UNITY in Diversity

A stratified approach
to the verification
of distributed algorithms

Tanja Vos

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Vos, Tanja Ernestina Jozefina

UNITY in Diversity, a stratified approach to the verification of distributed algorithms
Tanja Ernestina Jozefina Vos. - Utrecht:
Universiteit Utrecht, Faculteit Wiskunde en Informatica
Proefschrift Universiteit Utrecht. - Met index lit. opg.
- Met samenvatting in het Nederlands.
ISBN 90-393-2316-X
Trefw.: algoritmen / verificatie / wiskundige logica.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation Series 2000-02

Typeset with LaTeX.
Cover foto by Tanja Vos, La Paz-Bolivia.
Printed by Optima, Rotterdam.

UNITY in Diversity.
A stratified approach to the verification of distributed algorithms

Eenheid (UNITY) in verscheidenheid,
een gelaagde aanpak voor de verificatie van gedistribueerde algoritmen

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. Dr. H.O. Voorma,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op maandag 10 januari 2000 des namiddags te 12:45 uur

door

Tanja Ernestina Jozefina Vos

geboren op 8 oktober 1971 te Hilversum, Nederland

Promotoren: Prof. Dr. S.D. Swierstra
Prof. Dr. J-J.Ch. Meyer
Faculteit Wiskunde en Informatica, Universiteit Utrecht

Voor papa

Contents

Acknowledgements

I am very grateful to the many people who have supported me in writing this thesis.

First of all, my promotor and advisor Doaitse Swierstra, a remarkable person who, besides supervising me in writing this thesis, is always willing to help me with other things and give good advice. Moreover, I thank my co-promotor John-Jules Meyer for proof-reading my thesis and always making me feel good about the results I had achieved. I thank them both for offering me an AIO position at the University of Utrecht.

The members of the reading committee, in alphabetical order, Ralph Back, Wim Hesselink, Lambert Meertens, Jan van Leeuwen, en Frits Vaandrager for reviewing my thesis.

Wishnu Prasetya, not only did he proof-read my thesis, but he is always willing to answer any of my questions at any time. Even when he was still in Indonesia he always promptly replied to my emails.

Hielko Ophoff who helped me install HOL90, first on an SGI then an HP and finally, succesfully, on a SUN.

Tom Melham for inviting me to come to the University of Glasgow to work with him and Graham Collins on a problem I encountered.

John Harrison for his willingness to discuss research problems with me, and comming to Holland to tell about his research at the Third Dutch Prooftools day.

Kaisa Sere for inviting me to come to the Åbo Akademi in Turku, Finland for two weeks. I have had many inspiring discussions with her as well as Ralph Back and Marina Walden.

All the people on the HOL-mailing list (info-hol@lal.cs.byu.edu) who always, very rapidly, reply to any question you might have.

Piet van Oostrum, always prepared to enthusiastically answer any question related to LaTeX.

All the people currently or previously working at the computer systems group and the secretariat at the department of computer science of the UU. Special thanks go to Rob Cozzi, who, as our previous departemental manager, helped me several times with problems that were not related to computer science.

All my other collegues at the UU, and researchers working at related departments in KUN, TUE, UT, UvA and VU. Especially Niels Peek who put up with me as a room-companion for 4 years. Lambert Meertens for the inspiring discussions we had about my research, and Gerard Tel always providing me with everything I wanted to know about distributed algorithms.

I also want to thank all my friends. Especially Roeland van Vliet, his abilities to make me laugh are priceless.

Dick Ladage for designing my cover, and Santiago "Guapiiisimo" Valero Sanchéz for lending me his foto camera for taking the foto in La Paz-Bolivia after I lost my sister's camera.

Finally, I am grateful to my family for their moral support. Without them it would have been impossible to write this thesis. Special gratitude goes to my sister Yvonne who is always there for me.

Chapter 1

Introduction

Over the past twenty years, distributed computing has rapidly emerged as a separate area of computer science research in response to the difficulties experienced during the design of distributed systems. These difficulties stem from the fact that distributed systems are conceptually far more complex than a single computing unit. Whereas in the latter only one action can occur at the same time, in a distributed system – and its underlying distributed algorithms – the number of possibilities of what can happen when and where tends to be enormous due to the presence of non-determinism and parallelism. This makes it hard to design correct and reliable distributed systems.

Formal methods, the term by which the variety of mathematical modelling techniques that are applicable to computer system design and verification is meant, are advocated as a way of increasing the reliability of computer based systems. Many [BS92, BH95b, BS93b, BBL93, BH95a, BS93a, Bow93, BJ93, CGR93, CG92, GCR94, Hal90, Kem90, Nic91, RvH93, Rus94, WW93] believe that the use of formal methods currently offers the only intellectually defensible method for handling the software crisis that increasingly affects the world of embedded and distributed systems. Formal methods can be applied at three levels, providing different levels of reliability of the system developed.

At a basic level, formal methods may be used for specification of the system to be designed. The use of formal specification techniques can be of benefit in most cases. Using a formal specification language instead of natural language has the advantage that specifications are more concise and less ambiguous, which makes it easier to reason about them and helps to gain greater insight into and understanding of the problem solved. Furthermore, formal specifications serve as a valuable piece of documentation, which is essential for software maintenance purposes.

The next level of use is formal development, which involves formally specifying the program, proving that certain properties are satisfied, proving that undesirable properties are absent, and finally applying a refinement and decomposition calculus to the specification such that it may gradually be transformed into an efficient and concrete representation of the program. The proofs involved are pencil-and-paper proofs, which can be formal or informal, depending on the level of assurance that is required.

At the last, and most rigorous, level, the whole process of proof is mechanised. Hand proofs or design inevitably lead to human error occurring for all and even the simplest systems. Verifying the design process with a mechanical theorem prover reduces the possibility of errors. Although some argue that this can never eliminate errors completely since the program that does the verification itself may be incorrect, experience shows that theorem provers are very reliable, and definitely much more reliable than people. In addition to reducing errors, the use of theorem provers attributes to the understanding of the problem that is being solved, because during the verification process one is irrevocably confronted with every aspect of the program under construction.

Although formal methods and mechanical verification are becoming more and more accepted as the only intellectually defensible way in which the quality of both software and hardware can be improved, it should be remembered that they are *not* some universal panacea. In this context we refer to the following quote from C.A.R. Hoare:

> *Of course, there is no fool-proof methodology or magic formula that will ensure a good, efficient, or even feasible design. For that, the designer needs experience, insight, flair, judgement, invention. Formal methods can only stimulate, guide, and discipline our human inspiration, clarify design alternatives, assist in exploring their consequences, formalise and communicate design decisions, and help to ensure that they are correctly carried out.*

1.1 Objectives of this thesis

In this thesis we apply the last, and most rigorous, level of formal methods to the design and verification of distributed algorithms. One of the objectives of our research is to build re-usable libraries for a theorem prover which state the correctness and computational power of basic building blocks of distributed applications, such that these libraries can be used when developing large distributed applications which are composed of these basic blocks. Pursuing this goal we encounter two complex and time-consuming activities: using theorem provers, and studying and verifying distributed algorithms. Therefore, another aim of our research is to investigate which separate activities are most complex and time-consuming, and what can be done to change this.

Several aspects contribute to the time needed to mechanically verify the correctness of a distributed algorithm. First the tool itself, e.g. its efficiency, the amount of decision procedures that enable automatic verification, the steepness of its learning-curve, and its user-interfaces. Second, the complexity of the distributed algorithm and its correctness proof. Evidently, complexity that is inherent to some algorithm cannot be reduced, and a certain amount of time has to be spent in order to verify its correctness. However, unnecessary complexity can be introduced by inadequate descriptions or representations of algorithms, unstructured and badly motivated correctness proofs and strategies, and insufficient analysis oriented towards the discovery of classifications of similar algorithms and re-usable theories. In this thesis we show

that enhanced representations of (distributed) algorithms significantly influence the ease of reasoning about them. More specifically, we show that better representations of algorithms can reduce the time and effort needed to understand the functionality, applicability and properties of the algorithm. Moreover, we show that better representations can increase the ability to see similarities and differences between algorithms and learn how to invent and encapsulate new algorithms. As a result we finally obtain classifications of algorithms. Subsequently, we demonstrate how the proof effort and complexity of correctness proofs is reduced by constructing efficient proof strategies based on an analysis of the similarities of algorithms within a specific classification.

1.2 What we use

For the formal and mechanical verification of distributed algorithms in this thesis we use the HOL theorem proving environment [GM93], the UNITY programming theory [CM89], and an embedding of the latter in the former that is an extension of Prasetya's embedding [Pra95].

The HOL system [GM93] is an interactive mechanical proof assistant for conducting proofs in higher order logic, and provides an environment for defining other formal systems and proving statements about them. HOL does not attempt to prove theorems automatically, and thus is better described as a proof assistant, recording proof efforts along the way, and maintaining the security of the system at each point, but remaining essentially passive and directed by the user. HOL is, however, fully programmable, and enables the user to construct programs that automate whatever theorem-proving strategy he or she desires.

Chandy and Misra put forth the UNITY programming theory [CM89] as a vehicle for the study, design and verification of distributed computations. UNITY's success as a research tool is a direct result of its minimalistic philosophy, which enables researchers to focus attention on the essence of the problem rather than implementation details concerning programming languages and architectures. The UNITY theory consists of a programming language and an associated logic. Program design in UNITY commences with (informal) specifications of what the program is expected to accomplish, and then proposes an operational solution (i.e. a UNITY program) to meet those goals. The resulting UNITY program is an abstraction of the actual implementation: it describes *what* must be done, and not *when*, *where* and *how*. The process of program verification entails formalising a program specification in the UNITY logic, and decomposing and refining this specification until it is detailed enough to be directly proved from the program text.

Prasetya [Pra95] made an embedding of the programming logic UNITY in HOL, by extending the latter with all definitions required by the logic, and making all basic theorems of the logic available by proving them. The gain from this is that the formal design of programs can now be assisted by a mechanical verification with the theorem prover, resulting in a significant increase of the trustworthiness of the design. Moreover, he defined two extensions of the programming logic UNITY which are also embedded in HOL. The first extension of the programming logic UNITY concerns compositionality properties. A problem of UNITY is that progress properties are not

compositional with respect to parallel composition. That is, we cannot in general decompose a progress specification of a program into the specifications of its parallel components. Therefore one is unable to develop a component program in isolation, which is awkward. The extension presented is however compositional. The second extension regards self-stabilisation and convergence of programs. Roughly speaking, a self-stabilising program is a program that is capable of recovering from arbitrary transient failures. Obviously such a property is very useful, although the requirement to allow arbitrary failures may be too strong. The more general notion of convergence, which allows a program to recover only from certain failures, is used to express a more restricted form of self-stabilisation. Since self-stabilisation and convergence are considered to be essential for programs in safety-critical environments, e.g. distributed environments, this second extension is significant for our purposes. Moreover, an induction principle is formulated for convergence which is stronger than the one for UNITY's reach-to operator. As a consequence, a powerful technique for proving convergence has become available.

1.3 How to read this thesis

This thesis consists of two parts. The first part consists of Chapter 2 through 7, and contains the machinery that is needed for the applications in the second part.

Chapter 2: The HOL theorem proving environment

As implied by the title, this chapter describes the HOL theorem proving environment [GM93]. It is not meant to give an introduction to the system, but merely to explain those aspects of HOL that are needed in the rest of the thesis. The concepts treated are: terms and types in the HOL logic; Hilbert's ε choice operator; antiquotation and type abbreviations in HOL; how HOL can be extended by definitions, axioms, and the derivation of theorems; two different approaches to embed another (formal) system in HOL; and two ways to define partial functions in HOL.

Chapter 3: Basic programming theory

This chapter describes basic programming theory underlying the work in the rest of this thesis. The concepts treated are: program variables and their types, states, expressions, predicates, and actions. Although the mathematical theory in this chapter is not new, the combination of the choices made while embedding various basic programming concepts in higher order logic, as well as the notation, differ slightly from other work [Pra95, APP93, Gor89, BW90, WHLL92, Wri94, Lån94, Wri91, Age91]. Therefore, to avoid confusion this chapter precisely defines these concepts and their notation.

Chapter 4: UNITY

This chapter gives an overview of the UNITY theory and Prasetya's extensions. Again, only those concepts that are needed in the rest of the thesis are presented: the execution, syntax, and well-formedness of UNITY programs; UNITY proof logic, including Prasetya's extensions; and superposition refinement of UNITY

programs. This chapter presents a myriad of laws that are used during the decomposing and refining of UNITY specifications.

Chapter 5: Embedding HOL in UNITY

This chapter, describes the theories that are built on top of Prasetya's embedding in order to cope with the slightly different program-theoretic foundations presented in Chapter 3.

Chapter 6: A methodology and a case study

This chapter describes an extension of the UNITY [CM89] methodology for designing and/or mechanically verifying distributed algorithms, which is used in subsequent chapters. To illustrate the use of this methodology, a case study to design and verify a converging distributed sorting algorithm is presented. One aim of this case study is to highlight the different steps of the methodology, and for that reason, the problem tackled is relatively simple compared to real-life applications and the algorithms verified in later chapters. Another objective of this case study is to be able to reflect on the methodology and the time spent in each separate step (Section 6.3).

Chapter 7: Program refinement in UNITY

This chapter presents a new framework of program refinement, which is based on a refinement relation between UNITY programs. The main objective of introducing this new relation is to reduce the complexity of correctness proofs for existing classes of related distributed algorithms. Moreover, it is shown that this relation is also suitable for the stepwise development of programs, and incorporates most of the program transformations found in existing work on refinements

Chapter 8: The proof of the program is in the representation

This chapter argues that the enhanced representations of distributed algorithms can significantly influence the ease of reasoning about them. First, it is described *what* we consider to be a good representation of an algorithm, and *how* such representations can be obtained. Subsequently, better representations of two specific algorithms are constructed, and it is shown that these reduce the time and effort needed to understand the algorithms' functionality, applicability and properties, increase the ability to see similarities and differences between other algorithms and learn how encapsulate new algorithms, and as a result obtain a class of algorithms that we call distributed hylomorphisms. Finally, it is demonstrated how the proof effort and complexity of correctness proofs of these distributed hylomorphisms can be reduced by identifying a refinement ordering on them.

Chapter 9: Formally proving the correctness of distributed hylomorphisms

This chapter extensively describes the formal proofs of the correctness of distributed hylomorphisms. Although it is a tough chapter to read, we think we have succeeded in presenting structured correctness proofs that are highly intuitive due to our incremental, demand-driven construction of the invariant. Since

this chapter uses the refinement framework from Chapter 7, it also serves as an illustration of the latter's usage and effectiveness for reducing the complexity of correctness proofs for existing classes of related distributed algorithms.

The organisation of these chapters is bottom-up, meaning that a concept defined in chapter n is only used in a chapter m when $m \geq n$. Reading these chapters in the order presented may prevent the reader from swiftly arriving at the topics he or she is interested in. Consequently, we have compiled an extensive index that should enable the reader to start reading this thesis at any chapter (with the exception of 9), looking up desired definitions in a demand-driven way. Moreover, below we indicate which chapters one should read depending on the topics the reader is interested in.

- For a reader interested in the underlying embedding of UNITY, we advise:
  2 - 3 - 4 - 5 - Appendix B - Appendix D
- For a reader interested in how to manually add recursive data types to HOL:
  2 - 5 - Appendix B - Appendix D
- For a reader interested in the extension of the UNITY methodology used for the design and mechanical verification of distributed algorithms, we advise:
  4 - 6 - 8
- For a reader interested in refinement of UNITY programs, we advise:
  3 - 4 - 7 - Appendix C
- For a reader interested in the representations of, and reasoning about distributed algorithms, we advise:
  (glance through the beginning of) 4 - 8
- For a reader interested in the formal verification of distributed algorithms:
  3 - 4 - 7 - 8 - 9 - Appendix D

1.4 Using the results

Should one want to apply the results presented in this thesis, the safe way to do it is *not* to consider them as presented in the thesis, which after all was hand-typed and hence is likely to contain errors. Instead, one should take the results as they are reported by HOL, and contained in the libraries.

1.5 Presenting theorems

Almost all theorems in this thesis are computer-checked. Although for such theorems there is basically little need to present their proofs, some proofs will be presented with a view to:

- enabling the reader to validate it,
- showing how easily a theorem follows from some facts,
- illustrating a style of verification,
- giving some insight in proving a closely related problem,
- exposing the actual nature of a theorem.

Like [Pra95], when presenting computer-checked results (i.e. theorems and definitions) these will be marked by the names they are identified with in the HOL theories that we constructed. For example:

Theorem 1.5.1 PINK PANTHER *Pink_Panther_thm*

$$Fu \;=\; Fu \circ Fu$$

The number (1.5.1) and the name PINK PANTHER are how we refer to such a theorem. The name `Pink_Panther_thm` is how the theorem is called in HOL. *Implicitly, this means that the theorem is mechanically verified.* When referring to a theorem, or definition we – for the reader's convenience – include the page number in which the referred item can be found. The page number is printed as a subscript like in: Theorem $1.5.1_7$ or Theorem PINK PANTHER$_7$.

In this thesis we employ two formats for presenting theorems, which mean exactly the same:

$$p \wedge q \Rightarrow r \qquad \text{or} \qquad \frac{p \quad q}{r}$$

These two formats are used interchangeably, and merely practical reasons like readability and lay-out were paramount to the decision whether to use one or the other.

1.6 Notational conventions

Some notational conventions used in this thesis can be found in Appendix A, others are introduced throughout the text prior to the moment of usage. For the reader's convenience, we have compiled an extensive index in which every non-standard symbol that is used in this thesis can be found together with the number of the page on which this symbol is explained. Moreover, the index contains entries "*notational conventions and overloading*" and "*overloading traditional notation*" referring to all pages on which notation is introduced.

Chapter 2

The HOL theorem proving environment

The HOL system [GM93] is a mechanical proof assistant for conducting proofs in **H**igher **O**rder **L**ogic. It is free, comes with extensive documentation, libraries, an interactive help system, a myriad of web-sites providing information and a dynamic search engine for HOL information[1], and a mailing list[2]. HOL is based on the LCF approach [Pau87] to interactive theorem proving developed by Robin Milner in the early 1970s, and is an implementation of a version of Church's simple theory of types, a formalism dating back more than 50 years. Basically, higher order logic is a version of predicate calculus which allows for quantification over predicate and function symbols of any order.

HOL is built on top of the strict functional programming language SML. The HOL system defines SML types for the various logical entities (e.g. terms, types, theorems, theories). The SML type for theorems is an *abstract data-type* `thm`, and as a consequence, an object of type `thm` can only be constructed through a limited set of operations. There are certain predefined SML identifiers which are given values of type `thm` when the system is built. These values correspond to the five axioms underlying higher order logic. Moreover, there are several predefined SML functions that take theorems as arguments and return theorems as results. These functions correspond to the eight *primitive inference rules* underlying higher order logic. The SML type checker ensures that values of type `thm` can only be constructed through these predefined functions. Therefore, every value of type `thm` (i.e. a theorem) must be either one of the five axioms or have been obtained by applications of the predefined functions representing the eight primitive inference rules. In addition to primitive inference rules, there are many *derived inference rules* available in HOL. These are SML functions consisting of compositions of the eight primitive inference rules. Consequently, although the SML code for derived rules can be arbitrarily complex, they will never return a theorem that does not follow by valid logical inference.

[1] http://lal.cs.byu.edu/lal/hol-documentation.html
[2] info-hol@lal.cs.byu.edu

	standard notation	HOL notation
Denoting types	$x \in A$ or $x : A$	`--‘x:A‘--`

Proposition logic	$\neg p$, true, false
	$p \wedge q$, $p \vee q$
	$p \Rightarrow q$
		`--‘~p‘--`, `--‘T‘--`, `--‘F‘--`
		`--‘p /\ q‘--`, `--‘p \/ q‘--`
		`--‘p ==> q‘--`

Universal quantification
Existential quantification
	$(\forall x, y :: P)$
	$(\forall x : P.x : Q)$
	$(\exists x, y :: P)$
	$(\exists x : P.x : Q)$
		`--‘(!x y. P)‘--`
		`--‘(!y::P. Q)‘--`
		`--‘(?x y. P)‘--`
		`--‘(?x::P. Q)‘--`

Function application λ abstraction	$f.x$ $(\lambda x.\ E)$	`--‘f x‘--` `--‘(\x. E)‘--`
Sections	$(\wedge)\ p\ q$, $(+)\ x\ y$	`$/\ p q`, `$+ x y`
Conditional expression	if b then E_1 else E_2 (or $b \rightarrow E_1 \mid E_2$)	`--‘b => E1 | E2‘--`
Sets	$\{a, b\}$, $\{f.x \mid P.x\}$	`--‘{a,b}‘--`, `--‘{f x | P x}‘--`
Set operators	$x \in V$, $U \subseteq V$ $U \cup V$, $U \cap V$	`--‘x IN V‘--`, `--‘U SUBSET V‘--` `--‘U UNION V‘--`, `--‘U INTER V‘--`

Figure 2.1: The HOL Notation.

HOL is an interactive proof assistant: one types a formula and proves it step by step using any primitive strategy provided by HOL. When the proof is completed, the code constructing a theorem can be collected and stored in a file, to be given to others for the purpose of re-generating the proved fact, or simply for documentation purposes in case modifications are required.

HOL is not an automatic theorem prover. Unlike for example the Boyer-Moore theorem prover [BM88], HOL does not attempt to automatically prove theorems, but rather provides an environment and supporting tool to enable users to prove theorems. However, since HOL is programmable, the user is free to construct programs that automate whatever proof-strategy he or she desires.

This chapter describes only those aspects of HOL that are needed in the rest of this thesis. Section 2.1 describes HOL terms and types. Sections 2.2 and 2.3 explain Hilbert's ε-operator and antiquotation respectively. Section 2.4 outlines how HOL can be extended. Sections 2.5 and 2.6 respectively describe how to add definitions and prove theorems in HOL. Section 2.7 explains what an embedding is and briefly describes the two approaches that can be taken when constructing one. Section 2.8, finally, describes two mechanisms for defining partial functions within HOL.

2.1 HOL terms and types

All HOL formulae, also called logical terms, are represented in SML by a type called `term`. Figure 2.1 shows examples of how the standard notation is translated to the HOL notation. As the reader can see, the HOL notation is as close an ASCII nota-

tion can be to the standard notation. Anything in between `--‘...‘--` is parsed as a logical term. Terms of the HOL logic are quite similar to SML expressions, which can cause confusion since these terms have different types: logical types and SML types (called *object language types* and *meta-language types* respectively). The object language types of HOL terms are represented by meta-language type `hol_type`, and are denoted by expressions of the form `==‘:...‘==`. There is a built-in function `type_of`, which has SML type `term->hol_type` and returns the logical type (i.e object language type) of a HOL term. An example taken from [GM93] may elucidate these matters. Consider for example the logical term `(--‘(1,2)‘--)`

- It is a HOL term with object language type (i.e. logical type) `(==‘:num#num‘==)`.
- It is an SML expression with meta-language type `term`
- Evaluating `type_of --‘(1,2)‘--` results in `(==‘:num#num‘==)`.
- This object language type `(==‘:num#num‘==)` has SML type `hol_type`.
- In contrast consider the SML expression `((--‘1‘--), (--‘2‘--))` which has SML type `term#term`.

There are three classes of object language types in HOL:

- *type constants* are identifiers that name sets of values. Examples are `bool`, `num`, `real`, and `string` which denote the set of booleans, the set of natural numbers, the set of reals, and the set of strings respectively.
- *type variables* to denote “any type”. Names denoting type-variables must always be preceded by a `’` like in `==‘:’a‘==` or `==‘:’b‘==`.
- *compound types* are object language types that are built from other types using a type operator. Examples of type operators in HOL are: product (`#`), sum (`+`), function (`->`), lists (`list`), and sets (`set`).

In HOL one can define new type operators using the type definition package [Mel89, GM93]. The main SML function in this package is `define_type`, with which any *concrete recursive data type* can be defined in the HOL system. A concrete recursive data type is one of the form:

$$\begin{array}{rcl} \texttt{op} & = & \mathsf{C}_1\, t_1^1\, \ldots\, t_1^{k_1} \\ & | & \ldots \\ & | & \ldots \\ & | & \mathsf{C}_m\, t_1^m\, \ldots\, t_1^{k_m} \end{array} \tag{2.1.1}$$

where each C_i is a constructor function, and where each t_i^j:

- is either a type expression already defined as a type (this type expression must *not* include the type operator `op`),
- or is the name `op` itself.

Recursive types where some of the t_i^j’s are type expressions that do include the type operator `op`, are not concrete; for these types, the type definition package does not work, and the user has to define such a type manually (we shall see an example of this in Chapter 5). An example of a concrete recursive type that can be defined in HOL

using the type definition package is the following type representing labelled binary trees:

```
binTree  =  LEAF 'a
         |  NODE 'a binTree binTree
```

It can be defined in HOL using the `define_type` function:

```
val binTree_Axiom
    =
    define_type
     {name = "binTree_Axiom",
      fixities = [Prefix, Prefix],
      type_spec = `binTree = LEAF of 'a
                           | NODE of 'a => binTree => binTree`};
```

The `name` field contains the name under which the abstract characterisation theorem (which for these concrete recursive data types will have the form of a "primitive recursion theorem") of the new type will be stored. The `fixities` field indicates the fixities of the respective constructors (i.e. in the example, the fixities of both constructors LEAF and NODE are specified to be prefix). The `type_spec` field is a user-supplied specification of the concrete recursive type that is to be defined.

Having defined the new type operator `binTree`, we can define objects to be of type `binTree`. An example HOL-session illustrates this.

```
- (--`x:(num)binTree`--);
val it = (--`x`--) : term

- (==`:(num)binTree`==);
val it = (==`:num binTree`==) : hol_type
```

2.2 Hilbert's ε-operator

Hilbert's ε-operator is a primitive constant of higher order logic[Mel89, GM93]. Informally, its syntax and semantics are as follows. If $P[x]$ is a boolean term involving a variable x of type α, then $(\varepsilon x.\ P[x])$ denotes some value, say v, of type α such that $P[v]$ is true. If there is no such value (i.e. $\forall v \in \alpha : \neg P[v]$) then $(\varepsilon x.\ P[x])$ denotes some fixed but arbitrary chosen value of α[3]. For example:

- $\varepsilon n.\ 4 < n \wedge n < 6$ denotes the value 5
- $\varepsilon n.\ (\exists m :: n = 2 \times m)$ denotes an unspecified even natural number
- $\varepsilon n.\ n < n$ denotes an arbitrary natural number

The formalisation of the ε-operator in HOL is by the following theorem:

[3] A consequence, in HOL, types must be non-empty.

Theorem 2.2.1 *SELECT_AX*

$$\forall P :: (\exists x :: P.x) \;\Rightarrow\; (P.(\varepsilon x.\ P.x))$$

Consequently, ε can be used to obtain a logical term which provably denotes a value with a given property P from a theorem merely stating the existence of such a value.

2.3 Antiquotation

Within a quotation (i.e. `--‘...‘--`), expressions of the form (`^t`) (where `t` is an SML expression of meta-language type `term`) are called *antiquotations*. An antiquotation (`^t`) evaluates to the SML value of the expression `t`. As an exemplification, consider the following small HOL-session:

```
- val x = (--‘y /\ z‘--);
val x = (--‘y /\ z‘--) : term

- val p = (--‘^x \/ k‘--);
val p = (--‘y /\ z \/ k‘--) : term

-
```

Type abbreviations can be made using antiquotation and the SML function `ty_antiq` of type `hol_type->term`. One gives an SML name to the term representing the desired type, and then uses this name via antiquotation. The following HOL-session illustrates this:

```
- val numpair = ty_antiq (==‘:num # num‘==);
val numpair = (--‘(ty_antiq((==‘:num # num‘==)))‘--) : term

- val p = (--‘x:^numpair‘--);
val p = (--‘x‘--) : term

- type_of p;
val it = (==‘:num # num‘==) : hol_type

-
```

2.4 Extending HOL

HOL provides the user with a logic that can easily be extended by the definition of new constants, functions, relations and types. These extensions are organised into

units called *theories*. Theories are structured hierarchically to represent sequences of extensions. More specifically, subsequent pieces of work can be built on (i.e. extend) the definitions and theorems of an existing theory T by making T a *parent theory* of new theories, thus creating a *theory hierarchy*. A HOL theory is similar to a traditional theory of logic in that it contains definitions of new types and constants, and theorems which follow from the definitions. It differs from a traditional theory in that a traditional theory is considered to contain the infinite set of all possible theorems that *could be proved* from the definitions, whereas a HOL theory contains only the subset that *have been actually proved* (i.e. are objects of type `thm`).

Besides adding definitions, HOL also provides the ability to assert new axioms in the HOL logic. Since new axioms can introduce inconsistencies, one has to exercise caution when adding them. Asserting new axioms can be done at the user's responsibility when he or she is *convinced* that the axiom can be proved as a theorem and able to give sufficient justification to sustain this.

Extending HOL by using only definitions is called a definitional or conservative extension, since the security of HOL is not compromised. Although it is also possible to introduce silly definitions, these are nothing more than abbreviations, and hence cannot introduce inconsistencies.

2.5 Definitions in HOL

In HOL a definition is also a theorem, which states the meaning of the object that is being defined. Because the HOL notation is quite close to the standard mathematical notation, definition of new objects can, to some extent, be written in a natural way. New definitions are added to HOL using the SML function `new_definition`. As an example, the following HOL-session defines a new *constant* `F2R` denoting an operator that converts a function into a relation. The first argument to `new_definition` specifies the name under which the definition (i.e. theorem) is saved in the theory in which the definition is made (and, according to the notational conventions explained in Chapter 1, corresponds to the name in the upper right corner of Definition A.2.3$_{216}$).

```
- val F2R_DEF = new_definition
     ("F2R_DEF",
       (--‘F2R (f:’a->’b) = (\x y. (y = f x))‘--));

val F2R_DEF = |- !f. F2R f = (\x y. y = f x) : thm

-
```

◀

Recursive definitions can be added for those concrete recursive data types that have been added to HOL using the type definition package. The SML function available for this is called `new_recursive_definition`. For example, a recursive function on labelled binary trees that computes the sum of all the (numeral) values that reside at the nodes can be defined by adding the HOL constant `sum_binTree` as follows:

```
- val sum_binTree = new_recursive_definition
  {name = "sum_binTree",
   fixity = Prefix,
   def = (--`(sum_binTree (LEAF (x:num)) = x)
              /\
    (sum_binTree (NODE x t1 t2) = x + (sum_binTree t1) + (sum_binTree t2))`--),
   rec_axiom = binTree_Axiom};

val sum_binTree =
  |- (!x. sum_binTree (LEAF x) = x) /\
     (!t2 t1 x.
       sum_binTree (NODE x t1 t2) = x + sum_binTree t1 + sum_binTree t2) : thm
```

The `rec_axiom` field corresponds to the abstract characterisation theorem which was returned by the specific call to `define_type` used to define the concrete recursive data type of labelled binary trees.

2.6 Proving theorems in HOL

Within the HOL system, conjectures one wants to prove are called *goals*. To prove a goal one can start with known theorems, combine these to deduce new theorems, and continue until the desired goal itself is obtained as a theorem. Alternatively, one can start with the goal, work backwards by splitting it into simpler subgoals, and continue until all obtained subgoals can be reduced to known theorems. These methodologies are usually referred to as *forward proof* and *backward proof* respectively. Both methodologies for proving new theorems can be visualised by a *proof tree* that has the top-goal as its root, and the intermediate subgoals as its nodes. A forward proof constructs a proof tree from bottom to top, and a backward proof constructs it from top to bottom. A proof tree is generally defined to be *closed* if all leaves are known theorems, and, hence, the top-goal is proved.

As already indicated, in HOL, new theorems can only be generated by applying HOL inference rules to known theorems, i.e. axioms and previously proved facts; basically this comprises forward proof in HOL.

HOL also supports backward proof using the notion of tactics invented in 1970 by Robin Milner. In HOL a tactic is an SML function, the effect of which is to replace a goal with a set of subgoals which if proved are sufficient to prove the original goal. The effect of a tactic is essentially the inversion of an inference rule. Some examples of pre-defined SML functions implementing tactics in HOL are:

`MATCH_ACCEPT_TAC: thm -> tactic`, which proves a goal if it is an instance of the supplied theorem.

`MATCH_MP_TAC: thm -> tactic`, the effect of which corresponds to the inversion of the Modus Ponens inference rule. For example, suppose we have the following theorem:

```
LESS_TRANS |- !m n p. ((m < n) /\ (n < p)) ==> (m < p)
```

Suppose we want to prove the following goal for some `x` and `y`:

```
(--‘x < y‘--)
```

Applying `MATCH_MP_TAC LESS_TRANS` then results in the following subgoal:

```
(--‘?n. (x < n) /\ (n < y)‘--)
```

`REWRITE_TAC: (thm)list -> tactic`; `REWRITE_TAC`$[t_1, \ldots, t_n]$ transforms a goal by rewriting it with the given theorems $t_1, \ldots, t_n$.

Tactics can be composed using *tacticals*. Examples of the most used tacticals in HOL are `THEN`, `ORELSE` and `REPEAT`.

- `THEN : tactic->tactic->tactic`; if `T1` and `T2` are tactics, then `T1 THEN T2` is a tactic, which first applies `T1` and then applies `T2` to all the subgoals that were generated by `T1`.
- `ORELSE: tactic->tactic->tactic`; if `T1` and `T2` are tactics, then `T1 ORELSE T2` is a tactic, which first tries `T1`. If `T1` fails then it tries `T2`.
- `REPEAT: tactic->tactic`; if `T` is a tactic, then `REPEAT T` is a tactic that repeatedly applies `T` until it fails.

HOL provides a facility, called the *sub-goal package*, to interactively construct a backward proof. The package memorises the proof tree and the information needed to achieve the top-goal from achievements of the subgoals. The tree can be displayed, extended, and partly un-done. Whereas interactive forward proofs are also possible in HOL simply by applying inference rules interactively, HOL provides no facility to automatically record proof trees for forward proofs.

2.7 Embeddings

An extension of HOL (i.e. a theory hierarchy) that enables the mechanised reasoning of another (formal) system (e.g. a programming logic, programming language, a process algebra, a HDL) is what is called an *embedding* of the (formal) system in HOL. There are two approaches to mechanise another (formal) system in HOL: *shallow* embedding and a *deep* embedding.

A shallow embedding of some (formal) system, introduces new constants in the HOL logic to represent each construct of the system and defines these constants as functions that directly denote the construct's semantics. For example, a shallow embedding of the assign statement ($x := e$) of some programming language might be defined as:

$$\texttt{ASSIGN}.x.e.s \;=\; \lambda y.\ (y = x)\ \texttt{=>}(e.s) \mid (s.y)$$

where the constant `ASSIGN` is defined as a higher order function that takes the logical representations of variable x and expression e as its first and second argument, and a state s as its third argument. The state shall be some function from type `'var` to some type of values; and `ASSIGN` returns a new state in which x is bound to the value of expression e evaluated in state s. The use of such constants for each language construct makes parsing a text into the logic straightforward and properties of the

text can then be proved. This approach, however, makes it difficult to express the static semantics in the logic, and does not allow general properties about the language *itself* to be proved. These limitations are overcome by the alternative approach to mechanise a system in HOL, i.e. the deep embedding.

In a deep embedding, the abstract syntax of the language is defined as a type in the HOL logic, and the semantics is defined as recursive functions over this type. That is, instead of defining a construct *as* its semantic meanings, we define a construct as simply a syntactic object and then separately define its semantics. For example, continuing to the programming language, we shall add some data type to HOL (using `define_type`) to represent the various constructs (`cmd`) of the programming language:

```
val cmd = ...
        | ASSIGN 'var Expr
        | ...
        .
        | ...
```

The semantics are then defined using `new_recursive_definition`. The "deepness" of this embedding is determined by the approach that is used to embed the expressions (i.e. type `Expr`). In this respect we shall refer to a deeper embedding (i.e. when the expressions are shallowly embedded) and a deepest embedding (when the expressions are also deeply embedded).

2.8 Defining partial functions in HOL

Since in the HOL logic all functions are total, we need mechanisms to define partial functions as total ones. The traditional HOL solution is to leave partial functions unspecified for values not in the correct domain. For example, the function HD, that returns the first element of a list, is a partial function that is only defined on non-empty lists. In HOL it is defined as a total function HD : `('a)list` $\rightarrow$ `'a` that is specified by:

$$\mathsf{HD}(\mathsf{CONS}\, h\, t) \;=\; h$$

Consequently, applying HD to an empty list of type `('a)list`, will return a value of type `'a` denoted by HD []. Since the value HD [] is unspecified, nothing definite can be proved about it, and consequently this value can represent "undefinedness" returned by the partial function HD when applied to an element not in its domain.

An alternative solution is to deal with "undefinedness" explicitly, by introducing a constant $\bot$, of which it is known that for any given (non-empty) HOL type σ, $\bot \in \sigma$. Then, a partial function $f \in V \rightarrow \sigma_2$ where $V \subset \sigma_1$, is defined as the total function $f \restriction V$ $(\in \sigma_1 \rightarrow \sigma_2)$

$$\begin{array}{lcll} (f \restriction V).x & = & f.x\ , & \mathsf{if}\ x \in V \\ & = & \bot\ , & \mathsf{otherwise} \end{array}$$

The total function $f \restriction V$ is called the projection or restriction of f to the set V. The constant $\perp$ can be seen as "undefinedness" returned by the partial function f when applied to an element not in its domain. Below, the formal definition of projection is given, together with some properties:

Definition 2.8.1 Projection *Pj_DEF*

For all $f \in \sigma_1 \rightarrow \sigma_2$, $V \subseteq \sigma_1$, and $x \in \sigma_1$:

$$(x \in V \;\Rightarrow\; (f \restriction V).x = f.x) \;\wedge\; (x \notin V \;\Rightarrow\; (f \restriction V).x = \perp)$$

Theorem 2.8.2 Extension *Pj_EQ*

$$(f \restriction V = g \restriction V) \;=\; (\forall x : x \in V : f.x = g.x)$$

Theorem 2.8.3 Anti-monotonicity *Pj_EQ_MONO*

$$V \subseteq W \wedge (f \restriction W = g \restriction W) \;\Rightarrow\; (f \restriction V = g \restriction V)$$

Theorem 2.8.4 Extension by $\cup$ *Pj_EXTEND_BY_UNION*

$$(f \restriction (V \cup W) = g \restriction (V \cup W)) \;=\; (f \restriction V = g \restriction V) \wedge (f \restriction W = g \restriction W)$$

Theorem 2.8.5 composition *Pj_COMPO*

$$(f \restriction V \restriction W = f \restriction (V \cap W)$$

◀

Note that both methods, eliminate the need to introduce special rules for dealing with "undefinedness", since it is regarded as being an ordinary value representing the set of "uninteresting" but valid values.

The advanced reader who skips parts that appear to him too elementary may miss more than the less advanced reader who skips parts that appear to him too complex.

– George Pólya [Pól71]

Chapter 3

Basic programming theory

This chapter describes the basic programming theory underlying the work presented in the rest of this thesis. The concepts treated are states, program variables and their types, expressions, predicates, and actions.

Although the mathematical theory in this chapter is not new, the combination of the choices made while embedding various basic programming concepts in higher order logic, as well as the notation, differ slightly from other work [Pra95, APP93, Gor89, BW90, WHLL92, Wri94, Lån94, Wri91, Age91]. Therefore, to avoid confusion this chapter precisely defines these concepts and their notation. For instance, as in [Pra95, APP93, Gor89, BW90] states are represented as functions from variables to values, but unlike these works program variables can have different types. Moreover, as in [BW90] but unlike [Pra95, APP93, Gor89] the embedding of actions is deep (i.e. their syntax is defined as a type and their semantics is defined by a function over this type).

3.1 Program states

The state of a program is an assignment of values from a universe of values to the program's variables. Generally, there are two methods to concretely[1] represent the state of a program.

The first method is used by von Wright et al [WHLL92, Wri94] in a HOL embedding of The Refinement Calculus, by von Wright [Wri91] and Långbacka [Lån94] in their separate HOL embedding of TLA, and by Agerholm [Age91] in his formalisation of the weakest pre-condition calculus of a small imperative programming language. This method formalises states as tuples of values, where every component corresponds to a program variable (i.e. the value of this variable in that state). Consequently, program variables have no global names, and are identified by their position in the state tuple. For example, suppose we have a program which has variables x, y, and

[1] Another approach is an abstract model of the state space. This approach is used by Hensel et al [HHJ98] in a coalgebraic formalisation of object-oriented classes. Here the state space is modelled as a black box on which several operations (or methods) are defined that can be used to obtain information about the state space or modifying it.

z of type `num`, `num`, and `bool` respectively. The state in which $x = 1$, $y = 2$, and $z =$ T, is represented by the state (1, 2, T), and the state predicate $x > 0$ is represented by $(\lambda(x, y, z).\ x\ >\ 0)$. The major advantage of this method, is, as can be seen from the example above, that it is very easy to represent the state of programs in which different variables can take values of different types. The problem, however, with this method is that, since program variables have no global names, it becomes almost impossible to treat them in isolation. As a consequence, in the context of program composition, it will be difficult to define what a shared variable is. Moreover, if a program P has $\{x, y\}$ as its variables whose values range over `N`, and if Q has $\{a, b\}$ as its variables whose values also range over `N`, then both programs share the same universe of states, namely $\mathtt{N} \times \mathtt{N}$. If those variables are intended to be distinct then some trick will be required to impose the distinction. Another problem with this representation arises with programs that have a variable number of program variables. For example, the number of variables needed in a distributed sorting program on an arbitrary connected network N obviously depends on the number of nodes in N. Consequently the length of the tuples representing the states of this program also depends on N, and hence is not fixed, which is a problem in HOL because tuples are required to have a statically determined length, due to strong typing rules.

The second method is to represent states as functions from a universe `Var` of *all* program variables to a universe `Val` of all values these variables may take. This method is used by many who embed a program logic in HOL [Gor89, BW90, Hal91, And92b, And92a, APP93, Pra95]. Since a program P, having a set of variables V (i.e. $V \subseteq$ `Var`), can neither influence nor depend on the values of the variables outside V, state functions can be represented as *total* functions $s \in \mathtt{Var} \rightarrow \mathtt{Val}$, and it makes no difference whether or not s specifies values for variables outside V. For example, consider a program P with variables $\{u, v\}$. A state of P, where `Val` in the universe of `num`, can be:

- $(s.u = 1) \wedge (s.v = 2) \wedge (\forall x : x \notin \{u, v\} : (s.x = 0))$, specifying values for variables not in $\{u, v\}$, or
- $(s'.u = 1) \wedge (s'.v = 2)$, leaving the values for variables not in $\{u, v\}$ unspecified.

Evidently, executing P in state s gives the same result as executing program P in state s', $(s \restriction \{u, v\})$ or $(s' \restriction \{u, v\})$.

The major disadvantage of this second method is the way in which to represent states of programs in which different variables can take values of different types. In order to do this we need a multi-typed universe of values, which is the union of all program variables' types. For example, for a program that needs boolean and numeral typed variables, we can define

$$\begin{array}{lcl} \mathtt{num\_bool\_Val} & = & \mathsf{NUM}\ \mathtt{num} \\ & | & \mathsf{BOOL}\ \mathtt{bool} \end{array}$$

declaring a new data type constant[2] `num_bool_Val` and constructor functions NUM ($\in$ `num`$\rightarrow$`num_bool_Val`) and BOOL ($\in$ `bool`$\rightarrow$`num_bool_Val`), and take this type as our multi-typed universe of values. The problem is that destructor functions have to be defined, and all normal operations on numerals and booleans have to be lifted to work

[2]That is a type operator of arity 0.

on this new type using the destructor and constructor functions. This is very tedious, since, as the number of different types in a program increases, so does the number of destructor and constructor functions and different symbols to represent the lifted operations. And although on paper, with fancy notations and overloading, one can hide this excessive use of different symbols and destructor and constructor functions, when using a theorem prover this is not possible and consequently very disturbing for the user. Since this thesis builds on Prasetya's [Pra95] UNITY embedding in which it was already decided to represent states as functions from `Var` to (polymorphic type) `Val` because tuples were unsuitable to formulate compositionality results, we decided to deal with this problem as follows. Our embedding is suitable for programs in which different variables can take values of different types, without disturbing the user with the details of the accompanying multi-typed value space. In order to achieve this, a multi-typed value space is created and important operations on the component types of this space and their properties are lifted to this space in such a way that the use of destructor and constructor functions is hidden from the user as much as possible. Obviously, this is only a partial solution for the problem since there will always be programs which need more operations on the component types of this value space, and types outside the value space fixed by us. As far as the first need is concerned, HOL is extensible, and new operations can always be added, albeit at the cost of burdening the user with the use of destructor and constructor functions. The second need is more serious, and if there is no way around it, the value space cannot be used and the user has to create another one that suits his or her needs.

3.2 The universe of values

The multi-typed value space used in this thesis is recursively defined as follows:

```
Val  =  NUM num
     |  BOOL bool
     |  REAL real
     |  STR string                                   (3.2.1)
     |  SET (Val)set
     |  LIST (Val)list
     |  TREE (Val)ltree
```

declaring a type `Val` which denotes the set of all values which can be generated by using the constructor functions NUM, BOOL, REAL, STR, SET, LIST, and TREE. Consequently, states are functions $s \in$ `Var`$\rightarrow$`Val`. The set of all program states will be denoted by `State`.

Two kinds of types can be distinguished for the values that can be assigned to a program variable: their *actual* type (i.e. `Val`), and their *intended* type (i.e. `num`, `bool`, `real`, etcetera). In order to be able to declare the intended type of a program variable, we need functions that check whether a value of type `Val` has the intended type.

Theorem 3.2.1 CHECKING THE INTENDED TYPE

$\mathsf{is\_num}.v$	$=$	$\exists n :: (v = \mathsf{NUM}.n)$	is_num_THM
$\mathsf{is\_bool}.v$	$=$	$\exists b :: (v = \mathsf{BOOL}.b))$	is_bool_THM
$\mathsf{is\_real}.v$	$=$	$\exists r :: (v = \mathsf{REAL}.r)$	is_real_DEF
$\mathsf{is\_str}.v$	$=$	$\exists str :: (v = \mathsf{STR}.str)$	is_str_THM
$\mathsf{is\_set}.v$	$=$	$\exists set : (\mathsf{FINITE}.set) : (v = \mathsf{SET}.set)$	is_set_THM
$\mathsf{is\_list}.v$	$=$	$\exists l :: (v = \mathsf{LIST}.l)$	is_list_THM
$\mathsf{is\_tree}.v$	$=$	$\exists t :: (v = \mathsf{TREE}.t)$	is_tree_THM

Consequently, a type declaration of a program, declaring a variable x of type `num`, a variable y of type `bool`, and a variable z of type `real` corresponds to the following predicate for all states s:

$$\mathsf{is\_num}.(s.x) \wedge \mathsf{is\_bool}.(s.y) \wedge \mathsf{is\_real}.(s.z)$$

The destructor functions, accessing values of type `Val`, are defined as follows:

Definition 3.2.2 THE DESTRUCTOR FUNCTIONS

$\forall n :: \mathsf{evaln}.(\mathsf{NUM}.n)$	$=$	n	evaln
$\forall b :: \mathsf{evalb}.(\mathsf{BOOL}.b)$	$=$	b	evalb
$\forall r :: \mathsf{evalr}.(\mathsf{REAL}.r)$	$=$	r	evalr
$\forall s :: \mathsf{eval\_str}.(\mathsf{STR}.s)$	$=$	s	eval_str
$\forall s : \mathsf{FINITE}.s : \mathsf{eval\_set}.(\mathsf{SET}.s)$	$=$	s	eval_set
$\forall l :: \mathsf{eval\_list}.(\mathsf{LIST}.l)$	$=$	l	eval_list
$\forall t :: \mathsf{eval\_tree}.(\mathsf{TREE}.t)$	$=$	t	eval_tree

Note that these destructor functions are all partial functions defined as total ones by using the mechanism of leaving values outside the correct domain unspecified. As a consequence, nothing definite can be proved about e.g. $\mathsf{evalb}.(\mathsf{NUM}.n)$. Therefore, before one can conclude that e.g. $\mathsf{NUM}.(\mathsf{evaln}.v) = v$ one has to prove $\mathsf{is\_num}.v$. The following theorems capture this for all possible values in `Val`.

Theorem 3.2.3 CONSTRUCTING DE-CONSTRUCTED VALUES

$\forall v$	$: \mathsf{is\_num}.v :$	$\mathsf{NUM}.(\mathsf{evaln}.v)$	$=$	v	consn_o_evaln_IS_id
$\forall v$	$: \mathsf{is\_bool}.v :$	$\mathsf{BOOL}.(\mathsf{evalb}.v)$	$=$	v	consb_o_evalb_IS_id
$\forall v$	$: \mathsf{is\_real}.v :$	$\mathsf{REAL}.(\mathsf{evalr}.v)$	$=$	v	consr_o_evalr_IS_id
$\forall v$	$: \mathsf{is\_str}.v :$	$\mathsf{STR}.(\mathsf{eval\_str}.v)$	$=$	v	cons_str_o_eval_str_IS_id
$\forall v$	$: \mathsf{is\_set}.v :$	$\mathsf{SET}.(\mathsf{eval\_set}.v)$	$=$	v	cons_set_o_eval_set_IS_id
$\forall v$	$: \mathsf{is\_list}.v :$	$\mathsf{LIST}.(\mathsf{eval\_list}.v)$	$=$	v	cons_list_o_eval_list_IS_id
$\forall v$	$: \mathsf{is\_tree}.v :$	$\mathsf{TREE}.(\mathsf{eval\_tree}.v)$	$=$	v	cons_tree_o_eval_tree_IS_id

meaning	notation	definition
equality	x eq y	BOOL.$(x = y)$
inequality	x neq y	BOOL.$(x \neq y)$
increment	incr.x	NUM.(evaln.$x + 1$)
addition	x plus y	NUM.((evaln.x) + (evaln.y))
greater than	x gt y	BOOL.((evaln.x) > (evaln.y))
less than	x lt y	BOOL.((evaln.x) < (evaln.y))
greater or equal	x gte y	BOOL.((evaln.x) $\geq$ (evaln.y))
less or equal	x lte y	BOOL.((evaln.x) $\leq$ (evaln.y))
logical truth	True	BOOL.T
logical falsity	False	BOOL.F
conjunction	x And y	BOOL.((evalb.x) $\wedge$ (evalb.y))
disjunction	x Or y	BOOL.((evalb.x) $\vee$ (evalb.y))
implication	x Imp y	BOOL.((evalb.x) $\Rightarrow$ (evalb.y))
negation	Not.x	BOOL.($\neg$ (evalb.x))
universal quantification	Forall $x : W.x : P.x$	BOOL.($\forall x$: evalb.($W.x$) : (evalb.($P.x$)))
existential quantification	Exists $x : W.x : P.x$	BOOL.($\exists x$: evalb.($W.x$) : (evalb.($P.x$)))
construct lists	put.$x.l$	LIST.(CONS.x.(eval_list.l))
head of a list	head.l	hd.(eval_list.l)
tail of a list	tail.l	LIST.(tl.(eval_list.l))
set element	x INv s	BOOL.($x \in$ eval_set.s)
set characteristic	CHFv.s	($\lambda\, p$. BOOL.(CHF.(eval_set.s).p))
finite set	FINITEv.s	BOOL.(FINITE.(eval_set.s))
set cardinality	CARDv.s	NUM.(CARD.(eval_set.s))

Table 3.1: `Val`-lifted operators

Now we have to lift the operations on numerals, booleans, sets, etcetera, to work on the new type `Val`. The most frequently used `Val`-lifted operators are listed in Table 3.1, together with their new names and definitions. As a notational convention:

> *when it is clear from the context that a* `Val`*-lifted operator is used, the traditional notation for that operator will be overloaded.*

We end this section with the definition of several operations we shall need in subsequent chapters. These definitions concern `Val`-lifted operations on `Val`-typed values which return destructed values.

Definition 3.2.4 *INb*
For all x and s of type `Val`:
x INb s = evalb.(x INv s)

Definition 3.2.5 *CHFb*
For all x of type `Val`:
CHFb.s = evalb $\circ$ (CHFv.s)

Definition 3.2.6 *CARDn*
CARDn = evaln $\circ$ CARDv

Definition 3.2.7 *FINITEb*
FINITEb = evalb $\circ$ FINITEv

3.3 State-functions, -expressions, and -predicates

A *state-function* is a function of type `State`$\rightarrow\alpha$, where α is an arbitrary type. For example:

$$(\lambda s.\ \mathsf{evaln}.(s.x)) \tag{3.3.1}$$

is a state-function of type `State`$\rightarrow$`num` that, applied to a state s, evaluates (or destructs) the `Val`-typed value that is held by program-variable x in state s.

A *state-expression* is a state-function with target type `Val`. For example:

$$(\lambda s.\ s.x\ \mathsf{plus}\ s.y) \tag{3.3.2}$$

is a state expression, adding the value of variable x in state s to the value of variable y in state s. The set of all state-expressions will be denoted by `Expr`.

A *state-predicate* is a state-expression the values in whose range have intended type `bool`.

Definition 3.3.1 STATE-PREDICATE *is_BOOL_DEF*
For all $p \in \sigma\rightarrow$`Val`,

state_pred.p = $\forall s ::$ is_bool.($p.s$)

Note that the states in Definition 3.3.1 are assumed to be of arbitrary type σ, but can of course be instantiated with `State`. In order to obtain general definitions, we shall, whenever possible, represents states by arbitrary types that can be instantiated with `State`.

A state-predicate is used to describe a set of states satisfying a certain property. For instance:

$$(\lambda s.\ (s.x\ \mathsf{eq}\ s.y)\ \mathsf{And}\ \mathsf{Not}(s.y\ \mathsf{gte}\ (\mathsf{NUM}.2))) \tag{3.3.3}$$

Definition 3.3.2 State-lifting variables *VAR_EXPR_DEF*
For a variable v of actual type `Val`, and $s \in$ `State`:

$$\mathsf{VAR}.v.s \;=\; s.v$$

Definition 3.3.3 State-lifting constants *CONST_EXPR_DEF*
For a constant c of any type α, and s of arbitrary type σ:

$$\mathsf{CONST}.c.s \;=\; c$$

Definition 3.3.4 State-lifting unary operators *UN_APPLY_DEF*
For an unary operator O of type $\alpha{\rightarrow}\beta$, e of type $\sigma{\rightarrow}\alpha$, and s of type σ:

$$\mathsf{UN\_APPLY}.O.e.s \;=\; O.(e.s)$$

Definition 3.3.5 State-lifting binary operators *BI_APPLY_DEF*
For a binary operator O of type $\alpha{\rightarrow}\beta{\rightarrow}\gamma$, e_1 of type $\sigma{\rightarrow}\alpha$, e_2 of type $\sigma{\rightarrow}\beta$, and s of type σ:

$$\mathsf{BI\_APPLY}.O.e_1.e_2.s \;=\; O.(e_1.s).(e_2.s)$$

Figure 3.1: State-lifting

is a state-predicate describing all states s in which the value of variable x equals the value of variable y, and the value of variable y is not greater than or equal to two.

Note that in other work [Gor89, BW90, Pra95], where program variables can only take values of one and the same type, state-predicates can be declared to have type `State`$\rightarrow$`bool`. In this thesis program variables can take values of different (intended) types. Consequently, since state-predicates are just a special kind of state-expressions and hence can be used at the right hand side of an assignment to a variable having intended type `bool`, state-predicates must have type `State`$\rightarrow$`Val`.

Since state-expressions constitute the basis for constructing programs, and state-functions and state-predicates constitute the basis for reasoning about programs, we obviously need to improve their readability throughout the rest of this thesis, and, if possible, avoid having to use the `Val`-lifted operations from the previous section. Following [Gor89, BW90, Hal91, APP93, Pra95] this is done by state-lifting the various constructs from which state-functions can be build, and use overloading and notational conventions to denote these state-lifted constructs.

In this thesis, variables having actual type `Val`, constants of any type α, unary and binary operators of type $\alpha{\rightarrow}\beta$ and $\alpha{\rightarrow}\beta{\rightarrow}\gamma$ respectively, can all be state-lifted by using one of the functions listed in Figure 3.1.

To improve readability, the notational conventions introduced have to adhere to the traditional notation of the standard operators. We therefore adopt the convention that (on paper):

the state-lifted versions of standard operators are denoted by their traditional notation subscripted with an '$*$'

For example, state-function (3.3.1) on page 24 is denoted by:

$$\mathsf{evaln}_*\,.\,x_* \quad \text{where} \quad \begin{array}{rcl} \mathsf{evaln}_* & = & \mathsf{UN\_APPLY.evaln} \\ x_* & = & \mathsf{VAR}.x \end{array}$$

State-expression (3.3.2) on page 24 is denoted by:

$$x_* \;+_*\; y_* \quad \text{where} \quad +_* \;=\; \mathsf{BI\_APPLY.plus}$$

State predicate (3.3.3) on page 24 becomes:

$$(x_* =_* y_*) \wedge_* \neg_*(2_* \geq_* y_*) \quad \text{where e.g.} \quad \begin{array}{rcl} \wedge_* & = & \mathsf{BI\_APPLY.And} \\ \neg_* & = & \mathsf{UN\_APPLY.Not} \\ 2_* & = & \mathsf{CONST}.(\mathsf{NUM}.2) \end{array}$$

All other unary and binary operators, not appearing in the examples above, are state-lifted similarly using UN_APPLY and BI_APPLY respectively. The universal and existential quantifier cannot be state-lifted using one of the aforementioned functions. Consequently, they are state-lifted separately as follows:

Definition 3.3.6 STATE-LIFTING THE UNIVERSAL QUANTIFIER *eRES_qAND*

For all $W \in \alpha{\rightarrow}\mathtt{Val}$, $P \in \alpha{\rightarrow}\sigma{\rightarrow}\mathtt{Val}$

$$\forall_*\, x : W.x : P.x = (\lambda s.\mathsf{Forall}\, x : W.x : P.x.s)$$

Definition 3.3.7 STATE-LIFTING THE EXISTENTIAL QUANTIFIER *eRES_qOR*

For all $W \in \alpha{\rightarrow}\mathtt{Val}$, $P \in \alpha{\rightarrow}\sigma{\rightarrow}\mathtt{Val}$

$$\exists_*\, x : W.x : P.x = (\lambda s.\mathsf{Exists}\, x : W.x : P.x.s)$$

Although the $*$-notation improves readability, it is often already clear from the context that a state-lifted operator is meant and not its unlifted counterpart. Therefore, we adopt the further convention that:

when it is clear from the context that a state-lifted operator is used, the $*$ can be dropped, and the traditional notation for that operator is overloaded.

Although on paper this kind of overloading usually does not cause confusion, in HOL overloading is not possible, and different names have to be introduced. Summarising, there are three different notations for state-lifted operators:

- the traditional notation, subscripted with an $*$. Use: on paper; when it is not clear from the context that the state-lifted version of the operator is used.
- the traditional notation, overloaded. Use: on paper; when it is clear from the context that the state-lifted version of the operator is used. This one will be the most frequently used.

overloaded notation	meaning	HOL definition	HOL notation
$p = q$	$(\lambda s.\ p.s\ \mathsf{eq}\ q.s)$	EQ_DEF	p EQ q
$p \neq q$	$(\lambda s.\ p.s\ \mathsf{neq}\ q.s)$	nEQ_DEF	p nEQ q
$x + y$	$(\lambda s.\ x.s\ \mathsf{plus}\ y.s)$	ePLUS_DEF	x !+! y
$x < y$	$(\lambda s.\ x.s\ \mathsf{lt}\ y.s)$	LT_DEF	x !<! y
$x > y$	$(\lambda s.\ x.s\ \mathsf{gt}\ y.s)$	GT_DEF	x !>! y
$x \leq y$	$(\lambda s.\ x.s\ \mathsf{lte}\ y.s)$	LTE_DEF	x !<=! y
$x \geq y$	$(\lambda s.\ x.s\ \mathsf{gte}\ y.s)$	GTE_DEF	x !>=! y
true	$(\lambda s.\ \mathsf{True})$	eTT_DEF	true
false	$(\lambda s.\ \mathsf{False})$	eFF_DEF	false
$p \wedge q$	$(\lambda s.\ p.s\ \mathsf{And}\ q.s)$	eAND_DEF	p |/| q
$p \vee q$	$(\lambda s.\ p.s\ \mathsf{Or}\ q.s)$	eOR_DEF	p |\/| q
$p \Rightarrow q$	$(\lambda s.\ p.s\ \mathsf{Imp}\ q.s)$	eIMP_DEF	p |==>| q
$\neg p$	$(\lambda s.\ \mathsf{Not}.p)$	eNOT_DEF	not p
$(\forall i : W.i : P.i)$ $(\forall i : i \in W : P.i)$	$(\lambda s.\ (\mathsf{Forall}\ i : W.i : P.i.s))$	eRES_qAND	|!!|i::W.P i
$(\exists i : W.i : P.i)$ $(\exists i : i \in W : P.i)$	$(\lambda s.\ (\mathsf{Exists}\ i : W.i : P.i.s))$	eRES_qOR	|??|i::W.P i
$\mathsf{CONS}.x.l$	$(\lambda s.\ \ \mathsf{put}.(x.s).(l.s))$	PUT_DEF	PUT x l
$\mathsf{hd}.l$	$(\lambda s.\ \ \mathsf{head}.(l.s))$	HEAD_DEF	HEAD l
$\mathsf{tl}.l$	$(\lambda s.\ \ \mathsf{tail}.(l.s))$	TAIL_DEF	TAIL l
$x \in S$	$(\lambda s.\ x.s\ \mathsf{INv}\ S.s)$	INe_DEF	x INe S

Overloading of the standard operations, when it is clear from the context that p, q, x, y, and S are state predicates of type State→Val.

Table 3.2: Overloading of the standard operators on state-predicates

- the HOL notation, which, due to restricted possibilities of inventing new names in HOL, is not entirely consistent for the state-lifted operators, although an effort to establish the latter has been made. Use: in HOL definitions, theorems, and theories.

It is very important that the reader is well aware of these different notations. To emphasise this Table 3.2 summarises the most frequently used overloaded notations and their meaning for the state-lifted operators in this thesis as well as the notation used in the accompanying HOL theories.

Now that the notational conventions and precise characterisations of the notions state-function, -expression, and -predicate have been ascertained, the rest of this subsection describes some of their properties.

A state-predicate p is said to hold *everywhere* if $p.s$ holds for all states s:

Definition 3.3.8 EVERYWHERE OPERATOR *eSEQ_DEF*
For all $p \in \sigma \rightarrow \texttt{Val}$,

$$[p] \;=\; (\forall s :: \mathsf{evalb}.(p.s))$$

◀

Projection $\restriction$ (Definition $2.8.1_{18}$) can be lifted to the state-function level:

Definition 3.3.9 ON (STATE-)FUNCTIONS *p_Pj_DEF*
For all $s \in \sigma_1 \rightarrow \sigma_2$, $f \in (\sigma_1 \rightarrow \sigma_2) \rightarrow \alpha$, and $V \subseteq \sigma_1$,

$$f \restriction V = (\lambda s.\ (f.(s \restriction V)))$$

◀

A state-function f is *confined* by a set of variables V, denoted by $f\,\mathcal{C}\,V$, if f does not restrict the value of any variable outside V:

Definition 3.3.17 STATE-FUNCTION CONFINEMENT *CONF_DEF*
For all $f \in (\sigma_1 \rightarrow \sigma_2) \rightarrow \alpha$, and $V \subseteq \sigma_1$,

$$f\,\mathcal{C}\,V \;=\; (\forall s, t :: (s \restriction V = t \restriction V) \Rightarrow (f.s = f.t))$$

◀

For example, $x_* +_* 1_* <_* y_*$ is confined by $\{x, y\}$ but not by $\{x\}$. Note that if f is confined by V, f does not contain useful information about variables outside V, or similarly, f does not depend on the values of the variables outside V. The following theorem states this:

Theorem 3.3.18 CONFINEMENT RELATED TO PROJECTION *CONF_EQ_p_Pj*

$$f\,\mathcal{C}\,V \;=\; (f = (f \restriction V))$$

◀

Evidently, state-predicates true and false are confined by any set. Moreover, confinement is preserved by any state-function that is constructed using the state-lifting constructions from definitions 3.3.2 till 3.3.7, see Figure 3.2. As a rule of thumb, any state-function f is confined by $\mathsf{free}.f$, that is, the set of variables occurring free in f. However, $\mathsf{free}.f$ is not necessarily the smallest set which confines f, e.g. $\emptyset$ confines the state-predicate "$0 = x \vee 0 \neq x$". The following theorem states that the confinement operator is monotonic in its second argument.

Theorem 3.3.19 $\mathcal{C}$ MONOTONICITY *CONF_MONO*

$$V \subseteq W \wedge (f\,\mathcal{C}\,V) \Rightarrow (f\,\mathcal{C}\,W)$$

◀

Theorem 3.3.10 *CONF_VAR_EXPR*
For all variables v, and sets V of variables:

$$\frac{v \in V}{(\mathsf{VAR}.v)\,\mathcal{C}\,V}$$

Theorem 3.3.11 *CONF_CONST*
For all constants c of arbitrary type α, and sets V of variables:

$$(\mathsf{CONST}.c)\,\mathcal{C}\,V$$

Theorem 3.3.12 *CONF_UN_APPLY*
For all unary operators O of type $\mathtt{Val}\to\beta$, state-functions f, and sets V of variables:

$$\frac{f\,\mathcal{C}\,V}{(\mathsf{UN\_APPLY}.O.f)\,\mathcal{C}\,V}$$

Theorem 3.3.13 *CONF_BI_APPLY*
For all binary operators O of type $\mathtt{Val}\to\mathtt{Val}\to\gamma$, state-functions f_1, f_2, and sets V of variables:

$$\frac{(f_1\,\mathcal{C}\,V)\ \wedge\ (f_2\,\mathcal{C}\,V)}{(\mathsf{BI\_APPLY}.O.f_1.f_2)\,\mathcal{C}\,V}$$

Theorem 3.3.14 *CONF_FOLDR_BI_APPLY*
For all binary operators O of type $\mathtt{Val}\to\mathtt{Val}\to\mathtt{Val}$, state-functions f, lists of state-functions l, and sets V of variables:

$$\frac{(f\,\mathcal{C}\,V)\ \wedge\ (\forall g\ :\mathsf{is\_el}.g.l:\ (g\,\mathcal{C}\,V))}{(\mathsf{foldr}.(\mathsf{BI\_APPLY}.O).f.l)\,\mathcal{C}\,V}$$

See A.4.1$_{217}$ and A.4.4$_{217}$ for the definitions of is_el and foldr respectively.

Theorem 3.3.15 *CONF_eRES_qAND*
For all $W \in \alpha\to\mathtt{Val}$, $P \in \alpha\to\mathtt{State}\to\mathtt{Val}$, and sets V of variables:

$$\frac{\forall x\ :\ \mathsf{evalb}.(W.x)\ :\ ((P.x)\,\mathcal{C}\,V)}{(\forall_*\, x\ :\ W.x\ :\ P.x)\,\mathcal{C}\,V}$$

Theorem 3.3.16 *CONF_eRES_qOR*
For all $W \in \alpha\to\mathtt{Val}$, $P \in \alpha\to\mathtt{State}\to\mathtt{Val}$, and sets V of variables:

$$\frac{\forall x\ :\ \mathsf{evalb}.(W.x)\ :\ ((P.x)\,\mathcal{C}\,V)}{(\exists_*\, x\ :\ W.s\ :\ P.x)\,\mathcal{C}\,V}$$

Figure 3.2: Confinement is preserved by the state-lifting constructions.

3.4 Actions

In this thesis, the actions of a program can be multiple assignments or guarded multiple assignments. In order to be able to define transformation functions on actions, like for example augmented superposition of assignments or strengthening guards, we need to make a somewhat deeper embedding of actions than Prasetya [Pra95]. As a consequence, the abstract syntax of actions is defined by a recursive data type, and their semantics is defined as a recursive function on this type.

3.4.1 The abstract syntax of actions

The set of all actions is defined by the following recursive data type:

$$\begin{array}{lcl} \texttt{ACTION} & = & \mathsf{ASSIGN}\ (\texttt{Var})\texttt{list}\ (\texttt{Expr})\texttt{list} \\ & | & \mathsf{GUARD}\ \texttt{Expr ACTION} \end{array} \tag{3.4.1}$$

Note that expressions (i.e. elements of `Expr` (page 24)) are not deeply embedded. Hence we have an embedding that is deeper than Prasetya's but that is not the deepest one possible.

To stick to traditional notation, we adopt the convention that:

$\mathsf{ASSIGN}.[x, y, z].[e_1, e_2, e_3]$	is denoted by	$x, y, z := e_1, e_2, e_3$
$\mathsf{GUARD}.g.A$	is denoted by	**if** g **then** A

To check whether an action is a multiple assignment or a guarded action, we have the functions is_assign and is_guard, which are specified by:

Definition 3.4.1 *is_assign*

$$(\forall lv, le ::\ \mathsf{is\_assign}.(\mathsf{ASSIGN}.lv.le)) \ \wedge\ (\forall g, A ::\ \neg(\mathsf{is\_assign}.(\mathsf{GUARD}.g.A)))$$

Definition 3.4.2 *is_guard*

$$(\forall lv, le ::\ \neg(\mathsf{is\_guard}.(\mathsf{ASSIGN}.lv.le))) \ \wedge\ (\forall g, A ::\ \mathsf{is\_guard}.(\mathsf{GUARD}.g.A))$$

◀

To obtain various components of an action, like e.g. its guard, or the variables it assigns to, we have the following functions:

Definition 3.4.3 *assign_vars*

$$\begin{array}{lcl} \forall lv, le ::\ \mathsf{assign\_vars}.(\mathsf{ASSIGN}.lv.le) & = & lv \\ \forall g, A ::\ \mathsf{assign\_vars}.(\mathsf{GUARD}.g.A) & = & \mathsf{assign\_vars}.A \end{array}$$

Definition 3.4.4 *assign_exprs*

$$\begin{array}{lcl} \forall lv, le ::\ \mathsf{assign\_exprs}.(\mathsf{ASSIGN}.lv.le) & = & le \\ \forall g, A ::\ \mathsf{assign\_exprs}.(\mathsf{GUARD}.g.A) & = & \mathsf{assign\_exprs}.A \end{array}$$

◀

Definition 3.4.5 *assign_of*

$$\begin{array}{lll} \forall lv, le :: & \mathsf{assign\_of}.(\mathsf{ASSIGN}.lv.le) & = \quad \mathsf{ASSIGN}.lv.le \\ \forall g, A :: & \mathsf{assign\_of}.(\mathsf{GUARD}.g.A) & = \quad \mathsf{assign\_of}.A \end{array}$$

Definition 3.4.6 *guard_of*

$$\begin{array}{lll} \forall lv, le :: & \mathsf{guard\_of}.(\mathsf{ASSIGN}.lv.le) & = \quad \mathsf{true}_* \\ \forall g, A :: & \mathsf{guard\_of}.(\mathsf{GUARD}.g.A) & = \quad g \wedge_* (\mathsf{guard\_of}.A) \end{array}$$

The operator that models simultaneous execution of its argument assignments, shall be denoted by $\|$. Intuitively, it works like:

$$x, y := 1, 2 \,\|\, w, z := 3, 4 \text{ equals } x, y, z, w := 1, 2, 3, 4$$

Formally it is defined as a partial function on assignments, leaving the result on guarded actions unspecified:

Definition 3.4.7 Simultaneous execution of assignments *SIM*

$$\mathsf{ASSIGN}.lv_1.le_1 \,\|\, \mathsf{ASSIGN}.lv_2.le_2 = \mathsf{ASSIGN}.(lv_1 +\!\!+ lv_2).(le_1 +\!\!+ le_2)$$

Note that this operator could not have been defined without the deeper embedding of actions.

3.4.2 The semantics of actions

The semantics of an action from `ACTION`, is an *executable action* that can change the state of a program. Following [Pra95], we will represent an executable action as a relation on `State`. That is, an executable action a will have the type `State→State→bool`. The interpretation of $a.s.t$ is that t is a possible state resulting from execution of a at state s. For example, the executable action that does not change the value of any variable, and the executable action that forbids any transition, called skip and miracle respectively, are defined by:

Definition 3.4.8 Skip *skip_THM*

$$\mathsf{skip} = (\lambda s.t.\ s = t)$$

Definition 3.4.9 Miracle *MIRA*

$$\mathsf{miracle} = (\lambda s.t.\ \mathsf{false})$$

An action that is always ready to make a transition is called *always enabled*.

Definition 3.4.10 Always Enabled Action *ALWAYS_ENABLED*

$$\Box_{\mathsf{En}} a \;=\; (\forall s :: (\exists t :: a.s.t))$$

Synchronised execution of two executable actions a and b, i.e. a state-transition can be made only if both a and b agree on it, is modelled by the operator $\sqcap$.

Definition 3.4.11 Synchronisation Operator *rINTER*

$$a \sqcap b \;=\; (\lambda s.t.\; a.s.t \wedge b.s.t)$$

Projection $\restriction$ can be lifted to the action level as follows:

Definition 3.4.12 on `Action` *a_Pj_DEF*

$$(a \restriction V).s.t \;=\; a.(s \restriction V).(t \restriction V)$$

The executable action that implements an action a guarded with state-predicate g, can be defined by:

Definition 3.4.13 Executable guarded action *guard_action_DEF*

$$\mathsf{guard}.g.a \;=\; (\lambda s, t.\; (\mathsf{evalb}.(g.s) \Rightarrow a.s.t) \wedge (\neg(\mathsf{evalb}.(g.s)) \Rightarrow \mathsf{skip}.s.t))$$

Note that executing $\mathsf{guard}.g.a$ is state s, results in a skip when the guard g is disabled in state s. Consequently, we can prove the following theorem:

Theorem 3.4.14 Guarded action is always enabled *guarded_action_ALWAYS_ENABLED*

$$\forall g\; a \;::\; \frac{\Box_{\mathsf{En}}\, a}{\Box_{\mathsf{En}}\, \mathsf{guard}.g.a}$$

When defining an executable action that implements multiple assignment we have to be careful. In [Pra95] a single assignment is defined as follows:

$$\mathsf{update}_1.(x, e) \;=\; (\lambda s.t.\; t.x = e.s)$$

$$\mathsf{single\_assignment}_1.x.e \;=\; \mathsf{update}_1.(x, e) \;\sqcap\; \mathsf{skip} \restriction \{x\}^c$$

that is, x is assigned the value of expression e evaluated in state s, and all other variables which are not equal to x stay the same. Extending this definition of single assignments to multiple assignments gives us:

$$\mathsf{assign}_1.lv.le \;=\; \mathsf{foldr}.\sqcap.(\mathsf{skip} \restriction (\mathsf{l2s}.lv)^c).(\mathsf{map}.\mathsf{update}_1.(\mathsf{zip}.lv.le))$$

However, using this definition to assign a different value to the same variable we get the following.

$$
\begin{array}{ll}
 & \mathsf{assign}_1.[v,v].[2_*,3_*] \\
= & \text{(definition of } \mathsf{assign}_1\text{, Definitions A.5.8 and A.4.3 of l2s and zip)} \\
 & \mathsf{foldr}.\sqcap.(\mathsf{skip} \restriction \{v\}^c).(\mathsf{map.update}_1.[(v,2_*),(v,3_*)]) \\
= & \text{(definition of } \mathsf{update}_1\text{, Definitions A.4.4 and A.4.2 of foldr and map)} \\
 & (\lambda s\, t.t.v = 2) \sqcap (\lambda s\, t.t.v = 3) \sqcap (\lambda s\, t.s \restriction \{v\}^c = t \restriction \{v\}^c) \\
= & \text{(logic)} \\
 & (\lambda s\, t.\mathsf{false}) \\
= & \text{(Definition 3.4.9 of miracle)} \\
 & \mathsf{miracle}
\end{array}
$$

The UNITY programming notation [CM89] gives no clear semantics of actions that try to assign a different value to the same variable. However, miraculous actions are not allowed in UNITY programs.

Prasetya [Pra95] has dealt with miraculous actions by defining conditions a program has to satisfy in order for it to be a well-formed UNITY program, and ensuring that progress properties can only be proved for these well-formed UNITY programs. Obviously, including always-enabledness of all actions in the conditions of this well-formedness predicate rules out all miraculous actions.

Here, since we have a deep embedding of actions, we can prevent the construction of miraculous actions, and therefore eliminate the need to deal with them, altogether. To achieve this we have defined the executable action implementing multiple assignment as follows.

First, a function that updates a state by assigning a new value to some variable is defined:

Definition 3.4.15 Updating variables in a state *update_DEF*

$$\mathsf{update}.(x,c).s \;=\; (\lambda y.\ (y = x) \;\rightarrow\; c \mid s.y)$$

Then the executable assignment, assigning the value of state-expression e in some state s to variable x can be defined as:

$$\mathsf{single\_assign}.x.e \;=\; (\lambda s, t.\ t \;=\; \mathsf{update}.(x,(e.s)).s)$$

Generalising this to multiple assignments gives us:

Definition 3.4.16 Executable multiple assignment *assign_DEF*

$$\mathsf{assign}.lv.le \;=\; (\lambda s, t.\ t \;=\; \mathsf{foldr}.\circ.\mathsf{id}.(\mathsf{map.update}.(\mathsf{zip}.lv.(\mathsf{map}.(\lambda e.\ e.s).le))).s)$$

Now, trying to assign a different value to the same variable results in the following:

$$\mathsf{assign}.[v,v].[2_*,3_*].s.t$$

= (Definitions A.4.3 and A.4.2 of zip and map)
$t = \mathsf{foldr}.\circ.\mathsf{id}.[\mathsf{update}.(v,2), \mathsf{update}.(v,3)].s$
= (Definition A.4.4 of foldr)
$t = (\mathsf{update}.(v,2) \circ \mathsf{update}.(v,3) \circ \mathsf{id})\ s$
= (Definition A.2.1 of composition, and id)
$t = \mathsf{update}.(v,2).(\mathsf{update}.(v,3).s)$
= (Definition 3.4.15 of update)
$\forall x ::\ t.x = ((x = v)\ \rightarrow\ 2\ \mid\ ((x = v)\ \rightarrow\ 3\ \mid\ s.x))$

So, variable v is assigned the value of the first expression on the expression-list evaluated in s. Although it is debatable whether this is a desirable semantics for the assignment trying to assign two different values to the same variable, we can prove however that this definition of multiple assignments is always enabled.

Theorem 3.4.17 Multiple assignment is always enabled *actions_semantics*

$\forall lv\ le ::\ \Box_{\mathsf{En}}\ \mathsf{assign}.lv.le$

Now, the function defining the semantics of actions from ACTION can be given. It is called compile, since it sort of compiles an abstract representation of an action into an executable action. It has type:

$\mathsf{compile}\ \in\ \mathtt{ACTION} \rightarrow \mathtt{State} \rightarrow \mathtt{State} \rightarrow \mathtt{bool}$

and is defined as:

Definition 3.4.18 Compiling actions *compile_DEF*

$$\begin{array}{lll} \forall lv, le :: & \mathsf{compile}.(\mathsf{ASSIGN}.lv.le) & = \ \mathsf{assign}.lv.le \\ \forall g, A :: & \mathsf{compile}.(\mathsf{GUARD}.g.A) & = \ \mathsf{guard}.g.(\mathsf{compile}.A) \end{array}$$

For any action $A \in$ ACTION, we shall call compile.A the executable of A. Moreover, we adopt the convention that actions from the set ACTION are denoted by capitals, and that executables are denoted by small letters.

Finally it is not hard to prove that we can only construct actions which are always enabled.

Theorem 3.4.19 *compile_ACTION_ALWAYS_ENABLED*

$\forall A\ :\ A \in \mathtt{ACTION}\ :\ \Box_{\mathsf{En}}\ \mathsf{compile}.A$

3.4.3 The normal form and well-formedness of actions

Actions in ACTION can be finitely nested **if-then** constructs. Obviously, these can all be reduced to one single **if-then** construction, since e.g. if p and q are state-predicates,

then

$$\textbf{if } p \textbf{ then if } q \textbf{ then } A \tag{3.4.2}$$

has the same semantics as

$$\textbf{if } p \wedge_* q \textbf{ then } A \tag{3.4.3}$$

If A is an assignment, then we call 3.4.3 the *normal form* of 3.4.2. Formally, the normal form of an action is defined by the following function:

Definition 3.4.20 NORMAL FORM OF AN ACTION *NF*

$$\begin{array}{lrcl} \forall lv\ le :: & \mathsf{NF}.(\mathsf{ASSIGN}.lv.le) & = & \mathsf{ASSIGN}.lv.le \\ \forall g\ A :: & \mathsf{NF}.(\mathsf{GUARD}.g.A) & = & \mathsf{GUARD}.(g \wedge_* (\mathsf{guard\_of}.A)).(\mathsf{assign\_of}.A) \end{array}$$

◀

There are actions in the set `ACTION` that do not have a well defined semantics in the sense that compiling them yields a value about which nothing definite can be proved. Actions that do not have a well defined semantics are:

- multiple assignments ($\mathsf{ASSIGN}.lv.le$), where ($\#lv \neq \#le$). Since nothing definite can be proved about zip when it is applied to two lists of different lengths, nothing definite can be proved about the the semantics of multiple assignments (definition 3.4.16) when applied to lists of different lengths.

- guarded action of which the state-expression denoting the guard is not a state-predicate. Since nothing definite can be proved about evalb when it is applied to a value from `Val` that does not have intended type `bool`, nothing definite can be proved about the the semantics of guarded actions (definition 3.4.13) having a guard that is not a state-predicate.

Actions in `ACTION` that do have a well-defined semantics are called *well-formed*, and they are characterised as follows:

Definition 3.4.21 WELL-FORMED ACTION *WF_ACTION*

$$\begin{array}{lrcl} \forall lv\ le :: & \mathsf{WF\_action}.(\mathsf{ASSIGN}.lv.le) & = & (\mathsf{length}.lv = \mathsf{length}.le) \\ \forall g\ A :: & \mathsf{WF\_action}.(\mathsf{GUARD}.g.A) & = & (\mathsf{state\_pred}.g \wedge \mathsf{WF\_action}.A) \end{array}$$

◀

Note that actions where the same variable appears more than once in the variable list, do have a well-defined[3] semantics. Consequently, such actions are considered to be well-formed.

[3] Although this semantics may not be desirable.

3.4.4 Properties of actions

A set of variables is V *ignored-by* an action A, denoted by $V \nleftarrow A$, if executing A's executable in any state does not change the values of these variables. Variables in V^{c} *may* however be written by A.

Definition 3.4.22 VARIABLES IGNORED-BY ACTION *dIG_BY_DEF*

$$V \nleftarrow A \;=\; (\forall s, t :: \mathsf{compile}.A.s.t \Rightarrow (s \restriction V = t \restriction V))$$

Evidently, a well-formed assignment ignores all variables not occuring in its list of variables (i.e. at the left hand side of :=). Note that this is not necessarily the smallest set that is ignored by an assignment, e.g. $x := x$ ignores the whole universe of variables. A guarded action **if** g **then** A, ignores the same variables as A.

Theorem 3.4.23 VARIABLES IGNORED-BY ASSIGNMENT *compile_ACTION_ASSIGN_SAT_WC*

$$\frac{\mathsf{WF\_action}.(\mathsf{ASSIGN}.lv.le) \;\wedge\; (\forall v \;:\; \mathsf{is\_el}.v.lv \;:\; v \notin V)}{V \nleftarrow \mathsf{ASSIGN}.lv.le}$$

Theorem 3.4.24 VARIABLES IGNORED-BY GUARDED ACTION *compile_ACTION_GUARD_SAT_WC*

$$\frac{V \nleftarrow A}{V \nleftarrow \mathsf{GUARD}.g.A}$$

A set of variables V is said to be *invisible-to* an action A, denoted by $V \nrightarrow A$, if the values of the variables in V do not influence the result of A's executable, hence A only depends on the variables outside V.

Definition 3.4.25 VARIABLES INVISIBLE-TO ACTION *dINVI_DEF*

$$\begin{aligned} V \nrightarrow A \;=\; \forall s, t, s', t' :: \; & ((s \restriction V^{\mathsf{c}} = s' \restriction V^{\mathsf{c}}) \wedge (t \restriction V^{\mathsf{c}} = t' \restriction V^{\mathsf{c}}) \wedge \\ & (s' \restriction V = t' \restriction V) \wedge \mathsf{compile}.A.s.t) \\ & \Rightarrow \mathsf{compile}.A.s'.t' \end{aligned}$$

A set of variables V is invisible to a well-formed assignment, if non of the variables in V occur in the variable list of the assignment (i.e. at the left hand side of :=), and all expressions in the list of expressions of the assignment (i.e. at the right hand side of :=) do not depend on the values of the variables in V.

A set of variables V is invisible to an action A guarded with guard g, if it is invisible to A and the guard g does not depend on the values of the variables in V.

Theorem 3.4.26 VARIABLES INVISIBLE-TO ASSIGNMENT *compile_ACTION_ASSIGN_SAT_RC*

$$\frac{\mathsf{WF\_action}.(\mathsf{ASSIGN}.lv.le) \wedge (\forall v : \mathsf{is\_el}.v.lv : v \notin V) \wedge (\forall e : \mathsf{is\_el}.e.le : e\,\mathcal{C}\,V^{\mathsf{c}})}{V \twoheadrightarrow \mathsf{ASSIGN}.lv.le}$$

Theorem 3.4.27 VARIABLES INVISIBLE-TO GUARDED ACTION *compile_ACTION_GUARD_SAT_RC*

$$\frac{V \twoheadrightarrow A \wedge g\,\mathcal{C}\,V^{\mathsf{c}}}{V \twoheadrightarrow \mathsf{GUARD}.g.A}$$

The operators $\nleftarrow$ and $\twoheadrightarrow$ are both anti-monotonic in their first argument.

Theorem 3.4.28 $\nleftarrow$ ANTI-MONOTONICITY *dIG_BY_MONO*

$$(V \subseteq W) \wedge (W \nleftarrow A) \;\Rightarrow\; (V \nleftarrow A)$$

Theorem 3.4.29 $\twoheadrightarrow$ ANTI-MONOTONICITY *dINVI_MONO*

$$(V \subseteq W) \wedge (W \twoheadrightarrow A) \wedge (W \nleftarrow A) \;\Rightarrow\; (V \twoheadrightarrow A)$$

3.4.5 Transformations on actions

In this section we shall define two transformations on actions, namely strengthening guards and augmentation. Transforming an action A by strengthening its guard with state-predicate g, is defined as:

Definition 3.4.30 STRENGTHENING GUARDS OF ACTIONS *strengthen_guard*

$$\mathsf{strengthen\_guard}.g.A \;=\; \mathsf{GUARD}.(g \wedge_* \mathsf{guard\_of}.A).(\mathsf{assign\_of}.A)$$

An action (Ac) can be combined with an assignment (As) to yield an augmented action:

Definition 3.4.31 AUGMENTING AN ACTION *augment_DEF*

$$\mathsf{augment}.Ac.As \;=\; \mathsf{GUARD}.(\mathsf{guard\_of}.Ac).((\mathsf{assign\_of}.Ac) \parallel As)$$

When an action Ac is transformed by augmentation to yield $\mathsf{augment}.Ac.As$, we say that Ac is augmented with assignment As. Some properties of strengthening guards and augmentation can be found in Figure 3.3.

Theorem 3.4.32 PRESERVATION OF $\leftarrowtail$ *streng_guard_PRESERVES_IG_BY*

$$\frac{V \leftarrowtail A}{V \leftarrowtail \mathsf{strengthen\_guard}.g.A}$$

Theorem 3.4.33 PRESERVATION OF $\nrightarrow$ *streng_guard_PRESERVES_INVI*

$$\frac{V \nrightarrow A \wedge g\,\mathcal{C}\,V^{\mathsf{c}}}{V \nrightarrow \mathsf{strengthen\_guard}.g.A}$$

Theorem 3.4.34 *streng_augment_COMMUTE*

$$\mathsf{strengthen\_guard}.g.(\mathsf{augment}.Ac.As) = \mathsf{augment}.(\mathsf{strengthen\_guard}.g.Ac).As$$

Theorem 3.4.35 PRESERVATION OF $\leftarrowtail$ *augment_PRESERVES_IG_BY*

$$\frac{V \leftarrowtail Ac \wedge V \leftarrowtail As \wedge \mathsf{is\_assign}.As \wedge \mathsf{WF\_action}.Ac \wedge \mathsf{WF\_action}.As}{V \leftarrowtail \mathsf{augment}.Ac.As}$$

Theorem 3.4.36 *WF_augment*

$$\frac{\mathsf{is\_assign}.As \wedge \mathsf{WF\_action}.Ac \wedge \mathsf{WF\_action}.As}{\mathsf{WF\_action}.(\mathsf{augment}.Ac.As)}$$

Figure 3.3: Properties of strengthening guards and augmentation

3.5 Specification

Reasoning about actions can be done by means of Hoare triples [Hoa69]. If p and q are state-predicates, and A is an action, then $\{p\}\ A\ \{q\}$ means that if A is executed in any state satisfying p, it will end in a state satisfying q. Hoare triples can be defined as follows:

Definition 3.5.1 HOARE TRIPLE *HOAe_DEF*

$$\{p\}\ A\ \{q\} = (\forall s, t :: \mathsf{evalb}.(p.s) \wedge \mathsf{compile}.A.s.t \Rightarrow \mathsf{evalb}.(q.t))$$

Basic laws for Hoare triples, like precondition strengthening and postcondition weakening, can be found in e.g. [Gor89, Dij76, Gri81].

the simplicity of formal manipulations is at least as important as the expressive power of an operator

– Jayadev Misra

Chapter 4

UNITY

In this chapter we give an overview of the UNITY theory and Prasetya's extensions. We shall concentrate on those concepts that are needed in the rest of this thesis. For a more thorough treatment the reader is referred to [CM89, Pra95]. Sections 4.1 through 4.3 discuss the UNITY programming notation. Section 4.4 explains the UNITY specification and proof logic. Sections 4.5 and 4.6 respectively describe Prasetya's alternative progress, and convergence operator. Section 4.7 formalises superposition refinement of UNITY programs.

4.1 UNITY programs

A UNITY program consists of declarations of variables, a specification of their initial values, and a set of actions.

An execution of a UNITY program starts in a state satisfying the initial condition and is an infinite and interleaved execution of its actions. In each step of the execution some action is selected and executed atomically. The selection of actions is weakly fair, i.e. non-deterministic selection constrained by the following fairness rule:

> *Each action is scheduled for execution infinitely often, and hence cannot be ignored forever.*

Note that there is a difference between Dijkstra's guarded commands [Dij76] and UNITY. In the guarded command language, there is no fairness constraint. Consequently, only actions with enabled guards are eligible for execution. If an arbitrary action would be selected for execution, then it would be possible that an action whose guard is *false* is chosen forever, and hence no progress is guaranteed. In UNITY, however, it is not necessary to choose an action of which the guard is *true*. An action that does not change the program state (e.g. because the guard is *false*) may be selected for execution, since – due to the fairness constraint – it can only be executed a finite number of times. An action whose guard is enabled – if such an action exists – is selected eventually for execution, and hence some progress is guaranteed.

For illustration, consider the UNITY program in Figure 4.1. The read and write sections declare, respectively, the read and write variables of the program. The init

prog Example
read $\{a, x, y\}$
write $\{x, y\}$
init true
assign
  if $a = 0$ **then** $x := 1$
[] **if** $a \neq 0$ **then** $x := 1$
[] **if** $x \neq 0$ **then** $y, x := y + 1, 0$

Figure 4.1: The program Example

section specifies the initial states of the program. In Figure 4.1 the initial condition is true, meaning that the program may start in any state. The assign section lists the actions of the programs, separated by the [] symbol. The reader can verify that during the execution of this program eventually $x = 0$ will hold and if $y = C$, then eventually $y > C$ will hold. As far as UNITY is concerned, the actions of this program can be implemented sequentially, fully parallel or anything in between, as long as the atomicity and the fairness condition of UNITY are met. Consequently, in constructing a UNITY program one is encouraged to concentrate on the 'real' problem, and not to worry about ordering and allocation of the actions, as such are considered to be implementation issues.

A UNITY program P can be modelled by a quadruple (A, J, V_r, V_w) where $A \subseteq$ `ACTION` is a set consisting of P's actions, $J \in$ `Expr` is a state-predicate describing the possible initial states of P, and $V_r, V_w \subseteq$ `Var` are sets containing P's read and write variables respectively. The set of all possible quadruples (A, J, V_r, V_w) shall be denoted by `Uprog`. To access each component of such an `Uprog` object, the destructors **a**, **ini**, **r**, and **w** are introduced. They satisfy the following property:

Theorem 4.1.1 `Uprog` Destructors

$$P \in \texttt{Uprog} \;=\; (P \;=\; (\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P))$$

The operators on actions can now be lifted to the program level as follows:

Definition 4.1.2 Variables Ignored-by Program *dIG_BY_Pr*

$$V \not\Leftarrow P = \forall A : A \in \mathbf{a}P : V \not\leftarrow A$$

Definition 4.1.3 Variables Invisible-to Program *dINVI_Pr*

$$V \not\Rightarrow P = \forall A : A \in \mathbf{a}P : V \not\rightarrow A$$

Due to the absence of ordering in the execution of a UNITY program, the parallel composition of two programs can be modelled by simply merging the variables and

⟨*Unity Program*⟩ ::= **prog** ⟨*name of program*⟩
read ⟨*set of variables*⟩
write ⟨*set of variables*⟩
init ⟨*state-predicate*⟩
assign⟨*actions*⟩

⟨*actions*⟩ ::= ⟨*action*⟩ |
| ⟨*actions*⟩ [] ⟨*actions*⟩
| [] *i : quantification*$_i$ *:* ⟨*action*⟩$_i$
⟨*action*⟩ ::= ⟨*assignment*⟩ | ⟨*guarded action*⟩
⟨*assignment*⟩ ::= ⟨*variable-list*⟩ := ⟨*expr-list*⟩
| ⟨*assignment*⟩ || ⟨*assignment*⟩
⟨*variable-list*⟩ ::= ⟨*variable*⟩ {, ⟨*variable*⟩}
⟨*expr-list*⟩ ::= ⟨*expr*⟩ {, ⟨*expr*⟩}
⟨*guarded action*⟩ ::= if⟨*expr*⟩ then ⟨*action*⟩

Where ⟨*expr*⟩ are state-expressions from `Expr`, and ⟨*variable*⟩ are variables in the universe `Var`.

Figure 4.2: Syntax of UNITY programs.

actions of both programs. In UNITY parallel composition is denoted by []. In [CM89] the operator is also called *program union*.

Definition 4.1.4 Parallel Composition *dPAR*

$$P \,[]\, Q \;=\; (\mathbf{a}P \cup \mathbf{a}Q, \mathbf{ini}P \wedge \mathbf{ini}Q, \mathbf{r}P \cup \mathbf{r}Q, \mathbf{w}P \cup \mathbf{w}Q)$$

Parallel composition is reflexive, commutative, associative, and has the identity element $(\emptyset, \mathsf{true}, \emptyset, \emptyset)$.

4.2 The UNITY programming language

The syntax of UNITY programs in BNF notation is displayed in Figure 4.2; it deviates slightly from the one in [CM89] in that the **always** section has been omitted, and the **declare** section has been splitted into **read** and **write** parts.

prog states the name of the UNITY program that is being specified.

read and **write** sections declare the read and write variables of the program respectively. Unless stated otherwise, variables having different names are assumed to be different variables.

init states a state-predicate, specifying the initial condition of the UNITY program.

assign declares a set of *action*s separated by the symbol $[]$. A *quantified action* denotes a set of actions (separated by $[]$) which is obtained by instantiating each $\langle action \rangle_i$ from the *quantified action* with the appropriate instances of the variables bound by $quantification_i$. An important restriction on $quantification_i$, is that it should *not* name program variables whose values may change during program execution. This restriction guarantees that the set of actions of a UNITY program is fixed at all times, i.e. actions can neither be created nor deleted during program execution. An example of a quantified action is:

$$[]i : 0 \leq i < 10 : A[i] := 0 \qquad \text{(sometimes abbreviated by: } []_{0 \leq i < 10}\ A[i] := 0)$$

which generates a set of ten actions that assign 0 to the first 10 elements of some array A.

An action is an element from ACTION, and hence can be either a multiple assignment or a guarded action. Multiple assignments can be composed using the $\|$ operator[1] defined in Section 3.4.1.

4.3 The well-formedness of a UNITY program

Following [Pra95][2], a UNITY program must satisfy four syntactic requirements regarding its well-formedness:

i The program should have at least one action.
ii A write variable is also readable.
iii The actions of a program should only write to the declared write variables.
iv The actions of a program should only depend on the declared read variables.

The notion of *ignored-by* is used to formalise requirement ***iii***. All actions of a program P only write to the declared write variables, if and only if these actions do not change the values of variables that are not declared as write variables, i.e. variables in the set $(\mathbf{w}P)^{\mathsf{c}}$. So, the third requirement is precisely $(\mathbf{w}P)^{\mathsf{c}} \not\Leftarrow P$. The notion of *invisible-to* is used to formalise requirement ***iv***. All actions of a program P only depend on the declared read variables, if and only if changing the values of the variables that are not declared as read variables (i.e. variables in the set $(\mathbf{r}P)^{\mathsf{c}}$) will not influence a's result, hence a only depends on the variables outside $(\mathbf{r}P)^{\mathsf{c}}$ which are precisely the declared read variables. So the last syntactical requirement can be formalised by $(\mathbf{r}P)^{\mathsf{c}} \not\Rightarrow P$. Recall that any UNITY program is an object of type `Uprog`. Accordingly, a predicate Unity can be defined to express the well-formedness of an `Uprog` object. From here on, a "UNITY program" is an object satisfying the predicate Unity.

[1] Note that this is merely syntactic sugar.

[2] Actually in [Pra95] there is also the requirement that all actions should always be enabled. Since all actions from the universe `ACTION` are always enabled (Theorem 3.4.19) this requirement could be dropped.

Definition 4.3.1 Unity *dUNITY*

$$\mathsf{Unity}.P = (\mathbf{a}P \neq \emptyset) \ \wedge \ (\mathbf{w}P \subseteq \mathbf{r}P) \ \wedge \ ((\mathbf{w}P)^{\mathsf{c}} \not\Leftarrow P) \ \wedge \ ((\mathbf{r}P)^{\mathsf{c}} \not\Rightarrow P)$$

◀

Note that the identity element of program composition $[]$, is *not* a well-formed UNITY program.

4.4 UNITY specification and proof logic

UNITY logic is used to specify the correctness expectations or properties of a UNITY program. UNITY specifications, and program properties are built from state-predicates and relations on them. Traditionally, two kinds of program properties are distinguished:

- *Safety properties* stating that some undesirable behaviour does not occur;
- *Progress properties* stating that some desirable behaviour is eventually realised.

Consequently, the UNITY logic contains two basic relations on state-predicates corresponding to these properties. For a UNITY program P and state-predicates $p, q \in$ `Expr`, these are defined by:

Definition 4.4.1 Unless (Safety Property) *UNLESSe*

$$_P\vdash p \ \mathsf{unless} \ q \ = \ (\forall A : A \in \mathbf{a}P : \{p \wedge \neg q\} \ A \ \{p \vee q\})$$

Definition 4.4.2 Ensures (Progress Property) *ENSURESe*

$$_P\vdash p \ \mathsf{ensures} \ q \ = \ (\ _P\vdash p \ \mathsf{unless} \ q) \ \wedge \ (\exists A : A \in \mathbf{a}P : \{p \wedge \neg q\} \ A \ \{q\})$$

◀

Safety properties are described by the unless relation (definition 4.4.1). Intuitively, $_P\vdash p$ unless q implies that once p holds during an execution of P, it remains to hold at least until q holds. Note that this interpretation gives no information whatsoever about what p unless q means if p never holds during an execution.

Progress properties are described by the ensures relation. As can be seen from Definition 4.4.2, $_P\vdash p$ ensures q encompasses p unless q. Furthermore, it ensures that there exists an action that can – and, as a result of the weakly fair execution of UNITY programs, will – establish q.

Note that unless and ensures are defined in terms of properties of single actions, and consequently can be derived directly from the program text. For illustration, consider again the program Example in Figure 4.1. From the program text, it is easily verified that the following properties hold:

$$(\ _{\mathsf{Example}}\vdash (a = X) \ \mathsf{unless} \ \mathsf{false}) \text{ and } (\ _{\mathsf{Example}}\vdash (a = 0) \ \mathsf{ensures} \ (x = 1))$$

The first property states that the program Example cannot change the value of a. The second property describes single-action progress from, $a = 0$ to $x = 1$.

Theorem 4.4.3 Anti-Reflexivity *UNLESS_ANTI_REFL*

$$_P\vdash\ p \text{ unless } \neg p$$

Theorem 4.4.4 Conjunction *STABLEe_CONJ*

$$P:\ \frac{(\circlearrowleft p)\ \wedge\ (\circlearrowleft q)}{\circlearrowleft(p \wedge q)}$$

Theorem 4.4.5 *dIG_BY_and_CONF_IMP_STABLEe*

$$\frac{(V \not\Leftarrow P) \wedge (p\ \mathcal{C}\ V)}{_P\vdash\circlearrowleft p}$$

Theorem 4.4.6 Fixed Point Safety *is_SKIP_IMP_UNLESS*

$$P:\ \frac{\mathsf{FP}.p}{\forall r\ q\ ::\ (p \wedge r) \text{ unless } q}$$

Figure 4.3: Some properties of unless, $\circlearrowleft$, and FP

A state-predicate p is a *stable* predicate in program P, if, once p holds during any execution of P, it will remain to hold forever.

Definition 4.4.7 Stable Predicate *STABLEe*

$$_P\vdash\circlearrowleft p\ =\ _P\vdash p \text{ unless false}$$

Consequently, if p holds initially and is stable in program P, it will hold throughout any execution of P, and hence it is an *invariant* of P. Invariants shall be denoted by $_P\vdash\Box p$.

Definition 4.4.8 Invariant *INVe*

$$_P\vdash\Box J\ = ((\mathbf{ini}P \Rightarrow J) \wedge (\ _P\vdash\circlearrowleft J))$$

A state-predicate p is a *fixed-point* of program P, if, once predicate p holds during the execution of P, the program can no longer make any progress. In other words, once p holds during the execution of P, the program will subsequently behave as skip.

Definition 4.4.9 Fixed Point *FPe_DEF*

$$_P\vdash \mathsf{FP}.p = \forall A\ :\ A \in \mathbf{a}P\ :\ (\forall s\ t\ ::\ (\mathsf{evalb}.(p.s) \wedge \mathsf{compile}.A.s.t) \Rightarrow (s = t))$$

Theorem 4.5.1 unless COMPOSITIONALITY *UNLESSe_PAR_i*

$$({}_P\vdash p \text{ unless } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q) \quad = \quad ({}_{P[]Q}\vdash p \text{ unless } q)$$

Theorem 4.5.2 ensures COMPOSITIONALITY *ENSURESe_PAR*

$$\frac{({}_P\vdash p \text{ ensures } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q)}{{}_{P[]Q}\vdash p \text{ ensures } q}$$

Theorem 4.5.3 FP COMPOSITIONALITY *is_SKIP_PAR*

$$\frac{({}_P\vdash \mathsf{FP}.p) \ \wedge \ ({}_Q\vdash \mathsf{FP}.p)}{{}_{P[]Q}\vdash \mathsf{FP}.p}$$

Figure 4.4: Some properties of parallel compositions.

Figure 4.3 lists some properties of unless , $\circlearrowleft$, and FP that are needed somewhere in the rest of this thesis. For more properties the reader is referred to [CM89, Pra95]. As a notational convention: *when it is clear which program P is meant, it is omitted from a formula.* For example, p unless q is written instead of ${}_P\vdash p$ unless q. For laws:

$$P: \ \frac{\ldots (p \text{ unless } q) \ldots}{r \text{ unless } s} \ \text{ abbreviates } \ \frac{\ldots ({}_P\vdash p \text{ unless } q) \ldots}{{}_P\vdash r \text{ unless } s}$$

Moreover, we want to remind the reader of the notational conventions for state-lifted operators stated in Table 3.2_{27}.

As indicated before, the ensures relation describes single-action progress and therefore does not satisfy transitivity and disjunctivity properties. Consequently, it is not adequate for specifying general progress. To specify general progress properties in UNITY, the *leads-to* operator is used. It is denoted by $\mapsto$, and defined as the smallest transitive and disjunctive closure of ensures . Intuitively, ${}_P\vdash p \mapsto q$ implies that if p holds during an execution of P, then eventually q will hold. The precise definition and properties of $\mapsto$ can be found in [CM89]. They are not given here, since in this thesis Prasetya's [Pra95] variant of $\mapsto$ will be used to specify progress properties.

4.5 Prasetya's $\rightarrowtail$ operator

In [Pra95], Prasetya examines compositionality laws of UNITY's general progress operator ($\mapsto$) with respect to parallel composition []. Although fairly strong results in this area have been obtained by Singh [Sin89a], Prasetya shows that UNITY logic is not adequate to prove even stronger results on the parallel composition of write-disjoint programs (e.g. the Transparency Law). Subsequently, he introduces a new general progress operator with which these stronger results are provable. The operator, called *reach*, is denoted by $\rightarrowtail$, and is – after lifting it to work on state predicates of type `State`$\rightarrow$`Val` – defined (without overloading) as follows:

Definition 4.5.4 REACH OPERATOR *REACHe*
$(\lambda p, q.\ J\ {}_P\!\vdash p \rightarrowtail q)$ is defined as the smallest relation R satisfying:

$$(\boldsymbol{i}).\qquad \frac{p\,\mathcal{C}\,\mathbf{w}P\ \wedge\ q\,\mathcal{C}\,\mathbf{w}P\ \wedge\ ({}_P\!\vdash \circlearrowleft J)\ \wedge\ ({}_P\!\vdash J \wedge_* p\ \mathsf{ensures}\ q)}{R.p.q}$$

$$(\boldsymbol{ii}).\qquad \frac{R.p.q \wedge R.q.r}{R.p.r}$$

$$(\boldsymbol{iii}).\qquad \frac{(\forall i : \mathsf{evalb}.(W.i) : R.(p_i).q)}{R.(\forall_* i : W.i : p_i).q}$$

where $W \in \alpha{\rightarrow}\texttt{Val}$ characterises a non-empty set. ◀

Intuitively, $J\ {}_P\!\vdash p \rightarrowtail q$ means that J is a stable predicate in P and that P can progress from $J \wedge p$ to q. Note that:

- $p \rightarrowtail q$ describes progress made through the *writable part* of program P (viz. p and q are confined by the write variables of P). However, since a program can only make progress on its write variables, this should not be a hindrance [Pra95].
- the predicate J can be used to specify the non-writable part of the program, e.g. assumptions on the environment in which the program operates.

Prasetya proves that this alternative progress relation satisfies the Transparency law, and moreover, that it also satisfies the compositionality laws given by Singh [Sin89a]. Since we do not need these laws directly in this thesis, the reader is referred to [Pra95] for their exact characterisations. Some corollaries of these compositionality results, that we do need in this thesis, are stated in Figure 4.6. Other properties of $\rightarrowtail$ are listed in Figure 4.5. As a notational convention: *when it is clear which program P, or which stable predicate J are meant, these are omitted from a formula.* For example,

$$P, J:\ \frac{\ldots (p\ \mathsf{unless}\ q) \ldots}{r \rightarrowtail s}\ \text{abbreviates}\ \frac{\ldots ({}_P\!\vdash p\ \mathsf{unless}\ q) \ldots}{J\ {}_P\!\vdash r \rightarrowtail s}$$

Again we want to remind the reader of the notational conventions for state-lifted operators stated in Table 3.2_{27}. Moreover, for the confinement constraints we introduce the following convention:

$$p, q\,\mathcal{C}\mathbf{w}P \text{ abbreviates } p\,\mathcal{C}\ \mathbf{w}P\ \wedge\ q\,\mathcal{C}\ \mathbf{w}P$$

Properties $4.5.7_{47}$ through $4.5.13_{47}$ have corresponding laws for $\mapsto$ [CM89]. Property $4.5.5_{47}$ is specific for $\rightarrowtail$ and formally states that progress is made through the writable part of the program under the stability of J. Property $4.5.6_{47}$ states that the Substitution Law can be formally derived for the $\rightarrowtail$ operator. In [CM89], the Substitution Law for $\mapsto$ is stated as an axiom. This axiom was a source of anxiety because it was found that it makes the logic inconsistent. However, Sanders [San91] proposed

Theorem 4.5.5 Stable Background and Confinement *REACHe_IMP_STABLE* *REACHe_IMP_CONF*

$$P: \quad \frac{J \vdash p \rightarrowtail q}{\circlearrowleft J \ \wedge \ p,q \,\mathcal{C}\, \mathbf{w}P}$$

Theorem 4.5.6 Substitution *REACHe_SUBST*

$$P,J: \quad \frac{p,s \,\mathcal{C}\, \mathbf{w}P \ \wedge \ [J \wedge p \Rightarrow q] \ \wedge \ (q \rightarrowtail r) \ \wedge \ [J \wedge r \Rightarrow s]}{p \rightarrowtail s}$$

Theorem 4.5.7 Introduction *REACHe_ENS_LIFT,REACHe_IMP_LIFT*

$$P,J: \quad \frac{p,q \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (\circlearrowleft J) \ \wedge \ ([J \wedge p \Rightarrow q] \ \vee \ (J \wedge p \text{ ensures } q))}{p \rightarrowtail q}$$

Theorem 4.5.8 Reflexivity *REACHe_REFL*

$$P,J: \quad \frac{p \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (\circlearrowleft J)}{p \rightarrowtail p}$$

Theorem 4.5.9 Transitivity *REACHe_TRANS*

$$P,J: \quad \frac{(p \rightarrowtail q) \ \wedge \ (q \rightarrowtail r)}{p \rightarrowtail r}$$

Theorem 4.5.10 Case distinction *REACHe_DISJ_CASES*

$$P,J: \quad \frac{(p \wedge \neg r \rightarrowtail q) \ \wedge \ (p \wedge r \rightarrowtail q)}{p \rightarrowtail q}$$

Theorem 4.5.11 Cancellation *REACHe_CANCEL*

$$P,J: \quad \frac{q \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (p \rightarrowtail q \vee r) \ \wedge \ (r \rightarrowtail s)}{p \rightarrowtail q \vee s}$$

Theorem 4.5.12 Progress Safety Progress (PSP) *REACHe_PSP*

$$P,J: \quad \frac{r,s \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (r \wedge J \text{ unless } s) \wedge \ (p \rightarrowtail q)}{p \wedge r \rightarrowtail (q \wedge r) \vee s}$$

Theorem 4.5.13 Disjunction *REACHe_GEN_DISJe*

$$P,J: \quad \frac{(\forall i : i \in W : p.i \rightarrowtail q.i)}{(\exists i : i \in W : p.i) \rightarrowtail (\exists i : i \in W : q.i)} \quad \text{if } W \neq \emptyset$$

Figure 4.5: Properties of $\rightarrowtail$.

Theorem 4.5.14 *REACHe_PAR_SKIPe_IMP_REACHe*

$$\frac{J\ {}_{P_1}\!\vdash p \qquad q \wedge\ {}_{P_2}\!\vdash \mathsf{FP}.(\neg q) \wedge\ {}_{P_2}\!\vdash\ J}{J\ {}_{P_1 [\!] P_2}\!\vdash p \qquad q}$$

Theorem 4.5.15 *REACHe_WHILE_r_PAR_SKIPe_r_IMP_REACHe_PAR*

$$\frac{J\ {}_{P_1}\!\vdash p \qquad q \wedge\ {}_{P_2}\!\vdash \mathsf{FP}.r \wedge\ {}_{P_2}\!\vdash\ J \wedge\ {}_{P_1}\!\vdash r \ \mathsf{unless}\ q \wedge (p \Rightarrow r) \wedge (r\,\mathcal{C}\,\mathbf{w}(P_1 [\!] P_2))}{J\ {}_{P_1 [\!] P_2}\!\vdash p \qquad q}$$

Theorem 4.5.16 *REACHe_and_STABLEe_r_PAR_SKIPe_r_IMP_REACHe_PAR*

$$\frac{J\ {}_{P_2}\!\vdash p \qquad q \wedge\ {}_{P_1}\!\vdash \mathsf{FP}.r \wedge\ {}_{P_2}\!\vdash\ r \wedge\ {}_{P_1}\!\vdash\ J \wedge\ {}_{P_1}\!\vdash\ r \wedge (p \Rightarrow r) \wedge (r\,\mathcal{C}\,\mathbf{w}(P_1 [\!] P_2))}{J\ {}_{P_1 [\!] P_2}\!\vdash p \qquad q}$$

Figure 4.6: Some compositionality properties of $\rightarrowtail$.

an extension of the $\mapsto$ operator for which the Substitution Law can be derived, and hence guaranteed consistency of the logic. With respect to deriving the Substitution Law, Prasetya's and Sanders' progress operator are quite similar, although, being the primary reason for its introduction, Prasetya's operator is more suitable to derive compositionality results [Pra95].

We end this section by presenting the well-founded-induction principle for $\rightarrowtail$, which (for $\mapsto$ in [CM89]) is a standard technique to prove general progress properties.

Theorem 4.5.17 Bounded Progress *REACHe_WF_INDUCT*
For a well-founded relation $\prec$ over some set W, and metric $M \in \mathtt{State}{\rightarrow}W$:

$$P, J:\ \frac{q\,\mathcal{C}\,\mathbf{w}P\ \wedge\ (\forall m \in W : p \wedge (M = m) \rightarrowtail (p \wedge (M \prec m)) \vee q)}{p \rightarrowtail q}$$

note the overloading: $(M = m)$ and $(M \prec m)$ denote $(\lambda s.\ (M.s)\ \mathsf{eq}\ m)$ and $(\lambda s.\ (M.s) \prec m)$ respectively.

Intuitively this principle states the following. Suppose we have a *well-founded* relation $\prec$ over a set W, i.e. it is not possible to construct an infinite sequence of ever decreasing values in W, then a program P can, from p, either

- make progress to q, or
- maintain p while decreasing the value of m with respect to a $\prec$

Consequently, since $\prec$ is well-founded, it is not possible to decrease m forever, and hence eventually q will be established.

Theorem 4.5.18 DISJUNCTION *CONe_GEN_DISJ*

$$P, J: \quad \frac{(\forall i : i \in W : p.i \rightsquigarrow q.i)}{(\exists i : i \in W : p.i) \rightsquigarrow (\exists i : i \in W : q.i)} \quad \text{if } W \neq \emptyset$$

Theorem 4.5.19 CONJUNCTION *CONe_CONJ*
For all *non-empty* and *finite* sets W:

$$P, J: \quad \frac{(\forall i : i \in W : p.i \rightsquigarrow q.i)}{(\forall i : i \in W : p.i) \rightsquigarrow (\forall i : i \in W : q.i)}$$

Theorem 4.5.20 BOUNDED PROGRESS *CONe_WF_INDUCT*
For a well-founded relation $\prec$ over some set A, and metric $M \in \texttt{State} \rightarrow A$:

$$P, J: \quad \frac{(q \rightsquigarrow q) \ \wedge \ (\forall m \in A : p \wedge (M = m) \rightsquigarrow (p \wedge (M \prec m)) \vee q)}{p \rightsquigarrow q}$$

Theorem 4.5.21 ITERATION *Iterate_thm_CONe*
For arbitrary sets W,

$$P, J, L: \quad \frac{\begin{array}{c}(\circlearrowright((\forall x : x \in L : Q.x) \wedge J)) \wedge (\forall x : x \in L : Q.x \,\mathcal{C}\, \mathbf{w}P) \\ L \subseteq W \Rightarrow ((f.L) \subseteq W \ \wedge \ (\forall x : x \in L : Q.x) \rightsquigarrow (\forall x : x \in f.L : Q.x))\end{array}}{\forall n \ L : L \subseteq W \Rightarrow (\forall x : x \in L : Q.x) \rightsquigarrow (\forall x : x \in \mathsf{iterate}.n.f.L : Q.x)}$$

Figure 4.7: Properties of $\rightsquigarrow$.

4.6 Self-stabilisation and Prasetya's $\rightsquigarrow$ operator

The notion of self-stabilisation was first introduced by Dijkstra in [Dij74]. Roughly speaking, a self-stabilising program is a program which is capable of recovering from arbitrary transient failures of the environment in which the program is executing. Obviously such programs are very useful, although the requirement to allow *arbitrary* failures may be too strong. A more restricted form of self-stabilisation, called convergence, allows a program to recover only from certain failures. In [Pra95], a convergence operator is defined in terms of $\rightarrowtail$. The operator is denoted by $\rightsquigarrow$ and defined as follows:

Definition 4.6.1 CONVERGENCE *CONe*

$$J \ {}_P\vdash p \rightsquigarrow q \ \triangleq \ q \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (\exists q' :: (J \ {}_P\vdash p \rightarrowtail q' \wedge q) \ \wedge \ ({}_P\vdash\circlearrowright(J \wedge q' \wedge q)))$$

A program P *converges* from p to q under the stability of J (i.e. $J \ {}_P\vdash p \rightsquigarrow q$), if, given that ${}_P\vdash\circlearrowright J$, the program P started in p will eventually find itself in a situation where q holds and will remain to hold. Intuitively, a program P for which this holds can recover from failures which preserve the validity of p and the stability of J. The necessity of the predicate q' in the definition of $\rightsquigarrow$ is explained in [Pra95] as follows.

Theorem 4.6.2 Convergence Implies Progress *CONe_IMP_REACHe*

$$P, J : \frac{p \rightsquigarrow q}{p \rightarrowtail q}$$

Theorem 4.6.3 Substitution *CONe_SUBST*

$$P, J : \frac{p, s \,\mathcal{C}\, \mathbf{w}P \wedge [J \wedge p \Rightarrow q] \wedge \ (q \rightsquigarrow r) \ \wedge [J \wedge r \Rightarrow s]}{p \rightsquigarrow s}$$

Theorem 4.6.4 Introduction *CONe_ENSURES_LIFT, CONe_IMP_LIFT*

$$P, J : \frac{p, q \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (\circlearrowright J) \ \wedge \ (\circlearrowright (J \wedge q)) \ \wedge \ ([J \wedge p \Rightarrow q] \ \vee \ (p \wedge J \text{ ensures } q))}{p \rightsquigarrow q}$$

Theorem 4.6.5 Reflexivity *CONe_REFL*

$$P, J : \frac{p \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (\circlearrowright J) \ \wedge \ (\circlearrowright (J \wedge p))}{p \rightsquigarrow p}$$

Theorem 4.6.6 Transitivity *CONe_TRANS*

$$P, J : \frac{(p \rightsquigarrow q) \ \wedge \ (q \rightsquigarrow r)}{p \rightsquigarrow r}$$

Theorem 4.6.7 Case distinction *CONe_DISJ_CASES*

$$P, J : \frac{(p \wedge \neg r \rightsquigarrow q) \ \wedge \ (p \wedge r \rightsquigarrow q)}{p \rightsquigarrow q}$$

Theorem 4.6.8 Accumulation *CON_SPIRAL*

$$P, J : \frac{(p \rightsquigarrow q) \ \wedge \ (q \rightsquigarrow r)}{p \rightsquigarrow q \wedge r}$$

Theorem 4.6.9 Stable Strengthening *CONe_STAB_MONO_GEN*

$$P : \frac{q \,\mathcal{C}\, \mathbf{w}P \ \wedge \ (\circlearrowright (J_1 \wedge J_2)) \ \wedge \ J_1 \vdash p \rightsquigarrow q}{(J_1 \wedge J_2) \vdash p \rightsquigarrow q}$$

Theorem 4.6.10 Stable Shift *CONe_STABLE_SHIFT*

$$P : \frac{p' \,\mathcal{C}\, \mathbf{w}P \wedge (\circlearrowright J) \ \wedge \ (J \wedge p' \vdash p \rightsquigarrow q)}{J \vdash p' \wedge p \rightsquigarrow q}$$

Figure 4.8: More properties of $\rightsquigarrow$.

Suppose that a program P can progress from p to q. However, P may not remain in q immediately after the first time q holds. Instead, P may need several iterations before it finally remains within q. This is encoded by requiring that P converges to a predicate stronger than q. This predicate does not need to be fully described. It suffices to know that it implies q. As a consequence, Prasetya's notion of convergence is what by Burns, Gouda, and Miller called *pseudo-stabilisation* [BGM90].

Figures 4.8 and 4.7 list properties of the convergence operator. Most properties are analogous to those of $\rightarrowtail$. There is, however, one property that is satisfied by $\rightsquigarrow$ but not by $\rightarrowtail$ nor $\mapsto$, viz. Conjunctivity. Prasetya [Pra95] shows that, exploiting the conjunctivity of convergence, additional compositionality results for write disjoint programs can be proved, and a stronger induction principle than bounded progress (named round decomposition) can be formulated for $\rightsquigarrow$. Again, since we do not need these theorems directly, the reader is referred to [Pra95] for their exact characterisation.

4.7 Refining UNITY programs by superposition

In UNITY, refining programs by superposition is viewed as follows [CM89, pages 163-167]. Given a program P called the *underlying program*, variables of which are called *underlying variables*. It is required to transform the underlying program such that all its properties are preserved. The transformation, called *superposition*, on program P consists of introducing new variables called *superposed variables*, and then transforming the underlying program P such that the assignments of the underlying variables remain unaffected, though assignments to superposed variables may use the values of underlying variables. Consequently, superposition is a program transformation that adds new functionality to an algorithm in the form of additional variables and assignments to these variables.

It is recognised in [CM89] that the lack of appropriate syntactic mechanisms limits the algebraic treatment of superposition. Consequently, the description of superposition refinement in [CM89] is rather informal. In this thesis, however, we have a deep embedding of actions. That is we have defined the abstract syntax of actions by a recursive data type, and their semantics as a recursive function on this type. As a consequence, we are able to obtain and reason about various components of actions (e.g. guards, assignment variables, etcetera) from the universe `ACTION`, and, to some extent, we are able to compare actions from the universe `ACTION` (i.e. we have a notion of equality of actions). Consequently, we have more appropriate syntactic mechanisms which enable us to give a less informal treatment of superposition.

Within the UNITY framework [CM89] two rules of superposition are distinguished: restricted union superposition, and augmentation superposition.

In [CM89], the *restricted union superposition* rule states that an action A may be added to an underlying program provided that A does not assign to the underlying variables. Here we split this into two parts:

- first, defining the actual transformation of the program;

Let $P \in \texttt{Uprog}$, $A \in \texttt{ACTION}$, and $p, q, J \in \texttt{Expr}$.

Theorem 4.7.1 PRESERVATION OF unless AND ensures *RU_Superpose_PRESERVES_UNLESS* *RU_Superpose_PRESERVES_ENSURES*

$$\frac{p\,\mathcal{C}\,\mathbf{w}P \;\wedge\; q\,\mathcal{C}\,\mathbf{w}P \;\wedge\; \mathbf{w}P \nleftarrow A}{\begin{array}{l}(\ {}_P\!\vdash p \ \textsf{unless}\ q \ \Rightarrow\ {}_{\textsf{RU\_S}.P.A.iA}\!\vdash p \ \textsf{unless}\ q) \\ (\ {}_P\!\vdash p \ \textsf{ensures}\ q \ \Rightarrow\ {}_{\textsf{RU\_S}.P.A.iA}\!\vdash p \ \textsf{ensures}\ q)\end{array}}$$

Theorem 4.7.2 PRESERVATION OF AND *RU_Superpose_PRESERVES_REACH* *RU_Superpose_PRESERVES_CON*

$$\frac{J\,\mathcal{C}\,\mathbf{w}P \;\wedge\; \mathbf{w}P \nleftarrow A}{\begin{array}{l}(J\ {}_P\!\vdash p \ \rightarrowtail\ q \ \Rightarrow\ J\ {}_{\textsf{RU\_S}.P.A.iA}\!\vdash p \ \rightarrowtail\ q) \\ (J\ {}_P\!\vdash p \ \leadsto\ q \ \Rightarrow\ J\ {}_{\textsf{RU\_S}.P.A.iA}\!\vdash p \ \leadsto\ q)\end{array}}$$

Figure 4.9: Restricted Union Superposition preserves properties

- second, proving under which conditions this transformation preserves the properties of the underlying program.

Let A be an action from the universe ACTION, and let iA be a state-predicate describing the initial values of the superposed variables, then a program P can be refined by restricted union superposition using the transformation formally defined by:

Definition 4.7.5 RESTRICTED UNION SUPERPOSITION *RU_superpose_DEF*

Let $A \in \texttt{ACTION}$, $iA \in \texttt{Expr}$, and $P \in \texttt{Uprog}$.

$$\textsf{RU\_S}.P.A.iA \quad = \quad P \ [\!]\ (\{A\}, iA, \textsf{l2s.}(\textsf{assign\_vars.}A), \textsf{l2s.}(\textsf{assign\_vars.}A))$$

Theorems stating that properties are preserved under restricted union superposition are listed in Figure 4.9. Note that instead of requiring that the superposed action A does not write to the underlying variables, it is sufficient to require that the write variables of the underlying program are ignored by the action A.

In [CM89], the *augmentation superposition* rule states that an assignment As that does not assign to the underlying variables can be augmented to any assignment or assignment-part of actions of the underlying program. Again, we first define the actual transformation on the program, and second, prove theorems stating when properties are preserved. Let As be an assignment from the universe ACTION, and let iA be a state-predicate describing the initial values of the superposed variables, then a program P can be refined by augmentation superposition using the transformation rule formally defined by:

Let $As \in \texttt{ACTION}$, $iA \in \texttt{Expr}$, $P \in \texttt{Uprog}$, and $ACs \subseteq \texttt{ACTION}$.

Theorem 4.7.3 PRESERVATION OF unless AND ensures *AUG_Superpose_PRESERVES_UNLESS* *AUG_Superpose_PRESERVES_ENSURES*

$$\frac{\begin{array}{c} p\,\mathcal{C}\,\mathbf{w}P \wedge q\,\mathcal{C}\,\mathbf{w}P \wedge \mathbf{w}P \nleftarrow As \wedge \mathsf{is\_assign}.As \\ \mathsf{WF\_action}.As \wedge \forall Ac : Ac \in ACs : \mathsf{WF\_action}.Ac \end{array}}{\begin{array}{l} (\ {}_P\!\vdash p \ \mathsf{unless}\ q \ \Rightarrow\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\!\vdash p \ \mathsf{unless}\ q) \\ (\ {}_P\!\vdash p \ \mathsf{ensures}\ q \ \Rightarrow\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\!\vdash p \ \mathsf{ensures}\ q) \end{array}}$$

Theorem 4.7.4 PRESERVATION OF AND *AUG_Superpose_PRESERVES_REACH* *AUG_Superpose_PRESERVES_CON*

$$\frac{\begin{array}{c} J\,\mathcal{C}\,\mathbf{w}P \wedge \mathbf{w}P \nleftarrow A \wedge \mathsf{is\_assign}.As \\ \mathsf{WF\_action}.As \wedge \forall Ac : Ac \in ACs : \mathsf{WF\_action}.Ac \end{array}}{\begin{array}{l} (J\ {}_P\!\vdash p \ \rightarrowtail\ q \ \Rightarrow\ J\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\!\vdash p \ \rightarrowtail\ q) \\ (J\ {}_P\!\vdash p \ \leadsto\ q \ \Rightarrow\ J\ {}_{\mathsf{AUG\_S}.P.ACs.As.iA}\!\vdash p \ \leadsto\ q) \end{array}}$$

Figure 4.10: Augmentation Superposition preserves properties

Definition 4.7.6 AUGMENTATION SUPERPOSITION *AUG_superpose_DEF*

Let $As \in \texttt{ACTION}$, $iA \in \texttt{Expr}$, $P \in \texttt{Uprog}$, and $ACs \subseteq \texttt{ACTION}$.

$$\begin{array}{lcl} \mathsf{AUG\_S}.P.ACs.As.iA & = & (\{Ac \mid Ac \in \mathbf{a}P \wedge Ac \notin ACs\} \\ & & \quad \cup \\ & & \{\mathsf{augment}.Ac.As \mid Ac \in \mathbf{a}P \wedge Ac \in ACs\}, \\ & & \mathbf{ini}P \wedge_{*} iA, \\ & & \mathbf{r}P \cup \mathsf{I2s}.(\mathsf{assign\_vars}.As), \\ & & \mathbf{w}P \cup \mathsf{I2s}.(\mathsf{assign\_vars}.As)) \end{array}$$

Theorems stating that properties are preserved under augmentation superposition are listed in Figure 4.10. Note again that instead of requiring that the the assignment As does not write to the underlying variables, it is sufficient to require that the write variables of the underlying program are ignored by As.

4.8 Concluding remarks

In this chapter we have briefly presented UNITY and Prasetya's extensions. As already indicated we have concentrated on those aspects that are needed in the rest of this thesis, and used the version of UNITY as described in [CM89] and extensions as described in [Pra95]. Since the publication of [CM89] several improvements have been made in the theory, some of which are reflected in the *Notes on UNITY* which

can be found at:

http://www.cs.utexas.edu/users/psp/notesunity.html

Moreover, a new version of UNITY has been proposed by J. Misra. In "New UNITY", a new operator **co** is suggested for expressing safety properties, replacing unless. A notion of *transient* predicates form the basis for the progress properties, replacing ensures as the basic operator. We do not consider the new version in this thesis. For more information the reader is referred to [Mis94].

Chapter 5

Embedding UNITY in HOL

As already indicated, this thesis uses and builds upon Prasetya's [Pra95] UNITY embedding in HOL. However, as motivated in Chapter 3, the underlying programming theory used in this thesis differs slightly from Prasetya's [Pra95]. First of all, in order to enable general properties of the UNITY programming language (e.g. actions) itself to be proved, the embedding of actions used in this thesis is deeper than that of Prasetya. Second, the universe of values, the elements of which can be assigned to variables, is defined by the recursive data type `Val` ($3.2.1_{21}$), instead of by some polymorphic type. Therefore, since state-predicates are a special kind of state-expressions and hence can be used in the right hand side of assignments to variables, the type of state-predicates in not longer `State`→`bool` but becomes `State`→`Val`.

This chapter describes the theories built on top of Prasetya's embedding that cope with these slightly different program-theoretic foundations. Section 5.1 presents the theory hierarchy of the resulting embedding, and briefly describes the contents of each theory. Section 5.2 describes the extension of HOL with the type `Val` (i.e. the universe of values), and Section 5.3 explains how functions and operations working on values of type `Val` can be defined in HOL. Section 5.4 through 5.6 respectively describe the modelling of state-functions, actions and UNITY programs, and present some tactics that provide proof-support for proving properties like confinement on state-functions, invisibility of variables to actions, and well-formedness of UNITY programs. Section 5.7, finally, outlines how the UNITY operators in Prasetya's embedding are lifted to handle the new type of state-predicates, and the deeper representation of actions.

5.1 The theory hierarchy

Figure 5.1 shows the theory hierarchy of the UNITY embedding used in this thesis.

WP_UNITY is the theory of which the ancestry consists of Wishnu Prasetya's UNITY embedding [Pra95] (i.e. state-predicates have type `State`→`bool`, and actions are shallowly embedded.)

pvt contains the theorems that constitute an abstract axiomatisation, an induction principle, an "all constructors are distinct" theorem, and a cases theorem for

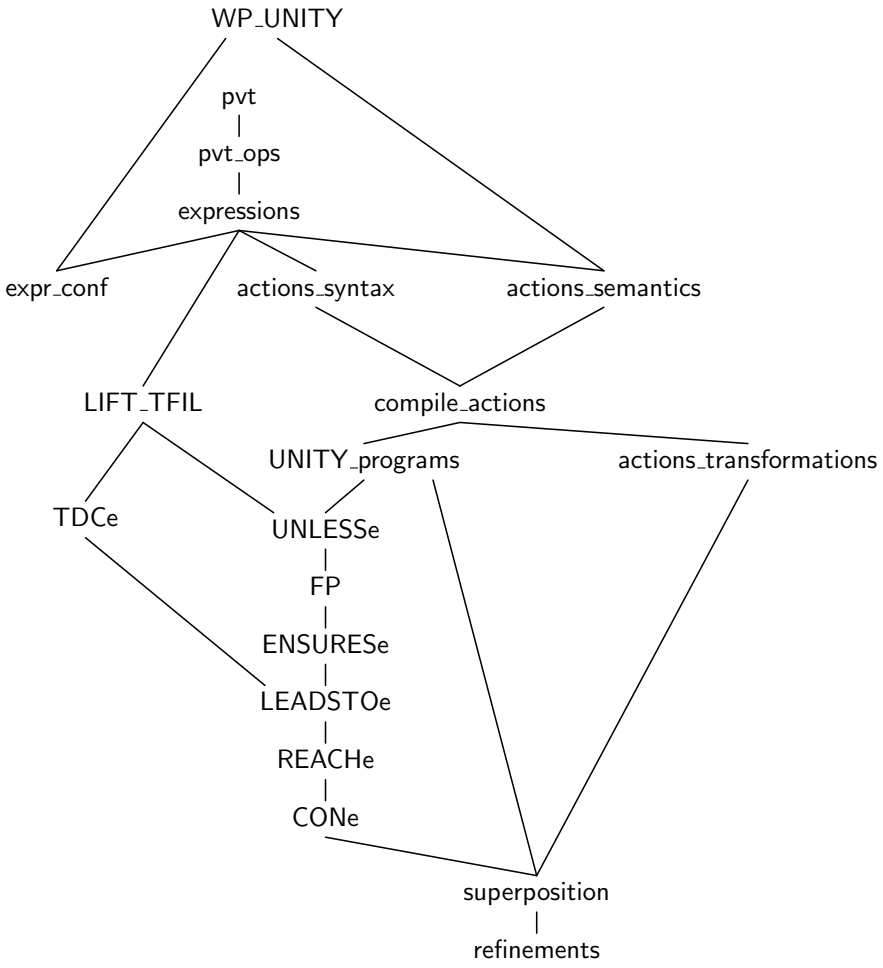


Figure 5.1: Theory hierarchy

the recursive data type `Val`. The theory contains an axiom, the justification of which is proved with HOL, and described in Appendix B.

pvt_ops consists of the intended type checking functions, the destructor functions, and the `Val`-lifted standard operations on values of type `Val` that were discussed in Section 3.2. (see Section 5.3)

expressions constitutes the theory on `State`-lifting the `Val`-lifted operations from theory pvt_ops conform to Section 3.3. (see Section 5.4)

expr_conf Contains theorems $3.3.10_{29}$ through $3.3.16_{29}$. Moreover, proof-support for proving confinement properties of certain state-functions is available in this theory. (see Section 5.4)

actions_syntax defines the data type that models the syntax of actions. Moreover, it contains the definitions of the functions for reasoning about abstract actions (i.e. definitions $3.4.1_{30}$ through $3.4.7_{31}$, $3.4.20_{35}$, and $3.4.21_{35}$). (see Section 5.5)

actions_semantics formalises the behaviour and properties of executable actions conform Section 3.4.2.

compile_actions defines how abstract actions are compiled into executable actions. Moreover, this theory contains theorems stating the properties of actions as delineated in Section 3.4.4, and proof-support for proving these properties of certain actions. (see Section 5.5)

actions_transformations embodies the definitions from Section 3.4.5.

UNITY_programs characterises a compile function for a whole UNITY program, and the notion of well-formedness of a UNITY program. (see Section 5.6)

superposition comprises the formalisation of superposition refinement from Section 4.7.

refinements contains definitions and theorems of a refinement concept that will be described in Chapter 7.

LIFT_TFIL holds definitions and theorems that facilitate the construction of the `Expr`-lifted UNITY operators in terms of the ones from WP_UNITY that work on predicates of type `State`→`bool`. (see Section 5.7)

TDCe encloses theory about transitive, disjunctive closures of `Expr`-lifted operators. (see Section 5.7)

The theories UNLESSe, FP, ENSURESe, LEADSTOe, REACHe, and CONe contain the definitions and corresponding theorems of the UNITY operators $\circlearrowleft$ and unless , FP, ensures , $\mapsto$, $\rightarrowtail$, and $\rightsquigarrow$ respectively, as they are presented in Chapter 4. (see Section 5.7)

5.2 The universe of values `Val`

As indicated in Chapter 3 (page 21), the multi-typed value space used in this thesis is recursively defined by the following data type:

```
Val  =  NUM num
     |  BOOL bool
     |  REAL real
     |  STR string
     |  SET (Val)set
     |  LIST (Val)list
     |  TREE (Val)ltree
```

This type is *not* a concrete recursive data type, since the type operator that is defined (i.e. `Val`) occurs inside the type expressions following the constructors SET, LIST, and TREE. Consequently, this type cannot be added to HOL using the type definition package (see Section 2.1), and has to be defined manually. Consistent to the way it is done in Tom Melham's type definition package [Mel89, GM93], the data type `Val` has to be defined in HOL by deriving an abstract characterisation theorem for it. An adequate and complete abstract characterisation theorem for any inductive type σ, asserts the unique existence of a function g satisfying a recursion equation whose form coincides with the primitive recursion scheme of this type σ (i.e. g is a paramorphism [Mee90]). The abstract characterisation theorem of the data type `Val` is displayed in, and added to HOL as, Axiom $5.3.1_{59}$. Our conviction that this axiom can be proved as a theorem in HOL and hence will not introduce any inconsistencies is based on the justification presented in Appendix B. In this appendix, we describe how we have manually added the following somewhat simpler recursive data type to HOL:

$$\begin{array}{lcl} s\texttt{Val} & = & \mathsf{NUM}\,\texttt{num} \\ & | & \mathsf{SET}\,(\texttt{Val})\texttt{set} \\ & | & \mathsf{LIST}\,(\texttt{Val})\texttt{list} \\ & | & \mathsf{TREE}\,(\texttt{Val})\texttt{ltree} \end{array}$$

Although this data type is simpler, one can see that it contains the same problematic aspects as `Val` (i.e. the constructors SET, LIST and TREE). From the verification activities described in Appendix B, it becomes clear that manually adding the recursive data type `Val` to HOL can be done analoguos to the way s`Val` is added. The representation, abstract characterisation and proof obligations for the additional constructors BOOL, REAL and STRING, will be analogous to those of NUM. However, as the number of constructors increases the proofs become long and tedious. Since all formal proofs necessary to prove the abstract characterisation theorem of the subtype s`Val` have been verified in HOL, we are convinced that the abstract characterisation theorem of `Val` can also be proved. Therefore, we have added it to HOL as an axiom, saving time that was spent on proving theorems of which we were not yet convinced that they held.

5.3 Functions and operations on `Val`

The theory pvt_ops contains the definitions of functions and operators on values of type `Val`. In [Mee90] it is proved that all functions with source type σ are expressible in the form of a paramorphism, i.e. are paramorphisms. Consequently, the intended type checking functions from Definition $3.2.1_{22}$, and the destructor functions from Definition $3.2.2_{22}$ are defined in HOL as paramorphisms on `Val`. For example, the destructor function evaln is added to HOL as follows. First, using Hilbert's choice operator, evaln is defined as a function that has the desired behaviour:

Definition 5.3.2 evaln *evaln_DEF*

$\mathsf{evaln} = \varepsilon g.(\forall n.\ (g.(\mathsf{NUM}.n)\ =\ n))$

◀

Axiom 5.3.1 ABSTRACT CHARACTERISATION OF `Val` *Val_Axiom*

$\forall f_n\ f_b\ f_r\ f_{str}\ f_{set}\ f_l\ f_t.$
$\exists!\mathsf{para}.$
$(\forall n.\ \mathsf{para}.(\mathsf{NUM}.n) = (f_n.n))$
$\wedge$
$(\forall b.\ \mathsf{para}.(\mathsf{BOOL}.b) = (f_b.b))$
$\wedge$
$(\forall r.\ \mathsf{para}.(\mathsf{REAL}.r) = (f_r.r))$
$\wedge$
$(\forall str.\ \mathsf{para}.(\mathsf{STR}.str) = (f_{str}.str))$
$\wedge$
$(\forall set.\ (\mathsf{FINITE}.set) \Rightarrow (\mathsf{para}.(\mathsf{SET}.set) = (f_{set}.(\mathsf{IMAGE}.(\mathsf{split.para}).set))))$
$\wedge$
$(\forall l.\ \mathsf{para}.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split.para}).l))$
$\wedge$
$(\forall t.\ \mathsf{para}.(\mathsf{TREE}.t) = f_t.(\mathsf{map\_tree}.(\mathsf{split.para}).t))$

Then we prove that the function evaln is a paramorphism:

Theorem 5.3.3 evaln IS A PARAMORPHISM ON `Val` *evaln*
$\forall n.\ \mathsf{evaln}.(\mathsf{NUM}.n) = n$

proof of 5.3.3
For arbitrary n we have to prove that:
$\quad \mathsf{evaln}.(\mathsf{NUM}.n) = n$
= (Definition 5.3.2)
$\quad (\varepsilon g.(\forall n.\ (g.(\mathsf{NUM}.n) = n))).(\mathsf{NUM}.n) = n$
$\Leftarrow$(Hilbert's ε (Theorem $2.2.1_{13}$))
$\quad \exists g.\ g.(\mathsf{NUM}.n) = n$
Specialising the conclusion of Axiom $5.3.1_{59}$, by substituting $(\lambda n.\, n)$ for f_n, we can conclude the (unique) existence of a paramorphism para for which it holds that $\mathsf{para}.(\mathsf{NUM}.n) = n$. Now the existentially quantified proof obligation from above can be proved by reducing it with the witness para.
end proof of 5.3.3

Theorem $5.3.3_{59}$ above is exactly the definition of evaln from Definition $3.2.2_{22}$. Note that is has been defined as a partial function by leaving the results of values not in the correct domain unspecified. If one wants to define evaln such that "undefinedness" is explicitly dealt with, first some constant of type `Val` has to be defined about which nothing can be proved, e.g.

```
new_constant {Name = "udef", Ty = ==‘:Val‘==} ;
```

Then evaln can be proved to be the paramorphism:

$\forall n.\ \mathsf{evaln}.(\mathsf{NUM}.n) = n$
$\forall b.\ \mathsf{evaln}.(\mathsf{BOOL}.b) = \mathtt{udef}$
$\forall r.\ \mathsf{evaln}.(\mathsf{REAL}.r) = \mathtt{udef}$
$\forall str.\ \mathsf{evaln}.(\mathsf{STR}.str) = \mathtt{udef}$
$\forall set.\ (\mathsf{FINITE}.set) \Rightarrow \mathsf{evaln}.(\mathsf{SET}.set) = \mathtt{udef}$
$\forall l.\ \mathsf{evaln}.(\mathsf{LIST}.l) = \mathtt{udef}$
$\forall t.\ \mathsf{evaln}.(\mathsf{TREE}.t) = \mathtt{udef}$

by extending the **proof of** 5.3.3 with specialising the universally quantified functions $f_b, f_r, f_{str}, f_{set}, f_l$ and f_t in Axiom $5.3.1_{59}$ with functions that given an argument of the correct type return the value `udef`.

All intended type checking and destructor functions of `Val` are added to HOL similarly. As already indicated in Chapter 3, all other functions and operators on values of type `Val` (like e.g. eq, plus, And) are defined using the constructor and destructor as in Table 3.1_{23}.

5.4 Variables, states, state-functions, and `State`-lifting

The universe of all variables is represented in HOL by a polymorphic type `'var`. Consequently, the universe of program-states is modelled by the type abbreviation

```
val State = ty_antiq (==‘:’var -> Val‘==);
```

For polymorphic type `'a`, state-functions are modelled by:

```
val Func = ty_antiq (==‘:^State -> ’a‘==);
```

Since state-expressions and state-predicates are functions from `State` to `Val`, the universe of state-expressions is modelled in HOL by the type abbreviation

```
val Expr = ty_antiq (==‘:^State -> Val‘==);
```

A state-function is called a *regular state-function* when it consists exclusively of applications of the state-lifting functions VAR, CONST, UN_APPLY, BI_APPLY, $\forall_*$, and $\exists_*$. For example:
`BI_APPLY $gte (VAR x) (CONST (NUM 5))`
is a regular state-function, whereas:
`(\s. (s x) gte (NUM 5))` is not.

A `State`-lifted unary or binary operator on state-functions is *regularly defined* when it is defined using UN_APPLY or BI_APPLY respectively. All unary and binary operators on state-functions in the theory expressions are defined regularly, and proved to have the desired meaning as outlined in Figure 3.2_{27}. For example the regularly `State`-

lifted definitions and corresponding theorems of equality, addition, greater than, and the destructor function of Val-typed values with intended type bool are:

HOL-definition 5.4.1

```
EQ_DEF    |-  $EQ = BI_APPLY $eq
PLUS_DEF  |-  $!+! = BI_APPLY $plus
GT_DEF    |-  $!>! = BI_APPLY $gt
EVALB_DEF |-  EVALB = UN_APPLY evalb
```

HOL-theorem 5.4.2

```
EQ_THM    |- p EQ q = (\s. (p s) eq (q s))
PLUS_THM  |- p !+! q = (\s. (p s) plus (q s))
GT_THM    |- p !>! q = (\s. (p s) gt (q s))
EVALB_THM |- EVALB p = (\s. evalb (p s))
```

◀

Note that, from the general types of UN_APPLY and BI_APPLY (see Definitions $3.3.4_{25}$ and $3.3.5_{25}$), we derive that the types of EQ, !+!, and !>! are the more general type $(\sigma\rightarrow$Val$)\rightarrow(\sigma\rightarrow$Val$)\rightarrow\sigma\rightarrow$Val, in which σ can be instantiated with State.

A State-lifted constant is *regularly defined*, when it is defined using CONST. For example, the regularly State-lifted constants truth, zero, one and empty list are:

HOL-definition 5.4.3

```
eTT_DEF    |-  true = CONST (BOOL T)
ZERO_DEF   |-  ZERO = CONST (NUM 0)
ONE_DEF    |-  ONE  = CONST (NUM 1)
EMPTY_LIST |-  EMPTY_LIST = CONST (LIST [])
```

◀

In this thesis, attempts have been made to construct state-functions as much as possible by applying regular state-functions and regularly defined State-lifted operators to regularly defined State-lifted constants. The reason for this is that these state-functions can be rewritten into regular state-functions, which is desirable since, as will be shown below, the theory expr_conf contains proof-support for proving confinement properties of regular state-functions.

The theory expr_conf contains the theorems $3.3.10_{29}$ till $3.3.16_{29}$ from Section 3.3, and uses these theorems to construct a tactic that generates simple verification conditions for proving confinement of regular state-functions. The tactic is called CONFINEMENT_TAC, and applied to any goal of the form $f\mathcal{C}V$, where f is a regular state-function, and V is a set of variables, it REPEATedly (i.e. continues applying it to all generated subgoals):

MATCH_ACCEPT with theorem $3.3.11_{29}$, which when successful proves the current subgoal

ORELSE

MATCH_MP with theorems $3.3.10_{29}$ or $3.3.12_{29}$ till $3.3.16_{29}$, generating new subgoals of the form:

- $f'\mathcal{C}V$, where f' is a regular state-function that is simpler than f. These subgoals are generated from the hypothesis of the theorem of which the conclusion successfully matches with the current subgoal. From the hypothesis of $3.3.15_{29}$ and the second conjunct of the hypothesis of $3.3.16_{29}$, subgoals of this form are generated by pushing the universal quantification restriction into the assumption.
- $v \in V$, generated from successful attempts to match the current subgoal with the conclusion of theorem $3.3.10_{29}$

Consequently, the set of verification conditions which will be returned by this tactic are of the form $v \in V$, for all free variables v occurring in f.

5.5 Actions

The data type that models the syntax of actions is a concrete recursive data type that can be automatically added to HOL using Tom Melham's [Mel89, GM93] data-type package (see Chapter 2). Below the HOL definition for adding the recursive type to HOL, and the definition of the universe `ACTION` of actions from the theory actions_syntax are displayed.

HOL-definition 5.5.1

```
val ACTION_Axiom
 =
  define_type
   {name = "ACTION_Axiom",
    type_spec =
     `ACTION_TYPE
      = ASSIGN of (('var)list) => ((('var->'a)->'a)list)
      | GUARD of (('var->'a)->'a) => ACTION_TYPE`,
    fixities = [Prefix,Prefix]
   };

val ACTION = ty_antiq(==`:('var,Val)ACTION_TYPE`==) ;
```

◀

Definitions $3.4.1_{30}$ till $3.4.7_{31}$ can now be added to HOL using the SML function `new_recursive_definition`. For example:

HOL-definition 5.5.2

```
val guard_of = new_recursive_definition
 {name = "guard_of",
  def = --`(guard_of ((GUARD g a):^ACTION) = (g |/\| (guard_of a)))
            /\
           (guard_of ((ASSIGN lv le):^ACTION) = true)`--,
  fixity = Prefix,
  rec_axiom = ACTION_Axiom};
```

◀

As an example, the following action (where x and y are variables of intended type `num`, and z is a variable of intended type `bool`):

$$\textbf{if } (x = y) \textbf{ then } x, y := (x + 1), 2 \parallel z := \mathsf{true} \tag{5.5.1}$$

is represented in HOL as:

```
GUARD ((VAR x) EQ (VAR y))
      ( (ASSIGN [x,y] [(VAR x) !+! ONE, TWO])
       SIM
        (ASSIGN [z] [true])
      )
```

where `TWO` is the regularly `State`-lifted constant 2.

An action $A \in$ `ACTION` is called *regular*, when the state-expressions in A's guard and in the right hand side of A's assignment part are all rewritable into regular state-expressions.

The theory compile_actions contains the compile function (Definition 3.4.18), that defines the semantics of abstract actions from `ACTION` in terms of the executable actions (from theory actions_semantics). Furthermore, the theory compile_actions contains the ignored-by and invisible-to properties of actions from Section 3.4.4, and two tactics that generate simple verification conditions for proving the ignored-by and invisible-to properties of regular actions from the universe `ACTION`. The tactics are called IG_BY_TAC and INVI_TAC respectively, and roughly described below.

The tactic IG_BY_TAC, when applied to a goal of the form $V \nleftarrow A$, where V is a set of variables, and $A \in$ `ACTION` is regular, shall REPEATedly:

MATCH_MP with theorems $3.4.23_{36}$ or $3.4.24_{36}$, generating new subgoals.
ORELSE
REWRITE with the definition of WF_Action ($3.4.21_{35}$), and length ($A.4.5_{217}$).

and consequently, returns a set of verification conditions of the form $v \notin V$, for all variables v that will be assigned by action A when it is enabled.

The tactic INVI_TAC, when applied to a goal of the form $V \nrightarrow A$, where V is a set of variables, and $A \in$ `ACTION` is regular, shall REPEATedly try to:

MATCH_MP with theorems $3.4.26_{37}$ or $3.4.27_{37}$, generating new subgoals.
ORELSE
REWRITE with the definition of WF_Action ($3.4.21_{35}$), and length ($A.4.5_{217}$).
ORELSE
CONFINEMENT_TAC is applied

and consequently, returns a set of verification conditions of the form $v \notin V$, for all variables v occurring free in action A.

5.6 UNITY programs

A set of actions from the universe `ACTION`, a set of variables, and finally the universe `Uprog` of UNITY programs are modelled by the following type abbreviation:

```
val ACTIONS = ty_antiq(==‘:^ACTION -> bool‘==);
val Var = ty_antiq(==‘:’var -> bool‘==);
val Uprog = ty_antiq(==‘:^ACTIONS # ^Expr # ^Var # ^Var‘==);
```

The destructors **a**, **ini**, **r**, and **w** used to access the components of an `Uprog` object are called `PROG`, `INIT`, `READ`, and `WRITE` in HOL. Parallel composition of programs ([]) is called `dPAR` in HOL, and is defined using the pred_set library [Mel92] as follows:

HOL-definition 5.6.1

```
|- !(Pr:^Uprog) (Qr:^Uprog).
      Pr dPAR Qr = ((PROG Pr) UNION (PROG Qr),
                    (INIT Pr) |/\| (INIT Qr),
                    (READ Pr) UNION (READ Qr),
                    (WRITE Pr) UNION (WRITE Qr))
```

◀

As a example, the following UNITY program:

prog Example
read $\{a, x, y\}$
write $\{x, y\}$
init true
assign
if $a = 0$ **then** $x := 1$ [] **if** $a > 0$ **then** $x := 1 \parallel y :=$ true

is defined in HOL as follows:

```
val Example = new_definition("Example",
  (--‘Example (a:’var) (x:’var) (y:’var)
    =
   ({GUARD ((VAR a) EQ ZERO) (ASSIGN [x] [ONE])
    ,
     GUARD ((VAR a) !>! ZERO) ((ASSIGN [x] [ONE]) SIM (ASSIGN [y] [true]))
    }
   , true , ({\sf CHF} {x,y}), ({\sf CHF} {a,x,y})
   )‘--));
```

Proving the well-formedness (definition $4.3.1_{43}$) of a UNITY program taken from the universe `Uprog` has now become much simpler because of the availability of the tactics IG_BY_TAC and INVI_TAC.

5.7 Program properties

Properties of UNITY programs are formalised using the operators of the UNITY logic described in Chapter 4. In WP_UNITY these operators work on state-predicates of type `State` to `bool`, and formalise properties of UNITY programs that consist of “shallowly embedded” actions. As already indicated, these operators from WP_UNITY

have to be lifted to handle state-predicates of type `State→Val`, and the deeper representation of actions. This section explains how these operators, including all theorems they satisfy, were lifted such that as much as possible results from WP_UNITY could be re-used.

First some results from WP_UNITY are represented. The universe of state-predicates in WP_UNITY is defined by the type abbreviation:

```
val Pred = ty_antiq (==‘:^State -> bool‘==);
```

State-lifting the standard boolean operators for negation, conjunction, and disjunction is done as follows:

HOL-definition 5.7.1

```
pNOT_DEF |- !(p:^Pred). NOT p = (\s. ~(P s))
pAND_DEF |- !(p:^Pred) (q:^Pred). p AND q = (\s. (p s) /\ (q s))
pOR_DEF  |- !(p:^Pred) (q:^Pred). p OR q = (\s. (p s) \/ (q s))
```

◀

The universes of actions and UNITY programs in WP_UNITY are modelled by the type abbreviations:

```
val action = ty_antiq (==‘:^State -> ^State -> bool‘==);
val actions = ty_antiq (==‘:^action -> bool‘==);
val sUprog = ty_antiq(==‘:^actions # ^Pred # ^Var # ^Var‘==);
```

The basic relations of the UNITY logic in WP_UNITY are defined as follows:

HOL-definition 5.7.2

```
|- !(p:^Pred) (a:^Action) (q:^Pred).
      HOA(p,a,q) = (!s t. p s /\ A s t ==> q t)
|- !(Pr:^sUprog) (p:^Pred) (q:^Pred).
      UNLESS Pr p q = (!a :: PROG Pr. HOA(p AND (NOT q),a,p OR q))
|- !(Pr:^sUprog) (p:^Pred) (q:^Pred).
      ENSURES Pr p q = UNITY Pr /\
                       UNLESS Pr p q /\
                       (?a :: PROG Pr. HOA(p AND (NOT q),a,q))
```

◀

Moreover, WP_UNITY contains a myriad of theorems (including those from Chapter 4) about properties these relations satisfy.

Given this short (and incomplete) overview of the UNITY operators in WP_UNITY, we shall now lift these operators to work on state-predicates of type `Expr`, and UNITY programs of type `Uprog`. First, to lift the UNITY operators, the following function is defined in the theory LIFT_TFIL:

HOL-definition 5.7.3

```
LIFT |- !(R:^Pred -> ^Pred -> bool).
          LIFT R = (\p q. R (EVALB p) (EVALB q))
```

◀

Then the lifted versions of the UNITY operators are defined as follows:

HOL-definition 5.7.4

```
UNLESSe  |- !(Pr:^Uprog)
               UNLESSe Pr = LIFT (UNLESS (compile Pr))
ENSURESe |- !(Pr:^Uprog)
               ENSURESe Pr = LIFT (ENSURES (compile Pr))
```

◀

Finally, it is proved that these lifted definitions have the intended meaning, i.e. the meaning as defined in Chapter 4. In order to prove this, we first have to define the `Expr`-lifted version of Hoare triples that deals with the deeper embedding of actions: (see Definition $3.5.1_{38}$):

HOL-definition 5.7.5

```
HOAe |- !(p:^Expr) (A:^ACTION) (q:^Expr).
          HOAe (p,A,q) = HOA(EVALB p, compile A, EVALB q)
```

◀

Consequently, it can be proved that the definitions of `UNLESSe` and `ENSURESe` coincide with Definitions $4.4.1_{43}$ and $4.4.2_{43}$ respectively:

HOL-theorem 5.7.6

```
UNLESSe_DEF  |- !(Pr:^Uprog) (p:^Expr) (q:^Expr)
                 UNLESSe Pr p q
                 = (!A :: PROG Pr. HOAe(p |/\| (not q),A,p |\/| q))
ENSURESe_DEF |- !(Pr:^Uprog) (p:^Expr) (q:^Expr)
                 ENSURESe Pr p q
                 = dUNITY Pr /\
                   UNLESSe Pr p q /\
                   (?A :: PROG Pr. HOAe(p |/\| (not q),A,q))
```

◀

Lifting the other UNITY operators like $\rightsquigarrow$, $\rightarrowtail$, and $\rightsquigarrow$ is done in the same way as described above. In order to prove that the lifted $\rightsquigarrow$ is indeed the least transitive and disjunctive closure of the lifted version of ensures , some additional theory about transitive and disjunctive closures of relations working on predicates of type `Expr` is constructed in the theory TDCe.

Having lifted all the UNITY operators this way, it has become fairly easy to prove all the theorems, presented in Chapter 4, that are satisfied by the lifted UNITY operators. Because the lifted versions of the UNITY operators are defined in terms of the old ones using the operator `LIFT`, most theorems are proved by rewriting and matching with the corresponding theorems from WP_UNITY.

5.8 Other UNITY tools

The HOL-UNITY library that is part of the HOL package [GM93] is based on the work of Andersen [And92b, And92a, APP93]. Like Prasetya's embedding [Pra95] it is a shallow embedding of UNITY. Unlike Prasetya's embedding, programs are simply defined as lists of actions, and no attention is payed to the acces modes (i.e. read or write) – needed by Prasetya to reason about compositionality – of variables in the program. Consequently, Prasetya's UNITY extensions regarding compositionality and convergence are not available in Andersen's embedding.

UNITY has also been embedded in other theorem provers. Goldschlag [Gol90a, Gol90b, Gol92] describes an embedding in the Boyer-Moore prover [BM88]. Brown and Mery [BM93] report on an embedding of UNITY within the B-method [Abr96]. Chetali [Che95] specified UNITY in the Larch Prover [GHG+93]. Heyd and Crégut [HC96] mechanised UNITY in Coq [CH88]. Paulson [Pau99] has embedded UNITY in Isabelle [Pau94].

A model checker for UNITY has been developed Kaltenbach [Kal96]. Recently, Thirioux [Thi98] has investigated the possibility of decision procedures for the automatic verification of UNITY properties.

Chapter 6

A methodology and a case study

This chapter describes an extension of the UNITY [CM89] methodology for designing and/or mechanically verifying distributed algorithms. Although the methodology is not new and may appear to many as totally obvious, we found it important to describe it in this chapter for three reasons. First, we want to make it clear to the reader what we mean by design and mechanical verification. In particular, which activities are done, what exactly is verified and how. Second, we want to give structure to subsequent chapters where this methodology is applied. Because, the more complex the problem or algorithm, the more difficult it gets to stay focused on what we are actually doing without serious distraction from difficult properties of the problem or algorithm itself. The reader can use this chapter as some sort of reference point that, when studying the verification activities in subsequent chapters, may provide surveyability. Last, we want to be able to analyse which phases of the methodology cost the most time, what is the reason for this, and what can be done to make them more time-efficient.

To illustrate the use of this methodology, Section 6.2 describes a case study to design and verify a converging distributed sorting algorithm. Since the aim of this case study is to highlight the different steps of the methodology, the tackled problem is relatively simple compared to real-life applications and the algorithms verified in later chapters. Section 6.3, finally, reflects on the methodology and the time spent in each separate step.

6.1 The methodology

The methodology described in this chapter consists of 5 phases. Depending on what one wants to achieve, the phases have to be performed in a specific order. We distinguish two scenarios.

(a) One wants to develop (i.e. design and mechanically verify) a new algorithm that solves a given problem P. For example, someone wants a reliable piece of

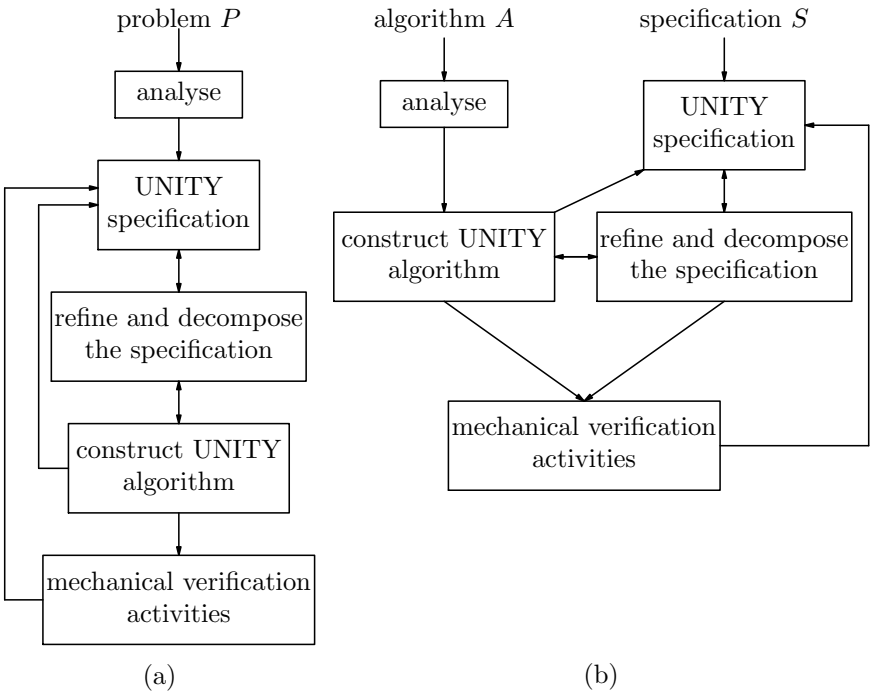


Figure 6.1: (a): designing and mechanically verifying a new algorithm. (b): mechanically verifying an existing algorithm.

software solving problem P. In this scenario, the steps can be done in the order of Figure 6.1(a).

(b) One wants to mechanically verify that an existing algorithm A satisfies specification S. For example, someone questions the claims that an existing algorithm A satisfies a specification S. In this scenario, the steps can be done in the order depicted in Figure 6.1(b).

Below the five phases are roughly described for the scenario developing a new algorithm as well as the scenario verifying an existing algorithm.

Analyse the given problem P, or the existing algorithm A.

(a) Analysing problem P should result in an informal specification and a good understanding of what the algorithm solving P is required to do. During this phase, the designer must try to obtain a clear idea and a good feeling of what his or her customer wants, needs and expects.

(b) Analysing an existing algorithm A should result in a good understanding of its structure, functionality and strategy.

Most people know the feeling one has after completely solving a problem: after solving it, the problem does not seem that difficult anymore as before. Obviously, this is not because the problem was not that difficult after all, but because

one has gone deeply into the problem trying to understand it. It is this feeling that one has to try to obtain after analysing the problem or algorithm, and not only after having completely solved and verified the problem. Obviously this is very difficult because it usually implies spending a lot of time not producing any concrete output. It is our opinion, however, that, although not concrete, the results achieved by obtaining this feeling at this stage are immense. The pace of the rest of the phases is increased and the complexity is reduced.

UNITY specification

(a) Create a formal specification of what the new algorithm is to do (i.e. formally specify which problem is to be solved). A *specification* must focus on the task and not on its eventual implementation, in other words it must specify *what* is to be done rather than *how*. By definition, a *formal specification* is a specification that is written in some specification language or logic, that has a sound mathematical basis [Win90]. Writing a formal specification is central in applying formal methods to the development of programs. Formal specifications help to crystallise vague ideas, to reveal ambiguities and to expose incompleteness in the understanding of the problem. There are many specification languages or logics, which differ mostly in their choice of semantic domain. The best notation to use is the one which relates most to the characteristics of the specific product being developed and the background of the individuals involved. In this thesis (following [Pra95]) the UNITY logic is chosen, for reasons already given in previous chapters.

(b) If the given specification S is not yet a UNITY specification, transform it into one.

When developing a new algorithm (i.e. scenario (a)), the two phases above are usually not done independently. Often they are considered as a single phase resulting in a formal specification of what the new algorithm is to do. However, in order to stress the importance of spending time on analysing the problem, we have made a clear distinction between these two phases.

Refine and decompose the formal specification. Much of program development in the UNITY methodology [CM89] consists of refining specifications (i.e. adding detail to them) and decomposing specifications (i.e. splitting them up into smaller and preferably simpler specifications). Refinement and decomposition commences by proposing a general solution strategy, by means of which the algorithm solves the specified problem. Then the formal specification is refined according to this proposed solution strategy. Finally, the refined specification is decomposed into a set of smaller specifications. Decomposition of the specification must continue until the progress parts of the specification are solely expressed in terms of ensures and unless . The reason for this is that ensures and unless describe one-step progress and safety properties.

(a) When developing a new algorithm these one-step progress and safety properties are used to construct the actions of the algorithm.

(b) When verifying an existing algorithm one-step progress and safety properties are the only ones that can be proved directly of the algorithm.

Sundry basic laws to refine and decompose formulae of the UNITY logic were presented in Chapter 4.

Construct UNITY program representing the new algorithm solving P, or the existing algorithm A.

(a) The idea in this step is that the new algorithm is written in such a way that the actions satisfy the one-step progress and safety properties of the refined and decomposed specification. This may be quite difficult, since refinement and decomposition can result in a myriad specifications. Nevertheless, refined specifications usually give a clear hint as to what kind of actions should or should not occur in the program, since the proposed solution strategy added some detail to how the specified problem could be solved. Moreover, decomposition is often motivated by some ideas related to the implementation of the resulting program. For example, in distributed environments designers may extensively exploit compositionality laws, so that they will have a separate specification for each part of the program instead of a large set of specifications for the complete program.

(b) Constructing a UNITY program representing an existing algorithm, is not a straightforward activity. Obviously, many different UNITY programs can be formulated that model the same algorithm. Finding the best one with respect to readability and reducing proof-effort is not easy.

This ends the formal specification and construction of a UNITY program. The next steps are concerned with the mechanical verification activities.

Represent the program in the HOL embedding of UNITY using the various techniques discussed in Section 5.6. This also includes decisions on how to represent each component of the program. For example, if arrays are used in the program then the representation of these arrays influences the ease with which certain manipulations can be carried out.

Prove that the program is well-formed. To prove the well-formedness of a program (i.e. prove that it satisfies the predicate dUnity), four conditions have to be checked:

(***i***). the program should have at least one action.
(***ii***). the declared write variables should also be declared as read variables.
(***iii***). no variable not declared as a write variable is written by the program
(***iv***). no variable not declared as a read variable can influence the program.

Prove that the program satisfies the specification. First the specification must be formalised in HOL. Second, a proof tree must be constructed according to the refinement and decomposition method from the second step. Closing this proof tree constitutes of proving that the program satisfies this refined and decomposed specification

Notice that designing and/or verifying an algorithm in practice never proceeds by consecutively working through all steps. Program development and verification is by no means a straight-forward one-pass process, but an iterative and non-linear process. A developer cannot make claims to having determined all of the requirements just because the third stage in the development process has been reached. Moreover, such claims should be considered dubious even during post-implementation phases. More than once, it will be inescapable that one has to revise previous steps, because one got an additional idea, because one forgot something or simply because the customer says so.

6.2 Case study: A distributed sorting algorithm

Our starting point is a decentralised communication network of processes in which: every process can execute a local algorithm; every process has a unique label that is used to identify its address; every process has a unique local variable that can store a data value.

Processes are connected via bi-directional communication links. We assume synchronous communication, in the sense that a process can only communicate with exactly one other process at the same time.

Moreover, we have an unstable environment, in which enemies lurk to tamper with the configuration of the network. In particular, we consider two kinds of enemies: firstly, *external agents* that can change the data values of the processes, and secondly, *daemons* that can deactivate and re-activate communication links.

We want to design a distributed program that **sorts** the network. That is a program that does not alter the multi-set of the processes values, and (according to some predefined orders, $\prec_p$ and $\prec_v$, on the labels and the data values respectively) will bring the network in a state in which, for any pair of processes, the order ($\prec_p$) on the labels of these processes is reflected in the order ($\prec_v$) on the data values that reside at these processes.

6.2.1 Analysis and formal specification

A decentralised communication network of processes is modelled by a tuple ($\mathbb{P}$,neighs), where

$\mathbb{P}$ is a finite set of the labels of the processes. Since every process has a unique label this implies that the cardinality of $\mathbb{P}$ equals the number of processes in the network. We assume $\mathbb{P}$ to have cardinality greater than 1, and thus do not consider one process to be a communication network.

neighs is a function that given some process $p \in \mathbb{P}$, gives the set of neighbours of p. In other words, for $p \in \mathbb{P}$, neighs.p constitutes the set of processes that are connected to p by a bi-directional communication link. Obviously, the function neighs has type[1] $\mathbb{P} \rightarrow \mathcal{P}(\mathbb{P})$. Since in HOL all functions are total, we capture

[1] $\mathcal{P}$ indicates the power-set of

this type by requiring that:

$$\forall p \in \mathbb{P} : \ \mathsf{neighs}.p \subseteq \mathbb{P}$$

Since we only consider communication between distinct processes, we shall not allow self-loops, and thus neighs must also satisfy[2]:

$$\forall p \in \mathbb{P}, \ q \in \mathsf{neighs}.p \ : p \neq q$$

Since communication links are bi-directional it also holds that:

$$\forall p, q \in \mathbb{P} : (q \in \mathsf{neighs}.p) = (p \in \mathsf{neighs}.q)$$

The set of (directed) communication links which are present in such a communication network of processes ($\mathbb{P}$, neighs), are formally defined as follows:

Definition 6.2.1 DIRECTED COMMUNICATION LINKS IN THE NETWORK ($\mathbb{P}$, neighs) *links*

$\mathsf{links}.\mathbb{P}.\mathsf{neighs} = \{(p, q) \mid (p \in \mathbb{P}) \wedge (q \in \mathsf{neighs}.p)\}$

For the sake of clarity, in this section we shall be consistent in calling elements of this set *communication links*. The communication links that are active (i.e. up) and over which communication can actually take place, shall be called *connections*. Note that the set of communication links is static, the set of connections is dynamic, and the latter is always $\subseteq$-ed in the first. The formal definition of a decentralised communication network is given below:

Definition 6.2.2 DECENTRALISED COMMUNICATION NETWORK *dNetwork_DEF*

$$\begin{array}{lcl} \mathsf{dNetwork}.\mathbb{P}.\mathsf{neighs} & = & \mathsf{FINITE}.\mathbb{P} \wedge \mathsf{card}.\mathbb{P} > 1 \\ & \wedge & \forall p \in \mathbb{P} : \ \mathsf{neighs}.p \subseteq \mathbb{P} \\ & \wedge & \forall p \in \mathbb{P}, \ q \in \mathsf{neighs}.p \ : p \neq q \\ & \wedge & \forall p, q \in \mathbb{P} : \ (q \in \mathsf{neighs}.p) = (p \in \mathsf{neighs}.q) \end{array}$$

The distribution of the local *variables* that store the data value that resides at a process, is given by a function D that maps a process-label to the *local variable* of that process. Since every process has a unique variable, the following holds:

Definition 6.2.3 *distinct_Sort_Vars*

$$\mathsf{distinct\_Sort\_Vars}.\mathbb{P}.D \ = \ \forall i, j \in \mathbb{P} : i \neq j \Leftrightarrow (D.i) \neq (D.j)$$

---

[2]Note that this condition can also be formalised by: $\forall p \in \mathbb{P} \ : \ p \notin \mathsf{neighs}.p$. We decided to use the other variant because it turned out to be more suitable to prove certain proof obligations.

The initial distribution of the *data values* in the network shall be denoted by the function I, which maps a process-label to the *data value* that is initially stored in the local variable of that process. As a result, in every state s the distribution of data values is given by $(s \circ D)$, and if s_0 is the initial state of the program, $I = (s_0 \circ D)$.

A network is defined to be sorted in some state s, when the following property is satisfied:

Definition 6.2.4 SORTED NETWORK *Sorted_DEF*

$$\mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v.s = \forall i, j \in \mathbb{P} : i \prec_p j \Rightarrow ((s \circ D).i) \prec_v ((s \circ D).j)$$

---

where $\prec_p$ and $\prec_v$ are orders on the process labels and the data values respectively.

Let us start with analysing which properties the order $\prec_p$ must satisfy under the assumption that $\prec_v$ is a total order.

First, it must be anti-symmetric. Consider a network in which every process contains a different data value, that is for all states s:

$$\forall i, j \in \mathbb{P} : (i \neq j) \Rightarrow ((s \circ D).i) \neq ((s \circ D).j)$$

Suppose that $\prec_p$ is not anti-symmetric, so there are processes i and j such that $(i \neq j) \wedge (i \prec_p j) \wedge (j \prec_p i)$ holds. From $(i \neq j)$ we deduce that $((s \circ D).i) \neq ((s \circ D).j)$, and from the anti-symmetric property of $\prec_v$ we derive that $((s \circ D).i) \prec_v ((s \circ D).j)$ and $((s \circ D).j) \prec_v ((s \circ D).i)$ cannot both hold at the same time. Consequently, the network cannot be sorted and thus $\prec_p$ must be anti-symmetric. Second, $\prec_p$ must be transitive, for again consider a network in which every process contains a different data value. Suppose that $\prec_p$ is not transitive, so there are i, j and k $(i \neq j \neq k)$ such that $i \prec_p j$ and $j \prec_p k$ and $k \prec_p i$ simultaneously hold. Again it can be concluded from the anti-symmetry property of $\prec_v$ that it is impossible to sort the network. Whether $\prec_p$ is reflexive, anti-reflexive, or neither makes no difference. Because if $i \prec_p i$ holds, the right-hand side of (6.2.4), restricted to the case that $i = j$, is valid since $\prec_v$ is reflexive; and if $\neg(i \prec_p i)$ holds then it is trivially valid. Finally, we assume that $\prec_p$ is non-empty, since if $\prec_p = \emptyset$ then (6.2.4) is a tautology and consequently there is no use in constructing a sorting program.

Now we shall analyse the environment in which the program will operate. Thus far, we have made two assumptions about the environment. First, we assumed an unstable environment in which external agents can change the data values of the processes, and daemons can tamper with the status (i.e. up or down) of the communication links. Second, we assumed a network of processes, in which two processes can only compare their data values if they have a connection (i.e. active communication link) between them, and a process can only compare its value with one other value at the same time (i.e. synchronous communication). Consequently, in order to sort the network, the order $\prec_v$ has to remain a total order on the data values in the network. Moreover, a sufficient number of data values have to be compared (i.e. a sufficient number of connections must be present) and appropriate actions must be taken according to the result of this comparison. Of course the way these values are compared and which actions will be taken accordingly are matters of *how* the

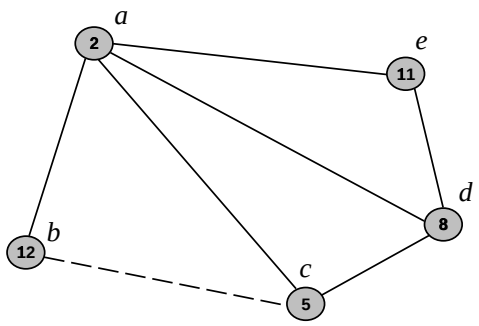


Figure 6.2: A network which cannot be sorted.

program will achieve *what* it is to do, and must not be part of the specification. But conditions on the environment in which the program is required to operate are matters of *what* the program is to do, viz. under what circumstances it must fulfil its requirements. Although at this stage, we may not make any assumptions whatsoever on how the program will achieve the requested results, nevertheless we have to take full account of the limitations the environment imposes upon the possible implementations of the program, by specifying and if necessary strengthening the properties of this environment. It is obvious that, in this case, we have to curtail the set of possible failures that can change the environment, and settle for convergence instead of self-stabilisation. First of all, we can only allow external agents that change data values in such a way that $\prec_v$ stays a total order on the data values. Second, we cannot allow arbitrary communication links to fail. Consider for example the network in Figure 6.2, where the labels (written above the processes) are characters, and the data values (written inside the processes) are numbers. As a result of an environmental failure, the communication link between processes b and c is down. Let $\prec_p$ and $\prec_v$ be the lexicographic order on characters and the less-than-or-equal ($\leq$) order on numbers respectively. There exists no implementation which does not alter the multi-set of the processes' values and can sort this network, since:

- the values of processes a, c, d and e are already sorted according to Definition (6.2.4), so these processes shall not undertake any action[3]
- the same holds for the processes a and b
- process b cannot compare its value with any of processes c, d and e, so nothing will be done by process b either.

Consequently, we must formalise a condition which states when there are still enough connections left for any implementation to sort the network. This condition then imposes a restriction on the set of failures from which the convergent program can recover. Moreover, we need to make the status (i.e. up or down) of the available communication links dependent of the state such that the presence of connections becomes dynamic. To establish the latter, we introduce variables $aC.i.j$ for each

[3]Note that by concluding that no action will be undertaken we assume local convergence.

$(i, j) \in \mathsf{links}.\mathbb{P}.\mathsf{neighs}$ of type boolean, and model link failures as follows:

$$\begin{array}{lcll} s.(aC.i.j) & = & \mathsf{true}, & \text{if link } (i,j) \text{ is up (i.e. an } \mathbf{a}\text{ctive } \mathbf{C}\text{onnection) in state } s \\ s.(aC.i.j) & = & \mathsf{false}, & \text{if link } (i,j) \text{ is down in state } s \end{array}$$

The state-predicate stating that (i, j) is an active connection is characterised by:

Definition 6.2.5 ACTIVE CONNECTION *AC_DEF*

$$\mathsf{AC}.i.j.s \;=\; i \in \mathsf{neighs}.j \wedge s.(aC.i.j)$$

◀

For all states s we define the set of active connections as follows:

Definition 6.2.6 SET OF ACTIVE CONNECTIONS *ACs_DEF*

$$\mathsf{ACs}.s = \{(i,j) \mid \mathsf{AC}.i.j.s\}$$

◀

A minimal and sufficient condition on the connections must imply that if the network is not yet sorted, then there must always be processes which recognise that their values are not sorted; in other words, among the pairs of connected processes whose labels are ordered by $\prec_p$, there must at least be one pair whose values are out-of-order. For, if this condition is not satisfied, it will always be possible to create an example as above in which the network cannot be sorted. Before this condition is formalised, first the definition of a Wrong Pair of processes is given.

Definition 6.2.7 WRONG PAIR OF PROCESSES *WP_DEF*

$$\mathsf{WP}.i.j.s = (i \prec_p j) \wedge \neg(((s \circ D).i) \prec_v ((s \circ D).j))$$

◀

Obviously,

Theorem 6.2.8 *NOT_Sorted_IMP_EXISTS_WP*

$$(\neg\mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v.s) \Leftrightarrow \exists i\, j \;:\; i \in \mathbb{P} \wedge j \in \mathbb{P} :\; \mathsf{WP}.i.j.s$$

◀

Thus the condition (from now on denoted by SufficientConnections) which we are looking for must satisfy in state s

$$\frac{\mathsf{SufficientConnections}.\mathsf{ACs}.\prec_p.s \;\wedge\; \neg\mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v.s}{\exists u\, v \;:\; u \in \mathbb{P} \wedge v \in \mathbb{P} :\; \mathsf{WP}.u.v.s \wedge \mathsf{AC}.u.v.s} \tag{6.2.1}$$

We have come up with a nice formalisation of the condition which uses the transitive closure of the *connections* which are ordered by $\prec_p$. First, we define the notion of the *transitive closure* of a relation R on a set A (denoted by R^{tr}) by induction as in [RW92].

Definition 6.2.9 TRANSITIVE CLOSURE *tr_DEF, tr_n_DEF*

If $N = \mathsf{card}.A$, then $R^{tr} = R_N$, where:

$$\begin{array}{lcl} (x,y) \in R_0 & = & (x,y) \in R \\ (x,y) \in R_{n+1} & = & (x,y) \in R_n \vee (\exists z \ : z \in A \ : \ (x,z) \in R_n \wedge (z,y) \in R_n) \end{array}$$

◀

Now, we state that the following definition of $\mathsf{SufficientConnections.ACs.}\prec_p$ satisfies (6.2.1), the proof of which shall be given below.

Definition 6.2.10 SUFFICIENT CONNECTIONS IN THE NETWORK *Sufficient_Connections*

$$\mathsf{SufficientConnections.ACs.}\prec_p.s = (\prec_p \subseteq (\prec_p \cap \mathsf{ACs}.s))^{tr})$$

◀

In order to verify that Definition 6.2.10 of $\mathsf{SufficientConnections.ACs.}\ \prec_p$ satisfies (6.2.1), assume that for some state s $(\prec_p \subseteq (\prec_p \cap \mathsf{ACs}.s)^{tr})$ and $\neg\mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v.s$ hold.

From (6.2.8) and the second assumption we can deduce that there exist a $i, j \in \mathbb{P}$, such that $\mathsf{WP}.i.j.s$ (and thus $i \prec_p j$). Consequently, the first assumption tells us that $(i,j) \in (\prec_p \cap \mathsf{ACs}.s)^{tr}$, i.e. $(i,j) \in (\prec_p \cap \mathsf{ACs}.s)_N$, where $N = \mathsf{card}.\mathbb{P}$. We now prove that for all processes $i, j \in \mathbb{P}$:

$$\frac{\mathsf{WP}.i.j.s \wedge (i,j) \in (\prec_p \cap \mathsf{ACs}.s)_N}{\exists u\, v \ : u \in \mathbb{P} \wedge v \in \mathbb{P} : \ \mathsf{WP}.u.v.s \wedge \mathsf{AC}.u.v.s}$$

by induction on N.

INDUCTION BASE**:** case 1

> Assume $\mathsf{WP}.i.j.s$ and $(i,j) \in (\prec_p \cap \mathsf{ACs}.s)_1$. Rewriting the second assumption gives us $(i,j) \in (\prec_p \cap \mathsf{ACs}.s)$, i.e. $(i,j) \in \mathsf{ACs}.s$, which together with the first assumption and Definition 6.2.6 establishes this case.

INDUCTION HYPOTHESIS**:** for all $M < N$ and for all processes $i, j \in \mathbb{P}$:

$$\frac{\mathsf{WP}.i.j.s \wedge (i,j) \in (\prec_p \cap \mathsf{ACs}.s)_M}{\exists u\, v \ : u \in \mathbb{P} \wedge v \in \mathbb{P} : \ \mathsf{WP}.u.v.s \wedge \mathsf{AC}.u.v.s}$$

INDUCTION STEP**:** case N

> Assume $\mathsf{WP}.i.j.s$ and $(i,j) \in (\prec_p \cap(\mathsf{ACs}.s))_N$. Rewriting the second assumption with 6.2.9 gives us two cases:
>
> - $(i,j) \in (\prec_p \cap (\mathsf{ACs}.s))_{N-1}$, this case is trivially proven by the INDUCTION HYPOTHESIS.

- $\exists k \ : k \in \mathbb{P} \ : \ (i,k) \in (\prec_p \cap (\mathsf{ACs}.s))_{N-1} \wedge (k,j) \in (\prec_p \cap (\mathsf{ACs}.s))_{N-1}$, since $\mathsf{WP}.i.j.s$, and thus $\neg((s \circ D).i) \prec_v ((s \circ D).j)$, we can conclude, due to the transitivity of $\prec_v$, that either $\neg((s \circ D).i) \prec_v ((s \circ D).k)$ holds, or $\neg((s \circ D).k) \prec_v ((s \circ D).j)$. Again the INDUCTION HYPOTHESIS proves this case.

□

Now, we are almost ready to construct the formal specification of what the program is to do. We have defined what the program must establish, i.e. sort a particular kind of network of processes, and we have defined under which conditions it must achieve this, i.e. there is a total order on the processes data values; there is an anti-symmetric and transitive relation on the processes labels; and there is a restriction upon the failures that may occur (6.2.10). There is, however, one, important but obvious, thing that must be embodied in the specification of the sorting program. During the activity of sorting the network, we want the distribution of the values among the processes to remain a permutation of the initial distribution (i.e. the one with which the program started). If we do not require this, a simple program which just assigns the same value to all processes would achieve, by reflexivity of $\prec_v$, a sorted network (according to Definition 6.2.4). Obviously, this is not what we want. Consequently, we need a definition of permutation:

Definition 6.2.11 PERMUTATION *PERM_DEF*

$$\mathsf{Permutation}.\mathbb{P}.D.I.s \quad = \quad \exists f :: \ (\mathsf{bijection}.f.\mathbb{P}.\mathbb{P}) \wedge (\forall i \in \mathbb{P} : ((s \circ D).i) = (I.(f.i)))$$

◀

The formal specification[4] of the program in terms of the convergence operator can be found in Figure 6.3. Note that I is not a program variable, but a *proof variable*, i.e. introduced to reason about the program (namely the permutation-part). Consequently, $(D = I)$ is not an initial *program* condition, but an initial *specification* condition, which, intuitively, makes the specification state that:

IF *the program is started in some state s and I is a snapshot of the distribution of data values in that state s, and* $\mathsf{SufficientConnections}.\mathsf{ACs}.\prec_p$, $\mathsf{Permutation}.\mathbb{P}.D.I$, *and* $\mathsf{Total}.\prec_v.D.\mathbb{P}$ *are stable in the program,*

THEN *the program will eventually find itself in a situation (i.e. a state) in which the network is sorted.*

6.2.2 Results of analysing

To corroborate the conjectures made in Section 6.1 concerning the importance of analysing the problem, this subsection shall enlarge on which results, besides a formal

[4]Be aware of the overloading. $(D = I)$ means $(\lambda s.\forall i \in \mathbb{P} : (s \circ D).i = I.i)$, $\wedge$ works on state-predicates, and $\mathsf{Total}.\prec_v.D.\mathbb{P}$ is a state-predicate that given some state s, indicates whether $\prec_v$ is a total order on $\{(s \circ D)i \mid i \in \mathbb{P}\}$.

Specification 6.2.12

$$\frac{\begin{array}{c}\mathsf{dNetwork}.\mathbb{P}.\mathsf{neighs} \wedge \mathsf{distinct\_Sort\_Vars}.\mathbb{P}.D \wedge (\prec_p \neq \emptyset) \\ \mathsf{AntiSymmetric}.\prec_p.\mathbb{P} \wedge \mathsf{Transitive}.\prec_p.\mathbb{P}\end{array}}{\begin{array}{l}(\ {}_{\mathsf{Sort}}\vdash \Box\mathsf{Total}.\prec_v.D.\mathbb{P}) \wedge \\ (\ {}_{\mathsf{Sort}}\vdash \Box\mathsf{SufficientConnections}.\mathsf{ACs}.\prec_p) \wedge \\ \mathsf{Permutation}.\mathbb{P}.D.I \wedge \mathsf{SufficientConnections}.\mathsf{ACs}.\prec_p \wedge \mathsf{Total}.\prec_v.D.\mathbb{P} \\ \qquad {}_{\mathbf{Sort}}\vdash (D = I) \leadsto \mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v\end{array}}$$

Figure 6.3: Formal specification.

specification, have been achieved in the previous subsection.

First, thorough analysis of the unstable environment has enabled us to crystallise two different techniques we have used to deal with our enemies.

External agents that tamper with data values are made latent by using the convergence operator and the specification condition $(D = I)$. For, if, during the execution of the program, some agent tampers with data values (in such a way that $\prec_v$ is still a total order on the data values), we interpret the resulting state s as a new initial state (and thus I as a snapshot of the distribution of data values in that state s). Again the convergence operator guarantees that the network will eventually get sorted and stay sorted.

To handle the daemons that disable or enable communication links we made the status of communications links dependent of the state of the program, specified that the program can only converge to the required situation if SufficientConnections is stable in the program, and assumed that SufficientConnections holds in the initial state (i.e. it is an invariant). For, if, during the execution of the program, some daemon deactivates a communication link in such a way that SufficientConnections still holds the specification tells us that the network will eventually get sorted and remain sorted.

Another nice result of the analysis in the previous section, is the compact and expressive formalisation of the condition SufficientConnections.

6.2.3 Refine the specification

This subsection describes the refinement and decomposition of the convergence-part of specification 6.2.12. Since HOL verification is done at later stages, the intermediate predicates, which result from refinement or decomposition by applying UNITY rules, have to be written down with accuracy. The validity of $\mathsf{dNetwork}.\mathbb{P}.\mathsf{neighs}$, $\mathsf{distinct\_Sort\_Vars}.\mathbb{P}.D$, $(\prec_p \neq \emptyset)$, $\mathsf{AntiSymmetric}.\prec_p.\mathbb{P}$, and $\mathsf{Transitive}.\prec_p.\mathbb{P}$ shall be implicitly assumed from now on. Consequently, the specification which will be refined is:

$$\mathbf{S_0}\ :\ \mathsf{Permutation}.\mathbb{P}.D.I \wedge \mathsf{SufficientConnections}.\mathsf{ACs}.\prec_p \wedge \mathsf{Total}.\prec_v.D.\mathbb{P} \quad {}_{\mathbf{Sort}}\vdash\ (D = I) \leadsto \mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v$$

The solution strategy[5], i.e. the strategy we want our program to employ in order to satisfy $\mathbf{S_0}$, is one that reduces the number of wrong pairs of processes.

Definition 6.2.13 SET OF WRONG PAIRS OF PROCESSES *WPs_DEF*

$$\mathsf{WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = \{(i,j) | i,j \in \mathbb{P} \wedge (i \prec_p j) \wedge \neg(((s \circ D).i) \prec_v ((s \circ D).j))\}$$

Definition 6.2.14 NUMBER OF WRONG PAIRS OF PROCESSES *nr_WPs_DEF*

$$\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = \mathsf{card}.(\mathsf{WPs}.\mathbb{P}.D.\prec_p.\prec_v.s)$$

◀

In other words, during the execution of a program – which uses this strategy to sort a network – progress is ensured since the number of wrong pairs of processes reduces. In a sorted network there are no wrong pairs of processes:

Theorem 6.2.15 *Sorted_EQ_nr_WPs_0*

$$\mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v = (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = 0)$$

◀

Consequently, since $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s$ is always a value from $\mathbb{N}_0$, the less-than ($<$) is known to be a well-founded relation on $\mathbb{N}_0$, and since the value of $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s$ reduces during the execution of a program that exploits our solution strategy, the network shall eventually get sorted.

We shall now refine $\mathbf{S_0}$ according to this proposed solution strategy. For the sake of readability:

- all predicates that have to be confined by the write variables of the program Sort, in order for some laws to be applicable, are gathered into a set called Conf which shall be expanded at the end of this section.
- The stability requirements (i.e. $\mathsf{Permutation}.\mathbb{P}.D.I$, $\mathsf{SufficientConnections.ACs}.\prec_p$ and $\mathsf{Total}.\prec_v.D.\mathbb{P}$) in $\mathbf{S_0}$ are omitted from the specifications. So $\mathbf{S_0}$ becomes:

  $$\mathbf{S_0} \ : \ (D = I) \rightsquigarrow \mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v$$

Before we continue it must be pointed out that when mechanical verification is attempted, one must be prepared to deal with every detail explicitly.

Let us start by rewriting specification $\mathbf{S_0}$ into a more suitable form. Since everything implies true, we can use Theorem 6.2.15 and $\rightsquigarrow$ SUBSTITUTION (Theorem $4.6.3_{50}$) to derive:

$$\mathbf{S_0} \ : \ (D = I) \rightsquigarrow \mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v$$

$\Leftarrow$ ($\rightsquigarrow$ SUBSTITUTION $4.6.3_{50}$)

[5]There are other possible solution strategies, see for example [CM89].

$$\mathbf{S_1} \ : \ \mathsf{true} \rightsquigarrow (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)$$

where, behind the scenes, the set Conf becomes: $\{(D = I), \mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v\}$

Now $\mathbf{S_1}$ shall be refined according to the solution strategy informally described above. Recall the Bounded Progress principle for $\rightsquigarrow$ (Theorem $4.5.20_{49}$), evidently our solution strategy is an instance of this principle. Let us apply this principle to $\mathbf{S_1}$:

$$\mathbf{S_1} \ : \ \mathsf{true} \rightsquigarrow (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)$$

$\Leftarrow$ ($\rightsquigarrow$ Bounded Progress $4.5.20_{49}$, $<$ well-founded on $\mathbb{N}_0$, $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v \in \mathbb{N}_0$)

$$\begin{array}{l}\mathbf{S_2} \ : \ (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0) \rightsquigarrow (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0) \\ \wedge \\ \mathbf{S_3} : \forall m \in \mathbb{N}_0 : \\ \quad (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m) \rightsquigarrow (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v < m \vee \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)\end{array}$$

$\mathbf{S_2}$ can be decomposed into smaller specifications, using Theorems $\rightsquigarrow$ Reflexivity and $\circlearrowleft$ Conjunction:

$$\mathbf{S_2} \ : \ (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0) \rightsquigarrow (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)$$

$\Leftarrow$ ($\rightsquigarrow$ Reflexivity $4.6.5_{50}$)

$$\begin{array}{l}\mathbf{S_{2a}} \ : \circlearrowleft(\mathsf{Permutation}.\mathbb{P}.D.I \ \wedge \ \mathsf{SufficientConnections.ACs}.\prec_p \ \wedge \mathsf{Total}.\prec_v.D.\mathbb{P}) \\ \wedge \\ \mathbf{S_{2b}} \ : \ (\circlearrowleft \mathsf{Permutation}.\mathbb{P}.D.I) \ \wedge \ \mathsf{SufficientConnections.ACs}.\prec_p \ \wedge \mathsf{Total}.\prec_v.D.\mathbb{P} \\ \qquad \wedge \ \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)\end{array}$$

$\Leftarrow$ ($\circlearrowleft$ Conjunction $4.4.4_{44}$)

$$\begin{array}{l}\mathbf{S_{2c}} \ : \ (\circlearrowleft \mathsf{Permutation}.\mathbb{P}.D.I) \wedge (\circlearrowleft \mathsf{SufficientConnections.ACs}.\prec_p) \\ \qquad \wedge (\circlearrowleft \mathsf{Total}.\prec_v.D.\mathbb{P}) \wedge (\circlearrowleft \ \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)\end{array}$$

Now, observe that for the case that $m = 0$, $\mathbf{S_3}$ boils down to $\mathbf{S_2}$. Since specifications $\mathbf{S_{2a}}$ and $\mathbf{S_{2b}}$ already cover this, we can assume $m > 0$ in decomposing $\mathbf{S_3}$. Consequently, $(\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m)$ implies $(\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v > 0)$, which is the same as $\neg\mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v$. The following lemma can easily be proved for all $m > 0$ using 6.2.5 and 6.2.1:

Lemma 6.2.16 *SuffCons_AND_not_Sorted_IMP_WP_AND_AC*

$$\frac{\mathsf{SufficientConnections.ACs}.\prec_p \wedge (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m) \wedge m > 0}{\exists i, j \in \mathbb{P} : (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m) \wedge \mathsf{WP}.i.j \ \wedge \ \mathsf{AC}.i.j}$$

◀

Now, we can rewrite $\mathbf{S_3}$ as follows:

$$\mathbf{S_3}: \forall m \in N_0 : (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m) \rightsquigarrow (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v < m \vee \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)$$

$\Leftarrow$ ($\rightsquigarrow$ Substitution $4.6.3_{50}$, lemma 6.2.16, and $m > 0$)

$$\mathbf{S_4}: \forall m : m > 0 : \exists i, j \in \mathbb{P} : (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m) \wedge \mathsf{WP}.i.j \wedge \mathsf{AC}.i.j \rightsquigarrow \exists i, j \in \mathbb{P} : \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v < m$$

$\Leftarrow$ ($\rightsquigarrow$ Disjunction $4.5.18_{49}$, and $\prec_p \neq \emptyset$)

$$\mathbf{S_5}: \forall m : m > 0 : \forall i, j \in \mathbb{P} : (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m) \wedge \mathsf{WP}.i.j \wedge \mathsf{AC}.i.j \rightsquigarrow (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v < m)$$

$\Leftarrow$ ($\rightsquigarrow$ Introduction $4.6.4_{50}$, $\circlearrowleft$ Conjunction $4.4.4_{44}$, and assumed validity of $\mathbf{S_{2c}}$)

$$\mathbf{S_{6a}}: \forall m : m > 0 : \circlearrowleft \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v < m$$
$$\wedge$$
$$\mathbf{S_{6b}}: \forall m : m > 0 : \forall i, j \in \mathbb{P} : \mathsf{Permutation}.\mathbb{P}.D.I \wedge \mathsf{SufficientConnections}.\mathsf{ACs}.\prec_p \wedge \mathsf{Total}.\prec_v.D.\mathbb{P} \wedge (\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m) \wedge \mathsf{WP}.i.j \wedge \mathsf{AC}.i.j \text{ ensures } \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v < m$$

The refinement is now completed since the progress parts are solely expressed in terms of ensures . So we have decomposed and refined specification $\mathbf{S_0}$ into:

$$\mathbf{S_0} \Leftarrow \mathbf{S_{2c}} \wedge \mathbf{S_{6a}} \wedge \mathbf{S_{6b}} \wedge \forall p : p \in \mathsf{Conf} : p \, \mathcal{C} \, \mathbf{wSort}$$

where $\mathsf{Conf} = \{(D = I), \mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v, \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v, \mathsf{WP}.i.j, \mathsf{AC}.i.j\}$.

6.2.4 Construct a program that satisfies this refined specification

Considering the properties of the network – principally the property that a process can only communicate with one other process at the same time – it is evident that the only thing two connected processes can do is compare their values and swap them if they are out-of-order with respect to the processes labels. The resulting program is presented in Figure 6.4; it performs a topological sort on the directed acyclic graph $G_{\prec_p} = (P, \{(u, v) \mid u \prec_p v\}$. Note that the variables $\{aC.i.j \mid i \in \mathbb{P} \wedge j \in \mathsf{neighs}.i\}$, although not actually written by the program Sort, are added to the write variables of the program Sort. The reason for this is that we assume an unstable environment in which there are daemons present that can write these variables. Although these daemons

prog Sort
read $\{D\,i \mid i \in \mathbb{P}\} \cup \{aC.i.j \mid i \in \mathbb{P} \wedge j \in \mathsf{neighs}.i\}$
write $\{D\,i \mid i \in \mathbb{P}\} \cup \{aC.i.j \mid i \in \mathbb{P} \wedge j \in \mathsf{neighs}.i\}$
init $\mathsf{SufficientConnections.ACs}.\prec_p \wedge \mathsf{Total}.\prec_v.D.\mathbb{P}$
assign

$$[]i,j : (i,j \in \mathbb{P}) : \quad \textbf{if } \mathsf{WP}.i.j \wedge \mathsf{AC}.i.j \quad \textbf{then}(D.i),(D.j) := (D.j),(D.i) \qquad (\mathsf{swap}.(i,j))$$

Figure 6.4: The sorting program

are not explicitly specified, their presence is implicitly implied by the occurrence of the variables $\{aC.i.j \mid i \in \mathbb{P} \wedge j \in \mathsf{neighs}.i\}$ in the write variables. Since the fact that these variables are assumed to be writable implies that they can somehow change.

6.2.5 Prove that the program satisfies the specification

In order to verify that UNITY program Sort satisfies specification 6.2.12, we have to show that, if $\mathsf{dNetwork}.\mathbb{P}.\mathsf{neighs}$, $\mathsf{distinct\_Sort\_Vars}.\mathbb{P}.D$, $(\prec_p \neq \emptyset)$, $\mathsf{AntiSymmetric}.\prec_p.\mathbb{P}$, and $\mathsf{Transitive}.\prec_p.\mathbb{P}$, then the program satisfies specification $\mathbf{S_0}$, and has invariants $(\vdash \Box\mathsf{SufficientConnections.ACs}.\prec_p)$ and $(\vdash \Box\mathsf{Total}.\prec_v.D.\mathbb{P})$ Before we do this, let us first look more closely at the program and some of its properties.

All actions which can be non-deterministically selected during the execution of the program, do the same: the data values of two connected processes are compared and swapped if and only if these values are out-of-order with respect to the processes' labels. Swapping the data values of two connected processes i and j, means that process i stores the data value of process j in its local variable and vice versa. Furthermore, only two processes at the same time are considered so that the data values of all other processes are unchanged. The following definition states the predicate which holds after executing an action that swapped two data values.

Definition 6.2.17 *Swapped_DEF*

$$\begin{aligned}\mathsf{Swapped}.\mathbb{P}.D.s.t \;=\; & \exists i\,j \;:i,j \in P : \\ & \quad \mathsf{WP}.i.j.s \\ & \wedge (\forall k \;: k \in P \wedge k \neq i \wedge k \neq j :\; (s \circ D).k = (t \circ D).k) \\ & \wedge (s \circ D).i = (t \circ D).j \wedge (s \circ D).j = (t \circ D).i\end{aligned}$$

Since every action either swaps data values or does nothing otherwise, the following can be easily inferred:

Theorem 6.2.18 *Swap_OR_SKIP*

For all actions $a \in \mathsf{aSort}$ and all states s and t:

$$\frac{a.s.t}{\mathsf{Swapped}.\mathbb{P}.D.s.t \ \vee \ s = t}$$

◀

Now let us turn to our specification and show that our program satisfies $\mathbf{S_{2c}}$, $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$. It can easily be verified that program Sort satisfies $\mathbf{S_{2c}}$.

- if an action of the program results in changing a process' data value then this value is exchanged (i.e. substituted) for a data value of another process. Consequently, the distribution of the data values among the processes remains a permutation of the initial distribution, i.e. $\mathsf{Permutation}.\mathbb{P}.D.I$ is stable.
- since for all $i, j \in \mathbb{P}$, the action $\mathsf{swap}.(i, j)$ does not alter the variables $aC.i.j$, $\mathsf{SufficientConnections.ACs}.\prec_p$ is stable in the program Sort
- since only out-of-order-pairs are swapped no swapping whatsoever will be done if the network is sorted, so $(\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = 0)$ is stable.

□ $(\mathbf{S_{2c}})$

Less trivial, however, is to recognise that the stability predicate $\mathbf{S_{6a}}$ and the progress property $\mathbf{S_{6b}}$ are satisfied. In order to show that they hold, it turns out to be sufficient to prove that our program employs the solution strategy that was introduced in section 6.2.3, that is if two processes swap their data values, then the total number of wrong pairs of processes decreases. In other words, we must prove that *if* the program finds itself in some state s in which it holds that there still exist wrong pairs of processes, *then if* t is the state in which the program results after swapping the data values of some connected wrong pair (which exists because of theorem 6.2.1), *then* the number of wrong pairs in state t is less than the number of wrong pairs in state s. More formally, in order to show that $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$ are satisfied by the program, it suffices to show the following theorem:

Theorem 6.2.19 *Decreasing_WPs*

For all $m \in N_0$ and states s and t:

$$\frac{\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = m \wedge \mathsf{Swapped}.\mathbb{P}.D.s.t}{\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.t < m}$$

◀

Let us first assume that 6.2.19 holds and prove that the program satisfies $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$.

To show that $\mathbf{S_{6a}}$ is satisfied it must be demonstrated that for all $m > 0$, for all actions $a \in \mathsf{aSort}$ and for all states s and t it holds that:

$$\frac{\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s < m \wedge a.s.t}{\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.t < m}$$

Assume $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s < m$ and $a.s.t$, then, following 6.2.18, there are two possible cases that can be distinguished:

- $s = t$, this immediately establishes the proof.
- $\mathsf{Swapped}.\mathbb{P}.D.s.t$ holds. From the assumption that $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s < m$ we can deduce that there exists a $k < m$ such that $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = k$. From 6.2.19 we then know that $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.t < k$, which establishes the proof since $k < m$.

□ ($\mathbf{S_{6a}}$)

Now let us turn to $\mathbf{S_{6b}}$, rewriting it with the definition of ensures (Definition $4.4.2_{43}$) gives us the following two proof obligations for all $m > 0$ and $i, j \in \mathbb{P}$:

$$
\begin{array}{l}
\quad \mathsf{Permutation}.\mathbb{P}.D.I \ \wedge \ \mathsf{SufficientConnections.ACs}.\prec_p \ \wedge \mathsf{Total}.\prec_v.D.\mathbb{P} \\
\quad \wedge \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v = m \ \wedge \ \mathsf{WP}.i.j \ \wedge \ \mathsf{AC}.i.j \\
\qquad \mathsf{unless} \\
\quad \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v < m \\
\wedge \\
\quad \exists a \ : a \in \mathbf{a}\mathsf{Sort} : \\
\qquad a.s.t \\
\qquad \wedge \mathsf{Permutation}.\mathbb{P}.D.I.s \ \wedge \ \mathsf{SufficientConnections.ACs}.\prec_p.s \ \wedge \ \mathsf{Total}.\prec_v.D.\mathbb{P} \\
\qquad \wedge ((\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = m) \wedge \ \mathsf{WP}.i.j.s \ \wedge \ \mathsf{AC}.i.j.s \\
\qquad \wedge \neg(\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s < m) \\
\qquad \Rightarrow \\
\qquad \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.t < m
\end{array}
$$

The first conjunct (i.e. unless -part) is easy to prove, for if, after executing action a:

- $s = t$ holds, then unless is trivially established
- $\mathsf{Swapped}.\mathbb{P}.D.s.t$ holds, then 6.2.19 confirms that $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.t < m$.

In order to prove the exists-part, we must find an action which reduces the number of wrong pairs of processes, given that there exists a connected wrong pair of processes (i, j). Obviously, using Theorem 6.2.19, this is the action $\mathsf{swap}.(i, j)$.
□ ($\mathbf{S_{6b}}$)

Recapitulating, we have shown that if 6.2.19 holds, then our program Sort satisfies the sub-specifications $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$. So to finish the verification of our program's satisfiability to $\mathbf{S_0}$, it suffices to show that all elements of Conf are elements of $\mathtt{Pred}.(\mathbf{w}\mathsf{Sort})$, and that theorem 6.2.19 holds. The latter is repeated below for convenience:
for all $m \in N_0$ and states s and t:

$$
\frac{\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = m \wedge \mathsf{Swapped}.\mathbb{P}.D.s.t}{\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.t < m} \tag{6.2.19}
$$

Assume $\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = m$ and $\mathsf{Swapped}.\mathbb{P}.D.s.t$. From these assumptions we can infer that there exists a pair of processes of which the data values are swapped during the state-transition from s to t, let us call this pair (i, j). Evidently it holds that $i \prec_p j \ \wedge \neg((s \circ V).i \prec_v (s \circ V).j)$. Now, for an arbitrary state s, we look at $\mathsf{WPs}.\mathbb{P}.D.\prec_p.\prec_v.s$ and split this set up in seven disjoint sets using i and j.

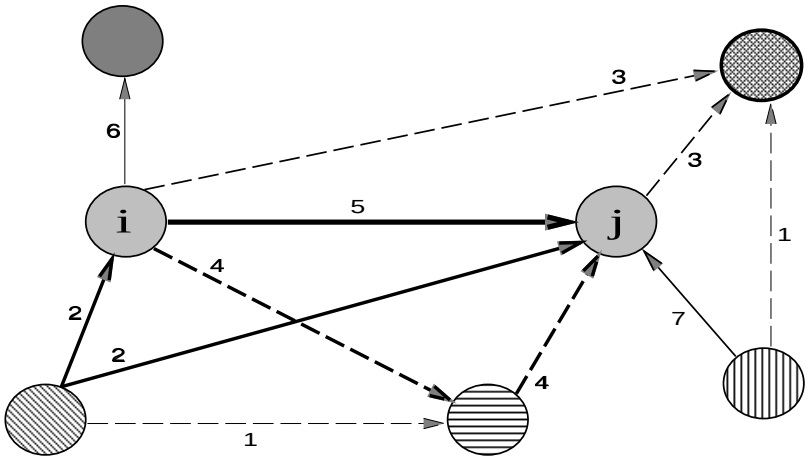


Figure 6.5: The possible edges in the set $\mathsf{WPs}.\mathbb{P}.D.\prec_p.\prec_v.s$, when $(i,j) \in \mathsf{WPs}.\mathbb{P}.D.\prec_p .\prec_v.s$.

For any state s:

$$\mathsf{WPs}.\mathbb{P}.D.\prec_p.\prec_v.s = \bigcup_{n=1}^{7} \mathsf{WP}^n.s \tag{6.2.2}$$

where

$$\mathsf{WP}^1.s = \{(x,y) | x,y \in P \wedge x \prec_p y \wedge \neg((s \circ V).x \prec_v (s \circ V).y) \wedge x \neq i \wedge x \neq j \wedge y \neq i \wedge y \neq j\}$$

$$\mathsf{WP}^2.s = \{(x,y) | x \in P \wedge x \neq i \wedge x \neq j \wedge x \prec_p i \wedge (y = i \vee y = j) \wedge \neg((s \circ V).x \prec_v (s \circ V).y)\}$$

$$\mathsf{WP}^3.s = \{(x,y) | y \in P \wedge y \neq i \wedge y \neq j \wedge j \prec_p y \wedge (x = i \vee x = j) \wedge \neg((s \circ V).x \prec_v (s \circ V).y)\}$$

$$\mathsf{WP}^4.s = \{(x,y) | ((x = i \wedge y \neq j \wedge i \prec_p y \wedge y \prec_p j) \vee (y = j \wedge x \neq i \wedge i \prec_p x \wedge x \prec_p j)) \wedge \neg((s \circ V).x \prec_v (s \circ V).y)\}$$

$$\mathsf{WP}^5.s = \{(i,j)\}$$

$$\mathsf{WP}^6.s = \{(i,y) | y \neq j \wedge i \prec_p y \wedge \neg(j \prec_p y) \wedge \neg(y \prec_p j) \wedge \neg((s \circ V).i \prec_v (s \circ V).y)\}$$

$$\mathsf{WP}^7.s = \{(x,j) | x \neq i \wedge x \prec_p j \wedge \neg(x \prec_p i) \wedge \neg(i \prec_p x) \neg((s \circ V).x \prec_v (s \circ V).j)\}$$

and:

$$\forall k,l : k,l = 1,2,\ldots,7 \wedge k \neq l : \mathsf{WP}^k.s \cap \mathsf{WP}^l.s = \emptyset \tag{6.2.3}$$

From Figure 6.5, which shows the directed graph $G_{\prec_p}$, one can deduce how the edges in the set $\mathsf{WPs}.\mathbb{P}.D.\prec_p.\prec_v.s$ are divided among the seven sets above. The numbers

n associated with the edges correspond with the set $\mathsf{WP}^n.s$ in which they reside. Edges in set $\mathsf{WP}^1.s$, are edges between two nodes which are neither i nor j. Edges in $\mathsf{WP}^2.s$ are the incoming edges of i, and those incoming edges of j which result from the transitivity of $\prec_p$ on the incoming edges of i and the edge (i, j). Set $\mathsf{WP}^3.s$ consists of the outgoing edges of j, and those outgoing edges of i which result from $\prec_p$'s transitivity on the outgoing edges of j and the edge (i, j). $\mathsf{WP}^4.s$ contains the incoming edges of a node k, which is neither i nor j, and for which it holds that $i \prec_p k \prec_p j$. Set $\mathsf{WP}^5.s$ is the singleton set with edge (i, j). $\mathsf{WP}^6.s$ comprises the edges (i, k), where k is not $\prec_p$-related to j (i.e. neither $k \prec_p j$ nor $j \prec_p k$ hold). Finally, $\mathsf{WP}^7.s$ is the set of the edges (k, j), where k is not $\prec_p$-related to i.

From 6.2.2 and 6.2.3, the validation of which are left as simple exercises to the reader, it follows that:

$$\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s \quad = \quad \sum_{n=1}^{7} \mathsf{card}.(\mathsf{WP}^n.s) \tag{6.2.4}$$

The proof of 6.2.19 now proceeds by comparing the cardinality of the different WP's in transition states s and t.

$\mathsf{card}.(\mathsf{WP}^1.s) = \mathsf{card}.(\mathsf{WP}^1.t)$**,** because the assumption $\mathsf{Swapped}.\mathbb{P}.D.s.t$ indicates that the data values of processes other than i and j do not change.

$\mathsf{card}.(\mathsf{WP}^2.s) = \mathsf{card}.(\mathsf{WP}^2.t)$ In order to prove this equality, it suffices to show that there exists a bijection f from $\mathsf{WP}^2.s$ to $\mathsf{WP}^2.t$. Below a function f is given which satisfies this constraint.

$$f = \lambda(x, y).\textbf{if } y = i \textbf{ then } (x, j) \textbf{ else } (x, i)$$

The proof that f is a bijection is left as an exercise to the reader.

$\mathsf{card}.(\mathsf{WP}^3.s) = \mathsf{card}.(\mathsf{WP}^3.t)$ A similar proof as the one for $\mathsf{card}.(\mathsf{WP}^2.s) = \mathsf{card}.(\mathsf{WP}^2.t)$ applies. A satisfactory bijection is:

$$f = \lambda(x, y).\textbf{if } x = i \textbf{ then } (j, y) \textbf{ else } (i, y)$$

$\mathsf{card}.(\mathsf{WP}^4.s) \leq \mathsf{card}.(\mathsf{WP}^4.t)$ To prove this, it is sufficient to verify that $\mathsf{WP}^4.t \subseteq \mathsf{WP}^4.s$. Suppose we have a pair $(x, y) \in \mathsf{WP}^4.t$, now we must show that $(x, y) \in \mathsf{WP}^4.s$. From the definition of $\mathsf{WP}^4.t$ we learn that there are two possibilities:

- $x = i$, $y \neq j$ and $\neg((t \circ V).i \prec_v (t \circ V).y)$.

  Assuming these conditions, we must show that $(i, y) \in \mathsf{WP}^4.s$. Because we already have that $x = i$, $y \neq j$, $i \prec_p y$ and $y \prec_p j$, we only have to prove that $\neg((s \circ V).i \prec_v (s \circ V).y)$, which is equal to $(s \circ V).y \prec_v (s \circ V).i$ since $\prec_v$ is a total order. From the assumption that $\neg((t \circ V).i \prec_v (t \circ V).y)$ it can be deduced that:

1. $y \neq i$, because of the reflexivity of $\prec_v$ and the fact that $\neg((t \circ V).i \prec_v (t \circ V).y)$ holds.
2. $(t \circ V).y \prec_v (t \circ V).i$, for $\prec_v$ is a total order.

From the assumption $\mathsf{Swapped}.\mathbb{P}.D.s.t$ and its presumed instantiation with (i, j), the following can be inferred:

3. $(t \circ V).i \prec_v (t \circ V).j$
4. $(t \circ V).j = (s \circ V).i$
5. $(s \circ V).y = (t \circ V).y$, because of (1) and $y \neq j$.

Now the transitivity and totality of $\prec_v$ establishes that $(s \circ V).y \prec_v (s \circ V).i$, since:

$$(s \circ V).y \overset{(5)}{=} (t \circ V).y) \overset{(2)}{\prec_v} (t \circ V).i \overset{(3)}{\prec_v} (t \circ V).j \overset{(4)}{=} (s \circ V).i$$

- $y = j$, $x \neq i$ and $\neg((t \circ V).x \prec_v (t \circ V).j)$.

  In this case we must show that $(x, j) \in \mathsf{WP}^4.s$, which again comes down to showing that $(s \circ V).j \prec_v (s \circ V).x$ holds. Analogous to the previous proof this can be done by showing:

$$(s \circ V).j \;=\; (t \circ V).i \;\prec_v\; (t \circ V).j \;\prec_v\; (t \circ V).x \;=\; (s \circ V).x$$

$\mathsf{card}.(\mathsf{WP}^5.s) < \mathsf{card}.(\mathsf{WP}^5.t)$**,** because $\mathsf{card}.(\mathsf{WP}^5.s)$ and $\mathsf{card}.(\mathsf{WP}^5.t)$ equal 1 and 0 respectively.

$\mathsf{card}.(\mathsf{WP}^6.s) \leq \mathsf{card}.(\mathsf{WP}^6.t)$ **and** $\mathsf{card}.(\mathsf{WP}^7.s) \leq \mathsf{card}.(\mathsf{WP}^7.t)$**,** proofs are analogous to that of $\mathsf{card}.(\mathsf{WP}^4.s) \leq \mathsf{card}.(\mathsf{WP}^4.t)$.

Since the above seven items indicate that

$$\sum_{n=1}^{7} \mathsf{WP}^n.t \;<\; \sum_{n=1}^{7} \mathsf{WP}^n.s$$

this completes the proof of 6.2.19, because, according to 6.2.4, this means:

$$\mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.t < \mathsf{nr\_WPs}.\mathbb{P}.D.\prec_p.\prec_v.s.$$

To complete the proof that the program satisfies specification 6.2.12, we still have to verify that the following:

1. $\forall p \;: p \in \mathsf{Conf} :\; p\, \mathcal{C}\, \mathbf{wSort}$
2. $(\;_{\mathsf{Sort}}\!\vdash \Box\mathsf{Total}.\prec_v.D.\mathbb{P})$
3. $(\;_{\mathsf{Sort}}\!\vdash \Box\mathsf{SufficientConnections}.\mathsf{ACs}.\prec_p$

Verification of the first condition is left as an exercise to the reader, the other conditions can easily be proved by applying definition $4.4.8_{44}$ and $\mathbf{S_{2c}}$.

6.2.6 Mechanical verification activities

We shall start by representing the program in the HOL embedding of UNITY. Since fancy notation like $\mathbb{P}$, $\prec_p$, and $\prec_v$ are not possible within HOL, these concepts shall in this section be denoted by `P`, `pOrd`, and `vOrd` respectively.

First, we shall look at the program variables, and which values can be assigned to them. The values that can be assigned to the $aC.i.j$ variables are booleans; the variables that can be assigned to the $D.i$ variables can be of any type on which a total order $\prec_v$ is defined. Obviously, since these different variables can take values of different types, we use the theory described in Chapter 3. Consequently, we presume that for all $i, j \in \mathbb{P}$, $D.i$ and $aC.i.j$ have actual type `Val`. Moreover, we shall assume that for all $i, j \in \mathbb{P}$, $aC.i.j$ has intended type `bool`, and leave the intended type of $D.i$ unspecified.

Second, we determine the required types of the labels of the processes in `P`. For consistency with subsequent chapters, where processes can be assigned to a variable having actual type `Val`, we shall consider `Val` to be the type of the labels of the processes in `P`. Since the labels can be of any type on which a non-empty, anti-symmetric and transitive order is defined, we shall leave the intended type of the labels unspecified. For convenience we define the following type synonym:

```
val process = ty_antiq(==‘:Val‘==);
```

The type declaration specifying the intended type of the `aC` variables becomes:

HOL-definition 6.2.20

```
val type_DECL_Sort = new_definition("type_DECL_Sort",
 (--‘type_DECL_Sort (aC:^process ->^process ->’var)
     =
     !(i:^process) (j:^process) (s:^State). (is_bool (s (aC i j)))‘--));
```

◀

In order to be able to define the state-predicates `WP` and `AC` (which are used as guards in the program) as regular state-functions (see Section 5.4) we shall assume that `P`, `neighs`, `pOrd` and `vOrd` have the following types:

- `P:Val`, where, the intended type of `P` is specified by `is_set.P`, and the intended type of the elements (being processes) are left unspecified.
- `neighs: ^process->Val`, where, for all processes `p` the intended type of value `neighs.p` is specified by `is_set.(neighs.p)`
- `pOrd: (^process # ^process)->Val`, where for all processes `p` and `q`, the intended type of `pOrd(p,q)` is specified by `is_bool.(pOrd(p, q))`
- `vOrd: Val->Val->Val`, where for two values `x` and `y` stored in the local variables `D` of two processes in `P`, the intended type of `vOrd x y` is specified by `is_bool.(vOrd x y)`

Note that pOrd is not defined as a curried function like vOrd. The reason for this is that, in Section 6.2.1, pOrd is used as relation (i.e. $i \prec_p j$ in Definition 6.2.4 on page 75) as well as a set of tuples (i.e. $(\prec_p \subseteq (\prec_p \cap \mathsf{ACs}.s))^{tr}$ in definition 6.2.10 on page 78). Although we can easily mix these two on paper, in HOL this is not possible. Defining pOrd to have type (^process # ^process)->Val enables us to use pOrd as a set as well as a relation.

Now WP and AC can be defined as regular state-predicates as follows.

HOL-definition 6.2.21

```
val WP_DEF = new_definition("WP_DEF",
  (--‘WP (i:^process) (j:^process) (D:^process->’var)
         (pOrd:(^process # ^process)->Val) (vOrd: Val->Val->Val)
      =
      (CONST (pOrd (i,j)))
      |/\|
      not (BI_APPLY vOrd (VAR (D i)) (VAR (D j)))‘--));
```

HOL-definition 6.2.22

```
val AC_DEF = new_definition("AC_DEF",
  (--‘AC (i:^process) (j:^process) (neighs:^process->Val)
         (aC:^process->^process ->’var)
      =
      (CONST (i INv (neighs j))) |/\| (VAR (aC i j))‘--));
```

◀

The set of all active connections is, like the order pOrd, modelled by a function of type (^process # ^process)->Val:

HOL-definition 6.2.23

```
val ACs_DEF = new_definition("ACs_DEF",
  (--‘ACs (neighs:^process->Val) (aC:^process->^process ->’var)
          (s:^State) (i,j):(^process # ^process)
      =
      (AC i j neighs aC s)‘--));
```

◀

Now, we shall formalise the condition Sufficient_Connections. Unfortunately, this condition can not be formalised as a regular state-function. Fortunately, however, we do not have to prove that the state-predicate Sufficient_Connections is confined by the write variables of the program sort, since it does not appear in the set Conf at the end of Section 6.2.3.

First, we add the inductive definition of a transitive closure of a relation R on a set A to HOL (see $6.2.9_{78}$) as a recursive function on numerals.

HOL-definition 6.2.24

```
val tr_n_DEF = new_recursive_definition
  {name = "tr_n"
  ,fixity = Prefix,
  ,def =
   (--‘((tr_n 0) = (\(R:(Val # Val)->Val) (A:Val) (x,y). (R (x,y))))
       /\
       (!(n:num).
         ((tr_n (SUC n)) = (\(R:(Val # Val)->Val) A (x,y).
                              ((tr_n n R A (x,y)) Or
                               (Exists z :: (CHFv A).
                                 ((tr_n n R A (x,z))
                                   And
                                  (tr_n n R A (z,y))))))))‘--)
  ,rec_axiom = (theorem "prim_rec" "num_Axiom")};

val tr_DEF = new_definition ("tr_DEF",
  (--‘tr (R:(Val # Val)->Val) (A:Val) ((x:Val),(y:Val)) =
       tr_n (CARDn A) R A (x,y)‘--));
```

To bring these HOL definitions into line with Definition 6.2.9$_{78}$ from Section 6.2.1, for a relation R on a set A

- the relation R^{tr} is modelled by `tr R A`
- the relation R_n is modelled by `tr_n n R A`

The state-predicate `Sufficient_Connections` is formalised below. Note that, since we have defined the operator `|/\|` to be of general type:
`('a->Val)->('a->Val)->'a->Val`
in stead of:
`(State->Val)->(State->Val)->State->Val`
it can be used here to model the intersection of `pOrd` and `ACs`.

HOL-definition 6.2.25

```
val Sufficient_Connections = new_definition("Sufficient_Connections",
 (--‘Sufficient_Connections (P:Val) (neighs: ^process->Val)
                            (aC:^process->^process ->'var)
                            (pOrd:(^process # ^process) ->Val)
     =
     |!!| i j :: (CHFv P) .
                 (CONST_EXPR (pOrd (i,j)))
                  |==>|
                 (\(s:^State). tr (pOrd |/\| (ACs neighs aC s)) P (i,j))‘--));
```

Finally the program `Sort` can be defined in HOL, as a quadruple of type `Uprog`.

HOL-definition 6.2.26

```
val Sort = new_definition("Sort",
  (--‘Sort (P:Val) (neighs:^process->Val)
           (D:^process->’var) (aC:^process->^process ->’var)
           (pOrd:(^process # ^process) ->Val) (vOrd:Val->Val->Val)
      =
      (CHF{GUARD ((WP i j D pOrd vOrd) |/\| (AC i j neighs aC))
                 (ASSIGN [(D i) ; (D j)]
                         [(VAR (D j)); (VAR (D i))]) | i INb P /\ j INb P}
      ,
       ((Sufficient_Connections P neighs aC pOrd):^State->Val)
       |/\| (total vOrd D P)
      ,
       CHF({D i | i INb P} UNION {aC i j | i INb P /\ j INb (neighs i)})
      ,
       CHF({D i | i INb P} UNION {aC i j | i INb P /\ j INb(neighs p)}))‘--));
```

The state-predicate defining whether the values in the network are sorted with respect to orders `pOrd` and `vOrd` is defined in HOL by the following regular state-predicate:

HOL-definition 6.2.27

```
val Sorted_DEF = new_definition("Sorted_DEF",
  (--‘Sorted (P:Val) (D:^process->’var)
             (pOrd:(^process # ^process) -> Val) (vOrd :Val->Val->Val)
      =
      |!!| i j : (CHFv P) :
                      (CONST (pOrd (i,j)))
                      |==>|
                      (BI_APPLY vOrd (VAR (D i)) (VAR (D j)))‘--));
```

The term that represents specification $\mathbf{S}_0$ is:

```
(--‘!(P:Val) (neighs:^process->Val) (I:^process->Val) (D:^process->’var)
     (aC:^process->^process->’var) (pOrd:(^process # ^process)->Val)
     (vOrd:Val -> Val -> Val).

        ((dNetwork P neighs) /\ (distinct_Sort_Vars P D)
         /\ ~(empty pOrd) /\ (anti_sym pORD P) /\ (transitive pOrd P))

     ==>

     CONe (Sort P neighs D aC pOrd vOrd)
          ((Permutation P D I) |/\| (Sufficient_Connections P neighs aC pOrd)
           |/\| (total vOrd D P))
          (|!!| i :: (CHFv P). (VAR (D i)) EQ (CONST (I i)))
          (Sorted P D pOrd vOrd)‘--)
```

prog Daemon
read $\{aC\,i\,j \mid i \in \mathbb{P} \wedge j \in \mathsf{neighs}.i\}$
write $\{aC\,i\,j \mid i \in \mathbb{P} \wedge j \in \mathsf{neighs}.i\}$
init $\mathsf{SufficientConnections.ACs}.\prec_p$
assign $[]i, j : (i, j \in \mathbb{P}) \wedge j \in \mathsf{neighs}.i$:

if $\mathsf{AC}.i.j \wedge \mathsf{SufficientConnections}.(\mathsf{ACs} - \{(i,j)\}).\prec_p$ ($\mathsf{de\text{-}activate}.(i,j)$)
then $aC.i.j := \neg aC.i.j$

[]

$[]i, j : (i, j \in \mathbb{P}) \wedge j \in \mathsf{neighs}.i$:

if $\neg\, \mathsf{AC}.i.j$
then $aC.i.j := \neg aC.i.j$ ($\mathsf{activate}.(i,j)$)

Figure 6.6: The daemon

where `empty`, `anti_sym`, and `transitive` are HOL definitions having the desired semantics. In order to prove this goal to be a theorem, first, a proof tree must be constructed using the refinement and decomposition strategy from Section 6.2.3, then the proof tree must be closed using the proof in Section 6.2.5.

6.2.7 Discussion

Although the tackled problem in this case study is simple, and its primary aim is to illustrate the application of the extended UNITY methodology, there are some subtleties that have to be highlighted.

We have modelled the presence of daemons that can de-activate and re-activate the communication links implicitly by adding the variables aC (denoting whether a link is active) to the write variables of our sorting program. What we would really want, however, is to model the daemons explicitly and prove that the composition of our sorting program with the daemons establish the specification. Unfortunately, this is not possible within the version of UNITY used here. Consider the explicit model of the daemons presented in Figure 6.6. It is not hard to see that these daemons satisfy the property:

$$_{\mathsf{Daemon}}\vdash \circlearrowleft \mathsf{SufficientConnections.ACs}.\prec_p$$

However,

$$\mathsf{Permutation}.\mathbb{P}.D.I \wedge \mathsf{SufficientConnections.ACs}.\prec_p \wedge \mathsf{Total}.\prec_v.D.\mathbb{P}$$
$$_{\mathsf{Sort}\,[]\,\mathsf{Daemon}}\vdash (D = I) \rightsquigarrow \mathsf{Sorted}.\mathbb{P}.D.\prec_p.\prec_v$$

cannot be proved, since we can construct a valid UNITY execution sequence of

Sort [] Daemon, in which the Deamon continuously de-activates the link over which Sort is about to communicate and hence prevents the network from getting sorted. Suppose we have the following network, $\mathbb{P} = \{x, y, z\}$ and links.$\mathbb{P}$.neighs $= \{(x, y); (y, z); (z, x)\}$. Then the following is a fair UNITY execution:

$$
\begin{array}{ll}
[& \mathsf{de\text{-}activate}.(x, y); \mathsf{swap}.(x, y); \mathsf{activate}.(x, y); \\
& \mathsf{de\text{-}activate}.(y, z); \mathsf{swap}.(y, z); \mathsf{activate}.(y, z); \\
& \mathsf{de\text{-}activate}.(z, x); \mathsf{swap}.(z, x); \mathsf{activate}.(z, x) \ ]^*
\end{array}
$$

Evidently, all swap actions is this sequence have no effect (i.e. skip) since when they are executed their guards are disabled. Consequently, the network shall not get sorted eventually.

The reason for this is that the notion of fairness in UNITY programs, which is implied by the UNITY fairness rule that constrains nondeterministic action selection during program execution, is weak fairness. That is [Fra86], an action will not be indefinitely postponed provided that its guard is *continuously enabled*. Consequently, since the guard of the action swap is not continuously enabled, the actual event of swapping to values can be postponed indefinitely. What we need to establish a sorted network is a notion of strong fairness, i.e. fairness that guarantees eventual occurrence of an action under the condition that its guard is *infinitely-often enabled*, but not necessarily continuously.

6.3 Reflections

Sketched in broad outlines, our experience with formal methods and theorem provers used during the development process of (distributed) programs is twofold:

- they increase one's confidence in the correctness of the proof
- using them takes time.

We found that, on paper, errors can easily creep into a proof, even when the proof is done with extreme care. In the main, this is caused by the implicit assumption of, at first sight trivial, lemmas which turn out to be invalid. With a prover this is out of the question, since every lemma has to be proven explicitly. Consequently, the correctness of all lemmas used in the proof is guaranteed. Another matter in which more confidence is obtained is the completeness of assumptions. The complete set of assumptions is determined *during* verification. One starts with an empty set of assumptions, and subsequently every emerging proof obligation that cannot be proved from earlier assumptions is added to the set of assumptions. Since the prover makes sure that every proof obligation is made explicit, we shall not forget any assumption. In addition, we are not likely to introduce superfluous assumptions, since the possibility to interactively construct proofs (which is available in many provers) considerably helps in determining whether something is provable or not.

In the light of the second experience, the development of the sorting program took approximately 6 man months, and included getting acquainted with HOL [GM93] and Prasetya's UNITY embedding [Pra95]. This is a lot of time, considering the complexity of the problem solved, and the size of the program which was proven. Three

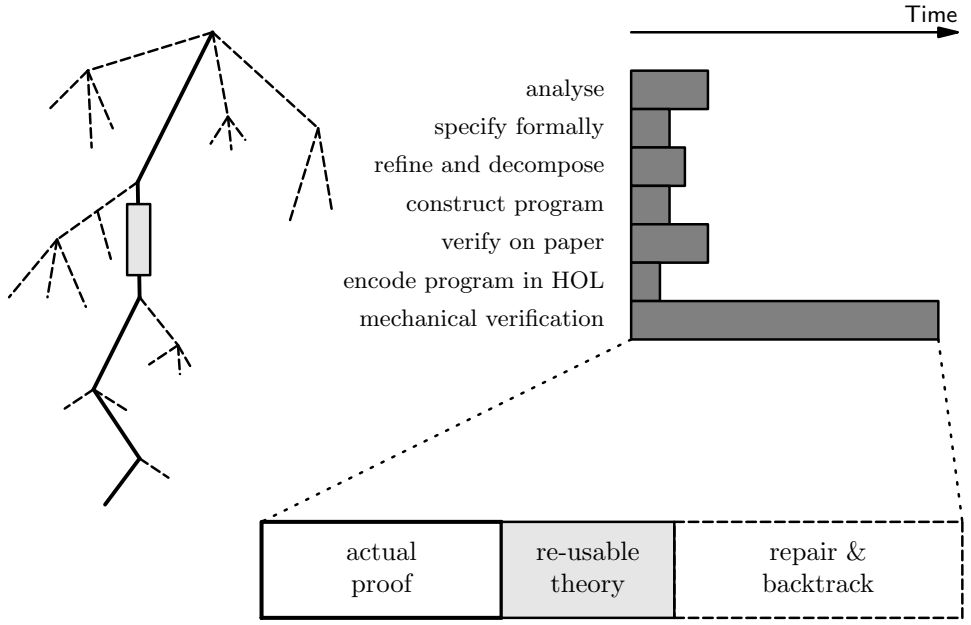


Figure 6.7: Proportionate estimates of time and effort.

important questions now arise.

Are these large amounts of time justifiable?

For complex safety-critical processes, where a run-time error or failure could result in death, injury, loss of property or environmental harm, the answer to this question is a definite yes [Lev86, BJ87, PvSK90, Lap90, Lev91, BS92, BS93b, BM92, RvH93, Bow93, BJ93, Rus94, GCR94, Lev95, Neu95]. Deaths due to poorly designed software have occurred; for example the accidents that happened with the Therac-25, a computerised radiation therapy machine. Between June 1985 and January 1987, six known accidents involved massive overdoses by this machine – with resultant deaths and serious injuries. For a thorough investigation of these accidents with the Therac-25 the reader is referred to [LT93, Tho94a, Tho94b].

Are these large amounts of time acceptable in industrial applications?

Obviously, considering the competition and the "time to market", the answer to the second question is no [Hal90, CG92, BS93a, CGR93, WW93, BH94, BH95a, CS95]. Consequently, one important goal of future research in the area of formal methods and mechanical verification is the reduction of the time which is necessary to formally and mechanically develop and verify safety-critical software. This immediately brings us to the last and most important question, namely:

Which part of the development is the most time-consuming, and what can we do about this?

For this last question we refer to Figure 6.7, in which the histogram shows the estimate of the proportionate time spent on the steps from section 6.1, and the tree represents the mechanical verification activities. Looking back, it can be said, with a vast degree of certainty, that most of the time has been spent on the machine-checked proofs. All in all, there are three activities that contribute to the amount of time used to mechanically check the correctness of the program.

1. the time spent on insufficient proof-strategies and the time needed to backtrack after these inadequacies are encountered, plus the time devoted to discover and repair mistakes and/or faulty definitions (the dotted lines)
2. construction of the actual proof (the solid line)
3. developing re-usable basic theories (the shaded rectangle) upon which the proof relies

Obviously, for more economical HOL-proofs, the dotted lines must be pruned and decrease in number, the solid lines must become shorter, and the shaded rectangle must become larger. Proof-engineering and the tools that are used to mechanically verify the proof are propagated as means to reach these goals.

6.3.1 Proof engineering

To prune and decrease the number of dotted lines, we found that it is recommendable, for complex proofs, to

> *make a pencil-and-paper proof, despite the fact that machine-checked verification will be done later.*

A common mistake is to think that precise and formal pencil-and-paper proofs are not required because all mistakes shall be filtered out during mechanical validation. Although the latter is partially true, such an attitude can cost lots of time and effort when a theorem prover is used. Discovering mistakes during verification in HOL is marvellous, for it demonstrates the necessity of using such a prover. On the other hand, however, mistakes have to be repaired, i.e. HOL definitions have to be changed and the proofs must be redone accordingly. This is a tedious and time-consuming process which sometimes could have been prevented by more accuracy *before* mechanical verification is attempted. So, summing up, the motto is: "Do not get sloppy just because mechanical verification will be done later, it can save you lots of time." Another advantage of making formal pencil-and-paper proofs, is that these proofs can serve as a proof strategy with which the theorem can be proved in HOL. Consequently, they facilitate constructing a HOL proof, and reduce the time which is spent on insufficient proof-strategies,

In order to create easier proofs and hence shorten the solid lines, an important precept is:

do not confuse software-engineering with proof-engineering.

While writing a program (i.e. modelling in a programming language; software-engineering) many choices have to be made, e.g. regarding the variables used (how many, of what type, for which purpose). In software-engineering these choices have to be made in such a way that the whole results in an efficient program. During verification (i.e. modelling in logic; proof-engineering), the same choices also have to be made. However, verification revolves around correctness of the program and not its efficiency. Consequently, in proof-engineering the decisions made can differ from those in software-engineering. For example, suppose there are parts in the proof for which it is beneficial if a network is modelled as a set of processes, and suppose there are other parts which prefer lists. We say, use both representations, add an assumption which indicates that they contain the same elements, and, at any point, use the one which is most suitable. In short:

If it makes your proof easier and speeds up the verification process: Just do it!

More re-usable theories, resulting in larger rectangles, can shorten the solid lines of future proofs. Identifying aspects of the proof as candidates for re-usable theory is not easy, and most of the time becomes clear after completing the proof. Some readers may now try to find page 70 because they remember something similar there, and indeed there is. Analogous to what is stated there, we propose analysing the problem, trying to obtain a good understanding of its essence, as an important means to advance the detection of re-usable components. Although at this point, "analysing" may still appear as vague to the reader, we promise to get more explicit in subsequent chapters.

6.3.2 Tools

On the side of the tools, topics that have been suggested in literature are the following. Improving the user interfaces of theorem provers [TBK92, Thé93, SB94, Sym95, AGMT95, Mer95, Gra96, Ait96, BL96, Ber97, Bac98], for better user interfaces imply better ways of visualising theories, enhanced accessibility, reducing the steepness of the learning curve, high quality of data presentation, and enhanced error messages.

Improving the efficiency of theorem provers. In fully-expansive theorem provers, like HOL, the proofs, generated in the system, are composed of applications of the primitive inference rules of the underlying logic. Advantages are that the prover's soundness only depends on the implementations of the primitive inference rules; and that users are free to write proof procedures without the risk of making the system unsound. The disadvantage, however, is that resolving proofs into simple primitive inferences can make the prover slow. In [Bou93] efficiency of fully-expansive theorem provers is improved by eliminating duplicated and unnecessary primitive inferences.

Increasing availability of decision procedures. Decision procedures are an important tool in theorem provers. They allow much low-level reasoning to be performed automatically. Lemmas and theorems that appear trivial may take many minutes or even days to prove by hand, especially for inexperienced users. Decision procedures

can relieve users of some of this burden. `ARITH_CONV`, for example, was very useful to us, and saved us lots of time. Unfortunately, the HOL system suffers from a relative lack of decision procedures. In addition to the Presburger procedure, the `taut` library provides a decision procedure for propositional tautologies, the library `faust` [SKK91] provides a decision procedure to check the validity of many formulae from first order predicate logic. Some references to work on decision procedures in HOL are [SKK93, ALW93, Har93a, Har94, Bou95, Bra96, Har96].

Lastly, combining theorem provers with other tools is an emerging field of research. For example, in [BSBG98] an interface between the CLAM proof planner [BvHHS90] and HOL is described. Roughly, the interface sends HOL goals to CLAM for planning, and translates plans back into HOL tactics that solve the initial goals. Recently, [Gun98] has presented a method that allows the results from external decision procedures, like BBD's and model checkers, to be incorporated within HOL90 without compromising the latter's logical consistency. Some other references on combining theorem provers with model checkers or other tools include [JS93, LC93, GL95, How96, BGG+98].

The whole of science is nothing more than a refinement of everyday thinking.

–Albert Einstein, Physics and Reality

Chapter 7

Program refinement in UNITY

Program refinement has received a lot of attention in the context of stepwise development of correct programs, since the introduction of transformational programming techniques by [Wir71, Hoa72, Ger75, BD77] in the seventies. This chapter presents a new framework of program refinement, that is based on a refinement relation between UNITY programs. The main objective of introducing this new relation it to reduce the complexity of correctness proofs for existing classes of related distributed algorithms. It is shown, however, that this relation is also suitable for the stepwise development of programs, and incorporates most of the program transformations found in existing work on refinements. Section 7.1 gives an overview of some of the existing work on program refinements that has inspired our notion of program refinement. Moreover, an exposition on the word refinement and its uses in technical contexts is given. Section 7.2 presents the motivation for having a new refinement relation, and its formal definition, the properties, and different uses. Section 7.3 finally concludes.

7.1 An overview of some existing work on refinements

Whereas the word *refinement* has been used in technical contexts in several related but subtly different ways, we can only give an overview after we have agreed on what is considered to be a refinement and, more important, what refinements are being considered. In Webster's college dictionary [Inc95], refinement is defined as:

refinement *n.* **1.** fineness or elegance of feeling, taste, manners, language, etc. **2.** an instance of this. **3.** the act or process of refining. **4.** the quality or state of being refined. **5.** a subtle point of distinction. **6.** an improved form of something. **7.** a detail or device added to improve something.

and all senses but **1** accord with the uses in computer science related contexts. We shall start by making a clear distinction between *program* refinement on the one hand and *property* refinement on the other.

Property refinement already occurred in Section 6.1 within the context of the UNITY methodology for developing distributed programs. Here, a high level UNITY specification – which, within the UNITY methodology, is a property and *not* a program – is refined by adding more detail to it (i.e. **7** of Webster's definition). The specification is improved in the sense that, being more detailed by exploiting some solution strategy, it gets easier to derive the final UNITY program that satisfies the initial specification. This kind of property refinement, or specification refinement is in some work also referred to as *reification* [Jac91].

Program refinement is the activity of transforming a complete program in order to improve something (i.e. **6** and **7.** of Webster's definition). This something can be the program itself (i.e. efficiency, costs, representation, etcetera), or the complexity of the correctness proof of the program. Although the *definition* that states when one program is considered to be a refinement of another differs among existing work on program refinements (see the sections below), the type or kind of program refinements (or program transformations) that are studied are generally the same. Before we discuss existing work on program refinement, we shall give the meanings of these different kinds of refinements.

data refinement is a program transformation where a (high-level, abstract) data structure is replaced by another (lower-level, concrete) data structure. It was first introduced in [Hoa72], and is very useful for improving the efficiency of programs.

atomicity refinement is a program transformation where a program with a coarse grain of atomicity is transformed into another program that uses a finer grain of atomicity. It is a useful transformation rule. On the one hand, proving algorithms with a coarse grain of atomicity is easier since fewer interleavings have to be considered. On the other hand, distributed algorithms that use a fine grain of atomicity are potentially faster as more processes may execute concurrently.

strengthening guards is a program transformation of which the name speaks for itself.

superposition refinement is a program transformation that, as we already discussed in Section 4.7, adds new functionality to an program in the form of additional variables and assignments to these variables.

The existing work that shall be discussed in the following sections is concerned with program refinements of distributed or concurrent programs.

7.1.1 The refinement calculus

The refinement calculus originates with Ralph Back [Bac78, Bac80] and was reintroduced by Joseph Morris [Mor89] and Carrol Morgan [Mor88, MG90, Mor90]. The calculus provides a framework for systematic program development.

The main idea behind the refinement calculus is considering both specifications and code to be *programs*. A notion of refinement is then defined on these *programs* as a reflexive and transitive relation that preserves total correctness[1]. More specifically, a program P is refined by another program P' (denoted by $P \leq P'$ or $P \sqsubseteq P'$) if, when both P and P' are started in the same state:

- if P terminates so does P'
- the set of final states of P' is contained in the set of final states of P

This notion of refinement is defined using Dijkstra's weakest pre-condition calculus [Dij76]. Note that this definition of refinement is *not* a property preserving refinement. All we know when $P \leq P'$ is that the input-output correctness is preserved; it does not guarantee that the behaviour of P' during execution, and thus its temporal properties, will be the same as the behaviour of P. Since the refinement calculus was originally designed for sequential programs total correctness was sufficient. The refinement calculus has however been lifted to work on both parallel [Bac89, Bac90, Ser90, BS91, Bac93] and reactive (or distributed) [Bac90, vW92b, BvW94, BS96] systems, by using action systems [BKS83, BKS84, BKS88] to model parallel and distributed systems as sequential programs. Although preserving total correctness is also sufficient for parallel systems, stepwise refinement of reactive or distributed systems also requires preservation of temporal properties. Consequently, in [Bac90, vW92b, BvW94, BS96] the notion of refinement was extended such that the preservation of temporal properties was guaranteed.

The development of a program within the refinement calculus framework consists of a sequence of correctness (or in the case of distributed systems, temporal properties) preserving refinement steps, starting with an initial high-level specification and ending with an efficient executable program. These correctness preserving refinement steps are formulated as program transformations rules $t \in \mathit{programs} \rightarrow \mathit{programs}$ and added to the refinement calculus framework by proving theorems of the form:

$$\forall P \in \mathit{programs} :: \frac{\text{Verification Conditions hold for } P}{P \leq t.P}$$

In other words if certain verification conditions are satisfied, then applying rule t to program P is a correctness (and in the case of distributed systems, temporal properties) preserving refinement step. Many transformation rules can be found in [Bac88, Bac89, BvW89, BvW90, Bac90, Ser90, BS91, vW92a, vW92b, Bac93, BS96, SW97, BvW98, BKS98], concerning among others, data refinement, guard strengthening, superposition refinement, and atomicity refinement (or changing the granularity).

Some other references on uses of the refinement calculus for distributed systems include [SW94a, SW94b, SW96], where the refinement steps are applied backwards in order to obtain a formal approach to reverse engineering distributed algorithms. In [Wal96, BW96, WS96, Wal98a, Wal98b, BW98] action systems and their refinements are formalised and applied within the B-method [Abr96].

[1] In [Bac81] a notion of partial correctness preserving refinement is studied.

7.1.2 Sanders' mixed specifications and refinement mappings

Sanders [San90] has introduced a mixed specification technique (called *mspecs*) to define a notion of *program* refinement in UNITY. An *mspec* incorporates both program text and a set of program properties. More specifically, an *mspec* consists of a *declare* section that contains a list of variables together with their types (the Cartesian product of these variables is referred to as the *state space* of the *mspec*); an *initially* section that contains a predicate that specifies the allowed initial values of the variables; an *assign* section that contains a set of conditional assignment statements that, in an operational view, constrain the behaviour of the program by specifying allowed state changes; a *property* section containing a set of program properties (expressed in a modified[2] version of the UNITY logic) that further constrain the allowed state changes, and for the progress properties, the allowed sequence of state changes.

Consequently, if the assign section is empty, an *mspec* is a standard UNITY specification, and if the properties section is empty an *mspec* is a standard UNITY program. An *mspec* is called implementable when all properties in the property section can be proved to hold for the actions in the assign section.

A benefit of specifying UNITY programs with a mixed specification is the following. Some desired program properties, like e.g. safety properties, are easier and more intuitively expressed using statements instead of logic, while others (usually progress properties) are better specified using logics [Lam83, Lam89]. In an *mspec* one can benefit from both possibilities, which is good since getting a specification right in the first place is crucial and not always easy.

A notion of refinement is *defined* on *mspec*s which is based on a refinement mapping [Lam83, LS84, AL91, Lam94, Lam96] $\mathcal{M}$ from the state space of the refinement to the state space of the original. It is denoted by $(G \, refines \, F)_{\mathcal{M}}$, and informally means:

- all initial conditions of G are mapped by $\mathcal{M}$ to the initial conditions on F
- if a state change from y_0 to y_1 is permitted by the assignments in the assign section of G, then either a state change from $\mathcal{M}.y_0$ to $\mathcal{M}.y_1$ is permitted by the the assignments in the assign section of F, or $\mathcal{M}.y_0$ equals $\mathcal{M}.y_1$.
- all properties of F are implied by the properties of G

Using this definition, several theorems are *proved* that state under which conditions a property that holds in an *mspec* can be considered to hold in a refinement. To give an indication of what these theorems look like, the $\mapsto$ preservation theorem is copied below: [San90, page 13]:

$$\frac{{}_F\vdash p \mapsto q \ \wedge \ (G \ refines \ F)_{\mathcal{M}} \qquad \forall i \ : (\ {}_F\vdash r_i \ \mathsf{ensures} \ q_i \text{ is used in the proof of } {}_F\vdash p \mapsto q) : \ {}_G\vdash r_i \circ \mathcal{M} \ \mathsf{ensures} \ q_i \circ \mathcal{M}}{{}_G\vdash p \circ \mathcal{M} \ \mapsto \ q \circ \mathcal{M}}$$

Similar theorems are given for preservation of unless , ensures , and fixed points. Moreover, a theorem is proved that states when the program transformation of replacing a shared variable by a message communication system is a property preserving (data) refinement. Stepwise derivation of programs within this framework now consists of a sequence of refined *mspec*s, starting with an *mspec* containing a high level

[2] The modified version was defined to eliminate the need of the substitution axiom [San91] (see Chapter 4)

of abstraction, and ending up with an *mspec* that is implementable.

7.1.3 A.K. Singh

In [SO89, Sin89b, Mis90, Sin91, Sin93] refinement of UNITY programs is investigated. Notions of property preserving and total correctness preserving (or fixed-point preserving, as it is called in [Sin93]) refinements are defined[3] as follows: [Sin93, page 511]:

Let F and G be two programs. G is a *property-preserving* refinement of F iff for all predicates p, q, the following two assertions hold:

- ${}_F\vdash p \text{ unless } q \;\Rightarrow\; {}_G\vdash p \text{ unless } q$
- ${}_F\vdash p \mapsto q \;\Rightarrow\; {}_G\vdash p \mapsto q$

Similarly, G is a *fixed-point preserving* refinement of F iff

- ${}_F\vdash \mathsf{true} \mapsto \mathsf{FP}_F \;\Rightarrow\; {}_G\vdash \mathsf{true} \mapsto \mathsf{FP}_G$
- $(\mathsf{FP}_G \wedge \mathsf{SI}_G) \;\Rightarrow\; (\mathsf{FP}_F \wedge \mathsf{SI}_F)$

where FP_P is the fixed point of a program P, i.e. it characterises the collection of states that are invariant under the execution of every statement in P; SI_P denotes the strongest invariant of a program P, i.e. it denotes the set of states reachable from the initial state.

Having *defined* these two notions of refinement, theorems are *proved* stating under which conditions certain program transformations are property-preserving and fixed-point preserving refinements. To give an indication of what these theorems look like, a theorem, stating the verification conditions under which strengthening the guard of a program is a property and fixed-point preserving refinement, looks like: [Sin89b, page 1] [Mis90] [Sin93, page 519]

Theorem Let F be a program and let $s :: A \textit{ if } p$ be a statement. Let statement $t :: A \textit{ if } p \wedge q$ be obtained by strengthening the guard of statement s. Then, program $F \,[\!]\, t$ is a property and a fixed-point preserving refinement of the program $F \,[\!]\, s$ if the following two conditions hold.

- ${}_F\vdash p \mapsto q$
- There exists a non-increasing function g from the program variables to a well-founded set such that ${}_F\vdash (g = k \;\wedge\; q) \text{ unless } (\neg p \;\vee\; g < k)$, for all k

In [Sin93] similar theorems are proved for program transformations like data refinement and atomicity refinement, and applied to to a number of examples.

7.1.4 Further reading

For some other work on refinement concepts within the UNITY (or a UNITY-like) framework, the reader can for example read [ZGK90, Jon90, Kor91, Udi95, UK96, Din97, GKSU98].

[3] The definitions of unless , ensures , and $\mapsto$ of Sanders' logic [San91] are used.

7.2 Another notion of refinement in UNITY

Like Sanders, but unlike Back and Singh, our refinement relation is *not* defined to be property or correctness preserving, and accordingly additional theorems have to be proved that state conditions under which properties of a program are preserved in its refinement. These conditions, however, do not look like the ones in Sanders, but relate to the verification conditions of the theorems in Back and Singh that argue about the property preservation of specific program transformation rules. The main difference between our refinement relation and the ones described in the previous sections, is that its purpose it not the stepwise derivation of correct programs but the reduction of complexity of correctness proofs of existing classes of related algorithms. The next section shall exemplify this.

7.2.1 Why another notion of refinement?

Guard strengthening and superposition are transformations for the step-wise development of programs, the formalisation of which was discussed in Chapter 4. This section exemplifies why these program refinements are sometimes insufficient to refine a program, and hence motivates the introduction of our new refinement relation.

Suppose we have a class of similar algorithms that seemingly establish the same progress in various ways. Most of the time, algorithms in such a class differ by having different mechanisms or control structures that influence their control flow and degree of non-determinism. Sometimes, however, adding such a mechanism or control structures, does not consist of one transformation, but a sequence (or composition) of transformations which as a whole are a property preserving transformation but on their own they are not. Consider, for example the following simple UNITY program which is in the class of algorithms that, started with initial values $x = 0$ and $y = 0$, increments both x and y until they have the value 10.

prog	P	
read	$\{x, y\}$	
write	$\{x, y\}$	
init	$x = 0 \wedge y = 0$	
assign	**if** $x \leq 10$ **then** $x := x + 1$	P_x
$[\!]$	**if** $y \leq 10$ **then** $y := y + 1$	P_y

Figure 7.1: Program P that increments both x and y until they have the value 10 ◀

It is easy to prove that $\mathsf{true}\ {}_P\!\vdash\ x = 0 \wedge y = 0 \leadsto x = 10 \wedge y = 10$ (see Figure 7.3). Another algorithm in this class is one that reduces the non-determinism of P in such a way that the value of x and y are incremented in an alternating way. Obviously, this more deterministic program also satisfies (for some J) $J \vdash x = 0 \wedge y = 0 \leadsto x = 10 \wedge y = 10$, and can be constructed by introducing a variable *x_turn* of type bool – the value of which indicates that it is x's turn – and transforming P as follows:

prog	Q	
read	$\{x, y, x\_turn\}$	
write	$\{x, y, x\_turn\}$	
init	$x = 0 \wedge y = 0 \wedge x\_turn = \mathsf{true}$	
assign	**if** $x < 10 \ \wedge \ x\_turn$ **then** $x := x + 1 \parallel x\_turn := \neg x\_turn$	Q_x
$[]$	**if** $y < 10 \ \wedge \ \neg x\_turn$ **then** $y := y + 1 \parallel x\_turn := \neg x\_turn$	Q_y

Figure 7.2: Program Q; reducing P's non-determinism

The machinery for superposition refinements in UNITY, formalised in Chapter 4, is inadequate for proving that this transformation is a property preserving one. This is because if we augment the P_x with assignment $x\_turn := \neg x\_turn$ to yield the program $\mathsf{AUG\_S}.P.\{P_x\}.(x\_turn := \neg x\_turn).(x\_turn = \mathsf{true})$, then we cannot subsequently augment action P_y (of $\mathsf{AUG\_S}.P.\{P_x\}.(x\_turn := \mathsf{false}).(x\_turn = \mathsf{true})$) with the assignment $x\_turn := \neg x\_turn$ and prove that the properties are preserved, since the write variables of $\mathsf{AUG\_S}.P.\{P_x\}.(x\_turn := \neg x\_turn).(x\_turn = \mathsf{true})$ (i.e. $\mathbf{w}P \cup \{x\_turn\}$) are *not* ignored by the assignment $x\_turn := \neg x\_turn$. Consequently, the formalisation of the UNITY superposition rules are not sufficient to prove preservation of properties under these kind of non-determinism reducing refinements. However, these refinements are very powerful for reducing the complexity of a correctness proof for a class of distributed programs. Non-deterministic programs are often easier to prove than more deterministic ones, since simplicity is gained by avoiding unnecessary determinism. To illustrate this we have displayed the proof of $x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$ for programs P and Q in Figures 7.3 and 7.4 respectively. It is not hard to see that the proof in Figure 7.3 is simpler than the proof in Figure 7.4. One reason for this is that, because of P's freedom to increment x and y whenever it wants (i.e. non-determinism), we are able to decompose the proof obligation $x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$ into the simpler proof obligations $x = 0 \rightsquigarrow x = 10$ and $y = 0 \rightsquigarrow y = 10$. For program Q this is an inefficacious proof strategy because x and y cannot be increased independently. Another reason is that, because of Q's restricted freedom to increase x and y (i.e. determinism), additional case distinctions on whether it is x's turn or not have to be made in order to be able to prove that progress can indeed be made.

Although this is just a simple example, it suggest that the total proof effort can be significantly reduced if we have a refinement relation supporting non-determinism reducing refinement. Since then, instead of laboriously proving properties directly for a more deterministic program Q, we can reduce the proof-complexity by proving these properties for the least deterministic variant P of Q, and conclude that these properties also hold for Q.

7.2.2 The formal definition of our refinement relation

We start by defining the refinement relation between two actions. Suppose we have two actions $A_l, A_r \in \texttt{ACTION}$, a state-predicate J, and a set of variables V, we say

$\mathsf{true}\ {}_P\vdash\ x=0\wedge y=0 \quad x=10\wedge y=10$
$\Leftarrow$ (Conjunction ($4.5.19_{49}$), Substitution ($4.6.3_{50}$))
$\mathsf{true}\ {}_P\vdash\ x\le 10 \quad x=10\ \wedge\ \mathsf{true}\ {}_P\vdash\ y\le 10 \quad y=10$
We continue with the first conjunct, the proof of the second conjunct is similar.
$\mathsf{true}\ {}_P\vdash\ x\le 10 \quad x=10$
$\Leftarrow$ (Bounded Progress ($4.5.20_{49}$), and ${}_P\vdash\ x=10$)
$\mathsf{true}\ {}_P\vdash\ x\le 10\wedge(10-x=k) \quad (x\le 10\wedge(10-x<k))\vee x=10$
$\Leftarrow$ (Case distinction ($4.6.7_{50}$) $x=10\vee x\neq 10$, Reflexivity ($4.6.5_{50}$), ${}_P\vdash\ x=10$, and Substitution ($4.6.3_{50}$))
$\mathsf{true}\ {}_P\vdash\ x<10\wedge(10-x=k) \quad x\le 10\wedge(10-x<k)$
$\Leftarrow$ (Introduction ($4.6.4_{50}$), and ${}_P\vdash\ x\le 10\wedge(10-x<k)$)
${}_P\vdash x<10\wedge(10-x=k)\ \mathsf{ensures}\ x\le 10\wedge(10-x<k)$

Figure 7.3: Proof of $\mathsf{true}\ {}_P\vdash\ x=0\wedge y=0 \rightsquigarrow x=10\wedge y=10$

Take $J=(\neg x\_turn\Rightarrow y=x-1)\wedge(x\_turn\Rightarrow y=x)$, and prove that ${}_Q\vdash\ J$.

$J\ {}_Q\vdash\ x=0\wedge y=0 \quad x=10\wedge y=10$
$\Leftarrow$ (Substitution ($4.6.3_{50}$))
$J\ {}_Q\vdash\ x\le 10\wedge y\le 10 \quad x=10\wedge y=10$
$\Leftarrow$ (Bounded Progress ($4.5.20_{49}$), and ${}_Q\vdash\ x=10\wedge y=10$)
$J\ {}_Q\vdash x\le 10\wedge y\le 10\wedge(20-x-y=k) \quad (x\le 10\wedge y\le 10\wedge(20-x-y<k))\vee(x=10\wedge y=10)$
$\Leftarrow$ (Case distinction ($4.6.7_{50}$) $y=10\vee y\neq 10$, Introduction ($4.6.4_{50}$), and Substitution ($4.6.3_{50}$) and $(J\wedge x\le 10\wedge y\le 10\wedge y=10)\Rightarrow(x=10\wedge y=10)$)
$J\ {}_Q\vdash x\le 10\wedge y<10\wedge(20-x-y=k) \quad (x\le 10\wedge y\le 10\wedge(20-x-y<k))\vee(x=10\wedge y=10)$
$\Leftarrow$ (Case distinction ($4.6.7_{50}$) $x=10\vee x\neq 10$, and Substitution ($4.6.3_{50}$))
$J\ {}_Q\vdash\ x=10\wedge y<10\wedge(20-x-y=k) \quad (x\le 10\wedge y\le 10\wedge(20-x-y<k))$
$\wedge$
$J\ {}_Q\vdash\ x<10\wedge y<10\wedge(20-x-y=k) \quad (x\le 10\wedge y\le 10\wedge(20-x-y<k))$
The first conjunct can be proved by Introduction ($4.6.4_{50}$), since $(J\wedge x=10\wedge y<10)$ implies $\neg x\_turn$, and thus
${}_Q\vdash J\wedge x=10\wedge y<10\wedge(20-x-y=k)\ \mathsf{ensures}\ x\le 10\wedge y\le 10\wedge(20-x-y<k)$
We continue with the second conjunct as follows:
$J\ {}_Q\vdash\ x<10\wedge y<10\wedge(20-x-y=k) \quad x\le 10\wedge y\le 10\wedge(20-x-y<k)$
$\Leftarrow$ (Case distinction ($4.6.7_{50}$) $(x\_turn\vee\neg(x\_turn))$)
$J\ {}_Q\vdash\ x<10\wedge y<10\wedge(20-x-y=k)\wedge x\_turn \quad x\le 10\wedge y\le 10\wedge(20-x-y<k)$
$\wedge$
$J\ {}_Q\vdash\ x<10\wedge y<10\wedge(20-x-y=k)\wedge\neg(x\_turn) \quad x\le 10\wedge y\le 10\wedge(20-x-y<k)$
$\Leftarrow$ (Introduction ($4.6.4_{50}$) on both conjuncts)
${}_Q\vdash\ J\wedge x<10\wedge y<10\wedge(20-x-y=k)\wedge x\_turn\ \mathsf{ensures}\ x\le 10\wedge y\le 10\wedge(20-x-y<k)$
$\wedge$
${}_Q\vdash\ J\wedge x<10\wedge y<10\wedge(20-x-y=k)\wedge\neg x\_turn\ \mathsf{ensures}\ x\le 10\wedge y\le 10\wedge(20-x-y<k)$

Figure 7.4: Proof of $J\ {}_Q\vdash\ x=0\wedge y=0 \rightsquigarrow x=10\wedge y=10$

that A_l is refined by A_r, or A_r refines A_l, with respect to V and J (denoted by $A_l \sqsubseteq_{V,J} A_r$), when:

- the conjunction of J with the guard of A_r is stronger then the guard of A_l.
- the results of A_l and A_r, both executed in the same state s where $J.s$ holds, on the variables in V are the same.

Definition 7.2.1 Action Refinement *A_ref_DEF*

Let A_l and A_r be two actions from the universe `ACTION`, J be a state predicate, and V be a set of variables, then action refinement is defined as follows:

$$\begin{array}{lcl} A_l \sqsubseteq_{V,J} A_r & = & \forall s \;::\; \mathsf{guard\_of}.A_r.s \wedge J.s \Rightarrow \mathsf{guard\_of}.A_l.s \\ & & \wedge \\ & & \forall s,t,t' \;::\; (\mathsf{compile}.A_l.s.t \wedge \mathsf{compile}.A_r.s.t' \wedge \mathsf{guard\_of}.A_r.s \wedge J.s) \\ & & \qquad \Rightarrow (t \restriction V = t' \restriction V) \end{array}$$

◀

The fact that action refinement is reflexive and transitive is captured by the following theorems.

Theorem 7.2.2 action refinement Reflexivity *A_ref_REFL*

For all $A \in$ `ACTION`, $J \in$ `Expr`, and sets of variables V:

$$A \sqsubseteq_{V,J} A$$

Theorem 7.2.3 action refinement Transitivity *A_ref_TRANS1*

For all $A_1, A_2, A_3 \in$ `ACTION`, $J_2, J_3 \in$ `Expr`, and sets of variables V_1, V_2 and V_3:

$$\frac{J_3 \Rightarrow J_2 \;\wedge\; V_3 \subseteq V_1 \;\wedge\; V_3 \subseteq V_2 \;\wedge\; A_1 \sqsubseteq_{V_1,J_2} A_2 \;\wedge\; A_2 \sqsubseteq_{V_2,J_3} A_3}{A_1 \sqsubseteq_{V_3,J_3} A_3}$$

◀

Next, we define our relation of program refinement. P is refined by Q, or Q refines P, with respect to some relation $\mathcal{R}$ and state-predicate J, (denoted by $P \sqsubseteq_{\mathcal{R},J} Q$), if we can decompose the actions of program Q into $\mathbf{a}Q_1$ and $\mathbf{a}Q_2$, such that

- $\mathcal{R}$ is a bitotal relation (see Definition A.3.3$_{217}$) on the two sets of actions $\mathbf{a}P$ and $\mathbf{a}Q_1$, i.e. for every action A_P in $\mathbf{a}P$ there exists at least one action in $\mathbf{a}Q_1$ to which $\mathbf{a}P$ is related by $\mathcal{R}$, and similarly for every action A_Q in $\mathbf{a}Q_1$ there exists at least one action in $\mathbf{a}P$ to which A_Q is related by $\mathcal{R}$.
- for all actions A_P of $\mathbf{a}P$ and A_Q of $\mathbf{a}Q_1$ that are related to each other by $\mathcal{R}$ (i.e. $A_P \;\mathcal{R}\; A_Q$ holds), we can prove that A_Q refines A_P with respect to the write variables of P and state-predicate J.

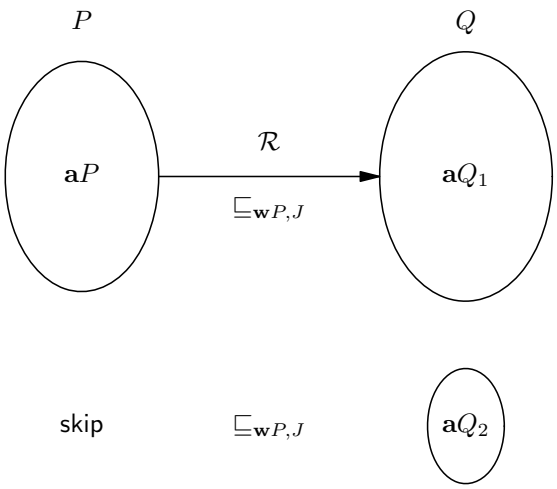


Figure 7.5: Program refinement in a picture.

- the actions of Q that are in $\mathbf{a}Q_2$ refine skip with respect to the write variables of P and J.

For those readers that are enlightened by pictures, in Figure 7.5 a depiction of program refinement is given. The formal definition of program refinement now reads:

Definition 7.2.4 Program Refinement *P_ref_DEF*

Let $P, Q \in$ `Uprog` be two UNITY programs, $\mathcal{R}$ be a relation, and $J \in$ `Expr` be a state predicate, then program refinement is defined as follows:

$$
\begin{array}{lll}
P \sqsubseteq_{\mathcal{R},J} Q & = \exists \mathbf{a}Q_1, \mathbf{a}Q_2 :: & \mathbf{a}Q = \mathbf{a}Q_1 \cup \mathbf{a}Q_2 \ \wedge\ \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.\mathbf{a}Q_1 \\
& & \wedge \\
& & \forall A_P\ A_Q\ : A_P \in \mathbf{a}P \ \wedge\ A_P\ \mathcal{R}\ A_Q\ : A_P \sqsubseteq_{\mathbf{w}P,J} A_Q \\
& & \wedge \\
& & \forall A_Q\ : A_Q \in \mathbf{a}Q_2\ : \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q
\end{array}
$$

Note that $P \sqsubseteq_{\mathcal{R},J} Q$ does not say anything about Q inheriting properties or correctness from P. Nor does it say anything about the explicit program transformations that were (or could have been) applied to P in order to obtain Q. Moreover note that, opposed to superposition refinement, $P \sqsubseteq_{\mathcal{R},J} Q$, does not necessarily imply that $\mathbf{w}P \subseteq \mathbf{w}Q$. Consider the two programs P and Q in Figure 7.6. Suppose z and w are different variables, then it can easily be seen that for any state-predicate J, and relation $\mathcal{R}$ defined by $\mathcal{R} = \{(aP_i, aQ_i) \mid i = 1, 2\}$, it holds that $P \sqsubseteq_{\mathcal{R},J} Q$. However, since z and w are different variables, $\mathbf{w}P \subseteq \mathbf{w}Q$ does not hold.

The following theorems state that program refinement is reflexive and under certain conditions also transitive.

prog	P		**prog**	Q	
read	$\{x, y, z\}$		**read**	$\{x, y, w\}$	
write	$\{x, y, z\}$		**write**	$\{x, y, w\}$	
init	$b = \mathsf{true}$		**init**	$b = \mathsf{true}$	
assign	$x := x + 1$	aP_1	**assign**	**if** $x \leq 15$ **then** $x := x + 1$	aQ_1
[]	$y := y + 1$	aP_2	[]	**if** $y \leq 20$ **then** $y := y + 1$	aQ_2
[]	$z := z$	aP_3	[]	$w := w + 1$	aQ_3

Figure 7.6: Q refines P

Theorem 7.2.5 Program refinement Reflexivity *P_ref_REFL*
For all $P \in \texttt{Uprog}$, and $J \in \texttt{Expr}$:

$$P \sqsubseteq_{=,J} P$$

Theorem 7.2.6 Program refinement Transitivity *P_ref_TRANS*
For all $P_1, P_2, P_3 \in \texttt{Uprog}$, and $J_2, J_3 \in \texttt{Expr}$:

$$\frac{J_3 \Rightarrow J_2 \ \wedge\ \mathbf{w}P_1 \subseteq \mathbf{w}P_2 \ \wedge\ P_1 \sqsubseteq_{\mathcal{R}_1,J_2} P_2 \ \wedge\ P_2 \sqsubseteq_{\mathcal{R}_2,J_3} P_3}{P_1 \sqsubseteq_{\mathcal{R}_1 \circ \mathcal{R}_2,J_3} P_3}$$

Reflexivity, and transitivity are necessary properties of a refinement relation, in order to make the latter suitable for the step-wise derivation of programs [Bac88]. However, our definition of refinement is not purely transitive in the sense that additional requirements on the component programs are demanded in the premises of Theorem 7.2.6 stating transitivity of $\sqsubseteq$. Suppose we want to derive program P_{n+1} from P_1 ($n > 1$) by the following sequence of refinements:

$$P_1 \sqsubseteq_{\mathcal{R}_1,J_2} P_2 \sqsubseteq_{\mathcal{R}_2,J_3} P_3 \sqsubseteq_{\mathcal{R}_3,J_4} P_4 \ldots \sqsubseteq_{\mathcal{R}_n,J_{n+1}} P_{n+1}$$

in order to conclude that

$$P_1 \sqsubseteq_{\mathcal{R}_1 \circ \ldots \circ \mathcal{R}_n,J_{n+1}} P_{n+1}$$

we have to prove that:

- the write variables of the program P_i in intermediate step $P_i \sqsubseteq_{\mathcal{R}_1 \circ \ldots \circ \mathcal{R}_i,J_{i+1}} P_{i+1}$ are included or equal to the write variables of program P_{i+1}
- the predicate J_{i+1} (which shall usually correspond to the strongest invariant of the program P_{i+1}) must be stronger than the predicate J_i (thus the strongest predicate of program P_i).

Consideration of the fact that the underlying transformations of these intermediate refinement steps are superposition, guard strengthening and atomicity refinement (see Section 7.2.6), these requirements are very natural. Consequently, our definition of refinement is very suitable for stepwise derivation and verification of distributed programs in UNITY.

7.2.3 Property preservation

Safety properties p unless q, and $\circlearrowleft J$, where p, q and J do not name any superposed variable, are always preserved under refinement of two UNITY programs.

Theorem 7.2.11 *P_ref_AND_SUPERPOSE_WRITE_PRESERVES_UNLESSe*

$$\frac{P \sqsubseteq_{\mathcal{R},J} Q \wedge \mathsf{Unity}.P \wedge \mathsf{Unity}.Q \wedge ({}_Q\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J) \quad \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (p\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (q\ \mathcal{C}\ W^{\mathsf{c}})}{{}_P\vdash p \text{ unless } q \Rightarrow {}_Q\vdash (J_Q \wedge p) \text{ unless } q}$$

Theorem 7.2.12 *P_ref_AND_SUPERPOSE_WRITE_PRESERVES_STABLEe*

$$\frac{P \sqsubseteq_{\mathcal{R},J} Q \wedge \mathsf{Unity}.P \wedge \mathsf{Unity}.Q \wedge ({}_Q\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J) \quad \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (p\ \mathcal{C}\ W^{\mathsf{c}})}{{}_P\vdash \circlearrowleft p \Rightarrow {}_Q\vdash \circlearrowleft (J_Q \wedge p)}$$

◀

The conditions $(p\ \mathcal{C}\ W^{\mathsf{c}})$ and $(q\ \mathcal{C}\ W^{\mathsf{c}})$, in the premises of the two theorems above, state that the values of state-predicates p, and q do not depend on the values of the variables in W. Note that when W is the set of variables that are superposed up on program P, these conditions are weaker then stating that p and q do not name any superposed variable (see Section 3.3 page 28).

Preservation of one-step progress properties (i.e. ensures) cannot be proved under our definition of refinement. Fortunately, preservation of reach and convergence properties can be proved, and in most situations these are all that are required.

Figure 7.7 shows the theorems stating verification conditions under which general progress properties are preserved by refinements. Theorem 7.2.7 is a generalisation of the theorem given in [Sin93] mentioned earlier in Section 7.1.3. It states verification conditions for property preservation not only under strengthening the guard of one action in a program, but under multiple compositions of guard strengthening, superposition and atomicity refinements on various actions in the program. Informally this theorem states that when a UNITY program Q refines P with respect to relation $\mathcal{R}$ and J, then the progress properties $p \rightarrowtail q$ and $p \leadsto q$ under the stability of predicate J_P in program P, are preserved under the stability of predicate $J_P \wedge J_Q$ in program Q, provided that the following verification conditions hold:

- $(J_P \wedge J_Q)$ is stable in Q.
- $(J_P \wedge J_Q)$ implies J
- p nor q depend on the values of the variables in W.

Let $\prec$ be a well-founded relation over some set A, and $M \in \texttt{State} \rightarrow A$.

Theorem 7.2.7 *P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_REACHe_GEN*
P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_CONe_GEN

$$\frac{\begin{array}{c} P \sqsubseteq_{R,J} Q \wedge \mathsf{Unity}.Q \wedge (\ {}_Q\vdash \quad J_P \wedge J_Q) \wedge (J_P \wedge J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\ \forall A_Q : A_Q \in \mathbf{a}Q \wedge (\exists A_P :: (A_P \in \mathbf{a}P) \wedge (A_P\ \mathcal{R}\ A_Q)) : (\mathsf{guard\_of}.A_Q\ \mathcal{C}\ \mathbf{w}Q) \\ \forall A_P : A_P \in \mathbf{a}P : (J_P \wedge J_Q)\ {}_Q\vdash \mathsf{guard\_of}.A_P \qquad (\exists A_Q :: (A_P\ \mathcal{R}\ A_Q) \wedge \mathsf{guard\_of}.A_Q) \\ \exists M :: (M\ \mathcal{C}\ \mathbf{w}Q) \wedge (\forall k : k \in A :\ {}_Q\vdash (J_P \wedge J_Q \wedge M = k)\ \mathsf{unless}\ (M \prec k)) \\ \quad \wedge \forall k\ A_P A_Q : k \in A \wedge A_P \in \mathbf{a}P \wedge A_P\ \mathcal{R}\ A_Q : \\ \qquad {}_Q\vdash (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q \wedge M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \vee M \prec k) \end{array}}{((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))\ \wedge\ ((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))}$$

Theorem 7.2.8 *P_ref_SUPERPOSE_PRESERVES_REACHe_GEN*
P_ref_SUPERPOSE_PRESERVES_CONe_GEN

$$\frac{\begin{array}{c} P \sqsubseteq_{R,J} Q \wedge \mathsf{Unity}.Q \wedge (\ {}_Q\vdash \quad J_P \wedge J_Q) \wedge (J_P \wedge J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\ \forall A_Q : A_Q \in \mathbf{a}Q \wedge (\exists A_P :: (A_P \in \mathbf{a}P) \wedge (A_P\ \mathcal{R}\ A_Q)) : (\mathsf{guard\_of}.A_Q\ \mathcal{C}\ \mathbf{w}Q) \\ \forall A_P : A_P \in \mathbf{a}P : (J_P \wedge J_Q)\ {}_Q\vdash \mathsf{guard\_of}.A_P \qquad (\exists A_Q :: (A_P\ \mathcal{R}\ A_Q) \wedge \mathsf{guard\_of}.A_Q) \\ \forall A_P\ A_Q : A_P \in \mathbf{a}P \wedge A_P\ \mathcal{R}\ A_Q :\ {}_Q\vdash (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q)\ \mathsf{unless}\ \neg(\mathsf{guard\_of}.A_P) \end{array}}{((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))\ \wedge\ ((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))}$$

Theorem 7.2.9 *P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_REACHe*
P_ref_SUPERPOSE_AND_WF_FUNC_PRESERVES_CONe

$$\frac{\begin{array}{c} P \sqsubseteq_{R,J} Q \wedge \mathsf{Unity}.Q \wedge (\ {}_Q\vdash \quad J_P \wedge J_Q) \wedge (J_P \wedge J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\ \forall A_P\ A_Q :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ : (J_P \wedge J_Q)\ {}_Q\vdash \mathsf{guard\_of}.A_P \qquad \mathsf{guard\_of}.A_Q \\ \exists M :: (M\ \mathcal{C}\ \mathbf{w}Q) \wedge (\forall k : k \in A :\ {}_Q\vdash (J_P \wedge J_Q \wedge M = k)\ \mathsf{unless}\ (M \prec k)) \\ \quad \wedge \forall k\ A_P A_Q : k \in A \wedge A_P \in \mathbf{a}P \wedge A_P\ \mathcal{R}\ A_Q : \\ \qquad {}_Q\vdash (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q \wedge M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \vee M \prec k) \end{array}}{((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))\ \wedge\ ((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))}$$

Theorem 7.2.10 *P_ref_AND_SUPERPOSE_WRITE_PRESERVES_REACHe*
P_ref_AND_SUPERPOSE_WRITE_PRESERVES_CONe

$$\frac{\begin{array}{c} P \sqsubseteq_{R,J} Q \wedge \mathsf{Unity}.Q \wedge (\ {}_Q\vdash \quad J_P \wedge J_Q) \wedge (J_P \wedge J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\ \forall A_P\ A_Q :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ : (J_P \wedge J_Q)\ {}_Q\vdash \mathsf{guard\_of}.A_P \qquad \mathsf{guard\_of}.A_Q \\ \forall A_P\ A_Q :\ A_P \in \mathbf{a}P\ \wedge\ A_P\ \mathcal{R}\ A_Q\ :\ {}_Q\vdash (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q)\ \mathsf{unless}\ \neg(\mathsf{guard\_of}.A_P) \end{array}}{((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))\ \wedge\ ((J_P\ {}_P\vdash p \qquad q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \qquad q))}$$

Figure 7.7: Preservation of $\rightarrowtail$ and $\leadsto$ properties.

- the guards of those actions A_Q of Q that are related by $\mathcal{R}$ to one or more actions from P are confined by the write variables of Q.
- for all actions A_P of program P; if the guard of A_P holds in Q, then eventually there will exists an action A_Q of Q that is related to A_P by $\mathcal{R}$, and the guard of which becomes true in Q. Consequently, if A_P can make progress in P, then eventually there exists at least one action of A_Q of Q that, when executed in Q, *can* make the same progress on the write variables of P as A_P does when executed in P.

  Note that this requirement is not enough to guarantee that A_Q indeed makes the same progress as A_P, since between the point in time that the guard of A_Q becomes true, and the actual execution of A_Q it is possible that the guard of A_Q is prematurely falsified and no progress is made by A_Q whatsoever. The next (and last) verification condition states that this premature falsification of the guard of A_Q cannot happen infinitely and hence ensures that eventually A_Q *will* make the same progress as A_P on the write variables of program P.
- for all actions A_P of program P and those actions A_Q of Q that are related to A_P by $\mathcal{R}$, there exists a function M that is non-increasing with respect to some well-founded relation $\prec$, such that: if the guard of A_Q is true and M equals some value k at any point during the execution of Q, then either:
  - the guard of A_P always holds, the value of M always remains k, and the guard of A_Q continues to hold forever, so both actions can make the same progress;
  - eventually M decreases or the guard of A_P becomes false, but at least until this happens, M remains k and the guard of A_Q continues to hold.

  Consequently, if the guard of A_Q is prematurely falsified while the guard of A_P still holds, then we know that the value of M has decreased. By the previous verification condition we know that eventually the guard of A_Q will become true again, and hence given a chance to execute. Again, the guard of A_Q can be prematurely falsified, and we have the same process all over again. However, the well-foundedness of $\prec$ guarantees that M cannot decrease infinitely, and hence that premature falsification of the guard of A_Q cannot happen infinitely.

Theorem 7.2.8 states a corollary of theorem 7.2.7. It can be proved by taking M to be a constant function. Theorems 7.2.9 and 7.2.10 state corollaries of 7.2.7 and 7.2.8 respectively. These can be proved by using the theorem stated below.

Theorem 7.2.13 *BITOTAL_IMP_GUARD_REACH_EXIST_GUARD*

$$\frac{(\exists A :: \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.A) \qquad \forall A_P\ A_Q\ :\ A_P \in \mathbf{a}P \wedge A_P\ \mathcal{R}\ A_Q\ :\ J\ {}_Q\!\vdash \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_Q}{\forall A_P : A_P \in \mathbf{a}P : J\ {}_Q\!\vdash \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P\ \mathcal{R}\ A_Q) \wedge \mathsf{guard\_of}.A_Q)}$$

◀

Note that the Theorems in Figure 7.7 state property preservation in refinements independently from the specific program transformations that were applied.

7.2.4 Guard strengthening and superposition refinement

Strengthening the guard of, or augmenting an assignment on an action A are action refinements of A.

Theorem 7.2.14 *augment_A_ref*

For all $A, As \in \texttt{ACTION}$, $J \in \texttt{Expr}$, and V a set of variables:

$$\frac{\mathsf{is\_assign}.As \wedge V \twoheadleftarrow As \wedge \mathsf{WF\_action}.A \wedge \mathsf{WF\_action}.As}{A \sqsubseteq_{V,J} \mathsf{augment}.A.As}$$

Theorem 7.2.15 *strengthen_guard_A_ref*

For all $A \in \texttt{ACTION}$, $g, J \in \texttt{Expr}$, and V a set of variables:

$$A \sqsubseteq_{V,J} \mathsf{strengthen\_guard}.g.A$$

◀

Consequently, restricted union superposition and augmentation superposition on a program P are program refinements of P.

Theorem 7.2.16 *RU_Superpose_P_ref*

For all $P \in \texttt{Uprog}$, $A \in \texttt{ACTION}$, $J, iA \in \texttt{Expr}$:

$$\frac{\mathbf{w}P \twoheadleftarrow A}{P \sqsubseteq_{=,J} \mathsf{RU\_S}.P.A.iA}$$

Theorem 7.2.17 *AUG_Superpose_P_ref*

For all $P \in \texttt{Uprog}$, $As \in \texttt{ACTION}$, $iA \in \texttt{Expr}$, , and $ACs \subseteq \texttt{ACTION}$:

$$\frac{\mathbf{w}P \twoheadleftarrow A \ \wedge \ \mathsf{is\_assign}.As \ \wedge \ \mathsf{WF\_action}.As}{\exists \mathcal{R} \ :: \ P \sqsubseteq_{\mathcal{R},J} \mathsf{AUG\_S}.P.ACs.As.iA}$$

The witness used to prove this theorem is[a]:
$(\mathcal{R} = \mathsf{f2r}.(\lambda A.(A \in ACs) \rightarrow \mathsf{augment}.A.As \mid A))$.

[a]See Definition A.2.3$_{216}$ for the definition of the function f2r that converts a function to a relation.

◀

7.2.5 Non-determinism reducing refinement

Our definition of refinement in the previous section incorporates multiple compositions of guard strengthening and superposition program transformations, without having to specify these individual transformations explicitly. The requirement that

$$\forall A_P \ A_Q \ : A_P \in \mathbf{a}P \ \wedge \ A_P \ \mathcal{R} \ A_Q \ : A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$$

takes care of (possibly multiple compositions of) guard strengthening and augmentation superpositions. The requirement

$$\forall A_Q \ : A_Q \in \mathbf{a}Q_2 \ : \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q$$

takes care of (possibly multiple compositions of) restricted union superpositions. As a consequence, non-determinism reducing refinements like the one presented in Section 7.2.1, can be handled by our definition of refinement. Consider again programs P and Q from Figures 7.1$_{106}$ and 7.2$_{107}$ respectively. By taking $\mathcal{R} = \{(P_i, Q_i) \mid i \in \{x, y\}\}$, we can prove that for any J, $P \sqsubseteq_{\mathcal{R},J} Q$ holds. The proof of this is displayed below to give the interested reader an idea of the concepts involved; it may however be skipped.

proof of: $P \sqsubseteq_{\mathcal{R},J} Q$

$=$ (rewriting with Definition 7.2.4)
$\exists \mathbf{a}Q_1, \mathbf{a}Q_2 ::$ $\mathbf{a}Q = \mathbf{a}Q_1 \cup \mathbf{a}Q_2 \ \wedge\ \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.\mathbf{a}Q_1$
$\wedge$
$\forall A_P\ A_Q\ : A_P \in \mathbf{a}P \ \wedge\ A_P\ \mathcal{R}\ A_Q\ : A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$
$\wedge$
$\forall A_Q\ : A_Q \in \mathbf{a}Q_2\ : \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q$
$\Leftarrow$ (Reduce goal using witnesses $\mathbf{a}Q$ and $\emptyset$ respectively)
$\mathbf{a}Q = \mathbf{a}Q \cup \emptyset \ \wedge\ \mathsf{bitotal}.\mathcal{R}.\mathbf{a}P.\mathbf{a}Q$
$\wedge\ (\forall A_P\ A_Q\ : A_P \in \mathbf{a}P \ \wedge\ A_P\ \mathcal{R}\ A_Q\ : A_P \sqsubseteq_{\mathbf{w}P,J} A_Q) \ \wedge\ (\forall A_Q\ : A_Q \in \emptyset\ : \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q)$
$\Leftarrow$ ($\mathcal{R}$ is a bitotal; properties of $\cup$, $\in$, and $\emptyset$)
$\forall A_P\ A_Q\ : A_P \in \mathbf{a}P \ \wedge\ A_P\ \mathcal{R}\ A_Q\ : A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$
$=$ (actions of programs P and Q, definition of $\mathcal{R}$)
$P_x \sqsubseteq_{\mathbf{w}P,J} Q_x \ \wedge\ P_y \sqsubseteq_{\mathbf{w}P,J} Q_y$
$=$ (We shall prove the one for P_x the other is similar; Rewrite with Definition 7.2.1)
$\forall s\ ::\ \mathsf{guard\_of}.Q_x.s \wedge J.s \Rightarrow \mathsf{guard\_of}.P_x.s$
$\wedge\ \forall s, t, t'\ ::\ (\mathsf{compile}.P_x.s.t \wedge \mathsf{compile}.Q_x.s.t' \wedge \mathsf{guard\_of}.Q_x.s \wedge J.s) \Rightarrow (t\ \ \mathbf{w}P = t'\ \ \mathbf{w}P)$
$=$ ($\mathsf{guard\_of}.Q_x.s = (s.x \leq 10 \wedge s.x\_turn$), and $\mathsf{guard\_of}.P_x.s = s.x \leq 10$)
$\forall s, t, t'\ ::\ (\mathsf{compile}.P_x.s.t \wedge \mathsf{compile}.Q_x.s.t' \wedge s.x \leq 10 \wedge s.x\_turn \wedge J.s) \Rightarrow (t\ \ \mathbf{w}P = t'\ \ \mathbf{w}P)$
Discharge the antecedents of this goal into the assumptions after rewriting with P_x and Q_x
$\mathbf{A}_1$: $\mathsf{compile}.(\mathsf{GUARD}.(x \leq 10).(\mathsf{ASSIGN}.[x].[x+1])).s.t$
$\mathbf{A}_2$: $\mathsf{compile}.(\mathsf{GUARD}.(x \leq 10 \wedge x\_turn).(\mathsf{ASSIGN}.[x, x\_turn].[x+1, \mathsf{false}])).s.t'$
$\mathbf{A}_3$: $s.x \leq 10 \ \wedge\ s.x\_turn \ \wedge\ J.s$
Rewriting $\mathbf{A}_1$ and $\mathbf{A}_2$ with the Definition 3.4.18 of compile, Definition 3.4.13 stating the semantics of guarded actions, Definition 3.4.16 stating the semantics of assignment, and assumption $\mathbf{A}_3$ gives us:
$t = (\lambda v.(v = x) \rightarrow s.x{+}1 \mid s.v) \wedge t' = (\lambda v.(v = x) \rightarrow s.x{+}1 \mid ((v = x\_turn) \rightarrow \mathsf{false} \mid s.v))$
Consequently, $(t\ \ \mathbf{w}P = t'\ \ \mathbf{w}P)$, which equals, $(t\ \ \{x, y\} = t'\ \ \{x, y\})$ holds.

end of proof

Proving that the property $\mathsf{true}\ {}_P\vdash\ x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$ of program P is indeed preserved by its non-determinism reducing refinement Q can be established using Theorem 7.2.9. We already have that:

$\mathbf{A}_1$: $\mathsf{true}\ {}_P\vdash\ x = 0 \wedge y = 0 \rightsquigarrow x = 10 \wedge y = 10$
$\mathbf{A}_2$: $\mathcal{R} = \{(P_i, Q_i) \mid i \in \{x, y\}\}$
$\mathbf{A}_3$: $J = (\neg x\_turn \Rightarrow (y = x - 1)) \vee (x\_turn \Rightarrow (x = y))$
$\mathbf{A}_3$: ${}_Q\vdash\circlearrowleft J$
$\mathbf{A}_3$: $P \sqsubseteq_{\mathcal{R},J} Q$

prog	P	**prog**	Q
read	$\mathbf{r}P$	**read**	$\mathbf{r}P$
write	$\mathbf{w}P$	**write**	$\mathbf{w}P$
init	$\mathbf{ini}P$	**init**	$\mathbf{ini}P$
assign	**if** $(\exists i \ : i \in S : \ g.i)$ **then** A	**assign**	$[]_{i \in S}$ **if** $g.i$ **then** A

Figure 7.8: Q refines P

Now Theorem 7.2.9, using witnesses $W = \{x\_turn\}$ and $M = 20 - x - y$, and taking $\prec$ to be $<$ on numbers, leaves us with the following proof obligations:

- ${}_Q\vdash (J \wedge M = k)$ unless $(M < k)$
- ${}_Q\vdash (J \wedge y < 10 \wedge \neg(x\_turn) \wedge M = k)$ unless $(\neg(y < 10) \vee M < k)$
- ${}_Q\vdash (J \wedge x < 10 \wedge x\_turn \wedge M = k)$ unless $(\neg(x < 10) \vee M < k)$
- $J \ {}_Q\vdash x < 10 \rightarrowtail x < 10 \ \wedge \ x\_turn$
- $J \ {}_Q\vdash y < 10 \rightarrowtail y < 10 \ \wedge \ \neg x\_turn$

Proving these obligations is not hard, and is left to the reader. This is a small example, and the proof-effort is not significantly reduced when we compare the proof obligations in the bullets above with the ones in Figure 7.4. However, we found that this example gives a good insight into the concepts that are involved when using non-determinism reducing refinements.

7.2.6 Atomicity refinement

Since our definition of refinements is based on a bitotal relation R which can relate one action in the original program to several actions in its refinement, our definition of refinement allows for some kind of atomicity relation. However, preserving general program properties, and hence not just total or partial correctness, severely constrains the kind of atomicity refinements that may be applied. Moreover, since atomicity refinement is not going to be used in applications presented later in this thesis, we shall not elaborate too much on this kind of refinement. In the rest of this section we shall present how a simple guard simplification (taken from [Sin93]), that results in a finer grain of atomicity, can be handled within our framework of refinement.

Consider the two programs in Figure 7.8, where S is a finite set, and i does not occur free in A. Evidently, programs P and Q keep executing action A until no element in S satisfies predicate g. Let $p =$ **if** $(\exists i \ : i \in S : g.i)$**then** A and $q.i =$ **if** $g.i$ **then** A. It easy to prove that the relation $\mathcal{R} = \{(p, q.i) \mid i \in S\}$ is bitotal on $\mathbf{a}P$ and $\mathbf{a}Q$, and consequently that for any J, $P \sqsubseteq_{\mathcal{R},J} Q$. To determine the conditions that need to be satisfied in order to conclude property preservation, Theorem $7.2.8_{113}$ can be used to conclude:

$$\frac{\begin{array}{c}\forall i : i \in S : g.i \ \mathcal{C} \ \mathbf{w}Q \\ \forall i : i \in S : \ {}_Q\vdash (J_P \wedge J_Q \wedge g.i) \text{ unless } \neg(\exists i : i \in S : g.i)\end{array}}{((J_P \ {}_P\vdash p \rightarrowtail q) \Rightarrow (J_P \wedge J_Q \ {}_Q\vdash p \rightarrowtail q)) \ \wedge \ ((J_P \ {}_P\vdash p \rightsquigarrow q) \Rightarrow (J_P \wedge J_Q \ {}_Q\vdash p \rightsquigarrow q))}$$

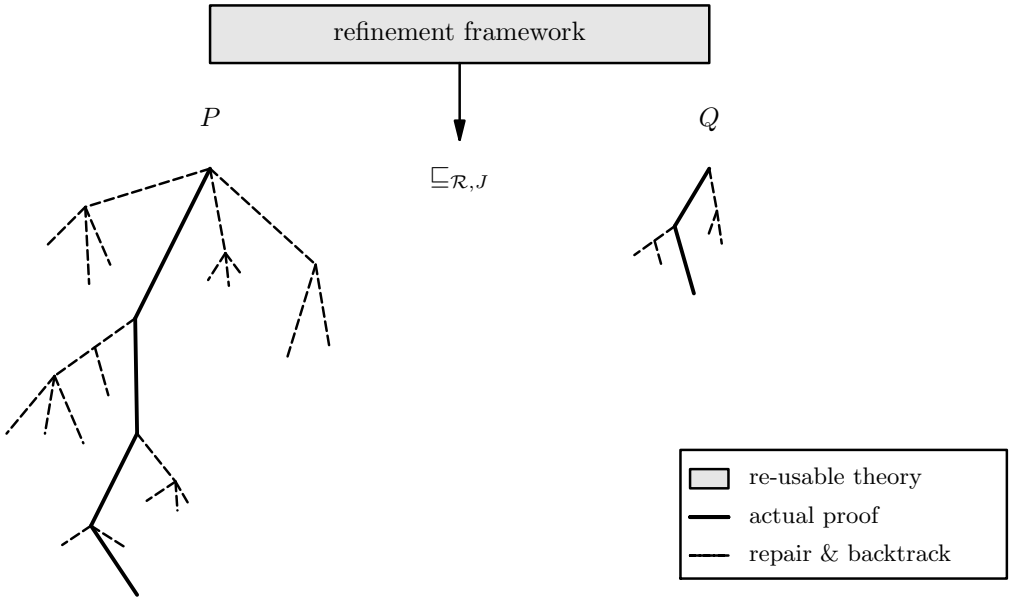


Figure 7.9: Reducing proof-effort and complexity.

for the programs P and Q as displayed in Figure 7.8. These conditions coincide with the ones required in [Sin93].

7.3 Conclusion

We have defined a refinement relation on programs that incorporates (possibly multiple compositions of) program transformations like guard strengthening, superposition, and atomicity refinement. Moreover, we have given theorems that state property preservation in refinements independently from the specific program transformations that were applied. Consequently, we have a general framework of refinements that, besides being suitable for the stepwise derivation of programs, is also efficient for the reduction of proof-effort when proving the correctness of a class of by refinement related algorithms. To pursue the tree pruning analogy from Chapter 6 (Section 6.3_{95}) we refer to Figure 7.9. Evidently, the intuition behind Figure 7.9 is that the use of refinements can shorten the the actual proof of a refinement (i.e. the solid line) since instead of proving the program from scratch we prove the simpler verification conditions of one of the theorems in Figure 7.7. Moreover, the amount of time spent on repairing and backtracking is reduced since having verified P's correctness we have obtained a good feeling about the workings of the algorithms in this particular class, and hence will it be less likely that we proceed on wrong proof-strategies.

Before a wise man ventures into a pit, he lowers a ladder – so he can climb out.

–Mishle (Shelomo)

Chapter 8

The proof of the program is in the representation

Many authors [LT87, LT89, Sto89, Tel89, Sch91, Len93, Cho94a, Haa94, Vaa95, Lyn96] recognise that studying and verifying distributed algorithms is a complex and time-consuming activity. Obviously, one reason for this is that distributed algorithms are inherently more difficult to understand than their sequential counterparts. As often indicated this is, among other, caused by the presence of non-determinism, and the necessity to deal with fairness issues.

Another significant factor, however, of the time spent on studying distributed algorithms, can be attributed to the way distributed algorithms and their correctness proofs are presented. This last aspect is the focus of this chapter, in which we argue that better representation of distributed algorithms significantly influences the ease of reasoning.

The structure of this chapter is as follows. Section 8.1 describes the class of algorithms that are used throughout this chapter as a vehicle for the exemplification of the concepts treated. Section 8.2 formalises the distributed system underlying these algorithms, and Section 8.3 formalises asynchronous communication. Section 8.4 describes existing work that has inspired the contents of this chapter. Section 8.5 enumerates the advantages of better representations of (distributed) algorithms. Section 8.6 formulates guidelines for a better representation of an algorithm. Section 8.7 presents the improved representations of the algorithms discussed in Section 8.1. Section 8.8 shows a new, least-deterministic variant of the algorithms, and Section 8.9 discusses similarities with a related distributed algorithm. Section 8.10 presents various applications of the algorithms. Section 8.11 introduces some notational conventions used in Section 8.12, which discusses a refinement relation identified on the class of algorithms. Section 8.13 presents the approach that is taken in Chapter 9 to prove the correctness of the algorithms in the class of distributed hylomorphisms. Section 8.14 finally concludes.

8.1 Distributed hylomorphisms

The class of algorithms considered in the rest of this thesis consists of what we shall refer to as *distributed hylomorphisms*. The term hylomorphism originated in the Dutch STOP project, and comes from the Greek preposition *hylo*($\acute{\upsilon}\lambda\eta$), meaning "matter", after the Aristotelian philosophy that form (= generated) and matter (= reduced) are the same. A hylomorphism is defined as the composition of an anamorphism and a catamorphism. The term anamorphism comes from the Greek preposition *ana* ($\alpha\nu\alpha$), meaning "upwards", and reflects the fact that an anamorphism builds up (or generates) some kind of structure starting from scratch. In the case of our distributed hylomorphisms, the anamorphism-part corresponds to the construction of a rooted spanning tree in a connected network. The term catamorphism, which has been coined by Lambert Meertens [Mee86], comes from the Greek preposition *cata* ($\kappa\alpha\tau\alpha$) meaning "downwards", and reflects the fact that a catamorphism walks through the structure generated by the anamorphism in order to achieve some goal. In the case of our distributed hylomorphisms this goal can be for example:

- propagation of information with feedback [Seg83];
- finding the ordering of all processes identities in a connected network of processes [Cha82], and based on this performing leader election [Tel94];
- finding the configuration of a network of processes [Cha82] (e.g. global termination detection [DS80, Fra80, Tel89, Tel94]);
- re-synchronisation [Fin79];
- computing a function of which each process holds part of the input (e.g. infimum or summation functions) [Tel94].

Summing up, our distributed hylomorphisms are algorithms that build a rooted spanning tree (RST) in the connected network of processes (i.e. ana-part) and use this RST to let the required information (i.e. the values of which the sum has to be computed, or the feedback of the information that has to be propagated through the network) flow from the leaves to the root of the spanning tree (i.e. cata-part).

$$\underbrace{\text{let the information flow from leaves to root of the RST}}_{\textbf{cata}} \circ \underbrace{\text{build an RST}}_{\textbf{ana}}$$

The two example algorithms used are Tarry's algorithm [Tar95] and Chang's ECHO algorithm [Cha82].

The Tarry algorithm is a traversal algorithm for arbitrary networks given by Tarry [Tar95] in **1895**.

The ECHO algorithm was introduced by Chang [Cha82] as an algorithm for detecting simple network properties. The algorithm does occur, however, albeit in slightly different contexts, under different names in other work. Dijkstra and Scholten [DS80] call ECHO algorithms *diffusing computations*, and in their paper "design a signalling scheme – to be superimposed on the diffusing computation proper – such that, when the diffusing computation proper has thus terminated, the fact of this completion will eventually be signalled back to the environment." In [Tel94] this algorithm is classified as a termination detection algorithm under the name "Dijkstra-Scholten Algorithm". Independently, Francez [Fra80] describes similar work, but refers to ECHO as a *distributed termination algorithm*. Segall [Seg83] names them

PIF algorithms, although **P**ropagation of **I**nformation with **F**eedback is just one possible application of ECHO algorithms. Tel classifies them as *wave algorithms* in [Tel94], and *total algorithms* in [Tel89].

8.2 The formalisation of the distributed system

Since the underlying networks of the distributed hylomorphisms considered in this thesis are connected centralised (or single source) communication networks employing asynchronous communication, these concepts are formalised in this and the following section.

A centralised communication network is modelled by the tuple $((\mathbb{P}, \mathsf{neighs}), \mathsf{starter})$, where

$(\mathbb{P}, \mathsf{neighs})$ is a decentralised communication network (see Definition 6.2.2$_{74}$).

starter is a process in $\mathbb{P}$ that distinguishes itself from all other processes (called the *followers*), in that it can spontaneously start the execution of its local algorithm (e.g. because it is triggered by some internal event). The *followers* can only start execution of their local algorithm after they have received a first message from some neighbour.

To spare the reader from continuously looking up the precise definition of a decentralised network in Section 6.2$_{74}$, we have expanded this definition below.

Definition 8.2.1 CENTRALISED COMMUNICATION NETWORK *Network_DEF*

$$\begin{array}{lcl} \mathsf{Network}.\mathbb{P}.\mathsf{neighs}.\mathsf{starter} & = & \mathsf{FINITE}.\mathbb{P} \wedge \mathsf{card}.\mathbb{P} > 1 \\ & \wedge & \mathsf{starter} \in \mathbb{P} \\ & \wedge & \forall p \in \mathbb{P} : \mathsf{neighs}.p \subseteq \mathbb{P} \\ & \wedge & \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p \ : p \neq q \\ & \wedge & \forall p, q \in \mathbb{P} \ : (q \in \mathsf{neighs}.p) = (p \in \mathsf{neighs}.q) \end{array}$$

◀

A *connected network* is a network in which every pair of processes is connected by a path of communication links. Let us define the set of processes that are reachable from processes in a set W by following at most one communication link:

Definition 8.2.2 ACCUMULATE NEIGHBOURS *Neighs_DEF*

$$\mathsf{Neighs}.\mathsf{neighs}.W = \{q \mid \exists p \ :: \ p \in W \wedge q \in \mathsf{neighs}.p\} \cup W$$

◀

If, for any $p \in \mathbb{P}$, there exists a number n such that the n-fold iterated application of the function Neighs.neighs on $\{p\}$ returns $\mathbb{P}$, then we can conclude that every pair of processes in $\mathbb{P}$ is connected by a path of communication links. Consequently, since $\mathsf{starter} \in \mathbb{P}$, the following is a valid definition of connected networks:

Definition 8.2.3 CONNECTED NETWORK *Connected_Network*

$$\begin{aligned}&\mathsf{Connected\_Network}.\mathbb{P}.\mathsf{neighs.starter}\\ =\ &\mathsf{Network}.\mathbb{P}.\mathsf{neighs.starter}\\ &\wedge\ \exists n\ ::\ \mathbb{P} = \mathsf{iterate}.n.(\mathsf{Neighs.neighs}).\{\mathsf{starter}\}\end{aligned}$$

Since we only consider communication networks that have at least two processes we have the following property of connected networks:

Theorem 8.2.4 *Connected_Network_IMP_EXISTS_neigh*

$$\frac{\mathsf{Connected\_Network}.\mathbb{P}.\mathsf{neighs.starter}\ \wedge\ p \in \mathbb{P}}{\exists q\ ::\ q \in \mathsf{neighs}.p}$$

8.3 Modelling bi-directional asynchronous communication

The type of communication employed in a communication network is assumed to be asynchronous, i.e. send and receive operations work on buffered channels.

To model asynchronous communication, a simple, general and re-usable theory is developed. Each algorithm using asynchronous communication in a communication network Network.$\mathbb{P}$.neighs.starter should have the following variables.

Definition 8.3.1 *ASYNC_Vars*

For functions nr_rec, nr_sent, M $\in \mathbb{P}\rightarrow\mathbb{P}\rightarrow$`Var`:

$$\begin{aligned}&\mathsf{ASYNC\_Vars}.\mathbb{P}.\mathsf{neighs.nr\_rec.nr\_sent.M}\\ =\ &\{\mathsf{nr\_rec}.p.q \mid p \in \mathbb{P}\ \wedge\ q \in \mathsf{neighs}.p\}\\ &\cup\ \{\mathsf{nr\_sent}.p.q \mid p \in \mathbb{P}\ \wedge\ q \in \mathsf{neighs}.p\}\\ &\cup\ \{\mathsf{M}.p.q \mid p \in \mathbb{P}\ \wedge\ q \in \mathsf{neighs}.p\}\end{aligned}$$

- the nr_rec.$p.q$ variables indicate the number of messages p has received from q via directed communication link (q, p).
- the nr_sent.$p.q$ variables indicate the number of messages p has sent to q via directed communication link (p, q).
- the M.$p.q$ variables represent the buffers that store messages in transit from p to q.

Obviously, the types of the nr_rec.$p.q$ and nr_sent.$p.q$ variables should be num, and that of the M.$p.q$ variables list. Consequently, in need of the possibility to let different variables take different types, these variables have *actual type* `Val`, and *intended type*

num, num and list of Val, respectively. Consequently, any algorithm using this communication theory should assume the type declaration of the communication variables stated below. To simplify notations, we shall omit nr_rec, nr_sent and M as parameters in subsequent definitions.

Definition 8.3.2 TYPES OF THE COMMUNICATION VARIABLES *ASYNC_type_decl*

$$\begin{array}{lcll} \mathsf{ASYNC\_type\_decl}.\mathbb{P}.\mathsf{neighs} & = & \forall p, q \in \mathbb{P}, s \in \mathtt{State} :: & \mathsf{is\_num}.(s.(\mathsf{nr\_rec}.p.q)) \\ & & & \mathsf{is\_num}.(s.(\mathsf{nr\_sent}.p.q)) \\ & & & \mathsf{is\_list}.(s.(\mathsf{M}.p.q)) \end{array}$$

◀

and incorporate the following initial condition for these variables:

Definition 8.3.3 INITIALISE THE COMMUNICATION VARIABLES *ASYNC_Init*

$$\begin{array}{lcll} \mathsf{ASYNC\_Init}.\mathbb{P}.\mathsf{neighs}.s & = & \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :: & s.(\mathsf{nr\_rec}.p.q) = \mathsf{NUM}.0 \\ & & & s.(\mathsf{nr\_sent}.p.q) = \mathsf{NUM}.0 \\ & & & s.(\mathsf{M}.p.q) = \mathsf{LIST}.[] \end{array}$$

◀

The theory contains the actions send and receive, and a state-predicate mit. To avoid any confusion, their definitions are stated below without any overloading. For the exact definitions of the state-lifted operators (e.g. `EQ` ,`PUT`, `!+!`) the reader is referred to Figure 3.1_{23}, the state-lifted constants `ONE` and `EMPTY_LIST` can be found in Definition $5.4.3_{61}$.

The action send, that sends a message m from p to q is defined below. The definition is straightforward: the message m is put in the buffer $\mathsf{M}.p.q$, and process p registers sending a message to q by incrementing the variable $\mathsf{nr\_sent}.p.q$.

Definition 8.3.4 SEND A MESSAGE m *send_DEF*

$$\begin{array}{l} \mathsf{send}.p.q.m = \mathsf{ASSIGN}.[\mathsf{M}.p.q\ ,\ \mathsf{nr\_sent}.p.q] \\ \qquad\qquad .[\mathtt{PUT}.m.(\mathsf{VAR}.(\mathsf{M}.p.q))\ ,\ (\mathsf{VAR}.(\mathsf{nr\_sent}.p.q))\,\mathtt{!+!ONE}] \end{array}$$

◀

The state-predicate mit, the name of which is an acronym for message in transit, can be used to check whether there is a message in transit from p to q in some state s.

Definition 8.3.5 CHECK WHETHER THERE IS A MESSAGE IN TRANSIT *mit*

$$\mathsf{mit}.p.q = \mathsf{not}.(\mathsf{VAR}.(\mathsf{M}.p.q)\ \mathtt{EQ}\ \mathtt{EMPTY\_LIST})$$

◀

The receive action contains two subtleties. First, we have defined it in such a way that it only has the desired effect when a message is indeed in transit. So the programmer

has to ensure that this action is only executed after checking and confirming the availability of a message to receive. Second, when receiving a message, the receiver usually does something with the received value and stores the result somewhere. Since we have no sequential composition of actions, we decided to parametrise receive with the appropriate parameters such that all these effects are established simultaneously. In other words, the receive action is parametrised with a function f and a variable v, such that the value of the received message is assigned to variable v after function f has been applied to it. More formally:

Definition 8.3.6 RECEIVE A MESSAGE WHEN THERE IS ONE IN TRANSIT *receive_DEF*

$$\begin{aligned}\mathsf{receive}.p.q.f.v = \mathsf{ASSIGN}.[&\mathsf{M}.q.p\ ,\ \mathsf{nr\_rec}.p.q\ ,\ v]\\ .[&\mathsf{TAIL}.(\mathsf{VAR}.(\mathsf{M}.q.p))\\ ,\ &(\mathsf{VAR}.(\mathsf{nr\_rec}.p.q))\ \texttt{!+!}\ \texttt{ONE}\\ ,\ &f.(\mathsf{HEAD}.(\mathsf{VAR}.(\mathsf{M}.q.p)))]\end{aligned}$$

◀

8.4 Related work

The ECHO algorithm constitutes an interesting case study because it is highly parallel and non-deterministic. As such, several people have already tackled the formal verification of this algorithm in order to illustrate various formal methods. In the rest of this section we describe some of this formal work chronologically. Less formal work on the ECHO algorithm includes [Cha82, Seg83, Sto89, Tel89, Tel94].

Chou [Cho94a] mechanically verifies a simple variant of the ECHO algorithm that only works on trees, and computes the sum (or any other result constructed using a commutative, associative operator that has an identity element) of the values that reside at the nodes of this tree. He admits that by restricting the network to a tree, he has been spared the complexity of an enormous amount of "true nondeterminism", but claims that his method can still be used when arbitrary networks are assumed [Cho94b] (although these results have not been mechanically verified). We are appealed by his approach of using programming logics, which is both eclectic and minimalistic (he uses just whatever he finds useful of TLA and UNITY). The verification method he proposes is what he calls an events-and-causality-based method, in which he combines the approaches of operational and assertional reasoning about distributed algorithms by: (a) first, introducing an *event algorithm* that is an operational representation of the causal patterns of events in the distributed algorithm, (b) second, proving properties of the event algorithm by operationally reasoning about events and causality, (c) showing that the event algorithm can simulate the distributed algorithm, and (d) finally translating the operational properties of the event algorithm into assertional properties of the distributed algorithm using a simulation technique [Cho93]. The underlying idea of his method is to perform as much reasoning as possible in terms of the event algorithm, since he claims this is more abstract and easier than assertionally reasoning about the distributed algorithm.

Chou [Cho95] has a somewhat pessimistic view of using assertional methods based on invariant reasoning:

> *the application of assertional techniques in practice is still an ad hoc, trial-and-error process that needs a great deal of ingenuity. A good example of this trial-and-error process is the construction of invariants, which is a major part of any assertional proof. The invariants one finds in literature often seem to be "pulled out of a hat" with little or no explanation as to how they are invented.*

Although we agree with Chou that *in the literature* indeed this often seems to be the case, we do not go along with the conclusion that this would be inherent to assertional verification methods. We think that the main reasons that in the literature invariants often seem to be "pulled out of a hat" are more practical. Papers and reports are often written *after* one has completely verified an algorithm, and at this point it is usually hard to retrieve the exact line of reasoning one employed during the verification process. One could, however, keep a "research-diary" of the steps one has undertaken while solving the problem. But even then, papers are generally supposed to be structured, have a certain (not too great) length, and be accessible to an as large as possible audience. Obviously, a paper that recaptures the actual verification process shall usually not confine to these conventions.

In response to Chou's pessimistic view of using assertional methods, and in an attempt to try and convince him otherwise, Vaandrager [Vaa95] verifies a variant of the ECHO algorithm using the I/O automata model [LT87, LT89]. Vaandrager's proof starts by specifying the algorithm as an I/O automaton *DSum* using standard precondition/effect notation [LT87, LT89]. Then a simple deterministic two-state I/O automaton S is introduced: one state implying that the algorithm is still busy, and one state implying that the algorithm is finished and the desired sum of all values is residing at the correct place. Subsequently, his correctness criterion consists of proving trace inclusion of *Dsum* in S (i.e. *traces(DSum)* $\subseteq$ *traces(S)*), and deadlock-freeness (i.e. *fairtraces(DSum)* $\subseteq$ *fairtraces(S)*). Trace inclusion is verified as follows: (a) a history relation from *DSum* to an I/O automaton $DSum^h$ is established, (b) then an invariant is "pulled out of a hat" and used to establish a prophecy relation from $DSum^h$ to an I/O automaton $DSum^{hp}$, (c) finally, the existence of a refinement from $DSum^{hp}$ to S is proved, which implies that *traces(DSum)* $\subseteq$ *traces(S)* [LV95]. The verification of deadlock-freeness relies on proof sketches trying to make a reasonable case for the fact that *fairtraces(DSum)* $\subseteq$ *fairtraces(S)* holds. Although we agree with Vaandragers position regarding the suitability of assertional methods, we do not find the setting of his paper very convincing and from the following quotation taken from [Cho95] we can infer that it indeed was not:

> *It should be pointed out that not everyone agrees with our view that the current assertional methods are inadequate in practice. In response to an earlier version of this paper [Cho94b], Vaandrager [Vaa95] argues that the current assertional methods already provide sufficient guidance for the practical verification of distributed algorithms. This difference in opinion can only be settled, to the extent that it can be settled, by comparing the*

proofs produced according to the two approaches. The readers are invited to make such a comparison.

In [Cho95] – which is clearly a response to [Vaa95] – Chou again verifies the same algorithm using his events-and-causality-based method, now combined with a variant of I/O automata. Although we must admit that the proof in [Cho95] is better motivated and more structured than the one in [Vaa95], it is our opinion – and consequently the main issue in this chapter – that the work of Chou as well as that of Vaandrager suffers from having a poor representation of the algorithm under consideration.

As far as we know, two other formal proofs of the same protocol exists.
- In [Hes97] essentially the same protocol is verified as the one in Chou and Vaandrager. Hesselink describes (and represents) the protocol using send, multicast and delay primitives that, when triggered by some data in input queues, atomically put data values in all output queues. He introduces an oracle to model non-determinism, and, slightly worried about the proof sketches in [Vaa95], verifies the correctness of the algorithm using an assertional method based on invariant reasoning in the Boyer-Moore theorem prover [BM88]. Although Hesselink's representation of the algorithm is not what we consider to be a good representation (Section 8.6), Hesselink does make an effort in specifying the basic algorithm independently from the specific application it is going to be used for (i.e. propagation of information without feedback, propagation of information with feedback, and summation).
- In [GS96, GMS97] an algebraic verification of the ECHO protocol is presented using μCRL [GP94], a process algebra which allows processes parametrised with data. The correctness of the algorithm is stated as a process equation, the proof of which consists of a combination of algebraic and assertional techniques and has been mechanically verified using PVS [ORSH95]. The representation of the algorithm using μCRL will be discussed in the next section.

8.5 The representation of distributed algorithms

The representation of a subject of study can significantly influence the time and effort needed; just imagine being forced to keep the accounts using only Roman numerals! Obviously, this also holds for the representation of algorithms. As indicated we claim that a better representation of distributed algorithms can significantly increase the competence to handle their complexity. More specifically, a better representation can:

- reduce the time and effort needed to understand the functionality and properties of the algorithm.
- increase the ability to see similarities with and differences from other algorithms, and learn how to invent and encapsulate new algorithms.
- reduce the proof effort and complexity of correctness proofs, and consequently the time needed to understand and trust the proof.

Surely this all sounds very nice, and few if any would disagree with these arguments. However, the most interesting part is what is meant by *better* representation of algorithms. One objective of this chapter is to give some pointers to *what* we mean by a

var rec_p : integer **init** 0 ; (* counts number of received messages *)
$father_p$: $\mathbb{P}$ **init** *udef*;

For the initiator p:
begin **forall** $q \in$ neighs p **do** send $\langle$**tok**$\rangle$ to q ;
while $rec_p <$ #neighs p **do**
begin receive $\langle$**tok**$\rangle$; $rec_p := rec_p + 1$ **end** ;
end
For non-initiators p:
begin receive $\langle$**tok**$\rangle$ from neighbour q ; $father_p := q$; $rec_p := rec_p + 1$;
forall $q \in$ neighs p, $q \neq father_p$ **do** send $\langle$**tok**$\rangle$ to q ;
while $rec_p <$ #neighs p **do**
begin receive $\langle$**tok**$\rangle$; $rec_p := rec_p + 1$ **end** ;
send $\langle$**tok**$\rangle$ to $father_p$
end

Figure 8.1: Tel's [Tel94] representation of the ECHO algorithm.

var $used_p[q]$: boolean **init** false for each $q \in$ neighs p
(* Indicates whether p has already sent to q *)
$father_p$: $\mathbb{P}$ **init** *udef*;

For the initiator p only, execute once:
begin $father_p := p$; choose $q \in$ neighs p ;
$used_p[q] := true$; send $\langle$**tok**$\rangle$ to q
end
For each process p, upon receipt of $\langle$**tok**$\rangle$ from q_0:
begin **if** $father_p = udef$ **then** $father_p := q_0$;
if $\exists q \in$ neighs $p : (q \neq father_p \wedge \neg used_p[q])$
then begin choose $q \in$ neighs $p \setminus \{father_p\}$
with $\neg used_p[q]$) ;
$used_p[q] := true$; send $\langle$**tok**$\rangle$ to q
end
else begin $used_p[father_p] := true$;
send $\langle$**tok**$\rangle$ to $father_p$
end
end

Figure 8.2: Tel's [Tel94] representation of the TARRY algorithm.

Internal: *MSG*
REPORT
Output: *RESULT*
State Variables: *busy*: **V**→**Bool**
par: **V**→**E**
total: **V**→**M**
cnt: **V**→**Int**
mq: **E**→**M***
Init: $\wedge \forall v : \neg busy[v]$
$\wedge \forall e : mq[e] = \text{if } e{=}\mathsf{e}_0 \text{ then append(0,empty) else empty}$

MSG(e : **E**, m : **M**)
Precondition:
$v = \mathsf{target}(e) \wedge m = \mathsf{head}(mq[e])$
Effect:
$mq[e] := \mathsf{tail}(mq[e])$

if $\neg busy[v]$ **then** $busy[v] := \mathsf{true}$
$par[v] := e$
$total[v] := \mathsf{weight}(v)$
$cnt[v] := \mathsf{size}(\mathsf{to}(v))$ - 1
for $f \in \mathsf{from}(v)$ / $\{e^{-1}\}$ **do** $mq[f] := \mathsf{append}(0,mq[f])$
else $total[v] := total[v] + m$
$cnt[v] := cnt[v]$ - 1

REPORT(e : **E**, m : **M**)
Precondition:
$v = \mathsf{source}(e) \neq \mathsf{v}_0 \wedge busy[v] \wedge cnt[v] = 0 \wedge e^{-1} = par[v] \wedge m = total[v]$
Effect:
$busy[v] := \mathsf{false}$
$mq[e] := \mathsf{append}(m,mq[e])$

RESULT(m : **M**)
Precondition:
$busy[\mathsf{v}_0] \wedge cnt[\mathsf{v}_0] = 0 \wedge m = total[\mathsf{v}_0]$
Effect:
$busy[\mathsf{v}_0] := \mathsf{false}$

Figure 8.3: Vaandrager's [Vaa95] representation of ECHO as an I/O automaton (*DSum*)

better representation, and *how* it can be obtained. Before we continue two remarks are in order.

First, *better* representation is a subjective matter. It depends on the upbringing, foreknowledge and taste of the people involved, and it depends on the formalisms used. For the purpose of illustration, Tel's[1] [2] [Tel94] representations of ECHO and TARRY can be found in Figures 8.1 and 8.2 respectively. Throughout [Tel94], which provides a very good, structured and exhaustive overview of all kinds of distributed algorithms, a Pascal-like pseudo-code is used to represent the algorithms. Pascal-like notation is a defensible choice since, because most computer scientists are familiar with Pascal, it makes the book accessible to a wide audience. Consider the representation of ECHO in Figure 8.1. After some inspection it is not too hard to understand the global workings of the algorithm. Although it is our opinion that, in order to understand the algorithm, more time and effort must be spent than necessary, and although we are not elated by this representation, it can be defended by referring to the taste, upbringing and foreknowledge of the writer and his expected readers. Now, observe TARRY in Figure 8.2. As will be shown later, ECHO and TARRY have lots of similarities, which is by no means clear from comparing the two algorithms in Figures 8.1 and 8.2: they employ different variables, TARRY has lots of indentation as a consequence of nested if-then-else constructs, and the local algorithms of the followers have a completely different structure.

The second remark we must make is that by better representation we do *not* claim that more formality is the key. To illustrate this we refer to Figures 8.3. 8.4 and 8.5 that display the before mentioned representations of the ECHO algorithm copied from [Vaa95], [Hes97], and [GS96, GMS97] respectively. The representation in [Cho95] is similar to the one in [Vaa95]. Although the I/O-automata model, Boyer-Moore theorem prover, and μCRL are formal methods, we, perhaps due to differences in upbringing and taste, find it very time-consuming to understand the workings of the algorithms thus represented. For example, from these representations, it is not clear at all that messages are being sent and received. For one to figure this out, one has to dive into the accompanying textual explanation. In the next section we shall list some characteristics of what we consider to be a good representation of an algorithm.

8.6 Better representation of algorithms: *What*

This section describes four characteristics of what we think to be a good representation of an algorithm.

(1) A good representation of an algorithm reveals similarities with other similar algorithms. Revelation of similar behaviour from the representation of algorithms:

   (a) creates opportunities to recognise classes of distributed algorithms, and consequently reduces the time spent on studying separate algorithms in

[1] Small alterations have been made, namely the *decide* events have been omitted.
[2] *udef* means undefined.

```
accept (signal, j, u) =
   explist := explist \{j}
   expcnt := expcnt - 1
   value := value + u
 if   parent = self  then
      parent := j
      delay (sendrep)
      mcast (Nhb.self \{j}, signal, self, 0)
 fi
end .
accept (sendrep)
   enabling expcnt = 0
:   send (parent, signal, self, value) ;
   value := 0
end .
```

Figure 8.4: Hesselink's representation of ECHO from [Hes97]

act $st, \overline{st}, st^* : \mathbb{N} \times \mathbb{N}$ (parameters: destination, source)
$ans, \overline{ans}, ans^* : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ (parameters: destination, source, value)
$\overline{rep} : \mathbb{N}$ (parameter: value)

comm $st \mid \overline{st} = st^*$
$ans \mid \overline{ans} = ans^*$

Definition (*Processes*)

$$
\begin{aligned}
&P(i,t:\mathbb{N}, N:nSet, p, w:\mathbb{N}, s:\mathbb{N}) = \\
&\quad [s=0] \Rightarrow \sum_{j:\mathbb{N}} st(i,j)P(i,t,rem(j,N),j,size(N)-1,1)+ \\
&\quad \sum_{j:\mathbb{N}} [j \in N \wedge s=1] \Rightarrow \overline{st}(j,i)P(i,t,rem(j,N),p,w,s)+ \\
&\quad \sum_{j,m:\mathbb{N}} [s=1] \Rightarrow ans(i,j,m)P(i,t+m,N,p,w-1,s)+ \\
&\quad \sum_{j:\mathbb{N}} [s=1] \Rightarrow st(i,j)P(i,t,N,p,w-1,s)+ \\
&\quad [i=0 \wedge N=\emptyset \wedge w=0 \wedge s=1] \Rightarrow \overline{rep}(t)P(i,t,N,p,w,2)+ \\
&\quad [i\neq 0 \wedge N=\emptyset \wedge w=0 \wedge s=1] \Rightarrow \overline{ans}(p,i,t)P(i,t,N,p,w,2)
\end{aligned}
$$

Figure 8.5: The representation of ECHO in μCRL from [GS96, GMS97]

these classes. (In our case this resulted in the class of distributed hylomorphisms, see Section 8.1.)

(b) reduces proof effort and complexity. Firstly, re-usable theory about similar behaviour can be constructed and properties about these can be proved first. That is, in the context of Figures 6.7_{96} and 7.9_{118}, more and bigger rectangles. Consequently, correctness proofs of the algorithms can become shorter and more elegant since the theory developed can be re-used in the proofs of the algorithms. Secondly, similar behaviour can be proved by similar proofs and proof-strategies. Consequently, proofs and proof-strategies can be re-used, resulting in reduced complexity of correctness proofs for similar algorithms. Thirdly, based on the similarities one can construct a relation (e.g. refinement) between the algorithms and use this relation to create theorems which give information about which properties, under certain conditions, are shared between related algorithms.

(c) creates opportunities for inventing new algorithms. As we will show later (Section 8.8), we have discovered a new algorithm which is a generalisation of both ECHO and TARRY, and the discovery of which almost naturally followed from our representations of the algorithms mentioned before.

(2) From a good representation of an algorithm one can immediately deduce some properties of the algorithm (e.g. invariants, communication strategies). Not only does this make them easier to understand, it also, again, reduces the complexity of verification.

(3) A good representation of an algorithm avoids too many details which are non-essential to, and hence impede with the understandability of the global workings and strategies of the algorithm. One should try to find the minimal amount of details necessary for capturing the main strategy of the algorithm, and abstract from specific applications as much as possible.

(4) A good representation of an algorithm hides formalities in such a way that the reader is gradually introduced to the different behaviours and strategies of the algorithm. This increases pedagogical effectiveness.

8.7 Better representation of TARRY and ECHO: *How*

To illustrate the guidelines given in the previous section and give some pointers on *how* to obtain a representation which is in compliance with these guidelines, we shall describe how we came to better representations of ECHO and TARRY.

8.7.1 Analysing distributed algorithms

We started by analysing the algorithms. Analysing these distributed algorithms was *not* a straightforward process. It consisted of playing with these algorithms, simulating small executions on paper, finding invariants and other properties, detecting similarities, and rewriting them in suitable forms. It is our opinion that this phase is the most important when one is concerned with finding a good representation for an algorithm. Furthermore, we found that it is important to try and find the least

prog Skeleton of ECHO and TARRY

init $(\forall p \in \mathbb{P} :\ (p = \mathit{starter}) \neq (\mathsf{idle}.p))$
$\wedge\ (\mathsf{father}.\mathit{starter} = \mathit{starter})$
$\wedge\ \mathsf{ASYNC\_Init}.\mathbb{P}.\mathsf{neighs}$
$\wedge\ \mathit{init}_{\Pi}$

assign

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$ (IDLE)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \mathit{collecting}_{\Pi}.p$ (COL)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathit{can\_propagate}.p.q \wedge \mathit{propagating}_{\Pi}.p$ (PROP)
then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p$ (DONE)
then (**if** $(q = (\mathsf{father}.p))$ **then** $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle$)

Figure 8.6: The skeleton of the local algorithm of process $p \in \mathbb{P}$ for distributed hylomorphisms $\Pi \in \{\text{ECHO, TARRY}\}$.

deterministic variant of the algorithms by trying to find the determinism that can be eliminated without changing the functionality of the algorithm. During these activities, one learns much about the algorithms and gets a good feeling about their structure and functionality, which leads to finding similarities between algorithms and consequently a better representation.

The rest of this subsection describes the global workings of the two algorithms in terms of similarities and differences.

Similar behaviour of TARRY and ECHO. Initially every *follower* is *idle*, the *starter* is non-*idle* and all communication channels are empty. A *follower* becomes active when it receives its very first message, and it marks the process from which it received this first message as its *father*. A non-*idle* process proceeds with two activities. The first being *propagation*, i.e. sending a message to all its neighbours except its father. The second being *collecting* one message from each of its neighbours. When the

starter has sent and received one message to and from all its neighbours (i.e. has completed *propagating* and *collecting*), it immediately is *done*, whereas a *follower* process p has completed *propagating* and *collecting* it first has to report to its father prior to becoming *done*.
The **differences between** TARRY **and** ECHO are in the communication protocols, more specifically when non-*idle* processes are allowed to collect or propagate a message. In the TARRY algorithm, a non-*idle* process p can only propagate to a neighbour if the last event of p was a receive event; otherwise it has to wait until it receives something. So, in TARRY, the *propagating* and *collecting* activities strictly alternate, and as a consequence there always will be at most one message in transit. In the ECHO algorithm, a non-*idle* process p can only receive a message after p has sent messages to all its neighbours except its father. So, the *propagating* activities must be completed before *collecting* information from non-father-neighbours.

8.7.2 Construct UNITY programs

Based on the similarities described in Section 8.7.1, we were able to construct a skeleton for all local algorithms (including the one for the *starter*) which encompasses TARRY and ECHO. The skeleton is displayed in Figure 8.6, using UNITY notation (the **read** and **write** sections have been omitted for readability). One can see that the skeleton complies with the guidelines given earlier:

- By hiding formalities about the exact characterisation of the predicates that constitute the program's guards, and choosing suitable names for them, a reader can obtain a good feeling about the global workings of the algorithm *before* he or she is introduced into the precise formalisations of these predicates. Moreover, this gradual introduction into the algorithms shall help them to understand the exact formalisation of these predicates better and faster.
- By implicitly subsuming the communication with all neighbours within the syntax (i.e. $[]_{q \in \mathsf{neighs}\, p}$), rather than stating it explicitly within the representation (i.e. **forall** ... **do**), the algorithm stays surveyable.
- The skeleton almost immediately reveals the similar behaviour of the algorithms as informally described at the end of subsection 8.7.1. Furthermore, parameterising those predicates that differ among the algorithms with the name of the algorithm, immediately shows where the differences are.
- Some properties of both algorithms can easily be extracted from the representation. For example, after some inspection one can see that the algorithms build some kind of spanning tree in the network, and that in both algorithms ($\neg \mathsf{idle}.p$) is a stable property for all processes p.
- Following [Tel94], the skeleton abstracts from specific applications of the algorithms, by leaving the contents of the messages that are being sent, and the ways these are processed upon receipt, unspecified. More specifically, we represent the receipt of a message by:

  $$\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \tag{8.7.1}$$

  instead of

  $$\mathsf{receive}.p.q.h.(V.p) \tag{8.7.2}$$

With respect to Definition $8.3.6_{124}$, using (8.7.2) would be more precise and formal. A disadvantage, however, of using (8.7.2) is that the reader can be distracted by the parameters h and $(V.p)$, and, in order to find out what their purpose is, has to look up the definition of receive or read the accompanying text. Although less formal and precise, (8.7.1) is much more intuitive, and prevents the reader from being distracted by unnecessary details, while he or she – upon first introduction to this algorithm – is trying to understand the global workings of the algorithm instead of its applications.

Another advantage of abstracting from specific applications of the algorithms is that the reader can develop his or her own feeling about possible uses of the algorithms. This will help the reader to understand the explanation of existing applications, and can result in ingenious new ones.

Some technicalities of the skeleton

Some design choices had to be made while developing the skeleton from Figure 8.6. First, we decided to introduce a variable idle for each process. Consequently, by assuming idle to be initially false for the *starter* and true for the followers, we eliminated the need to distinguish between the local algorithms of the *starter* and the *followers*.

Second, we added the initial condition $\mathsf{father}.\mathit{starter} = \mathit{starter}$. The reason for this will be explained in the next chapter, where the formal proofs of the algorithms are described.

Finally, the DONE actions of the algorithm, which are actually one action for each process, are modelled by a set of actions:

$$[]_{q \in \mathsf{neighs}\, p} \quad \textbf{if } \mathit{finished\_collecting\_and\_propagating.p} \wedge \neg\, \mathit{reported\_to\_father.p} \\ \textbf{then } (\textbf{if } (q = (\mathsf{father}.p)) \textbf{ then } \mathsf{send}.p.q.\langle \textbf{mes} \rangle)$$

instead of just by

$$\textbf{if } \mathit{finished\_collecting\_and\_propagating.p} \wedge \neg\, \mathit{reported\_to\_father.p} \\ \textbf{then } \mathsf{send}.p.(\mathsf{father}.p).\langle \textbf{mes} \rangle$$

The reason for this is that a program that includes this last action cannot be proved to be well-formed according to Definition $4.3.1_{43}$. Proving well-formedness of a UNITY program includes proving the *syntactic* requirement that actions should only write to the declared write variables. In the case of the action above this results in the proof-obligation that $\mathsf{nr\_sent}.p.(\mathsf{father}.p)$ is a write variable, which only holds when $p \in \mathbb{P}$ and $\mathsf{father}.p \in \mathsf{neighs}.p$. Although $\mathsf{father}.p \in \mathsf{neighs}.p$ shall hold during any execution of the program (i.e. it is a *semantic* property) we cannot prove this at a syntactical level. Consequently, we are forced to model the DONE actions for each process as a set of actions of which only one eventually has an enabled guard.

The last technicality raises an important point. Although almost any programming notation has shortcomings which can prevent one from constructing certain nice representations, this is no reason to give up on good representations altogether. One just has to try to work around these shortcomings. To illustrate this, and to stress that we do not promote UNITY as the best notation to use, we refer to Figure 8.7 which depicts a UNITY representation of the ECHO algorithm that does not have any of the characteristics listed in Section 8.6.

if idle.$p \wedge \exists q \in$ neighs.p : M.$q.p \neq []$
then (λq. father.p, idle.p, nr_rec.$p.q$, $\langle$**mes**$\rangle$, M.$q.p$
:= q, false, nr_rec.$p.q + 1$, hd.(M.$q.p$), tl.(M.$q.p$)
) (εq. $q \in$ neighs.$p \wedge$ (M.$q.p \neq []$))
[]

if $\neg$ idle.$p \wedge \exists q \in$ neighs.p : M.$q.p \neq []$
$\wedge$ ($\forall q \in$ neighs.p : (($q \neq$ father.p) $\Rightarrow$ (nr_sent.$p.q = 1$)))
then (λq. nr_rec.$p.q$,$\langle$**mes**$\rangle$, M.$q.p$:= nr_rec.$p.q + 1$, hd.(M.$q.p$), tl.(M.$q.p$)
) (εq. $q \in$ neighs.$p \wedge$ (M.$q.p \neq []$))
[]

if $\neg$ idle.$p \wedge \exists q \in$ neighs.p : $q \neq$ father.$p \wedge$ nr_sent.$p.q = 0$
then (λq. nr_sent.$p.q$, M.$p.q$:= nr_sent.$p.q + 1$, $\langle$**mes**$\rangle$: M.$p.q$
) (εq. $q \in$ neighs.$p \wedge q \neq$ father.$p \wedge$ nr_sent.$p.q = 0$)
[]

if $\neg$ idle.$p \wedge$ nr_sent.p.(father.p) $= 0 \wedge$ ($\forall q \in$ neighs.p : nr_rec.$q.p = 1$)
$\wedge$ (father.$p \in$ neighs.p) $\wedge$ ($\forall q \in$ neighs p : $q \neq$ father $p \Rightarrow$ nr_sent.$p.q = 1$)
then nr_sent.p.(father.p), M.p.(father.p)
:= nr_sent.p.(father.p) $+ 1$, $\langle$**mes**$\rangle$: M.p.(father.p)

Figure 8.7: A bad representation of the ECHO algorithm

Capturing the similarities

Besides the similarities revealed by the skeleton in Figure 8.6$_{132}$, the analysis also teaches us that:

- when a process is *collecting* this implies that it has not yet received messages from all its neighbours;
- when a process is *propagating* this implies that it has not yet sent to all its neighbours that are not its father;
- p *can propagate to* q when p has not yet sent to q, and q is not its father;
- when a process is *finished_collecting_and_propagating*, then it has received from all its neighbours and it has sent to all its non-father-neighbours;
- when a process has not yet reported to its father, it has not yet sent a message to its father;
- when a process p is *done* it has sent and received a message to and from all of its neighbours (i.e. including its father).

To capture these similarities we introduce the following state-predicates[3]: (Definitions 8.7.1$_{136}$ through 8.7.7$_{136}$):

[3]Note the overloading, see Table 3.2$_{27}$.

Definition 8.7.1 RECEIVED FROM ALL NEIGHBOURS *rec_from_all_neighs*
$rec\_from\_all\_neighs.p = \forall q \in \mathsf{neighs}.p : \mathsf{nr\_rec}.p.q = 1$

Definition 8.7.2 SENT TO ALL NON-FATHER-NEIGHBOURS *sent_to_all_except_f*
$sent\_to\_all\_non\_fathers.p = \forall q \in \mathsf{neighs}.p : (q \neq \mathsf{father}\ p) \Rightarrow (\mathsf{nr\_sent}.p.q = 1)$

Definition 8.7.3 p CAN PROPAGATE TO q *cp*
$can\_propagate.p.q = (\mathsf{nr\_sent}.p.q = 0) \wedge (q \neq \mathsf{father}.p)$

Definition 8.7.4 *finished_collecting_and_propagating*
$finished\_collecting\_and\_propagating.p$
$= rec\_from\_all\_neighs.p \wedge sent\_to\_all\_non\_fathers.p$

Definition 8.7.5 REPORTED TO FATHER *reported_to_f*
$reported\_to\_father.p = (\mathsf{nr\_sent}.p.(\mathsf{father}.p) = 1)$

Definition 8.7.6 SENT TO ALL NEIGHBOURS *sent_to_all_neighs*
$sent\_to\_all\_neighs.p = \forall q \in \mathsf{neighs}.p : \mathsf{nr\_sent}.p.q = 1$

Definition 8.7.7 *done*
$done.p = rec\_from\_all\_neighs.p \wedge sent\_to\_all\_neighs.p$

◀

Note that for the specific implementations of ECHO and TARRY, the $\mathsf{nr\_rec}.p.q$ variables for process p could have been replaced by one variable $\mathsf{nr\_rec}.p$ (analogous to Tel's rec_p) which increments each time a message is received. The disadvantages are twofold.

First, the *rec_from_all_neighs* predicate has to be changed to $\mathsf{nr\_rec}.p = \#\mathsf{neighs}\ p$, which is not as informative as the characterisation given above in that the latter, besides stating that #neighs messages have been received, also states that exactly one message has been received from each neighbour.

Second, the communication theory would not be as re-usable as it is now, since it would not be possible to use it for algorithms that need to keep track of the source of an incoming message.

Handling the differences

As mentioned in section 8.7.1, TARRY and ECHO differ in when non-*idle* processes are allowed to collect or propagate a message, or, in other words, in the characterisation of the predicates *collecting* and *propagating*

In the ECHO algorithm, a non-*idle* process p can only receive a message, after p has sent messages to all its non-father-neighbours. So, the *propagating* activities must be completed before starting *collecting* from non-father-neighbours. Consequently:

prog ECHO

init $(\forall p \in \mathbb{P} :\ (p = \mathit{starter}) \neq (\mathsf{idle}.p))$
$\wedge\ (\mathsf{father}.\mathit{starter} = \mathit{starter})$
$\wedge\ \mathsf{ASYNC\_Init}.\mathbb{P}.\mathsf{neighs}$

assign

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$ (IDLE)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \mathit{collecting}_{\mathrm{ECHO}}.p$ (COL)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathit{can\_propagate}.p.q \wedge \mathit{propagating}_{\mathrm{ECHO}}.p$ (PROP)
then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p$ (DONE)
then (**if** $(q = (\mathsf{father}.p))$ **then** $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle$)

Figure 8.8: The local algorithm of process $p \in \mathbb{P}$ of the ECHO algorithm.

Definition 8.7.8 *propagating_ECHO*
$\mathit{propagating}_{\mathrm{ECHO}}.p = \neg\ \mathit{sent\_to\_all\_non\_fathers}.p$

Definition 8.7.9 *collecting_ECHO*
$\mathit{collecting}_{\mathrm{ECHO}}.p = \neg\ \mathit{rec\_from\_all\_neighs}.p \wedge \neg\ \mathit{propagating}_{\mathrm{ECHO}}.p$

Instantiating Π with ECHO in Figure 8.6 specifies the whole algorithm. For ease of reference the resulting ECHO algorithm is given in Figure 8.8

In the TARRY algorithm, a non-*idle* process p can only propagate to a neighbour if the last event of p was a receive event; otherwise it has to wait until it receives something. So, the *propagating* and *collecting* activities alternate. In order to represent this, the skeleton in Figure 8.6 needs some minor adjustments (see Figure 8.9). We introduce a new boolean-typed variable $\mathsf{le\_rec}\ p$ (i.e. last event was a receive) for every process p, which we initialise to false for all processes except the *starter*. Fur-

prog TARRY

init $(\forall p \in \mathbb{P} :\ (p = \mathit{starter}) \neq (\mathsf{idle}.p))$
$\wedge\ (\mathsf{father}.\mathit{starter} = \mathit{starter})$
$\wedge\ \mathsf{ASYNC\_Init}.\mathbb{P}.\mathsf{neighs}$
$\wedge\ \boxed{\forall p \in \mathbb{P} : (p = \mathit{starter}) \neq (\neg \mathsf{le\_rec}.p)}$

assign

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$ (IDLE)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false}$
$\parallel \boxed{\mathsf{le\_rec}.p := \mathsf{true}}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \boxed{\mathit{collecting}_{\text{TARRY}}.p}$ (COL)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \boxed{\mathsf{le\_rec}.p := \mathsf{true}}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathit{can\_propagate}.p.q \wedge \boxed{\mathit{propagating}_{\text{TARRY}}.p}$ (PROP)
then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle \parallel \boxed{\mathsf{le\_rec}.p := \mathsf{false}}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p$ (DONE)
then (**if** $(q = (\mathsf{father}.p))$ **then** $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle \parallel \boxed{\mathsf{le\_rec}.p := \mathsf{false}}$)

Figure 8.9: The local algorithm of process $p \in \mathbb{P}$ of the TARRY algorithm.

thermore, we add the assignments ($\mathsf{le\_rec}.p := \mathsf{true}$) and ($\mathsf{le\_rec}.p := \mathsf{false}$) to the **then** clauses of (COL) and (PROP) respectively. Consequently, for non-*idle* processes p, the value of $\mathsf{le\_rec}\ p$ indicates whether the last event of p was a receive event. Finally, we characterise the *collecting* and *propagating* predicates as follows:

Definition 8.7.10 *propagating_Tarry*
$\mathit{propagating}_{\text{TARRY}}.p = \neg\ \mathit{sent\_to\_all\_non\_fathers}.p \wedge (\mathsf{le\_rec}.p)$

Definition 8.7.11 *collecting_Tarry*
$\mathit{collecting}_{\text{TARRY}}.p = \neg\ \mathit{rec\_from\_all\_neighs}.p \wedge \neg(\mathsf{le\_rec}.p)$

prog PLUM

init $(\forall p \in \mathbb{P} :\ (p = \mathit{starter}) \neq (\mathsf{idle}.p))$
$\wedge\ (\mathsf{father}.\mathit{starter} = \mathit{starter})$
$\wedge\ \mathsf{ASYNC\_Init}.\mathbb{P}.\mathsf{neighs}$

assign

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$ (IDLE)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \boxed{\mathit{collecting}_{\text{PLUM}}.p}$ (COL)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathit{can\_propagate}.p.q \wedge \boxed{\mathit{propagating}_{\text{PLUM}}.p}$ (PROP)
then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p$ (DONE)
then (**if** $(q = (\mathsf{father}.p))$ **then** $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle$)

Figure 8.10: The local algorithm of process $p \in \mathbb{P}$ of the PLUM algorithm.

Figure 8.9 depicts our representation of TARRY. As the reader can see, similarities between ECHO and TARRY are immediately recognisable upon looking at the algorithms.

8.8 A least deterministic version: PLUM

From our representations of ECHO and TARRY we almost naturally came up with a less deterministic variant of the algorithms, which is also a distributed hylomorphism. The new algorithm, which we have named the PLUM[4] algorithm, allows a process to freely merge its propagating and collecting actions as long as it has not yet received messages from all its neighbours, and it has not yet sent to all its neighbours that are

[4]This name originated at the Marktoberdorf Summerschool 1996 over several glasses of PLUM wine.

not its father. The characterisation of the *propagating* and *collecting* predicates for this specific algorithm are:

Definition 8.8.1 *propagating_PLUM*
$propagating_{\text{PLUM}}.p = \neg\ sent\_to\_all\_non\_fathers.p$

Definition 8.8.2 *collecting_PLUM*
$collecting_{\text{PLUM}}.p = \neg\ rec\_from\_all\_neighs.p$

◀

Substituting Π for PLUM in Figure 8.6, together with these characterisations, constitutes the PLUM algorithm. For ease of reference the whole algorithm is displayed in Figure 8.10.

8.9 Similarities with other algorithms: DFS

Another distributed hylomorphism that has a close relationship with TARRY, is the classical Depth First Search (DFS) algorithm [Che83, Tel94]. The characterisation of the *propagating* and *collecting* predicates for the DFS algorithm are identical to those of TARRY.:

Definition 8.9.1 *propagating_DFS*
$propagating_{\text{DFS}}.p = propagating_{\text{TARRY}}.p$

Definition 8.9.2 *collecting_DFS*
$collecting_{\text{DFS}}.p = collecting_{\text{TARRY}}.p$

◀

The difference with TARRY is in the lesser freedom to choose a neighbour to send a message to in the propagating phase. More specifically, for a non-idle process p in its propagating phase (i.e. there are still non-father-neighbours to which p has not yet sent) whose last event was receiving a message from some neighbour q:

- if p can propagate a message back to q, i.e. q is not p's father, and p has not yet sent to q, then p has to send a message back to this process q
- otherwise it can act like in TARRY, and just pick any non-father-neighbour to which it has not yet sent a message (i.e. to which it can propagate)

In order to be able to formalise and check these conditions each process in the DFS algorithm, needs to remember the identity of the sender of its last incoming message. In order to make this possible, some minor adjustments have to be made to the local algorithms from Figure 8.9. First, we introduce a new variable $\mathsf{lp\_rec}.p$ (last process of which p has received a message) for every process p. Second, we add the assignment ($\mathsf{lp\_rec}.p := q$) to the **then** clauses of (IDLE) and (COL). Finally, we split up the propagating phase of all processes p, into

- (PROP_LP_REC): propagating messages to the process p has received its last message from if this is allowed, (i.e. $can\_propagate.p.q \wedge q = \mathsf{lp\_rec}.p$). (Note

prog DFS

init $(\forall p \in \mathbb{P} :\ (p = \mathit{starter}) \neq (\mathsf{idle}.p))$
$\wedge\ (\mathsf{father}.\mathit{starter} = \mathit{starter})$
$\wedge\ \mathsf{ASYNC\_Init}.\mathbb{P}.\mathsf{neighs}$
$\wedge\ \forall p \in \mathbb{P} : (p = \mathit{starter}) \neq (\neg \mathsf{le\_rec}.p)$

assign

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$ (IDLE)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{father}.p{:=}q \parallel \mathsf{idle}.p{:=}\mathsf{false}$
$\parallel \mathsf{le\_rec}.p{:=}\mathsf{true} \parallel \boxed{\mathsf{lp\_rec}.p{:=}q}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \boxed{\mathit{collecting}_{\mathrm{DFS}}.p}$ (COL)
then $\mathsf{receive}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{le\_rec}.p{:=}\mathsf{true} \parallel \boxed{\mathsf{lp\_rec}.p{:=}q}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathit{cp}.p.q \wedge \boxed{\mathit{propagating}_{\mathrm{DFS}}.p} \wedge \boxed{q = \mathsf{lp\_rec}.p}$ (PROP_LP_REC)
then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{le\_rec}.p{:=}\mathsf{false}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathit{cp}.p.q \wedge \boxed{\mathit{propagating}_{\mathrm{DFS}}.p} \wedge \boxed{\neg(\mathit{cp}.p.(\mathsf{lp\_rec}.p))}$ (PROP_NOT_LP_REC)
then $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{le\_rec}.p{:=}\mathsf{false}$
[]

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p$ (DONE)
then (**if** $(q = (\mathsf{father}.p))$ **then** $\mathsf{send}.p.q.\langle \mathbf{mes} \rangle \parallel \mathsf{le\_rec}.p{:=}\mathsf{false}$)

(**NB**: *can_propagate* is abbreviated by *cp*)

Figure 8.11: The local algorithm of process $p \in \mathbb{P}$ of the DFS algorithm.

that similar to the (DONE)-actions this cannot be modelled by one action, but has to be modelled by a set of actions.)

- (PROP_NOT_LP_REC): propagating messages to an arbitrary neighbour q for which it holds that *can_propagate.p.q*, if it is not allowed to send a message to lp_rec.p (i.e. $\neg$(*can_propagate.p.*(lp_rec.p))).

Figure 8.11 depicts our representation of DFS. Again, differences and similarities with ECHO, TARRY, and PLUM are immediately visible. Moreover, DFS can be seen as the most deterministic algorithm of the four algorithms mentioned in this chapter, since almost all non-determinism has been eliminated.

8.10 Applications of distributed hylomorphisms

As indicated, distributed hylomorphisms can be used for various applications like:
- propagation of information with feedback,
- and the computation of summation functions of which each process in the network holds part of the input.

Precise discussion of these applications, however, does not allow us to leave the contents of the messages that are being sent, and the ways these are processed upon receipt, unspecified. Consequently, we have to make the representation of our algorithms more precise. The most general way to do this, is by parametrising the algorithms in such a way that specific applications can be defined as *instantiations* of the underlying algorithm. Then we are still able to abstract from specific applications of the algorithm by universal quantification over its arguments. From the discussion in Section 8.7.2 we can deduce that, in order to make the algorithms suitable for the characterisation of specific applications, they will have to be parametrised by:

- a state-expression iA, which can be used to specify an additional application specific initial condition
- a function $h \in \mathbb{P} \rightarrow$ `Expr` $\rightarrow$ `Expr`, that, given a process p, specifies how p should handle messages upon receipt
- a function PROP_mes $\in \mathbb{P} \rightarrow$ `Expr`, that, given a process p, specifies what message p should send in its propagating phase
- a function DONE_mes $\in \mathbb{P} \rightarrow$ `Expr`, that, given a process p, specifies which message p finally has to be sent to its father

Moreover, additional variables ($\{\mathsf{V}.p \mid p \in \mathbb{P}\}$) are needed to store the results of receiving and processing messages. Figure 8.12 shows the result for the PLUM algorithm; analogous adjustments have to be made for the ECHO, TARRY, and DFS algorithm.

The subsections below discuss the above mentioned applications of distributed hylomorphisms for a connected communication network, and all implicitly assume the validity of Connected_Network.$\mathbb{P}$.neighs.starter.

8.10.1 Termination of distributed hylomorphisms

Termination of distributed hylomorphisms means that when the algorithm is started in the initial state, eventually each process will reach the situation in which it neither sends nor receives any more messages, and all communication channels will be empty.

PLUM.$iA.h.$PROP_mes.DONE_mes

$=$

prog PLUM

init $(\forall p \in \mathbb{P} :\ (p = starter) \neq (\mathsf{idle}.p))$
$\wedge\ (\mathsf{father}.starter = starter)$
$\wedge\ \mathsf{ASYNC\_Init}.\mathbb{P}.\mathsf{neighs}$
$\wedge\ \boxed{iA}$

read $\mathsf{ASYNC\_Vars}.\mathbb{P}.\mathsf{neighs} \cup \{\mathsf{idle}.p \mid p \in \mathbb{P}\} \cup \{\mathsf{father}.p \mid p \in \mathbb{P}\} \cup \{\mathsf{V}.p \mid p \in \mathbb{P}\}$

write $\mathsf{ASYNC\_Vars}.\mathbb{P}.\mathsf{neighs} \cup \{\mathsf{idle}.p \mid p \in \mathbb{P}\} \cup \{\mathsf{father}.p \mid p \in \mathbb{P}\} \cup \{\mathsf{V}.p \mid p \in \mathbb{P}\}$

assign

$[]_{p \in \mathbb{P}}$

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$ (IDLE)
then $\mathsf{receive}.p.q.\boxed{(h.p).(V.p)} \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false}$

$[]$

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge collecting_{\mathrm{PLUM}}.p$ (COL)
then $\mathsf{receive}.p.q.\boxed{(h.p).(V.p)}$

$[]$

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge can\_propagate.p.q \wedge propagating_{\mathrm{PLUM}}.p$ (PROP)
then $\mathsf{send}.p.q.\boxed{(\mathrm{PROP\_mes}.p)}$

$[]$

$[]_{q \in \mathsf{neighs}.p}$ **if** $finished\_collecting\_and\_propagating.p \wedge \neg reported\_to\_father.p$ (DONE)
then (**if** $(q = (\mathsf{father}.p))$ **then** $\mathsf{send}.p.q.\boxed{(\mathrm{DONE\_mes}.p)}$)

Figure 8.12: The PLUM algorithm, parametrised with $iA \in \mathtt{Expr}$, $h \in \mathbb{P} \to \mathtt{Expr} \to \mathtt{Expr}$, PROP_mes $\in \mathbb{P} \to \mathtt{Expr}$, and DONE_mes $\in \mathbb{P} \to \mathtt{Expr}$

Termination of a distributed hylomorphism is independent of the contents of the messages that are being sent and how these are processed upon receipt. Consequently, the correctness criterion for $\Pi \in \{\text{PLUM, ECHO, TARRY, DFS}\}$ can, for some invariant J_Π of Π, be specified as:

Specification 8.10.1 *HYLO_Π*

$\forall iA,\ h,\ \text{PROP\_mes},\ \text{DONE\_mes} ::$

$$J_\Pi{}_{\ \Pi.iA.h.\text{PROP\_mes}.\text{DONE\_mes}} \vdash \mathbf{ini}(\Pi.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \rightsquigarrow (\forall p : p \in \mathbb{P} : done.p)$$

◀

Evidently, once this correctness criterion is proved for some invariant J_Π of Π, it can be inferred for all specific iA, h, PROP_mes, and DONE_mes. As will be shown in subsequent sections, this significantly reduces the proof effort for specific applications of our base algorithms. The precise characterisation of the invariants J_Π ($\Pi \in \{\text{PLUM, ECHO, TARRY, DFS}\}$) will be constructed in the next chapter and are left unspecified here, for now it is enough to know that J_Π is an invariant of Π with which is it is possible to prove Specification 8.10.1.

8.10.2 Propagation of information with feedback

Propagation of information with feedback (PIF) is the problem [Seg83] of broadcasting a piece of information I to all processes in a connected network in such a way that all the processes "know" when they have finished participating in the broadcast, one process in particular (the *starter*) "knows" that the broadcast is completed, and upon completion all processes own the piece of information I. A process p "knows" that it has finished participating in the broadcast when it has received and sent messages from and to all its neighbours, i.e. when *done.p*. Moreover, when the *starter* is *done* it "knows" that the broadcast is completed. To make Π suitable for the PIF application, instantiating Π such that:

- the starter initially has I stored in its local variable ($V.starter$)
- in the (PROP) phase of process p, the value stored in ($V.p$) is sent
- upon receipt of a message m in the (IDLE) phase of process p, this value is copied to variable $V.p$.
- the messages received in the COL phases are discarded
- the contents of the messages sent in the DONE phase remains unspecified

More formally, this is expressed by:

Definition 8.10.2
For arbitrary DONE_mes $\in$ process$\rightarrow$Expr:

$$\text{PIF}_\Pi = \Pi.(\lambda s.\ ((s \circ V).starter) = I).\mathsf{id}.(\lambda p.\ \mathsf{VAR}.(V.p)).\text{DONE\_mes}$$

◀

The formal specification of PIF applications reads:

Specification 8.10.3

$$(J_\Pi \wedge J_{\mathrm{PIF}}) \ {}_{\mathrm{PIF\_\Pi}}\vdash \ \mathbf{ini}\mathrm{PIF\_}\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : \mathit{done}.p) \wedge (\forall p : p \in \mathbb{P} : (V.p) = I)$$

◀

where J_{PIF} is some invariant of PIF_Π stating additional safety behaviour. Using the fact that PIF_Π is an instantiation of Π, the correctness criterion above can easily be proved by applying the following decomposition strategy:

$$\begin{array}{l}
\quad (J_\Pi \wedge J_{\mathrm{PIF}}) \ {}_{\mathrm{PIF}_\Pi}\vdash \ \mathbf{ini}\mathrm{PIF}_\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : \mathit{done}.p) \wedge (\forall p : p \in \mathbb{P} : (V.p) = I) \\
\Leftarrow (\rightsquigarrow \textsc{Substitution}\ (4.6.3_{50}) \text{ and } 8.10.2 \text{ (i.e. } \mathbf{ini}\mathrm{PIF}_\Pi \text{ includes } \mathbf{ini}\Pi)) \\
\quad (J_\Pi \wedge J_{\mathrm{PIF}}) \ {}_{\mathrm{PIF}_\Pi}\vdash \ \mathbf{ini}\Pi \wedge \mathbf{ini}\mathrm{PIF}_\Pi \\
\qquad\qquad\qquad\qquad \rightsquigarrow \\
\qquad\qquad\qquad\qquad (\forall p : p \in \mathbb{P} : \mathit{done}.p) \wedge (\forall p : p \in \mathbb{P} : (V.p) = I) \\
\Leftarrow (\rightsquigarrow \textsc{Conjunction}\ (4.5.19_{49}) \text{ and } \rightsquigarrow \textsc{Stable Strengthening}\ (4.6.9_{50})) \\
\quad J_\Pi \ {}_{\mathrm{PIF}_\Pi}\vdash \ \mathbf{ini}\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : \mathit{done}.p) \\
\quad \wedge \\
\quad {}_{\mathrm{PIF}_\Pi}\vdash \circlearrowleft J_\Pi \wedge J_{\mathrm{PIF}} \\
\quad \wedge \\
\quad (J_\Pi \wedge J_{\mathrm{PIF}}) \ {}_{\mathrm{PIF}_\Pi}\vdash \ \mathbf{ini}\mathrm{PIF}_\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : (V.p) = I) \\
\Leftarrow (\text{Definition 8.10.2, Specification 8.10.1 proves the first conjunct}) \\
\quad (\ {}_{\mathrm{PIF}_\Pi}\vdash \circlearrowleft J_\Pi \wedge J_{\mathrm{PIF}}) \\
\quad \wedge \\
\quad (J_\Pi \wedge J_{\mathrm{PIF}}) \ {}_{\mathrm{PIF}_\Pi}\vdash \ \mathbf{ini}\mathrm{PIF}_\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : (V.p) = I)
\end{array}$$

These last two conjuncts are application-specific.

8.10.3 Computation of summation functions

Suppose that every process $p \in \mathbb{P}$ in the network has a unique local variable that stores some data value. The distribution of local variables in the network in given by a function $V : \mathbb{P} \rightarrow \mathtt{Var}$, and consequently, in any state s the distribution of local data values is given by $(s \circ V)$.

Computation of a *summation function* in such a network is based upon a commutative monoid[5] $(\oplus, e)$, a function $f \in \mathbb{P} \rightarrow \mathtt{Val}$, and is defined by:

Definition 8.10.4 Computation of summation *SUM*

$$\mathsf{SUM}.\oplus.e.f.\mathbb{P} = \mathsf{foldr}.e.\oplus.(\mathsf{map}.f.(\mathsf{s2l}.\mathbb{P}))$$

◀

Suppose $(\oplus,e)$ is a commutative monoid. It is straightforward [Cho94a, Tel94, Vaa95, Hes97, GS96, GMS97] to modify the algorithms from Figures 8.8 through 8.11 such

[5] A commutative, associative operator $\oplus$ with an identity element e.

that the result of $\mathsf{SUM}.\oplus.e.V.\mathbb{P}$ is computed and eventually resides at the *starter* (i.e. is stored in the variable $V.starter$). Instantiate Π such that:

- D_0 contains the initial distribution of the data values
- in the (PROP) phase of process p, e is sent
- in the (DONE) phase of process p, $\mathsf{VAR}.(V.p)$ is sent
- upon receiving a message m in the (IDLE) and (COL) phase, this value m is $\oplus$-ed to the data value that resides in $(V.p)$, and the result is stored in $(V.p)$.

Let us denote the specific SUM-application of Π by SUM_Π.

Definition 8.10.5 *SUM_Π*

$$\begin{aligned}\mathrm{SUM}_\Pi = \Pi.&(\lambda s.\ D_0 = (s \circ V))\\ &.(\lambda p\ m.\ \mathsf{BI\_APPLY}.\oplus.(\mathsf{VAR}.(V.p)).m)\\ &.(\lambda p.\ \mathsf{CONST}.e)\\ &.(\lambda p.\ \mathsf{VAR}.(V.p))\end{aligned}$$

◀

The formal specification of these summation algorithms, stating the correctness criterion they should satisfy, reads:

Specification 8.10.6 *Π_SUMMATION_SPEC*

$$\frac{\mathsf{MONOID}.\oplus.e \wedge \mathsf{COMMUTATIVE}.\oplus}{J_\Pi \wedge J_S \ {}_{\mathrm{SUM}_\Pi}\!\vdash\ \mathbf{ini}\mathrm{SUM}_\Pi \rightsquigarrow (V.starter) = \mathsf{SUM}.\oplus.e.D_0.\mathbb{P}}$$

◀

where J_S is some invariant stating additional safety behaviour of SUM_Π. Proving this correctness criterion for SUM_Π, in such a way that progress properties already proved for Π are inherited, can now be done using the following strategy. Suppose $(\oplus, e)$ is a commutative monoid:

$$\begin{array}{l}
\quad J_\Pi \wedge J_S \ {}_{\mathrm{SUM}_\Pi}\!\vdash\ \mathbf{ini}\mathrm{SUM}_\Pi \rightsquigarrow (V.starter) = \mathsf{SUM}.\oplus.e.D_0.\mathbb{P}\\
\Leftarrow (\rightsquigarrow \textsc{Substitution}\ (4.6.3_{50}), \text{and} \rightsquigarrow \textsc{Stable Strengthening}\ (4.6.9_{50}))\\
\quad J_\Pi \ {}_{\mathrm{SUM}_\Pi}\!\vdash\ \mathbf{ini}\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : done.p)\\
\quad \wedge\\
\quad {}_{\mathrm{SUM}_\Pi}\!\vdash\!\circlearrowright (J_\Pi \wedge J_S)\\
\quad \wedge\\
\quad J_\Pi \wedge J_S \wedge (\forall p : p \in \mathbb{P} : done.p) \Rightarrow (V.starter) = \mathsf{SUM}.\oplus.e.D_0.\mathbb{P}\\
\Leftarrow (\text{Definition 8.10.5, and Specification 8.10.1})\\
\quad {}_{\mathrm{SUM}_\Pi}\!\vdash\!\circlearrowright (J_\Pi \wedge J_S)\\
\quad \wedge\\
\quad J_\Pi \wedge J_S \wedge (\forall p : p \in \mathbb{P} : done.p) \Rightarrow (V.starter) = \mathsf{SUM}.\oplus.e.D_0.\mathbb{P}
\end{array}$$

Consequently, the only thing left to do is find the characterisation of invariant J_S that establishes the last conjunct. Note that this proof strategy is not ad hoc, and neither are invariants "pulled out of a hat". The proof strategy is based and invented as to

establish inheritance of already proven facts about the underlying algorithm (i.e. Π). The invariant J_S, still left unspecified up to this point, has to be constructed so as to satisfy the second conjunct of the derivation above. Although some ingenuity is required in order to come up with this invariant, its construction is guided by the availability of information on its use within the process of verifying the correctness criterion.

The thought behind the construction of the invariant J_S is that any message sent during the execution of the algorithm is:

- either e (in the PROP phases), or
- the data value $(V.p)$ residing at some process p (in the DONE phases). However, after sending $(V.p)$ process p will be *done*.

Hence, since e is the identity element of $\oplus$, the desired sum $\mathsf{SUM}.\oplus.e.D_0.\mathbb{P}$ will, in any state s, be: the sum of values that reside at the processes that are not *done*, added to the sum of values that are in transit in s. Thus we define J_S to be the following state-predicate:

Definition 8.10.7 *Invariant_SUM*

$$
\begin{array}{lll}
J_S & \triangleq & \lambda\, s.\ (\ \mathsf{SUM}.\oplus.e.D_0.\mathbb{P} \\
& & \quad = (s \circ V).starter \\
& & \quad\ \oplus \\
& & \quad\ \mathsf{SUM}.\oplus.e.(s \circ V).\{p \mid p \in \mathbb{P} \wedge (p \neq starter) \wedge \neg done.p\} \\
& & \quad\ \oplus \\
& & \quad\ \text{(the values in the communication channels)} \\
& & \ )
\end{array}
$$

◀

Using this invariant, we have to prove:

Theorem 8.10.8 *all_done_IMP_starter_has_SUM*

$$\frac{\mathsf{MONOID}.\oplus.e \wedge \mathsf{COMMUTATIVE}.\oplus}{J_\Pi \wedge J_S \wedge (\forall p : p \in \mathbb{P} : done.p) \Rightarrow (V.starter) = \mathsf{SUM}.\oplus.e.D_0.\mathbb{P}}$$

◀

---

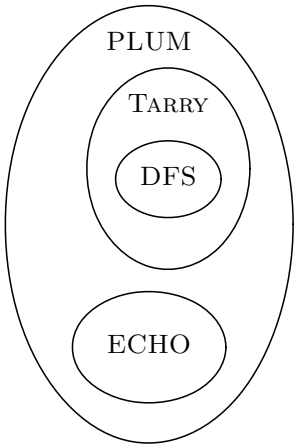
and, finally, we have to prove:

Theorem 8.10.9 *STABLEe_Invariant_SUM_*Π

$$\text{SUM\_}\Pi \vdash \circlearrowleft (J_\Pi \wedge J_S)$$

◀

Note that invariant J_Π is still unspecified at this point, and its precise characterisation is not needed to be able to derive a proof strategy for the summation application. Consequently, any instantiation of PLUM, ECHO, TARRY and DFS algorithms that maintains safety property J_S can be used to compute the sum. This means that not only do the enhanced representations increase the readability of the algorithms, they



$\forall A \in \{\text{IDLE, COL, PROP, DONE}\},\ p \in \mathbb{P},\ q \in \mathsf{neighs}.p$

$\mathcal{R}\_\text{PLUM\_ECHO}.(A_{\text{PLUM}}.p.q).(A_{\text{ECHO}}.p.q)$
$\mathcal{R}\_\text{PLUM\_TARRY}.(A_{\text{PLUM}}.p.q).(A_{\text{TARRY}}.p.q)$
$\mathcal{R}\_\text{TARRY\_DFS}.(A_{\text{TARRY}}.p.q).(A_{\text{DFS}}.p.q)$

(a) (b)

Figure 8.13: (a) refinement relation on PLUM, ECHO, TARRY, and DFS, (b) bitotal relations

also clearly separate the progress and safety properties. Hence, the correctness of an application of one of these algorithms (which consists of an extension of the algorithm) can be proved by verifying the safety property of the extension and inheriting the progress proof (including invariant J_Π that was needed to prove this progress) of the original algorithm.

8.11 Some notational conventions

For the sake of readability in the subsequent sections, we introduce some notational conventions.

For every $A \in \{\text{IDLE, COL, PROP, DONE}\}$, we use $A_\Pi.p.q$ to indicate that specific action of distributed hylomorphism Π for processes $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$. For the propagating actions of DFS, $\text{PROP}_{\text{DFS}}.p.q$ means either PROP_LP_REC.$p.q$ or PROP_NOT_LP_REC.$p.q$.

For every $\Pi \in \{$ PLUM, ECHO, TARRY, DFS$\}$ and arbitrary iA, h, PROP_mes, and DONE_mes we write Π to denote $\Pi.iA.h.\text{PROP\_mes}.\text{DONE\_mes}$. Therefore, theorems containing Π will implicitly be universally quantified by iA, h, PROP_mes, and DONE_mes.

The state predicate *can_propagate* from Definition 8.7.3 is abbreviated by *cp*.

8.12 A refinement ordering on distributed hylomorphisms

Based on the analysis in the previous sections, an intuitively clear operational refinement relation can be identified on PLUM, ECHO, TARRY and DFS which is based on a specific semantic model of (UNITY) programs where execution sequences exist of sets of sequences of possible receive and send events. The proposed refinement ordering is visualised with Venn-diagrams in Figure 8.13(a). Formalising this refinement ordering within our framework of refinements (Chapter 7) is straightforward. The bitotal relations, with respect to which the different refinements are proved, are listed in Figure 8.13(b) (their precise characterisations can be found in Appendix D.5). Their definitions are straightforward, in that they relate all IDLE, COL, PROP and DONE actions of the original program to the corresponding actions in the refinement. For the relation between TARRY and DFS this results in $\text{PROP}_{\text{TARRY}}.p.q$ being related to both PROP_LP_REC.$p.q$ and PROP_NOT_LP_REC.$p.q$. Although tedious, proving the bitotality of these relations and subsequently verifying the refinement ordering depicted in Figure 8.10.3 is reasonably easy. For the sake of completeness, the resulting refinement theorems are listed below.

Theorem 8.12.1 *PLUM_refines_ECHO*

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM\_ECHO},\ J} \text{ECHO}$$

Theorem 8.12.2 *PLUM_refines_Tarry*

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM\_TARRY},\ J} \text{TARRY}$$

Theorem 8.12.3 *Tarry_refines_DFS*

$$\forall J :: \text{TARRY} \sqsubseteq_{\mathcal{R}\_\text{TARRY\_DFS},\ J} \text{DFS}$$

◀

8.13 Correctness of distributed hylomorphisms

This section describes the approach that is taken to prove the correctness of the algorithms in the class of distributed hylomorphisms. The main objective of the approach is to reduce proof effort and complexity by re-using as many results as possible.

As already set out in the previous section, if termination of a distributed hylomorphism has been proved, the correctness of any application of this algorithm can be verified by using this termination property and verifying some additional safety behaviour inherent to the specific application. Moreover, the refinement ordering

on the distributed hylomorphisms indicates that, when termination has been proved for PLUM, then it can be deduced for ECHO, Tarry and DFS by using one of the property preserving theorems from our refinement framework. Consequently, the most efficient approach to verify the correctness of all distributed hylomorphisms (including every specific application) consists of the following steps:

(1) prove refinement relations amongst distributed hylomorphisms
(2) prove termination for the one that is refined by all others
(3) prove termination for all others using the refinement framework
(4) prove correctness of various applications

This approach is reflected in the resulting hierarchy of HOL theories, which is depicted in Figure 8.14. This hierarchy is a continuation of the theory hierarchy in Figure 5.1_{56}, in that the refinements, actions_transformations, and pvt_ops theories in Figure 8.14 correspond to those in Figure 5.1_{56}.

network is the theory about centralised and decentralised connected networks described in Section 8.2.

RST constitutes theory about rooted spanning trees, which is necessary when the proof strategy delineated in Section 8.1 is employed during the verification of distributed hylomorphisms. (The contents of this theory shall be discussed in the next chapter.)

communication contains the theory about asynchronous communication from Section 8.3.

Distributed_Hylomorphisms embodies definitions $8.7.1_{136}$ through $8.7.7_{136}$.

PLUM formalises the PLUM algorithm, and contains the theorem stating that it is a well-formed UNITY program according to Definition $4.3.1_{43}$. (See Appendix D.)

PLUM_INV defines and proves the invariant of PLUM. The exact formalisation of this invariant and its construction will be described in the next chapter.

ANA_PLUM contains the proof of the anamorphism part of the distributed hylomorphism, i.e. the construction of a rooted spanning tree.

CATA_PLUM contains the proof of the catamorphism part, i.e. using the rooted spanning tree to establish the desired result.

HYLO_PLUM combines the anamorphism and catamorphism part to prove termination of PLUM.

SUM_PLUM defines the specific summation application of the PLUM algorithm (Definition 8.10.5), and verifies Theorems 8.10.6 through 8.10.9.

ECHO, Tarry, DFS formalise ECHO, Tarry, and DFS respectively; contain theorems about their well-formedness (Definition $4.3.1_{43}$); define the bitotal relations depicted in Figure 8.13(b); and verify theorems Theorems 8.12.1 through 8.12.3. (See Appendix D.)

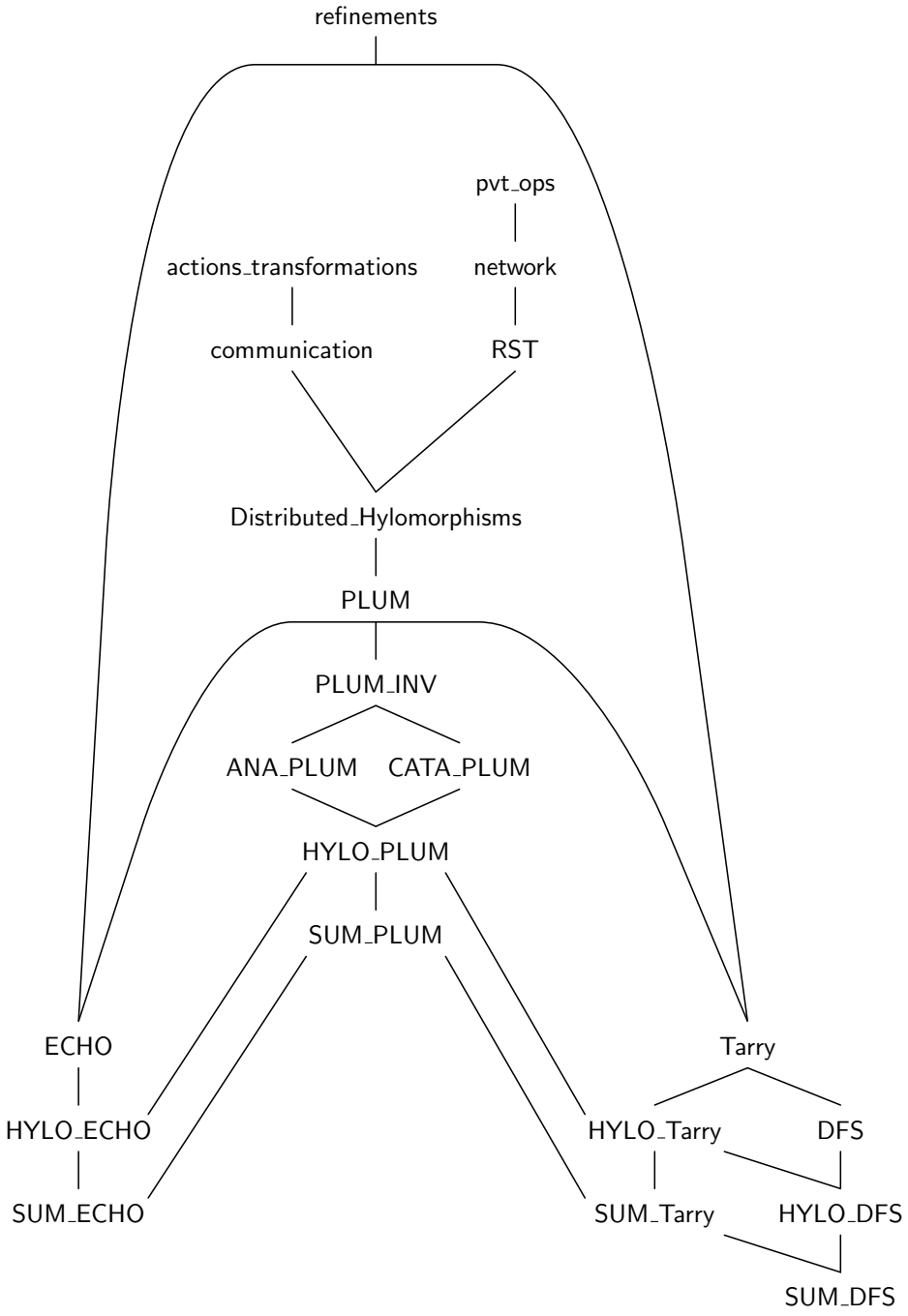


Figure 8.14: Theory hierarchy

HYLO_ECHO, HYLO_Tarry, HYLO_DFS prove termination of ECHO, TARRY, and DFS respectively by using the refinement framework described in Chapter 7.

SUM_ECHO, SUM_Tarry, SUM_DFS define the specific of summation applications of ECHO, TARRY, and DFS respectively (Definition 8.10.5); and verify Theorems 8.10.6 and 8.10.9 by using the refinement framework described in Chapter 7.

We end this section by analysing the time spent on the various mechanical verification activities involved in proving the correctness of each distributed hylomorphism. Since the next chapter shall treat the formal verification activities in detail, we shall be brief and concentrate on justifying the time spans indicated.

Verification of termination of the PLUM algorithms was the most time-consuming part (4 months), since, starting from scratch, we had to:
- formalise the PLUM algorithm and prove its well-formedness
- construct and verify PLUM's invariant (J_{PLUM})
- invent the exact proof strategy and apply it

Verifying PLUM's suitability for the computation of summation functions was done in two days. Most of this time was spent on verification of the invariant J_S (definition 8.10.7), which involved proving lots of additional theorems on lists.

However, having verified termination and summation for PLUM, the formalisations and the correctness proofs of the the other distributed hylomorphisms became significantly easier and less time-consuming.

Formalising ECHO, TARRY, and DFS is simple, since these are defined by augmenting and strengthening the guards of actions from PLUM. Their precise definitions can be found in Appendix D).

Verification of summation for ECHO was done in 1 hour. Once Theorem $7.2.12_{112}$ was used to deduce that J_S is also an invariant of ECHO, the rest of the proof could be done analogously to that of PLUM. Verification of summation for TARRY and DFS was done analogously to that of ECHO and hence – since we already knew how to do this – took even less time.

Verification of termination of ECHO took 3 days. Theorem $7.2.10_{113}$ was used to deduce termination. Since no variables are superposed upon PLUM to construct ECHO, and only the COL-actions vary among PLUM and ECHO, application of Theorem $7.2.10_{113}$ resulted in only one non-trivial proof obligation:

$$(J_{\text{PLUM}} \wedge J_{\text{ECHO}}) \ {}_{\text{ECHO}}\vdash \mathsf{guard\_of}.\text{COL}_{\text{PLUM}}.p.q \rightarrowtail \mathsf{guard\_of}.\text{COL}_{\text{ECHO}}.p.q$$

for processes $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$, which comes down to proving:

$$(J_{\text{PLUM}} \wedge J_{\text{ECHO}})\ {}_{\text{ECHO}}\vdash\ \begin{array}{l}\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs}.p\\ \rightarrowtail\\ \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs}.p\\ \wedge\ \mathit{sent\_to\_all\_non\_fathers}.p\end{array}$$

Proving this is not difficult: no additional invariant for ECHO is needed (i.e. J_{ECHO} can be true); and the remainder of the proof turned out to be similar to a theorem proved during the verification of termination of PLUM.

Verification of termination of TARRY took 18 days, which is six times as long as the verification of ECHO. The reason for this is that Theorem $7.2.9_{113}$ has to be used in order to be able to deduce termination; an application of this theorem introduces some time-consuming verification activities:

- proving that PLUM's invariant does not depend on the superposed variables $\mathsf{le\_rec}.p$
- constructing and verifying a non-decreasing function over the variables of TARRY
- verifying that for processes $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$:

  $$(J_{\text{PLUM}} \wedge J_{\text{TARRY}})\ {}_{\text{TARRY}}\vdash \mathsf{guard\_of}.\text{COL}_{\text{PLUM}}.p.q \rightarrowtail \mathsf{guard\_of}.\text{COL}_{\text{TARRY}}.p.q$$

  which comes down to proving:

  $$(J_{\text{PLUM}} \wedge J_{\text{TARRY}})\ {}_{\text{TARRY}}\vdash\ \begin{array}{l}\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs}.p\\ \rightarrowtail\\ \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs}.p\\ \wedge\ \neg(\mathsf{le\_rec}.p)\end{array}$$

  In order to be able to verify this it is necessary to specify and verify the invariant J_{TARRY} in such a way that it captures the alternating receive and send behaviour which is inherent to TARRY.
- verifying that for processes $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$:

  $$(J_{\text{PLUM}} \wedge J_{\text{TARRY}})\ {}_{\text{TARRY}}\vdash \mathsf{guard\_of}.\text{PROP}_{\text{PLUM}}.p.q \rightarrowtail \mathsf{guard\_of}.\text{PROP}_{\text{TARRY}}.p.q$$

  which comes down to proving:

  $$(J_{\text{PLUM}} \wedge J_{\text{TARRY}})\ {}_{\text{TARRY}}\vdash\ \begin{array}{l}\neg\mathsf{idle}.p \wedge \mathit{cp}.p.q \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.p\\ \rightarrowtail\\ \neg\mathsf{idle}.p \wedge \mathit{cp}.p.q \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.p\\ \wedge\ (\mathsf{le\_rec}.p)\end{array}$$

  Proving this proof obligation is not easy: when a non-idle process p in PLUM is allowed to send a message to q, this same process in TARRY has to wait until it receives a message (i.e. $\mathsf{le\_rec}.p$ becomes true). Many messages may be sent in the network until $\mathsf{le\_rec}.p$ becomes true, and consequently this proof obligation has to be proved by a well-foundedness argument (using $4.5.17_{48}$). Fortunately, the non-decreasing function over the variables of TARRY constructed in order to apply Theorem $7.2.9_{113}$ could be re-used in the well-foundedness argument.

	Termination	Computation of summation functions
PLUM	4 months	2 days
ECHO	3 days	1 hour
TARRY	18 days	30 minutes
DFS	8 days	30 minutes

Table 8.1: Time spent on *mechanically* verifying various distributed hylomorphisms

Verification of termination of DFS took 8 days. TARRY's invariant could easily be deduced for DFS using Theorem $7.2.12_{112}$. Since every PROP-action of TARRY is bitotally related to two different actions from DFS (namely PROP_LP_REC and PROP_NOT_LP_REC), $7.2.7_{113}$ has to be used to infer termination of DFS. Although like TARRY, application of this theorem results in some non-trivial proof obligations, these were proved with less effort, since:

- the non-decreasing function constructed for TARRY could be re-used
- no additional invariant properties had to be proved for TARRY
- verification of the proof obligations:

$$(J_{\text{PLUM}} \wedge J_{\text{TARRY}})\ {}_{\text{DFS}}\vdash \mathsf{guard\_of}.\text{COL}_{\text{TARRY}}.p.q \rightarrowtail \mathsf{guard\_of}.\text{COL}_{\text{DFS}}.p.q$$

and

$$(J_{\text{PLUM}} \wedge J_{\text{TARRY}})\ {}_{\text{DFS}}\vdash \mathsf{guard\_of}.\text{PROP}_{\text{TARRY}}.p.q \rightarrowtail \exists A :: (A \in \mathbf{a}\text{DFS}) \wedge \mathsf{guard\_of}.A.p.q$$

was established by proofs similar to those for TARRY.

This ends the brief analysis of the verification activities involved in proving distributed hylomorphisms. As already indicated, the next chapter shall discuss them in more detail.

8.14 Conclusions

Analysing a class of algorithms, detecting similarities, and constructing a good representation in such a way that

- a least deterministic variant can be invented,
- a refinement ordering can be identified,
- and the representation abstracts from specific applications,

has led to the construction of an efficient approach to verify the correctness of the class of algorithms. To validate these claims, we have summarised the time spent on

the various (mechanical) verification activities involved in proving the correctness of the class of distributed hylomorphisms in Table 8.1. The table confirms that for the class of distributed hylomorphisms

- finding the least deterministic variant and starting with verification of its termination, significantly reduces the time spent on proving the correctness of the other distributed hylomorphisms
- and, as a consequence of the previous bullet, our framework of refinements from the previous chapter is effective for the reduction of proof effort
- moreover, abstracting from applications reduces the complexity of their formalisation and verification.

Don't worry if you don't immediately understand the strategy behind a proof you are reading. Just try to follow the justifications of the steps, and the strategy will eventually become clear. If it doesn't, a second reading of the proof might help.

– D.J. Velleman [Vel94]

Chapter 9

Formally proving the correctness of distributed hylomorphisms

This chapter presents detailed formal proofs of the correctness of distributed hylomorphisms with respect to their termination. We use the approach delineated in Chapter 8, i.e. first prove termination of PLUM in Section 9.1, and then use the refinements framework from Chapter 7 to derive termination of the other distributed hylomorphisms (Sections 9.3 through 9.5).

To refute the statement made in [Cho95], that the construction of invariants is a "trial-and-error" process that needs a great deal of ingenuity, we shall construct our invariant J_{PLUM} incrementally in a demand driven way *during* the process of verification. Although this results in an intuitive and structured invention of the invariant, it also contributes to the spaciousness of this chapter.

9.1 Proving termination of PLUM

Verifying the correctness of termination for PLUM is done following the second and third step of the UNITY methodology described in Chapter 6. The UNITY specification, stating termination of PLUM, was already stated in the previous chapter and, for some invariant J_{PLUM} of PLUM, reads:

Theorem 9.1.1 *HYLO_PLUM*

$$\begin{array}{l} \forall iA,\ h,\ \text{PROP\_mes},\ \text{DONE\_mes} :: \\ \quad J_{\text{PLUM}}\ \ {}_{\text{PLUM}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}} \vdash \ \ \mathbf{ini}(\text{PLUM}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \rightsquigarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall p : p \in \mathbb{P} : \mathit{done}.p \end{array}$$

Theorem 9.1.2 VARIABLES IGNORED BY IDLE *Vars_IG_BY_IDLE*

$\{\mathsf{idle}.p, \mathsf{father}.p, \mathsf{M}.q.p, \mathsf{nr\_rec}.p.q, \mathsf{V}.p\}^{\mathsf{c}} \not\leftarrow \textsc{idle}.p.q$

Theorem 9.1.3 VARIABLES IGNORED BY COL *Vars_IG_BY_COL*

$\{\mathsf{M}.q.p, \mathsf{nr\_rec}.p.q, \mathsf{V}.p\}^{\mathsf{c}} \not\leftarrow \textsc{col}.p.q$

Theorem 9.1.4 VARIABLES IGNORED BY PROP *Vars_IG_BY_PROP*

$\{\mathsf{M}.p.q, \mathsf{nr\_sent}.p.q\}^{\mathsf{c}} \not\leftarrow \textsc{prop}.p.q$

Theorem 9.1.5 VARIABLES IGNORED BY DONE *Vars_IG_BY_DONE*

$\{\mathsf{M}.p.q, \mathsf{nr\_sent}.p.q\}^{\mathsf{c}} \not\leftarrow \textsc{done}.p.q$

Figure 9.1: Variables ignored by the actions from PLUM

This specification is refined and decomposed – using the laws of the UNITY logic which were presented in Chapter 4 – until it is expressed in one-step progress (i.e. ensures) and safety (i.e. ↻) properties that can be proved directly from the actions of the PLUM algorithm (see Figure 8.12_{143}).

9.1.1 Incremental, demand-driven construction of invariants

As already stated, we shall construct our invariant J_{PLUM} incrementally in a demand driven way *during* the process of refinement and decomposition. More specific, at the begin of the refinement and decomposition, the invariant J_{PLUM} is unspecified. Subsequently, at those points in the proof where an invariant is needed we propose a candidate cJ^i_{PLUM} for part of the invariant which suffices for that particular point in the proof. After decomposition, we gather all the candidates we have proposed during the refinement and decomposition of the initial specification, and from them deduce the minimal invariant J_{PLUM} that implies all the proposed candidates. To give a clear indication when a candidate for part of the invariant is proposed we shall mark this point by:

➰ $cJ^i_{\text{PLUM}} = \ldots$

Once introduced it is assumed that J_{PLUM} implies the candidate, since this shall be ensured at the end of the decomposition. Similarly, we shall assume the stability of J_{PLUM} throughout the whole process of refinement and decomposition. Finally, we will call a candidate that is proposed for being part of the invariant, an *invariant-candidate*.

Theorem 9.1.6 *guard_of_IDLE*

$\mathsf{guard\_of}.(\text{IDLE}.p.q) = \mathsf{idle}.p \wedge \mathsf{mit}.q.p$

Theorem 9.1.7 *guard_of_COL*

$\mathsf{guard\_of}.(\text{COL}.p.q) = \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs}.p$

Theorem 9.1.8 *guard_of_PROP*

$\mathsf{guard\_of}.(\text{PROP}.p.q)$
$= \neg(\mathsf{idle}.p) \wedge (\mathsf{nr\_sent}.p.q = 0) \wedge (q \neq (\mathsf{father}.p)) \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.p$

Theorem 9.1.9 *guard_of_DONE*

$\mathsf{guard\_of}.(\text{DONE}.p.q)$
$= \mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p \wedge (q = (\mathsf{father}.p))$

Figure 9.2: Guards of the actions from PLUM

9.1.2 PLUM's variables and actions

During the verification, we shall assume that all of PLUM's variables are distinct. That is, e.g. for the idle variables it is assumed that:

$$\forall p, q \in \mathbb{P} : (\mathsf{idle}.p = \mathsf{idle}.q) = (p = q)$$

Similar properties are assumed for the V, father, nr_rec, nr_sent, and M variables. Moreover, we assume that the various kinds of variables are different, e.g. for the idle variables we assume:

$$\begin{aligned} \forall p, q, r \in \mathbb{P} : \quad & (\mathsf{idle}.p \neq \mathsf{V}.q) \wedge (\mathsf{idle}.p \neq \mathsf{father}.q) \wedge (\mathsf{idle}.p \neq \mathsf{nr\_rec}.q.r) \\ & (\mathsf{idle}.p \neq \mathsf{nr\_sent}.q.r) \wedge (\mathsf{idle}.p \neq \mathsf{M}.q.r) \end{aligned}$$

Again similar properties are assumed for the V, father, nr_rec, nr_sent, and M variables. The HOL definition capturing these properties of PLUM's variables is called distinct_PLUM_Vars, we do not present it here, since obviously it is very tedious and takes up a lot of space. It can be found in Appendix D in Definition D.1.4_{254}.

Another assumption, implicitly made during the verification activities in this chapter, is the (intended) type declaration of the communication variables (see Definition 8.3.2_{123}): ASYNC_type_decl.$\mathbb{P}$.neighs

Theorems 9.1.2_{158} through 9.1.5_{158} indicate which variables are written by the various actions of the PLUM algorithm. (For the definition of $\nleftarrow$ see 3.4.22_{36}.) Since we assume the validity of distinct_PLUM_Vars, we know that if, for example, $(p \neq p')$, then

action IDLE.$p.q$ does not write to the variables $\mathsf{idle}.p'$, $\mathsf{father}.p'$, $\mathsf{M}.q.p'$, $\mathsf{nr\_rec}.p'.q$, and $\mathsf{V}.p'$.

For ease of referring to the guards of the various actions of PLUM, Theorems $9.1.6_{159}$ through $9.1.9_{159}$ state them.

9.1.3 Presenting proofs of unless and ensures properties

During the refinement and decomposition of the specification, various one-step safety (i.e. unless) and progress (i.e. ensures) properties have to be verified. To enhance the readability of their proofs, this section shall introduce the proof format for the verification of these properties.

The proof obligations stating ensures -properties are introduced through an application of the $\rightsquigarrow$ INTRODUCTION ($4.6.4_{50}$) theorem. More specifically, applying this theorem results in proof obligations of the form::

$$\vdash (J_{\text{PLUM}} \wedge x) \text{ ensures } y$$

Rewriting with Definitions $4.4.1_{43}$ and $4.4.2_{43}$ gives us:

$$\left.\begin{array}{l} \forall A \in \mathbf{a}\text{PLUM}, s, t \in \texttt{State}: \\ \quad \mathsf{evalb}.(J_{\text{PLUM}}.s) \wedge \mathsf{evalb}.(x.s) \wedge \neg \mathsf{evalb}.(y.s) \wedge \mathsf{compile}.A.s.t \\ \quad \Rightarrow \\ \quad (\mathsf{evalb}.(J_{\text{PLUM}}.t) \wedge \mathsf{evalb}.(x.t)) \vee \mathsf{evalb}.(y.t) \end{array}\right\} \text{unless} - \text{part}$$

$$\wedge$$

$$\left.\begin{array}{l} \exists A \in \mathbf{a}\text{PLUM} : \forall s, t \in \texttt{State}: \\ \quad \mathsf{evalb}.(J_{\text{PLUM}}.s) \wedge \mathsf{evalb}.(x.s) \wedge \neg \mathsf{evalb}.(y.s) \wedge \mathsf{compile}.A.s.t \\ \quad \Rightarrow \\ \quad \mathsf{evalb}.(y.t) \end{array}\right\} \text{exists} - \text{part}$$

To prevent tedious rewriting with the definitions of unless and ensures , and repeated discharging of the hypotheses at the left hand side of the implications, we introduce the proof-format displayed in Figure 9.3_{161}.

9.1.4 Some more theorems, notation and assumptions

Figure 9.4_{161} displays some simple theorems that turn out to be useful during the verification, they all follow naturally from the Definitions $8.7.1_{136}$ through $8.7.7_{136}$.

During the whole process of verification, we shall assume that we have a connected centralised communication network. i.e. $\mathsf{Connected\_Network}.\mathbb{P}.\mathsf{neighs.starter}$.

Moreover, during the process of decomposition:

$\vdash$ and $_{\text{PLUM}}\vdash$ abbreviate J_{PLUM} $_{\text{PLUM}}.iA.h.\text{PROP\_}\mathsf{mes}.\text{DONE\_}\mathsf{mes}\vdash$

$\vdash (J_{\text{PLUM}} \wedge x)$ ensures y

unless-part.
IDLE.$p'.q'.s.t$
the proof that is displayed here, implicitly assumes the validity of
- evalb.$(J_{\text{PLUM}}.s)$ (and hence because of the assumed stability of J_{PLUM} (Section 9.1.1) also evalb.$(J_{\text{PLUM}}.t)$)
- evalb.$(x.s)$
- $\neg$evalb.$(y.s)$
- compile.$(\text{IDLE}.p'.q').s.t$

and aims to verify that evalb.$(x.t) \vee$ evalb.$(y.t)$.
COL.$p'.q'.s.t$ dito, but then for COL
PROP.$p'.q'.s.t$ dito, but then for PROP
DONE.$p'.q'.s.t$ dito, but then for DONE

exists-part: directly after the colon we shall write that action A that is used to reduce the existential quantification.
Then, we present a proof that – under the implicit assumptions that evalb.$(J_{\text{PLUM}}.s)$, evalb.$(x.s)$, and $\neg$evalb.$(y.s) \wedge$ compile.$A.s.t$ – verifies that the action establishes the desired progress (i.e. evalb.$(y.t)$).

Figure 9.3: The proof-format for the verification of ensures -properties

Theorem 9.1.10 *not_evalb_sent_2_all_except_f*
For all processes $p \in \mathbb{P}$ and states $s \in$ State:

$\neg$*sent_to_all_non_fathers.p.s*
$= \exists q : q \in$ neighs.$p \ \wedge \ q \neq s.(\text{father}.p) \ \wedge \ s.(\text{nr\_sent}.p.q) \neq 1$

Theorem 9.1.11 *not_evalb_rec_from_all_neighs*
For all processes $p \in \mathbb{P}$ and states $s \in$ State:

$\neg$*rec_from_all_neighs.p.s* $= \exists q : q \in$ neighs.$p \ \wedge \ s.(\text{nr\_rec}.p.q) \neq 1$

Theorem 9.1.12 *finished_and_sent_2_f_IMP_sent_2_all_neighs*
For all processes $p \in \mathbb{P}$ and states $s \in$ State:

$$\frac{\textit{finished\_collecting\_and\_propagating.p.s} \wedge \textit{reported\_to\_father.p.s}}{\textit{sent\_to\_all\_neighs.p.s}}$$

Figure 9.4: Some useful theorems

for arbitrary $iA \in \texttt{Expr}$, $h \in \mathbb{P}\rightarrow\texttt{Expr}\rightarrow\texttt{Expr}$, PROP_mes $\in \mathbb{P}\rightarrow\texttt{Expr}$, and DONE_mes $\in \mathbb{P}\rightarrow\texttt{Expr}$, for which hold that:

$\forall p, e : p \in \mathbb{P} \wedge e\, \mathcal{C}\ \mathbf{w}\mathrm{PLUM} : (h.p.e)\, \mathcal{C}\ \mathbf{w}\mathrm{PLUM}$
$\forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p\, \mathcal{C}\ \mathbf{w}\mathrm{PLUM} \ \wedge\ \mathsf{DONE\_mes}.p\, \mathcal{C}\ \mathbf{w}\mathrm{PLUM}$

These assumptions about h, PROP_mes, and DONE_mes are needed to prove that PLUM (as it is displayed in Figure 8.12_{143}) is a well-formed UNITY program (i.e. satisfies the predicate Unity, see Theorem $\mathrm{D.1.10}_{255}$).

9.1.5 Refinement and decomposition strategy

The global strategy applied to decompose the specification stating termination of distributed hylomorphisms, was already described in Section 8.1_{120}, and is inherent to the structure of distributed hylomorphisms:

$$\underbrace{\text{let the information flow from leaves to root of the RST}}_{\textbf{cata}} \circ \underbrace{\text{build an RST}}_{\textbf{ana}}$$

Distributed hylomorphisms build an RST by flooding messages to all processes in such a way that:

- when an idle process p receives its first message from q, it marks q as its father and opens its floodgate by becoming non-idle
- non-idle processes only flood (i.e. propagate) messages to non-father-neighbours.

Consequently, the shape of the rooted spanning tree is established by the father relation, once all processes have become non-idle. The construction of the tree, however, is finished only when

(1) every process has sent messages to all its neighbours that are not its father (i.e. it has sent messages to all of its non-father-neighbours)
(2) all messages meant in (1) are actually received (i.e. every process has received messages from all of its non-child-neighbours)

Requirement (1) is captured by the definition of *sent_to_all_non_fathers*, given earlier in Chapter 8 (Definition $8.7.2_{136}$). Requirement (2) is, for some process $p \in \mathbb{P}$, characterised by the following definition:

Definition 9.1.13 RECEIVED FROM ALL NON-CHILDREN *rec_from_all_non_child*

$rec\_from\_all\_non\_children.p = \forall q \in \mathsf{neighs}.p : (p \neq (\mathsf{father}.q)) \Rightarrow (\mathsf{nr\_rec}.p.q = 1)$

---

this predicate states that process p has at least received messages from those neighbours of which p is not the father. Thus, in other words, p has at least received messages from all its non-child-neighbours.

Applying this global proof strategy to the initial specification results in the following **anamorphism**- and **catamorphism**-part:

$\vdash \mathbf{ini}(\mathrm{PLUM}.iA.h.\mathrm{PROP\_mes}.\mathrm{DONE\_mes}) \rightsquigarrow \forall p : p \in \mathbb{P} : done.p$

$\Leftarrow(\rightsquigarrow$ TRANSITIVITY $(4.6.6_{50}))$

$$\left.\begin{array}{ll} \vdash & \textbf{ini}(\text{PLUM}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \\ & \rightsquigarrow \\ & (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \\ & \wedge(\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\ & \wedge(\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \end{array}\right\} \textbf{anamorphism} - \textbf{part}$$

$\wedge$

$$\left.\begin{array}{ll} \vdash & (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \\ & \wedge(\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\ & \wedge(\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \\ & \rightsquigarrow \\ & \forall p : p \in \mathbb{P} : \mathit{done}.p \end{array}\right\} \textbf{catamorphism} - \textbf{part}$$

9.1.6 Verification of the anamorphism part

Decomposition of the **anamorphism**-part is straightforward and follows naturally from the discussion in the previous section: first prove that the shape of the RST is established by proving that all processes eventually become non-idle (**ana_1**); then prove that all processes end the construction of the RST by sending messages to all their non-father-neighbours (**ana_2**); finally prove that all messages sent in order to construct the RST are eventually received (**ana_3**).

$$\left.\begin{array}{ll} \vdash & \textbf{ini}(\text{PLUM}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \\ & \rightsquigarrow \\ & (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \\ & \wedge(\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\ & \wedge(\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \end{array}\right\} \textbf{anamorphism} - \textbf{part}$$

$\Leftarrow(\rightsquigarrow$ ACCUMULATION $(4.6.8_{50})$, twice)

$$\left.\begin{array}{ll} \vdash & \textbf{ini}(\text{PLUM}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \\ & \rightsquigarrow \\ & \forall p \in \mathbb{P} : \neg \mathsf{idle}.p \end{array}\right\} \textbf{ana\_1}$$

$\wedge$

$$\left.\begin{array}{ll} \vdash & \forall p \in \mathbb{P} : \neg \mathsf{idle}.p \\ & \rightsquigarrow \\ & \forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p \end{array}\right\} \textbf{ana\_2}$$

$\wedge$

$$\left.\begin{array}{ll} \vdash & (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \wedge (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\ & \rightsquigarrow \\ & (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \wedge (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \end{array}\right\} \textbf{ana\_3}$$

The verification of ana_1

Decomposition of **ana_1** proceeds by induction on the structure of the connected network underlying the PLUM algorithm. That is, we prove that when a process p is non-idle, then eventually all its neighbours will become non-idle. Consequently, from the connectivity of the network it can be deduced that since the *starter* is non-idle, eventually all processes will be non-idle.

$\vdash$ **ini**(PLUM.$iA.h$.PROP_mes.DONE_mes) $\rightsquigarrow \forall p \in \mathbb{P} : \neg\mathsf{idle}.p$ } **ana_1**
$\Leftarrow$($\rightsquigarrow$ SUBSTITUTION ($4.6.3_{50}$), using characterisation of initial condition PLUM)
$\vdash \forall p \in \{starter\} : \neg\mathsf{idle}.p \rightsquigarrow \forall p \in \mathbb{P} : \neg\mathsf{idle}.p$
$\Leftarrow$(rewrite with the definition of Connected_Network ($8.2.3_{122}$))
$\vdash \forall p \in \{starter\} : \neg\mathsf{idle}.p \rightsquigarrow \forall p \in \mathsf{iterate}.n.(\mathsf{Neighs.neighs}).starter : \neg\mathsf{idle}.p$
$\Leftarrow$($\rightsquigarrow$ ITERATE ($4.5.21_{49}$))
$\forall L \subseteq \mathbb{P} : \vdash \forall p \in L : \neg\mathsf{idle}.p \rightsquigarrow \forall p \in \mathsf{Neighs.neighs}.L : \neg\mathsf{idle}.p$
$\Leftarrow$($\rightsquigarrow$ SUBSTITUTION ($4.6.3_{50}$), prepare for $\rightsquigarrow$ CONJUNCTION ($4.5.19_{49}$))
$\forall L \subseteq \mathbb{P} : \vdash \forall p \in L, \forall q \in \mathsf{neighs}.p : \neg\mathsf{idle}.p \wedge \neg\mathsf{idle}.p \rightsquigarrow \forall p \in L, \forall q \in \mathsf{neighs}.p : \neg\mathsf{idle}.q$
$\Leftarrow$($\rightsquigarrow$ CONJUNCTION ($4.5.19_{49}$), three times)
$\forall L \subseteq \mathbb{P}, p \in L, q \in \mathsf{neighs}.p : \ (\vdash \neg\mathsf{idle}.p \rightsquigarrow \neg\mathsf{idle}.p) \ \wedge \ (\vdash \neg\mathsf{idle}.p \rightsquigarrow \neg\mathsf{idle}.q)$

The first conjunct can be proved using $\rightsquigarrow$ REFLEXIVITY ($4.6.5_{50}$), and the stability of $\neg\mathsf{idle}.p$, stated below:

Theorem 9.1.14 *STABLEe_not_idle*

$\forall p \in \mathbb{P} : \ {}_{\text{PLUM}}\vdash \circlearrowright \neg\mathsf{idle}.p$

We now proceed with the second conjunct. Since q is assumed to be an arbitrary neighbour of p, we have to make a distinction as to whether q is p's father or not.

$\Leftarrow$($\rightsquigarrow$ CASE DISTINCTION ($4.6.7_{50}$))
$\forall L \subseteq \mathbb{P}, p \in L, q \in \mathsf{neighs}.p :$
$\underbrace{\vdash \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p) \rightsquigarrow \neg\mathsf{idle}.q}_{\textbf{ana\_1.1}} \ \wedge \ \underbrace{\vdash \neg\mathsf{idle}.p \wedge (q \neq \mathsf{father}.p) \rightsquigarrow \neg\mathsf{idle}.q}_{\textbf{ana\_1.2}}$

Examine the first conjunct **ana_1.1**, we need to verify that when a process p is non-idle, then eventually its father will be non-idle. When a process p is not idle, it has received a message from its father. Hence its father is not idle since otherwise it would not have been able to send a message to p. Therefore, the first conjunct should be provable from the invariant as follows: for arbitrary $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$:

$\vdash \neg\mathsf{idle}.p \wedge q = \mathsf{father}.p \rightsquigarrow \neg\mathsf{idle}.q$
$\Leftarrow$($\rightsquigarrow$ INTRODUCTION ($4.6.4_{50}$))
$((J_{\text{PLUM}} \wedge \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \Rightarrow \neg\mathsf{idle}.q) \ \wedge \vdash \ \circlearrowright (J_{\text{PLUM}} \wedge \neg\mathsf{idle}.q)$

In order to establish this proof we introduce our first candidate for part of the invariant J_{PLUM}:

$$cJ^1_{\text{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \neg\mathsf{idle}.p \wedge q = \mathsf{father}.p \Rightarrow \neg\mathsf{idle}.q$$

Obviously, when J_{PLUM} implies cJ^1_{PLUM}, the stability of J_{PLUM}, and the stability of $(\neg\mathsf{idle}.q)$ (stated in Theorem 9.1.14) establish **ana_1.1**.

The second conjunct **ana_1.2**, states that when a process p is non-idle, then eventually its non-father neighbours will be non-idle. Evidently, when p is non-idle, it shall eventually send a message to its non-father neighbour q; moreover, q shall eventually receive this message and, when not already non-idle, shall become non-idle. This is reflected in the following decomposition strategy: for arbitrary $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$:

$$\begin{array}{l} \vdash \neg\mathsf{idle}.p \wedge q \neq \mathsf{father}.p \rightsquigarrow \neg\mathsf{idle}.q \\ \Leftarrow (\rightsquigarrow \text{TRANSITIVITY } (4.6.6_{50})) \\ \quad \underbrace{\vdash \neg\mathsf{idle}.p \wedge q \neq \mathsf{father}.p \rightsquigarrow \mathsf{nr\_sent}.p.q = 1}_{\textbf{ana\_1.2.1}} \;\wedge\; \underbrace{\vdash \mathsf{nr\_sent}.p.q = 1 \rightsquigarrow \neg\mathsf{idle}.q}_{\textbf{ana\_1.2.2}} \end{array}$$

ana_1.2.1 can be proved using $\rightsquigarrow$ INTRODUCTION $(4.6.4_{50})$, leaving us with the proof obligations:

$$\begin{array}{l} \vdash\ \circlearrowleft(J_{\text{PLUM}} \wedge \mathsf{nr\_sent}.p.q = 1) \\ \wedge \\ \vdash (J_{\text{PLUM}} \wedge \neg\mathsf{idle}.p \wedge q \neq \mathsf{father}.p)\ \mathsf{ensures}\ (\mathsf{nr\_sent}.p.q = 1) \end{array}$$

Stability of $(\mathsf{nr\_sent}.p.q = 1)$ can be proved separately from invariant J_{PLUM}, since, for all $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$, the guards of PROP.$p.q$ and DONE.$p.q$ imply that $\mathsf{nr\_sent}.p.q = 0$. The proof is straightforward and the resulting theorem is presented below.

Theorem 9.1.15 *STABLEe_nr_sent_is_1*

$$\forall p, q \in \mathbb{P} :\ {}_{\text{PLUM}}\vdash \circlearrowleft(\mathsf{nr\_sent}.p.q = 1)$$

Consequently,

$$\begin{array}{l} \vdash\ \circlearrowleft(J_{\text{PLUM}} \wedge (\mathsf{nr\_sent}.p.q = 1)) \\ \Leftarrow (\circlearrowleft \text{CONJUNCTION } 4.4.4_{44}) \\ \quad \vdash\ \circlearrowleft J_{\text{PLUM}}\ \wedge\ \vdash\ \circlearrowleft(\mathsf{nr\_sent}.p.q = 1) \end{array}$$

Which is proved by the assumed stability of J_{PLUM}, and Theorem 9.1.15 from above.

The validation of the ensures -property is below:

$$\vdash (J_{\text{PLUM}} \wedge \neg\mathsf{idle}.p \wedge q \neq \mathsf{father}.p)\ \mathsf{ensures}\ (\mathsf{nr\_sent}.p.q = 1)$$

unless-part

IDLE.$p'.q'.s.t$

- if $p \neq p'$, then the variables idle.p and father.p are not written by IDLE.$p'.q'.s.t$ and thus $s.(\mathsf{idle}.p) = t.(\mathsf{idle}.p)$ and $s.(\mathsf{father}.p) = t.(\mathsf{father}.p)$.
- if $p = p'$, then $(s = t)$ since the guard of IDLE.$p'.q'.s.t$ is disabled by $\neg s.(\mathsf{idle}.p)$. (see the explanation on the implicit assumptions implied by the presentation of ensures -properties from Section 9.1.3).

COL.$p'.q'.s.t$, PROP.$p'.q'.s.t$, DONE.$p'.q'.s.t$ do not write to the idle and father variables (Theorems $9.1.3_{158}$ through $9.1.5_{158}$).

exists-part: PROP.$p.q.s.t$.

In order to verify that this action indeed sends a message to its neighbour q, we have to prove that its guard is enabled in state s. More specific (Theorem $9.1.8_{159}$) this comes down to verifying that:

$$\neg s.(\mathsf{idle}.p) \wedge (s.(\mathsf{nr\_sent}.p.q) = 0) \wedge (q \neq s.(\mathsf{father}.p)) \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.p.s$$

since the implicit assumptions of the presentation of ensures -properties tell us that $\neg s.(\mathsf{idle}.p)$, $(q \neq s.(\mathsf{father}.p))$, and $(s.(\mathsf{nr\_sent}.p.q) \neq 1)$, and hence Theorem $9.1.10_{161}$ implies that $\neg \mathit{sent\_to\_all\_non\_fathers}.p.s$, the following proof obligation remains:

$$s.(\mathsf{nr\_sent}.p.q) = 0$$

In order to prove this, we need to propose an additional candidate for part of the invariant. Since we have that $(s.(\mathsf{nr\_sent}.p.q) \neq 1)$, the invariant-part that suffices here, is a predicate stating that the number of messages a process has sent to a neighbour is always 0 or 1.

$$cJ^2_{\mathrm{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathsf{nr\_sent}.p.q = 0 \ \vee \ \mathsf{nr\_sent}.p.q = 1$$

This ends the validation of **ana_1.2.1**.

Using Theorem $9.1.14_{164}$, the assumed stability of J_{PLUM}, ⟲ CONJUNCTION $4.4.4_{44}$, and ⇝ INTRODUCTION $(4.6.4_{50})$, the proof obligation **ana_1.2.2** can be reduced to:

$$\vdash (J_{\mathrm{PLUM}} \wedge \mathsf{nr\_sent}.p.q = 1) \ \mathsf{ensures} \ (\neg \mathsf{idle}.q)$$

unless-part

IDLE.$p'.q'.s.t$, COL.$p'.q'.s.t$ do not write to the nr_sent variables (Theorems $9.1.2_{158}$ and $9.1.3_{158}$).

PROP.$p'.q'.s.t$

- If $(p \neq p')$ or $(q \neq q')$, the variable nr_sent.$p.q$ is not written.
- If $(p = p')$ and $(q = q')$, then $s = t$ since the guard of PROP.$p'.q'.s.t$ is disabled by $(s.(\mathsf{nr\_sent}.p'.q') = 1)$.

DONE.$p'.q'.s.t$

- If $(p \neq p')$ or $(q \neq q')$ the variable nr_sent.$p.q$ is not written.
- Suppose $(p = p')$ and $(q = q')$.

- If $q' \neq s.(\mathsf{father}.p')$ then the guard of DONE.$p'.q'.s.t$ is disabled and hence $s = t$.
- Suppose $q' = s.(\mathsf{father}.p')$.
  - If $\neg$*finished_collecting_and_propagating.p.s*, then, from Theorem 9.1.9$_{159}$, we can deduce that the guard of DONE.$p'.q'.s.t$ is disabled, and hence that $s = t$.
  - If *finished_collecting_and_propagating.p.s*, then using Definition 8.7.4$_{136}$ we have that p has *sent_to_all_non_fathers* in state s. Moreover, since we know that $(s.(\mathsf{nr\_sent}.p'.(s.(\mathsf{father}.p'))) = 1)$ we have that (Theorem 9.1.12$_{161}$) *sent_to_all_neighs.p.s* and thus *done.p.s*. Consequently, the guard of DONE.$p'.q'.s.t$ is disabled and hence $s = t$.

exists-part: IDLE.$q.p.s.t$

In order to verify that process q indeed receives a message from its neighbour p, and becomes non-idle we have to prove that the guard of IDLE.$q.p.s.t$ is enabled in state s. Using Theorem 9.1.6$_{159}$, and the assumption that $s.(\mathsf{idle}.p)$ this comes down to verifying that:

$\mathsf{mit}.p.q.s$

The implicit assumptions and the already proposed invariant-candidates cJ^1_{PLUM} and cJ^2_{PLUM} do not give enough information to prove this. Consequently, we shall again have to construct some additional invariant-candidates. Intuitively, when a message is in transit from p to q this will always mean that $(\mathsf{nr\_rec}.q.p < \mathsf{nr\_sent}.p.q)$. Moreover, when a process p is idle this means that is has not yet received any message and hence all its nr_rec variables are 0. Proposing these as candidates for part of the invariant, enables us to prove the current exists-part. Since we have here that q is idle and $s.(\mathsf{nr\_sent}.p.q = 1)$, we can deduce that $(s.(\mathsf{nr\_rec}.q.p) < s.(\mathsf{nr\_sent}.p.q))$ and hence $\mathsf{mit}.p.q.s$.

$cJ^3_{\mathrm{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathsf{idle}.p \Rightarrow \mathsf{nr\_rec}.p.q = 0$

$cJ^4_{\mathrm{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : (\mathsf{nr\_rec}.q.p < \mathsf{nr\_sent}.p.q) = \mathsf{mit}.p.q$

This establishes the proof of **ana_1.2.2**, **ana_1.2**, and hence **ana_1**. For future reference the results are summarised in Figure 9.5$_{168}$.

The verification of ana_2

Proving that a non-idle process shall eventually send messages to all its non-father-neighbours can be proved by re-using **ana_1.2.1** (Theorem 9.1.16$_{168}$). The following derivation aims at bringing **ana_2** into the correct form for application of **ana_1.2.1**. (The notes ♫, with which some of the derivation steps are marked, can be ignored here. Their purpose will become clear later on.)

Theorem 9.1.16 ana_1.2.1 *not_idle_CON_sent_2_neighs_ex_f*

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : J_{\text{PLUM PLUM}} \vdash \neg\mathsf{idle}.p \wedge (q \neq \mathsf{father}.p) \rightsquigarrow \mathsf{nr\_sent}.p.q = 1$

Theorem 9.1.17 ana_1.2.2 *sent_to_q_CON_not_idle_q*

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : J_{\text{PLUM PLUM}} \vdash \mathsf{nr\_sent}.p.q = 1 \rightsquigarrow \neg\mathsf{idle}.q$

Theorem 9.1.18 ana_1.1 *not_idle_CON_idle_father*

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : J_{\text{PLUM PLUM}} \vdash \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p) \rightsquigarrow \neg\mathsf{idle}.q$

Theorem 9.1.19 ana_1.2 *not_idle_CON_not_idle_neighs*

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : J_{\text{PLUM PLUM}} \vdash \neg\mathsf{idle}.p \wedge (q \neq \mathsf{father}.p) \rightsquigarrow \neg\mathsf{idle}.q$

Theorem 9.1.20 ana_1 *Init_CON_all_not_idle*

$J_{\text{PLUM PLUM}} \vdash \mathbf{ini}(\text{PLUM}.iA.h.\text{PROP\_mes.DONE\_mes}) \rightsquigarrow \forall p \in \mathbb{P} : \neg\mathsf{idle}.p$

Figure 9.5: Verification of **ana_1**

$\vdash \quad \forall p \in \mathbb{P} : \neg\mathsf{idle}.p \rightsquigarrow \forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p \Big\}$ **ana_2**

$\Leftarrow$($\rightsquigarrow$ Substitution ($4.6.3_{50}$), $8.7.2_{136}$; prepare for $\rightsquigarrow$ Conjunction ($4.5.19_{49}$)) (♫)

$\vdash \quad \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \neg\mathsf{idle}.p \rightsquigarrow \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : (\neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \vee (\mathsf{nr\_sent}.p.q = 1)$

$\Leftarrow$($\rightsquigarrow$ Conjunction ($4.5.19_{49}$), twice) (♫)

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \vdash \neg\mathsf{idle}.p \rightsquigarrow (\neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \vee (\mathsf{nr\_sent}.p.q = 1)$

$\Leftarrow$($\rightsquigarrow$ Case distinction ($4.6.7_{50}$))

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :$

$\vdash \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p) \rightsquigarrow (\neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \vee (\mathsf{nr\_sent}.p.q = 1)$

$\wedge$

$\vdash \neg\mathsf{idle}.p \wedge (q \neq \mathsf{father}.p) \rightsquigarrow (\neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \vee (\mathsf{nr\_sent}.p.q = 1)$

$\Leftarrow$($\rightsquigarrow$ Substitution ($4.6.3_{50}$) on the right hand side of both conjuncts)

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :$

$\vdash \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p) \rightsquigarrow \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)$

$\wedge$

$\vdash \neg\mathsf{idle}.p \wedge (q \neq \mathsf{father}.p) \rightsquigarrow (\mathsf{nr\_sent}.p.q = 1)$

$\Leftarrow$(Second conjunct is proved by Theorem $9.1.16_{168}$)

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \vdash \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p) \rightsquigarrow \neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)$

$\Leftarrow$($\rightsquigarrow$ Reflexivity ($4.6.5_{50}$), $\circlearrowleft$Conjunction $4.4.4_{44}$, and assumed stability of J_{PLUM})

$\vdash \circlearrowleft \neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p)$

This stability predicate is straightforward to prove since a non-idle process stays non-idle (Theorem $9.1.14_{164}$) and does not write to its father variables.

Theorem 9.1.21 *STABLEe_not_idle_AND_q_IS_f_p*

$\forall p, q \in \mathbb{P} :\ {}_{\text{PLUM}}\vdash \circlearrowleft \neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p)$

For future reference we again summarise:

Theorem 9.1.22 ana_2 *not_idle_CON_not_propagating*

$J_{\text{PLUM}}\ {}_{\text{PLUM}}\vdash \forall p \in \mathbb{P} : \neg \mathsf{idle}.p \rightsquigarrow \forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p$

Theorem 9.1.23 *not_idle_AND_q_IS_f_p_CON_REFL*

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : J_{\text{PLUM}}\ {}_{\text{PLUM}}\vdash \neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p) \ \rightsquigarrow \neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p)$

Verification of ana_3

Proving **ana_3** comes down to verifying that when a message is sent, it shall eventually be received. In order to derive this proof obligation, we proceed as follows:

$$\left.\begin{array}{ll} \vdash & (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \wedge (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\ & \rightsquigarrow \\ & (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \wedge (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \end{array}\right\} \textbf{ana\_3}$$

$\Leftarrow$($\rightsquigarrow$ Substitution ($4.6.3_{50}$), Definition $8.7.2_{136}$, and Definition $9.1.13_{162}$)

$$\begin{array}{ll} \vdash & \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \neg \mathsf{idle}.p \wedge ((q \neq \mathsf{father}.p) \Rightarrow (\mathsf{nr\_sent}.p.q = 1)) \\ & \rightsquigarrow \\ & \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \neg \mathsf{idle}.p \wedge ((q \neq \mathsf{father}.p) \Rightarrow (\mathsf{nr\_rec}.q.p = 1)) \end{array}$$

$\Leftarrow$($\rightsquigarrow$ Conjunction ($4.5.19_{49}$), twice)

$$\begin{array}{l} \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \\ \quad \begin{array}{ll} \vdash & \neg \mathsf{idle}.p \wedge ((q \neq \mathsf{father}.p) \Rightarrow (\mathsf{nr\_sent}.p.q = 1)) \\ & \rightsquigarrow \\ & \neg \mathsf{idle}.p \wedge ((q \neq \mathsf{father}.p) \Rightarrow (\mathsf{nr\_rec}.q.p = 1)) \end{array} \end{array}$$

$=$ (logic)

$$\begin{array}{l} \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \\ \quad \begin{array}{ll} \vdash & (\neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \ \vee \ (\neg \mathsf{idle}.p \wedge (\mathsf{nr\_sent}.p.q = 1)) \\ & \rightsquigarrow \\ & (\neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \ \vee \ (\neg \mathsf{idle}.p \wedge (\mathsf{nr\_rec}.q.p = 1)) \end{array} \end{array}$$

$\Leftarrow$($\rightsquigarrow$ Disjunction ($4.5.18_{49}$))

$$\begin{array}{l} \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \\ \quad \vdash \quad \neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p) \rightsquigarrow \neg \mathsf{idle}.p \wedge (q = \mathsf{father}.p) \end{array}$$

$$
\begin{array}{l}
\quad \wedge \\
\quad\ \vdash \quad \neg\mathsf{idle}.p \wedge (\mathsf{nr\_sent}.p.q = 1) \rightsquigarrow \neg\mathsf{idle}.p \wedge (\mathsf{nr\_rec}.q.p = 1) \\
\Leftarrow\text{(First conjunct is proved by Theorem } 9.1.23_{169}) \\
\quad \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \\
\quad\ \vdash \quad \neg\mathsf{idle}.p \wedge (\mathsf{nr\_sent}.p.q = 1) \rightsquigarrow \neg\mathsf{idle}.p \wedge (\mathsf{nr\_rec}.q.p = 1) \\
\Leftarrow(\rightsquigarrow \textsc{Conjunction}\ (4.5.19_{49})) \\
\quad \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \\
\quad (\vdash \quad \neg\mathsf{idle}.p \rightsquigarrow \neg\mathsf{idle}.p) \ \wedge\ (\vdash \quad \mathsf{nr\_sent}.p.q = 1 \rightsquigarrow \mathsf{nr\_rec}.q.p = 1) \\
\Leftarrow\text{(First conjunct is proved using } \rightsquigarrow \textsc{Reflexivity}\ (4.6.5_{50}), \text{ and Theorem } 9.1.14_{164}) \\
\quad \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \vdash \mathsf{nr\_sent}.p.q = 1 \rightsquigarrow \mathsf{nr\_rec}.q.p = 1
\end{array}
$$

So we have to prove that when a process p sends a message to a neighbour q, then q shall eventually receive this message. Since nothing is known about q, there are two possibilities:

q **is non-idle** In this case the execution of COL.$q.p$ shall ensure that p's message is eventually received.

q **is idle** This case is more subtle, since it is not ensured that execution of IDLE.$q.p$ shall receive p's message. In illustration, suppose another neighbour r ($r \neq p$) has also sent a message to the idle process q. If q decides to receive r's message before it receives the one from p, then q registers r as its father and becomes non-idle. Consequently, subsequent executions of q's IDLE-actions will behave like skip and therefore shall not be responsible for the receipt of p's message. In this case q's COL actions will ensure that p's message is eventually received.

This is reflected in the following proof:

$$
\begin{array}{l}
\quad \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \vdash \mathsf{nr\_sent}.p.q = 1 \rightsquigarrow \mathsf{nr\_rec}.q.p = 1 \\
\Leftarrow(\rightsquigarrow \textsc{Case Distinction}\ (4.6.7_{50})) \\
\quad \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p \\
\quad \underbrace{\vdash \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \rightsquigarrow \mathsf{nr\_rec}.q.p = 1}_{\textbf{ana\_3.1}} \\
\quad \wedge \\
\quad \underbrace{\vdash \mathsf{nr\_sent}.p.q = 1 \wedge \mathsf{idle}.q \rightsquigarrow \mathsf{nr\_rec}.q.p = 1}_{\textbf{ana\_3.2}}
\end{array}
$$

As indicated, when q is non-idle (**ana_3.1**) the execution of COL.$q.p$ shall ensure that p's message is eventually received. Consequently, $\rightsquigarrow$ INTRODUCTION ($4.6.4_{50}$) is applied to **ana_3.1** giving us: for arbitrary $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$

$$
\begin{array}{l}
\quad \vdash \ \circlearrowleft J_{\mathrm{PLUM}} \wedge \mathsf{nr\_rec}.q.p = 1 \\
\quad \wedge \\
\quad \vdash (J_{\mathrm{PLUM}} \wedge \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q)\ \mathsf{ensures}\ (\mathsf{nr\_rec}.q.p = 1)
\end{array}
$$

Stability of ($\mathsf{nr\_rec}.q.p = 1$) cannot be proved separately from the stability of J_{PLUM}. The reason for this is that – unlike the guards of PROP.$p.q$ and DONE.$p.q$ that imply

that ($\mathsf{nr\_sent}.p.q = 0$) and hence allow for the separate verification of $\circlearrowleft(\mathsf{nr\_sent}.p.q = 1)$ – the guards of IDLE.$p.q$ and COL.$p.q$ actions do *not* imply that ($\mathsf{nr\_rec}.p.q = 0$). However, in combination with the proposed invariant-candidates they do. cJ^3_{PLUM} implies that when q is idle, $\mathsf{nr\_rec}.q.p = 0$. Therefore, when the guard of IDLE.$q.p$ (Definition $9.1.6_{159}$) is enabled the validity J_{PLUM} implies $\mathsf{nr\_rec}.q.p = 0$. cJ^4_{PLUM}, together with cJ^2_{PLUM}, implies that when $\mathsf{mit}.q.p$ holds, $\mathsf{nr\_rec}.q.p = 0$. Therefore, when the guard of COL.$q.p$ (Definition $9.1.7_{159}$) is enabled the validity J_{PLUM} implies $\mathsf{nr\_rec}.q.p = 0$. Consequently, we have the following theorem:

Theorem 9.1.24 *STABLEe_Invariant_AND_nr_rec_is_1*

$$\forall p, q \in \mathbb{P} : \ {}_{\text{PLUM}}\vdash \circlearrowleft(J_{\text{PLUM}} \wedge \mathsf{nr\_rec}.p.q = 1)$$

The validation of the ensures -property is below:

$$\vdash (J_{\text{PLUM}} \wedge \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q) \ \mathsf{ensures} \ (\mathsf{nr\_rec}.q.p = 1)$$

unless-part

IDLE.$p'.q'.s.t$

- if $(p' = q)$, then $(s = t)$ since the guard of IDLE.$p'.q'.s.t$ is disabled by $\neg s.(\mathsf{idle}.q)$.
- if $(p' \neq q)$ the variables $\mathsf{idle}.q$ and $\mathsf{nr\_sent}.p.q$ are not written

COL.$p'.q'.s.t$ does not write to idle and nr_sent variables (Theorem $9.1.3_{158}$).

PROP.$p'.q'.s.t$

- If $(p \neq p')$ or $(q \neq q')$ the variable $\mathsf{nr\_sent}.p.q$ is not written. (idle variables are not written at all by PROP)
- If $(p = p')$ and $(q = q')$, then $(s = t)$ since the guard of PROP.$p'.q'.s.t$ is disabled by $(s.(\mathsf{nr\_sent}.p'.q') = 1)$.

DONE.$p'.q'.s.t$

- If $(p \neq p')$ or $(q \neq q')$ the variable $\mathsf{nr\_sent}.p.q$ is not written. (idle variables are not written at all by DONE)
- Suppose $(p = p')$ and $(q = q')$.
  - If $q' \neq s.(\mathsf{father}.p')$ then, from Theorem $9.1.9_{159}$, we can deduce that the guard of DONE.$p'.q'.s.t$ is disabled and hence $s = t$.
  - Suppose $q' = s.(\mathsf{father}.p')$.
    - If $\neg$*finished_collecting_and_propagating.p.s*, then, from Theorem $9.1.9_{159}$, we can deduce that the guard of DONE.$p'.q'.s.t$ is disabled and hence $s = t$.
    - If *finished_collecting_and_propagating.p.s*, then using Definition $8.7.4_{136}$ we have *sent_to_all_non_fathers.p.s*. Moreover, since p' has already sent to its father (i.e. $(s.(\mathsf{nr\_sent}.p'.(s.(\mathsf{father}.p'))) = 1)$) we have that (Theorem $9.1.12_{161}$) *sent_to_all_neighs.p.s* and thus *done.p.s*. Consequently, the guard of DONE.$p'.q'.s.t$ is disabled and hence $s = t$.

exists-part: COL.$q.p.s.t$

In order to verify that process q indeed receives a message from its neighbour p, and establishes $t.(\mathsf{nr\_rec}.q.p) = 1$ we have to prove that the guard of COL.$q.p.s.t$ is

enabled in state s, and $s.(\mathsf{nr\_rec}.q.p) = 0$. Since $s.(\mathsf{nr\_rec}.q.p) \neq 1$, Theorem $9.1.11_{161}$ gives us $\neg \mathit{rec\_from\_all\_neighs}.q$. Using Theorem $9.1.7_{159}$, and the assumption that $\neg s.(\mathsf{idle}.p)$ the proof obligations that remain are:

$\mathsf{mit}.p.q.s \wedge s.(\mathsf{nr\_rec}.q.p) = 0$
$= (cJ^4_{\text{PLUM}}$, and the assumption that $s.(\mathsf{nr\_sent}.p.q) = 1)$
$s.(\mathsf{nr\_rec}.q.p) < 1 \wedge s.(\mathsf{nr\_rec}.q.p) = 0$
$=$ (arithmetic)
$s.(\mathsf{nr\_rec}.q.p) = 0$

Again, looking at the assumptions and the already proposed invariant-candidates, we do not have enough information to prove this. Consequently, we introduce the following candidate, which obviously suffices in this case.

$\frown\!\!\!\!\!\!\frown\ cJ^5_{\text{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : (\mathsf{nr\_rec}.p.q = 0) \vee (\mathsf{nr\_rec}.p.q = 1)$

We hereby end the proof of **ana_3.1**.

Theorem 9.1.25 ana_3.1 *not_idle_AND_neigh_has_sent_CON_rec*

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \ J_{\text{PLUM}} \ {}_{\text{PLUM}}\vdash \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \rightsquigarrow \mathsf{nr\_rec}.q.p = 1$ ◀

We continue with **ana_3.2** using the strategy delineated earlier on page 170.

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \vdash \mathsf{nr\_sent}.p.q = 1 \wedge \mathsf{idle}.q \rightsquigarrow \mathsf{nr\_rec}.q.p = 1$
$\Leftarrow$($\rightsquigarrow$ Transitivity $(4.6.6_{50})$)
$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :$
$\vdash \mathsf{nr\_sent}.p.q = 1 \wedge \mathsf{idle}.q \rightsquigarrow \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \wedge (\exists r : \mathsf{nr\_rec}.q.r = 1)$
$\wedge$
$\vdash \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \wedge (\exists r : \mathsf{nr\_rec}.q.r = 1) \rightsquigarrow \mathsf{nr\_rec}.q.p = 1$

Using $\rightsquigarrow$ Substitution $(4.6.3_{50})$, the second conjunct can be reduced to, and hence proved by, Theorem $9.1.25_{172}$. The first conjunct is proved by $\rightsquigarrow$ Introduction $(4.6.4_{50})$:

$\vdash \circlearrowleft(J_{\text{PLUM}} \wedge \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \wedge (\exists r : \mathsf{nr\_rec}.q.r = 1))$
$\wedge$
$\vdash (J_{\text{PLUM}} \wedge \mathsf{nr\_sent}.p.q = 1 \wedge \mathsf{idle}.q)$
ensures
$(\mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \wedge (\exists r : \mathsf{nr\_rec}.q.r = 1))$

The stability requirement can be proved using $\circlearrowleft$ Conjunction $4.4.4_{44}$, Theorems $9.1.14_{164}$, $9.1.15_{165}$, and $9.1.24_{171}$. The proof of the ensures-property is similar to that of **ana_3.1** on the understanding that IDLE.$q.p.s.t$ in instantiated in the exists-part instead of COL.$q.p.s.t$.

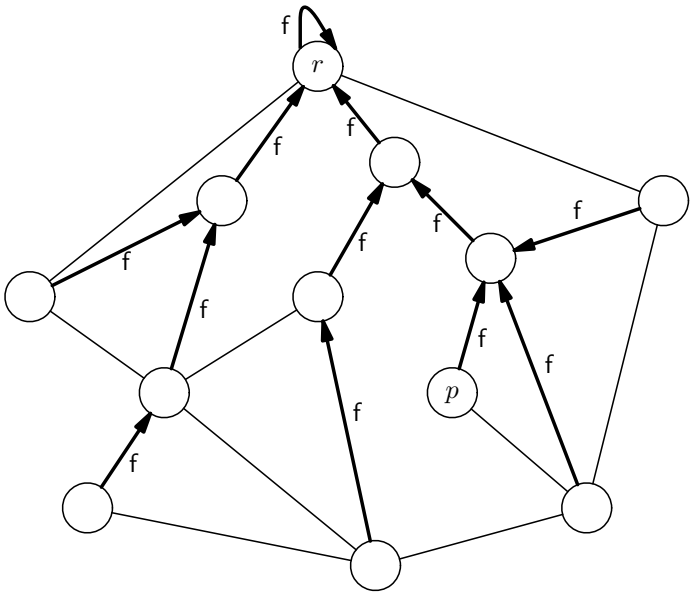


Figure 9.6: Rooted spanning tree; process p has depth 3.

Theorem 9.1.26 ana_3.2 *idle_AND_neigh_has_sent_CON_rec*

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \ J_{\text{PLUM PLUM}}\vdash \mathsf{nr\_sent}.p.q = 1 \wedge \mathsf{idle}.q \rightsquigarrow \mathsf{nr\_rec}.q.p = 1$

Theorem 9.1.27 ana_3 *not_propagating_and_not_idle_CON_not_idle_rec_from_all_non_child*

$$J_{\text{PLUM PLUM}}\vdash \ (\forall p \in \mathbb{P} : \neg\mathsf{idle}.p) \wedge (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \rightsquigarrow (\forall p \in \mathbb{P} : \neg\mathsf{idle}.p) \wedge (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p)$$

9.1.7 Theory on rooted spanning trees

A *rooted spanning tree* of a connected communication network ($\mathbb{P}$, neighs) (see Figure 9.6) is a directed graph and consists of:

- a unique designated process r of the network which is considered to be the root of the tree, and hence has no outgoing edges to other processes in the network.
- a subset of communication links of the network, such that for all processes $p \in \mathbb{P}$ it holds that there is a unique path from p to r in the tree.

The tree is characterised by a process r and a function $\mathsf{f} \in \mathbb{P} \rightarrow \mathbb{P}$ (see Figure 9.6). To formalise the fact that the root is a process in the network, and has no outgoing

edges to any other process, we define

$$(r \in \mathbb{P}) \wedge (\mathsf{f}.r = r)$$

Consequently, since the communication links in the tree have to be a subset of those in the network, f has to satisfy:

$$\forall p \in \mathbb{P} : (p \neq r) \Rightarrow (\mathsf{f}.p \in \mathsf{neighs}.p)$$

For ease of reference, when $q = \mathsf{f}.p$, we call q the *ancestor* or *father* of p, and similarly p the *descendant* or *child* of q. To specify that for every process $p \in \mathbb{P}$ there is a unique path from p to r in the tree, we define the depth of a process p, as follows:

Definition 9.1.28 *depth*

$$\mathsf{depth}.\mathsf{f}.r.p.k = (r = \mathsf{iterate}.k.\mathsf{f}.p) \wedge \forall m < k : \ (r \neq \mathsf{iterate}.m.\mathsf{f}.q)$$

◀

In words, process p has depth k, if the shortest path from p to r in the tree has length k. Since f is a function, the existence of a unique path from p to r equals the existence of a shortest path from p to r in the tree. Consequently, the requirement that for every process $p \in \mathbb{P}$ there has to be a unique path from p to r in the tree can be characterised by:

$$\forall p \in \mathbb{P} : \ \exists k : \ \mathsf{depth}.\mathsf{f}.r.p.k$$

Summarising, we have the following definition of a rooted spanning tree of a connected network ($\mathbb{P}$, neighs).

Definition 9.1.29 Rooted Spanning Tree *RST*

$$\begin{array}{lcl} \mathsf{RST}.\mathsf{f}.r.\mathbb{P}.\mathsf{neighs} & = & (r \in \mathbb{P}) \wedge (r = \mathsf{f}.r) \\ & & \forall p \in \mathbb{P} : \ (p \neq r) \Rightarrow (\mathsf{f}.p \in \mathsf{neighs}.p) \\ & & \forall p \in \mathbb{P} : \ \exists k : \ \mathsf{depth}.\mathsf{f}.r.p.k \end{array}$$

◀

Since every process in a rooted spanning tree has a unique depth, we can categorise processes into levels by using their depths. This is depicted in Figure 9.7. The set of processes at level k is defined as follows:

Theorem 9.1.30 *level*

$$\mathsf{level}.\mathbb{P}.\mathsf{f}.r.k = \{p \mid p \in \mathbb{P} \wedge \mathsf{depth}.\mathsf{f}.r.p.k\}$$

◀

When it is clear from the context which $\mathbb{P}$, f, and r are used, we shall abbreviate $\mathsf{level}.\mathbb{P}.\mathsf{f}.r.k$ by $\mathsf{level}.k$.

The height of a rooted spanning tree is defined to be the maximum of the depths of all processes in the underlying network:

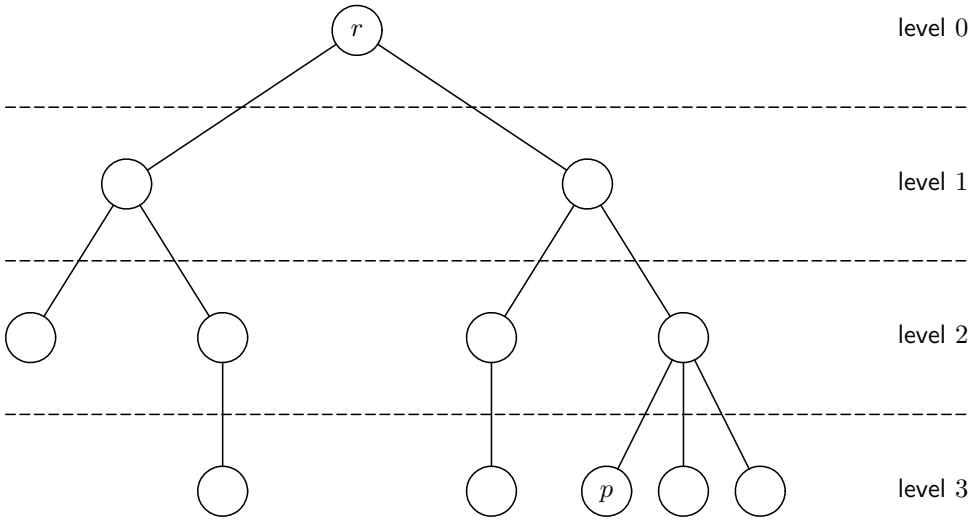


Figure 9.7: Processes categorised into levels.

Definition 9.1.31 HEIGHT OF TREE *height*

$$\mathsf{height}.\mathbb{P}.\mathsf{f}.r.\mathsf{neighs}.h = (h = \mathsf{max}.\{k \mid p \in \mathbb{P} \wedge \mathsf{depth}.\mathsf{f}.r.p.k\})$$

Again, when it is clear from the context which $\mathbb{P}$, f, r, and neighs are used, we shall abbreviate $\mathsf{height}.\mathbb{P}.\mathsf{f}.r.\mathsf{neighs}.h$ by $\mathsf{height}.h$. The reader can check that the height of the rooted spanning tree in Figure 9.6 is 4. Moreover, it is not hard to see that:

Theorem 9.1.32 *RST_has_height*

$$\frac{\mathsf{Connected\_Network}.\mathbb{P}.\mathsf{neighs}.\mathit{starter} \wedge \mathsf{RST}.\mathsf{f}.r.\mathbb{P}.\mathsf{neighs}}{\exists h : \mathsf{height}.\mathbb{P}.\mathsf{f}.r.\mathsf{neighs}.h}$$

9.1.8 Verification of the catamorphism part

$$\vdash \quad \left.\begin{array}{l} (\forall p \in \mathbb{P} : \neg\mathsf{idle}.p) \\ \wedge\ (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\ \wedge\ (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \\ \rightsquigarrow \\ \forall p : p \in \mathbb{P} : \mathit{done}.p \end{array}\right\} \textbf{catamorphism} - \textbf{part}$$

First of all we need to construct the function $\mathsf{f} \in \mathbb{P} \rightarrow \mathbb{P}$, that characterises the rooted spanning tree. Obviously, the father variables were set as to define such a function.

Consequently, we start by bringing this function f into the left hand side of $\rightsquigarrow$ as follows. In order to avoid confusion between the type of father and f we explicitly denote the state s in the last conjunct of the left hand side of $\rightsquigarrow$. As a consequence '$=$' is overloaded to denote eq (see Table 3.1_{23}), and *not* $=_*$ (see pages 25 through 27)).

$$
\begin{array}{l}
\Leftarrow(\rightsquigarrow \textsc{Substitution}\ (4.6.3_{50})) \\
\quad \vdash \quad \exists \mathsf{f} \in \mathbb{P} \rightarrow \mathbb{P} : \\
\qquad (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \\
\qquad \wedge\ (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\
\qquad \wedge\ (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \\
\qquad \wedge\ (\forall p \in \mathbb{P} : (\lambda s.\ \mathsf{f}.p = (s \circ \mathsf{father}).p)) \\
\qquad \rightsquigarrow \\
\qquad \forall p : p \in \mathbb{P} : \mathit{done}.p \\
\Leftarrow(\rightsquigarrow \textsc{Disjunction}\ (4.5.18_{49})) \\
\quad \forall \mathsf{f} \in \mathbb{P} \rightarrow \mathbb{P} : \\
\quad \vdash \quad (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \\
\qquad \wedge\ (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\
\qquad \wedge\ (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \\
\qquad \wedge\ (\forall p \in \mathbb{P} : (\lambda s.\ \mathsf{f}.p = (s \circ \mathsf{father}).p)) \\
\qquad \rightsquigarrow \\
\qquad \forall p : p \in \mathbb{P} : \mathit{done}.p
\end{array}
$$

Second, we have to prove that we have indeed built a rooted spanning tree. That is, we need to bring the conjunct $\mathsf{RST}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}$ into the left hand side of $\rightsquigarrow$. Using $\rightsquigarrow$ Substitution $(4.6.3_{50})$ this means we have to prove that:

$$
\begin{array}{lll}
\forall s \in \mathtt{State} : & J_{\textsc{plum}}.s & \\
& \wedge\ \forall p \in \mathbb{P} : \neg\ s.(\mathsf{idle}.p) & \\
& \wedge\ \forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p.s & \\
& \wedge\ \forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p.s & \\
& \wedge\ \forall p \in \mathbb{P} : \mathsf{f}.p = (s \circ \mathsf{father}).p & (9.1.33) \\
& \Rightarrow & \\
& (\mathit{starter} = \mathsf{f}.\mathit{starter}) & \mathbf{P}_1 \\
& \forall p \in \mathbb{P} :\ (p \neq \mathit{starter}) \Rightarrow (\mathsf{f}.p \in \mathsf{neighs}.p) & \mathbf{P}_2 \\
& \forall p \in \mathbb{P} :\ \exists k :\ \mathsf{depth}.\mathsf{f}.\mathit{starter}.p.k & \mathbf{P}_3
\end{array}
$$

Evidently, in order to be able to prove this, we shall need to invent some new candidates for part of the invariant. The first invariant-candidate follows naturally from the proof obligation $\mathbf{P}_1$. Since initially the *starter* is defined to be non-idle and $\mathsf{father}.\mathit{starter}$ equals[1] *starter*, the following is a valid (Theorem $9.1.21_{169}$) invariant-candidate[2]:

[1] Note that in order to be able to prove that this is an invariant we need the initial condition stating that: $\mathsf{father}.\mathit{starter} = \mathit{starter}$ (see page 134).

[2] Again we explicitly denote the state to avoid confusion. Consequently, and contrary to the other invariant-candidates, $=$, $\wedge$, $\neg$, and $\Rightarrow$ are not overloaded to denote `State`-lifted operators, but rather `Val`-lifted operators, see Table 3.1_{23}.

$$\overset{\frown}{\text{@@@@@@@}} cJ^6_{\text{PLUM}} = (\lambda s.\ (s \circ \mathsf{father}).starter = starter \ \wedge\ \neg s.(\mathsf{idle}.starter))$$

The next invariant-candidates are introduced as to establish proof obligation $\mathbf{P}_2$ and $\mathbf{P}_3$ respectively. Since processes only receive messages from their neighbours, and once non-idle never change the value of their father variable again, we propose:

$$\overset{\frown}{\text{@@@@@@@}} cJ^7_{\text{PLUM}} = (\lambda s.\ \forall p \in \mathbb{P} :\ \ (p \neq starter) \wedge \neg s.(\mathsf{idle}.p) \Rightarrow ((s \circ \mathsf{father}).p \in \mathsf{neighs}.p))$$

$$\overset{\frown}{\text{@@@@@@@}} cJ^8_{\text{PLUM}} = (\lambda s.\ \forall p \in \mathbb{P} :\ \neg s.(\mathsf{idle}.p) \Rightarrow \exists k :\ \mathsf{depth}.(s \circ \mathsf{father}).starter.p.k)$$

It is not hard to see that these candidates are sufficient to prove $9.1.33_{177}$.

Theorem 9.1.33 *all_not_idle_IMP_RST*

For all $\mathsf{f} \in \mathbb{P} \rightarrow \mathbb{P}$, $s \in$ `State`:

$$\frac{J_{\text{PLUM}}.s \wedge (\forall p \in \mathbb{P} : \neg\ s.(\mathsf{idle}.p)) \wedge (\forall p \in \mathbb{P} : sent\_to\_all\_non\_fathers.p.s)\\ (\forall p \in \mathbb{P} : rec\_from\_all\_non\_children.p.s) \wedge (\forall p \in \mathbb{P} : \mathsf{f}.p = (s \circ \mathsf{father}).p)}{\mathsf{RST}.\mathbb{P}.\mathsf{f}.starter.\mathsf{neighs}}$$

◀

For arbitrary $\mathsf{f} \in \mathbb{P} \rightarrow \mathbb{P}$, we now proceed with the catamorphism part as follows:

$$\begin{array}{ll} \vdash & (\forall p \in \mathbb{P} : \neg\mathsf{idle}.p)\\ & \wedge (\forall p \in \mathbb{P} : sent\_to\_all\_non\_fathers.p)\\ & \wedge (\forall p \in \mathbb{P} : rec\_from\_all\_non\_children.p)\\ & \wedge (\forall p \in \mathbb{P} : (\lambda s.\ \mathsf{f}.p = (s \circ \mathsf{father}).p))\\ & \rightsquigarrow\\ & \forall p : p \in \mathbb{P} : done.p \end{array}$$

$\Leftarrow$($\rightsquigarrow$ Substitution ($4.6.3_{50}$), using Theorem $9.1.33_{177}$)[3]

$$\begin{array}{ll} \vdash & (\forall p \in \mathbb{P} : \neg\mathsf{idle}.p)\\ & \wedge (\forall p \in \mathbb{P} : sent\_to\_all\_non\_fathers.p)\\ & \wedge (\forall p \in \mathbb{P} : rec\_from\_all\_non\_children.p)\\ & \wedge (\forall p \in \mathbb{P} : (\lambda s.\ \mathsf{f}.p = (s \circ \mathsf{father}).p))\\ & \wedge (\lambda s.\ \mathsf{RST}.\mathbb{P}.\mathsf{f}.starter.\mathsf{neighs})\\ & \rightsquigarrow\\ & \forall p : p \in \mathbb{P} : done.p \end{array}$$

$\Leftarrow$($\rightsquigarrow$ Stable Shift ($4.6.10_{50}$))

[3]Note that RST is *not* a state-predicate. We have `State`-lifted it by enclosing it in between ($\lambda s.\ \ldots$).

$$
\begin{array}{lll}
(\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) & & \\
\wedge\ (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) & & \\
\wedge\ (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) & & \\
\wedge\ (\forall p \in \mathbb{P} : (\lambda s.\ \mathsf{f}.p = (s \circ \mathsf{father}).p)) & & \\
\wedge\ (\lambda s.\ \mathsf{RST}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}) & \vdash & \mathsf{true} \\
 & & \rightsquigarrow \\
 & & \forall p : p \in \mathbb{P} : \mathit{done}.p
\end{array}
$$

Before continuing with this proof obligation, it shall be clear that we need to do something about its readability. For this we introduce the following definition, which contains all conjuncts located at the left hand side of $\vdash$ (including J_{PLUM}, which is there implicitly (Section 9.1.3)). We call it J_{ana} since it refers to properties that were established during the anamorphism part.

Definition 9.1.34 *Invar_and_ANA*

$$
\begin{array}{lll}
J_{\mathsf{ana}} & = & J_{\text{PLUM}} \\
 & & \wedge\ (\forall p \in \mathbb{P} : \neg \mathsf{idle}.p) \\
 & & \wedge\ (\forall p \in \mathbb{P} : \mathit{sent\_to\_all\_non\_fathers}.p) \\
 & & \wedge\ (\forall p \in \mathbb{P} : \mathit{rec\_from\_all\_non\_children}.p) \\
 & & \wedge\ (\forall p \in \mathbb{P} : (\lambda s.\ \mathsf{f}.p = (s \circ \mathsf{father}).p)) \\
 & & \wedge\ (\lambda s.\ \mathsf{RST}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs})
\end{array}
$$

◀

Using $\circlearrowleft$ Conjunction ($4.4.4_{44}$), $9.1.15_{165}$, $9.1.24_{171}$, $9.1.21_{169}$, and the assumed validity of J_{PLUM}, we can derive:

Theorem 9.1.35 *STABLe_Invar_and_ANA*

$$
{}_{\text{PLUM}}\vdash \circlearrowleft J_{\mathsf{ana}}
$$

◀

This reduces our current proof obligation to:

$$
J_{\mathsf{ana}} \vdash \mathsf{true} \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p
$$

Now we can proceed with the proof strategy presented in Section 9.1.5; that is prove that the required information flows from the leaves to the root of the rooted spanning tree. In the case of proving termination this comes down to proving that when the leaves of the RST are *done*, then eventually all the processes will be *done*. From Theorem $9.1.32_{175}$ we can deduce the height h of the RST, and consequently we know that the leaves of the RST equal the processes at level h. Therefore we decompose our proof obligation as follows:

$$
\begin{array}{l}
\quad J_{\mathsf{ana}} \vdash \mathsf{true} \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p \\
\Leftarrow (\rightsquigarrow \text{Substitution } (4.6.3_{50}), \text{ Definition } 9.1.34_{178}, \text{ and Theorem } 9.1.32_{175}) \\
\quad J_{\mathsf{ana}} \vdash (\exists h.\mathsf{height}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}.h) \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p \\
\Leftarrow (\rightsquigarrow \text{Disjunction } (4.5.18_{49}))
\end{array}
$$

$$
\begin{array}{l}
\quad \forall h:\ J_{\mathsf{ana}} \vdash \mathsf{height}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}.h \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p \\
\Leftarrow (\rightsquigarrow \textsc{Transitivity}\ (4.6.6_{50})) \\
\quad \underbrace{\forall h:\ J_{\mathsf{ana}} \vdash \mathsf{height}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}.h \rightsquigarrow \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p}_{\textbf{cata\_1}} \\
\quad \wedge \\
\quad \underbrace{\forall h:\ J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p}_{\textbf{cata\_2}}
\end{array}
$$

Verification of cata_1

Since leaves have no descendants (i.e. children), and J_{ana} states that:

- all processes have received messages from all their non-child-neighbours
- all processes have sent messages to all their non-father-neighbours

we can prove that the leaves (i.e. the processes at $\mathsf{level}\ h$ in a RST of height h) have finished their collecting and propagating phases:

Theorem 9.1.36 *height_Invar_IMP_leaves_finished*

$$\frac{J_{\mathsf{ana}} \wedge \mathsf{height}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}.h}{\forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{finished\_collecting\_and\_propagating}.p}$$

◀

Consequently, we can proceed with **cata_1** as follows:

$$
\begin{array}{l}
\quad \forall h:\ J_{\mathsf{ana}} \vdash \mathsf{height}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}.h \rightsquigarrow \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p \\
\Leftarrow (\rightsquigarrow \textsc{Substitution}\ (4.6.3_{50}), \text{using Theorem } 9.1.36_{179}) \\
\quad \forall h:\ J_{\mathsf{ana}} \vdash \quad \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{finished\_collecting\_and\_propagating}.p \\
\qquad\qquad\qquad\quad \rightsquigarrow \\
\qquad\qquad\qquad\quad \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p \\
\Leftarrow (\rightsquigarrow \textsc{Conjunction}\ (4.5.19_{49})) \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \\
\quad\ J_{\mathsf{ana}} \vdash \mathit{finished\_collecting\_and\_propagating}.p \ \rightsquigarrow\ \mathit{done}.p
\end{array}
$$

Since the *rec_from_all_neighs* part of *done* (see Definition $8.7.7_{136}$) was already established by the validity of *finished_collecting_and_propagating* (see Definition $8.7.4_{136}$), we continue as follows:

$$
\begin{array}{l}
\Leftarrow (\text{Definition } 8.7.7_{136} \text{ and Definition } 8.7.4_{136}) \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \\
\quad\ J_{\mathsf{ana}} \vdash \quad \mathit{rec\_from\_all\_neighs}.p \ \wedge\ \mathit{finished\_collecting\_and\_propagating}.p \\
\qquad\qquad\qquad\quad \rightsquigarrow \\
\qquad\qquad\qquad\quad \mathit{rec\_from\_all\_neighs}.p \ \wedge\ \mathit{sent\_to\_all\_neighs}.p \\
\Leftarrow (\rightsquigarrow \textsc{Conjunction}\ (4.5.19_{49})) \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \\
\quad\ J_{\mathsf{ana}} \vdash \mathit{rec\_from\_all\_neighs}.p \rightsquigarrow \mathit{rec\_from\_all\_neighs}.p \\
\quad \wedge
\end{array}
$$

$$J_{\mathsf{ana}} \vdash \mathit{finished\_collecting\_and\_propagating.p} \rightsquigarrow \mathit{sent\_to\_all\_neighs.p}$$

The first conjunct can easily be proved by $\rightsquigarrow$ Reflexivity ($4.6.5_{50}$), $\circlearrowleft$ Conjunctivity ($4.4.4_{44}$), and Theorem $9.1.24_{171}$.

For the second conjunct, we argue as follows. When a *follower* process has finished its collecting and propagating phase, it is ready to sent its final message to its father after which it becomes *done* and hence has *sent_to_all_neighs*. However, when the *starter* has *finished_collecting_and_propagating*, and hence *sent_to_all_non_fathers*, it has already *sent_to_all_neighs*, since cJ^6_{PLUM} states that the father of the *starter* is the *starter* itself; and the definition of Network (Definition $8.2.1_{121}$) defines that a process cannot be a neighbour of itself.

Theorem 9.1.37 *sent_2_all_except_f_starter_IMP_sent_2_all_neighs_starter*

$$\frac{J_{\mathrm{PLUM}} \wedge \mathit{sent\_to\_all\_non\_fathers.starter}}{\mathit{sent\_to\_all\_neighs.starter}}$$

◀

Consequently, we make the following case distinction: (note that this is a case distinction on the outermost level, *not* inside $\vdash$ using $\rightsquigarrow$ Case Distinction ($4.6.7_{50}$))

$$\begin{array}{l}
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter.h}) : \\
\quad\quad J_{\mathsf{ana}} \vdash \mathit{finished_collecting_and_propagating.p} \rightsquigarrow \mathit{sent_to_all_neighs.p} \\
\Leftarrow((p = \mathit{starter}) \vee (p \neq \mathit{starter})) \\
\quad\quad J_{\mathsf{ana}} \vdash \mathit{finished_collecting_and_propagating.starter} \rightsquigarrow \mathit{sent_to_all_neighs.starter} \\
\quad \wedge \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter.h}), p \neq \mathit{starter} : \\
\quad\quad J_{\mathsf{ana}} \vdash \mathit{finished_collecting_and_propagating.p} \rightsquigarrow \mathit{sent_to_all_neighs.p}
\end{array}$$

Evidently, the first conjunct can be proved by $\rightsquigarrow$ Introduction ($4.6.4_{50}$), using Theorem $9.1.37_{180}$, Theorem $9.1.15_{165}$, and Definition $8.7.4_{136}$. We carry on with the second conjunct by noticing that when a process has *finished_collecting_and_propagating*, it has already sent a message to its father or not.

$$\begin{array}{l}
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter.h}), p \neq \mathit{starter} : \\
\quad\quad J_{\mathsf{ana}} \vdash \mathit{finished_collecting_and_propagating.p} \rightsquigarrow \mathit{sent_to_all_neighs.p} \\
\Leftarrow(\rightsquigarrow \text{Case Distinction } (4.6.7_{50})) \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter.h}), p \neq \mathit{starter} : \\
\quad\quad J_{\mathsf{ana}} \vdash \quad \mathit{finished_collecting_and_propagating.p} \wedge \mathit{reported_to_father.p} \\
\quad\quad\quad\quad\quad\quad \rightsquigarrow \\
\quad\quad\quad\quad\quad\quad \mathit{sent_to_all_neighs.p} \\
\quad \wedge \\
\quad\quad J_{\mathsf{ana}} \vdash \quad \mathit{finished_collecting_and_propagating.p} \wedge \neg \mathit{reported_to_father.p} \\
\quad\quad\quad\quad\quad\quad \rightsquigarrow \\
\quad\quad\quad\quad\quad\quad \mathit{sent_to_all_neighs.p}
\end{array}$$

The first conjunct can again be easily proved by $\rightsquigarrow$ INTRODUCTION ($4.6.4_{50}$), using Theorem $9.1.12_{161}$, and Theorem $9.1.15_{165}$.

Progress stated in the second conjunct is ensured by the DONE action of process p. Consequently:

$$
\begin{array}{l}
\forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h), p \neq \mathit{starter}: \\
\quad J_{\mathsf{ana}} \vdash \quad \mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p \\
\qquad\qquad \rightsquigarrow \\
\qquad\qquad \mathit{sent\_to\_all\_neighs}.p \\
\Leftarrow (\rightsquigarrow \text{SUBSTITUTION } (4.6.3_{50}), \text{ to recognise guard of DONE}) \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h), p \neq \mathit{starter}: \\
\quad J_{\mathsf{ana}} \vdash \quad \exists q \in \mathsf{neighs}.p: \quad \mathit{finished\_collecting\_and\_propagating}.p \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \neg \mathit{reported\_to\_father}.p \ \wedge (q = \mathsf{father}.p) \\
\qquad\qquad \rightsquigarrow \\
\qquad\qquad \exists q \in \mathsf{neighs}.p: \quad \mathit{sent\_to\_all\_neighs}.p \\
\Leftarrow (\rightsquigarrow \text{DISJUNCTION } (4.5.18_{49})) \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h), p \neq \mathit{starter}, q \in \mathsf{neighs}.p: \\
\quad J_{\mathsf{ana}} \vdash \quad \mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p \\
\qquad\qquad \wedge (q = \mathsf{father}.p) \\
\qquad\qquad \rightsquigarrow \\
\qquad\qquad \mathit{sent\_to\_all\_neighs}.p \\
\Leftarrow (\rightsquigarrow \text{INTRODUCTION } (4.6.4_{50}), \text{ Theorem } 9.1.15_{165}) \\
\quad \forall h, p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h), p \neq \mathit{starter}, q \in \mathsf{neighs}.p: \\
\quad \vdash \quad J_{\mathsf{ana}} \wedge \mathit{finished\_collecting\_and\_propagating}.p \wedge \neg \mathit{reported\_to\_father}.p \\
\qquad\qquad \wedge (q = \mathsf{father}.p) \\
\qquad\qquad \mathsf{ensures} \\
\qquad\qquad \mathit{sent\_to\_all\_neighs}.p
\end{array}
$$

As the reader can verify, this ensures -property can be easily proved. This ends the verification of:

Theorem 9.1.38 *finished_collecting_and_propagating_CON_done*

$$
\begin{array}{lll}
\forall h: & J_{\mathsf{ana}} \vdash & \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h): \mathit{finished\_collecting\_and\_propagating}.p \\
& & \rightsquigarrow \\
& & \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h): \mathit{done}.p
\end{array}
$$

◀

---

and consequently, of **cata_1**:

Theorem 9.1.39 cata_1 *height_h_CON_all_done_at_height_h*

$$\forall h: \ J_{\mathsf{ana}} \vdash \mathsf{height}.\mathbb{P}.\mathsf{f}.\mathit{starter}.\mathsf{neighs}.h \rightsquigarrow \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h): \mathit{done}.p$$

◀

Verification of cata_2

The proof of **cata_2** proceeds by induction on h.

INDUCTION BASE: case 0

$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.0) : \mathit{done}.p \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p$$

INDUCTION HYPOTHESIS:

$$\forall h : \ J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p$$

INDUCTION STEP: case $(h+1)$

$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)) : \mathit{done}.p \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p$$

proof of INDUCTION BASE

Since the only process residing at $\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.0$ is the starter, and the *starter* can only be *done* when all other processes are *done*, the INDUCTION BASE can be proved by $\rightsquigarrow$ INTRODUCTION $(4.6.4_{50})$ as follows:

$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.0) : \mathit{done}.p \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p$
$\Leftarrow$($\rightsquigarrow$ SUBSTITUTION $(4.6.3_{50})$, Definition $9.1.30_{174}$)
$J_{\mathsf{ana}} \vdash \mathit{done}.\mathit{starter} \rightsquigarrow \forall p \in \mathbb{P} : \mathit{done}.p$
$\Leftarrow$($\rightsquigarrow$ INTRODUCTION $(4.6.4_{50})$)
$\vdash \circlearrowleft (J_{\mathsf{ana}} \wedge \forall p \in \mathbb{P} : \mathit{done}.p)$
$\wedge$
$\forall s \in \mathtt{State}.J_{\mathsf{ana}}.s \wedge \mathit{done}.\mathit{starter}.s \Rightarrow \forall p \in \mathbb{P} : \mathit{done}.p.s$

The stability predicate can be proved by $\circlearrowleft$ CONJUNCTION $(4.4.4_{44})$, using Theorem $9.1.15_{165}$, Theorem $9.1.24_{171}$, and $9.1.35_{178}$. To prove the second conjunct, assume for arbitrary states s:
$\mathbf{A}_1 : \ J_{\mathsf{ana}}.s$
$\mathbf{A}_2 : \ \mathit{done}.\mathit{starter}.s$
$\mathbf{A}_3 : \ p \in \mathbb{P}$
We prove $\mathit{done}.p.s$ by contradiction, by assuming that:
$\mathbf{A}_4 : \ \neg \mathit{done}.p.s$

and proving that $\neg \mathit{done}.\mathit{starter}.s$, which establishes false with $\mathbf{A}_2$.

The proof strategy will be the following. Since process p is not *done*, we know that is has not yet sent a message to its father. Consequently, p's father has not yet received a message from p, and hence cannot be done. Iterating this argument until the father of the process under consideration is the *starter*, will establish the proof.

However, in order to apply this strategy, we shall have to introduce two new invariant-candidates since, as the reader can verify, the ones introduced until now do

not suffice. We propose:

$$\text{cJ}^{9}_{\text{PLUM}} = \forall p, q \in \mathbb{P} : \neg(\mathsf{idle}.p) \wedge \neg \mathit{done}.p \wedge (q = \mathsf{father}.p) \;\; \Rightarrow \mathsf{nr\_sent}.p.q = 0$$

So we can deduce that when a process p is not done, it has not yet sent a message to its father. Furthermore, we propose the invariant-candidate that states that the number of messages a process q has received from p is always less than or equal to the number of messages p has sent to q:

$$\text{cJ}^{10}_{\text{PLUM}} = \forall p, q \in \mathbb{P} : \mathsf{nr\_rec}.q.p \leq \mathsf{nr\_sent}.p.q$$

So we can deduce that when p has not yet sent a message to some neighbour q, q has not yet received a message from p. When a process q still has neighbours p from which it has not received a message (i.e. it holds that $\mathsf{nr\_rec}.q.p = 0$), we can prove (using $\text{cJ}^{5}_{\text{PLUM}}$) that q has not *rec_from_all_neighs* and hence is not *done*. Consequently, equipped with the new invariant-candidates proposed above, we can now prove that when p is not *done*, neither is its father:

Theorem 9.1.40 *not_done_IMP_f_not_done*

For all states $s \in \texttt{State}$:

$$\frac{J_{\text{PLUM}}.s \wedge p \in \mathbb{P} \wedge \neg s.(\mathsf{idle}.p) \wedge \neg \mathit{done}.p.s \wedge (q = (s \circ \mathsf{father}).p)}{\neg \mathit{done}.q.s}$$

◀

Subsequently, by induction we can prove that:

Theorem 9.1.41 *not_done_IMP_iterate_f_not_done*

For all states $s \in \texttt{State}$:

$$\frac{J_{\text{PLUM}}.s \wedge p \in \mathbb{P} \wedge \neg s.(\mathsf{idle}.p) \wedge \neg \mathit{done}.p.s}{\forall m, q : (q = \mathsf{iterate}.m.(s \circ \mathsf{father}).p) \Rightarrow \neg \mathit{done}.q.s}$$

◀

Consequently, using invariant-part $\text{cJ}^{8}_{\text{PLUM}}$ we can prove that:

Theorem 9.1.42 *not_done_IMP_starter_not_done*

For all states $s \in \texttt{State}$:

$$\frac{J_{\text{PLUM}}.s \wedge p \in \mathbb{P} \wedge \neg s.(\mathsf{idle}.p) \wedge \neg \mathit{done}.p.s}{\neg \mathit{done}.\mathit{starter}.s}$$

◀

Assumptions $\mathbf{A}_1$, $\mathbf{A}_3$, $\mathbf{A}_4$, Theorem 9.1.42$_{183}$, and the characterisation of J_{ana} (Definition 9.1.34$_{178}$) now establish that $\neg \mathit{done}.\mathit{starter}.s$.

end of proof Induction Base

proof of INDUCTION STEP

$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)) : \mathit{done}.p \leadsto \forall p \in \mathbb{P} : \mathit{done}.p$$
$\Leftarrow$($\leadsto$ TRANSITIVITY ($4.6.6_{50}$), and INDUCTION HYPOTHESIS)
$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)) : \mathit{done}.p \leadsto \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p$$

The intuitive idea behind the proof strategy for this last proof obligation is the following: because processes at $\mathsf{level}\ (h+1)$ are done, these have sent messages to their fathers who all reside at $\mathsf{level}\ h$; eventually all processes at $\mathsf{level}\ h$ shall receive these messages and (since already having *sent_to_all_non_fathers* and *rec_from_all_non_children* (J_{ana})) will have *finished_collecting_and_propagating*; consequently, all processes at $\mathsf{level}\ h$ will eventually send a message to their father and become *done*.

$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)) : \mathit{done}.p \leadsto \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p$$
$\Leftarrow$($\leadsto$ TRANSITIVITY ($4.6.6_{50}$))
$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)) : \mathit{done}.p \leadsto \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{finished\_collecting\_and\_propagating}.p$$
$$\wedge$$
$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{finished\_collecting\_and\_propagating}.p \leadsto \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{done}.p$$
$\Leftarrow$(The second conjunct is proved by Theorem $9.1.38_{181}$)
$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)) : \mathit{done}.p \leadsto \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.h) : \mathit{finished\_collecting\_and\_propagating}.p$$
$\Leftarrow$($\leadsto$ SUBSTITUTION ($4.6.3_{50}$), and Definitions $8.7.4_{136}$, $8.7.7_{136}$, $9.1.34_{178}$, $9.1.30_{174}$)
$$J_{\mathsf{ana}} \vdash \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)), q \in \mathsf{neighs}.p : \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \leadsto \forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)), q \in \mathsf{neighs}.p : \mathsf{nr\_rec}.q.p = 1$$
$\Leftarrow$($\leadsto$ CONJUNCTION ($4.5.19_{49}$), twice)
$$\forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)), q \in \mathsf{neighs}.p :$$
$$J_{\mathsf{ana}} \vdash \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \leadsto \mathsf{nr\_rec}.q.p = 1$$
$\Leftarrow$($\leadsto$ STABLE STRENGTHENING ($4.6.9_{50}$), Definition $9.1.34_{178}$, and Theorem $9.1.35_{178}$)
$$\forall p \in (\mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)), q \in \mathsf{neighs}.p :$$
$$J_{\text{PLUM}} \vdash \mathsf{nr\_sent}.p.q = 1 \wedge \neg\mathsf{idle}.q \leadsto \mathsf{nr\_rec}.q.p = 1$$

Since $p \in \mathsf{level}.\mathbb{P}.\mathsf{f}.\mathit{starter}.(h+1)$, implies $p \in \mathbb{P}$, Theorem $9.1.25_{172}$ establishes the INDUCTION STEP.

end of proof INDUCTION STEP

$$cJ^1_{\text{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \neg\mathsf{idle}.p \wedge q = \mathsf{father}.p \Rightarrow \neg\mathsf{idle}.q$$

$$cJ^2_{\text{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathsf{nr\_sent}.p.q = 0 \ \vee \ \mathsf{nr\_sent}.p.q = 1$$

$$cJ^3_{\text{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathsf{idle}.p \Rightarrow \mathsf{nr\_rec}.p.q = 0$$

$$cJ^4_{\text{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : (\mathsf{nr\_rec}.q.p < \mathsf{nr\_sent}.p.q) = \mathsf{mit}.p.q$$

$$cJ^5_{\text{PLUM}} = \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : (\mathsf{nr\_rec}.p.q = 0) \vee (\mathsf{nr\_rec}.p.q = 1)$$

$$cJ^6_{\text{PLUM}} = (\lambda s.\ (s \circ \mathsf{father}).\mathit{starter} = \mathit{starter} \ \wedge \ \neg s.(\mathsf{idle}.\mathit{starter}))$$

$$cJ^7_{\text{PLUM}} = (\lambda s.\ \forall p \in \mathbb{P} :\ (p \neq \mathit{starter}) \wedge \neg s.(\mathsf{idle}.p) \Rightarrow ((s \circ \mathsf{father}).p \in \mathsf{neighs}.p))$$

$$cJ^8_{\text{PLUM}} = (\lambda s.\ \forall p \in \mathbb{P} :\ \neg s.(\mathsf{idle}.p) \Rightarrow \exists k :\ \mathsf{depth}.(s \circ \mathsf{father}).\mathit{starter}.p.k)$$

$$cJ^9_{\text{PLUM}} = \forall p, q \in \mathbb{P} : \neg(\mathsf{idle}.p) \wedge \neg\mathit{done}.p \wedge (q = \mathsf{father}.p) \Rightarrow \mathsf{nr\_sent}.p.q = 0)$$

$$cJ^{10}_{\text{PLUM}} = \forall p, q \in \mathbb{P} : \mathsf{nr\_rec}.q.p \leq \mathsf{nr\_sent}.p.q$$

Figure 9.8: Invariant-candidates proposed during refinement and decomposition

9.1.9 Construction of the invariant

As indicated in Section 9.1.1 the invariant J_{PLUM} is constructed such that it implies all the candidates that were proposed during the process of refinement and decomposition. All the proposed candidates are collected in Figure 9.8. Finding the minimal invariant is now like a nice puzzle. In order to solve this puzzle, we shall start by analysing the different candidates. The first thing we notice is that:

$$cJ^2_{\text{PLUM}} \wedge cJ^{10}_{\text{PLUM}} \ \Rightarrow \ cJ^5_{\text{PLUM}}$$

Consequently, aiming for minimality, cJ^5_{PLUM} can be dropped. Subsequently, we shall start verifying the stability of the conjunction of the remaining candidates. That is, we verify that:

$$\vdash \circlearrowleft \quad \begin{array}{l} cJ^1_{\text{PLUM}} \ \wedge \ cJ^2_{\text{PLUM}} \ \wedge \ cJ^3_{\text{PLUM}} \ \wedge \ cJ^4_{\text{PLUM}} \ \wedge \ cJ^6_{\text{PLUM}} \\ \wedge\, cJ^7_{\text{PLUM}} \ \wedge \ cJ^8_{\text{PLUM}} \ \wedge \ cJ^9_{\text{PLUM}} \ \wedge cJ^{10}_{\text{PLUM}} \end{array}$$

During these verification activities, two more invariant-candidates had to be proposed. One, – cJ^{11}_{PLUM} below – had to be introduced to prove the stability of cJ^4_{PLUM}; and another – cJ^{12}_{PLUM} below – was needed in order to prove the stability of cJ^8_{PLUM} and cJ^9_{PLUM}. Since the verification activities are straightforward we shall not describe them here, and just state the two invariant-candidates:

Definition 9.1.43 PLUM's invariant *Invariant_DEF*

$J_{\text{PLUM}} =$

$$
\begin{array}{ll}
\quad \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \neg \mathsf{idle}.p \wedge q = \mathsf{father}.p \Rightarrow \neg \mathsf{idle}.q & cJ^{1}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathsf{nr\_sent}.p.q = 0 \ \vee\ \mathsf{nr\_sent}.p.q = 1 & cJ^{2}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathsf{idle}.p \Rightarrow \mathsf{nr\_rec}.p.q = 0 & cJ^{3}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : (\mathsf{nr\_rec}.q.p < \mathsf{nr\_sent}.p.q) = \mathsf{mit}.p.q & cJ^{4}_{\text{PLUM}} \\
\wedge\ \mathsf{father}.\mathit{starter} = \mathit{starter} \ \wedge\ \neg(\mathsf{idle}.\mathit{starter}) & cJ^{6}_{\text{PLUM}} \\
\wedge\ \forall p \in \mathbb{P} :\ (p \neq \mathit{starter}) \wedge \neg(\mathsf{idle}.p) \Rightarrow (\mathsf{father}.p \in \mathsf{neighs}.p) & cJ^{7}_{\text{PLUM}} \\
\wedge\ (\lambda s.\ \forall p \in \mathbb{P} :\ \neg s.(\mathsf{idle}.p) \Rightarrow \exists k :\ \mathsf{depth}.(s \circ \mathsf{father}).\mathit{starter}.p.k) & cJ^{8}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : \neg(\mathsf{idle}.p) \wedge \neg \mathit{done}.p \wedge (q = \mathsf{father}.p) \Rightarrow \mathsf{nr\_sent}.p.q = 0 & cJ^{9}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : \mathsf{nr\_rec}.q.p \leq \mathsf{nr\_sent}.p.q & cJ^{10}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : \mathsf{M}.p.q = [] \vee (\exists x : \mathsf{M}.p.q = [x]) & cJ^{11}_{\text{PLUM}} \\
\wedge\ \forall p, q \in \mathbb{P} : \mathsf{idle}.p \Rightarrow \mathsf{nr\_sent}.p.q = 0 & cJ^{12}_{\text{PLUM}}
\end{array}
$$

Theorem 9.1.44 *STABLEe_Invariant*

$$_{\text{PLUM}}\vdash \circlearrowleft J_{\text{PLUM}}$$

Theorem 9.1.45 *INVe_Invariant*

$$_{\text{PLUM}}\vdash \Box J_{\text{PLUM}}$$

Figure 9.9: PLUM's invariant

$\curvearrowright\!\!\!\!\!\!\!\!\curvearrowleft\ cJ^{11}_{\text{PLUM}} = \forall p, q \in \mathbb{P} : \mathsf{M}.p.q = [] \vee (\exists x : \mathsf{M}.p.q = [x])$

Stating that, on every communication channel there is no message in transit , or precisely one.

$\curvearrowright\!\!\!\!\!\!\!\!\curvearrowleft\ cJ^{12}_{\text{PLUM}} = \forall p, q \in \mathbb{P} : \mathsf{idle}.p \Rightarrow \mathsf{nr\_sent}.p.q = 0$

Stating that idle processes have not yet sent messages to their neighbours.

Finally, we construct our invariant consisting of the conjunction of: cJ^{1}_{PLUM} through cJ^{12}_{PLUM} with the exception of cJ^{5}_{PLUM}. The resulting definition, together with the theorems stating stability and invariance in PLUM are in Figure 9.9$_{186}$. In the characterisation of J_{PLUM} (Definition 9.1.43$_{186}$), all logical operators, except for those in cJ^{8}_{PLUM}, are overloaded to denote their `State`-lifted versions.

9.2 An intermediate review

Reckoning with the fact that the reader might have lost track as to what we are aiming at, this section shall put the result of the previous section into perspective with respect to Chapter 8. Thus far, we have proved termination, as described in Section 8.10.1, for the specific case where Π equals the PLUM algorithm. Returning to Figure 8.14_{151}: Section 9.1.5 corresponds to the theory HYLO_PLUM; Section 9.1.6 to the theory ANA_PLUM; Section 9.1.7 to the theory RST; Section 9.1.8 to the theory CATA_PLUM; Section 9.1.9 to the theory PLUM_INV.

As explained in Section 8.13, proving termination of PLUM first, results in the most efficient approach to verify the correctness of all distributed hylomorphisms, since termination of the other distributed hylomorphisms, can then be verified using the refinement framework described in Chapter 7. The latter shall be described in the following three sections.

9.3 Proving termination of ECHO

This section shall describe how termination of the ECHO algorithm is proved using the refinements framework from Chapter 7, and the already proved fact that:

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM}\_\text{ECHO},\ J} \text{ECHO}$$

The UNITY specification reads:

Theorem 9.3.1 *HYLO_ECHO*

$$\begin{array}{l}\forall iA,\ h,\ \text{PROP\_mes},\ \text{DONE\_mes} :: \\ \quad J_{\text{PLUM}} \wedge J_{\text{ECHO}}{}_{\ \text{ECHO}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}} \vdash \ \mathbf{ini}(\text{ECHO}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightsquigarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall p : p \in \mathbb{P} : done.p\end{array}$$

◀

where invariant J_{ECHO} captures additional safety properties for ECHO (if any). Again, J_{ECHO} shall, if necessary, be constructed incrementally in a demand-driven way following the conventions described in Section 9.1.1.

Using $\circlearrowleft$PRESERVATION Theorem $7.2.12_{112}$, it is straightforward to derive that J_{PLUM} is also (Theorem $9.1.44_{186}$) a stable predicate in ECHO.

Theorem 9.3.2 *STABLEe_Invariant_in_ECHO*

$$_{\text{ECHO}}\vdash \circlearrowleft J_{\text{PLUM}}$$

◀

The stability of: $_{\text{ECHO}}\vdash\circlearrowleft J_{\text{PLUM}} \wedge J_{\text{ECHO}}$ will be implicitly assumed throughout the verification process, and verified when the precise characterisation of J_{ECHO} has been established. For ease of reference, Figure 9.10 displays theorems about the guards of

Theorem 9.3.3 *guard_of_IDLE_ECHO*

$\mathsf{guard\_of}.(\text{IDLE}_{\text{ECHO}}.p.q) = \mathsf{guard\_of}.(\text{IDLE}.p.q)$

Theorem 9.3.4 *guard_of_COL_ECHO*

$\mathsf{guard\_of}.(\text{COL}_{\text{ECHO}}.p.q) = \mathsf{guard\_of}.(\text{COL}.p.q) \wedge \mathit{sent\_to\_all\_non\_fathers}.p$

Theorem 9.3.5 *guard_of_PROP_ECHO*

$\mathsf{guard\_of}.(\text{PROP}_{\text{ECHO}}.p.q) = \mathsf{guard\_of}.(\text{PROP}.p.q)$

Theorem 9.3.6 *guard_of_DONE_ECHO*

$\mathsf{guard\_of}.(\text{DONE}_{\text{ECHO}}.p.q) = \mathsf{guard\_of}.(\text{DONE}.p.q)$

Figure 9.10: Guards of the actions from ECHO

ECHO's actions. For readability we introduce the notational convention that:

$\vdash$ and $_{\text{ECHO}}\vdash$ now abbreviate $J_{\text{PLUM}} \wedge J_{\text{ECHO}}\ {}_{\text{ECHO}}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}\vdash$

for arbitrary $iA \in \mathtt{Expr}$, $h \in \mathbb{P}\rightarrow\mathtt{Expr}\rightarrow\mathtt{Expr}$, PROP_mes $\in \mathbb{P}\rightarrow\mathtt{Expr}$, and DONE_mes $\in \mathbb{P}\rightarrow\mathtt{Expr}$, for which hold that:

$\forall p, e : p \in \mathbb{P} \wedge e\ \mathcal{C}\ \mathbf{w}\text{PLUM} : (h.p.e)\ \mathcal{C}\ \mathbf{w}\text{PLUM}$
$\forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p\ \mathcal{C}\ \mathbf{w}\text{PLUM}\ \wedge\ \mathsf{DONE\_mes}.p\ \mathcal{C}\ \mathbf{w}\text{PLUM}$

Note that PLUM's write variables are the same as those of ECHO (see Definition D.2.2$_{255}$).

Again, we implicitly assume the validity of distinct_ECHO_Vars (see Definition D.2.4$_{255}$), ASYNC_type_decl.$\mathbb{P}$.neighs, and Connected_Network.$\mathbb{P}$.neighs.starter.

9.3.1 Using refinements to derive termination of ECHO

As indicated in Chapter 8, termination of ECHO was proved using the property preserving Theorem 7.2.10$_{113}$.

$_{\text{ECHO}}\vdash \mathbf{ini}(\text{ECHO}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \rightsquigarrow \forall p : p \in \mathbb{P} : \mathit{done}.p$

$\Leftarrow$(Theorem 7.2.10$_{113}$, 9.1.1$_{157}$, 8.12.1$_{149}$, and D.2.10$_{256}$))

$$\exists W :: (\mathbf{w}\text{ECHO} = \mathbf{w}\text{PLUM} \cup W) \wedge (J_{\text{PLUM}}\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}\text{PLUM} \subseteq W^{\mathsf{c}})$$

$$\wedge$$

$$\forall A_P\ A_E\ :\ A_P \in \mathbf{a}\text{PLUM}\ \wedge\ A_P\ \mathcal{R}_{\text{-PLUM-ECHO}}\ A_E\ :$$
$$_{\text{ECHO}}\vdash \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_E$$

$$\wedge$$

$$\forall A_P\ A_E\ :\ A_P \in \mathbf{a}\text{PLUM}\ \wedge\ A_P\ \mathcal{R}_{\text{-PLUM-ECHO}}\ A_Q\ :$$
$$_{\text{ECHO}}\vdash (J_{\text{PLUM}} \wedge J_{\text{ECHO}} \wedge \mathsf{guard\_of}.A_E)\ \mathsf{unless}\ \neg(\mathsf{guard\_of}.A_P)$$

Since no variables are superimposed on PLUM in order to construct ECHO, the first conjunct can be proved by instantiation with $\emptyset$. Subsequently, using:

- the characterisation of $\mathcal{R}_{\text{-PLUM-ECHO}}$ (Figure 8.13_{148})
- the Theorems from Figure 9.10_{188}, stating that the guards of the $\text{IDLE}_{\text{ECHO}}$, $\text{PROP}_{\text{ECHO}}$, and $\text{DONE}_{\text{ECHO}}$ actions are equal to those of PLUM
- anti-reflexivity of unless (Theorem $4.4.3_{44}$)
- reflexivity of $\rightarrowtail$ (Theorem $4.5.8_{47}$)
- the implicit assumption stating stability of $(J_{\text{PLUM}} \wedge J_{\text{ECHO}})$

we can reduce the second and the third conjunct to:

$$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :$$
$$_{\text{ECHO}}\vdash \mathsf{guard\_of}.\text{COL}.p.q \rightarrowtail \mathsf{guard\_of}.\text{COL}_{\text{ECHO}}.p.q \quad \Big\} \mathbf{reach-part}$$

$$\wedge$$

$$_{\text{ECHO}}\vdash \begin{array}{l} J_{\text{PLUM}} \wedge J_{\text{ECHO}} \wedge \mathsf{guard\_of}.\text{COL}_{\text{ECHO}}.p.q \\ \mathsf{unless} \\ \neg\mathsf{guard\_of}.\text{COL}.p.q \end{array} \Big\} \mathbf{unless-part}$$

The **unless**-part is not hard to verify and will be left up to the enthusiastic reader. In order to prove it, the current conjuncts from J_{PLUM} suffice, and hence no additional safety properties have to be added to J_{ECHO}.

The proof of the **reach-part** proceeds by rewriting with Theorem $9.1.7_{159}$ and $9.3.4_{188}$:

$$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :$$
$$_{\text{ECHO}}\vdash \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p}$$
$$\rightarrowtail$$
$$\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p} \wedge \mathit{sent\_to\_all\_non\_fathers.p}$$

$\Leftarrow$($\rightarrowtail$ Case distinction ($4.5.10_{47}$))

$$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :$$
$$_{\text{ECHO}}\vdash \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p} \wedge \mathit{sent\_to\_all\_non\_fathers.p}$$
$$\rightarrowtail$$
$$\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p} \wedge \mathit{sent\_to\_all\_non\_fathers.p}$$

$$\wedge$$

$$_{\text{ECHO}}\vdash \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p} \wedge \neg \mathit{sent\_to\_all\_non\_fathers.p}$$
$$\rightarrowtail$$
$$\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs.p} \wedge \mathit{sent\_to\_all\_non\_fathers.p}$$

$\Leftarrow$($\rightarrowtail$ Reflexivity ($4.5.8_{47}$) proves the first conjunct)

$$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p :$$

$$_{\text{ECHO}}\vdash \quad \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg rec\_from\_all\_neighs.p \wedge \neg sent\_to\_all\_non\_fathers.p$$
$$\rightarrowtail$$
$$\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg rec\_from\_all\_neighs.p \wedge sent\_to\_all\_non\_fathers.p$$

$\Leftarrow$($\rightarrowtail$ SUBSTITUTION ($4.5.6_{47}$), to bring into correct form for $\rightarrowtail$ PSP ($4.5.12_{47}$))

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p$:

$$_{\text{ECHO}}\vdash \quad (\neg\mathsf{idle}.p \wedge \neg sent\_to\_all\_non\_fathers.p)$$
$$\wedge$$
$$(\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg rec\_from\_all\_neighs.p)$$
$$\rightarrowtail$$
$$(sent\_to\_all\_non\_fathers.p \wedge (\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg rec\_from\_all\_neighs.p))$$
$$\vee$$
$$(\neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg rec\_from\_all\_neighs.p \wedge sent\_to\_all\_non\_fathers.p)$$

$\Leftarrow$($\rightarrowtail$ PSP ($4.5.12_{47}$))

$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p$:

$$\left.\begin{array}{ll} _{\text{ECHO}}\vdash & J_{\text{PLUM}} \wedge J_{\text{ECHO}} \\ & \wedge\ \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg rec\_from\_all\_neighs.p \\ & \ \mathsf{unless} \\ & \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg rec\_from\_all\_neighs.p \\ & \wedge\ sent\_to\_all\_non\_fathers.p \end{array}\right\}\ \textbf{PSP} - \textbf{unless}$$

$\wedge$

$\forall p \in \mathbb{P}$:

$$\left.\begin{array}{ll} _{\text{ECHO}}\vdash & \neg\mathsf{idle}.p \wedge \neg sent\_to\_all\_non\_fathers.p \\ & \rightarrowtail \\ & sent\_to\_all\_non\_fathers.p \end{array}\right\}\ \textbf{PSP} - \textbf{reach}$$

The proof of the **PSP-unless**-part is not complicated, again the characterisation of J_{PLUM} suffices, and hence no additional safety properties have to be added to J_{ECHO}. Note, that at this point J_{ECHO} can be substituted by true.

We shall proceed with the **PSP-reach**-part. If we look at it closely, we can see that it resembles **ana_2**, a proof obligation we encountered during the verification of termination of PLUM (see pages 163, 167). Obviously, if we can transform the **PSP-reach**-part into a **ana_2**, we can re-use the proof-strategy used to prove **ana_2** in the context of PLUM, to prove the **PSP-reach**-part in the context of ECHO. Since **ana_2**'s proof-strategy uses conjunctivity of $\rightsquigarrow$ (theorem $4.5.19_{49}$), and $\rightarrowtail$ does not have this property, we first replace $\rightarrowtail$ by $\rightsquigarrow$:

$\Leftarrow$($\rightsquigarrow$ CONVERGENCE IMPLIES PROGRESS ($4.6.2_{50}$))

$\forall p \in \mathbb{P}$:

$$_{\text{ECHO}}\vdash \quad \neg\mathsf{idle}.p \wedge \neg sent\_to\_all\_non\_fathers.p$$
$$\rightsquigarrow$$
$$sent\_to\_all\_non\_fathers.p$$

Then, we apply a $\rightsquigarrow$ SUBSTITUTION ($4.6.3_{50}$) step similar to the ♫-marked-substitution step made on page 168 to obtain:

$\forall p \in \mathbb{P}$:

$$_{\text{ECHO}}\vdash \quad \forall q \in \mathsf{neighs}.p : \neg\mathsf{idle}.p \\ \rightarrowtail \\ \forall q \in \mathsf{neighs}.p : (\neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \vee (\mathsf{nr\_sent}.p.q = 1)$$

Subsequently, we apply a conjunction step similar to the ♫-marked-conjunction step made on page 168. Now, our proof obligation has become equal to that of **ana_2** only now in the context of ECHO:

$$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \ _{\text{ECHO}}\vdash \neg\mathsf{idle}.p \rightsquigarrow (\neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p)) \vee (\mathsf{nr\_sent}.p.q = 1)$$

Consequently, the same proof strategy applies. Inspecting **ana_2**'s proof strategy on page 168 this comes down to proving:

Theorem 9.3.7 *STABLEe_not_idle_AND_q_IS_f_p_in_ECHO*

$$\forall p, q \in \mathbb{P} : \ _{\text{ECHO}}\vdash \circlearrowleft(\neg\mathsf{idle}.p \wedge (q = \mathsf{father}.p))$$

---

which is straightforward, using the stability preserving Theorem $7.2.12_{112}$. Moreover, we need an ECHO equivalent for Theorem $9.1.16_{168}$ (i.e. **ana_1.2.1**, page 165). Again, the proof-strategy of **ana_1.2.1** can be re-used. Returning to page 165, we can see this comes down to proving the following two properties. First,

---

Theorem 9.3.8 *STABLEe_nr_sent_is_1_in_ECHO*

$$\forall p, q \in \mathbb{P} : \ _{\text{ECHO}}\vdash \circlearrowleft(\mathsf{nr\_sent}.p.q = 1)$$

---

which again is easy using stability preserving Theorem $7.2.12_{112}$. Second,

$$_{\text{ECHO}}\vdash (J_{\text{PLUM}} \wedge J_{\text{ECHO}} \wedge \neg\mathsf{idle}.p \wedge q \neq \mathsf{father}.p) \ \mathsf{ensures} \ (\mathsf{nr\_sent}.p.q = 1)$$

This last proof obligation can be proved similarly to that of the ensures-part of **ana_1.2.1** (see page 165), and doing so, the unless-part of the ensures-part of **ana_1.2.1** can be inherited by using unless-preserving Theorem $7.2.11_{112}$.

This ends the verification of the **reach-part**. Since no additional safety properties have to be proved for ECHO, we can define J_{ECHO} to be true.

Definition 9.3.9 *Invariant_ECHO*

$$J_{\text{ECHO}} = \mathsf{true}$$

---

since true is trivially stable, this ends verification of termination of ECHO. Although the definition for J_{ECHO} might appear superfluous, we decided to include it for two reasons. The first one being preservation of consistency throughout this chapter. The second reason is that by explicitly defining J_{ECHO} to be true, it immediately becomes

clear that PLUM and ECHO have the same safety properties.

9.4 Proving termination of TARRY

This section shall describe how termination of the TARRY algorithm is proved using the refinements framework from Chapter 7, and the already proven fact that:

$$\forall J :: \text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM}\_\text{TARRY},\ J} \text{TARRY}$$

The UNITY specification reads:

Theorem 9.4.1 *HYLO_Tarry*

$$\forall iA,\ h,\ \text{PROP\_mes},\ \text{DONE\_mes} ::$$
$$J_{\text{PLUM}} \wedge J_{\text{TARRY}}\ \ {}_{\text{TARRY}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}}\vdash\ \ \mathbf{ini}(\text{TARRY}.iA.h.\text{PROP\_mes}.\text{DONE\_mes})$$
$$\rightsquigarrow$$
$$\forall p : p \in \mathbb{P} : done.p$$

---

where invariant J_{TARRY} captures additional safety properties for TARRY. Again, J_{TARRY} shall be constructed incrementally in a demand-driven way following the conventions described in Section 9.1.1.

Using $\circlearrowleft$PRESERVATION Theorem $7.2.12_{112}$, it is straightforward to derive that J_{PLUM} is also (Theorem $9.1.44_{186}$) a stable predicate in TARRY.

Theorem 9.4.2 *STABLEe_Invariant_in_Tarry*

$${}_{\text{TARRY}}\vdash \circlearrowleft J_{\text{PLUM}}$$

The stability of: ${}_{\text{TARRY}}\vdash\circlearrowleft(J_{\text{PLUM}} \wedge J_{\text{TARRY}})$ will be implicitly assumed throughout the verification process, and verified when the precise characterisation of J_{TARRY} has been established. For ease of reference, Figure 9.11 displays theorems about the guards of TARRY's actions. For readability we introduce the notational convention that:

$$\vdash \text{ and } {}_{\text{TARRY}}\vdash \text{ now abbreviate } J_{\text{PLUM}} \wedge J_{\text{TARRY}}\ \ {}_{\text{TARRY}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}}\vdash$$

for arbitrary $iA \in \texttt{Expr}$, $h \in \mathbb{P}\rightarrow\texttt{Expr}\rightarrow\texttt{Expr}$, PROP_mes $\in \mathbb{P}\rightarrow\texttt{Expr}$, and DONE_mes $\in \mathbb{P}\rightarrow\texttt{Expr}$, for which hold that:

$$\forall p, e : p \in \mathbb{P} \wedge e\ \mathcal{C}\ \mathbf{w}\text{PLUM} : (h.p.e)\ \mathcal{C}\ \mathbf{w}\text{PLUM}$$
$$\forall p : p \in \mathbb{P} : \text{PROP\_mes}.p\ \mathcal{C}\ \mathbf{w}\text{PLUM}\ \wedge\ \text{DONE\_mes}.p\ \mathcal{C}\ \mathbf{w}\text{PLUM}$$

Although in order to conclude that TARRY is a well-formed UNITY program (i.e. satisfies the predicate Unity (see Theorem $D.3.10_{258}$)) it would be sufficient to assume confinement by the write variables of TARRY, we need to assume confinement by PLUM's write variables here in order to be able to deduce termination of PLUM

Theorem 9.4.3 *guard_of_IDLE_Tarry*

$\mathsf{guard\_of}.(\text{IDLE}_{\text{TARRY}}.p.q) = \mathsf{guard\_of}.(\text{IDLE}.p.q)$

Theorem 9.4.4 *guard_of_COL_Tarry*

$\mathsf{guard\_of}.(\text{COL}_{\text{TARRY}}.p.q) = \mathsf{guard\_of}.(\text{COL}.p.q) \wedge \neg \mathsf{le\_rec}.p$

Theorem 9.4.5 *guard_of_PROP_Tarry*

$\mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) = \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \mathsf{le\_rec}.p$

Theorem 9.4.6 *guard_of_DONE_Tarry*

$\mathsf{guard\_of}.(\text{DONE}_{\text{TARRY}}.p.q) = \mathsf{guard\_of}.(\text{DONE}.p.q)$

Figure 9.11: Guards of the actions from TARRY

(see implicit assumptions made in Section 9.1.4). Obviously, confinement by PLUM's write variables implies (using Theorem $\mathcal{C}$ MONOTONICITY ($3.3.19_{28}$)) confinement by TARRY's write variables.

Again, we implicitly assume the validity of distinct_Tarry_Vars (see Definition $\text{D}.3.4_{257}$), ASYNC_type_decl.$\mathbb{P}$.neighs, and Connected_Network.$\mathbb{P}$.neighs.starter.

9.4.1 Using refinements to derive termination of TARRY

As indicated in Chapter 8, termination of TARRY is proved using property preserving Theorem $7.2.9_{113}$. The reason for using this theorem is that Theorem $7.2.10_{113}$ – which is easier and hence preferable – cannot be used since its application results in the following, not provable, proof obligation:

$$\begin{array}{ll} {}_{\text{TARRY}}\vdash & J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \\ & \mathsf{unless} \\ & \neg \mathsf{guard\_of}.(\text{PROP}.p.q) \end{array}$$

The reason why this cannot be proved is because, during the execution of TARRY, it is possible that the guard of $\text{PROP}_{\text{TARRY}}.p.q$ is falsified while the guard of $\text{PROP}.p.q$ still holds (see also Section 7.2.3, page 114). For the sake of clarity, we shall elucidate this below. We rewrite the unless-property from above, using Definition $4.4.1_{43}$, Theorem $9.1.8_{159}$ and Theorem $9.4.5_{193}$. (Note that we have omitted evalb and compile):

$$
\begin{array}{l}
\forall A \in \mathbf{a}\textsc{Tarry}, s, t \in \texttt{State} : \\
\quad J_{\textsc{plum}}.s \wedge J_{\textsc{Tarry}}.s \wedge \neg s.(\mathsf{idle}.p) \wedge \neg \mathit{sent\_to\_all\_non\_fathers.p.s} \\
\quad \wedge \mathit{can\_propagate.p.q.s} \wedge s.(\mathsf{le\_rec}.p) \wedge A.s.t \\
\quad \Rightarrow \\
\qquad (J_{\textsc{plum}}.t \wedge J_{\textsc{Tarry}}.t \wedge \neg t.(\mathsf{idle}.p) \wedge \neg \mathit{sent\_to\_all\_non\_fathers.p.t} \\
\qquad \wedge \mathit{can\_propagate.p.q.t} \wedge t.(\mathsf{le\_rec}.p)) \\
\quad \vee \\
\qquad (t.(\mathsf{idle}.p) \vee \mathit{sent\_to\_all\_non\_fathers.p.t} \vee \neg \mathit{can\_propagate.p.q.t})
\end{array}
$$

We have to prove this for arbitrary actions of TARRY. Consider the propagating action $\textsc{prop}_{\textsc{tarry}}.p.q'$, with $(q \neq q')$. Assume for arbitrary states s and t that:
$\mathbf{A}_1$: $J_{\textsc{plum}}.s \wedge J_{\textsc{Tarry}}.s$
$\mathbf{A}_2$: $\neg s.(\mathsf{idle}.p) \wedge \neg \mathit{sent\_to\_all\_non\_fathers.p.s} \wedge \mathit{can\_propagate.p.q.s} \wedge s.(\mathsf{le\_rec}.p)$
$\mathbf{A}_3$: $\textsc{prop}_{\textsc{tarry}}.p.q'.s.t$
$\mathbf{A}_4$: $(q \neq q')$
If p cannot propagate to q' in state s, then $s = t$ and there is no problem in the sense that the conclusion of the implication stated above can be proved. However, suppose p can propagate to q' (i.e. $\mathit{can\_propagate.p.q'.s}$). Then the guard of $\textsc{prop}_{\textsc{tarry}}.p.q'.s.t$ is enabled and execution of this action establishes: $\neg t.(\mathsf{le\_rec}.p)$. Consequently, the guard of $\textsc{prop}_{\textsc{tarry}}.p.q$ is disabled in state t, and in order to prove the conclusion of the implication we have to prove that the guard of $\textsc{prop}.p.q$ is also disabled in state t. That is, we have to prove one of:

$$t.(\mathsf{idle}.p) \vee \mathit{sent\_to\_all\_non\_fathers.p.t} \vee \neg \mathit{can\_propagate.p.q.t}$$

However,

- $t.(\mathsf{idle}.p)$ cannot be proved, since from $\mathbf{A}_2$ we know that p is non-idle in state s, and since PROP-actions do not write to idle-variables we know that p is still non-idle in state t.
- $\neg \mathit{can\_propagate.p.q.t}$ cannot be proved, since from $\mathbf{A}_2$ we know that, in state s, p can propagate to q ($\mathit{can\_propagate.p.q.s}$), and since $(q \neq q')$ we know that p can still propagate to q in state t (i.e. $\mathit{can\_propagate.p.q.t}$).
- $\mathit{sent\_to\_all\_non\_fathers.p.t}$ is not necessarily valid. It can hold in state t, but it might as well be the case that is does not.

Consequently, we cannot prove the unless-property from above. What we need is a function which is non-increasing with respect to some well-founded relation, and which decreases when a message is sent. Since then, we can ensure that this kind of premature falsification of the guard of $\textsc{prop}_{\textsc{tarry}}.p.q$, while the guard of $\textsc{prop}.p.q$ still holds, cannot happen infinitely often.

As an aside: The guards of IDLE and DONE actions in TARRY are equal to those of PLUM (Theorems $9.4.3_{193}$ and $9.4.6_{193}$). Consequently, for these actions, a unless-property similar to the one above can if necessary be proved using unless ANTI-REFLEXIVITY $4.4.3_{44}$.

For the COL-actions, the construction of a non-increasing function is *not* required, since we can, if necessary, prove that when the guard of $\textsc{col}_{\textsc{tarry}}.p.q$ (Theorem $9.4.4_{193}$) is falsified, then so is the guard of $\textsc{col}.p.q$. This is because,

intuitively, TARRY has the additional invariant that there is always at most one message in transit. Therefore, if some action $\text{COL}_{\text{TARRY}}.p.q'$ $(q \neq q')$ receives the message that is in transit from q' to p and as a consequence falsifies the guard of $\text{COL}_{\text{TARRY}}.p.q$ by setting $\mathsf{le\_rec}.p$ to true, then we can prove that afterwards there are no messages at all in transit and hence that the guard of $\text{COL}.p.q$ cannot be true.

So, since the least complicated property preservation Theorem ($7.2.10_{113}$) cannot be used to derive termination of TARRY, we move on to the second least complicated one, i.e. $7.2.9_{113}$. Since the bitotal relation defined on the actions of PLUM and TARRY is one-to-one, this one turns out to be sufficient.

$$_{\text{TARRY}}\vdash \mathbf{ini}(\text{TARRY}.iA.h.\text{PROP\_MES.DONE\_MES}) \rightsquigarrow \forall p : p \in \mathbb{P} : \mathit{done}.p$$

$\Leftarrow$(Theorem $7.2.9_{113}$, $9.1.1_{157}$, $8.12.2_{149}$, and $\text{D}.3.11_{258}$))
For some well-founded relation $\prec$:

$$\begin{array}{l}
\quad \exists W :: (\mathbf{w}\text{TARRY} = \mathbf{w}\text{PLUM} \cup W) \wedge (J_{\text{PLUM}}\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}\text{PLUM} \subseteq W^{\mathsf{c}}) \\
\wedge \\
\quad \left.\begin{array}{l} \forall A_P\ A_T\ :\ A_P \in \mathbf{a}\text{PLUM}\ \wedge\ A_P\ \mathcal{R}_{\text{_PLUM_TARRY}}\ A_T\ : \\ \quad {}_{\text{TARRY}}\vdash \mathsf{guard_of}.A_P \rightarrowtail \mathsf{guard_of}.A_T \end{array}\right\} \mathbf{reach-part} \\
\wedge \\
\quad \left.\begin{array}{l} \exists M :: (M\ \mathcal{C}\ \mathbf{w}\text{TARRY}) \\ \quad \wedge \\ \quad \forall k :: {}_{\text{TARRY}}\vdash (J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge M = k)\ \mathsf{unless}\ (M \prec k) \\ \quad \wedge \\ \quad \forall k\ A_P\ A_T : A_P \in \mathbf{a}\text{PLUM} \wedge A_P\ \mathcal{R}_{\text{_PLUM_TARRY}}\ A_T : \\ \qquad {}_{\text{TARRY}}\vdash \quad (J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge \mathsf{guard_of}.A_T \wedge M = k) \\ \qquad\qquad\quad \mathsf{unless} \\ \qquad\qquad\quad (\neg(\mathsf{guard_of}.A_P) \vee M \prec k) \end{array}\right\} \mathbf{unless-part}
\end{array}$$

Since $\mathsf{le\_rec}.p$ variables are superimposed on PLUM in order to obtain TARRY, the first conjunct is instantiated with the set $\{\mathsf{le\_rec}.p \mid p \in \mathbb{P}\}$. Proving that J_{PLUM} is confined by the complement of this set is tedious but straightforward, since the variables $\mathsf{le\_rec}$ do not appear in it.

Verification of the **unless-part** involves the construction of a function over the variables of TARRY, that is non-increasing with respect to some well-founded relation $\prec$. From the discussion above, we can deduce that we need a function that decreases when a message is sent. However, it turns out that the verification of the **reach-part** involves an application of $\rightarrowtail$ BOUNDED PROGRESS ($4.5.17_{48}$) that needs a function that decreases not only when a message is sent, but also when a message is received. Consequently, we shall continue with the construction of a function over the variables of TARRY, that is non-increasing with respect to some well-founded relation $\prec$, and that decreases when a message is sent as well as received. Obviously, this function can then be used for both purposes.

Construction of a non-increasing function

Constructing a non-increasing function that decreases when a message is sent, and when a message is received is not complicated. Observe the following:

- the sending of a message is always accompanied by incrementing a nr_sent variable
- similarly, receiving a message is always accompanied by incrementing a nr_rec variable
- from J_{PLUM} it follows that at most one message is sent over each directed communication link
- consequently, at most one message is received over each directed communication link
- consequently, the total amount of messages sent and received has an upper-bound, that equals twice the cardinality of the set of directed communication links (see Definition $6.2.1_{74}$)

From these observations a non-increasing function is constructed as follows. First, we define the upper-bound on the total amount of messages sent *and* received.

Definition 9.4.7 *MAX_MAIL*

$$\mathsf{MAX\_MAIL} = 2 \times \mathsf{card}.(\mathsf{links}.\mathbb{P}.\mathsf{neighs})$$

Next, we define the total amount of messages that a process $p \in \mathbb{P}$ has sent, and respectively received, in some state s.

Definition 9.4.8 NUMBER OF MESSAGES SENT BY PROCESSES p *NR_SENT*

$$\mathsf{NR\_SENT}.p.s = \sum_{q \in \mathsf{neighs}.p} s.(\mathsf{nr\_sent}.p.q)$$

Definition 9.4.9 NUMBER OF MESSAGES RECEIVED BY PROCESSES p *NR_REC*

$$\mathsf{NR\_REC}.p.s = \sum_{q \in \mathsf{neighs}.p} s.(\mathsf{nr\_rec}.p.q)$$

The total amount of messages that are sent, and respectively received, in the whole network of processes can be defined as follows:

Theorem 9.4.12 *rec_from_all_p_EQ_NR_REC_EQ_CARD_p*

$$\forall p \in \mathbb{P}, s \in \texttt{State} : \frac{J_{\text{PLUM}}.s}{rec\_from\_all\_neighs.p = (\mathsf{NR\_REC}.p.s = \mathsf{card}.(\mathsf{neighs}.p))}$$

Theorem 9.4.13 *sent_2_all_p_EQ_NR_SENT_EQ_CARD_p*

$$\forall p \in \mathbb{P}, s \in \texttt{State} : \frac{J_{\text{PLUM}}.s}{sent\_to\_all\_neighs.p = (\mathsf{NR\_SENT}.p.s = \mathsf{card}.(\mathsf{neighs}.p))}$$

Theorem 9.4.14 *NR_REC_leq_CARD*

$$\forall p \in \mathbb{P}, s \in \texttt{State} : \frac{J_{\text{PLUM}}.s}{\mathsf{NR\_REC}.p.s \leq \mathsf{card}.(\mathsf{neighs}.p)}$$

Theorem 9.4.15 *NR_REC_SUC_NR_SENT_IMP_not_sent_2_all*

$$\forall p \in \mathbb{P}, s \in \texttt{State} : \frac{J_{\text{PLUM}}.s \wedge (\mathsf{NR\_REC}.p.s = \mathsf{NR\_SENT}.p.s + 1)}{\neg sent\_to\_all\_neighs.p}$$

Theorem 9.4.16 *sent_2_all_except_f_IMP_SUC_NR_SENT_EQ_CARD*

$$\forall p \in \mathbb{P}, s \in \texttt{State} : \frac{J_{\text{PLUM}}.s \wedge sent\_to\_all\_non\_fathers.p.s \wedge \neg sent\_to\_all\_neighs.p.s}{\mathsf{NR\_SENT}.p.s + 1 = \mathsf{card}.(\mathsf{neighs}.p)}$$

Figure 9.12: Some properties of NR_REC and NR_SENT

Definition 9.4.10 TOTAL NUMBER OF MESSAGES SENT IN THE NETWORK *TOTAL_NR_SENT*

$$\mathsf{TOTAL\_NR\_SENT}.s = \sum_{p \in \mathbb{P}} \mathsf{NR\_SENT}.p.s$$

Definition 9.4.11 TOTAL NUMBER OF MESSAGES RECEIVED IN THE NETWORK *TOTAL_NR_REC*

$$\mathsf{TOTAL\_NR\_REC}.s = \sum_{p \in \mathbb{P}} \mathsf{NR\_REC}.p.s$$

Finally, we define our non-increasing function as follows:

Definition 9.4.17 NON-INCREASING FUNCTION OVER THE VARIABLES OF TARRY *Y_DEF*

$$Y.s = \mathsf{MAX\_MAIL} - (\mathsf{TOTAL\_NR\_SENT}.s + \mathsf{TOTAL\_NR\_REC}.s)$$

The value of Y only depends on the variables nr_rec and nr_sent. Since these are write variables of TARRY is it easy to verify that:

Theorem 9.4.18 *CONF_Y_Write_Vars_Tarry*

$$Y \; \mathcal{C} \; \mathbf{w}\text{TARRY}$$

The following lemma states that whenever a message is sent or received – because the guard of one of TARRY's actions is enabled – the value of Y decreases.

Lemma 9.4.19 *A_DECR_Y*

For arbitrary processes $p \in \mathbb{P}$, $q \in \mathsf{neighs}.p$, and actions A;
$A \in \{\text{IDLE}_{\text{TARRY}}, \text{COL}_{\text{TARRY}}, \text{PROP}_{\text{TARRY}}, \text{DONE}_{\text{TARRY}}\}$:

$$\forall k :: \frac{J_{\text{PLUM}}.s \wedge A.p.q.s.t \wedge \mathsf{guard\_of}.(A.p.q).s \wedge (Y.s = k)}{Y.t < k}$$

Using this lemma, it is straightforward to prove that, during the execution of TARRY, Y is non-increasing with respect to the well-founded relation $<$ on numerals.

Theorem 9.4.20 *DECREASING_DECR_FUNCTION*

For arbitrary characterisations of J_{TARRY}:

$$\forall k :: {}_{\text{TARRY}}\vdash (J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge Y = k) \; \mathsf{unless} \; (Y < k)$$

Verification of the unless-part

Return to page 195 for the **unless-part**. Instantiating this proof obligation with Y, and rewriting with Theorems $9.4.18_{198}$ and $9.4.20_{198}$ results in the following proof obligation:

$$\begin{array}{l} \forall k \; A_P \; A_T : A_P \in \mathbf{a}\text{PLUM} \wedge A_P \; \mathcal{R}_{\text{-PLUM\_TARRY}} \; A_T : \\ \quad {}_{\text{TARRY}}\vdash \quad (J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge \mathsf{guard\_of}.A_T \wedge Y = k) \\ \qquad\quad \mathsf{unless} \\ \qquad\quad (\neg(\mathsf{guard\_of}.A_P) \vee Y < k) \end{array}$$

Proving this is straightforward using the characterisation of $\mathcal{R}_{\text{-PLUM\_TARRY}}$ from Figure 8.13_{148}, and Lemma $9.4.19_{198}$. Note that, since Y is constructed as to decrease when a message is sent as well as when a message is received, we do not have to use the proof strategy delineated in the aside on page 194 for the COL actions. Consequently, constructing a non-increasing function that decreases upon the sending as well as upon receiving of a message is not only more efficient since it is re-usable in the proof of the **reach-part**, it also simplifies the verification of the **unless-part**.

Verification of the reach-part

We shall now continue with the **reach-part**, which is re-displayed below for convenience.

$$\forall A_P\ A_T\ :\ A_P \in \mathbf{a}\text{PLUM}\ \wedge\ A_P\ \mathcal{R}_{\text{-PLUM-TARRY}}\ A_T\ :\\ \quad {}_{\text{TARRY}}\vdash \mathsf{guard\_of}.A_P \rightarrowtail \mathsf{guard\_of}.A_T$$

Subsequently, using:

- the characterisation of $\mathcal{R}_{\text{-PLUM-TARRY}}$ (Figure 8.13_{148})
- Theorems $9.4.3_{193}$ and $9.4.6_{193}$, stating that the guards of the $\text{IDLE}_{\text{TARRY}}$, and $\text{DONE}_{\text{TARRY}}$ actions are equal to those of PLUM
- reflexivity of $\rightarrowtail$ (Theorem $4.5.8_{47}$)
- the implicit assumption stating stability of $(J_{\text{PLUM}} \wedge J_{\text{TARRY}})$

we reduce the **reach-part** for arbitrary $p \in \mathbb{P}$ and $q \in \mathsf{neighs}.p$, as follows:

$$\left.\begin{array}{l} {}_{\text{TARRY}}\vdash \mathsf{guard\_of}.(\text{COL}.p.q) \rightarrowtail \mathsf{guard\_of}.(\text{COL}_{\text{TARRY}}.p.q) \end{array}\right\} \mathbf{reach} - \text{COL} - \mathbf{part}$$
$$\wedge$$
$$\left.\begin{array}{l} {}_{\text{TARRY}}\vdash \mathsf{guard\_of}.(\text{PROP}.p.q) \rightarrowtail \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \end{array}\right\} \mathbf{reach} - \text{PROP} - \mathbf{part}$$

Verification of reach-COL-part

Rewriting with the characterisations of the guards (Theorem $9.1.7_{159}$ and $9.4.4_{193}$) gives:

$$\begin{array}{ll} {}_{\text{TARRY}}\vdash & \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs}.p \\ & \rightarrowtail \\ & \neg\mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge \neg \mathit{rec\_from\_all\_neighs}.p \wedge \neg\mathsf{le\_rec}.p \end{array}$$

Due to the alternating sending and receiving of messages, which is inherent to TARRY, we know that it must be provable that there is always at most one message in transit during the execution of TARRY's algorithm. This means that if there is a message in transit, it is the only one, and hence the event last executed by all processes was a send-event and thus not a receive-event. Consequently, the above proof obligation must be provable from the invariant, by using $\rightarrowtail$ INTRODUCTION ($4.5.7_{47}$). In order to establish this we propose the following invariant-candidate:

$$\text{cJ}^1_{\text{TARRY}} = (\exists p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathit{mit}.p.q) \Rightarrow (\forall p \in \mathbb{P} : \neg\mathsf{le\_rec}.p)$$

which, evidently, suffices to establish the **reach-COL-part**.

Verification of reach-PROP-part

Rewriting with the characterisations of $\text{PROP}_{\text{TARRY}}$'s the guard ($9.4.5_{193}$) gives:

$$ {}_{\text{TARRY}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}.p.q) \rightarrowtail \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \mathsf{le\_rec}.p$$

If p's last event was a receive event this is easy to prove:

$\Leftarrow(\rightarrowtail$ Case Distinction $(4.5.10_{47})$, p's last event was a receive event or not)
$\quad$ $_{\text{Tarry}}\vdash$ $\mathsf{guard\_of}.(\text{prop}.p.q) \wedge \mathsf{le\_rec}.p \rightarrowtail \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \mathsf{le\_rec}.p$
$\wedge$
$\quad$ $_{\text{Tarry}}\vdash$ $\mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg\mathsf{le\_rec}.p \rightarrowtail \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \mathsf{le\_rec}.p$
$\Leftarrow(\rightarrowtail$ Reflexivity $(4.5.8_{47})$, proves the first conjunct)
$\quad$ $_{\text{Tarry}}\vdash$ $\mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg\mathsf{le\_rec}.p \rightarrowtail \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \mathsf{le\_rec}.p$

To explain the proof-strategy that is used to verify the conjunct from above, we refer to Figure 9.13_{201}. The p and q in this figure correspond to the p and q in the current proof-obligation, x, y, z, and w are arbitrary processes. We already indicated that, during an execution of Tarry's algorithm, there is always at most one message in transit. This message is indicated with a $\bullet$ in Figure 9.13. In Figure 9.13(b), this message is in transit from w to z, and hence from invariant-candidate cJ^1_{Tarry} we can infer that $\forall p \in \mathbb{P} : \neg\mathsf{le\_rec}.p$ In 9.13(a) this message has just been received by x, and hence we can infer that $\mathsf{le\_rec}.x$. In order to establish our current proof obligation, we need to invent a proof strategy that enables us to prove that this message shall eventually reach p such that the latter can set $\mathsf{le\_rec}.p$ to true. Suppose that $\mathsf{guard\_of}.(\text{prop}.p.q)$ holds, and that the last event of p was *not* a receive event. Using Theorem $9.1.8_{159}$):

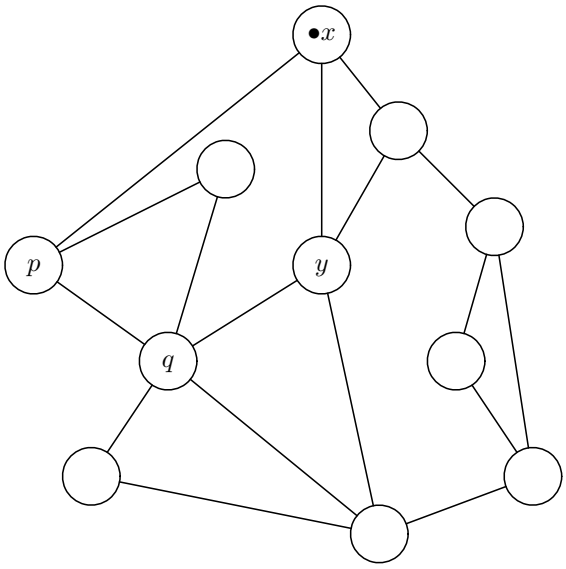
$$\neg\mathsf{idle}.p \wedge cp.p.q \wedge \neg sent\_to\_all\_non\_fathers.p \wedge \neg\mathsf{le\_rec}.p \quad (\maltese)$$

If the current situation is that of Figure 9.13(a), then x has just received the message, and hence $\mathsf{le\_rec}.x$ holds. Since we have assumed that $\neg\mathsf{le\_rec}.p$, we know that $(x \neq p)$. There are now two possibilities: either $\text{prop}_{\text{Tarry}}.x.y$ or action $\text{done}_{\text{Tarry}}.x.y$ is enabled (y is arbitrary) and will execute. Consequently, we know that a message will be sent and hence that Y will decrease. Since $(x \neq p)$, we know that $(\maltese)$ still holds, and subsequently, we have arrived in a situation similar to that of Figure 9.13(b).

If the current situation is that of Figure 9.13(b), then either $\text{idle}_{\text{Tarry}}.z.w$ or action $\text{col}_{\text{Tarry}}.z.w$ is enabled. If $(z = p)$, then we know that $\mathsf{le\_rec}.p$ will become true, and hence we are ready. If $(z \neq p)$, then we know that, since the message will be received by z, again Y shall decrease. Since $(z \neq p)$, we know that $(\maltese)$ still holds, and subsequently, we have arrived again in a situation similar to that of Figure 9.13(a).

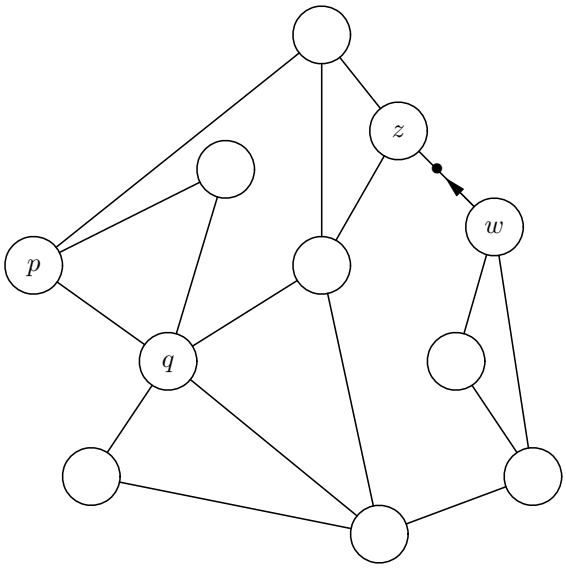
Since we have already proved that Y is a non-increasing function with respect to the well-founded relation $<$, we know that we cannot infinitely proceed from the situation in Figure 9.13(a) to the situation in Figure 9.13(b). Therefore, we shall eventually end in Figure 9.13(b) where $(z = p)$, and hence $\mathsf{le\_rec}.p$ will be set to true.

$_{\text{Tarry}}\vdash$ $\mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg\mathsf{le\_rec}.p \rightarrowtail \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \mathsf{le\_rec}.p$
$\Leftarrow(\rightarrowtail$ Bounded Progress $(4.5.17_{48})$, using $Y)$
$_{\text{Tarry}}\vdash$ $\mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg\mathsf{le\_rec}.p \wedge (Y = k)$
$\quad\quad\rightarrowtail$
$\quad\quad\mathsf{guard\_of}.(\text{prop}.p.q) \wedge ((\neg\mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$



$\exists x \in \mathbb{P} : \mathsf{le\_rec}.x$

(a)



$\forall x \in \mathbb{P} : \neg \mathsf{le\_rec}.x$

(b)

Figure 9.13: Possible situations when $\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p$ holds

$\Leftarrow$($\rightarrowtail$ Case distinction ($4.5.10_{47}$): situation of Figure 9.13(a), or 9.13(b))

$$\left.\begin{array}{ll} \text{Tarry}\vdash & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\exists x \in \mathbb{P} : \mathsf{le\_rec}.x) \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}\right\} \mathbf{9.13(a)}$$

$\wedge$

$$\left.\begin{array}{ll} \text{Tarry}\vdash & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p) \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}\right\} \mathbf{9.13(b)}$$

Verification of 9.13(a)

We shall proceed with proof-obligation **9.13(a)**, using the proof-strategy explained above. That is, we shall need to decompose the proof-obligation in such a way that we can use $\rightarrowtail$ Introduction ($4.5.7_{47}$) to prove that either $\text{prop}_{\text{Tarry}}.x.y$ or $\text{done}_{\text{Tarry}}.x.y$ will decrease Y. First, we shall identify process x (from Figure 9.13(a)) in the left hand side of $\rightarrowtail$ as follows:

$$\begin{array}{ll} \text{Tarry}\vdash & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\exists x \in \mathbb{P} : \mathsf{le\_rec}.x) \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}$$

$\Leftarrow$ ($\rightarrowtail$ Substition ($4.5.6_{47}$), $\rightarrowtail$ Disjunction ($4.5.13_{47}$), and $(x \neq p)$ since $(\neg \mathsf{le\_rec}.p \wedge \mathsf{le\_rec}.x)$)

$\forall x \in \mathbb{P}, (x \neq p)$:

$$\begin{array}{ll} \text{Tarry}\vdash & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge \mathsf{le\_rec}.x \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}$$

Whether $\text{prop}_{\text{Tarry}}.x.y$ or $\text{done}_{\text{Tarry}}.x.y$ is the action that will decrease Y, depends on whether x has *sent_to_all_non_fathers*, or not. Therefore, we proceed making the following case distinction:

$\Leftarrow$($\rightarrowtail$ Case Distinction ($4.5.10_{47}$))
$\forall x \in \mathbb{P}, (x \neq p)$:

$$\left.\begin{array}{ll} \text{Tarry}\vdash & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \\ & \wedge\ \mathsf{le\_rec}.x \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.x \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}\right\} \begin{array}{l} \mathbf{9.13(a)} \\ -\text{PROP} \end{array}$$

$\wedge$

$$\left.\begin{array}{ll} \text{Tarry}\vdash & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \\ & \wedge\ \mathsf{le\_rec}.x \wedge \mathit{sent\_to\_all\_non\_fathers}.x \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{prop}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}\right\} \begin{array}{l} \mathbf{9.13(a)} \\ -\text{DONE} \end{array}$$

Verification of 9.13(a)-PROP

The proof strategy for **9.13(a)**-PROP shall consists of using $\rightarrowtail$ Introduction ($4.5.7_{47}$), and proving that, for some y, $\text{prop}_{\text{Tarry}}.x.y$ ensures that the value of Y decreases.

Consequently, we have to substitute the left hand side $\rightarrowtail$ in such a way that it implies the existence of an y such that the guard of $\text{PROP}_{\text{TARRY}}.x.y$ holds. In order to be able to do this it suffices to prove that for arbitrary states s:

$$J_{\text{PLUM}}.s \wedge J_{\text{TARRY}}.s \wedge s.(\mathsf{le\_rec}.x) \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.x.s$$
$$\Rightarrow$$
$$\exists y \in \mathsf{neighs}.x : \neg\mathsf{idle}.x \wedge \mathit{cp}.x.y.s \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.x.s \wedge s.(\mathsf{le\_rec}.x)$$

Using Theorem $9.1.10_{161}$, and cJ^2_{PLUM} from J_{PLUM}, it is straightforward to prove that:

Theorem 9.4.21 *not_sent_2_all_except_f_IMP_cp*

$$\forall p \in \mathbb{P} : \frac{J_{\text{PLUM}}.s \wedge \neg \mathit{sent\_to\_all\_non\_fathers}.p.s}{\exists q \in \mathsf{neighs}.p : \mathit{cp}.p.q.s}$$

◀

Consequently, it remains to prove that x is non-idle. Since the fact that x has not *sent_to_all_non_fathers* is *not* sufficient to deduce this, we need a new invariant-candidate for J_{TARRY}. Evidently, the one that suffices here is:

$$\text{〰〰〰} \; cJ^2_{\text{TARRY}} = \forall p \in \mathbb{P} : \mathsf{le\_rec}.p \;\Rightarrow \neg\mathsf{idle}.p$$

Subsequently, **9.13(a)**-PROP is established as follows:

$\Leftarrow(\rightarrowtail$ SUBSTITUTION $(4.5.6_{47})$, cJ^2_{TARRY}, and Theorems $9.4.5_{193}$ and $9.4.21_{203})$
$\forall x \in \mathbb{P}, (x \neq p)$:
$\quad {}_{\text{TARRY}}\vdash \quad \exists y \in \mathsf{neighs}.x$:
$\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg\mathsf{le\_rec}.p \wedge (Y = k) \wedge \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.x.y)$
$\qquad \rightarrowtail$
$\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg\mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$
$\Leftarrow(\rightarrowtail$ DISJUNCTION $(4.5.13_{47})$, $\rightarrowtail$ INTRODUCTION $(4.5.7_{47}))$
$\forall x \in \mathbb{P}, (x \neq p), y \in \mathsf{neighs}.x$:
$\quad {}_{\text{TARRY}}\vdash \quad J_{\text{PLUM}} \wedge J_{\text{TARRY}}$
$\qquad \wedge \; \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg\mathsf{le\_rec}.p \wedge (Y = k) \wedge \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.x.y)$
$\qquad \mathsf{ensures}$
$\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg\mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$

Proving this ensures-property is straightforward using Lemma $9.4.19_{198}$.

Verification of 9.13(a)-DONE

The proof strategy for **9.13(a)**-DONE is similar to that of **9.13(a)**-PROP. That is, we use $\rightarrowtail$ INTRODUCTION $(4.5.7_{47})$ and prove that $\text{DONE}_{\text{TARRY}}.x.y$ ensures that the value of Y decreases. Again we have to substitute the left hand side $\rightarrowtail$ in such a way that it implies the guard of $\text{DONE}_{\text{TARRY}}.x.y$. However, since the guard of $\text{DONE}_{\text{TARRY}}$ is never enabled for the *starter*, we first have to prove that $(x \neq \mathit{starter})$. In order to do this we prove **9.13(a)**-DONE for the case when $(x = \mathit{starter})$ and $(x \neq \mathit{starter})$.

Verification of 9.13(a)-DONE when $x = starter$

We have to prove that, when $(starter \neq p)$,

$$\begin{array}{ll} {\scriptstyle \text{TARRY}}\vdash & \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \\ & \wedge\ \mathsf{le\_rec}.starter \wedge sent\_to\_all\_non\_fathers.starter \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}$$

Since the guard of $\text{DONE}_{\text{TARRY}}$ is never enabled for the *starter*, the only possible way to proceed here is: use $\rightarrowtail$ INTRODUCTION ($4.5.7_{47}$), and subsequently prove that the left hand side of the $\rightarrowtail$ in conjunction with J_{PLUM} and J_{TARRY} evaluates to false. So assume, for some state s, it holds that:

$\mathbf{A}_1$: $J_{\text{PLUM}}.s \wedge J_{\text{TARRY}}.s$

$\mathbf{A}_2$: $\mathsf{guard\_of}.(\text{PROP}.p.q).s \wedge \neg s.(\mathsf{le\_rec}.p)$

$\mathbf{A}_3$: $s.(\mathsf{le\_rec}.starter) \wedge sent\_to\_all\_non\_fathers.starter.s$

We shall now try to reach a contradiction. From $\mathbf{A}_2$, we can, using Definitions $8.7.2_{136}$ through $8.7.7_{136}$ and $9.1.8_{159}$, deduce that:

$\mathbf{A}_4$: $\neg done.p.s$

As a result, from Theorem $9.1.42_{183}$ together with assumptions $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_4$, we can infer that:

$\mathbf{A}_5$: $\neg done.starter.s$

From Theorem $9.1.37_{180}$ and assumption $\mathbf{A}_3$, we can derive that:

$\mathbf{A}_6$: $sent\_to\_all\_neighs.starter$

Since the *starter*'s last event was a receive event, we can argue, due to the alternating send and receive behaviour of TARRY, that the *starter* has *rec_from_all_neighs*, and consequently ($\mathbf{A}_6$ and Definition $8.7.7_{136}$) is *done*. Obviously, this establishes the desired contradiction with assumption $\mathbf{A}_5$. In order to be able to prove that the *starter* is indeed done, we need to introduce a new invariant-candidate for TARRY. Since initially, the le_rec variable of the *starter* is set to true, we state the following candidate:

$$\begin{array}{|ll|} \hline \overbrace{\quad\quad\quad}\, cJ^3_{\text{TARRY}} & = \quad \mathsf{le\_rec}.starter \Rightarrow (\mathsf{NR\_SENT}.starter = \mathsf{NR\_REC}.starter) \\ & \quad\ \wedge \\ & \quad\ \neg \mathsf{le\_rec}.starter \Rightarrow (\mathsf{NR\_SENT}.starter = \mathsf{NR\_REC}.starter + 1) \\ \hline \end{array}$$

Using $9.4.12_{197}$ and $9.4.13_{197}$, this candidate suffices to prove – under the assumptions stated above – that the *starter* is *done*.

Verification of 9.13(a)-DONE when $x \neq starter$

Now we know that x is not the *starter*, we have to substitute the left hand side of $\rightarrowtail$ in such a way that it implies the guard of $\text{DONE}_{\text{TARRY}}.x.y$. According to Theorems $9.4.6_{193}$, $9.1.9_{159}$ and Definition $8.7.4_{136}$, it suffices to prove that for arbitrary states s:

$$
\begin{array}{l}
J_{\text{PLUM}}.s \wedge J_{\text{TARRY}}.s \wedge s.(\mathsf{le\_rec}.x) \wedge \mathit{sent\_to\_all\_non\_fathers}.x.s \\
\Rightarrow \\
\quad \exists y \in \mathsf{neighs}.x : \quad \mathit{rec\_from\_all\_neighs}.x.s \\
\qquad\qquad\qquad\quad \wedge\ \mathit{sent\_to\_all\_non\_fathers}.x.s \\
\qquad\qquad\qquad\quad \wedge\ \neg \mathit{sent\_to\_all\_neighs}.x.s \\
\qquad\qquad\qquad\quad \wedge\ (y = (\mathsf{father}.x))
\end{array}
$$

Similar to the line of reasoning above, we introduce the following invariant-candidate for this purpose:

$$
\begin{array}{lll}
\overbrace{\qquad\qquad}^{cJ^4_{\text{TARRY}}} & = \forall p \in \mathbb{P} : & \mathsf{le\_rec}.p \Rightarrow (\mathsf{NR\_REC}.p = \mathsf{NR\_SENT}.p + 1) \\
& & \wedge \\
& & \neg \mathsf{le\_rec}.p \Rightarrow (\mathsf{NR\_REC}.p = \mathsf{NR\_SENT}.p)
\end{array}
$$

Subsequently, 9.13(a)-DONE for the case that $(x \neq \mathit{starter})$ is established as follows:

$\Leftarrow$($\rightarrowtail$ SUBSTITUTION $(4.5.6_{47})$, cJ^4_{TARRY}, $9.1.9_{159}$, $9.4.6_{193}$, $9.4.12_{197}$, and $9.4.16_{197}$)

$$
\begin{array}{l}
\forall x \in \mathbb{P}, (x \neq p), (x \neq \mathit{starter}) : \\
\quad {}_{\text{TARRY}}\vdash \quad \exists y \in \mathsf{neighs}.x : \\
\qquad\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge \mathsf{guard\_of}.(\text{DONE}_{\text{TARRY}}.x.y) \\
\qquad\qquad \rightarrowtail \\
\qquad\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))
\end{array}
$$

$\Leftarrow$($\rightarrowtail$ DISJUNCTION $(4.5.6_{47})$, $\rightarrowtail$ INTRODUCTION $(4.5.7_{47})$)

$$
\begin{array}{l}
\forall x \in \mathbb{P}, (x \neq p), (x \neq \mathit{starter}) : \\
\quad {}_{\text{TARRY}}\vdash \quad J_{\text{PLUM}} \wedge J_{\text{TARRY}} \\
\qquad\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge \mathsf{guard\_of}.(\text{DONE}_{\text{TARRY}}.x.y) \\
\qquad\qquad \mathsf{ensures} \\
\qquad\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))
\end{array}
$$

Proving this ensures-property is straightforward using Lemma $9.4.19_{198}$.

Verification of 9.13(b)

For convenience, the proof obligation tackled in in this section is re-displayed below (from page 202):

$$
\begin{array}{l}
{}_{\text{TARRY}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p) \\
\qquad\qquad \rightarrowtail \\
\qquad\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))
\end{array}
$$

Here we shall employ the proof-strategy explained on page 200. That is, we shall need to decompose the proof-obligation in such a way that we can use $\rightarrowtail$ INTRODUCTION $(4.5.7_{47})$ to prove that either $\text{IDLE}_{\text{TARRY}}$ or $\text{COL}_{\text{TARRY}}$ decreases Y or establishes $\mathsf{le\_rec}.p$. From Figure 9.13(b), we know that in this situation, there is a message in transit somewhere in the network. Moreover, using cJ^4_{PLUM}, cJ^2_{PLUM}, cJ^2_{PLUM} and Definition $8.7.1_{136}$, it is not hard to prove that:

Theorem 9.4.22 *mit_IMP_not_rec_from_all_neighs*

$$\forall p \in \mathbb{P}, q \in \mathsf{neighs}.p : \frac{J_{\text{PLUM}}.s \wedge \mathsf{mit}.q.p.s}{\neg \mathit{rec\_from\_all\_neighs}.p.s}$$

◀

Consequently, we can substitute the left hand side of $\rightarrowtail$ as follows: (we use the names z and w since these correspond to Figure 9.13(b))

$\Leftarrow(\rightarrowtail$ Substitution $(4.5.6_{47})$, cJ^1_{TARRY}, Theorem $9.4.22_{206})$
$\forall z \in \mathbb{P}, w \in \mathsf{neighs}.z :$
$_{\text{TARRY}}\vdash$ $\exists z \in \mathbb{P}, w \in \mathsf{neighs}.z :$
$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p)$
$\wedge\ \mathsf{mit}.w.z \wedge \neg \mathit{rec\_from\_all\_neighs}.z$
$\rightarrowtail$
$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$
$\Leftarrow(\rightarrowtail$ Disjunction $(4.5.13_{47}))$
$\forall z \in \mathbb{P}, w \in \mathsf{neighs}.z :$
$_{\text{TARRY}}\vdash$ $\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p)$
$\wedge\ \mathsf{mit}.w.z \wedge \neg \mathit{rec\_from\_all\_neighs}.z$
$\rightarrowtail$
$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$

If $(z = p)$, the proof obligation from above can be proved using $\rightarrowtail$ Introduction $(4.5.7_{47})$, since execution of $\text{COL}_{\text{TARRY}}.p.w$ will ensure that $\mathsf{le\_rec}.p$ is set to true.

Suppose $(z \neq p)$. Whether $\text{IDLE}_{\text{TARRY}}.z.w$ or $\text{COL}_{\text{TARRY}}.z.w$ is the action that will decrease Y, depends on whether z is idle or not. Therefore, we proceed as follows:

$\Leftarrow(\rightarrowtail$ Case Distinction $(4.5.10_{47}))$
$\forall z \in \mathbb{P}, w \in \mathsf{neighs}.z, (z \neq p) :$
$_{\text{TARRY}}\vdash$ $\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p)$
$\wedge\ \mathsf{mit}.w.z \wedge \neg \mathit{rec\_from\_all\_neighs}.z \wedge \mathsf{idle}.z$
$\rightarrowtail$
$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$
$\wedge$
$_{\text{TARRY}}\vdash$ $\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p)$
$\wedge\ \mathsf{mit}.w.z \wedge \neg \mathit{rec\_from\_all\_neighs}.z \wedge \neg \mathsf{idle}.z$
$\rightarrowtail$
$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$
$\Leftarrow(\rightarrowtail$ Substitution $(4.5.6_{47})$ on both conjuncts, using $9.1.6_{159}$, $9.1.7_{159}$, $9.4.3_{193}$, $9.4.4_{193})$
$\forall z \in \mathbb{P}, w \in \mathsf{neighs}.z, (z \neq p) :$
$_{\text{TARRY}}\vdash$ $\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p)$
$\wedge\ \mathsf{guard\_of}.(\text{IDLE}_{\text{TARRY}}.z.w)$
$\rightarrowtail$
$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p))$
$\wedge$

$$\begin{array}{ll} {\scriptstyle \text{TARRY}}\vdash & \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k) \wedge (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p) \\ & \wedge\ \mathsf{guard\_of}.(\text{COL}_{\text{TARRY}}.z.w) \\ & \rightarrowtail \\ & \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge ((\neg \mathsf{le\_rec}.p \wedge (Y < k)) \vee (\mathsf{le\_rec}.p)) \end{array}$$

Both conjuncts can be proved using $\rightarrowtail$ INTRODUCTION ($4.5.7_{47}$), and Lemma $9.4.19_{198}$.

This ends the verification of **9.13(b)**, and hence of the **reach**-PROP-**part** (page 199), and consequently of the termination of TARRY. The one thing that remains to be done, is constructing TARRY's additional invariant. Gathering all the candidates introduced (i.e. cJ^1_{TARRY} through cJ^4_{TARRY}), analysing them, and verifying the stability of their conjunction results in the need to introduce yet three more invariant-candidates. Again, since the verification activities are not all that exciting, we shall just state the required candidates. The first one comes as no surprise and states that, if there is a message in transit it is the only one:

$$\overset{\frown}{\text{mmmmm}}cJ^5_{\text{TARRY}} = \quad \forall p, x \in \mathbb{P}, q \in \mathsf{neighs}.p, y \in \mathsf{neighs}.x : \\ \mathsf{mit}.p.q \wedge \mathsf{mit}.x.y \Rightarrow (p = x) \wedge (q = y)$$

The second and the third one together state that if there is no message in transit, then there is exactly one process that has received a message:

$$\overset{\frown}{\text{mmmmm}}\ cJ^6_{\text{TARRY}} = \neg(\exists p \in \mathbb{P}, q \in \mathsf{neighs}.p : \mathsf{mit}.p.q) \Rightarrow (\exists p \in \mathbb{P} : \mathsf{le\_rec}.p)$$

$$\overset{\frown}{\text{mmmmm}}\ cJ^7_{\text{TARRY}} = \forall p, q \in \mathbb{P} : \mathsf{le\_rec}.p \wedge \mathsf{le\_rec}.q \Rightarrow (p = q)$$

Since cJ^1_{TARRY} and cJ^6_{TARRY} can be coalesced into one candidate using equality, we have derived a characterisation of J_{TARRY} that is displayed in Figure 9.14.

9.5 Proving termination of DFS

This section shall describe how termination of the DFS algorithm is proved using the refinements framework from Chapter 7, and the already proven fact that:

$$\forall J :: \text{TARRY} \sqsubseteq_{\mathcal{R}\_\text{TARRY}\_\text{DFS},\ J} \text{DFS}$$

The UNITY specification reads:

Theorem 9.5.1 *HYLO_DFS*

$$\begin{array}{ll} \forall iA,\ h,\ \text{PROP\_mes},\ \text{DONE\_mes} :: & \\ J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge J_{\text{DFS}\ \ \text{DFS}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}} \vdash & \mathbf{ini}(\text{DFS}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \\ & \rightsquigarrow \\ & \forall p : p \in \mathbb{P} : \mathit{done}.p \end{array}$$

◀

Definition 9.4.23 TARRY'S ADDITIONAL INVARIANT *Invariant_Tarry_Part*

$J_{\text{TARRY}} =$

$$
\begin{array}{lll}
 & (\exists p \in \mathbb{P}, q \in \mathsf{neighs}.p : mit.p.q) = (\forall p \in \mathbb{P} : \neg \mathsf{le\_rec}.p) & cJ^1_{\text{TARRY}}, cJ^6_{\text{TARRY}} \\
\wedge & \forall p \in \mathbb{P} : \mathsf{le\_rec}.p \Rightarrow \neg \mathsf{idle}.p & cJ^2_{\text{TARRY}} \\
\wedge & \mathsf{le\_rec}.starter \Rightarrow (\mathsf{NR\_SENT}.starter = \mathsf{NR\_REC}.starter) & \\
 & \wedge \neg \mathsf{le\_rec}.starter \Rightarrow (\mathsf{NR\_SENT}.starter = \mathsf{NR\_REC}.starter + 1) & cJ^3_{\text{TARRY}} \\
\wedge & \forall p \in \mathbb{P} : \quad \mathsf{le\_rec}.p \Rightarrow (\mathsf{NR\_REC}.p = \mathsf{NR\_SENT}.p + 1) & \\
 & \qquad \wedge \neg \mathsf{le\_rec}.p \Rightarrow (\mathsf{NR\_REC}.p = \mathsf{NR\_SENT}.p) & cJ^4_{\text{TARRY}} \\
\wedge & \forall p, x \in \mathbb{P}, q \in \mathsf{neighs}.p, y \in \mathsf{neighs}.x : & \\
 & \quad \mathsf{mit}.p.q \wedge \mathsf{mit}.x.y \Rightarrow (p = x) \wedge (q = y) & cJ^5_{\text{TARRY}} \\
\wedge & \forall p, q \in \mathbb{P} : \mathsf{le\_rec}.p \wedge \mathsf{le\_rec}.q \Rightarrow (p = q) & cJ^7_{\text{TARRY}}
\end{array}
$$

Theorem 9.4.24 *STABLEe_Invariant_Tarry*

$$_{\text{TARRY}}\vdash \circlearrowleft J_{\text{PLUM}} \wedge J_{\text{TARRY}}$$

Theorem 9.4.25 *INVe_Invariant_Tarry*

$$_{\text{TARRY}}\vdash \Box J_{\text{PLUM}} \wedge J_{\text{TARRY}}$$

Figure 9.14: TARRY's invariant

where invariant J_{DFS} captures additional safety properties for DFS (if any). Using $\circlearrowleft$ PRESERVATION Theorem $7.2.12_{112}$, it is straightforward to derive:

Theorem 9.5.2 *STABLEe_Invariant_in_DFS*

$$_{\text{DFS}}\vdash \circlearrowleft (J_{\text{PLUM}} \wedge J_{\text{TARRY}})$$

The stability of: $_{\text{DFS}}\vdash\circlearrowleft (J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge J_{\text{DFS}})$ will be implicitly assumed throughout the verification process. For ease of reference, Figure 9.15 displays theorems about the guards of DFS's actions. And again, for readability we introduce the notational convention that:

$\vdash$ and $_{\text{DFS}}\vdash$ now abbreviate $J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge J_{\text{DFS}}$ $_{\text{DFS}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}}\vdash$

for arbitrary $iA \in \texttt{Expr}$, $h \in \mathbb{P}\rightarrow\texttt{Expr}\rightarrow\texttt{Expr}$, PROP_mes $\in \mathbb{P}\rightarrow\texttt{Expr}$, and DONE_mes $\in \mathbb{P}\rightarrow\texttt{Expr}$, for which hold that:

$\forall p, e : p \in \mathbb{P} \wedge e\, \mathcal{C}\, \mathbf{w}\text{PLUM} : (h.p.e)\, \mathcal{C}\, \mathbf{w}\text{PLUM}$
$\forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p\, \mathcal{C}\, \mathbf{w}\text{PLUM} \ \wedge\ \mathsf{DONE\_mes}.p\, \mathcal{C}\, \mathbf{w}\text{PLUM}$

Theorem 9.5.3 *guard_of_IDLE_DFS*

$\mathsf{guard\_of}.(\text{IDLE}_{\text{DFS}}.p.q) = \mathsf{guard\_of}.(\text{IDLE}_{\text{TARRY}}.p.q)$

Theorem 9.5.4 *guard_of_COL_DFS*

$\mathsf{guard\_of}.(\text{COL}_{\text{DFS}}.p.q) = \mathsf{guard\_of}.(\text{COL}_{\text{TARRY}}.p.q)$

Theorem 9.5.5 *guard_of_PROP_lp_rec_DFS*

$\mathsf{guard\_of}.(\text{PROP\_LP\_REC}.p.q) = \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \wedge q = \mathsf{lp\_rec}.p$

Theorem 9.5.6 *guard_of_PROP_not_lp_rec_DFS*

$\mathsf{guard\_of}.(\text{PROP\_NOT\_LP\_REC}.p.q) = \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \wedge \neg cp.p.(\mathsf{lp\_rec}.p)$

Theorem 9.5.7 *guard_of_DONE_DFS*

$\mathsf{guard\_of}.(\text{DONE}_{\text{DFS}}.p.q) = \mathsf{guard\_of}.(\text{DONE}_{\text{TARRY}}.p.q)$

Figure 9.15: Guards of the actions from DFS

To remind the reader of the reason why we need to assume confinement by PLUM's write variables and not DFS's write variables, he or she is referred to page 192.

Again, we implicitly assume the validity of distinct_DFS_Vars (see Definition D.4.4$_{259}$), ASYNC_type_decl.$\mathbb{P}$.neighs, and Connected_Network.$\mathbb{P}$.neighs.starter.

9.5.1 Using refinements to derive termination of DFS

As indicated in Chapter 8, termination of DFS is proved using property preserving Theorem 7.2.7$_{113}$. The reasons for using this Theorem are twofold. First, since every PROP action in TARRY is bitotally related to two actions in DFS (namely PROP_LP_REC and PROP_NOT_LP_REC), we need to be able to pick one of those DFS PROP-actions when proving that the guards of TARRY's PROP-actions eventually implies the guards of related DFS's PROP-actions. Consequently, we cannot use preservation theorems 7.2.10$_{113}$ or 7.2.9$_{113}$. The second reason for using 7.2.7$_{113}$ is *not* because 7.2.8$_{113}$ cannot be used, but because it reduces proof effort. As we have seen during TARRY's verification, Lemma 9.4.19$_{198}$ was very useful when proving unless and ensures properties that involved Y. A similar lemma can easily be proved for the actions of DFS, and hence verification of unless and ensures properties involving Y in the context of DFS will be simple too.

Lemma 9.5.8 *A_DECR_Y*

For arbitrary processes $p \in \mathbb{P}$, $q \in \mathsf{neighs}.p$, and actions $A \in \{\text{IDLE}_{\text{DFS}}, \text{COL}_{\text{DFS}}, \text{PROP\_LP\_REC}, \text{PROP\_NOT\_LP\_REC}, \text{DONE}_{\text{DFS}}\}$:

$$\forall k :: \frac{J_{\text{PLUM}}.s \wedge A.p.q.s.t \wedge \mathsf{guard\_of}.(A.p.q).s \wedge (Y.s = k)}{Y.t < k}$$

◀

Therefore, we decided to use $7.2.7_{113}$, although a function that is non-increasing with respect to some well-founded relation is *not* needed in order to be able to prove that falsification of the guards of DFS's PROP-actions go hand in hand with the falsification of the guards of TARRY's PROP-actions.

As a result, the initial specification stating termination of DFS is decomposed as follows:

$$_{\text{DFS}}\vdash \mathbf{ini}(\text{DFS}.iA.h.\text{PROP\_mes}.\text{DONE\_mes}) \rightsquigarrow \forall p : p \in \mathbb{P} : done.p$$

$\Leftarrow$(Theorem $7.2.8_{113}$, $9.4.1_{192}$, $8.12.3_{149}$, and $\text{D}.4.12_{260}$))
For some well-founded relation $\prec$:

$$\begin{array}{l}
\quad \exists W :: (\mathbf{w}\text{DFS} = \mathbf{w}\text{TARRY} \cup W) \wedge ((J_{\text{PLUM}} \wedge J_{\text{TARRY}})\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}\text{TARRY} \subseteq W^{\mathsf{c}}) \\
\wedge \\
\quad \forall A_D : A_D \in \mathbf{a}\text{DFS} \wedge (\exists A_T :: A_T \in \mathbf{a}\text{TARRY} \wedge (A_T\ \mathcal{R}_{\text{-TARRY_DFS}}\ A_D)) : \\
\qquad (\mathsf{guard_of}.A_D\ \mathcal{C}\ \mathbf{w}\text{DFS}) \\
\wedge \\
\left.\begin{array}{l}
\quad \forall A_T\ A_D\ :\ A_T \in \mathbf{a}\text{TARRY} \\
\qquad _{\text{DFS}}\vdash\ \mathsf{guard_of}.A_T \\
\qquad\qquad \rightarrowtail \\
\qquad\qquad (\exists A_D :: (A_T\ \mathcal{R}_{\text{-TARRY_DFS}}\ A_D) \wedge \mathsf{guard_of}.A_D)
\end{array}\right\} \mathbf{reach-part} \\
\wedge \\
\left.\begin{array}{l}
\quad \exists M :: (M\ \mathcal{C}\ \mathbf{w}\text{DFS}) \\
\qquad \wedge \\
\qquad \forall k :: {}_{\text{DFS}}\vdash (J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge J_{\text{DFS}} \wedge M = k)\ \mathsf{unless}\ (M \prec k) \\
\qquad \wedge \\
\qquad \forall k\ A_T\ A_D : A_T \in \mathbf{a}\text{TARRY} \wedge A_T\ \mathcal{R}_{\text{-TARRY_DFS}}\ A_D : \\
\qquad\quad _{\text{DFS}}\vdash\ (J_{\text{PLUM}} \wedge J_{\text{TARRY}} \wedge J_{\text{DFS}} \wedge \mathsf{guard_of}.A_D \wedge M = k) \\
\qquad\qquad\quad \mathsf{unless} \\
\qquad\qquad\quad (\neg(\mathsf{guard_of}.A_T) \vee M \prec k)
\end{array}\right\} \mathbf{unless-part}
\end{array}$$

Since $\mathsf{lp\_rec}.p$ variables are superimposed on TARRY in order to obtain DFS, the first conjunct is instantiated with the set $\{\mathsf{lp\_rec}.p \mid p \in \mathbb{P}\}$. Proving that J_{PLUM} and J_{TARRY} are confined by the complement of this set is tedious but straightforward, since the variables $\mathsf{le\_rec}$ do not appear in it. Similarly, proving that the guards of the actions in DFS are confined by DFS's write variables (i.e. the second conjunct) is not complicated.

The **unless-part** is now easy to prove by instantiating with Y (Definition $9.4.17_{197}$):

- proving that Y is confined by the write variables of DFS is easy using Theorem $9.4.18_{198}$ and monotonicity of confinement $3.3.19_{28}$
- proving that Y is non-increasing in DFS, can be proved using unless PRESERVATION Theorem $8.10.3_{145}$, and Theorem $7.2.11_{112}$.
- proving that falsification of the guards of DFS's actions go hand in hand with the falsification of the guards of related TARRY's actions is easy using Lemma $9.5.8_{210}$.

For the **reach-part**, the IDLE, COL, and DONE cases can be proved using $\rightarrowtail$ INTRODUCTION $(4.5.7_{47})$. As a consequence, we are left with the PROP case:

$$\begin{array}{ll} {}_{\text{DFS}}\vdash & \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \\ & \rightarrowtail \\ & (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY\_DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D) \end{array}$$

This case states that: from a situation in which $\mathsf{guard\_of}.(\text{PROP}.p.q)$ holds, we will eventually reach a situation in which either the guard of action PROP_LP_REC.$p.q$ or PROP_NOT_LP_REC.$p.q$ holds. To explain the proof-strategy that is used to verify this proof obligation, we refer to Figure 9.16. The p and q in the picture correspond to the p and q in the proof obligation, z is an arbitrary process. In Figure 9.16 we are in the situation that the guard of $\text{PROP}_{\text{TARRY}}.p.q$ holds, that is (Theorem $9.4.5_{193}$):

$$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \mathsf{le\_rec}.p$$

Process p has just received the message, and therefore is the only process that can do something. There are now two possibilities:

$q = \mathsf{lp\_rec}.p$ In this case the guard of PROP_LP_REC.$p.q$ holds and we are done.

$q \neq \mathsf{lp\_rec}.p$ In this case the guard of PROP_LP_REC.$p.q$ cannot hold. Again there are two possibilities:

$\neg\ cp.p.(\mathsf{lp\_rec}.p)$, that is p is not allowed to propagate a message to the process it has received its last message from. In this case, p can pick any non-father-neighbour to which it has not yet sent a message. Evidently, we can pick q, and as a consequence, the guard of PROP_NOT_LP_REC.$p.q$ is enabled.

$cp.p.(\mathsf{lp\_rec}.p)$ In this case p *has to* send a message to the process it has received its last message from, and since this is not q, neither the guard of PROP_LP_REC.$p.q$ nor PROP_NOT_LP_REC.$p.q$ holds. If z (from Figure 9.16) is equal to $\mathsf{lp\_rec}.p$, then the guard of PROP_LP_REC.$p.z$ is enabled and consequently p shall send a message to z. Since $(z \neq q)$, we know that afterwards the following holds:

$$\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg\mathsf{le\_rec}.p$$

Now we find ourself in the situation in Figure 9.13_{201}(b), from which we can transfer to situation in Figure 9.13_{201}(a) or Figure 9.16. Again, a well-foundedness argument, using $\rightarrowtail$ BOUNDED PROGRESS $(4.5.17_{48})$, shall enable us to prove that

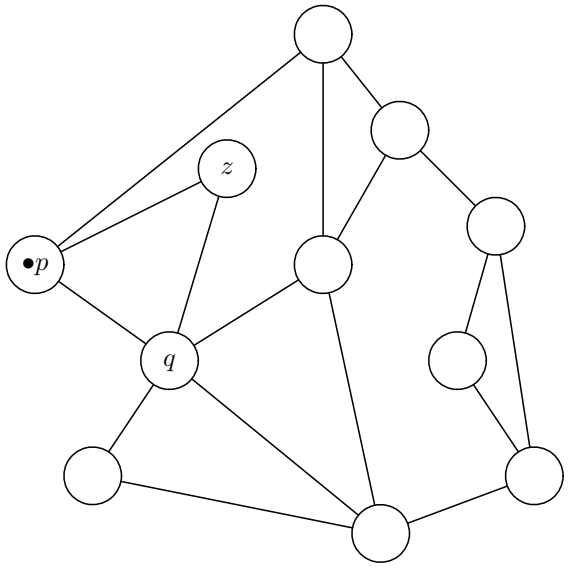


Figure 9.16: Situation when $\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \mathsf{le\_rec}.p$ holds

we cannot infinitely go back and forth between these situations, and therefore that eventually the guard of PROP_LP_REC.$p.q$ or PROP_NOT_LP_REC.$p.q$ will be enabled.

Consequently, when we use non-increasing function Y again for this well-foundedness argument, the proof of DFS's **reach**-PROP-**part** shall resemble that of TARRY's (see page 200). Therefore we shall only present the begin of the proof, which is slightly different from TARRY.

$$
\begin{array}{ll}
 & {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \\
 & \qquad \rightarrowtail \\
 & \qquad (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY-DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D) \\
\Leftarrow & (\rightarrowtail \text{CASE DISTINCTION } (4.5.10_{47})) \\
 & {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \wedge q = \mathsf{lp\_rec}.p \\
 & \qquad \rightarrowtail \\
 & \qquad (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY-DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D) \\
\wedge & \\
 & {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \wedge q \neq \mathsf{lp\_rec}.p \\
 & \qquad \rightarrowtail \\
 & \qquad (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY-DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D) \\
\Leftarrow & (\rightarrowtail \text{INTRODUCTION } (4.5.7_{47}), \text{ and } 9.5.5_{209} \text{ proves first conjunct,} \\
 & \quad \rightarrowtail \text{CASE DISTINCTION } (4.5.10_{47}) \text{ on second conjunct}) \\
 & {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \wedge q \neq \mathsf{lp\_rec}.p \wedge \neg cp.p.(\mathsf{lp\_rec}.p) \\
 & \qquad \rightarrowtail \\
 & \qquad (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY-DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D)
\end{array}
$$

$\wedge$

$\quad {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \wedge q \neq \mathsf{lp\_rec}.p \wedge cp.p.(\mathsf{lp\_rec}.p)$
$\qquad \rightarrowtail$
$\qquad (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY\_DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D)$

$\Leftarrow$($\rightarrowtail$ INTRODUCTION ($4.5.7_{47}$), and $9.5.6_{209}$ proves first conjunct,
$\quad \rightarrowtail$ TRANSITIVITY ($4.5.9_{47}$) on second conjunct)

$\quad {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}_{\text{TARRY}}.p.q) \wedge q \neq \mathsf{lp\_rec}.p \wedge cp.p.(\mathsf{lp\_rec}.p)$
$\qquad \rightarrowtail$
$\qquad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p$

$\wedge$

$\quad {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p$
$\qquad \rightarrowtail$
$\qquad (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY\_DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D)$

$\Leftarrow$($\rightarrowtail$ INTRODUCTION ($4.5.7_{47}$), PROP_LP_REC.p.($\mathsf{lp\_rec}.p$) establishes $\neg$ $\mathsf{le\_rec}.p$,
$\quad \rightarrowtail$ BOUNDED PROGRESS ($4.5.17_{48}$) on second conjunct)

$\quad {}_{\text{DFS}}\vdash \quad \mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y = k)$
$\qquad \rightarrowtail$
$\qquad (\mathsf{guard\_of}.(\text{PROP}.p.q) \wedge \neg \mathsf{le\_rec}.p \wedge (Y < k))$
$\qquad \vee$
$\qquad (\exists A_D :: (\text{PROP}_{\text{TARRY}}.p.q\ \mathcal{R}_{\text{-TARRY\_DFS}}\ A_D) \wedge \mathsf{guard\_of}.A_D)$

From here, the proof is similar to that of TARRY (starting at page 200), and hence is not repeated. We end the verification of DFS's termination by observing that the verification of DFS did not need any more safety properties, and thus that J_{DFS} can defined to be true.

Definition 9.5.9 *Invariant_DFS*

$J_{\text{DFS}} = \mathsf{true}$

9.6 Concluding remarks

Although this is a tough chapter to read (as well as write), we think we have succeeded in presenting intuitive and structured proofs of the correctness of distributed hylomorphisms with respect to their termination. Due to the incremental, demand-driven construction of the invariant, the latter is not "pulled out of a hat" [Cho95], and the purpose of its various conjuncts are well motivated. Moreover, since, various property preservation theorems are necessary throughout the verification process, this chapter also serves as an illustration of the usage and effectiveness of the refinement framework from Chapter 7.

Appendix A

Miscellaneous notation, theories and theorems

This appendix presents miscellaneous notation, theories and theorems that appear throughout this thesis. Although of some concepts treated in this appendix it can be assumed that everyone has a *notion* of what they mean, the reason that a separate chapter in devoted to these concepts is to obtain unequivocalness when referring to them. For experience shows that minor differences between the notions people have about formal concepts, can cause confusion and misunderstanding. Moreover, when theorem provers are used, absolute accuracy is mandatory.

A.1 Universal and existential quantification

Universal quantification is denoted by: $\forall x :: P.x$
Restricted universal quantification is denoted by: $\forall x : Q.x : P.x$,
meaning: $(\forall x :: Q.x \Rightarrow P.x)$
When restricting predicate Q involves set membership in some set S (for example $Q.x$ equals $x \in S$), we also write: $\forall x \in S : P.x$.

Existential quantification is denoted by: $\exists x :: P.x$
Restricted existential quantification is denoted by: $\exists x : Q.x : P.x$,
meaning: $(\exists x :: Q.x \wedge P.x)$
When restricting predicate Q involves set membership in some set S (for example $Q.x$ equals $x \in S$), we also write: $\exists x \in S : P.x$.

A.2 Functions

The fact that x is an element of type t is denoted by $x \in t$. Consequently, a function f with source type s and target type t is denoted by $f \in s \rightarrow t$.
Function application is denoted by a dot, i.e. for function $f \in s \rightarrow t$, applying f to

some x from s is denoted by $f.x$.

Sectioning infix operators to convert them to prefix higher order functions (i.e. sections), is done by simply using the infix operator as a prefix. More formally:

$$e_1 \oplus e_2 \quad \text{is sectioned by writing} \quad \oplus.e_1.e_2$$

Let $f \in \alpha \to \beta$, $g \in \gamma \to \alpha$, A be a set with elements of type α, and B a set with elements of type β.

Definition A.2.1 FUNCTION COMPOSITION *o_DEF*

$$\forall f\ g ::\ f \circ g\ =\ (\lambda x.\ f.(g.x))$$

Definition A.2.2 SPLIT *split*

$$\forall f\ x ::\ \mathsf{split}.f.x\ =\ (f.x, x)$$

Definition A.2.3 CONVERT FUNCTION TO A RELATION *F2R_DEF*

$$\forall f\ ::\ \mathsf{f2r}.f = (\lambda x\ y.y = (f.x))$$

Definition A.2.4 SURJECTION *SURJECTION_DEF*

$$\begin{array}{lcl} \forall f\ A\ B :: \mathsf{surjection}.f.A.B & = & (\forall x :: (x \in A) \Rightarrow (f.x \in B)) \\ & & \wedge\ (\forall y :: (y \in B) \Rightarrow (\exists x :: (x \in A) \wedge (y = f.x))) \end{array}$$

Definition A.2.5 INJECTION *INJECTION_DEF*

$$\forall f\ A :: \mathsf{injection}.f.A\ =\ (\forall x, y \in A : ((f.x) = (f.y)) \Rightarrow (x = y))$$

Definition A.2.6 BIJECTION *BIJECTION_DEF*

$$\forall f\ A\ B :: \mathsf{bijection}.f.A.B\ =\ (\mathsf{surjection}.f.A.B) \wedge (\mathsf{injection}.f.A)$$

A.3 Relations

Let $R \in \alpha \to \beta \to \mathsf{bool}$, $S \in \beta \to \gamma \to \mathsf{bool}$, A be a set with elements of type α and B a set with elements of type β.

Definition A.3.1 RELATION COMPOSITION *rSEQ*

$$\forall R\ S :: R \circ S = (\lambda x\ y.\exists z :: xRz\ \wedge\ zSy)$$

Definition A.3.2 JUNCTION *junc_DEF*

$$\forall R\ S\ A\ B :: (R \triangledown S).A.B.x.y = (x \in A \wedge xRy) \vee (x \in B \wedge xSy)$$

A relation R is *bitotal* on sets A and B, when for every element in A there exists at least one element in B to which it is related, and similarly for B.

Definition A.3.3 BITOTAL RELATION *BITOTAL_DEF*

$$\begin{aligned}\forall R\ A\ B :: \mathsf{bitotal}.R.A.B \quad = \quad & \forall x\ y :: ((x \in A) \wedge (xRy)) \Rightarrow (y \in B) \\ & \wedge (\forall x :: (x \in A) \Rightarrow (\exists y :: (y \in B) \wedge xRy)) \\ & \wedge (\forall y :: (y \in B) \Rightarrow (\exists x :: (x \in A) \wedge xRy))\end{aligned}$$

Theorem A.3.4 *F2R_BITOTAL_is_SURJECTION*

$\forall f\ A\ B :: \mathsf{surjection}.f.A.B = \mathsf{bitotal}.(\mathsf{f2r}.f).A.B$

A.4 Lists

In the built-in theory `list`, a type `('a)list` is defined to denote the set of all finite lists having elements of type `'a`. The constructor functions that are used to construct any list-structured value of type `('a)list` are:

$[\,] \in$ `('a)list`
CONS $\in$ `'a` $\rightarrow$ `('a)list` $\rightarrow$ `('a)list`

Below definitions and theorems are stated. In these definitions, 0 and SUC refer to the constructor functions that are used to construct any natural number in `num`.

Theorem A.4.1 LIST ELEMENT *IS_EL*

$(\forall x :: \neg \mathsf{is\_el}.x.[]) \wedge (\forall x\ y\ l :: \mathsf{is\_el}.y.(\mathsf{CONS}.x.l) = (y = x) \vee \mathsf{is\_el}.y.l)$

Definition A.4.2 MAP *MAP*

$(\forall f :: \mathsf{map}.f.[] = []) \wedge (\forall f\ x\ l :: \mathsf{map}.f.(\mathsf{CONS}.x.l) = \mathsf{CONS}.(f\ x).(\mathsf{map}.f.l))$

Definition A.4.3 ZIP *ZIP*

$(\mathsf{zip}.([], []) = []) \wedge$
$(\forall x_1\ l_1\ x_2\ l_2 :: \mathsf{zip}.(\mathsf{CONS}.x_1.l_1, \mathsf{CONS}.x_2.l_2) = \mathsf{CONS}.(x_1, x_2).(\mathsf{zip}.(l_1, l_2)))$

Definition A.4.4 FOLDR *FOLDR*

$(\forall f\ e :: \mathsf{foldr}.f.e.[] = e) \wedge (\forall f\ e\ x\ l :: \mathsf{foldr}.f.e.(\mathsf{CONS}.x.l) = f.x.(\mathsf{foldr}.f.e.l))$

Definition A.4.5 LENGTH *LENGTH*

$(\mathsf{length}.[] = 0) \wedge (\forall x\ l :: \mathsf{length}.(\mathsf{CONS}.x.l) = (\mathsf{length}.l)) + 1$

Definition A.4.6 EVERY *EVERY*

$(\forall P :: \mathsf{every}.P.[] = \mathsf{true}) \wedge (\forall P\ h\ t :: \mathsf{every}.P.(\mathsf{CONS}.h.t) = P.h \wedge \mathsf{every}.P.t)$

Definition A.4.7 SUM *SUM*

$(\mathsf{sum}.[] = 0) \wedge (\forall x\ l :: \mathsf{sum}.(\mathsf{CONS}.x.l) = x + (\mathsf{sum}.l))$

Theorem A.4.8 MAP COMPOSTION *MAP_o*

$\forall f\ g\ l :: \mathsf{map}.f.(\mathsf{map}.g.l) = \mathsf{map}.(f \circ g).l$

Theorem A.4.9 *ZIP_MAP_EQ_MAP_split*

$\forall f\ l ::\ \mathsf{zip}.((\mathsf{map}.f.l), l)\ =\ \mathsf{map}.(\mathsf{split}.f).l$

Theorem A.4.10 *IS_EL_MAP*

$\forall Q\ f\ l ::\ (\forall x : \mathsf{is\_el}.x.(\mathsf{map}.f.l)\ :\ Q.x)\ =\ (\forall x : \mathsf{is\_el}.x.l\ :\ Q.(f.x))$

Definition A.4.11 *interval*

$\mathsf{interval}.0 = [] \ \wedge\ \mathsf{interval}.(\mathsf{SUC}.n) = \mathsf{CONS}.n.(\mathsf{interval}.n)$

Theorem A.4.12 APPENDING LISTS *APPEND*

$\forall l\ ::\ ([] +\!\!+\ l) = l\ \wedge\ \forall l_1\ l_2\ x\ ::\ ((\mathsf{CONS}.x.l_1) +\!\!+\ l_2) = (\mathsf{CONS}.x.(l_1 +\!\!+\ l_2))$

Definition A.4.13 DELETING AN ELEMENT *DEL*

$(\forall y\ ::\ \mathsf{del}.y.[] = [])\ \wedge$
$(\forall y\ l\ x\ ::\ \mathsf{del}.y.(\mathsf{CONS}.x.l) = ((x = y)\ \rightarrow\ l\ \mid\ \mathsf{CONS}.x.(\mathsf{del}.y.l)))$

Definition A.4.14 FIRST ELEMENT OF A LIST *HD*

$\forall x\ l\ ::\ \mathsf{hd}.(\mathsf{CONS}.x.l) = x$

Definition A.4.15 TAIL OF A LIST *TL*

$\forall x\ l\ ::\ \mathsf{tl}.(\mathsf{CONS}.x.l) = l$

Definition A.4.16 INDEXED ELEMENTS *EL*

$\forall l\ ::\ \mathsf{el}.0.l = \mathsf{hd}.l\ \wedge\ \forall n\ l\ ::\ \mathsf{el}.(\mathsf{SUC}.n).l = \mathsf{el}.n.(\mathsf{tl}.l)$

Definition A.4.17 LIST CONTAINS NO DUPLICATE ELEMENTS *NO_DUPLICATES*

$\forall l\ ::\ \mathsf{no\_duplicates}.l = \forall n\ k\ :\ n < \mathsf{length}.l\ \wedge\ k < \mathsf{length}.l\ \wedge\ n \neq k :\ \mathsf{el}.n.l \neq \mathsf{el}.k.l$

A.5 Sets

Definition A.5.1 CHARACTERISTIC SET PREDICATE *IN_DEF*

$\forall s\ x ::\ \mathsf{CHF}.s.x\ =\ (x \in s)$

Definition A.5.2 IMAGE *IMAGE_DEF*

$\forall f\ s ::\ \mathsf{image}.f.s\ =\ \{f.x\ \mid x\ \in\ s\}$

Definition A.5.3 INSERT *INSERT_DEF*

$\forall x\ s ::\ x\ \mathsf{insert}\ s\ =\ \{y\ \mid\ (y = x)\ \vee\ y\ \in\ s\}$

Definition A.5.4 COMPLEMENT OF A SET

$\forall s :: s^{\mathsf{c}}\ =\ \{x \mid x \notin s\}$

Theorem A.5.5 *IN_IMAGE*

$\forall y\ s\ f ::\ y\ \in\ \mathsf{image}.f.s\ =\ (\exists x.\ (y\ =\ (f.x))\ \wedge\ x\ \in\ s)$

Theorem A.5.6 *IMAGE_EQ*

$$\forall f\ g\ s ::\ \frac{\forall x.\ (f.x) = (g.x)}{\mathsf{image}.f.s\ =\ \mathsf{image}.g.s}$$

The predicate FINITE is true of finite sets and false for infinite ones.

Definition A.5.7 Cardinality of sets *set*

$(\mathsf{card}.\{\} = 0)\ \wedge$
$\forall s ::\ \mathsf{FINITE}.s$
$\Rightarrow$
$(\forall x ::\ \mathsf{card}.(x\ \mathsf{insert}\ s) = ((x\ \in\ s) => \mathsf{card}.s\ \mid\ \mathsf{card}.s\ +\ 1))$

Definition A.5.8 converting lists to sets *L2S_DEF*

$(\mathsf{l2s}.[]\ =\ \{\})\ \wedge$
$(\forall x\ l ::\ \mathsf{l2s}.(\mathsf{CONS}.x.l)\ =\ x\ \mathsf{insert}\ (\mathsf{l2s}.l))$

Theorem A.5.9 *L2S_MAP_EQ_IMAGE*

$\forall f\ l ::\ \mathsf{l2s}.(\mathsf{map}.f.l)\ =\ \mathsf{image}.f.(\mathsf{l2s}.l)$

Theorem A.5.10 *L2S_FINITE*

$\forall l ::\ \mathsf{FINITE}.(\mathsf{l2s}.l)$

Theorem A.5.11 *IN_L2S_IS_EL*

$\forall l\ x ::\ (x\ \in\ (\mathsf{l2s}.l))\ =\ (\mathsf{is\_el}.x.l)$

A.6 Converting (finite) sets to lists

Because sets are unordered, a function that converts sets to lists *cannot* be defined by set-induction on finite sets, like e.g. the definition of function card:

$\mathsf{s2l}.\{\} = [] \wedge$
$\forall s ::\ \mathsf{FINITE}.s$
$\Rightarrow$
$(\forall x ::\ \mathsf{s2l}.(x\ \mathsf{insert}\ s) = ((x\ \in\ s) => \mathsf{s2l}.s\ \mid\ \mathsf{CONS}.x.(\mathsf{s2l}.s)))$

Defining s2l this way, reults in that $\mathsf{s2l}.\{2,3,4\}$ is not equal to $\mathsf{s2l}.\{4,3,2\}$, and this is clearly not what we want.

The correct definition of the function s2l has been constructed as follows. First, we define when a function f : num$\rightarrow\alpha$ is a bijection from a subset of num to a set containing elements of type α:

Definition A.6.1 Bijection from subset of num to a set *is_BIJ_NUM_to*

f is_BIJ_NUM_to s
$= \forall n\ m : n < \mathsf{card}.s \wedge m < \mathsf{card}.s : (f.n = f.m) \Rightarrow n = m$
$\wedge\ s = \{f.n \mid n < \mathsf{card}.s\}$

Then we define such a bijection for a set s using Hilbert's choice operator (Section 2.2):

Definition A.6.2 A bijection for a set *set_BIJ*

$\mathsf{set\_BIJ}.s = \varepsilon f.\ f$ is_BIJ_NUM_to s

and prove that it indeed is a bijection as defined in definition A.6.1:

Theorem A.6.3 *set_BIJ_DEF*

$\forall s :: \mathsf{FINITE}.s \Rightarrow \mathsf{set\_BIJ}.s$ is_BIJ_NUM_to s

Now the idea is, to define the function $\mathsf{s2l}.s$ by mapping the set bijection $\mathsf{set\_BIJ}.s$ as defined above on a list containing the elements 0 to $\mathsf{card}.s$.

Definition A.6.4 Converting finite sets to lists *S2L_L2S*

$\mathsf{s2l}.s = \mathsf{map}.(\mathsf{set\_BIJ}.s).(\mathsf{interval}.(\mathsf{card}.s))$

The rest of this section lists some theorems.

Theorem A.6.5 *IN_IS_EL_S2L*

$\forall s :: \mathsf{FINITE}.s \Rightarrow \forall x :: (x \in s) = (\mathsf{is\_el}.x.(\mathsf{s2l}.s))$

Theorem A.6.6 *L2S_S2L_id*

$\forall s :: \mathsf{FINITE}.s \Rightarrow (s = \mathsf{l2s}.(\mathsf{s2l}.s))$

Theorem A.6.7 *NO_DUPLICATES_S2L*

$\forall s :: \mathsf{FINITE}.s \Rightarrow \mathsf{no\_duplicates}.(\mathsf{s2l}.s)$

Theorem A.6.8 *IS_EL_DEL_S2L*

$$\forall s\ x\ y :: \frac{\mathsf{FINITE}.s \wedge \mathsf{is\_el}.x.(\mathsf{s2l}.s) \wedge \mathsf{is\_el}.y.(\mathsf{del}.x.(\mathsf{s2l}.s))}{x \neq y}$$

Theorem A.6.9 *S2L_split*

$$\forall s\ x :: \frac{\mathsf{FINITE}.s \wedge \neg(x \in s)}{\exists l_1\ l_2 :: (\mathsf{s2l}.(x\ \mathsf{insert}\ s) = (l_1 +\!\!+ [x] +\!\!+ l_2)) \wedge (s = \mathsf{l2s}.(l_1 +\!\!+ l_2))}$$

A.7 The iteration operator

Definition A.7.1 ITERATE *iterate*

$$\begin{array}{lcl} \mathsf{iterate}.0.f.x & = & x \\ \mathsf{iterate}.(\mathsf{SUC}.n).f.x & = & f.(\mathsf{iterate}.n.x) \end{array}$$

Theorem A.7.2 *iterate_SUC*

$\forall n\ f\ x\ ::\ \mathsf{iterate}.(\mathsf{SUC}.n).f.x = \mathsf{iterate}.n.f.(f.x)$

Theorem A.7.3 *iterate_PRE*

$\forall n\ : n \geq 1 :\ \forall f\ x\ ::\ \mathsf{iterate}.n.f.x = \mathsf{iterate}.(\mathsf{PRE}.n).f.(f.x)$

where PRE is the predecessor function on natural numbers, characterised as follows:

Theorem A.7.4 *PRE*

$(\mathsf{PRE}.0 = 0)\ \wedge\ (\forall n ::\ \mathsf{PRE}.(\mathsf{SUC}.n) = n)$

Theorem A.7.5 *iterate_FP*

$$\forall n\ f\ x\ y\ ::\ \frac{y = \mathsf{iterate}.n.f.x \wedge y = f.y}{\forall m\ : m \geq n :\ y = \mathsf{iterate}.m.f.x}$$

Theorem A.7.6 *iterate_DISTR_UNION*

$$\forall X\ Y\ f\ ::\ \frac{f.(X \cup Y) = (f.X) \cup (f.Y)}{\forall n\ ::\ \mathsf{iterate}.n.f.(X \cup Y) = \mathsf{iterate}.n.f.X \cup \mathsf{iterate}.n.f.Y}$$

Theorem A.7.7 *LESS_k_iterate_IMP_LESS_PRE_k_iterate_SUC*

$$\forall k\ f\ q\ r\ ::\ \frac{\forall m\ : m < k :\ \mathsf{iterate}.m.f.q \neq r)}{\forall m\ : m < \mathsf{PRE}.k :\ \mathsf{iterate}.(\mathsf{SUC}.m).f.q \neq r}$$

Theorem A.7.8 *LESS_k_iterate_SUC_IMP_LESS_SUC_iterate*

$$\forall k\ f\ q\ r\ ::\ \frac{(q \neq r) \wedge (\forall m\ : m < k :\ \mathsf{iterate}.(\mathsf{SUC}.m).f.q \neq r)}{\forall m\ : m < \mathsf{SUC}.k :\ \mathsf{iterate}.m.f.q \neq r}$$

Theorem A.7.9 *iterate_thm*

For transitive and reflexive relations R:

$$\forall P\ f\ Q\ ::\ \frac{\begin{array}{c}(\forall L\ : L \subseteq P :\ (f.L) \subseteq P) \\ (\forall L\ : L \subseteq P :\ R.(\forall x\ : x \in L :\ Q.x)(\forall x\ : x \in (f.L) :\ Q.x))\end{array}}{\forall n\ L\ : L \subseteq P :\ R.(\forall x\ : x \in L :\ Q.x).(\forall x\ : x \in (\mathsf{iterate}.n.f.L) :\ Q.x)}$$

Theorem A.7.10 *iterate_EQ*

$$\forall n\ f\ g\ x\ ::\ \frac{\forall m\ : m < n :\ f.(\mathsf{iterate}.m.f.x) = g.(\mathsf{iterate}.m.f.x)}{\mathsf{iterate}.n.f.x = \mathsf{iterate}.n.g.x}$$

Theorem A.7.11 *iterate_IN_P*

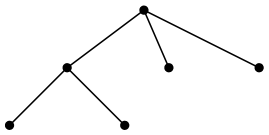
$$\forall f\ P\ ::\ \frac{\forall p\ : p \in P\ : (f.p) \in P}{\forall q\ : q \in P :\ \forall n\ ::\ (\mathsf{iterate}.n.f.q) \in P}$$

A.8 Labelled trees

In the built-in HOL theory tree, a type `tree` is defined to denote the set of all ordered trees of which the nodes can branch any (finite) number of times. A constructor function

node ∈ (tree)list → tree

can be used to construct any value of type tree. For example, the tree:



is denoted by the term:
node.[node.[node.[]; node.[]]; node.[]; node.[]]

Figure A.1: Some tree

The size of a tree is defined to be the number of nodes in that tree:

Definition A.8.1 SIZE OF TREES *size*

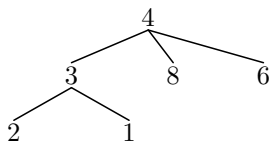
$\forall tl ::$ size.(node tl) = sum.(map.size.tl) + 1

◀

In the built-in HOL theory ltree a type of *labelled* trees (called ('a)ltree) is defined by a type definition, following the way this is described in Section B.1 and [Mel89]. Labelled trees have the same structure as values of the defined type `tree`. The only difference is that a labelled tree of type ('a)ltree has a value associated with each of its nodes. The constructor:

Node ∈ 'a → (('a)ltree)list → ('a)ltree

can be used to construct any value of type ('a)ltree. For example,



is denoted by the term:
Node.4.[Node.3.[Node.2.[]; Node.1.[]]; Node.6.[]; Node.8.[]]

Figure A.2: Some labelled tree

There is a function available that given an labelled tree of type ('a)ltree returns the shape of the tree:

shape.t ∈ ('a)ltree → tree

For example, applying shape to the labelled tree in Figure A.2, returns the tree in Figure A.1. The function that returns the list of values that are associated with the nodes of an labelled tree of type ('a)ltree is:

$$\mathsf{values}.t \in (\texttt{'a})\texttt{ltree} \rightarrow (\texttt{'a})\texttt{list}$$

A pair $(t, l) \in (\texttt{tree} \times (\texttt{'a})\texttt{list})$ for which it holds that the size of t equals the length of l can be used to create a labelled tree:

Definition A.8.2 IS_LTREE *Is_ltree*

$$\forall t\ l ::\ \mathsf{Is\_ltree}.(t, l) = (\mathsf{lenght}.l = \mathsf{size}.t)$$

Theorem A.8.3 CAN CREATE LTREE FROM shape AND values *Is_ltree_REP_ltree_lemma*

$$\forall t ::\ \mathsf{Is\_ltree}.(\mathsf{shape}.t, \mathsf{values}.)$$

◀

There is an induction principle on labelled trees:

Theorem A.8.4 LTREE INDUCTION *ltree_Induct*

$$\forall P ::\ \frac{(\forall t ::\ \mathsf{every}.P.t\ \Rightarrow\ (\forall h ::\ P.(\mathsf{Node}.h.t)))}{\forall l ::\ P.l}$$

◀

Finally, analoguous to the functions on lists, we have (defined in the theory more_ltrees):

Definition A.8.5 MAP ON TREES *MAP_TREE_DEF*

$$\forall v\ t ::\ \mathsf{map\_tree}.f.(\mathsf{Node}.v.t)\ =\ \mathsf{Node}.(f.v).(\mathsf{map}.(\mathsf{map\_tree}.f).t)$$

Definition A.8.6 ZIP ON TREES *ZIP_TREE_DEF*

For all v_1, v_2, t_1, t_2:

$$\frac{\mathsf{length}.t_1 = \mathsf{length}.t_2}{\mathsf{zip\_tree}.(\mathsf{Node}.v_1.t_1, \mathsf{Node}.v_2.t_2)\ =\ \mathsf{Node}.(v_1, v_2).(\mathsf{map.zip\_tree}.(\mathsf{zip}.(t_1, t_2)))}$$

Definition A.8.7 EVERY ON TREES *EVERY_TREE_DEF*

$$(\forall P\ h\ t ::\ \mathsf{every\_tree}.P.(\mathsf{Node}.h.t)\ =\ P.h\ \wedge\ \mathsf{every}.(\mathsf{every\_tree}.P).t$$

◀

Appendix B

Justification for axiomatising the abstract characterisation theorem of `Val`

In this appendix, we will describe how we manually defined the following *not* concrete recursive data type *s*`Val` in HOL.

$$
\begin{array}{lcl}
s\texttt{Val} & = & \mathsf{NUM}\,\texttt{num} \\
& | & \mathsf{SET}\,(\texttt{Val})\texttt{set} \\
& | & \mathsf{LIST}\,(\texttt{Val})\texttt{list} \\
& | & \mathsf{TREE}\,(\texttt{Val})\texttt{ltree}
\end{array}
$$

The contents of this appendix serves as a justification for the axiomatic extension of HOL with the following recursive data type `Val` described in Chapter 5.

$$
\begin{array}{lcl}
\texttt{Val} & = & \mathsf{NUM}\,\texttt{num} \\
& | & \mathsf{BOOL}\,\texttt{bool} \\
& | & \mathsf{REAL}\,\texttt{real} \\
& | & \mathsf{STR}\,\texttt{string} \\
& | & \mathsf{SET}\,(\texttt{Val})\texttt{set} \\
& | & \mathsf{LIST}\,(\texttt{Val})\texttt{list} \\
& | & \mathsf{TREE}\,(\texttt{Val})\texttt{ltree}
\end{array}
$$

Subsection B.1 outlines the general approach one has to follow when manually defining a recursive data type in HOL. Subsection B.2 and B.3 describe the application of this approach to the data type *s*`Val`. All results in this appendix have been mechanically verified with HOL.

B.1 The general approach for defining a new type in HOL

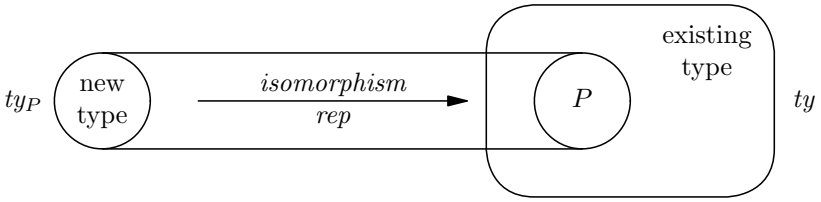
The approach to defining a new logical type as described in [Mel89], involves the following three steps:

1. find an appropriate non-empty subset of an existing type to represent the new type
2. extend the syntax of logical types to include a new type symbol, and use a type definition axiom to relate this new type to its representation
3. derive from the type definition axiom and the properties of the representing type a set of theorems that serves as an "axiomatisation" of the new type.

In the first step, a model for the new type is given by specifying a set of values that will be used to represent it. This is done by defining a predicate P on an existing type such that the set of values satisfying P is non-empty and has exactly the properties that the new type is expected to have.

In the second step, the syntax of types is extended to include a new type symbol which denotes the set of values of the new type. In HOL, this can be done by means of *type definition axioms*, a mechanism formalised by Mike Gordon in [Gor85] which defines a new type by adding a definitional axiom to the logic asserting that the new type is isomorphic to an appropriate non-empty subset of an existing type. The SML function for doing this in HOL is:

```
new_type_definition :  {name:string, pred:term, inhab_thm:thm} → thm
```



If ty is an existing type of the HOL logic, and P is a term of type $ty \to$ `bool` denoting a non-empty[1] (i.e. we can prove $\vdash \exists x :: P\, x$) subset of ty, then evaluating:

`new_type_definition :` $\{$`name = "` ty_P`"`, `pred =` P, `inhab_thm =` $\vdash \exists x :: P\, x\ \}$

results in ty_P being declared as a new type symbol characterised by the following definitional axiom:

[1] Due to the formalisation of Hilbert's ε-operator, HOL types must be non-empty (see also Section 2.2).

$$\vdash \exists rep :: (\forall x\ y :: (rep.x = rep.y) \Rightarrow x = y) \\ \wedge\ (\forall r :: P.r = (\exists x :: r = rep.x)) \qquad (ty_P\_\mathsf{TY\_DEF})$$

which states that the set of values denoted by the new type is isomorphic to, and consequently has the same properties as, the subset of ty specified by P. By adding this definition to the logic, the new type ty_P is defined in terms of existing type ty, and the isomorphism rep can be thought of as a representation function that maps a value of the new type ty_P to the value of type ty that represents it. The type definition axiom (ty_P_TY_DEF) above, asserts only the *existence* of a bijection from ty_P to the corresponding subset of ty. To introduce constants that in fact denote this isomorphism and its inverse, the following SML function is provided:

```
define_new_type_bijections :
            {ABS:string, REP:string, name:string, tyax:thm} → thm
```

If REP_ty_P and ABS_ty_P are the desired names under which to store the representation function and its inverse, then evaluating:

```
define_new_type_bijections
            {ABS = "ABS_tyP", REP = "REP_tyP",
             name = "tyP_ISO_DEF", tyax = tyP_TY_DEF}
```

defines two constants REP_ty_P:$ty \rightarrow ty_P$ and ABS_ty_P:$ty_P \rightarrow ty$, and creates the following theorem which is stored under the name ty_P_ISO_DEF:

$$\vdash (\forall a :: \mathsf{ABS}\_ty_P.(\mathsf{REP}\_ty_P.a) = a) \\ \wedge\ (\forall r :: P.r = (\mathsf{REP}\_ty_P.(\mathsf{ABS}\_ty_P.r) = r)) \qquad ty_P\_\mathsf{ISO\_DEF}$$

It is straightforward to prove that the representation and abstraction functions are injective (one-to-one) and surjective (onto), using provided SML functions:

$$\vdash (\forall a\ a' :: (\mathsf{REP}\_ty_P.a = \mathsf{REP}\_ty_P.a') = (a = a')) \qquad ty_P\_\mathsf{REP\_ONE\_ONE}$$
$$\vdash \forall r :: P.r = (\exists a :: r = \mathsf{REP}\_ty_P.a) \qquad ty_P\_\mathsf{REP\_ONTO}$$
$$\vdash \forall r\ r' :: P.r \Rightarrow P.r' \Rightarrow ((\mathsf{ABS}\_ty_P.r = \mathsf{ABS}\_ty_P.r') = (r = r')) \qquad ty_P\_\mathsf{ABS\_ONE\_ONE}$$
$$\vdash \forall a :: \exists r :: (a = \mathsf{ABS}\_ty_P.r) \wedge P.r \qquad ty_P\_\mathsf{ABS\_ONTO}$$

So, after the second step we actually have the new type ty_P, we know that the values of this type have exactly the same properties as the values in the subset P of ty, and we have a representation and abstraction function to go back and forth between ty_P and ty. Consequently, stating that some property H is true for all elements of the new type ty_P is equal to stating that for all elements in P, H is true of their image under ABS_ty_P:

$$\vdash (\forall x :: (H.x)) = (\forall r :: (P.r) \Rightarrow (H.(\mathsf{ABS}\_ty_P.r))) \qquad ty_P\_\mathsf{PROP}$$

In the third step, a collection of theorems is proved that state abstract characterisations of the new type. These characterisations capture the essential properties of the new type without reference to the way its values are represented and therefore

acts as an abstract "axiomatisation" of it. For an inductive type σ, the assertion of the unique existence of a function g satisfying a recursion equation whose form coincides with the primitive recursion scheme of this type σ – that is, g is a paramorphism [Mee90] – provides an adequate and complete abstract characterisation for σ. From this characterisation it follows that every value of σ is constructed by one or more applications of σ's constructors, and consequently completely determines the values of σ up to isomorphism without reference to the way these are represented. Moreover, in [Mee90] it is proved that all functions with source type σ are expressible in the form of paramorphism g.

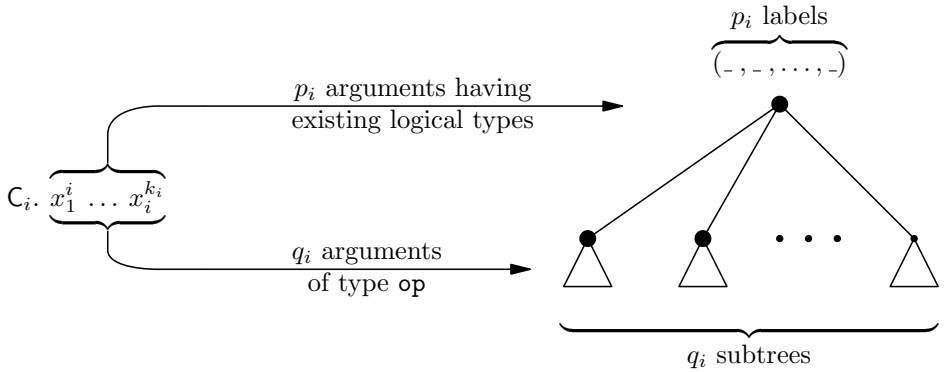
B.2 The representation and type definition

In [Mel89] a concrete recursive type of the form:

$$\begin{array}{lcl} \mathtt{op} & = & \mathsf{C}_1\ t_1^1\ \ldots\ t_1^{k_1} \\ & | & \ldots \\ & | & \mathsf{C}_m\ t_1^m\ \ldots\ t_1^{k_m} \end{array} \tag{B.2.1}$$

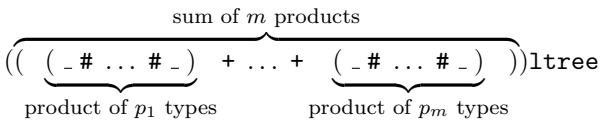
is represented by a non-empty subset of labelled trees (see Appendix A for labelled trees (())`ltree`). Each of the m constructors C_i, $1 \leq i \leq m$, of the concrete recursive data type is represented by a labelled tree using the scheme outlined below:

Let us consider the following instantiation of the ith constructor: $\mathsf{C}_i.x_i^1\ \ldots\ x_i^{k_i}$, where each x_i^j is of type t_i^j which can be an existing logical type, or is the type `op` itself. Let p_i denote the number of arguments which have existing types and let q_i be the number of arguments which have type `op`, where $p_i + q_i = k_i$, then the abstract value of `op` denoted by $\mathsf{C}_i.x_i^1\ \ldots\ x_i^{k_i}$ can be represented by a labelled tree which has p_i values associated with its root node, and q_i subtrees (for the recursive occurences of `op`). In a diagram:



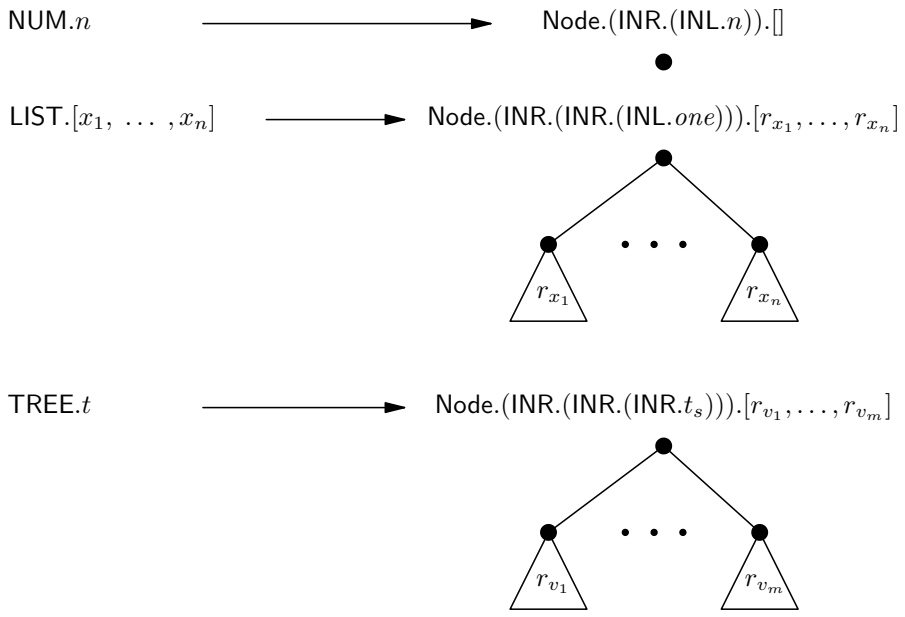
the root of the representing tree is labelled by a p_i-tuple of values. Each of these values is one of the p_i arguments to C_i which are not of type `op`. When $p_i = 0$, the representing tree is labelled with *one*, the one and only element of type `one`. The q_i subtrees shown in the diagram are the representations of the arguments of C_i that have type `op`. When $q_i = 0$, the tree will have no subtrees. Each of the m constructors

can be represented by a labelled tree in this way, and consequently the representing type for `op` will be:



The subset predicate P can now be defined to specify a subset of labelled subtrees of the above type.

This method from [Mel89] only has to be adjusted a bit, in order to represent a subset of the new data type *s*`Val`. Let us be ignorant of the sets for a while, and start using the ideas outlined above. That is, we use ((`one + num + one + tree`))`ltree` as the representing type, and make representations for the constructors NUM, LIST and TREE as follows:

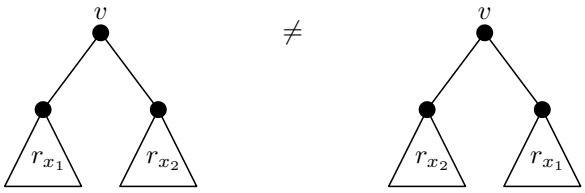


where, r_x denotes the representation as a ((`one + num + one + tree`))`ltree` of value x of type *s*`Val`. The t_s in the root of the representation tree of TREE t denotes the shape of the tree t (i.e. FST.(REP_ltree.t)), and $[v_1, \ldots, v_m]$ is the list of *s*`Val` typed values stored at the nodes of tree t, (i.e. SND.(REP_ltree.t)). Although t_s is not an argument to the constructor TREE we can use this position at the root of the representation tree to store the shape of the *s*`Val` tree which we obviously need in order to be able to go from the representation as a ((`one + num + one + tree`))`ltree` – where all the values of the *s*`Val` tree are put in a list not containing any information about the shape of the original tree – to an abstract value of type *s*`Val`.

As already indicated, the sets in the new type s`Val` constitue a problem when proceeding with the method outlined above. When representing $\mathsf{SET}.\{x_1, \ldots, x_n\}$ as a labelled ltree of which the subtrees are the representations of the values x_1,...,x_n the resulting representation function is *not* an injection, since:

$$\mathsf{SET}.\{x_1, x_2\} = \mathsf{SET}.\{x_2, x_1\}$$

but,



The solution is to represent $\mathsf{SET}.\{x_1, \ldots, x_n\}$ by an equivalence class of ltrees in which ltrees like the two above are considered to be equivalent. Consequently, the existing type to represent our new type s`Val` by shall consist of equivalence classes of ltrees, that is:

$$((\texttt{one + num + one + tree}))\texttt{ltree} \rightarrow \texttt{bool}$$

Before the subset predicate can be defined, we first need to formalise the equivalence relation equiv, that given an ltree of type ((`one + num + one + tree`))`ltree`, returns the equivalence class of that ltree:

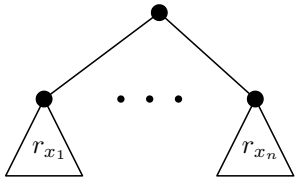
$$\begin{aligned}&\mathsf{equiv} : ((\texttt{one + num + one + tree}))\texttt{ltree} \rightarrow \\ &\qquad ((\texttt{one + num + one + tree}))\texttt{ltree} \rightarrow \texttt{bool}\end{aligned}$$

The representation of a:

$\mathsf{NUM}.n$ value obviously has to consist of the equivalence class containing only the ltree $(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$. Consequently, $\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$ must return a function that only delivers *true* for argument $(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$.

$\mathsf{SET}.\{x_1, \ldots, x_n\}$ value, has to consist of the class containing all ltrees that are equivalent to:

$$\mathsf{Node}.(\mathsf{INL}.one).[r_{x_1}, \ldots, r_{x_n}]$$



For the SET case this must be the class of ltrees:

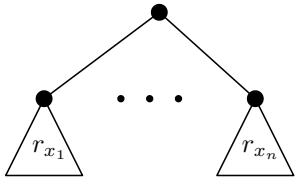
- that have (INL.*one*) at their root
- of which the **sets** of images of their subtrees under equivalence are identical. Note that, because of the absence of ordering in sets, the requirement that these particular sets are identical ensures that two ltrees as displayed on page 230 are equivalent.

Consequently, equiv.(Node.(INL.*one*).tl_1) must return a function that only delivers *true* when given an argument (Node.(INL.*one*).tl_2) such that:

image.equiv.(l2s.tl_1) = image.equiv.(l2s.tl_2)

LIST.xs value, has to consist of the class containing all ltrees that are equivalent to:

Node.(INR.(INR.(INL.*one*))).$[r_{x_1}, \ldots, r_{x_n}]$



For the LIST case this must be the class of ltrees:

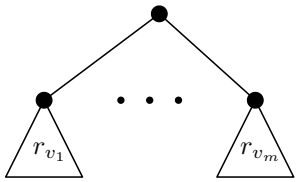
- have (INR.(INR.(INL.*one*))) at their root
- of which the **list** of images of their subtrees under equivalence are identical.

Consequently, equiv.(Node.(INR.(INR.(INL.*one*))).tl_1) must return a function that only delivers *true* for an argument (Node.(INR.(INR.(INL.*one*))).tl_2) such that:

map.equiv.tl_1 = map.equiv.tl_2

TREE.t value, has to consist of the class containing all ltrees that are equivalent to:

Node.(INR.(INR.(INR.t_s))).$[r_{v_1}, \ldots, r_{v_m}]$



Where t_s denotes the shape of tree t, and $[v_1, \ldots, v_m]$ is the list of s`Val` typed values stored at the nodes of tree t. For the TREE case this must be the class of ltrees:

- have (INR.(INR.(INR.t_s))) at their root
- of which the **list** of images of their subtrees under equivalence are identical.

Consequently, equiv.(Node.(INR.(INR.(INR.t_s))).tl_1), must return a function that only delivers *true* for an argument (Node.(INR.(INR.(INR.t_s))).tl_2) such that:

map.equiv.tl_1 = map.equiv.tl_2

Below the formal definition of equiv is given:

Definition B.2.1 EQUIVALENCE RELATION *equiv_DEF*

$$
\begin{array}{l}
\mathsf{equiv}.(\mathsf{Node}.v_1.tl_1).(\mathsf{Node}.v_2.tl_2) = \\
\quad (v_1 = v_2) \\
\wedge \\
\quad ((tl_1 = tl_2 \wedge (\exists n :: \ v_1 = \mathsf{INR}.(\mathsf{INL}.n))) \\
\quad \vee \\
\quad (\mathsf{image}.\mathsf{equiv}.(\mathsf{l2s}.tl_1) = \mathsf{image}.\mathsf{equiv}.(\mathsf{l2s}.tl_2) \wedge \mathsf{ISL}.v_1) \\
\quad \vee \\
\quad (\mathsf{map}.\mathsf{equiv}.tl_1 = \mathsf{map}.\mathsf{equiv}.tl_2) \\
\qquad \wedge \\
\quad (v_1 = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one)) \vee \exists t :: \ v_1 = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t))))
\end{array}
$$

◀

Proving that the relation equiv is an equivalence relation is tedious but straightforward. Using the very nice way to represent equivalence relations from [Har93b], we have:

Theorem B.2.2 equiv IS AN EQUIVALENCE RELATION *equiv_EQUIV_REL*

$$\mathsf{equiv}.t_1.t_2 = (\mathsf{equiv}.t_1 = \mathsf{equiv}.t_2)$$

◀

The subset predicate P that has to specify a non-empty subset of equivalence classes of ltrees can now be defined as the quotient set of an appropriate subset Q of ltrees by the equivalence relation equiv. Looking at the representations of the different s`Val` values, we can see that this Q must contain ltrees ($\mathsf{Node}.v.tl$) for which hold that:

(1) if ($v = \mathsf{INR}.(\mathsf{INL}.n)$) for some number n at their root , then $tl = []$.
(2) if ($v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t))$) for some tree t, then t and tl form an ltree
(3) for all ltrees in tl, (1) and (2) from above also hold.

in a formula:

Definition B.2.3 *Q_DEF*

$$
\begin{array}{l}
Q.(\mathsf{Node}.v.tl) \ = \\
\quad (\exists n :: (v = \mathsf{INR}.(\mathsf{INL}.n))) \Rightarrow tl = [] \\
\wedge \quad (\exists t :: \ v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t))) \Rightarrow \mathsf{ls\_ltree}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)), tl) \\
\wedge \quad (\forall t :: \ t \in tl \Rightarrow Q.t)
\end{array}
$$

◀

Finally, the subset predicate P is defined as the quotient set of Q by equiv:

Definition B.2.4 *Is_pvt_REP*
$P = Q/\mathsf{equiv}$

Theorem B.2.5 *Is_pvt_REP_THM*
$P = (\lambda s.\, \exists t :: (s = \mathsf{equiv}.t) \wedge (Q.t))$

◀

It is easy to prove that P is not empty, and consequently we can use SML functions `new_type_definition` and `define_new_type_bijections` to extend the syntax of logical types to include our new type $s\mathtt{Val}$, define the type bijections $\mathsf{ABS}\_s\mathtt{Val}$ and $\mathsf{REP}\_s\mathtt{Val}$ between $s\mathtt{Val}$ and P, and prove that these are injective and surjective:

Definition B.2.6 *s*Val_ISO_DEF
$(\forall a :: \mathsf{ABS}\_s\mathtt{Val}.(\mathsf{REP}\_s\mathtt{Val}.a) = a) \wedge (\forall r :: P.r = (\mathsf{REP}\_s\mathtt{Val}.(\mathsf{ABS}\_s\mathtt{Val}.r) = r))$

Theorem B.2.7 *s*Val_REP_ONE_ONE
$(\forall a\, a' :: (\mathsf{REP}\_s\mathtt{Val}.a = \mathsf{REP}\_s\mathtt{Val}.a') = (a = a'))$

Theorem B.2.8 *s*Val_REP_ONTO
$\forall r :: P.r = (\exists a :: r = \mathsf{REP}\_s\mathtt{Val}.a)$

Theorem B.2.9 *s*Val_ABS_ONE_ONE
$\forall r\, r' :: P.r \Rightarrow P.r' \Rightarrow ((\mathsf{ABS}\_s\mathtt{Val}.r = \mathsf{ABS}\_s\mathtt{Val}.r') = (r = r'))$

Theorem B.2.10 *s*Val_ABS_ONTO
$\forall a :: \exists r :: (a = \mathsf{ABS}\_s\mathtt{Val}.r) \wedge P.r$

Theorem B.2.11 *s*Val_PROP
$(\forall x :: (H.x)) = (\forall r :: (P.r) \Rightarrow (H.(\mathsf{ABS}\_s\mathtt{Val}.r)))$

◀

B.3 The axiomatisation

The abstract axiomatisation of $s\mathtt{Val}$ will be based upon four constructors:

NUM	:	$\mathtt{num} \rightarrow s\mathtt{Val}$
SET	:	$(s\mathtt{Val})\mathtt{set} \rightarrow s\mathtt{Val}$
LIST	:	$(s\mathtt{Val})\mathtt{list} \rightarrow s\mathtt{Val}$
TREE	:	$(s\mathtt{Val})\mathtt{ltree} \rightarrow s\mathtt{Val}$

To define the constructors, we need a function that given an equivalence class of ltrees returns an element of that equivalence class. We will call this function pick, and define it using Hilbert's ε-operator (see Section 2.2). It satisfies the following properties:

Definition B.3.1 pick *pick*

$\mathsf{pick}.c = \varepsilon t.\ c.t$

Theorem B.3.2 *equiv_pick_REP_pvt*

$\mathsf{equiv} \circ \mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val} = \mathsf{REP\_}s\mathtt{Val}$

Theorem B.3.3 *Q_pick_REP_pvt*

$\forall x :: Q.((\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).x)$

◀

Now the constructors can be defined as follows (see Appendix A for the definition of s2l).

Definition B.3.4 *NUM_DEF*

$\mathsf{NUM}.n = \mathsf{ABS\_}s\mathtt{Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]))$

Definition B.3.5 *SET_DEF*

$\mathsf{SET}.s = \mathsf{ABS\_}s\mathtt{Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).(\mathsf{s2l}.s))))$

Definition B.3.6 *LIST_DEF*

$\mathsf{LIST}.l = \mathsf{ABS\_}s\mathtt{Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).l)))$

Definition B.3.7 *TREE_DEF*

$$\mathsf{TREE}.t = \mathsf{ABS\_}s\mathtt{Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t)))))\ .(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).(\mathsf{values}.t)))$$

◀

Having defined the constructors, the theorem which abstractly characterises the new type $s\mathtt{Val}$, by stating the unique existence of a paramorphism para has to be proved.

Theorem B.3.8 Abstract characterisation of $s\mathtt{Val}$ *pvt_Axiom*

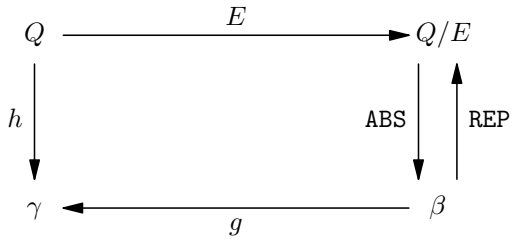
$$\begin{array}{l}
\forall f_n\ f_s\ f_l\ f_t ::\\
\quad \exists!\mathsf{para} ::\\
\qquad (\forall n ::\ \mathsf{para}.(\mathsf{NUM}.n) = f_n.n)\\
\quad \wedge\\
\qquad (\forall s ::\ (\mathsf{FINITE}.s) \Rightarrow (\mathsf{para}.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split.para}).s)))\\
\quad \wedge\\
\qquad (\forall l ::\ \mathsf{para}.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split.para}).l))\\
\quad \wedge\\
\qquad (\forall t ::\ \mathsf{para}.(\mathsf{TREE}.t) = f_t.(\mathsf{map_tree}.(\mathsf{split.para}).t))
\end{array}$$

◀

The proof of Theorem B.3.8 consists of two parts, the proof of the existence of a paramorphism para, and the proof that such a paramorphism is unique.

The existence proof is based upon the following theorem about quotient sets. Informally, this theorem states that: for all equivalence relations E on α, and subsets

Q on α, if ABS and REP are mutually inverse bijections between some set β and Q/equiv, and h is a function of type $\alpha \to \gamma$ that does not disthinguish between different elements in the same equivalence class, then there exists a unique function g of type $\beta \to \gamma$ such that the following diagram commutes:



Theorem B.3.9 QUOTIENT SETS *QUOTIENT_THM*

For all equivalence relations E on α; for all Q defining a subset of α; for all ABS : (α $\to$ `bool`) $\to \beta$ and REP : $\beta \to (\alpha \to$ `bool`), abstraction and representation functions respectively, the following theorem holds for all functions h of type $\alpha \to \gamma$:

$$\frac{\begin{array}{c}(\forall a :: \mathsf{ABS}.(\mathsf{REP}.a) = a) \ \wedge \ (\forall r :: \ ((Q/E).r) = (\mathsf{REP}.(\mathsf{ABS}.r) = r)) \\ (\forall t_1\ t_2 :: \ (E.t_1.t_2) \ \Rightarrow \ (h.t_1 = h.t_2))\end{array}}{\exists! g :: \ \forall t :: \ (Q.t) \ \Rightarrow \ (g.(\mathsf{ABS}.(E.t)) = (h.t))}$$

Instantiating Theorem B.3.9 with ((`one` + `num` + `one` + `tree`))`ltree` for α, equiv for E, ABS_s`Val` for ABS, REP_s`Val` for REP, and Q, obviously makes g a good candidate for para. Applying modus ponens to this instantiation and s`Val`_ISO_DEF gives us a unique function g of type s`Val` $\to \gamma$ for which it holds that:

$$\frac{\forall t_1\ t_2 :: \ (\mathsf{equiv}.t_1.t_2) \ \Rightarrow \ (h.t_1 = h.t_2)}{\forall t :: \ (Q.t) \ \Rightarrow \ (g.(\mathsf{ABS\_}s\texttt{Val}.(\mathsf{equiv}.t)) = (h.t))} \tag{B.3.1}$$

Using theorem B.3.3 (and Theorem A.8.3), it is easy to prove that:

Theorem B.3.10 *Q_NUM_REP*

$\forall n :: \ Q \ (\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$

Theorem B.3.11 *Q_SET_REP*

$\forall s :: \ Q.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}).(\mathsf{s2l}.s)))$

Theorem B.3.12 *Q_LIST_REP*

$\forall l :: Q.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}).l))$

Theorem B.3.13 *Q_TREE_REP*

$\forall t :: Q.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t))))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}).(\mathsf{values}.t)))$

These theorems together with (B.3.1) and the definitions of the constructors give us:

$$\forall t_1\ t_2 :: (\mathsf{equiv}.t_1.t_2) \Rightarrow (h.t_1 = h.t_2)$$

$$\begin{array}{lcl}
\forall n :: g.(\mathsf{NUM}.n) & = & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]) \\
\forall s :: g.(\mathsf{SET}.s) & = & h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_}s\mathtt{Val}).(\mathsf{s2l}.s))) \\
\forall l :: g.(\mathsf{LIST}.l) & = & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_}s\mathtt{Val}).l)) \\
\forall t :: g.(\mathsf{TREE}.t) & = & h.\ (\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t)))) \\
 & & \qquad .(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_}s\mathtt{Val}).(\mathsf{values}.t)))
\end{array}$$

Consequently, in order to finish the existence part of the proof of Theorem B.3.8, we have to find a function h, that satisfies the following properties:

$$\begin{array}{ll}
(\boldsymbol{i}). & \forall t_1\ t_2 :: (\mathsf{equiv}.t_1.t_2) \Rightarrow (h.t_1 = h.t_2) \\
(\boldsymbol{ii}). & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]) \\
 & \quad = f_n.n \\
(\boldsymbol{iii}). & h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_}s\mathtt{Val}).(\mathsf{s2l}.s))) \\
 & \quad = f_s.(\mathsf{image}.(\mathsf{split}.g).s), \text{ for all finite sets } s \\
(\boldsymbol{iv}). & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_}s\mathtt{Val}).l)) \\
 & \quad = f_l.(\mathsf{map}.(\mathsf{split.para}).l) \\
(\boldsymbol{v}). & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t))))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_}s\mathtt{Val}).(\mathsf{values}.t))) \\
 & \quad = f_t.(\mathsf{map_tree}.(\mathsf{split.para}).t)
\end{array}$$

We claim that the following function, defined by "primitive recursion" on ltrees, satisfies these conditions, and to finish the proof of the existence part of theorem B.3.8, it only remains for us to validate this claim:

$$\begin{array}{l}
\forall v\ tl :: h.(\mathsf{Node}.v.tl) = k.(\mathsf{map}.h.tl).v.tl, \text{ where:} \\
\quad k = \lambda xs\ v\ tl. \\
\qquad \mathsf{ISL}.v \rightarrow f_s.(\mathsf{l2s}.(\mathsf{zip}.(xs, (\mathsf{map}.(\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).tl)))) \\
\qquad\quad | \quad \mathsf{ISL}.(\mathsf{OUTR}.v) \texttt{ => } f_n.(\mathsf{OUTL}.(\mathsf{OUTR}.v)) \\
\qquad\quad | \quad \mathsf{ISL}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)) \texttt{ => } f_l.(\mathsf{zip}.(xs, (\mathsf{map}.(\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).tl))) \\
\qquad\quad | \quad f_t.(\mathsf{zip_tree} \\
\qquad\qquad ((\mathsf{ABS_ltree}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)), xs)), \\
\qquad\qquad (\mathsf{ABS_ltree}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)), \mathsf{map}.(\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).tl))))
\end{array}$$

In order to prove requirement ($\boldsymbol{i}$), we have the following theorem, the proof of which is straightforward and tedious and hence will not be given here.

Theorem B.3.14 *ltree_Axiom_PRESERVES_equiv*

For all functions h of type $((\texttt{one + num + one + tree}))\texttt{ltree} \to \gamma$ defined by "primitive recursion" on $((\texttt{one + num + one + tree}))\texttt{ltree}$s (i.e. having the form $(\forall v\, tl :: h.(\mathsf{Node}.v.tl) = k.(\mathsf{map}.h.tl).v.tl)$ for an arbitrary function k of type $(\gamma)\texttt{list} \to ((\texttt{one + num + one + tree}))\texttt{list} \to \gamma$):

$$\frac{\begin{array}{l}\forall xs_1\ xs_2\ tl_1\ tl_2\ v :: \\ \quad (\mathsf{equiv}.(\mathsf{Node}.v.tl_1).(\mathsf{Node}.v.tl_2) \wedge \\ \quad\ (\mathsf{ISL}.v)\ \Rightarrow\ (\mathsf{length}.xs_1 = \mathsf{length}.tl_1)\ \wedge\ (\mathsf{length}.xs_2 = \mathsf{length}.tl_2) \wedge \\ \qquad\qquad ((\mathsf{l2s}.(\mathsf{zip}.(xs_1, \mathsf{map}.\mathsf{equiv}.tl_1))) = (\mathsf{l2s}.(\mathsf{zip}.(xs_2, \mathsf{map}.\mathsf{equiv}.tl_2))) \\ \quad\ (\mathsf{ISR}.v)\ \Rightarrow\ (xs_1 = xs_2)) \\ \quad \Rightarrow \\ \quad ((k.xs_1.v.tl_1) = (k.xs_2.v.tl_2))\end{array}}{\forall t_1\ t_2 :: (\mathsf{equiv}.t_1.t_2)\ \Rightarrow\ (h.t_1 = h.t_2)}$$

◀

Proving that our function k satisfies the premise of theorem B.3.14 is again straightforward and tedious since, as the proof of theorem B.3.14 itself, it involves lots of lemmas involving ZIP. We shall not give this proof here either, and consider (***i***) to be proven.

It is easy to prove that h satisfies property (***ii***):

$\quad h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$
$= $ (definition h and k)
$\quad f_n.(\mathsf{OUTL}.(\mathsf{OUTR}.(\mathsf{INR}.(\mathsf{INL}.n))))$
$= $ (OUTL, OUTR, INR, INL)
$\quad f_n.n$

In order to show that h satisfies property (***iii***), we first need to prove:

$$\forall s :: \mathsf{map}.(h \circ \mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}).(\mathsf{s2l}.s) = \mathsf{map}.g.(\mathsf{s2l}.s) \tag{B.3.2}$$

proof of B.3.2
Let us consider an arbitrary set s of $s\texttt{Val}$-typed values. Since $h \circ \mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}$ is mapped only on elements in $(\mathsf{s2l}\ s)$, it will be sufficient to prove:
$\quad \forall t : t \in (\mathsf{s2l}\ s) : g = (h \circ \mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val})$
From the definition of Q (B.2.3) and theorem B.3.11, we know that:
$\quad \forall x : x \in (\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}).(\mathsf{s2l}.s)) : (Q.x)$ holds,
and thus (Theorem A.4.10)
$\quad \forall t : t \in (\mathsf{s2l}.s) : (Q.((\mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}).t))$
From (***i***) and (B.3.1) we can now deduce that
$\quad \forall t : t \in (\mathsf{s2l}.s) : (g \circ \mathsf{ABS\_}s\texttt{Val} \circ \mathsf{equiv} \circ \mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val}) = (h \circ \mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val})$
which with theorem B.3.2 and $s\texttt{Val\_ISO\_DEF}$, rewrites to:
$\quad \forall t : t \in (\mathsf{s2l}.s) : g = (h \circ \mathsf{pick} \circ \mathsf{REP\_}s\texttt{Val})$
end proof of B.3.2

Now we can proceed with the proof of property (***iii***) as follows:

$$
\begin{array}{l}
\quad h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).(\mathsf{s2l}.s))) \\
= \text{(definition } h\text{)} \\
\quad k.(\mathsf{map}.h.(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).(\mathsf{s2l}.s))).(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).(\mathsf{s2l}.s)) \\
= (\mathsf{map} \text{ composition (Theorem A.4.8) and B.3.2)} \\
\quad k.(\mathsf{map}.g.(\mathsf{s2l}.s)).(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).(\mathsf{s2l}.s)) \\
= \text{(definition } k\text{)} \\
\quad f_s.(\mathsf{l2s}.(\mathsf{zip}.(\mathsf{map}.g.(\mathsf{s2l}.s), \mathsf{map}.(\mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_}s\mathtt{Val}).(\mathsf{s2l}.s))))) \\
= (\mathsf{map} \text{ composition (Theorem A.4.8), theorem B.3.2 and } s\mathtt{Val}\_\mathsf{ISO\_DEF}) \\
\quad f_s.(\mathsf{l2s}.(\mathsf{zip}.(\mathsf{map}.g.(\mathsf{s2l}.s), (\mathsf{s2l}.s)))) \\
= (\mathsf{zip} \text{ and } \mathsf{split} \text{ (Theorem A.4.9))} \\
\quad f_s.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{split}.g).(\mathsf{s2l}.s))) \\
= (\mathsf{l2s}, \mathsf{map} \text{ and } \mathsf{image} \text{ (Theorem A.5.9))} \\
\quad f_s.(\mathsf{image}.(\mathsf{split}.g).(\mathsf{l2s}.(\mathsf{s2l}.s))) \\
= (s \text{ is a finite set (Theorem A.6.6))} \\
\quad f_s.(\mathsf{image}.(\mathsf{split}.g).s)
\end{array}
$$

The proofs of properties (***iv***) and (***v***) are similar to the proof of (***iii***) and will not be given. We hereby finish the proof of the existence part of theorem B.3.8, and continue with the proof that the existing paramorphism is unique. That is we shall prove that: for all function x and y of type $s\mathtt{Val} \to \gamma$:

$$
\frac{\begin{array}{l}
\forall n ::\ x.(\mathsf{NUM}.n) = f_n.n \\
\forall s ::\ \mathsf{finite}.s \Rightarrow (x.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.x).s)) \\
\forall l ::\ x.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.x).l) \\
\forall t ::\ x.(\mathsf{TREE}.t) = f_t.(\mathsf{map\_tree}.(\mathsf{split}.x).t) \\
\forall n ::\ y.(\mathsf{NUM}.n) = f_n.n \\
\forall s ::\ \mathsf{finite}.s \Rightarrow (y.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.y).s)) \\
\forall l ::\ y.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.y).l) \\
\forall t ::\ y.(\mathsf{TREE}.t) = f_t.(\mathsf{map\_tree}.(\mathsf{split}.y).t)
\end{array}}{(x = y)} \qquad \text{(B.3.3)}
$$

In order to be able to prove this, we first need an induction theorem for type $s\mathtt{Val}$.

Theorem B.3.15 INDUCTION ON $s\mathtt{Val}$ *pvt_Induct*

For all properties H,

$$
\frac{\begin{array}{l}
\forall n ::\ H.(\mathsf{NUM}.n) \\
\forall s ::\ ((\mathsf{finite}.s) \wedge (\forall p ::\ (p \in s) \Rightarrow (H.p))) \Rightarrow (H.(\mathsf{SET}.s)) \\
\forall l ::\ (\mathsf{every}.H.l) \Rightarrow (H.(\mathsf{LIST}.l)) \\
\forall t ::\ (\mathsf{every\_tree}.H.t) \Rightarrow (H.(\mathsf{TREE}.t))
\end{array}}{\forall p ::\ H.p}
$$

◀

The proof of this induction theorem is not too hard. Here we shall only give a sketchy proof to give the reader an idea. We start with the following lemma, that is easy to prove using sVal_PROP.

Lemma B.3.16 *induct_lemma4*

$(\forall p :: H.p) = (\forall t\ r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow (H \circ \mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).t)$

To prove theorem B.3.15 we assume:

$\mathbf{A}_1$) $\forall n :: H.(\mathsf{NUM}.n)$
$\mathbf{A}_2$) $\forall s :: ((\mathsf{finite}.s) \wedge (\forall p :: (p \in s) \Rightarrow (H.p))) \Rightarrow (H.(\mathsf{SET}.s)))$
$\mathbf{A}_3$) $\forall l :: (\mathsf{every}.H.l) \Rightarrow (H.(\mathsf{LIST}.l))$
$\mathbf{A}_4$) $\forall t :: (\mathsf{every\_tree}.H.t) \Rightarrow (H.(\mathsf{TREE}.t))$

we have to prove that:

$(\forall p :: H.p)$
(= lemma B.3.16)
$(\forall t\ r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow (H \circ \mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).t)$
(⇐ `ltree` induction (Theorem A.8.4) and definition of every (Definition A.4.6))
for arbitrary h and tl, we have to prove:

$$\frac{\begin{array}{c}((r = \mathsf{equiv}.(\mathsf{Node}.h.tl)) \wedge (Q.(\mathsf{Node}.h.tl))) \\ \forall t :: (t \in tl) \Rightarrow \forall r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow ((H \circ \mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).t)\end{array}}{((H \circ \mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).(\mathsf{Node}.h.tl))}$$

Moving the antecedents of this proof obligation into the assumptions, we get for an arbitrary h and tl that:

$\mathbf{A}_5$) $\forall t :: (t \in tl) \Rightarrow (\forall r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow ((H \circ \mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).t))$
$\mathbf{A}_6$) $r = \mathsf{equiv}.(\mathsf{Node}.h.tl)$
$\mathbf{A}_7$) $Q.(\mathsf{Node}.h.tl)$

The proof that $(H \circ \mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).(\mathsf{Node}.h.tl)$, now proceeds by case distinction on h. We shall prove the SET case (i.e. $\mathsf{ISL}.h$), the other cases are similar. So suppose:

$\mathbf{A}_8$) $\mathsf{ISL}.h$

From the definition of equiv, and the properties of Q, pick, ABS_sVal and REP_sVal it follows that:

Lemma B.3.17 *SET_L2S_EQ_ABS*

For all lists tl of ((one + num + one + tree))ltrees:

$$\frac{\forall t :: (t \in tl) \Rightarrow (Q.t)}{(\mathsf{SET}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS\_}s\mathtt{Val} \circ \mathsf{equiv}).tl))) = (\mathsf{ABS\_}s\mathtt{Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).tl)))}$$

◀

Continuing with the proof of B.3.15:

$$\begin{array}{ll}
 & (H \circ \mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).(\mathsf{Node}.h.tl) \\
= & (\mathbf{A}_8, \text{ the type of } h, \textit{one}, \text{ and } \circ) \\
 & H.(\mathsf{ABS_}s\mathtt{Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).tl))) \\
= & (\text{rewriting assumption } \mathbf{A}_7 \text{ with } Q, \text{ and lemma B.3.17}) \\
 & H.(\mathsf{SET}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).tl))) \\
\Leftarrow & (\text{assumption } \mathbf{A}_2) \\
 & \mathsf{finite}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).tl)) \\
 & \wedge \\
 & \forall p :: (p \in (\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).tl))) \Rightarrow (H.p) \\
= & (\text{lists are } \mathsf{finite} \text{ (Theorem A.5.10)}) \\
 & \forall p :: (p \in (\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).tl))) \Rightarrow (H.p) \\
= & (\text{element of } \mathsf{l2s} \text{ and } \mathsf{map} \text{ (Theorems A.5.9, A.5.5 and A.5.11)}) \\
 & \forall p :: (\exists t :: (t \in tl) \wedge (((\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).t) = p)) \Rightarrow (H.p)
\end{array}$$

Making the antecedents of this proof obligation into assumptions, gives us an t, such that for arbitrary p:

$$\begin{array}{ll}
 & \mathbf{A}_9) \ t \in tl \\
 & \mathbf{A}_{10}) \ p = ((\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).t)
\end{array}$$

leaving us with proof obligation:

$$\begin{array}{ll}
 & H.p \\
= & (\text{assumption } \mathbf{A}_{10}) \\
 & H.((\mathsf{ABS_}s\mathtt{Val} \circ \mathsf{equiv}).t) \\
\Leftarrow & (\text{Modus ponens assumption } \mathbf{A}_9 \text{ and the Induction Hypothesis } (\mathbf{A}_5)) \\
 & \exists r :: (r = \mathsf{equiv}.t) \wedge (Q.t) \\
= & (\text{rewriting assumption } \mathbf{A}_7 \text{ with } Q, \text{ and assumption } \mathbf{A}_9) \\
 & \exists r :: (r = \mathsf{equiv}.t)
\end{array}$$

Instantiating with $\mathsf{equiv}\ t$ proves this case. As already indicated the other cases (where $\mathsf{ISR}\ h$) are similar, the NUM case is trivial, and for the LIST and TREE cases, theorems similar to B.3.17 had to be proved (see Appendix 8.10.3, LIST_EQ_ABS and TREE_EQ_ABS respectively).

Now that an induction theorem on $s\mathtt{Val}$ is available, it is straightforward to prove the uniqueness. Assume the premises of proof obligation (B.3.3). We have to prove:

$$
\begin{array}{l}
\quad x \;=\; y \\
= \text{ (function equality)} \\
\quad \forall p :: (x.p) = (y.p) \\
\Leftarrow \text{ (}s\texttt{Val}\text{ Induction, } H = (\lambda p.\ (x.p) = (y.p))) \\
\quad \forall n :: \ (x.(\mathsf{NUM}.n)) = (y.(\mathsf{NUM}.n)) \\
\quad \wedge \\
\quad \forall s :: ((\mathsf{finite}.s) \wedge (\forall p :: (p \in s) \Rightarrow ((x.p) = (y.p))) \Rightarrow ((x.(\mathsf{SET}.s)) = (y.(\mathsf{SET}.s)))) \\
\quad \wedge \\
\quad \forall l :: \ (\mathsf{every}.(\lambda p.\ (x.p) = (y.p)).l) \Rightarrow ((x.(\mathsf{LIST}.l)) = (y.(\mathsf{LIST}.l))) \\
\quad \wedge \\
\quad \forall t :: \ (\mathsf{every\_tree}.(\lambda p.\ (x.p) = (y.p)).t) \Rightarrow ((x.(\mathsf{TREE}.t)) = (y.(\mathsf{TREE}.t)))
\end{array}
$$

The first conjunct immediately follows from the premises of (B.3.3). We shall continue to prove the SET case, the LIST and TREE cases are similar. Suppose, for an arbitrary set s with s`Val` typed values:

$$
\begin{array}{l}
\mathbf{A'}_1)\ \mathsf{finite}.s \\
\mathbf{A'}_2)\ \forall p :: \ (p \ \in \ s) \Rightarrow ((x.p) = (y.p))
\end{array}
$$

we have to prove that: $(x.(\mathsf{SET}.s)) = (y.(\mathsf{SET}.s))$. From the premises of (B.3.3), we can deduce that:

$$
\begin{array}{l}
\mathbf{A'}_3)\ (x.(\mathsf{SET}.s)) = f_s.(\mathsf{image}.(\mathsf{split}.x).s) \\
\mathbf{A'}_4)\ (y.(\mathsf{SET}.s)) = f_s.(\mathsf{image}.(\mathsf{split}.y).s)
\end{array}
$$

$$
\begin{array}{l}
\quad (x.(\mathsf{SET}.s)) = (y.(\mathsf{SET}.s)) \\
= \text{ (assumptions } \mathbf{A'}_3 \text{ and } \mathbf{A'}_4) \\
\quad f_s.(\mathsf{image}.(\mathsf{split}.x).s) = f_s.(\mathsf{image}.(\mathsf{split}.y).s) \\
\Leftarrow \\
\quad (\mathsf{image}.(\mathsf{split}.x).s) = (\mathsf{image}.(\mathsf{split}.y).s) \\
\Leftarrow \text{ (Theorem A.5.6)} \\
\quad \forall p :: \ (p \ \in \ s) \Rightarrow ((\mathsf{split}.x.p) = (\mathsf{split}.y.p)) \\
= \text{ (definition split)} \\
\quad \forall p :: \ (p \ \in \ s) \Rightarrow (((x.p), p) = ((y.p), p)) \\
= \text{ (pairs)} \\
\quad \forall p :: \ (p \ \in \ s) \Rightarrow (x.p) = (y.p)
\end{array}
$$

Assumption $\mathbf{A'}_2$ proves this SET case, and, as indicated, the LIST and TREE cases are similar. This completes the outline of the uniqueness part, and consequently the entire proof, of the abstract characterisation theorem of s`Val` (Theorem B.3.8).

B.4 Defining the recursive data type `Val`

It is not difficult to see that manually adding the recursive data type `Val` to HOL can be done analoguos to the way s`Val` is added. The representation, abstract char-

acterisation and proof obligations for the additional constructors BOOL, REAL and STRING, will be analogous to those of NUM. However, as the number of constructors increase, so does the number of INRs and INLs, and consequently the proofs will become long and tedious. Since all formal proofs necessary to prove the abstract characterisation theorem of the subtype s`Val` have been verified in HOL, we are convinced that the abstract characterisation theorem of `Val` can also be proved. Consequently, we add the abstract characterisation theorem of data type `Val` as an axiom, using the following approach.

First, we define a new constant representing our equivalence relation:

```
val plus = ty_antiq (==‘:one + num + bool + real + string + one + tree‘==);

new_constant {Name="equiv", Ty = ==‘:(^plus)ltree -> (^plus)ltree -> bool‘==};
```

Then, we define the subset predicate P' analogous to the way the subset predicate P is defined for s`Val`:

Definition B.4.1 *Q_t_DEF*

$$
\begin{array}{ll}
Q'.(\mathsf{Node}.v.tl) \;=\; & \\
\quad & (\exists n :: (v = \mathsf{INR}.(\mathsf{INL}.n))) \Rightarrow tl = [] \\
\quad & (\exists b :: (v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.b)))) \Rightarrow tl = [] \\
\quad & (\exists r :: (v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.r))))) \Rightarrow tl = [] \\
\quad & (\exists str :: (v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.str)))))) \Rightarrow tl = [] \\
\wedge \quad & (\exists t :: \; v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t)))))) \\
\quad & \Rightarrow \mathsf{Is\_ltree}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.v))))), tl) \\
\wedge \quad & (\forall t :: \; t \in tl \Rightarrow Q'.t)
\end{array}
$$

Definition B.4.2 *Is_pvt_REP*

$P' \;=\; Q'/$`equiv`

◀

Again, it is not difficult to prove that P' is not empty, and consequently we can use `new_type_definition` to extend the syntax of logical types to include our new type `Val`. Subsequently, we define the constructor functions as constants of the desired types:

```
new_constant {Name="NUM", Ty = ==‘:num -> Val‘==};
new_constant {Name="BOOL", Ty = ==‘:bool -> Val‘==};
new_constant {Name="REAL", Ty = ==‘:real -> Val‘==};
new_constant {Name="STR", Ty = ==‘:string -> Val‘==};
new_constant {Name="SET", Ty = ==‘:(Val)set -> Val‘==};
new_constant {Name="LIST", Ty = ==‘:(Val)list -> Val‘==};
new_constant {Name="TREE", Ty = ==‘:(Val)ltree -> Val‘==};
```

Finally, we add the abstract characterisation theorem of data type `Val` as an axiom using SML function `new_open_axiom`.

Appendix C

Proofs of the refinement theorems

This appendix presents detailed proofs of the Theorems $7.2.11_{112}$ and $7.2.7_{113}$ stating the conditions under which unless and $\rightarrowtail$ properties are preserved under refinement. The other theorems in Chapter 7 are corollaries of these two theorems.

C.1 Preservation of unless

Theorem 7.2.11 *P_ref_AND_SUPERPOSE_WRITE_PRESERVES_UNLESSe*

$$\frac{P \sqsubseteq_{\mathcal{R},J} Q \wedge \mathsf{Unity}.P \wedge \mathsf{Unity}.Q \wedge (\ {}_Q\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J) \quad \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (p\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (q\ \mathcal{C}\ W^{\mathsf{c}})}{{}_P\vdash p \text{ unless } q \Rightarrow\ {}_Q\vdash (J_Q \wedge p) \text{ unless } q}$$

◀

---

proof of 7.2.11

Assume the following:

$\mathbf{A}_1$: $P \sqsubseteq_{\mathcal{R},J} Q$
$\mathbf{A}_2$: $\mathsf{Unity}.P \wedge \mathsf{Unity}.Q$
$\mathbf{A}_3$: $(\ {}_Q\vdash \circlearrowleft J_Q) \wedge (J_Q \Rightarrow J)$
$\mathbf{A}_4$: $\mathbf{w}Q = \mathbf{w}P \cup W$
$\mathbf{A}_5$: $(p\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (q\ \mathcal{C}\ W^{\mathsf{c}})$
$\mathbf{A}_6$: ${}_P\vdash p \text{ unless } q$

From $\mathbf{A}_5$ and the definition of confinement $(3.3.17_{28})$ we can infer:

$\mathbf{A}_7$: $\forall t, t' :\ (t \restriction W^{\mathsf{c}} = t' \restriction W^{\mathsf{c}}) \Rightarrow (p.t = p.t' \wedge q.t = q.t')$

From $\mathbf{A}_6$, the definitions of unless $(4.4.1_{43})$, and the definition of Hoare triples $(3.5.1_{38})$ we can infer:

$\mathbf{A}_8$: $\forall A_P : A_P \in \mathbf{a}P :$
$\quad \forall s, t : \mathsf{compile}.A_P.s.t \wedge \mathsf{evalb}.(p.s) \wedge \neg(\mathsf{evalb}.(q.s)) \Rightarrow \mathsf{evalb}.(p.t) \vee \mathsf{evalb}.(q.t)$

Now we have to prove the following:

${}_Q\vdash (J_Q \wedge p)$ unless q

= (Definitions of unless $(4.4.1_{43})$ and Hoare triples $(3.5.1_{38})$)

$\forall A_Q \in \mathbf{a}Q : \forall s, t :$
$\quad \mathsf{compile}.A_Q.s.t \wedge \mathsf{evalb}.(J_Q.s) \wedge \mathsf{evalb}.(p.s) \wedge \neg(\mathsf{evalb}.(q.s))$
$\quad \Rightarrow$
$\quad (\mathsf{evalb}.(J_Q.t) \wedge \mathsf{evalb}.(p.t)) \vee \mathsf{evalb}.(q.t)$

Choose an arbitrary A_Q, and assume for arbitrary states s and t that:

$\mathbf{A}_9$: $A_Q \in \mathbf{a}Q$
$\mathbf{A}_{10}$: $\mathsf{compile}.A_Q.s.t$
$\mathbf{A}_{11}$: $\mathsf{evalb}.(J_Q.s) \wedge \mathsf{evalb}.(p.s) \wedge \neg(\mathsf{evalb}.(q.s))$

Now we have to prove that $(\mathsf{evalb}.(J_Q.t) \wedge \mathsf{evalb}.(p.t)) \vee \mathsf{evalb}.(q.t)$. From $\mathbf{A}_3$, we know that ${}_Q\vdash \circlearrowleft J_Q$, and consequently, using assumptions $\mathbf{A}_9$, $\mathbf{A}_{10}$, $\mathbf{A}_{11}$ and the definition of $\circlearrowleft$ $(4.4.7_{44}$ and $4.4.1_{43})$ we can conclude that $\mathsf{evalb}.(J_Q.t)$. Thus, we are left with the following proof obligation:

$\mathsf{evalb}.(p.t) \vee \mathsf{evalb}.(q.t)$

Case $\neg(\mathsf{guard\_of}.A_Q.s)$

In this case $\mathbf{A}_{10}$ implies $s = t$, and thus assumption $\mathbf{A}_{10}$ establishes the validity of $\mathsf{evalb}.(p.t) \vee \mathsf{evalb}.(q.t)$.

$\square_{(\neg(\mathsf{guard\_of}.A_Q.s))}$

Case $\mathsf{guard\_of}.A_Q.s$

$\mathbf{A}_{12}$: $\mathsf{guard\_of}.A_Q.s$

From $\mathbf{A}_1$ it follows that:

$\mathbf{A}_{13}$: $\mathbf{a}Q = \mathbf{a}Q_1 \cup \mathbf{a}Q_2$
$\mathbf{A}_{14}$: bitotal.$\mathcal{R}$.$\mathbf{a}P$.$\mathbf{a}Q_1$
$\mathbf{A}_{15}$: $\forall A_Q : A_Q \in \mathbf{a}Q_2 : \mathsf{skip} \sqsubseteq_{\mathbf{w}P,J} A_Q$

Case $A_Q \in \mathbf{a}Q_1$

$\mathbf{A}_{16}$: $A_Q \in \mathbf{a}Q_1$

From $\mathbf{A}_{14}$ we can conclude that there exists an action A_P, such that:

$\mathbf{A}_{17}$: $A_P \in \mathbf{a}P$
$\mathbf{A}_{18}$: $A_P \ \mathcal{R} \ A_P$
$\mathbf{A}_{19}$: $A_P \sqsubseteq_{\mathbf{w}P,J} A_Q$

From $\mathbf{A}_{17}$ and the always-enabledness of actions in the universe `ACTION` ($3.4.19_{34}$) we know that there exists a state t' such that

$\mathbf{A}_{20}$: compile.A_P.s.t'

and consequently from $\mathbf{A}_{19}$, $\mathbf{A}_{20}$, the definition of action refinement ($7.2.1_{109}$), and $\mathbf{A}_{10}$, $\mathbf{A}_{11}$, $\mathbf{A}_{12}$ and $\mathbf{A}_3$ we can infer that:

$\mathbf{A}_{21}$: $t \restriction \mathbf{w}P = t' \restriction \mathbf{w}P$

Moreover, using $\mathbf{A}_2$, $\mathbf{A}_{10}$, $\mathbf{A}_{20}$, and the definition of a well-formed UNITY program ($4.3.1_{43}$), and the definition of ignored variables ($3.4.22_{36}$) we can conclude:

$\mathbf{A}_{22}$: $s \restriction \mathbf{w}P^{\mathsf{c}} = t' \restriction \mathbf{w}P^{\mathsf{c}}$
$\mathbf{A}_{23}$: $s \restriction \mathbf{w}Q^{\mathsf{c}} = t \restriction \mathbf{w}Q^{\mathsf{c}}$

From $\mathbf{A}_{23}$ we can derive the following:

$s \restriction \mathbf{w}Q^{\mathsf{c}} = t \restriction \mathbf{w}Q^{\mathsf{c}}$
$=$ ($\mathbf{A}_4$)
$s \restriction (\mathbf{w}P \cup W)^{\mathsf{c}} = t \restriction (\mathbf{w}P \cup W)^{\mathsf{c}}$
$=$ (complement of set union, and projection composition ($2.8.5_{18}$))
$s \restriction \mathbf{w}P^{\mathsf{c}} \restriction W^{\mathsf{c}} = t \restriction \mathbf{w}P^{\mathsf{c}} \restriction W^{\mathsf{c}}$
$=$ ($\mathbf{A}_{22}$)
$t' \restriction \mathbf{w}P^{\mathsf{c}} \restriction W^{\mathsf{c}} = t \restriction \mathbf{w}P^{\mathsf{c}} \restriction W^{\mathsf{c}}$
$\Rightarrow$ (definition of projection ($2.8.1_{18}$))
$\forall x : x \in W^{\mathsf{c}} : (t' \restriction \mathbf{w}P^{\mathsf{c}}).x = (t \restriction \mathbf{w}P^{\mathsf{c}}).x$

This together with $\mathbf{A}_{20}$ establishes $t \restriction W^{\mathsf{c}} = t' \restriction W^{\mathsf{c}}$, which with $\mathbf{A}_7$ gives:

$\mathbf{A}_{24}$: $p.t = p.t' \ \wedge \ q.t = q.t'$

From assumptions $\mathbf{A}_8$, $\mathbf{A}_{11}$, $\mathbf{A}_{17}$, $\mathbf{A}_{20}$ we can conclude that $\mathsf{evalb}.(p.t') \wedge \mathsf{evalb}.(q.t')$, and thus $\mathbf{A}_{24}$ establishes this case.

$\square_{A_Q \in \mathbf{a}Q_1}$

Case $A_Q \in \mathbf{a}Q_2$

From $\mathbf{A}_9$, $\mathbf{A}_{15}$, the definition of action refinement ($7.2.1_{109}$), executable skip ($3.4.8_{31}$), $\mathbf{A}_3$, $\mathbf{A}_{10}$, $\mathbf{A}_{11}$ and $\mathbf{A}_{12}$ and we can conclude:

$\mathbf{A}_{25}$: $(s \restriction \mathbf{w}P = t \restriction \mathbf{w}P)$

Again, using $\mathbf{A}_2$ and the definition of a well-formed UNITY program ($4.3.1_{43}$), and the definition of ignored variables ($3.4.22_{36}$) we can conclude:

$\mathbf{A}_{26}$: $s \restriction \mathbf{w}Q^{\mathsf{c}} = t \restriction \mathbf{w}Q^{\mathsf{c}}$

Rewriting this in the same way as the previous case, gives the desired result.

$\square_{A_Q \in \mathbf{a}Q_2}$

$\square_{\mathsf{guard\_of}.A_Q.s}$

end of proof 7.2.11

C.2 Preservation of $\rightarrowtail$

Theorem 7.2.7 *P_ref_SUPERPOSE_AND_DECR_FUNC_PRSRVS_REACHe_GEN*
P_ref_SUPERPOSE_AND_DECR_FUNC_PRSRVS_CONe_GEN

Let $\prec$ be a well-founded relation over some set A, and $M \in \mathtt{State} \rightarrow A$.

$$\frac{\begin{array}{c} P \sqsubseteq_{R,J} Q \wedge \mathsf{Unity}.Q \wedge (\ {}_Q\vdash \circlearrowleft J_P \wedge J_Q) \wedge (J_P \wedge J_Q \Rightarrow J) \\ \exists W :: (\mathbf{w}Q = \mathbf{w}P \cup W) \wedge (J_P\ \mathcal{C}\ W^{\mathsf{c}}) \wedge (\mathbf{w}P \subseteq W^{\mathsf{c}}) \\ \forall A_Q : A_Q \in \mathbf{a}Q \wedge (\exists A_P :: (A_P \in \mathbf{a}P) \wedge (A_P\ \mathcal{R}\ A_Q)) : (\mathsf{guard\_of}.A_Q\ \mathcal{C}\ \mathbf{w}Q) \\ \forall A_P : A_P \in \mathbf{a}P : (J_P \wedge J_Q)\ {}_Q\vdash \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P\ \mathcal{R}\ A_Q) \wedge \mathsf{guard\_of}.A_Q) \\ \exists M :: (M\ \mathcal{C}\ \mathbf{w}Q) \wedge (\forall k : k \in A :\ {}_Q\vdash (J_P \wedge J_Q \wedge M = k)\ \mathsf{unless}\ (M \prec k)) \\ \wedge\ \forall k\ A_P A_Q : k \in A \wedge A_P \in \mathbf{a}P \wedge A_P\ \mathcal{R}\ A_Q : \\ \quad {}_Q\vdash (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q \wedge M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \vee M \prec k) \end{array}}{((J_P\ {}_P\vdash p \rightarrowtail q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \rightarrowtail q)) \ \wedge\ ((J_P\ {}_P\vdash p \rightsquigarrow q) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \rightsquigarrow q))}$$

◀

proof of 7.2.7 ($\rightarrowtail$-part)

Assume the following for a well-founded relation $\prec$:

$\mathbf{A}_1$: $P \sqsubseteq_{\mathcal{R},J} Q$
$\mathbf{A}_2$: $\mathsf{Unity}.Q$
$\mathbf{A}_3$: ${}_Q\vdash \circlearrowleft (J_P \wedge J_Q) \wedge (J_P \wedge J_Q \Rightarrow J)$
$\mathbf{A}_4$: $\mathbf{w}Q = \mathbf{w}P \cup W \ \wedge \ J_P \,\mathcal{C}\, W^{\mathsf{c}} \wedge \mathbf{w}P \subseteq W^{\mathsf{c}}$
$\mathbf{A}_5$: $\forall A_Q : A_Q \in \mathbf{a}Q \wedge (\exists A_P :: (A_P \in \mathbf{a}P) \wedge (A_P \,\mathcal{R}\, A_Q)) : (\mathsf{guard\_of}.A_Q \,\mathcal{C}\, \mathbf{w}Q)$
$\mathbf{A}_6$: $\forall A_P : A_P \in \mathbf{a}P : (J_P \wedge J_Q)\ {}_Q\vdash\ \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: (A_P \,\mathcal{R}\, A_Q) \wedge \mathsf{guard\_of}.A_Q)$
$\mathbf{A}_7$: $M \,\mathcal{C}\, \mathbf{w}Q$
$\mathbf{A}_8$: $\forall k ::\ {}_Q\vdash (J_P \wedge J_Q \wedge M = k)\ \mathsf{unless}\ (M \prec k)$
$\mathbf{A}_9$: $\forall k\ A_P\ A_Q : A_P \in \mathbf{a}P \wedge A_P \,\mathcal{R}\, A_Q :$
$\quad {}_Q\vdash (J_P \wedge J_Q \wedge \mathsf{guard\_of}.A_Q \wedge M = k)\ \mathsf{unless}\ (\neg(\mathsf{guard\_of}.A_P) \vee M \prec k)$

We have to prove that:

$$J_P\ {}_P\vdash p \rightarrowtail q \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \rightarrowtail q)$$

For this we use the following theorem directly taken from [Pra95]; it states an induction principle for the $\rightarrowtail$ operator that corresponds to the latter's definition ($4.5.4_{46}$):

Theorem C.2.1 INDUCTION *REACHe_INDUCT1*
For transitive and disjunctive R:

$$P, J : \ \frac{(\forall p, q :: (p \,\mathcal{C}\, (\mathbf{w}P) \ \wedge \ q \,\mathcal{C}\, (\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \wedge (J \wedge p\ \mathsf{ensures}\ q)) \Rightarrow R.p.q)}{(p \rightarrowtail q) \Rightarrow R.p.q}$$

---

take $R = (\lambda p\ q.\ J_P \wedge J_Q\ {}_Q\vdash p \rightarrowtail q)$. Since we already have $\rightarrowtail$-TRANSITIVITY, and $\rightarrowtail$-DISJUNCTION, we are left with the following proof-obligation:

$$\forall p\ q :: (p \,\mathcal{C}\, \mathbf{w}P \wedge q \,\mathcal{C}\, \mathbf{w}P \wedge \circlearrowleft J_P \wedge (J_P \wedge p\ \mathsf{ensures}\ q)) \Rightarrow (J_P \wedge J_Q\ {}_Q\vdash p \rightarrowtail q)$$

Choose arbitrary p and q, and assume:

$\mathbf{A}_{10}$: $p \,\mathcal{C}\, \mathbf{w}P \wedge q \,\mathcal{C}\, \mathbf{w}P$
$\mathbf{A}_{11}$: ${}_P\vdash \circlearrowleft J_P$
$\mathbf{A}_{12}$: ${}_P\vdash\ J_P \wedge p\ \mathsf{ensures}\ q$

Theorem $3.3.19_{28}$, stating confinement monotonicity, together with $\mathbf{A}_4$ and $\mathbf{A}_{10}$ gives:

$\mathbf{A}_{13}$: $p \,\mathcal{C}\, \mathbf{w}Q \wedge q \,\mathcal{C}\, \mathbf{w}Q$

Rewriting $\mathbf{A}_{12}$ with the definition of ensures ($4.4.2_{43}$), we know that there exists an action A_P, such that:

$\mathbf{A}_{14}$: ${}_P\vdash\ J_P \wedge p\ \mathsf{unless}\ q$
$\mathbf{A}_{15}$: $A_P \in \mathbf{a}P$
$\mathbf{A}_{16}$: $\forall s\ t :: (\mathsf{evalb}.(J_P.s) \wedge \mathsf{evalb}.(p.s) \wedge \neg\mathsf{evalb}.(q.s) \wedge \mathsf{compile}.A_P.s.t) \Rightarrow \mathsf{evalb}.(q.t)$

$\mathbf{A}_{17}$: Unity.P

Now we have to prove that:

$$J_P \wedge J_{Q\ Q} \vdash p \rightarrowtail q$$

$\Leftarrow$ ($\rightarrowtail$ Bounded Progress ($4.5.17_{48}$), $\mathbf{A}_7$, $\prec$ is well-founded)

$$\forall k ::\ J_P \wedge J_{Q\ Q} \vdash p \wedge M{=}k \rightarrowtail (p \wedge M \prec k) \vee q$$

$\Leftarrow$ ($\rightarrowtail$ Case distinction ($4.5.10_{47}$))

$$\left.\begin{array}{l} \forall k ::\ J_P \wedge J_{Q\ Q} \vdash \quad p \wedge M{=}k \wedge \neg(\mathsf{guard\_of}.A_P) \\ \qquad\qquad\qquad\quad \rightarrowtail \\ \qquad\qquad\qquad\quad (p \wedge M \prec k) \vee q \end{array}\right\} \textbf{false-guard-}A_P\textbf{-part}$$

$$\wedge$$

$$\left.\begin{array}{l} \forall k ::\ J_P \wedge J_{Q\ Q} \vdash \quad p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \\ \qquad\qquad\qquad\quad \rightarrowtail \\ \qquad\qquad\qquad\quad (p \wedge M \prec k) \vee q \end{array}\right\} \textbf{true-guard-}A_P\textbf{-part}$$

Before we prove these two conjuncts we first prove the following lemma.

lemma 1: $\forall s :: \mathsf{evalb}.(J_P.s) \wedge \mathsf{evalb}.(p.s) \wedge \neg\mathsf{evalb}.(\mathsf{guard\_of}.A_P.s) \Rightarrow \mathsf{evalb}.(q.s)$

Choose an arbitrary state s, and assume that:

$$\mathsf{evalb}.(J_P.s) \wedge \mathsf{evalb}.(p.s) \wedge \neg\mathsf{evalb}.(\mathsf{guard\_of}.A_P.s)$$

Since the guard of A_P is false, $\mathsf{compile}.A_P.s.t = (s = t)$, instantiating $\mathbf{A}_{16}$ with state s and rewriting with these assumptions gives us:

$$(\forall t : (\neg\mathsf{evalb}.(q.s) \wedge s = t) \Rightarrow \mathsf{evalb}.(q.t))$$

which equals $\mathsf{evalb}.(q.s)$.

□**lemma1**

false-guard-A_P-part

Theorem $\rightarrowtail$ Introduction ($4.5.7_{47}$, implication-part), assumptions $\mathbf{A}_3$, $\mathbf{A}_7$ and $\mathbf{A}_{13}$, and **lemma 1** establish this case.

□**false-guard-A_P-part**

true-guard-A_P-part

For arbitrary k we have to prove that:

$$J_P \wedge J_{Q\ Q} \vdash\ p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \rightarrowtail (p \wedge M \prec k) \vee q$$

= (logics)

$J_P \wedge J_Q \ {}_Q\!\vdash\ p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \rightarrowtail ((p \wedge M{\prec}k) \vee q) \ \vee \ ((p \wedge M{\prec}k) \vee q)$

$\Leftarrow$ ($\rightarrowtail$ CANCELLATION ($4.5.11_{47}$), $\mathbf{A}_7$, $\mathbf{A}_{13}$)

$$\left.\begin{array}{l} J_P \wedge J_Q \ {}_Q\!\vdash\ p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \\ \qquad\qquad \rightarrowtail \\ \qquad\qquad (p \wedge M{\prec}k) \vee q \vee (p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q)) \end{array}\right\} \mathbf{C}_1$$

$\wedge$

$$J_P \wedge J_Q \ {}_Q\!\vdash\ p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q) \rightarrowtail (p \wedge M{\prec}k) \vee q \ \} \mathbf{C}_2$$

Before we continue with the proofs of these conjunct, we shall first prove another lemma.

lemma 2: ${}_Q\!\vdash (J_P \wedge J_Q) \wedge p \ \mathsf{unless}\ q$

= (logics)
${}_Q\!\vdash J_Q \wedge (J_P \wedge p) \ \mathsf{unless}\ q$
$\Leftarrow$ (preservation of unless ($7.2.11_{112}$), $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, $\mathbf{A}_4$, $\mathbf{A}_{14}$, $\mathbf{A}_{17}$)
$(J_P \wedge p)\ \mathcal{C}\ W^{\mathsf{c}} \ \wedge \ q\ \mathcal{C}\ W^{\mathsf{c}}$
$\Leftarrow$ (confinement of binary operators ($3.3.13_{29}$) on first conjunct, and $\mathbf{A}_4$)
$p\ \mathcal{C}\ W^{\mathsf{c}} \ \wedge \ q\ \mathcal{C}\ W^{\mathsf{c}}$
$\Leftarrow$ (confinement monotonicity ($3.3.19_{28}$) on both conjuncts)
$p\ \mathcal{C}\ \mathbf{w}P \ \wedge \ q\ \mathcal{C}\ \mathbf{w}P \ \wedge \ \mathbf{w}P \subseteq W^{\mathsf{c}}$
Assumption $\mathbf{A}_{10}$ and $\mathbf{A}_4$ establishes this case.

$\square_{\mathbf{lemma2}}$

proof of $\mathbf{C}_1$

$$\left.\begin{array}{l} J_P \wedge J_Q \ {}_Q\!\vdash\ p \wedge M{=}k \wedge \mathsf{guard\_of}.A_P \\ \qquad\qquad \rightarrowtail \\ \qquad\qquad (p \wedge M{\prec}k) \vee q \vee (p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q)) \end{array}\right\} \mathbf{C}_1$$

= (logics)

$$\begin{array}{l} J_P \wedge J_Q \ {}_Q\!\vdash\ M{=}k \wedge \mathsf{guard\_of}.A_P \wedge p \\ \qquad\qquad \rightarrowtail \\ \qquad\qquad ((M{\prec}k \vee (M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q))) \wedge p) \vee q \end{array}$$

$\Leftarrow$ ($\rightarrowtail$ PSP ($4.5.12_{47}$), $\mathbf{A}_{13}$, **lemma 2**)

$$\begin{array}{l} J_P \wedge J_Q \ {}_Q\!\vdash\ M{=}k \wedge \mathsf{guard\_of}.A_P \\ \qquad\qquad \rightarrowtail \\ \qquad\qquad M{\prec}k \vee (M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q)) \end{array}$$

= (logics)

$$\begin{array}{l} J_P \wedge J_Q \ {}_Q\!\vdash\ \mathsf{guard\_of}.A_P \wedge M{=}k \\ \qquad\qquad \rightarrowtail \\ \qquad\qquad ((\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q) \wedge M{=}k) \vee M{\prec}k \end{array}$$

$\Leftarrow$ ($\rightarrowtail$ PSP ($4.5.12_{47}$), $\mathbf{A}_7$)

$J_P \wedge J_Q \ {}_Q\!\vdash \mathsf{guard\_of}.A_P \rightarrowtail (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q)$
$\wedge$
${}_Q\!\vdash M = k \wedge J_P \wedge J_Q \ \mathsf{unless}\ M \prec k$

Assumptions $\mathbf{A}_6$, $\mathbf{A}_{15}$ and $\mathbf{A}_8$ establish this.

$\square_{\mathbf{C}_1}$

proof of $\mathbf{C}_2$

$J_P \wedge J_Q \ {}_Q\!\vdash p \wedge M{=}k \wedge (\exists A_Q :: A_P \mathcal{R} A_Q \wedge \mathsf{guard\_of}.A_Q) \rightarrowtail (p \wedge M{\prec}k) \vee q \ \}\ \mathbf{C}_2$
$\Leftarrow$ ($\rightarrowtail$ Substitution ($4.5.6_{47}$), $\mathbf{A}_5$, $\mathbf{A}_7$, and $\mathbf{A}_{13}$)
$J_P \wedge J_Q \ {}_Q\!\vdash \ (\exists A_Q :: A_P \mathcal{R} A_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q)$
$\rightarrowtail$
$(\exists A_Q :: A_P \mathcal{R} A_Q \wedge ((p \wedge M{\prec}k) \vee q))$
$\Leftarrow$ ($\rightarrowtail$ Disjunction ($4.5.13_{47}$), $\mathbf{A}_5$, $\mathbf{A}_7$, and $\mathbf{A}_{13}$)
$\forall A_Q : A_P \mathcal{R} A_Q : J_P \wedge J_Q \ {}_Q\!\vdash p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \rightarrowtail (p \wedge M{\prec}k) \vee q$
$\Leftarrow$ ($\rightarrowtail$ Introduction ($4.5.7_{47}$), $\mathbf{A}_3$, $\mathbf{A}_5$,$\mathbf{A}_7$ and $\mathbf{A}_{13}$)
$\forall A_Q : A_P \mathcal{R} A_Q : \ {}_Q\!\vdash \ J_P \wedge J_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q$
$\mathsf{ensures}$
$(p \wedge M{\prec}k) \vee q$

Assume:

$\mathbf{A}_{18}$: $A_P \mathcal{R} A_Q$

We are left with the proof obligations (definition of $\mathsf{ensures}$ ($4.4.2_{43}$))

$$\left.\begin{array}{l} {}_Q\!\vdash \ J_P \wedge J_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \\ \ \mathsf{unless} \\ (p \wedge M{\prec}k) \vee q \end{array}\right\} \textbf{unless-part}$$

$\wedge$

$$\left.\begin{array}{l} \exists A_Q : A_Q \in \mathbf{a}Q : \\ \quad \{J_P \wedge J_Q \wedge p \wedge M{=}k \wedge \mathsf{guard\_of}.A_Q \wedge \neg((p \wedge M{\prec}k) \vee q)\} \\ \quad a \\ \quad \{(p \wedge M{\prec}k) \vee q\} \end{array}\right\} \textbf{exists-part}$$

proof of the unless-part

Assume for arbitrary actions a, and states s and t:

$\mathbf{A}_{19}$: $a \in \mathbf{a}Q$
$\mathbf{A}_{20}$: $\mathsf{compile}.a.s.t$
$\mathbf{A}_{21}$: $\mathsf{evalb}.(J_P.s) \wedge \mathsf{evalb}.(J_Q.s) \wedge \mathsf{evalb}.(p.s) \wedge (M.s = k) \wedge \mathsf{evalb}.(\mathsf{guard\_of}.A_Q.s)$
$\mathbf{A}_{22}$: $\neg(M.s \prec k) \wedge \neg\mathsf{evalb}.(q.s))$

We have to prove that:

$(\mathsf{evalb}.(J_P.t) \wedge \mathsf{evalb}.(J_Q.t) \wedge \mathsf{evalb}.(p.t) \wedge (M.t = k) \wedge \mathsf{evalb}.(\mathsf{guard\_of}.A_Q.t))$

$\vee$
$(\mathsf{evalb}.(p.t) \wedge (M.t \prec k))$
$\vee$
$\mathsf{evalb}.(q.t)$

From **lemma 2** and assumptions $\mathbf{A}_{19}$, $\mathbf{A}_{20}$, $\mathbf{A}_{21}$ and $\mathbf{A}_{22}$ we know that:

$\mathbf{A}_{23}$: $(\mathsf{evalb}.(J_P.t) \wedge \mathsf{evalb}.(J_Q.t) \wedge \mathsf{evalb}.(p.t)) \vee \mathsf{evalb}.(q.t)$

If $\mathsf{evalb}.(q.t)$ holds, then the proof has been established. So assume:

$\mathbf{A}_{24}$: $\neg\mathsf{evalb}.(q.t)$

Then assumptions $\mathbf{A}_{23}$ and $\mathbf{A}_{24}$ leave us with the proof obligation:

$((M.t = k) \wedge \mathsf{evalb}.(\mathsf{guard\_of}.A_Q.t)) \vee (M.t \prec k)$

From $\mathbf{A}_8$, $\mathbf{A}_9$, $\mathbf{A}_{19}$, $\mathbf{A}_{20}$, $\mathbf{A}_{21}$, $\mathbf{A}_{22}$ and the definition of unless $(4.4.1_{43})$ we can deduce:

$\mathbf{A}_{25}$: $M.t = k \ \vee \ M.t \prec k$
$\mathbf{A}_{26}$: $\mathsf{evalb}.(\mathsf{guard\_of}.A_P.s)$
$\Rightarrow$
$(\mathsf{evalb}.(\mathsf{guard\_of}.A_Q.t) \wedge (M.t = k)) \vee \neg(\mathsf{evalb}.(\mathsf{guard\_of}.A_P.t)) \vee (M.t \prec k)$

Using $\mathbf{A}_{25}$, if $M.t \prec k$ then the proof has been established. Suppose $M.t = k$. From $\mathbf{A}_1$, $\mathbf{A}_3$, $\mathbf{A}_{18}$, $\mathbf{A}_{21}$, and the definition of action refinement $(7.2.1_{109})$, we can conclude $\mathsf{evalb}.(\mathsf{guard\_of}.A_P.s)$, and hence assumption $\mathbf{A}_{26}$ gives:

$\mathbf{A}_{27}$: $\mathsf{evalb}(\mathsf{guard\_of}.A_Q.t) \vee \neg(\mathsf{evalb}.(\mathsf{guard\_of}.A_P.t))$

Suppose $\mathsf{evalb}.(\mathsf{guard\_of}.A_P.t)$ holds, then $\mathbf{A}_{27}$ establishes the proof. To reach a contradiction, we assume that:

$\mathbf{A}_{28}$: $\neg(\mathsf{evalb}.(\mathsf{guard\_of}.A_P.t))$

Now **lemma 1**, $\mathbf{A}_{28}$, $\mathbf{A}_{23}$, $\mathbf{A}_{24}$ imply $\mathsf{evalb}.(q.t)$ which obviously contradicts $\mathbf{A}_{24}$. $\square_{\mathbf{unless-part}}$

proof of the exists-part:
The action that does the trick is A_Q (introduced in $\mathbf{A}_{18}$). From $\mathbf{A}_1$ we know that $\mathcal{R}$ is bitotal, and hence using $\mathbf{A}_{15}$, $\mathbf{A}_{18}$, and the definition of a bitotal relation $(\mathrm{A}.3.3_{217})$ we can infer that A_Q is indeed an action in $\mathbf{a}Q$. Assume for arbitrary states s and t that:

$\mathbf{A}_{29}$: $\mathsf{evalb}.(J_P.s) \wedge \mathsf{evalb}.(J_Q.s) \wedge \mathsf{evalb}.(p.s) \wedge (M.s = k) \wedge \wedge \mathsf{evalb}.(\mathsf{guard\_of}.A_Q.s)$
$\mathbf{A}_{30}$: $\neg(M.s \prec k) \wedge \neg\mathsf{evalb}.(q.s)$

$\mathbf{A}_{31}$: $\mathsf{compile}.A_Q.s.t$

We are left with the proof obligation:

$(p.t \wedge (M.t \prec k)) \vee \mathsf{evalb}.(q.t)$

From $\mathbf{A}_{15}$ and the always-enabledness of actions in the universe `ACTION` ($3.4.19_{34}$) we know that there exists a state t' such that

$\mathbf{A}_{32}$: $\mathsf{compile}.A_P.s.t'$

and consequently from $\mathbf{A}_1$, $\mathbf{A}_{18}$, the definition of action refinement ($7.2.1_{109}$), and assumptions $\mathbf{A}_3$, $\mathbf{A}_{29}$, $\mathbf{A}_{31}$, $\mathbf{A}_{32}$ we can infer that:

$\mathbf{A}_{33}$: $t \restriction \mathbf{w}P = t' \restriction \mathbf{w}P$

From assumption $\mathbf{A}_{16}$, $\mathbf{A}_{29}$, and $\mathbf{A}_{32}$ we can conclude that:

$\mathbf{A}_{34}$: $\mathsf{evalb}.(q.t')$

Finally from $\mathbf{A}_{33}$, $\mathbf{A}_{34}$, and $\mathbf{A}_{10}$ we can conclude $\mathsf{evalb}.(q.t)$.
□**exists−part**

□ $\mathbf{C}_2$

□**true-guard-**A_P**-part**

end of proof of Theorem 7.2.7 ($\rightarrowtail$-part)

Appendix D

The formalisation of distributed hylomorphisms

This appendix contains the definitions and theorems formalising distributed hylomorphisms as they appear in the HOL theories depicted in Figure 8.14_{151}. For readabilty, however, we again sometimes omit nr_rec, nr_sent, and M as parameters in some definitions or theorems. It must be noted that in HOL this is not possible, when one of these functions is used in the right hand side of a definition, it has to be a parameter of the constant that is being defined.

D.1 PLUM

First, PLUM's initial condition, write variables and read variables are defined.

Definition D.1.1 *Initial_Condition_PLUM*

Initial_Condition_PLUM.$\mathbb{P}$.neighs.$starter$.father.idle
$= \neg(\text{idle}.starter) \wedge \forall p : p \in \mathbb{P} : \ ((p \neq starter) \Rightarrow \text{idle}.p)$
$\wedge\ (\text{father}.starter = starter) \wedge$ ASYNC_Init.$\mathbb{P}$.neighs

Definition D.1.2 *Write_Vars_PLUM*

Write_Vars_PLUM.$\mathbb{P}$.neighs.V.father.idle
$=$ ASYNC_Vars.$\mathbb{P}$.neighs $\cup \{\text{idle}.p \mid p \in \mathbb{P}\} \cup \{\text{father}.p \mid p \in \mathbb{P}\} \cup \{\text{V}.p \mid p \in \mathbb{P}\}$

Definition D.1.3 *Read_Vars_PLUM*

Read_Vars_PLUM.$\mathbb{P}$.neighs.V.father.idle
$=$ Write_Vars_PLUM.$\mathbb{P}$.neighs.V.father.idle

The following definition specifies that all variables declared as write and read variables in PLUM are distinct. (Since the communication variables play an important role in this definition, we have not omitted them as parameters.)

Definition D.1.4 *distinct_PLUM_Vars*

distinct_PLUM_Vars.$\mathbb{P}$.neighs.nr_rec.nr_sent.M.V.father.idle
$= \forall p, q, r, s ::$
$((\mathsf{idle}.p = \mathsf{idle}.q) = (p = q)) \wedge ((\mathsf{nr\_sent}.p.q = \mathsf{nr\_sent}.r.s) = ((p = r) \wedge (q = s)))$
$((\mathsf{V}.p = \mathsf{V}.q) = (p = q)) \wedge ((\mathsf{nr\_rec}.p.q = \mathsf{nr\_rec}.r.s) = ((p = r) \wedge (q = s)))$
$((\mathsf{father}.p = \mathsf{father}.q) = (p = q)) \wedge ((\mathsf{M}.p.q = \mathsf{M}.r.s) = ((p = r) \wedge (q = s)))$
$(\mathsf{idle}.p \neq \mathsf{V}.q) \wedge (\mathsf{idle}.p \neq \mathsf{father}.q) \wedge (\mathsf{idle}.p \neq \mathsf{nr\_sent}.q.r) \wedge (\mathsf{idle}.p \neq \mathsf{nr\_rec}.q.r)$
$(\mathsf{idle}.p \neq \mathsf{M}.q.r) \wedge (\mathsf{V}.p \neq \mathsf{father}.q) \wedge (\mathsf{V}.p \neq \mathsf{nr\_sent}.q.r) \wedge (\mathsf{V}.p \neq \mathsf{nr\_rec}.q.r)$
$(\mathsf{V}.p \neq \mathsf{M}.q.r) \wedge (\mathsf{father}.p \neq \mathsf{nr\_sent}.q.r) \wedge (\mathsf{father}.p \neq \mathsf{nr\_rec}.q.r)$
$(\mathsf{father}.p \neq \mathsf{M}.q.r) \wedge (\mathsf{nr\_rec}.p.q \neq \mathsf{nr\_sent}.r.s) \wedge (\mathsf{nr\_rec}.p.q \neq \mathsf{M}.r.s)$
$(\mathsf{nr\_sent}.p.q \neq \mathsf{M}.r.s)$

In the following definitions, PLUM's actions are defined.

Definition D.1.5 *IDLE*

IDLE.$p.q$.idle.h.V.father
$=$ strengthen_guard.(VAR.(idle.p) $\wedge$ mit.$q.p$)
.receive.$p.q.h.$(V.p) $\parallel$ father.$p :=$ CONST.$q \parallel$ idle.$p :=$ false

Definition D.1.6 *COL*

COL.$p.q$.neighs.idle.h.V
$=$ strengthen_guard.($\neg$VAR.(idle.p) $\wedge$ mit.$q.p \wedge$ $collecting_{PLUM}.p$)
.receive.$p.q.h.$(V.p)

Definition D.1.7 *PROP*

PROP.$p.q$.neighs.idle.PROP_mes.father
$=$ strengthen_guard.($\neg$VAR.(idle.p) $\wedge$ $cp.p.q \wedge$ $propagating_{PLUM}.p$)
.send.$p.q$.PROP_mes.

Definition D.1.8 *DONE*

DONE.$p.q$.neighs.DONE_mes.father
$=$ strengthen_guard.($finished\_collecting\_and\_propagating.p$
$\wedge \neg reported\_to\_father.p \wedge (q =$ father.$p))$
.send.$p.q$.DONE_mes.

Subsequently, PLUM is defined as an object of type `Uprog` as follows:

Definition D.1.9

PLUM.$\mathbb{P}$.neighs.$starter$.iniA.h.PROP_mes.DONE_mes.V.idle.father
=
(
{IDLE.$p.q$.idle.$(h.p)$.V.father | $p \in \mathbb{P} \wedge q \in$ neighs.p}
$\cup$
{COL.$p.q$.neighs.idle.$(h.p)$.V | $p \in \mathbb{P} \wedge q \in$ neighs.p}
$\cup$
{PROP.$p.q$.neighs.idle.(PROP_mes.p).father | $p \in \mathbb{P} \wedge q \in$ neighs.p}
$\cup$
{DONE.$p.q$.neighs.(DONE_mes.p).father | $p \in \mathbb{P} \wedge q \in$ neighs.p}
,
Initial_Condition_PLUM.$\mathbb{P}$.neighs.$starter$.father.idle $\wedge$ iniA
,
Read_Vars_PLUM.$\mathbb{P}$.neighs.V.father.idle
,
Write_Vars_PLUM.$\mathbb{P}$.neighs.V.father.idle
)

Finally, the following theorem states the conditions under which PLUM is a well-formed UNITY program (i.e. satisfies the predicate Unity).

Theorem D.1.10 *dUNITY_PLUM*

$$\frac{\begin{array}{c}\text{Network.}\mathbb{P}.starter.\text{neighs} \\ \forall p, e : p \in \mathbb{P} \wedge e \; \mathcal{C}\text{Write\_Vars\_PLUM} : (h.p.e) \; \mathcal{C} \; \text{Write\_Vars\_PLUM} \\ \forall p : p \in \mathbb{P} : \text{PROP\_mes.}p \; \mathcal{C} \; \text{Write\_Vars\_PLUM} \; \wedge \; \text{DONE\_mes.}p \; \mathcal{C} \; \text{Write\_Vars\_PLUM}\end{array}}{\text{Unity.(PLUM.}\mathbb{P}.\text{neighs.}starter.\text{iniA.}h.\text{PROP\_mes.DONE\_mes.V.idle.father)}}$$

D.2 ECHO

ECHO's initial condition, write and read variables are the same as those of PLUM.

Definition D.2.1 *Initial_Condition_ECHO*

Initial_Condition_ECHO.$\mathbb{P}$.neighs.$starter$.idle.father
= Initial_Condition_PLUM.$\mathbb{P}$.neighs.$starter$.idle.father

Definition D.2.2 *Write_Vars_ECHO*

Write_Vars_ECHO.$\mathbb{P}$.neighs.V.idle.father = Write_Vars_PLUM.$\mathbb{P}$.neighs.V.idle.father

Definition D.2.3 *Read_Vars_ECHO*

Read_Vars_ECHO.$\mathbb{P}$.neighs.V.idle.father = Read_Vars_PLUM.$\mathbb{P}$.neighs.V.idle.father

Definition D.2.4 *distinct_ECHO_Vars*

distinct_ECHO_Vars.$\mathbb{P}$.neighs.nr_rec.nr_sent.M.V.father.idle
= distinct_PLUM_Vars.$\mathbb{P}$.neighs.nr_rec.nr_sent.M.V.father.idle

ECHO's actions are defined in terms of PLUM's action.

Definition D.2.5 *IDLE_ECHO*

IDLE_ECHO.$p.q$.idle.h.V.father
= IDLE.$p.q$.idle.h.V.father

Definition D.2.6 *COL_ECHO*

COL_ECHO.$p.q$.neighs.idle.h.V
=strengthen_guard.$(\neg propagating_{ECHO}.p)$
.COL.$p.q$.neighs.idle.h.V

Definition D.2.7 *PROP_ECHO*

PROP_ECHO.$p.q$.neighs.idle.PROP_mes.father
= PROP.$p.q$.neighs.idle.PROP_mes.M.father

Definition D.2.8 *DONE_ECHO*

DONE_ECHO.$p.q$.neighs.DONE_mes.father
= DONE.$p.q$.neighs.DONE_mes.father

Definition D.2.9

ECHO.$\mathbb{P}$.neighs.$starter$.iniA.h.PROP_mes.DONE_mes.V.idle.father
=
(
{IDLE_ECHO.$p.q$.idle.$(h.p)$.V.father | $p \in \mathbb{P} \wedge q \in$ neighs.p}
$\cup$
{COL_ECHO.$p.q$.neighs.idle.$(h.p)$.V | $p \in \mathbb{P} \wedge q \in$ neighs.p}
$\cup$
{PROP_ECHO.$p.q$.neighs.idle.(PROP_mes.p).father | $p \in \mathbb{P} \wedge q \in$ neighs.p}
$\cup$
{DONE_ECHO.$p.q$.neighs.(DONE_mes.p).father | $p \in \mathbb{P} \wedge q \in$ neighs.p}
,
Initial_Condition_ECHO.$\mathbb{P}$.neighs.$starter$.idle.father $\wedge$ iniA
,
Read_Vars_ECHO.$\mathbb{P}$.neighs.V.idle.father
,
Write_Vars_ECHO.$\mathbb{P}$.neighs.V.idle.father
)

Theorem D.2.10 *dUNITY_ECHO*

$$\frac{\begin{array}{c}\text{Network.}\mathbb{P}.starter.\text{neighs} \\ \forall p, e : p \in \mathbb{P} \wedge e \;\mathcal{C}\; \text{Write\_Vars\_ECHO} : (h.p.e) \;\mathcal{C}\; \text{Write\_Vars\_ECHO} \\ \forall p : p \in \mathbb{P} : \text{PROP\_mes.}p \;\mathcal{C}\; \text{Write\_Vars\_ECHO} \;\wedge\; \text{DONE\_mes.}p \;\mathcal{C}\; \text{Write\_Vars\_ECHO}\end{array}}{\text{Unity.(ECHO.}\mathbb{P}\text{.neighs.}starter\text{.iniA.}h\text{.PROP\_mes.DONE\_mes.V.idle.father)}}$$

D.3 Tarry

TARRY has additional variables le_rec for each process in $\mathbb{P}$. For the starter the value of le_rec is initially true, and for the followers it is initially false.

Definition D.3.1 *Initial_Condition_Tarry*

$$\begin{aligned}&\mathsf{Initial\_Condition\_Tarry}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{idle}.\mathsf{father}.\mathsf{le\_rec}\\ &= \mathsf{Initial\_Condition\_PLUM}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{idle}.\mathsf{father}\\ &\quad \wedge (\mathsf{le\_rec}.\mathit{starter}) \wedge \forall p : p \in \mathbb{P} : \ ((p \neq \mathit{starter}) \Rightarrow \neg \mathsf{le\_rec}.p)\end{aligned}$$

Definition D.3.2 *Write_Vars_Tarry*

$$\begin{aligned}&\mathsf{Write\_Vars\_Tarry}.\mathbb{P}.\mathsf{neighs}.\mathsf{V}.\mathsf{idle}.\mathsf{father}.\mathsf{le\_rec}\\ &= \mathsf{Write\_Vars\_Tarry}.\mathbb{P}.\mathsf{neighs}.\mathsf{V}.\mathsf{idle}.\mathsf{father} \cup \{\mathsf{le\_rec}.p \mid p \in \mathbb{P}\}\end{aligned}$$

Definition D.3.3 *Read_Vars_Tarry*

$$\begin{aligned}&\mathsf{Read\_Vars\_Tarry}.\mathbb{P}.\mathsf{neighs}.\mathsf{V}.\mathsf{idle}.\mathsf{father}.\mathsf{le\_rec}\\ &= \mathsf{Write\_Vars\_Tarry}.\mathbb{P}.\mathsf{neighs}.\mathsf{V}.\mathsf{idle}.\mathsf{father}.\mathsf{le\_rec}\end{aligned}$$

Definition D.3.4 *distinct_Tarry_Vars*

$$\begin{aligned}&\mathsf{distinct\_Tarry\_Vars}.\mathbb{P}.\mathsf{neighs}.\mathsf{nr\_rec}.\mathsf{nr\_sent}.\mathsf{M}.\mathsf{V}.\mathsf{father}.\mathsf{idle}.\mathsf{le\_rec}\\ &= (\mathsf{distinct\_PLUM\_Vars}.\mathbb{P}.\mathsf{neighs}.\mathsf{nr\_rec}.\mathsf{nr\_sent}.\mathsf{M}.\mathsf{V}.\mathsf{father}.\mathsf{idle}) \wedge\\ &\quad \forall p, q, r, s ::\\ &\qquad ((\mathsf{le\_rec}.p = \mathsf{le\_rec}.q) = (p = q)) \wedge (\mathsf{le\_rec}.p \neq \mathsf{idle}.q) \wedge (\mathsf{le\_rec}.p \neq \mathsf{V}.q)\\ &\qquad (\mathsf{le\_rec}.p \neq \mathsf{father}.q) \wedge (\mathsf{le\_rec}.p \neq \mathsf{nr\_sent}.r.s) \wedge (\mathsf{le\_rec}.p \neq \mathsf{nr\_rec}.r.s)\\ &\qquad (\mathsf{le\_rec}.p \neq \mathsf{M}.r.s)\end{aligned}$$

TARRY's actions are defined in terms of PLUM's actions.

Definition D.3.5 *IDLE_Tarry*

$$\begin{aligned}&\mathsf{IDLE\_Tarry}.p.q.\mathsf{idle}.h.\mathsf{V}.\mathsf{father}.\mathsf{le\_rec}\\ &= \mathsf{augment}.(\mathsf{IDLE}.p.q.\mathsf{idle}.h.\mathsf{V}.\mathsf{father})\\ &\qquad .(\mathsf{le\_rec.p} := \mathsf{true})\end{aligned}$$

Definition D.3.6 *COL_Tarry*

$$\begin{aligned}&\mathsf{COL\_Tarry}.p.q.\mathsf{neighs}.\mathsf{idle}.h.\mathsf{V}.\mathsf{le\_rec}\\ &= \mathsf{strengthen\_guard}.(\neg \mathsf{VAR}.(\mathsf{le\_rec}.p))\\ &\qquad .\mathsf{augment}.(\mathsf{COL}.p.q.\mathsf{neighs}.\mathsf{idle}.h.\mathsf{V})\\ &\qquad\quad .(\mathsf{le\_rec.p} := \mathsf{true})\end{aligned}$$

Definition D.3.7 *PROP_Tarry*

$$\begin{aligned}&\mathsf{PROP\_Tarry}.p.q.\mathsf{neighs}.\mathsf{idle}.\mathsf{PROP\_mes}.\mathsf{father}.\mathsf{le\_rec}\\ &= \mathsf{strengthen\_guard}.(\mathsf{VAR}.(\mathsf{le\_rec}.p))\\ &\qquad .\mathsf{augment}.(\mathsf{PROP}.p.q.\mathsf{neighs}.\mathsf{idle}.\mathsf{PROP\_mes}.\mathsf{father})\\ &\qquad\quad .(\mathsf{le\_rec.p} := \mathsf{false})\end{aligned}$$

Definition D.3.8 *DONE_Tarry*

$$
\begin{array}{l}
\mathsf{DONE\_Tarry}.p.q.\mathsf{neighs}.\mathsf{DONE\_mes.f.le\_rec} \\
= \mathsf{augment}.\mathsf{DONE}.p.q.\mathsf{neighs}.\mathsf{DONE\_mes.father} \\
\qquad .(\mathsf{le\_rec.p} := \mathsf{false})
\end{array}
$$

Definition D.3.9

$$
\begin{array}{l}
\mathsf{Tarry}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec} \\
= \\
(\{\mathsf{IDLE\_Tarry}.p.q.\mathsf{idle}.(h.p).\mathsf{V.f.le\_rec} \mid p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p\} \\
\ \cup \\
\ \{\mathsf{COL\_Tarry}.p.q.\mathsf{neighs.idle}.(h.p).\mathsf{V.le\_rec} \mid p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p\} \\
\ \cup \\
\ \{\mathsf{PROP\_Tarry}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec} \mid p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p\} \\
\ \cup \\
\ \{\mathsf{DONE\_Tarry}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{father.le\_rec} \mid p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p\} \\
, \\
\ \mathsf{Initial\_Condition\_Tarry}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{idle.father.le\_rec} \wedge \mathsf{iniA} \\
, \\
\ \mathsf{Read\_Vars\_Tarry}.\mathbb{P}.\mathsf{neighs.V.idle.father.le\_rec} \\
, \\
\ \mathsf{Write\_Vars\_Tarry}.\mathbb{P}.\mathsf{neighs.V.idle.father.le\_rec} \\
)
\end{array}
$$

Theorem D.3.10

$$
\frac{\begin{array}{c}
\mathsf{Network}.\mathbb{P}.\mathit{starter}.\mathsf{neighs} \\
\forall p, e : p \in \mathbb{P} \wedge e \,\mathcal{C}\, \mathsf{Write\_Vars\_Tarry} : (h.e) \,\mathcal{C}\, \mathsf{Write\_Vars\_Tarry} \\
\forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p \,\mathcal{C}\, \mathsf{Write\_Vars\_Tarry} \ \wedge\ \mathsf{DONE\_mes}.p \,\mathcal{C}\, \mathsf{Write\_Vars\_Tarry}
\end{array}}{\mathsf{Unity}.(\mathsf{Tarry}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec}}
$$

From Chapter 9 we know (see page 192) that termination detection can only be proved using the refinement framework if, for all distributed hylomorphisms, we assume that:

$$
\begin{array}{l}
\forall p, e : p \in \mathbb{P} \wedge e \,\mathcal{C}\, \mathbf{w}\mathrm{PLUM} : (h.p.e) \,\mathcal{C}\, \mathbf{w}\mathrm{PLUM} \\
\forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p \,\mathcal{C}\, \mathbf{w}\mathrm{PLUM} \ \wedge\ \mathsf{DONE\_mes}.p \,\mathcal{C}\, \mathbf{w}\mathrm{PLUM}
\end{array}
$$

Consequently, the following theorem has a more suitable form for concluding well-formedness of TARRY during the verification activities in Chapter 9 (Section 9.4):

Theorem D.3.11 *dUNITY_Tarry*

$$
\frac{\begin{array}{c}
\mathsf{Network}.\mathbb{P}.\mathit{starter}.\mathsf{neighs} \\
\forall p, e : p \in \mathbb{P} \wedge e \,\mathcal{C}\, \boxed{\mathsf{Write\_Vars\_PLUM}} : (h.e) \,\mathcal{C}\, \mathsf{Write\_Vars\_Tarry} \\
\forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p \,\mathcal{C}\, \mathsf{Write\_Vars\_Tarry} \ \wedge\ \mathsf{DONE\_mes}.p \,\mathcal{C}\, \mathsf{Write\_Vars\_Tarry}
\end{array}}{\mathsf{Unity}.(\mathsf{Tarry}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec}}
$$

D.4 DFS

DFS has additional variables lp_rec for each process in $\mathbb{P}$ which do not need to be initialised.

Definition D.4.1 *Initial_Condition_DFS*

Initial_Condition_DFS.$\mathbb{P}$.neighs.$starter$.idle.father.le_rec
= Initial_Condition_Tarry.$\mathbb{P}$.neighs.$starter$.idle.father

Definition D.4.2 *Write_Vars_DFS*

Write_Vars_DFS.$\mathbb{P}$.neighs.V.idle.father.le_rec.lp_rec
= Write_Vars_Tarry.$\mathbb{P}$.neighs.V.idle.father $\cup$ $\{$lp_rec.$p \mid p \in \mathbb{P}\}$

Definition D.4.3 *Read_Vars_DFS*

Read_Vars_DFS.$\mathbb{P}$.neighs.V.idle.father.le_rec.lp_rec
= Write_Vars_DFS.$\mathbb{P}$.neighs.V.idle.father.le_rec.lp_rec

Definition D.4.4 *distinct_DFS_Vars*

distinct_DFS_Vars.$\mathbb{P}$.neighs.nr_rec.nr_sent.M.V.father.idle.le_rec.lp_rec
= distinct_Tarry_Vars.$\mathbb{P}$.neighs.nr_rec.nr_sent.M.V.father.idle.le_rec $\wedge$
$\forall p, q, r, s ::$
((lp_rec.p = lp_rec.q) = ($p = q$)) $\wedge$ (lp_rec.$p \neq$ idle.q) $\wedge$ (lp_rec.$p \neq$ V.q)
(lp_rec.$p \neq$ father.q) $\wedge$ (lp_rec.$p \neq$ nr_sent.$r.s$) $\wedge$ (lp_rec.$p \neq$ nr_rec.$r.s$)
(lp_rec.$p \neq$ M.$r.s$) $\wedge$ (lp_rec.$p \neq$ le_rec.q))

DFS's actions are defined in terms of DFS's actions.

Definition D.4.5 *IDLE_DFS*

IDLE_DFS.$p.q$.idle.h.V.father.lp_rec
= augment.(IDLE_Tarry.$p.q$.idle.h.V.father.le_rec)
.(lp_rec.$p := q$)

Definition D.4.6 *COL_DFS*

COL_DFS.$p.q$.neighs.idle.h.V.le_rec.lp_rec
= augment.(COL_Tarry.$p.q$.neighs.idle.h.V.le_rec)
.(lp_rec.$p := q$)

Definition D.4.7 *PROP_lp_rec_DFS*

PROP_lp_rec_DFS.$p.q$.neighs.idle.PROP_mes.father.le_rec.lp_rec
= strengthen_guard.((CONST.q) = (VAR.(lp_rec.p)))
.(PROP_Tarry.$p.q$.neighs.idle.PROP_mes.father.le_rec)

Definition D.4.8 *PROP_not_lp_rec_DFS*

$$\begin{aligned}&\mathsf{PROP\_not\_lp\_rec\_DFS}.p.q.\mathsf{neighs.idle.PROP\_mes.father.le\_rec.lp\_rec}\\ &= \mathsf{strengthen\_guard}.(\forall x : x \in \mathsf{neighs}.p \wedge (\mathsf{CONST}.x) = (\mathsf{VAR}.(\mathsf{lp\_rec}.p)) :\neq cp.p.x)\\ &\qquad .(\mathsf{PROP\_Tarry}.p.q.\mathsf{neighs.idle.PROP\_mes.father.le\_rec})\end{aligned}$$

Definition D.4.9 *DONE_DFS*

$$\begin{aligned}&\mathsf{DONE\_DFS}.p.q.\mathsf{neighs.DONE\_mes.father.le\_rec}\\ &= \mathsf{DONE\_Tarry}.p.q.\mathsf{neighs.DONE\_mes.father.le\_rec}\end{aligned}$$

Definition D.4.10

$$\begin{aligned}&\mathsf{DFS}.\mathbb{P}.\mathsf{neighs}.starter.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec.lp\_rec}\\ &=\\ &(\{\mathsf{IDLE\_DFS}.p.q.\mathsf{idle}.(h.p).\mathsf{V.father.le\_rec.lp\_rec} \mid p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p\}\\ &\ \cup\\ &\ \{\mathsf{COL\_DFS}.p.q.\mathsf{neighs.idle}.(h.p).\mathsf{V.le\_rec.lp\_rec} \mid p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p\}\\ &\ \cup\\ &\ \{\mathsf{PROP\_lp\_rec\_DFS}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec.lp\_rec} \mid p \in \mathbb{P} \wedge\\ &\qquad\qquad q \in \mathsf{neighs}.p\}\\ &\ \cup\\ &\ \{\mathsf{PROP\_not\_lp\_rec\_DFS}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec.lp\_rec} \mid p \in \mathbb{P} \wedge\\ &\qquad\qquad q \in \mathsf{neighs}.p\}\\ &\ \cup\\ &\ \{\mathsf{DONE\_DFS}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{M.father.le\_rec} \mid p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p\}\\ &,\\ &\ \mathsf{Initial\_Condition\_DFS}.\mathbb{P}.\mathsf{neighs}.starter.\mathsf{idle.father.le\_rec} \wedge \mathsf{iniA}\\ &,\\ &\ \mathsf{Read\_Vars\_DFS}.\mathbb{P}.\mathsf{neighs.V.idle.father.le\_rec}\\ &,\\ &\ \mathsf{Write\_Vars\_DFS}.\mathbb{P}.\mathsf{neighs.V.idle.father.le\_rec}\\ &)\end{aligned}$$

Theorem D.4.11

$$\frac{\begin{array}{c}\mathsf{Network}.\mathbb{P}.starter.\mathsf{neighs}\\ \forall p, e : p \in \mathbb{P} \wedge e\ \mathcal{C}\ \mathsf{Write\_Vars\_DFS} : (h.e)\ \mathcal{C}\ \mathsf{Write\_Vars\_DFS}\\ \forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p\ \mathcal{C}\ \mathsf{Write\_Vars\_DFS}\ \wedge\ \mathsf{DONE\_mes}.p\ \mathcal{C}\ \mathsf{Write\_Vars\_DFS}\end{array}}{\mathsf{Unity}.(\mathsf{DFS}.\mathbb{P}.\mathsf{neighs}.starter.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec.lp\_rec}}$$

And, again in more suitable form:

Theorem D.4.12 *dUNITY_DFS*

$$\frac{\begin{array}{c}\mathsf{Network}.\mathbb{P}.starter.\mathsf{neighs}\\ \forall p, e : p \in \mathbb{P} \wedge e\ \mathcal{C}\ \boxed{\mathsf{Write\_Vars\_PLUM}} : (h.e)\ \mathcal{C}\ \mathsf{Write\_Vars\_DFS}\\ \forall p : p \in \mathbb{P} : \mathsf{PROP\_mes}.p\ \mathcal{C}\ \mathsf{Write\_Vars\_DFS}\ \wedge\ \mathsf{DONE\_mes}.p\ \mathcal{C}\ \mathsf{Write\_Vars\_DFS}\end{array}}{\mathsf{Unity}.(\mathsf{DFS}.\mathbb{P}.\mathsf{neighs}.starter.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec.lp\_rec}}$$

D.5 Refinement orderings

D.5.1 PLUM and ECHO

The theorems characterising the bitotal relation between the actions of PLUM and ECHO are stated below, and can be found in the theory ECHO (see Figure 8.14$_{151}$).

Theorem D.5.1 *R_PLUM_ECHO_IDLE_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\mathrm{PLUM\_ECHO}.(\mathsf{IDLE}.p.q.\mathsf{idle}.(h.p).\mathsf{V}.\mathsf{father}) \\ \quad .(\mathsf{IDLE\_ECHO}.p.q.\mathsf{idle}.(h.p).\mathsf{V}.\mathsf{father})\end{array}}$$

Theorem D.5.2 *R_PLUM_ECHO_COL_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\mathrm{PLUM\_ECHO}.(\mathsf{COL}.p.q.\mathsf{neighs}.\mathsf{idle}.(h.p).\mathsf{V}) \\ \quad .(\mathsf{COL\_ECHO}.p.q.\mathsf{neighs}.\mathsf{idle}.(h.p).\mathsf{V})\end{array}}$$

Theorem D.5.3 *R_PLUM_ECHO_PROP_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\mathrm{PLUM\_ECHO}.(\mathsf{PROP}.p.q.\mathsf{neighs}.\mathsf{idle}.(\mathsf{PROP\_mes}.p).\mathsf{father}) \\ \quad .(\mathsf{PROP\_ECHO}.p.q.\mathsf{neighs}.\mathsf{idle}.(\mathsf{PROP\_mes}.p).\mathsf{father})\end{array}}$$

Theorem D.5.4 *R_PLUM_ECHO_DONE_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\mathrm{PLUM\_ECHO}.(\mathsf{DONE}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{father}) \\ \quad .(\mathsf{DONE\_ECHO}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{father})\end{array}}$$

The following theorem states conditions under which $\mathcal{R}\_\mathrm{PLUM\_ECHO}$ is bitotal.

Theorem D.5.5 *BITOTAL_R_PLUM_ECHO*

$$\frac{\begin{array}{c}\mathsf{Network}.\mathbb{P}.\mathsf{neighs}.\mathit{starter} \\ \mathsf{distinct\_ECHO\_Vars}.\mathbb{P}.\mathsf{neighs.nr\_rec.nr\_sent.M.V.father.idle}\end{array}}{\begin{array}{l}\mathsf{bitotal}.\mathcal{R}\_\mathrm{PLUM\_ECHO} \\ \quad .\mathbf{a}(\mathsf{PLUM}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father}) \\ \quad .\mathbf{a}(\mathsf{ECHO}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father})\end{array}}$$

Finally, we can prove that ECHO refines PLUM. (Note that in Theorem 8.12.1$_{149}$ the hypothesis were assumed implicitly):

Theorem D.5.6 *ECHO_refines_PLUM*

$$\frac{\begin{array}{c}\mathsf{Network}.\mathbb{P}.\mathsf{neighs}.\mathit{starter} \\ \mathsf{distinct\_ECHO\_Vars}.\mathbb{P}.\mathsf{neighs.nr\_rec.nr\_sent.M.V.father.idle}\end{array}}{\mathrm{PLUM} \sqsubseteq_{\mathcal{R}\_\mathrm{PLUM\_ECHO},\, J} \mathrm{ECHO}}$$

D.5.2 PLUM and TARRY

The theorems characterising the bitotal relation between the actions of PLUM and TARRY are stated below, and can be found in the theory Tarry (see Figure 8.14$_{151}$).

Theorem D.5.7 *R_PLUM_Tarry_IDLE_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\text{PLUM\_TARRY}.(\mathsf{IDLE}.p.q.\mathsf{idle}.(h.p).\mathsf{V.father}) \\ \quad .(\mathsf{IDLE\_Tarry}.p.q.\mathsf{idle}.(h.p).\mathsf{V.f.le\_rec})\end{array}}$$

Theorem D.5.8 *R_PLUM_Tarry_COL_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\text{PLUM\_TARRY}.(\mathsf{COL}.p.q.\mathsf{neighs.idle}.(h.p).\mathsf{V}) \\ \quad .(\mathsf{COL\_Tarry}.p.q.\mathsf{neighs.idle}.(h.p).\mathsf{V.le\_rec})\end{array}}$$

Theorem D.5.9 *R_PLUM_Tarry_PROP_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\text{PLUM\_TARRY}.(\mathsf{PROP}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father}) \\ \quad .(\mathsf{PROP\_Tarry}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec})\end{array}}$$

Theorem D.5.10 *R_PLUM_Tarry_DONE_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\begin{array}{l}\mathcal{R}\_\text{PLUM\_TARRY}.(\mathsf{DONE}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{father}) \\ \quad .(\mathsf{DONE\_Tarry}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{father.le\_rec})\end{array}}$$

The following theorem states conditions under which $\mathcal{R}\_\text{PLUM\_TARRY}$ is bitotal.

Theorem D.5.11 *BITOTAL_R_PLUM_Tarry*

$$\frac{\begin{array}{c}\mathsf{Network}.\mathbb{P}.\mathsf{neighs}.\mathit{starter} \\ \mathsf{distinct\_Tarry\_Vars}.\mathbb{P}.\mathsf{neighs.nr\_rec.nr\_sent.M.V.father.idle.le\_rec}\end{array}}{\begin{array}{l}\mathsf{bitotal}.\mathcal{R}\_\text{PLUM\_TARRY} \\ \quad .\mathbf{a}(\mathsf{PLUM}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father}) \\ \quad .\mathbf{a}(\mathsf{Tarry}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec})\end{array}}$$

Finally, we can prove that TARRY refines PLUM. (Note that in Theorem 8.12.2$_{149}$ the hypothesis were assumed implicitly):

Theorem D.5.12 *Tarry_refines_PLUM*

$$\frac{\begin{array}{c}\mathsf{Network}.\mathbb{P}.\mathsf{neighs}.\mathit{starter} \\ \mathsf{distinct\_Tarry\_Vars}.\mathbb{P}.\mathsf{neighs.nr\_rec.nr\_sent.M.V.father.idle.le\_rec}\end{array}}{\text{PLUM} \sqsubseteq_{\mathcal{R}\_\text{PLUM\_TARRY},\ J} \text{TARRY}}$$

D.5.3 TARRY and DFS

The theorems characterising the bitotal relation between the actions of TARRY and DFS are stated below, and can be found in the theory DFS (see Figure 8.14$_{151}$).

Theorem D.5.13 *R_Tarry_DFS_IDLE_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\mathcal{R}_{\text{-TARRY\_DFS}}.(\mathsf{IDLE\_Tarry}.p.q.\mathsf{idle}.(h.p).\mathsf{V.f.le\_rec}) \\ .(\mathsf{IDLE\_DFS}.p.q.\mathsf{idle}.(h.p).\mathsf{V.father.le\_rec.lp\_rec})}$$

Theorem D.5.14 *R_Tarry_DFS_COL_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\mathcal{R}_{\text{-PLUM\_DFS}}.(\mathsf{COL\_Tarry}.p.q.\mathsf{neighs.idle}.(h.p).\mathsf{V.le\_rec}) \\ .(\mathsf{COL\_DFS}.p.q.\mathsf{neighs.idle}.(h.p).\mathsf{V.le\_rec.lp\_rec})}$$

Theorem D.5.15 *R_Tarry_DFS_PROP_lp_rec_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\mathcal{R}_{\text{-TARRY\_DFS}}.(\mathsf{PROP\_Tarry}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec}) \\ .(\mathsf{PROP\_lp\_rec\_DFS}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec.lp\_rec})}$$

Theorem D.5.16 *R_Tarry_DFS_PROP_not_lp_rec_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\mathcal{R}_{\text{-TARRY\_DFS}}.(\mathsf{PROP\_Tarry}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec}) \\ .(\mathsf{PROP\_not\_lp\_rec\_DFS}.p.q.\mathsf{neighs.idle}.(\mathsf{PROP\_mes}.p).\mathsf{father.le\_rec.lp\_rec})}$$

Theorem D.5.17 *R_Tarry_DFS_DONE_DEF*

$$\frac{p \in \mathbb{P} \wedge q \in \mathsf{neighs}.p}{\mathcal{R}_{\text{-TARRY\_DFS}}.(\mathsf{DONE\_Tarry}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{father.le\_rec}) \\ .(\mathsf{DONE\_DFS}.p.q.\mathsf{neighs}.(\mathsf{DONE\_mes}.p).\mathsf{M.father.le\_rec})}$$

The following theorem states conditions under which $\mathcal{R}_{\text{-TARRY\_DFS}}$ is bitotal.

Theorem D.5.18 *BITOTAL_R_Tarry_DFS*

$$\frac{\mathsf{Network}.\mathbb{P}.\mathsf{neighs}.\mathit{starter} \\ \mathsf{distinct\_DFS\_Vars}.\mathbb{P}.\mathsf{neighs.nr\_rec.nr\_sent.M.V.father.idle.le\_rec.lp\_rec}}{\mathsf{bitotal}.\mathcal{R}_{\text{-TARRY\_DFS}} \\ .\mathbf{a}(\mathsf{Tarry}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec}) \\ .\mathbf{a}(\mathsf{DFS}.\mathbb{P}.\mathsf{neighs}.\mathit{starter}.\mathsf{iniA}.h.\mathsf{PROP\_mes.DONE\_mes.V.idle.father.le\_rec.lp\_rec})}$$

Finally, we can prove that DFS refines TARRY. (Note that in Theorem $8.12.3_{149}$ the hypothesis were assumed implicitly):

Theorem D.5.19 *DFS_refines_Tarry*

$$\frac{\mathsf{Network}.\mathbb{P}.\mathsf{neighs}.\mathit{starter} \\ \mathsf{distinct\_DFS\_Vars}.\mathbb{P}.\mathsf{neighs.nr\_rec.nr\_sent.M.V.father.idle.le\_rec.lp\_rec}}{\text{TARRY} \sqsubseteq_{\mathcal{R}_{\text{-TARRY\_DFS}},\, J} \text{DFS}}$$

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Age91] S. Agerholm. Mechanizing program verification in HOL. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Proceedings of the 1991 International Workshop on HOL Theorem Proving and its Applications*, pages 208–222, Davis, August 1991. IEEE Computer Society Press.

[AGMT95] S. Aitken, P. Gray, T.F. Melham, and M. Thomas. Interactive theorem proving: An emperical study of user activity. *Journal of Symbolic Computation*, 1995.

[Ait96] S. Aitken. An Analysis of Errors in Interactive Proof Attempts. Technical report, Department of Computer Science, University of Glasgow, 1996.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 165–175, 1988. Also available as DEC SRC Technical Report 29, 1988.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. Also [AL88].

[ALW93] M.D. Aagaard, M.E. Leeser, and P.J. Windley. Toward a super duper hardware tactic. In J.J. Joyce and C.H. Segers, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*. Springer-Verlag, August 1993.

[And92a] F. Andersen. HOL-UNITY, 1992. `http://lal.cs.byu.edu/lal/holdoc/library.html`.

[And92b] F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, March 1992.

[APP93] F. Andersen, K.D. Petersen, and J.S. Petterson. Program verification using HOL-UNITY. In J.J. Joyce and C.H. Segers, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*. Springer-Verlag, August 1993.

[Bac78] R.J.R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978.

[Bac80] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, the Netherlands, 1980.

[Bac81] R.J.R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.

[Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bac89] R.J.R. Back. A method for refining atomicity in parallel algorithms. In E. Odijk, M. Rem, and J.C. Syre, editors, *PARLE '89*, volume 366 of *LNCS*, pages 199–216. Springer-Verlag, 1989.

[Bac90] R.J.R. Back. Refinement calculus, Part II: Parallel and reactive programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 67–93. Springer-Verlag, 1990.

[Bac93] R.J.R. Back. Atomicity Refinement in a Refinement Calculus framework. Reports on computer science and mathematics Series A, No. 141, Åbo Akademi, 1993.

[Bac98] R. Backhouse, editor. *User Interfaces for Theorem Provers: An International Workshop*, Eindhoven, 1998. `http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.

[BBL93] J.P. Bowen, P. Breuer, and K. Lano. A compendium of formal techniques for software maintenance. *IEE/BCS Software Engineering Journal*, 8(5):253–262, September 1993.

[BD77] R.M. Burstall and J. Darlington. Some transformations for developing recursive programs. *Journal of the ACM*, 24(1):44–676, 1977.

[Ber97] Y. Bertot, editor. *User Interfaces for Theorem Provers: An International Workshop*, Nice, 1997. `http://www-sop.inria.fr/croap/events/uitp97-papers.html`.

[BGG+98] K. Bhargavan, C.A. Gunter, E.L. Gunter, M. Jackson, D. Obradovic, and P. Zave. The village telephone system: A case study in formal software engineering. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *LNCS*, pages 49–66, Canberra, Australia, 1998. Springer Verlag.

[BGM90] J. Burns, M. Gouda, and R. Miller. Stabilization and Pseudo-Stabilization. Technical report, University of Texas, TR-90-13, May 1990.

[BH94] J.P. Bowen and M. Hinchey. Seven more myths of formal methods: Dispelling industrial prejudices. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94:Industrial Benefit of Formal Methods*, volume 873 of *LNCS*, pages 105–117. Springer-Verlag, 1994.

[BH95a] J.P. Bowen and M. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH95b] J.P. Bowen and M. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[BJ87] F.P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[BJ93] R.W. Butler and S.C. Johnson. Formal methods for life-critical software. In *AIAA Computing in Aerospace 9 Conference*, pages 319–329, San Diego, October 19-21 1993.

[BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralized control. *2nd ACM SIGACT-SIGOPS Symposium on Distributed Computing*, pages 131–142, 1983.

[BKS84] R.J.R. Back and R. Kurki-Suonio. A case study in constructing distributed algorithms: distributed exchange sort. In *Proceedings of the Winter School on Theoretical Computer Science*, pages 1–33. Finnish Society of Information Processing Science, 1984.

[BKS88] R.J.R. Back and R. Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming languages and Systems*, 10(4):513–554, 1988.

[BKS98] M. Bonsangue, J.N. Kok, and K. Sere. An approach to object-orientation in action systems. In *Proceedings of Mathematics of Program Construction (MPC'98)*, volume 1422 of *LNCS*, pages 68–95. Springer-Verlag, 1998.

[BL96] M. Butler and T. Långbacka. Program derivation using the refinement calulator. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 93–108, Turku, Finland, 1996. Springer Verlag.

[BM88] R.S. Boyer and J. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[BM92] L.M. Barroca and J.A. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579–599, 1992.

[BM93] N. Brown and D. Mery. A proof environment for concurrent programs. In *FME'93*, volume 670 of *LNCS*, pages 196–215. Springer-Verlag, 1993.

[Bou93] R.J. Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge, December 1993.

[Bou95] R.J. Boulton. Combining decision procedures in the HOL system. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *LNCS*, pages 75–89, Aspen Grove, Utah, USA, September 1995. Springer-Verlag.

[Bow93] J.P. Bowen. Formal methods in safety-critical standards. In *Proceedings of the 1993 Software Engineering Standards Symposium (SESS'93)*, pages 168–177, Brighton, UK, 30 August - 3 September 1993. IEEE Computer Society Press.

[Bra96] S.H. Brackin. Deciding crypographic protocols adequacy with HOL: The implementation. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 61–76, Turku, Finland, 1996. Springer Verlag.

[BS91] R.J.R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, (12):17–30, 1991.

[BS92] J.P. Bowen and V. Stavridou. Formal methods and software safety. In H.H. Frey, editor, *Safety of computer control systems 1992 (SAFECOMP '92)*, pages 93–98, Zürich, Switzerland, October 1992. IFAC Symposium, Pergamon Press.

[BS93a] J.P. Bowen and V. Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 183–195. Springer-Verlag, 1993.

[BS93b] J.P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEEE Software Engineering Journal*, 8(4):189–209, July 1993.

[BS96] R.J.R. Back and K. Sere. From action systems to modular systems. *Software – Concepts and Tools*, (17):26–39, 1996.

[BSBG98] R.J. Boulton, K. Slind, A. Bundy, and M. Gordon. An Interface between CLAM and HOL. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *LNCS*, pages 87–104, Canberra, Australia, 1998. Springer Verlag.

[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*, volume 449 of *LNAI*, 1990.

[BvW89] R.J.R. Back and J. von Wright. Command lattices, variable environments and data refinement. Reports on computer science and mathematics Series A, Åbo Akademi, 1989.

[BvW90] R.J.R. Back and J. von Wright. Refinement calculus, Part I: Sequential nondeterministic programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 42–66. Springer-Verlag, 1990.

[BvW94] R.J.R. Back and J. von Wright. Trace refinement of action systems. In *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR'94)*, volume 836 of *LNCS*, pages 367–384. Springer-Verlag, 1994.

[BvW98] R.J.R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[BW90] R.J.R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[BW96] M. Butler and M. Waldén. Distributed system development in B. In *Proceedings of the First B Conference*, pages 155–168, 1996. Also available as Technical Report TUCS 1996, No. 53.

[BW98] M. Butler and M. Waldén. Parallel programming with the B Method. In E. Sekerinski and K. Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, chapter 5, pages 183–195. Springer-Verlag, 1998.

[CG92] D. Craigen and S. Gerhart. An international survey of industrial applications of formal methods. In *Z User Workshop, London 1992*, pages 1–5. Springer-Verlag, 1992.

[CGR93] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 250–267. Springer-Verlag, 1993.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Cha82] E.J.H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, 1982.

[Che83] T.-Y. Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, 9(4):504–512, 1983.

[Che95] B. Chetali. Formal Verification of Concurrent Programs: How to Specify UNITY Using the Larch Prover. Technical Report 2475, INRIA Lorraine, 1995.

[Cho93] C-T Chou. Predicates, temporal logic, and simulations. In J.J. Joyce and C.-J.H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop*, volume 780 of *LNCS*. Springer-Verlag, 1993.

[Cho94a] C-T Chou. Mechanical verification of distributed algorithms in higher order logic. In T.F. Melham and J. Camilleri, editors, *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *LNCS*. Springer-Verlag, September 1994.

[Cho94b] C-T Chou. Practical Use of the Notions of Events and Causality in Reasoning about Distributed Algorithms. Technical Report CSR-940035, UCLA, October 1994.

[Cho95] C-T Chou. Using Operational Intuition about Events and Causality in Assertional Proofs. Technical Report CSR-950013, UCLA, February 1995.

[CM89] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, May 1989.

[CS95] D.A. Cyrluk and M.K. Srivas. Theorem proving: not an esteric diversion, but the unifying framework for industrial verification. In *Proceedings of the IEEE Conference on Computer Design (ICCD'95)*, Texas, Austin, October 1995.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Din97] J. Dingel. Approximating UNITY. In *Proceedings of the 2nd International Conference on Synchronization Models and Languages (COORDINATION'97)*, volume 1282 of *LNCS*, pages 320–337, Berlin, Germany, September 1997. Springer Verlag.

[DS80] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[Fin79] S.G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Transactions*, 27:840–845, 1979.

[Fra80] N. Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, 1980.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.

[GCR94] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 11:21–28, January 1994.

[Ger75] S.L. Gerhart. Correctness preserving program transformations. In *Proceedings of the 2nd ACM Conference of Principles of Programming Languages*, pages 54–66, 1975.

[GHG+93] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, 1993.

[GKSU98] H.J.M. Goeman, J.N. Kok, K. Sere, and R.T. Udink. Coordination in the ImpUnity Framework. *Science of computer programming*, 31:313–334, 1998.

[GL95] E.L. Gunter and L. Libkin. Interfacing HOL90 with a functional database query language. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *LNCS*, Aspen Grove, Utah, USA, September 1995. Springer-Verlag.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[GMS97] J.F. Groote, F. Monin, and J. Springintveld. A Computer Checked Algebraic Verification of a Distributed Summation Algorithm. Technical Report CSR-97-14, Eindhoven University of Technology, October 1997.

[Gol90a] D.M. Goldschlag. Mechanically verifying concurrent programs with the boyer-moore prover. *IEEE Transaction on Software Engineering*, 16(9):1005–1023, September 1990.

[Gol90b] D.M. Goldschlag. Mechanizing UNITY. In *Programming Concepts and Methods*, pages 387–414. Elsevier Science Publishers B.V. (North-Holland), 1990.

[Gol92] D.M. Goldschlag. *Mechanically Verifying Concurrent Programs.* PhD thesis, Computational Logic Inc., Austin, Texas., 1992.

[Gor85] M.J.C. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[Gor89] M.J.C. Gordon. Mechanizing programming logics in higher order logic. In P.A. Subrahmanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–489. Springer-Verlag, 1989.

[GP94] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in computing, pages 26–62, 1994.

[Gra96] P. Gray, editor. *User Interfaces for Theorem Provers: An International Workshop*, York, 1996. `http://www.cs.york.ac.uk/~nam/uit/proceedings.html`.

[Gri81] D. Gries. *The Science of Computer Programming*. Springer-Verlag, 1981.

[GS96] J.F. Groote and J. Springintveld. Algebraic Verification of a Distributed Summation Algorithm. Technical Report CS-R9640, CWI, October 1996.

[Gun98] E.L. Gunter. Adding external decision procedures to HOL90 securely. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *LNCS*, pages 143–152, Canberra, Australia, 1998. Springer Verlag.

[Haa94] P.J.M. van Haaften. *Distributed Optimisation Algorithms for Network Problems*. PhD thesis, University of Utrecht, 1994.

[Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[Hal91] R. Hale. Reasoning about software. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Proceedings of the 1991 International Workshop on HOL Theorem Proving and its Applications*, pages 52–58, Davis, August 1991. IEEE Computer Society Press.

[Har93a] J. Harrison. A HOL decision procedure for elementary real algebra. In J.J. Joyce and C.H. Segers, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*. Springer-Verlag, August 1993.

[Har93b] J. Harrison. Constructing the real numbers in HOL. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications (A-20)*, pages 145–164. Elsevier Science Publications BV North Holland, IFIP, 1993.

[Har94] J. Harrison. Binary decision diagrams as a HOL derived rule. In T.F. Melham and J. Camilleri, editors, *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *LNCS*. Springer-Verlag, September 1994.

[Har96] J. Harrison. Stålmarck's Algorithm as a HOL derived rule. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 221–234, Turku, Finland, 1996. Springer Verlag.

[HC96] B. Heyd and P. Crégut. A modular coding of UNITY in COQ. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 251–266, Turku, Finland, 1996. Springer Verlag.

[Hes97] W.H. Hesselink. A mechanical proof of Segall's PIF algorithm. *Formal Aspects of Computing*, 9:208–226, 1997.

[HHJ98] U. Hensel, M. Huisman, and H. Jacobs, B. Tews. Reasoning about classes in object-oriented languages: Logic models and tools. In C. Hankin, editor, *Programming languages and systems*, volume 1381 of *LNCS*, pages 105–121, 1998.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computers programs. *Communications of the ACM*, 12:576–583, 1969.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, (1):271–281, 1972.

[How96] D.J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 267–281, Turku, Finland, 1996. Springer Verlag.

[Inc95] Random House Inc. *Random House Webster's College Dictionary*. 1995. ISBN 0-679-43886-6.

[Jac91] J. Jacob. The varieties of refinement. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, pages 441–455. Springer-Verlag, 1991.

[Jon90] B. Jonsson. On decomposing and refining specifications of distributed systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 361–385. Springer-Verlag, 1990.

[JS93] J.J. Joyce and C.H. Segers. The HOL-Voss system: model-checking inside a general-purpose theorem-prover. In J.J. Joyce and C.H. Segers, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*. Springer-Verlag, August 1993.

[Kal96] M. Kaltenbach. *Interactive Verification Exploiting Program Design Knowledge: A Model-Checker for UNITY*. PhD thesis, The University of Austin, 1996.

[Kem90] R.A. Kemmerer. Integrating formal methods into the development process. *IEEE Software*, pages 37–50, September 1990.

[Kor91] J. Kornerup. Refinement in UNITY. Technical Report TR-91-18, Department of Computer Sciences, University of Texas at Austin, 1991.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam96] L. Lamport. Refinement on state-based formalisms. Technical Report 01, DEC SRC, 1996.

[Lån94] T. Långbacka. A HOL formalization of the temporal logic of actions. In T.F. Melham and J. Camilleri, editors, *Higher Order Theorem Proving and Its Application*, volume 859 of *LNCS*, pages 332–345. Springer-Verlag, 1994.

[Lap90] J.-C. Laprie. On the assesment of safety-critical software systems. In *12th International Conference on Software Engineering*, pages 222–227, Nice, France, March 1990.

[LC93] J.-Y. Lu and S.-K. Chin. Linking Higher Order Logic to a VLSI CAD system. In J.J. Joyce and C.H. Segers, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*. Springer-Verlag, August 1993.

[Len93] P.J.A Lentfert. *Distributed Hierarchical Algorithms*. PhD thesis, University of Utrecht, 1993.

[Lev86] N.G. Leveson. Software safety: Why, what, and how. *Computing Surveys*, 18(2):125–163, June 1986.

[Lev91] N.G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, February 1991.

[Lev95] N.G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Compagny, 1995.

[LS84] S.S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, 1984.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on principles of Distributed Computing*, pages 137–151, 1987.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[LT93] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, pages 18–41, July 1993.

[LV95] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214 – 233, September 1995.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., San Fransisco, California, 1996.

[Mee86] L. Meertens. Algorithmics – towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proccedings of the CWI symposium on Mathematics and Computer Science*, volume 1 of *CWI monographs*, pages 289–334. North-Hollnad, 1986.

[Mee90] L. Meertens. Paramorphisms. Technical Report CS-R9005, CWI, Amsterdam, 1990.

[Mel89] T.F. Melham. Automating recursive type definitions in higher order logic. In P.A. Subrahmanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[Mel92] T.F. Melham. The HOL pred_sets library, 1992. `http://lal.cs.byu.edu/lal/holdoc/library.html`.

[Mer95] N. Merriam, editor. *User Interface Design for Theorem Proving Systems: An International Workshop*, Glasgow, 1995. `http://www.cs.york.ac.uk/~nam/uitp95-report/report.html`.

[MG90] C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, (27):481–503, 1990.

[Mis90] J. Misra. More on strengthening the guard. *Notes on UNITY*, 19-90, 1990. `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[Mis94] J. Misra. A logic for concurrent programming. Can be obtained at: `http://www.cs.utexas.edu/users/psp/newunity.html`, April 1994.

[Mor88] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), 1988.

[Mor89] J.M. Morris. Laws of data refinement. *Acta Informatica*, (26):287–308, 1989.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[Neu95] P.G. Neumann. *Computer Related Risks*. Addison-Wesley, 1995.

[Nic91] J.E. Nicholls. Domains of application for formal methods. In *Z User Workshop, York 1991*, pages 145–156. Springer-Verlag, 1991.

[ORSH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pau87] L.C. Paulson. *Logic and Computation:Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Pau99] L.C. Paulson. Mechanizing UNITY in Isabelle. Technical Report 467, Computer Lab, 1999.

[Pól71] G. Pólya. *Induction and Analogy in Mathematics*, volume 1 of *Mathematics and plausible reasoning*. Princeton University Press, 1971.

[Pra95] W. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Utrecht University, Oct 1995.

[PvSK90] D.L. Parnas, J. van Schouwen, and S.P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, June 1990.

[Rus94] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[RvH93] J. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[RW92] K.A. Ross and C.R.B. Wright. *Discrete Mathematics*. Prentice-Hall, 1992.

[San90] B.A. Sanders. Stepwise refinement of mixed specifications of concurrent programs. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, IFIP, pages 1–25. Elsevier Science Publishers B.V., 1990.

[San91] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[SB94] T. Schubert and J. Biggs. A tree-based, graphical interface for large proof development. In *1994 International Workshop on the HOL Theorem Proving System and its Applications*, 1994.

[Sch91] A.A. Schoone. *Assertional Verification in Distributed Computing*. PhD thesis, University of Utrecht, 1991.

[Seg83] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29:23–35, 1983.

[Ser90] K. Sere. *Stepwise Derivation of Parallel Algorithms*. PhD thesis, Åbo Akademi, May 1990.

[Sin89a] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989. `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[Sin89b] A.K. Singh. On strengthening the guard. *Notes on UNITY*, 07-89, 1989. `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[Sin91] A.K. Singh. Program refinement in fair transition systems. In *Conference on Parallel Architectures and Languages Europe (PARLE '91)*, volume 506 of *LNCS*, pages 128–147. Springer-Verlag, 1991.

[Sin93] A.K. Singh. Program refinement in fair transitions systems. *Acta Informatica*, (30):503–535, 1993.

[SKK91] K. Schneider, T. Kropf, and R. Kumar. Integrating a first-order automatic prover in the HOL environment. In M. Archer, Joyce J.J., Levitt K.N., and Windley P.J., editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*. IEEE Computer Society Press, 1991.

[SKK93] K. Schneider, R. Kumar, and T. Kropf. Eliminating higher-order quantifiers to obtain decision procedures for hardware verification. In J.J. Joyce and C.H. Segers, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*. Springer-Verlag, August 1993.

[SO89] A.K. Singh and R. Overbeek. Derivation of efficient parallel programs: An example from genetic sequence analysis. *International Journal of Parallel Programming*, 18(6):447–484, December 1989.

[Sto89] F.A. Stomp. *Design and Verification of Distributed Network Algorithms: Foundations and Applications*. PhD thesis, Technische Universiteit Eindhoven, 1989.

[SW94a] K. Sere and M. Waldén. Verification of a distributed algorithm due to Chu. In *Proceedings of the Thirteenth Annual Symposium on Principles of Distributed Computing (PODC'94)*, page 391, 1994. Full report available as Technical Report Åbo Akademi 1994, No. 156.

[SW94b] K. Sere and M. Waldén. Verification of a distributed algorithm due to Tajibnapis. In *Proceedings of the 5th Nordic Workshop on Program Correctness*, pages 161–172, 1994.

[SW96] K. Sere and M. Waldén. Reverse engineering distributed algorithms. *Software Maintenance: Research and Practice*, (8):117–144, 1996.

[SW97] K. Sere and M. Waldén. Data refinement of remote procedures. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, volume 1281 of *LNCS*, pages 267–294, 1997.

[Sym95] D. Syme. A new interface for HOL - ideas, issues and implementation. In *1995 Conference on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *LNCS*, Aspen Grove, Utah, september 1995. Springer-Verlag.

[Tar95] M.G. Tarry. Le problème des labyrinthes. *Nouvelles Annales de Mathématique*, (149):187–190, 1895.

[TBK92] L. Théry, Y. Bertot, and G. Kahn. Real Theorem Provers Deserve Real User-Interfaces. Technical Report 1684, Institut National de Recherche en Informatique et en Automatique., 1992.

[Tel89] G Tel. *The Structure of Distributed Algorithms.* PhD thesis, University of Utrecht, 1989.

[Tel94] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[Thé93] L. Théry. A proof development system for the HOL theorem prover. In *International Workshop on Higher Order Logic and its applications*, Vancouver, 1993.

[Thi98] X. Thirioux. Automatically proving UNITY safety properties with arrays and quantifiers. In J. Rolim, editor, *Parallel and Distributed Processing*, volume 1388 of *LNCS*, pages 833–843. Springer-Verlag, 1998.

[Tho94a] M. Thomas. A proof of incorrectness using the lp theorem prover: the editing problem in the Therac-25. *High Integrity Systems Journal*, 1(1):35–48, 1994.

[Tho94b] M. Thomas. The story of the Therac-25 in lotos. *High Integrity Systems Journal*, 1(1):3–15, 1994.

[Udi95] R.T. Udink. *Program Refinement in UNITY-like Environments.* PhD thesis, University of Utrecht, 1995.

[UK96] R.T. Udink and J.N. Kok. The RPC-Memory specification problem: UNITY + Refinement Calculus. In *Formal Systems Specification*, volume 1169 of *LNCS*, pages 521–540. 1996.

[Vaa95] F.W. Vaandrager. Verification of a Distributed Summation Algorithm. Technical Report CS-R9505, CWI, January 1995.

[Vel94] D.J. Velleman. *How to Prove it.* Cambridge University Press, 1994.

[vW92a] J. von Wright. Data Refinement and the Simulation Method. Reports on computer science and mathematics Series A, No. 137, Åbo Akademi, 1992.

[vW92b] J. von Wright. Data Refinement with Stuttering. Reports on computer science and mathematics Series A, No. 138, Åbo Akademi, 1992.

[Wal96] M. Waldén. Formal derivation of a distributed load balancing algorithm. In *Proceedings of the 7th Nordic Workshop on Programming Theory*, pages 508–527, 1996. Also technical report Åbo Akademi, Reports on computer science and mathematics, Series A, No. 172.

[Wal98a] M. Waldén. Distributed load balancing. In E. Sekerinski and K. Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, chapter 7, pages 255–300. Springer-Verlag, 1998.

[Wal98b] M. Waldén. Layering distributed algorithms within the B-Method. In *Proceedings of the 2nd International B Conference*, volume 1393 of *LNCS*, pages 243–260. Springer-Verlag, 1998.

[WHLL92] J. von Wright, J. Hekanaho, O. Luostarinen, and T. Långbacka. Mechanising some Advanced Refinement Concepts. Technical Report Ser. A. No. 140, Åbo Akademi, 1992.

[Win90] J.M. Wing. A specifier's introduction to formal methods. *IEEE Comp*, (9):8–24, Sept 1990.

[Wir71] N. Wirth. Program development by stepwise refinement. *CACM*, (14):221–227, 1971.

[Wri91] J. von Wright. Mechanising the temporal logic of actions in HOL. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Proceedings of the 1991 International Workshop on HOL Theorem Proving and its Applications*, pages 155–159, Davis, August 1991. IEEE Computer Society Press.

[Wri94] J. von Wright. Program refinement by theorem prover. In *BCS FACS 6th Refinement Workshop – Theory and Practice of Formal Software Development*. City University, London, UK, January 1994.

[WS96] M. Waldén and K. Sere. Refining action systems within B-Tool. In *Proceedings of Formal Methods Europe (FME'96)*, volume 1051 of *LNCS*, pages 85–104, 1996. Also available as Technical Report TUCS 1996, No. 31.

[WW93] D. Weber-Wulff. Selling formal methods to industry. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 671–678. Springer-Verlag, 1993.

[ZGK90] S. Zhou, R. Gerth, and R. Kuiper. Transformation preserving properties and properties preserved by transformations in fair transition systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 353–367. Springer-Verlag, 1990.

Samenvatting

In dit proefschrift wordt gekeken naar de toepassing van *formele methoden* om de correctheid van *gedistribueerde computer systemen* te verifiëren.

Gedistribueerde computer systemen zijn opgebouwd uit onderling communicerende computers die veelal geografisch verspreid liggen. Communicatie tussen de verbonden computers in deze gedistribueerde systemen kan gaan via het telefoonnet, via glasvezel kabels of zelfs via de satelliet, en bestaat uit het sturen en ontvangen van berichten. Het meest bekende voorbeeld van een gedistribueerd computer system is natuurlijk het internet, maar ook aan de mogelijkheid om wereldwijd "te kunnen pinnen" of vliegreizen te boeken liggen gedistribueerde systemen ten grondslag. Wat een gedistribueerd systeem precies doet, hangt af van de manier waarop de verbonden computers met elkaar samenwerken en communiceren. Een geheel van regels aangaande wie, welk bericht, wanneer en waar naartoe mag sturen teneinde een bepaald doel te bereiken wordt een *gedistribueerd algoritme* genoemd. Het nadenken en redeneren over gedistribueerde systemen is veel moeilijker dan over één opzichzelf staande computer. Dit komt omdat in een gedistribueerd systeem meerdere computers tegelijkertijd actief zijn (*paralellisme*), het niet altijd te voorspellen is wat er gaat gebeuren (*determinisme*), en de afzonderlijke computers in het systeem geen notie hebben van de globale toestand van het systeem. Hierdoor is het moeilijk om gedistribueerde algoritmen te ontwerpen en te verifiëren of ze daadwerkelijk doen waarvoor ze bestemd zijn (dus of ze correct zijn met betrekking tot het bepaalde doel waarvoor ze ontworpen zijn).

Om dit probleem het hoofd te bieden wordt al sinds eind jaren 70, intensief onderzoek gedaan naar *formele methoden*: wiskundige technieken om het gedrag van systemen te beschijven (*formele specificatie*) en eigenschappen ervan te bewijzen (*formele verificatie*). Formele methoden kunnen op 3 niveaus worden toegepast. Welk niveau moet worden toegepast hangt af van de gewenste betrouwbaarheid van het systeem dat ontworpen wordt. Het eerste niveau bestaat enkel uit het schrijven van een formele specificatie waarin het gedrag van het systeem ondubbelzinning wordt vastgelegd. De taal waarin zo'n formele specificatie wordt geschreven heet een *specificatie taal*. De specificatie taal die gebruikt wordt in dit proefschrift heet UNITY, en is speciaal ontworpen voor het beschrijven van en redeneren over gedistribueerde systemen. Het tweede niveau houdt in dat naast het schrijven van een formele specificatie ook bewezen wordt dat bepaalde gewenste eigenschappen aanwezig zijn en dat ongewenste eigenschappen afwezig zijn. De bewijzen op dit niveau gebeuren veelal op papier en kunnen formeel of informeel zijn. Het laatste, en meest rigoureuze, niveau van het gebruik van formele methoden is als alle bewijzen gemaakt op het vorige niveau

worden gechecked met behulp van een computer programma. Deze techniek noemen we *mechanische verificatie* en het computer programma dat gebruikt wordt voor de verificatie heet een *stelling bewijzer*. Dit laatste niveau van formele methoden geeft de meeste betrouwbaarheid aangezien de fouten die eventueel nog gemaakt kunnen worden bij het bewijzen op papier nu van de baan zijn. In dit proefschrift hebben wij dit laatste niveau toegepast om te redeneren over de correctheid van gedistribueerde algoritmen. De stelling bewijzer die gebruikt is om de resultaten in dit proefschrift te bewijzen heet HOL, wat een acroniem is voor Hogere Orde Logica. Nu kan iemand zich afvragen: wie heeft de correctheid van deze stelling bewijzer bewezen en wie garandeert mij dat datgene dat bewezen is met een stelling bewijzer ook daadwerkelijk correct is? Het hele HOL systeem is gebaseerd op 5 *axiomas* (een niet bewezen, maar als grondslag aanvaarde waarheid) en 8 *bewijsregels* (regels waarmee uit al bekende waarheden, nieuwe waarheden kunnen worden geconstrueerd). HOL neemt aan dat deze axiomas en bewijsregels waar zijn, en alle andere dingen die bewezen kunnen worden met HOL volgen enkel en alleen uit de waarheid van deze 13 aannamen. Omdat het hier gaat over axiomas en bewijsregels die al vele honderden jaren bestudeerd zijn en waarover consensus aangaande hun zinvolheid en waarheid is bereikt, kunnen we ervan uitgaan dat HOL betrouwbaar is. Het gebruik van stelling bewijzers tijdens de verificatie van gedistribueerde algoritmen verkleint de kans op fouten in deze algoritmen dus aanzienlijk, maar er hangt een prijskaartje aan. Het gebruiken van stelling bewijzers is moeilijk en kost veel tijd; tot nu toe is het effectief gebruiken van stelling bewijzers slechts voorbehouden aan een handjevol specialisten en het is dus ook begrijpelijk dat het bedrijfsleven twijfelt aan de economische haalbaarheid van het toepassen van stelling bewijzers tijdens het ontwikkelen van programmatuur.

Samenvattend kijkt dit proefschrift naar de toepassing van mechanische verificatie om de correctheid van *gedistribueerde algoritmen* te verifiëren. Hierbij komen we twee moeilijke en tijdrovende activiteiten tegen: het gebruik van stelling bewijzers enerzijds en het redeneren en nadenken over gedistribueerde algoritmen anderzijds. Een ander doel van ons onderzoek is dan ook om te kijken naar wat het meest complex en tijdrovend is, en wat daaraan gedaan kan worden. We kijken hierbij vooral naar gedistribueerde algoritmen. Het moge duidelijk zijn dat de complexiteit die inherent is aan een of ander gedistribueerd algoritme niet gereduceerd kan worden, en dat een bepaalde hoeveelheid tijd nodig zal zijn om zijn correctheid te bewijzen. Het is echter zo dat onnodige complexiteit geïntroduceerd kan worden door slechte representaties van een algoritmen, ongestructureerde en slecht gemotiveerde correctsheidsbewijzen, en onvoldoende analyse om classificaties van algoritmen te ontdekken. In dit proefschrift laten wij zien dat de representatie van gedistribueerde algoritmen, het redeneren en nadenken over deze algoritmen positief kan beïnvloeden. We laten zien dat betere een representatie:

- de tijd en moeite die nodig is om het gedrag en de functionaliteit van een gedistribueerd algoritme te begrijpen, aanzienlijk kan verminderen
- het makkelijker maakt om overeenkomsten en verschillen met andere algoritmen te zien
- de mogelijkheid om nieuwe algoritmen te bedenken vergroot
- de complexiteit van correctsheids bewijzen aanzienlijk reduceert

Om deze argumenten te staven bekijken we een klasse van algoritmen die wij *gedis-*

tribueerde hylomorphismen noemen. Gedistribueerde hylomorphismen zijn algoritmen die gebruikt kunnen worden om globale eigenschappen van een gedistribueerd systeem te bepalen, of om alle aangesloten computers in het systeem snel van een bepaalde gebeurtenis op de hoogte te stellen. Verschillende gedistribueerde hylomorphismen die we in de literatuur tegen zijn gekomen staan bekent onder de namen ECHO, Tarry and DFS. We verbeteren de representaties van deze algoritmen op zo'n manier dat de overeenkomsten en verschillen tussen de drie algoritmen meteen duidelijk worden uit de representatie. Het wordt hierdoor mogelijk gemaakt om over de algoritmen samen te redeneren, in plaats van over elk algoritme apart; bovendien heeft het ons in staat gesteld een nieuw gedistribueerd hylomorphisme te bedenken dat we PLUM[1] hebben genoemd. Uiteindelijk bewijzen we de correctheid van al deze gedistribueerde algoritmen, en laten zien dat de verbeterde representaties hebben geleid tot een efficiënte gestructureerde bewijsstrategie.

[1]Plum betekent "pruim" in het engels, en de naam is dan ook tot stand gekomen onder het drinken van enkele flessen pruimenwijn tijdens de Marktoberdorf Summerschool van 1996.

Curriculum Vitae

Tanja Ernestina Jozefina Vos

8 oktober 1971
Geboren te Hilversum, Nederland.

1984-1990
VWO, Laar & Berg te Laren

1990-1995
Doctoraal Informatica (cum laude) aan de Universiteit van Utrecht.

1995-1999
Assistent in Opleiding aan het Informatica Instituut van de Universiteit Utrecht.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computer Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction.* Faculty of Mathematics and Computer Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02