-free μ-calculus the encoding is cumbersome.

Huch [Huc99] explores the verification of ERLANG programs using abstract interpretation and model checking. The interpretation is proved to preserve all paths of the standard operational semantics, but may introduce new ones due to possible introduction of non-determinism in the abstraction of (deterministic) branches. Since ERLANG is untyped the abstraction does not use types as abstract values but rather abstract patterns involving process identifiers and atoms that occur in a program. In the implementation a finite-state fragment of ERLANG is considered, where function call patterns are restricted, a finite number of processes are spawned, and restriction queues to a fixed size. As program logic linear temporal logic (LTL) is used, and because the abstraction yields a finite state transition system, model checking is decidable [VW86].

Finally Giesl and Arts [GA01] show how techniques for analysing the termination properties of conditional term rewriting systems (CRTS) are applicable to ERLANG. The paper considers the query lookup protocol of the Mnesia distributed database also analysed using our proof assistant tool EVT by Arts and Dam [AD99]. The authors transform the correctness properties of the protocol into a termination problem and then proceed to show that using refinements on dependency pair techniques [AG97], the protocol can be proven to terminate without manual intervention.

Chapter 8

Conclusion

8.1 Summary

This thesis has presented a number of interrelated results. First, we have developed a clean formal semantics for the ERLANG programming language used within the telecommunications industry to develop complex distributed applications. The semantics considers a real programming language, and addresses aspects of ERLANG which have rarely been treated in formalisations of other languages and systems, most notably error detection and recovery in connection with concurrency.

Next, we have presented a proof system for reasoning about applications programmed in ERLANG and about aspects of its formal semantics. One key feature of the proof system is the uniform treatment of program behaviour and program data; both aspects are captured through an encoding into first-order logic with explicit fixed point operators. Inductive and co-inductive proof arguments are handled through the application of a single proof rule that discharges goals through an inductive argument on ordinals, resulting from the approximation and unfolding of fixed point definitions.

A further key point of the proof system, or perhaps rather of the advocated proof method, is compositionality; splitting the correctness proof of a program into the subproblems of proving correctness properties of its subcomponents. In proofs about data this method is considered very natural; in proofs about behaviour the technique is required in order to analyse infinite state behaviour. Compositional proof rules are easily derived due to the clean separation between operational semantics and other proof rules. In fact it is possible to derive all the compositional reasoning schemes used in the thesis by application of the classical cut rule of Gentzen–style proof systems.

The proof system has been implemented in a proof assistant tool, the ERLANG Verification Tool (EVT) which offers substantial support for verifying ERLANG programs: high-level tactics for deriving next states using the transition relation, parsers and pretty-printers for ERLANG, and so on.

As an illustration of the framework a number of prototypical ERLANG programs have been verified with respect to correctness properties formulated in a specification

logic. Such correctness properties are permitted to describe both the states of a program and its actions. Parts of these proof have been checked using the ERLANG Verification Tool. Proofs are typically semi-automated: the proof assistant can correctly decide on many steps but strategic decisions, such as the exact nature of an inductive argument, or the decomposition of a proof task in smaller steps, require manual intervention.

8.2 Impact

The impact of this thesis falls into the two categories: achievements that are of direct concern primarily to other researchers in the field, and results that have an impact on practise in software production. The overall goal of this thesis is to bridge the gap between research and software practise, more concretely concerning the verification of software written in ERLANG.

This goal has necessitated development in a number of areas of research, with results of a much wider interest than the treatment of ERLANG. The scope of the effort represents a contribution in that it demonstrates feasibility: we have shown that it is possible to address a real programming language, with data, with concurrency, with message passing, and with error handling. No restriction to the finite state domain for purposes of verification has been made. The full scope of verification techniques from mathematics, such as inductive and co-inductive reasoning schemas, and compositional reasoning, is applicable to ERLANG programs in our framework. Finite-state verification subproblems can be automated, and the means of automation can be tailored to the particular problem at hand. For instance, a set of lemmas to perform symbolic computations over some data domain can be used to augment and guide the process of state exploration. Moreover the treatment of ERLANG has been on the level of code, rather than specifications. In a number of examples we have shown the feasibility of reasoning about program code, rather than about an abstract specification of its behaviour.

A first technical contribution is the demonstration that behaviour and data can be treated uniformly. In the thesis this is achieved by encoding both concerns through fixed point definitions, and by devising a powerful proof system in which the distinction between co-inductive and inductive proof schemas largely disappears. Furthermore, the method for checking the validity of fixed point induction is an interesting variation of existing schemes based on tagging or constant techniques. It is expected that the global nature of the method will provide a natural setting for work on discovering induction schemes.

In the verification of ERLANG programs we have found that compositional reasoning is a useful proof technique on many levels: purely functional reasoning involving classical post- and pre-conditions, reasoning about expressions with side effects, splitting parallel compositions to combat state explosion, and so on. As the thesis demonstrates all these compositional reasoning schemes can be traced back to instances of application of the classical cut rule in Gentzen style proof systems. The reason for this clean result is the concrete semantics based reasoning style adopted in the thesis. We hope that this thesis will inspire other efforts in program reasoning to follow a similar approach with regards to the interplay between operational semantics, specification

logics and proof system support for compositional reasoning.

Another contribution is the proof assistant tool itself. By making it publicly available we aim to stimulate further research in the exploration of program behaviour using a fixed point approach, not necessarily always addressing the ERLANG programming language. An example of this can be found in the European research project Verificard which is considering using the proof assistant to support compositional reasoning about JavaCard programs, a dialect of Java suitable for programming Smartcard applications.

The industrial impact of the work is also promising. As evidenced in a number of case studies it is feasible to address substantial programs. It is fair to say, however, that as in most other verification approaches including model checking, training is necessary before the verification technique and tool implementation can be used effectively. Perhaps the most promising scenario is for trained personnel to formally verify critical parts of the ERLANG runtime environment such as the process supervision manager that enables automatic restart of abnormally terminated processes, or prominent components of the ERLANG platform such as the Mnesia database manager. Reliable operation of almost any substantial ERLANG application depends on the correct functioning of these components, and there remain aspects in their operation under adverse conditions that are very difficult to test for. A formalisation, and verification, as made feasible through the existence of this framework would contribute to the overall trust in the underlying programming platform.

8.3 Future Work

The ERLANG language itself is changing: higher-order functions and records have been added to language implementations while the work reported here took place. New features are expected to be added; a formalisation effort for a living language can never end.

More concretely, the present formalisation of the language is far from complete. One particular class of applications which cannot be analysed using the current formalisation of ERLANG are the time critical systems. Other features, which the present formalisation does not adequately address, include distribution (the distribution of processes onto nodes), fairness (scheduling is fair, processes cannot starve forever) and the privacy of processes (fresh process identifiers should be unknown until they are communicated).

To facilitate more convenient proofs a number of aspects of the formalisation need further consideration. One crucial point concerns the current language semantics, which is a small-step one, and therefore provides a natural and easy treatment of concurrency. However, large fragments of typical ERLANG programs are usually side-effect-free. For the formal analysis of such code fragments a natural (big step) semantics can be more convenient, reducing a number of proof steps. The work of Gurov and Chugunov [GC00] facilitates reasoning about side-effect-free code fragments through the annotation of function definitions with post and pre-conditions.

A second concern is library code. The majority of non-trivial applications make use of standard library components for programming client-server applications or finite

state machines. To support efficient reasoning about programs using such components standard reasoning patterns need to be developed. As a solution, Arts and Noll [AN00] recast the asynchronous handshake between client and server utilising the client-server pattern as a synchronous operation, by adding specialised knowledge about the functions invoking the client-server pattern to the operational semantics of ERLANG.

Considering the current implementation of the proof system it is clear that we cannot hope to compete with mature proof assistants like Isabelle or PVS, for instance regarding theories for data. A worthwhile future task is then to study how to efficiently code the global rule of discharge in these proof assistants; or alternatively to transform arguments using a global discharge rule into local induction schemes. Another option is to factor the verification problem for ERLANG programs into two distinct parts: one concerning program behaviour and best conducted in our own proof assistant, and the other concerning general arguments about data and best addressed in a good first-order theorem proving tool.

The verification approach described in the thesis is certainly not applicable only to the ERLANG language. For instance, the European project VerifiCard is considering a related approach. There a compositional operational semantics with an associated proof system is being developed by Barthe, Gurov and Huisman [BGH01].

References

[AB01] T. Arts and C. Benac Earle. Development of a verified Erlang program for resource locking. In *Proc. FMICS 2001, GMD Report No.91*, pages 109–122, 2001.

[ABKR86] P. America, J. Bakker, J. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Conference Record of the 13th Symposium on Principles of Programming Languages*, pages 194–205, 1986.

[AD99] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99—Formal Methods, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 682–700. Springer, 1999.

[AdBKR89] P. America, J. de Bakker, J. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83, 1989.

[ADFG98] T. Arts, M. Dam, L. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In *Proc. CADE'98,* Lecture Notes in Artificial Intelligence*, vol. 1421, pp. 38–41*, 1998.

[AG97] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In *TAPSOFT: 7th International Joint Conference on Theory and Practice of Software Development*, 1997.

[AN00] T. Arts and T. Noll. Verifying generic Erlang client-server implementations. *Proc. IFL 2000*, Aachener Informatik-Berichte(00-7):387–402, 2000.

[And94] H. R. Andersen. On model checking infinite-state systems. In *In proceedings of Logical Foundations of Computer Science 1994, Lecture Notes in Computer Science 813*, pages 8–17. Springer-Verlag, 1994.

[ANN98] T. Amtoft, F. Nielson, and H. Nielson. Behaviour analysis and safety conditions: a case study in CML. In *FASE'98, LNCS 1382*, pages 255–269. Springer-Verlag, 1998.

[Arm97] J. Armstrong. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203. ACM Press, 1997.

[ASW94] H. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ-calculus. In *In Proceedings of LICS'94*, 1994.

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

[Bar98] J. Barklund. Specification of the STANDARD ERLANG programming language. Final draft (0.6), Ericsson Computer Science Laboratory, 1998.

[BBG93] M.A. Bezem, R. Bol, and J.F. Groote. A formal verification of the alternating bit protocol in the calculus of constructions (revised version). Technical Report 88, Logic Group Preprint Series, Utrecht University, March 1993. Original version appeared as M.A. Bezem and J.F. Groote: A formal verification of the alternating bit protocol in the calculus of constructions.

[BBG97] M. Bezem, R. N. Bol, and J. F. Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects of Computing*, 9(1):1–48, 1997.

[BDJ+01] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. M. de Sousa. A formal executable semantics of the JavaCard platform. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP 2001), LNCS 2028*. Springer, 2001.

[BFM99] J.P. Bodeveix, M. Filali, and C. A. Muñoz. A formalization of the B-Method in Coq and PVS. In *FM99: The World Congress in Formal Methods, User Group Meeting 2. The B-Method: Applying B in an Industrial Context: Tools, Lessons and Techniques*, September 1999.

[BG93] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. Technical Report 99, Department of Philosophy, Utrecht University, 1993. To appear in the Computer Journal Volume 37(4).

[BG94] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In *Proceedings of the CONCUR '94 Conference on Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer-Verlag, 1994.

[BGH01] G. Barthe, D. Gurov, and M. Huisman. Compositional specification and verification of control flow based security properties of multi–application programs. In *Proc. FTfJP'01*, 2001.

[BKKM95] J.J. Brunekreef, J.-P. Katoen, R.L.C. Koymans, and S. Mauw. Algebraic specification of dynamic leader election protocols in broadcast networks. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Proceedings of the workshop on Algebra of Communicating Processes ACP94, Workshops in Computing*, pages 338–358. Springer-Verlag, 1995.

[BKKM96] J.J. Brunekreef, J.-P. Katoen, R.L.C. Koymans, and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9(4):157–171, 1996.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[BR98] S. Blau and J. Rooth. AXD 301 - a new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.

[Bra93] J. C. Bradfield. A proof assistant for symbolic model-checking. In *Proceedings of CAV'92, LNCS 663*, pages 316–329. Springer-Verlag, 1993.

[Bru95] J.J. Brunekreef. Process specification in a UNITY format. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Proceedings of the workshop on Algebra of Communicating Processes ACP94, Workshops in Computing*, pages 319–337. Springer-Verlag, 1995.

[BS92] J. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.

[BT98] Y. Bertot and L. Thery. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7):161–194, February 1998.

[BV99] J. Barklund and R. Virding. ERLANG 4.7.3 reference manual. Draft (0.7), Ericsson Computer Science Laboratory, 1999.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 449*, pages 647–648. Springer-Verlag, 1990.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[CAB+86] R.L. Constable, S.F. Allen, H.M Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[Cam90] A. Camillieri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.

[CGJ+00] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 2000–030, Department of Information Technology, Uppsala University, Sweden, November 2000.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76, 1988.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison Wesley, 1988.

[CP88] R. Cleaveland and P. Panangaden. Type theory and concurrency. *Journal of Parallel Programming*, 17:153–206, 1988.

[CW96] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Däc00] B. Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. Technical report, Department of Teleinformatics, Royal Institute of Technology, Sweden, November 2000.

[Dam95] M. Dam. Compositional proof systems for model checking infinite state processes. In *Proc. CONCUR'95,* Lecture Notes in Computer Science, 962:12–26, 1995.

[Dam98] M. Dam. Proving properties of dynamic process networks. *Information and Computation*, 140:95–114, 1998.

[Dam01] M. Dam. Proof systems for π-calculus logics. In de Queiroz, editor, *Logics for Concurrency and Synchronisation*. Oxford University Press, 2001. To appear.

[dB68] N. de Bruijn. The mathematical language automath. In *Symposium on Automatic Demonstration, volume 125 of Lectures Notes in Mathematics*, pages 29–61. Springer-Verlag, 1968.

[DF98] M. Dam and L. Fredlund. On the verification of open distributed systems. In *Proc. of the ACM Symposium on Applied Computing*, pages 532–540, 1998.

[DFG98a] M. Dam, L. Fredlund, and D. Gurov. Compositional verification of Erlang programs. In *Proceedings of the 1998 International Workshop on Formal Methods for Industrial Critical Systems*, Amsterdam, The Netherlands, 1998.

[DFG98b] M. Dam, L. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In H. Langmaack, A. Pnueli, and W.-P. de Roever, editors, *Compositionality: the Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 150–185. Springer, 1998.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, 1993.

[DG00a] M. Dam and D. Gurov. Compositional verification of CCS processes. pages 2247–256, 2000.

[DG00b] M. Dam and D. Gurov. μ-calculus with explicit points and approximations. In: *Proc. FICS'2000*, 2000.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, (3):245–260, 1982.

[DPLT96] P. Degano, C. Priami, L. Leth, and B. Thomsen. Analysis of Facile Programs: A Case Study. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages*, volume 1192, pages 345–369, Stockholm, 1996. Springer-Verlag, Berlin.

[EGGP00] R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the formal semantics of design languages: a compilation approach using abstract state machines. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

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

[FG99] L. Fredlund and D. Gurov. A framework for formal reasoning about open distributed systems. In *Proc. ASIAN'99,* Lecture Notes in Computer Science, volume 1742, pages 87–100, 1999.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, 1996.

[FGN+] L. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *Software Tools for Technology Transfer*. submitted, conditionally accepted for publication after reviewing.

[FH95] W. Ferreira and M. Hennessy. Towards a semantic theory of CML (extended abstract). In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1995.

[FHJ96] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 201–212, Philadelphia, Pennsylvania, 24–26 1996.

[FW94] M. Fröhlich and M. Werner. The graph visualization system daVinci – a user interface for applications. Technical Report 5/94, Department of Computer Science; Universität Bremen, 1994.

[GA01] J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Journal of Applicable Algebra in Engineering, Communication and Computing*, 12(1):39–72, 2001.

[GC00] D. Gurov and G. Chugunov. Verification of Erlang programs: Factoring out the side-effect-free fragment. In *Proceedings of the 2000 International Workshop on Formal Methods for Industrial Critical Systems*, Berlin, Germany, April 2000.

[Ge93] M.J.C. Gordon and T.F.Melham (eds.). *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge Press, 1993.

[Gen69] G. Gentzen. *Investigations into logical deduction*. North Holland, 1969.

[GH98] D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 123–142, Canberra, Australia, September 1998. Springer-Verlag.

[GK93] J.F. Groote and H. Korver. A correctness proof of the bakery protocol in μCRL. Technical Report 80, Department of Philosophy, Utrecht University, 1993.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[GMP90] A. Giacalone, P. Mishra, and S. Prasad. Operational and algebraic semantics for facile: A symmetric integration of concurrent and functional programming. In *Proceedings of ICALP 90, LNCS 443*, pages 765–779. Springer-Verlag, 1990.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation,LNCS 78*. Springer-Verlag, New York, 1979.

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.

[GP94] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, Proceedings of the International Workshop on Semantics of Specification Languages. *Workshops in Computing*, pages 231–250. Springer-Verlag, 1994.

[GS95] J.F. Groote and M.P.A. Sellink. Confluency for process verification. In *Proceedings of the CONCUR '95 Conference on Concurrency Theory, volume 962 of Lecture Notes in Computer Science*, pages 204–218. Springer-Verlag, 1995.

[Gur95] Y. Gurevich. Evolving algebras: Lipari guide. In E. Borger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Gur98] D. Gurov. *Specification and Verification of Communicating Systems with Value Passing*. PhD thesis, Dept. of Computer Science, University of Victoria, 1998.

[GvdP93] J.F. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. a case study in computer checked algebraic verification. Technical Report 100, Department of Philosophy, Utrecht University, 1993.

[HM80] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proceedings of ICALP, Lecture Notes in Computer Science*, volume 85, pages 295–309, 1980.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.

[Hol97] G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, (23):279–295, 1997.

[Huc99] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, 1999.

[Hui00] M. Huisman. *Reasoning about Java Programs in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2000.

[JPS00] E. Johansson, M. Pettersson, and K. Sagonas. A high performance Erlang system. In *In proceedings of ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP 2000)*, 2000.

[Kor94] H. Korver. *Protocol Verification in* μCRL. PhD thesis, University of Amsterdam, 1994.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.

[KS94] H. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In *Proceedings of the* 2^{nd} *International Symposium on Theoretical Aspects of Computer Software, Sendai, Japan*, volume 789 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 1994.

[KS96] H. Korver and A. Sellink. On automating process algebra proofs. Technical Report 154, Department of Philosophy, Utrecht University, 1996.

[Lar88] K. G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In M. Dauchet and M. Nivat, editors, *Proceedings 13th Coll. on Trees in Algebra and Programming, CAAP'88, Nancy, France, 21–24 March 1988*, volume 299, pages 215–230. Springer-Verlag, Berlin, 1988.

[Lin93] H. Lin. A verification tool for value-passing processes. In *Proceedings of the 13th International Symposium on Protocol Specification, Testing and Verification*. North-Holland, 1993.

[Lin95] H. Lin. PAM: A process algebra manipulator. *Formal Methods in System Design: An International Journal*, 7(3):243–259, November 1995.

[Lin96] A. Lindgren. A prototype of a soft type system for Erlang. Technical Report 91, Computing Science Department, Uppsala University, April 1996.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[MBG00] R. Monroy, A. Bundy, and I. Green. Planning proofs of equations in CCS. *Automated Software Engineering*, 7(3):263–304, 2000.

[Mel94] T.F. Melham. A mechanized theory of the pi-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.

[Mil80] R. Milner. A Calculus of Communicating Systems. Springer-Verlag, 1980. Volume 92 of *Lecture Notes in Computer Science*.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Mon97] R. Monroy. *Planning proofs of correctness of CCS systems*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1997.

[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.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[MTH97] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML – Revised*. MIT Press, 1997.

[MW97] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 136–149, Amsterdam, The Netherlands, June 1997.

[Nes93] M. Nesi. Value-passing CCS in HOL. In *Proc. Higher Order Logic Theorem Proving and its Applications : 6th International Workshop,* Lecture Notes in Computer Science*, vol. 780, pp. 352–365*, 1993.

[Nes99] M. Nesi. Formalising a value-passing calculus in HOL. *Formal Aspects of Computing*, 11(2):160–199, 1999.

[NN92] H. Nielson and F. Nielson. *Semantics with Applications*. Wiley and Sons, Chichester, 1992.

[NN93] F. Nielson and H. Nielson. From CML to process algebras. In *Proceesings of CONCUR '93*, pages 493–508, Berlin, Heidelberg, and New York, 1993. Springer-Verlag.

[NvOP00] T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*. IOS Press, 2000.

[ORR$^+$96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. CAV'96,* Lecture Notes in Computer Science, 1102:411–414, 1996.

[Par70] D. Park. Fixpoint induction and proof of program semantics. *Machine Intelligence*, **5**:59–78, 1970.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer Verlag (LNCS 828), 1994.

[Pet82] G.L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. ACM Transactions on Programming Languages and Systems, 4(4):758–762, 1982.

[Pet96] M. Pettersson. A definition of Erlang (draft). Manuscript, Department of Computer and Information Science, Linköping University, 1996.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Aarhus University report DAIMI FN-19, 1981.

[PR97] P. Panangaden and J. H. Reppy. The essence of Concurrent ML. In Flemming Nielson, editor, *ML with Concurrency*. Springer-Verlag, 1997.

[Rat97] J. Rathke. *Symbolic Techniques for Value-Passing Calculi*. PhD thesis, University of Sussex, 1997.

[Rep93] J. H. Reppy. Concurrent ML: Design, application and semantics. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning (LNCS 693)*, pages 165–198. Springer-Verlag, 1993.

[Röc00] C. Röckl. *On the Mechanized Validation of Infinite-State and Parameterized Reactive and Mobile Systems*. PhD thesis, Fakultät für Informatik, Technical University of Munchen, 2000.

[Sel94] A. Sellink. Verifying process algebra proofs in type theory. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages (Workshops in Computing)*, pages 314–338. Springer Verlag, 1994.

[Sel96] A. Sellink. *Computer-Aided Verification of Protocols, The Type Theoretic Approach*. PhD thesis, Utrecht University, 1996.

[SFH92] D. Sahlin, T. Franzén, and S. Haridi. An intuitionistic predicate logic theorem prover. In *Journal of Logic and Computation*, 2(5):619–656, October 1992.

[Sim95] A. Simpson. Compositionality via cut-elimination: Hennessy-Milner logic for an arbitrary GSOS. In *Proc. LICS*, pages 420–430, 1995.

[Sti85] C. Stirling. A complete compositional modal proof system for a subset of CCS. In *Proc. ICALP: Annual International Colloquium on Automata, Languages and Programming*, 1985.

[Sti92] C. Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science* Vol. 2 *(S. Abramsky, D. Gabbay, T. Maibaum (eds.)), Oxford University Press*, pages 478–563, 1992.

[Sti01] C. Stirling. *Modal and temporal properties of processes*. Springer Verlag, 2001.

[SW91] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

[Tar55] A. Tarski. A lattice-theoretical fixedpoint theorem and its applications. *Pacific Journal of Mathematics*, 83:157–167, 1955.

[Tor97] S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.

[Vaa90] F. Vaandrager. A process algebra semantics of pool. In *Applications of process algebra, volume 17 of Tracts in Theoretical Computer Science, Cambridge University Press.*, pages 173–236, 1990.

[Vaa93] F. Vaandrager. Uitwerking take-home tentamen protocolverificatie. Unpublished manuscript, in Dutch, 1993.

[vdBHJP00] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verication of sequential java programs. In *14th International Workshop on Algebraic Development Techniques (WADT'99), volume 1827 LNCS*, pages 1–21, 2000.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. Logic in Computer Science*, pages 332–344, 1986.

[Wig01] U. Wiger. Four-fold increase in productivity and quality – industrial-strength functional programming in telecom-class products. In *Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems (FEmSYS 2001)*, 2001.

[Wik99] C. Wiklander. Verification of Erlang programs using SPIN. Technical report, Department of Teleinformatics, Royal Institute of Technology, March 1999.

[Win90] G. Winskel. Compositional checking of validity on finite state processes. In *Proceedings of CONCUR 1990, LNCS 458*. Springer-Verlag, 1990.

[Win91] G. Winskel. A note on model checking the modal ν-calculus. *Theoretical Computer Science*, 83:157–187, 1991.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, Cambridge, MA, 1993.

[Wou01] A.G. Wouters. Manual for the μCRL toolset (version 2.07). Technical Report To appear, CWI, Amsterdam, 2001.

[WWN99] January 1, 2000, the day the earth will stand still. *Weekly World News*, December 1999.

[XL90] L. Xinxin and K.G. Larsen. Compositionality through an operational semantics of contexts. In *Proceedings of ICALP'90, LNCS 443*, pages 526–539. Springer-Verlag, 1990.