A Semantic Study of Data Independence with Applications to Model Checking

Ranko S. Lazić
Christ Church

A thesis submitted for the degree of D.Phil.
Hilary Term 1999

Programming Research Group
Oxford University Computing Laboratory

Abstract

Informally speaking, a concurrent system P is data independent with respect to a data type X if and only if it cannot perform any operations involving values of type X, but it can only input such values, store them, output them, and perform equality tests between them. In that case, P makes sense if any data type is instantiated for X, or in other words P is parameterised by X.

This thesis consists of a semantic study of data independence, and its applications in proving theorems which enable model checking to be used for verifying that data independent concurrent systems satisfy their specifications for all instantiations of the data independent type, in a finite time. Weak data independence, where formal operations whose result type is X are allowed, is also studied.

The language used for expressing concurrent systems is a combination of the process algebra CSP and a typed λ-calculus with inductive types. It is a smaller and more theoretically presented version of CSP_M, the machine-readable version of CSP used in tools. In this language, a closed term is regarded to be data independent with respect to X if and only if it has X as a free type variable, i.e. if and only if it is polymorphic in X.

A symbolic operational semantics of the language is presented and studied. Instead of replacing all variables by values as in ordinary operational semantics, variables whose types are type variables are left as symbols. This enables a concurrent system which is (possibly weakly) data independent with respect to X to be interpreted by a single symbolic labelled transition system, in which X is still a type variable. We also present and study symbolic normalisation. This is a procedure for transforming symbolic labelled transition systems into a form where any trace (i.e. sequence of events) can be executed in at most one way.

Logical relations for the language are defined, and it is shown that the Basic Lemma holds for them. This demonstrates formally that the language is parametrically polymorphic, and provides an entirely denotational way of reasoning about data independence.

This operational and denotational semantic study is then applied in proving theorems which provide threshold collections for the problem of whether, given a concurrent system *Impl* and a specification *Spec* which are data independent with respect to X, *Impl* satisfies *Spec* for all instantiations of X. Here, threshold collections are sufficient finite collections of concrete data types, in the sense that if *Impl* satisfies *Spec* for those data types instantiated for X, then it does for all instantiations, or for all instantiations by larger (i.e. of greater cardinality) concrete data types. We also present theorems which state that, if *Impl* does not satisfy *Spec* for some instantiation of X, then it does not for all instantiations, or for all instantiations with some property.

As a case study, we show how the theorems obtained can be applied in verifying a cache. This is especially interesting because there are two data independent types, those of addresses and values, where equality tests between addresses are present, but not between values.

Acknowledgements

I am most grateful to Mrs Hajrija & Mr Boris Vukobrat and Copechim France S.A. for a scholarship (1992–1995, where the first two years were spent studying for a BA) and for continued care, to Merton College Oxford for a Domus & Harmsworth Senior Scholarship (1995–1997), and to Christ Church Oxford for a Junior Research Fellowship (from 1997).

This thesis would not have been possible without much help, technical and otherwise, from my supervisor Professor Bill Roscoe. In particular, together with Helen MacCarthy, he provided the starting point for the work. I first met Bill in the summer of 1991, and before becoming my D.Phil. supervisor, he was my undergraduate tutor.

Many colleagues, including Stephen Brookes, Ping Gao, Michael Goldsmith, Ramin Hojati, Dominic Hughes, Martin Hyland, Norris Ip, Ralph Loader, Gavin Lowe, Oege de Moor, Kedar Namjoshi, Luke Ong, Corin Pitcher, Ben Worrell and Irfan Zakiuddin have also helped, in various ways. In particular, many thanks are due to Bill, Michael, Ping, Dominic and Ben for proof-reading, and to Luke for acting as an examiner for my Confirmation of D.Phil. Status (in 1997).

My former and present office-mates Francisco Pinheiro, Ana Cavalcanti, Corina Cîrstea, Richard Forster, Dominic, Ben, Yu Qian Zhou and Quentin Miller, as well as others in the laboratory, have made it an enjoyable place to work in. The making of pasta in the departmental microwaves is believed to have been started by Dominic, and since then Ben has shown that it is possible to almost completely live in the department. Amusing office decorations and Christmas toys have been provided by Quentin.

I am specially grateful to my parents Slobodan and Milica and brother Rastko, my godfather Renos and his family, Fr Ian, and the monks and nuns of the Stavropegic Monastery of St John the Baptist in Essex, for much love, encouragement and care.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is a semantic study of data independence. It is motivated by needs of model checking, which is a completely automatic way of verifying that concurrent systems satisfy their specifications. Given a concurrent system and a specification which are data independent, the results of the thesis enable model checking to be used to verify satisfaction for all instantiations of the data independent type, in a finite time.

In the first section, we shall introduce computer-aided verification, which is the broad area that the thesis belongs to, and point at the main problems in it. We shall then, in the following three sections, describe the problem that the thesis is addressing, outline the work which will be presented, and consider its relations to work by other authors.

1.1 Computer-aided verification

Any computer hardware or software should function correctly, and in some application fields, their correct functioning is critical. However, they are among the most complex creations of humankind, and consequently errors are frequent in their development. As many well-known examples have unfortunately shown, unnoticed errors are often present in completed products.

The purpose of computer-aided verification is to make the presence of errors less likely by using computer tools. Given a specification *Spec* and a (proposed) implementation *Impl*, such tools can be used to check whether *Spec* is satisfied by *Impl*, and if the answer is negative, to obtain further information. The tools can either be applied to completed designs, or which is more often the case, they can be used in combination with other methods (such as testing) during the course of development.

The two main approaches to computer-aided verification are model checking and theorem proving.

1.1.1 Model checking

Model checkers are tools which, given a *Spec* and an *Impl*, can completely automatically determine whether *Spec* is satisfied by *Impl*, and provide further information in case of a negative answer. They are based on exhaustive exploration of state spaces, which can be performed either explicitly or implicitly.

For most model checkers, *Impl* is a concurrent system, and *Spec* can either be a formula of a temporal logic, or an automaton of a certain kind, or another concurrent system. In the last case, the relation of satisfaction can either be some kind of equivalence, or some kind of partial ordering. (The latter possibility, where the partial ordering is refinement, will be a characteristic of the framework in this thesis: see Sections 2.1.3, 2.1.4 and 2.6.)

A price which is paid for complete automation is that *Spec* and *Impl* need to be concrete, i.e. that any of their parameters need to be instantiated with concrete values, and that they need to have finitely many states, in an appropriate sense.

In recent years, model checking has demonstrated itself to be very successful in practice, and applicable to industrial-size problems. Some general references are [McM93, IP96, CAV].

1.1.2 Theorem proving

Theorem provers are tools which can be used to prove that a given *Spec* is satisfied by a given *Impl*. In contrast to model checkers, they are usually not completely automatic, but require assistance from the user.

Spec and *Impl* are expressed in formalisms which allow the statement that *Spec* is satisfied by *Impl* to be expressed as a formula in a logic. Based on deduction rules for the logic, a theorem prover can be employed to attempt to prove the formula. Depending on which logic and which deduction rules are used, it may also be possible to obtain useful information in the case where no proof is found.

An advantage of theorem provers over model checkers is that *Spec* and *Impl* do not necessarily need to be concrete, or finite-state. If they are not concrete, there is a possibility of a single proof showing that *Spec* is satisfied by *Impl* for all instantiations of their parameters.

Except for the requirement of user assistance, theorem proving (which is sometimes called computer-aided deduction) has also demonstrated itself to be very successful in practice, in particular for verification of parameterised or infinite-state systems. Some general references are [CADE, CAV].

1.1.3 Parameterised systems

Concurrent systems often have one or more parameters, such as:

- data types (for example, a communication protocol is likely to have as a parameter the data type of values which it communicates);
- numbers of identical parallel components in networks, or more generally network structure;
- capacities of buffered channels.

If *Spec* and *Impl* have parameters, one usually wants to verify that *Spec* is satisfied by *Impl* for all instantiations of the parameters. If one or more of the parameters can be instantiated by infinitely many different values (as is usually the case with each of the three kinds of parameter mentioned above), it is not immediately possible to perform such a verification completely automatically. This is because, as we remarked in Sections 1.1.1 and 1.1.2, model checkers can only be applied for one instantiation at a time, and theorem provers are likely to require user assistance.

More formally, we can state the following decision problem, which we call the *Parameterised Verification Problem* (or *PVP*):

- Given a *Spec* and an *Impl* which have parameters, is *Spec* satisfied by *Impl* for all instantiations of the parameters?

As has been shown in [AK86], the PVP is in general undecidable.[1]

[1]For an introduction to computability, see e.g. [Cut80].

In recent years, much research has been seeking to establish restrictions under which the PVP either becomes decidable, or there exist algorithms that can usefully be applied to it (but that do not always terminate). Indeed, this is the area to which the work presented in this thesis belongs (see Sections 1.2 and 5.6).

A number of references on the PVP which are not given in this thesis (see Section 1.4) can be found in [Laz98].

1.1.4 State explosion

The state space of a network is in the worst case the cartesian product of the state spaces of its components, and so the number of states of a network may grow exponentially in how many components it has. A similar phenomenon is that the number of states of a system may grow exponentially in sizes (i.e. cardinalities) of data types which are instantiated for its data type parameters (see the beginning of Section 1.1.3).

Since model checking is based on exhaustive exploration of state spaces (see the beginning of Section 1.1.1), these exponential dependences can result in major decreases of efficiency, which often is the main obstacle in practical applications of model checking. At least in the case of networks, this problem is known as the State-explosion Problem.

Overcoming the state-explosion problem has therefore been a central topic of theoretical and practical research. Some of the techniques which have been proposed and successfully used in practice are:

- abstraction, e.g. [CGL94, LG+95, DF95, Dam96];
- symbolic execution, e.g. [HL95a, SB95, CZ+97];
- binary decision diagrams (or BDDs), e.g. [Bry86, BC+92, McM93];
- partial-order methods, e.g. [Val89, PL90];
- elimination of symmetry, e.g. [CE+96, ES95, ID96a, Jen96];
- equivalence-preserving transformations, e.g. [RG+95, ST97, Kok98];
- exploitation of data independence, e.g. [ID96a, HB95, HDB97, LR98].

We have included techniques for exploitation of data independence because they can be used to replace a data type which was instantiated for a data type parameter by a smaller data type. This is one example where verifying parameterised systems (see Section 1.1.3) overlaps with tackling the state-explosion problem.

1.1.5 Infinite-state systems

Concurrent systems sometimes have infinitely many states. This is often due to one or more parameters having been instantiated by infinite values. For example, a concurrent system may be infinite-state because it involves a buffered channel of infinite capacity.

Given a *Spec* and an *Impl* such that *Impl* is infinite-state, the verification cannot immediately be performed completely automatically, because model checkers require *Spec* and *Impl* to be finite-state (see Section 1.1.1) and theorem provers are likely to require user assistance (see Section 1.1.2).

It has therefore been an active research topic to find classes of such *Spec* and *Impl* for which there exist useful algorithms for verification (that ideally always terminate). See e.g. [INF96, INF97, INF98].

This topic often overlaps with verifying parameterised systems (see Section 1.1.3), and with tackling the state-explosion problem (see Section 1.1.4).

1.2 The problem this thesis addresses

This thesis addresses a version of the Parameterised Verification Problem (see Section 1.1.3) in which the parameters are data types with respect to which *Spec* and *Impl* are *data independent*.

1.2.1 Data independence

Informally speaking, a concurrent system P is data independent with respect to a data type X if and only if it cannot perform any operations involving values of type X, but it can only input such values, store them, output them, and perform equality tests between them. In that case, P makes sense if any data type is substituted for X, or in other words X is a data type parameter of P (see Section 1.1.3).

If a formal language is used for expressing concurrent systems, then data independence with respect to X can be formalised as a syntactic property. In the language which is used in this thesis, a closed term P is regarded to be data independent with respect to X if and only if it has X as a *free type variable*, i.e. if and only if it is *polymorphic* in X (see Section 2.7.1). In a more complex language such as CSP_M [Sca92, Sca98, Ros98a], which is the version of the process algebra CSP [Hoa85, Ros98a] used in tools, additional restrictions are required (see [Laz99]).

Alternatively, data independence can be formalised as a property of denotational semantics of concurrent systems, as was first done in [Wol86].

Although data independence is a strong property, it is possessed by many practical concurrent systems. For example:

- Communication protocols are typically data independent with respect to the type of values they communicate.
- Memories, databases and systems involving them are frequently data independent with respect to the types of addresses and values.
- Agents and the spy in security protocols (see [Ros98a, Section 15.3] and references given there) are typically data independent or nearly so with respect to the types of agent identities, nonces and keys.
- Nodes in a system consisting of many nodes are sometimes data independent with respect to the type of their identities, although the whole system is typically not.

(Some further details about these examples can be found in Section 2.7.1.)

1.2.2 The problem

If a specification *Spec* and a (proposed) implementation *Impl* are data independent with respect to X (which needs to be appropriately interpreted in frameworks in which *Spec* is not a concurrent system), one usually wants to verify that *Spec* is satisfied by *Impl* for all instantiations of X, i.e. for all data types substituted for X. Since there are infinitely many different data types which can be substituted for X, it is not immediately possible to perform such a verification completely automatically, as we remarked in Section 1.1.3.

However, considerations of specific examples suggest that, at least under appropriate additional conditions, such verifications can be performed completely automatically. In particular, they suggest that it suffices to verify that *Spec* is satisfied by *Impl* with appropriate small finite data types substituted for X. (For the framework we shall be using in this thesis, such conjectures were made in [RM94]: see Section 5.7.)

Therefore, the aim of the thesis is to establish, based on a semantic study of data independence, additional conditions under which the following decision problem either becomes decidable, or there exist algorithms that can usefully be applied to it (but that do not always terminate):

- Given a *Spec* and an *Impl* which are data independent with respect to X, is *Spec* satisfied by *Impl* for all instantiations of X?

It would be practically useful if such algorithms (including decision procedures) could at least partly be implemented by existing model checkers (see Section 1.1.1). In particular, a principal part of the aim of the thesis is to produce algorithms which are based on 'threshold collections', which are finite collections of data types such that if *Spec* is satisfied by *Impl* for those types substituted for X, then it is satisfied for all types, or for all types of larger size (i.e. cardinality).

As in the statement of the decision problem above, we shall be concentrating on the case where there is one data independent type X. If there is more than one such type, the results obtained in the thesis can be applied to one at a time. (An example of this can be found in the case study in Chapter 6.)

1.2.3 Weak data independence

The thesis will also study *weak data independence*, which is more general than data independence in that it allows formal operations whose result type is X (see Section 2.7.2). In other words, a concurrent system P can be thought of as weakly data independent with respect to X if and only if values of type X can affect the control-flow of P only through equality tests.

1.2.4 Parameterisation by network size/structure

The thesis will be directly concerned only with versions of the PVP (see Section 1.1.3) in which the parameters are data types. However, as other authors have shown, its results can also be applied to verifying systems which are parameterised by network size/structure (see Section 5.5.5).

1.3 The work presented in the thesis

The thesis presents a study of data independence in terms of operational semantics and denotational semantics, applications of the study in proving theorems which address the problem stated above, and a nontrivial example of how the theorems can be applied.

1.3.1 The language

The language we shall be using for expressing concurrent systems is a combination of the process algebra CSP [Hoa85, Ros98a] and a typed λ-calculus with inductive types [Mit90, GLT89, Loa97, Bir98], and Chapter 2 is devoted to presenting its syntax and denotational semantics. This language is a smaller and more theoretically presented version of CSP$_M$ [Sca92, Sca98, Ros98a], which is the machine-readable version of CSP used in tools.

The combination is such that the whole language is an applied typed λ-calculus with process types. We shall first define the syntax of types and their denotational semantics in terms of partial orders. The syntax of terms will then be defined by typing rules, and their denotational semantics will be defined by induction on those. Well-definedness, which will turn out not to be trivial, will then be considered.

1.3.2 Refinement as the notion of correctness

For this language, as for CSP, the notion of correctness is *refinement*. In other words, both specifications and (proposed) implementations are processes (which is the same as concurrent systems), and *Spec* is considered to be satisfied by *Impl* if and only if *Spec* is refined by *Impl*, which is the case if and only if any possible behaviour of *Impl* is also a possible behaviour of *Spec*. We shall define this formally, relative to the finite-traces, stable-failures and failures/divergences models. These are the three denotational models of processes which will be considered in the thesis, and which are supported by currently existing tools for CSP (see Sections 2.1.2–2.1.4).

1.3.3 Data independence and weak data independence

The fact that the language is typed will enable data independence and weak data independence to be defined as simple syntactic properties of terms. Data independence will allow formal constants, i.e. free variables of the data independent type.

1.3.4 Symbolic operational semantics

As we have indicated in Section 1.2, the aim of the thesis is to address, based on a semantic study of data independence, a decision problem which is concerned with how the truth of refinements between data independent processes changes with varying the instantiation of the data independent type. Since refinement is defined in terms of denotational semantics, this will require knowledge of how the denotational semantics of a data independent process changes with varying the instantiation.

Symbolic operational semantics, that Chapter 3 is devoted to, will provide a way of tackling this problem. In contrast to denotational semantics, symbolic operational semantics does not require data independent types to be assigned concrete partial orders, but allows them to be left as type variables. The single interpretation of a data independent process in symbolic operational semantics will then act as a bridge among all the various interpretations in the denotational semantics, by translations to and from it.

Operational semantics in general is at a lower level of abstraction than denotational semantics: it interprets a program by specifying how it executes. In the standard operational semantics of CSP [Ros98a, Chapter 7], this amounts to interpreting a process by a labelled transition system (LTS), which is a directed graph whose nodes interpret the states of the process and whose arcs (which are called transitions) are labelled by events (i.e. actions). In symbolic operational semantics, the idea is the same, except that instead of assigning concrete values, some or all variables are left as symbols. The variables we shall be leaving as symbols will be those of data independent types, which is how we shall be able to interpret processes whose data independent types are left as uninterpreted type variables. The resulting interpretations will be *symbolic labelled transition systems (SLTSs).*

More precisely, we shall present and study symbolic operational semantics for weakly data independent processes in our language, so that in addition to variables of weakly data independent types, operations involving such types will also be left as symbols.

We shall then state a few conditions on terms (in particular, on processes), which will be important in the rest of the thesis. Their purpose is to restrict the degrees to which a term can contain equality tests between values of (weakly) data independent types or nondeterminism whose effects are not immediately apparent. We shall also examine the effects that each of the conditions has on SLTSs.

1.3.5 Symbolic normalisation

Symbolic normalisation will be presented and studied also in Chapter 3. This is a procedure for transforming SLTSs of data independent processes into a form where any trace (i.e. sequence of events) can be executed in at most one way, which will be used in proving a central theorem of the thesis (see Section 5.4).

1.3.6 Logical relations

In a typed language, a 'logical relation' is an assignment of relations to types which is such that, for each type construct (such as forming function types), the relation assigned to any constructed type is obtained in a specific way from the relations assigned to its component types. Thus any logical relation can be obtained from the relations it assigns to type variables and ground types, by induction on the structure of types.

The so-called Basic Lemma of Logical Relations (see e.g. [Mit90, OHe96, Plo80, Rey83, Wad89]) states that, for any logical relation such that the relations it assigns to ground types are appropriate (in a precise sense), we have that, for any term, related assignments to its free term variables yield related denotational interpretations.

The Basic Lemma has many applications in both theory and practice of typed languages. On the theoretical side, it can for example be used for cutting out junk elements from denotational interpretations of types with universally quantified type variables (see e.g. the references given above). On the practical side, the Basic Lemma can for example be used for reasoning about abstraction and data refinement (see e.g. [KO+97]).

In Chapter 4, we shall define logical relations for the language in this thesis and present the Basic Lemma for them. This will show formally that the language is 'parametrically polymorphic', or in other words that any term 'behaves uniformly' for different instantiations of its free type variables.

In cases when the Basic Lemma can be applied to data independent or weakly data independent processes, it will provide more direct ways of relating their various denotational interpretations than by translating to and from their symbolic operational interpretations (see the remarks above about Chapter 3).

1.3.7 Threshold collections

In Chapter 5, we shall address directly the decision problem which has been the main motivation for the work presented in the thesis (see Section 1.2). Namely, given two processes *Spec* and *Impl* which are (possibly weakly) data independent with respect to X, is *Spec* refined by *Impl* for all instantiations of X and the associated constants and operations?

More precisely, we shall present theorems which provide 'threshold collections'. These are sufficient finite collections of instantiations of X and the associated constants and operations, in the sense that if the refinement holds for those instantiations, then it holds for all instantiations, or for all instantiations which substitute larger (i.e. of greater cardinality) data types for X.

Additionally, we shall present theorems which state that, if *Spec* is not refined by *Impl* for some instantiation of X and the associated constants and operations, then the refinement fails for all instantiations, or for all instantiations with some property. Thus, if the refinement fails for some instantiation from a threshold collection which is provided by a theorem of the former kind, a theorem of this latter kind may apply and give more information.

As we have indicated in Section 1.2, both addressing the stated decision problem, and aiming for the two particular kinds of theorems, have in turn been motivated by the needs of practical model checking.

1.3.8 Further developments

A few further developments which either could not be presented in detail in this thesis because of space limitations, or which are by other authors, will also be mentioned in Chapter 5. These will include: a theorem which applies to a more general property than data independence, in that it allows arrays whose indices and values come from two type variables, as well as many-valued predicates on the type of indices; proving security protocols with model checkers by using techniques from this thesis; and combining the work in this thesis with induction, for proving with model checkers systems which are parameterised by network size/structure.

1.3.9 A case study: verifying a cache

Chapter 6 contains a more substantial example than those elsewhere in the thesis. Namely, it will show how the theorems presented in Chapter 5 can be applied in verifying a cache. This is especially interesting because there will be two data independent types, those of addresses and values, where equality tests between addresses will be present, but not between values.

1.3.10 Tools

Sections 3.8 and 5.6 are specifically devoted to discussing applications of the work in the thesis to practical model checking.

1.4 Relations to work by other authors

This thesis is certainly not the first piece of literature to study data independence, or to exploit it in model checking.

One contribution of the thesis is that the semantic study of data independence and weak data independence (in Chapters 3 and 4, and in parts of Chapter 5), and its applications in proving theorems which facilitate verification by model checking (in Chapter 5), have been performed in a framework in which the language for expressing concurrent systems is a combination of the process algebra CSP [Hoa85, Ros98a] and a typed λ-calculus with inductive types [Mit90, GLT89, Loa97, Bir98], and in which the notion of correctness is refinement (see Section 1.3.2). A number of distinguishing features of this framework, and some of the challenges they had posed, will be discussed in Section 2.3.

In particular, the semantic study has been performed both in terms of operational semantics (by symbolic operational semantics and symbolic normalisation, in Chapter 3) and denotational semantics (by logical relations, in Chapter 4). On the other hand, the extensive resources of

CSP in applied theory [Ros98a], such as security and fault-tolerance, enable wide applicability of the theorems in Chapter 5.

For some of the work presented in the thesis, we are not aware of comparable work by other authors. This is the case at least with symbolic normalisation (see Sections 3.6 and 3.7), logical relations (see Chapter 4) and some of the theorems which are either presented or mentioned in Chapter 5.

Much more detailed and specific discussions of the relations to work by other authors can be found in the 'Notes' sections, at the ends of each of the Chapters 2–7.

Chapter 2

The language

In this chapter, we present the language we shall be using for expressing concurrent systems, and define its denotational semantics.

The thesis can be seen as consisting of two kinds of work: general techniques which are not specific to our language, and their applications which are. The primary purpose of the applications are the practical theorems (see Chapter 5), but they are also of interest because they deal with a number of important features of our language (see Section 2.3) that are typically not present elsewhere.

In the first two sections we discuss the process algebra CSP [Hoa85, Ros98a] and its integration with typed λ-calculus [Mit90, GLT89, Loa97, Bir98], which is how our language is obtained. We point to the main distinguishing features of the language in the third section. Its syntax and denotational semantics are given in the fourth and fifth sections. In the sixth and seventh sections, we formally define refinement (the notion of correctness in CSP and in our language), data independence and weak data independence.

2.1 About CSP

Our language is based on the process algebra CSP [Hoa85, Ros98a]. CSP is a notation for describing concurrent systems, i.e. computer systems (software or hardware) which consist of a number of simultaneously existing processes that can communicate with each other and with the overall environment. It is also a collection of mathematical models and techniques for reasoning about such systems.

Concurrent systems can be found in many different areas of computing, and at various levels of abstraction. For example, each of the following is a concurrent system:

- a number of workstations connected by a network;
- a parallel computer, containing a number of processors;
- a VLSI circuit in which different subcomponents can run concurrently.

Reactive systems can also be considered as belonging to the same category: they are systems that repeatedly interact with their environment, but do not necessarily contain any concurrency themselves (for example, a cash-point machine).

In CSP, a concurrent system is seen in terms of its communications; in other words, we abstract away from internal details of its processes. Thus CSP is most immediately suited to systems in which all communication is explicit, through physical or virtual channels.

Alternatively, concurrent systems can be seen in terms of their states, where a state can be thought of as a snapshot of all the memories in the system, taken between computation steps; this view is particularly appropriate for systems in which communication is done implicitly, by reads and writes to a shared memory. CSP can be applied to such systems also: for example, we can treat a shared memory as another process with reads and writes to it modelled as communications, and we can add special communications between the system and the environment to reveal properties of its states.

In some application areas, it is important to be able to assume fairness [Fra86], i.e. that no process can be left waiting forever to be executed. In recent years, ways of incorporating fairness into CSP have been developed [Bro94, Ros98a].

Thus CSP is a general formalism for describing and analysing concurrent systems, both communication-based ones and state-based ones [DV90]. It is important to note that the analysing is on the level of patterns of communications and properties of states; for instance, CSP is not suited to studying efficiency of algorithms for parallel computers.

Since the difference between concurrent systems and processes does not exist in CSP, enabling nested parallel compositions, we shall from now on use just the term 'process'.

2.1.1 Events and communication

All a CSP process can do is perform *events*, which are its means of communicating with other processes and with the environment. (In the literature on other process algebras, the term *actions* is often used instead.)

When two processes P and Q are executing in parallel, there is a specified set S of events through which they communicate. At any point, each can perform one of a number of events, which depend on their current states. Any event from S that they both offer, they can perform together, so that it becomes a single event of the parallel composition. Other than those, each can on its own perform only events that are not in S.

Example 2.1.1 Let

$$\begin{aligned}
&\mathit{SCOPY}_1 = \mathit{in} \longrightarrow \mathit{mid} \longrightarrow \mathit{SCOPY}_1\\
&\mathit{SCOPY}_2 = \mathit{mid} \longrightarrow \mathit{out} \longrightarrow \mathit{SCOPY}_2\\
&S = \{\mathit{mid}\}
\end{aligned}$$

so that SCOPY_1 can perform the event in and then the event mid repeatedly, SCOPY_2 can perform the event mid and then the event out repeatedly, and S consists of the event mid.

Then their parallel composition, written

$$\mathit{SCOPY}_1 \underset{S}{\parallel} \mathit{SCOPY}_2$$

is equivalent to the process SCSC in

$$\begin{aligned}
\mathit{SCSC} &= \mathit{in} \longrightarrow \mathit{SCSC}'\\
\mathit{SCSC}' &= \mathit{mid} \longrightarrow ((\mathit{in} \longrightarrow \mathit{out} \longrightarrow \mathit{SCSC}')\\
&\qquad\qquad\quad \Box\ (\mathit{out} \longrightarrow \mathit{SCSC}))
\end{aligned}$$

In the state of SCSC' after performing mid, there is a choice whether to perform in followed by out and become SCSC' again, or to perform out and go back to SCSC. The choice is present because, in that state of the parallel composition, each of SCOPY_1 and SCOPY_2 can perform an event outside of S, namely in and out respectively.

$SCOPY_1$ and $SCOPY_2$ can be thought of as one-place buffers without data, so that $SCSC$ is a two-place buffer without data in which the channel *mid* linking $SCOPY_1$ and $SCOPY_2$ remained visible. ■

Events in CSP are considered to be *instantaneous*, so that a communication which in practice takes a period of time has to be modelled by perhaps two events marking its beginning and end. Similarly, all communications are *synchronous* in that they require simultaneous participation of both parties (as with the event *mid* in $SCOPY_1 \underset{S}{\|} SCOPY_2$ above), so that asynchronous communicating needs to be modelled by perhaps providing explicit buffers.

Observe that the event *mid* is still present in $SCOPY_1 \underset{S}{\|} SCOPY_2$, enabling further parallel components communicating through it to be added. In this way, CSP supports *many-way synchronisation*, where more than two processes participate in performing a single event.

The invisible event, nondeterminism and divergence

So far we have mentioned only visible events. In addition to those, there is a special invisible event called τ.

One way of introducing τ events is by *hiding* visible events, i.e. by making them unobservable by the environment.

Example 2.1.2 In the parallel composition in Example 2.1.1, we can for example hide the event *mid*.

Since τ events cannot be observed by the environment, the resulting process, written as

$(SCOPY_1 \underset{S}{\|} SCOPY_2) \setminus S$

is equivalent to SB in

$SB = in \longrightarrow SB'$
$SB' = (in \longrightarrow out \longrightarrow SB') \,\square\, (out \longrightarrow SB)$

Having hidden the internal channel between the two one-place buffers $SCOPY_1$ and $SCOPY_2$, the resulting process is now more clearly a two-place buffer without data. ■

The other way of introducing τ events is by nondeterministic choice operators (see Section 2.5.2 and [Hoa85, Ros98a]). Since τ events are unobservable, a nondeterministic choice can be modelled by a state in which a number of τ events are possible, each leading to a branch of the choice. Such states, perhaps with a number of visible events offered additionally, can also be produced by hiding, which means that hiding can result in nondeterminism. This, however, did not happen in the example above.

In CSP, there is an assumption that whenever all parties are ready to perform a communication, the communication may be performed immediately.[1] This applies most clearly to τ events: since a τ event does not require participation of the environment, it may be executed as soon as it is reached. Visible events are viewed differently, as they are potential communications with the environment or with processes that can be added in parallel later.

In particular, a possibility of performing infinitely many consecutive τ events is seen as an error, known as *livelock* or *divergence*. It is a possibility of engaging in internal activity for infinitely long, without offering any visible events to the environment.

[1]CSP is an untimed process algebra. There is, however, a large literature on timed process algebras, including Timed CSP [RR88, Sch98] which is an extension of CSP that can be used to reason about real time. For ways of reasoning about discrete time in basic CSP, using special *tock* events, see Chapter 14 of [Ros98a].

Input and output

An input is written as $c?x$, where c is a name, thought of as a channel name, and x is a variable. It stands for receiving a value along c into x. A more general form is $c?x : S$, which can receive only values from a set S.

Formally, inputs are no more than abbreviations for choices of visible events, so that

$$c?x : S \longrightarrow P(x)$$

is equivalent to

$$\Box_{x:S}\, c.x \longrightarrow P(x)$$

which offers a choice among all visible events $c.x$ with x from S and after each behaves like a process $P(x)$.

Outputs are written as $c!r$, where r is a value that is being output along c, or more precisely a term or expression evaluating to it. They have exactly the same meaning as events $c.r$; the exclamation mark is written to aid intuition.

Example 2.1.3 We can add data to our one-place buffers as follows:

$$\begin{aligned}&COPY_1 = in?x \longrightarrow mid!x \longrightarrow COPY_1\\ &COPY_2 = mid?x \longrightarrow out!x \longrightarrow COPY_2\\ &S = \{mid.*\}\end{aligned}$$

where the definition of S means that it includes all events along the channel mid.

Following what we did in Examples 2.1.1 and 2.1.2, we can form a two-place buffer with data by putting $COPY_1$ and $COPY_2$ in parallel over S and then hiding all events in S. The resulting process

$$BImpl = (COPY_1 \underset{S}{\|} COPY_2) \setminus S$$

is equivalent to B in

$$\begin{aligned}B &= in?x \longrightarrow B'(x)\\ B'(x) &= (in?y \longrightarrow out!x \longrightarrow B'(y))\\ &\quad\ \Box\ (out!x \longrightarrow B)\end{aligned}$$

■

2.1.2 Denotational models

All three kinds of semantics [van90] have been developed and extensively studied for CSP [Ros98a]: operational, denotational and algebraic. In this thesis, we shall primarily be using the operational and denotational approaches.

In denotational semantics, each program is assigned a mathematical object that is in some sense its meaning. The objects, called semantic values, are drawn from denotational models, which are typically mathematical structures such as partial orders or topological spaces. A denotational model can then be used to reason formally and prove theorems about the programs, for instance to validate syntactic transformations and inferences.

There have been a number of denotational models developed for CSP. They are all *behavioural*, in the sense that the semantic value of a process is the set of all its behaviours of an appropriate kind.

The following are the three models we shall be concentrating on. They are also the models supported by tools (see Section 2.1.4).

The finite-traces model. In this model, the semantic value of a process is the set of all finite sequences of visible events, called traces, that it can perform.

For example, the semantic value of $SCOPY_1$ from Example 2.1.1 is

$$\{\langle in, mid\rangle^n, \langle in, mid\rangle^{n}{}^\frown\langle in\rangle \mid n \in \mathbb{N}\}$$

where t^n stands for t repeated n times, and $^\frown$ is the symbol for concatenation.

The stable-failures model. Here the semantic value of a process consists of two sets: the set of all its finite traces (as above), and the set of all its stable failures, where a stable failure is a finite trace t together with a set of events A all of which can be refused in a *stable state* of the process after t. The sets A are called refusal sets, or just refusals.

A state is called stable when it has no τ events available. By a stable state after t we mean a stable state that is reachable by some number of τ events after an execution of t, where the execution of t may also involve any numbers of τ events before each of the visible ones.

The failures/divergences model. Here the semantic value also consists of two sets: the set of all stable failures of the process, and the set of all its divergences, where a divergence is a trace after which the process can diverge, i.e. perform infinitely many consecutive τ events.

In fact, the first set does not contain only the stable failures, but also all (t, A) with t a divergence and A any set. The intuition behind adding these failures is that, after potential divergence, a process is considered to be 'broken' and thus able to do anything. They also make divergent processes minimal in the refinement ordering (see Section 2.1.3).

The three models are progressively more detailed. The finite-traces model can be used for reasoning only about safety properties (i.e. that the process cannot perform any 'wrong' traces), whereas in the failures/divergences model one can reason both about safety properties and liveness properties (i.e. that the process *will* offer desired events after appropriate traces).

2.1.3 Refinement

The primary ordering associated with the three models above, and most other denotational models for CSP, is the reverse subset order. Thus, in the finite-traces model, a semantic value is 'greater' than another if it contains fewer traces. In the other two models, the ordering is pointwise, so that for instance both fewer failures and fewer divergences are required.

The reverse subset ordering is known as the *refinement* ordering, because when a process has fewer behaviours than another, one can think of it as less nondeterministic, and thus 'better' from the point of view of any user. In other words, a process is refined by any other process that is bigger than it in the ordering.

The refinement ordering is also central in the theory [Hoa85, Ros98a]. For example, semantic values of recursively defined processes, such as the ones in Examples 2.1.1–2.1.3, are the least or greatest (depending on the model) fixed points with respect to the refinement ordering.

A remarkable feature of CSP is that the refinement ordering provides the notion of correctness of processes: both specifications (i.e. correctness conditions) and implementations are processes, and the specification is satisfied if its is refined by the implementation. This is indeed the notion of correctness we shall be considering. (There is a lot of similarity with correctness through language containment [Kur90, CBK90].)

Example 2.1.4 Consider the process $BUFF$ in

$$\begin{aligned}
&BUFF = in?x \longrightarrow BUFF'(x)\\
&BUFF'(x) = ((in?y \longrightarrow out!x \longrightarrow BUFF'(y)) \sqcap STOP)\\
&\qquad\qquad \Box\ (out!x \longrightarrow BUFF)
\end{aligned}$$

where $\sqcap$ is the nondeterministic choice operator and $STOP$ is a process that does nothing, i.e. cannot perform any visible or invisible events.

$BUFF$ behaves like the two-place buffer *BImpl* in Example 2.1.3, except that, when it contains a single piece of data x, it can nondeterministically decide not to accept another input but to be able only to output x. This in fact exactly captures the standard minimal requirements of what it means to be a two-place buffer [Hoa85, Ros98a], so that $BUFF$ is the most nondeterministic two-place buffer. In other words, a process P refines $BUFF$ if and only if P is a two-place buffer, so that $BUFF$ is *the two-place buffer specification.*

In particular, *BImpl* refines $BUFF$ in each of the finite-traces, stable-failures and failures/divergences models (see Section 2.1.2). On the other hand, $BUFF$ refines *BImpl* in the finite-traces model since in that model $Q \sqcap STOP = Q$ for any Q so that $BUFF = BImpl$, but in the two more complex models this reverse refinement is not true. More precisely, the set of all failures of $BUFF$ is not a subset of the set of all failures of *BImpl* because the former contains failures such as

$$(\langle (in, v)\rangle, \{(in, w)\})$$

which cannot be in the latter. ■

The reason why refinement is adequate as the notion of correctness is the wealth of CSP operators, such as nondeterministic choice and hiding, which make it possible to write correctness conditions as processes and to extract relevant aspects of implementations. Secondly, no extra work is required to support methods such as compositional verification (as all the standard CSP operators are monotonic with respect to refinement) and step-wise verification (as refinement is a partial ordering and thus transitive). For more on this topic, see [Ros98a, particularly pp. 46–47].

For example, each of the following properties are expressible by one or more refinements:

- determinism of a process, i.e. that from the point of view of the environment it behaves deterministically although it can contain nondeterministic choices and τ events (see Section 2.1.1);
- security of a process, in the sense that no information flow from a high level user to a low level one should be possible;
- separability of a process, i.e. that it is decomposable into a parallel composition of processes which have given disjoint sets of events;
- fault tolerance of a process, i.e. that it can fully or partially recover from errors such as those caused by unreliable or potentially faulty components;
- feature interaction in telecomms.

(For the corresponding theoretical developments and case studies, see [Ros98a], especially Chapter 12, and references given there.)

2.1.4 Tools

There are a number of tools for CSP. Most significant for this thesis are model checkers, which can check automatically and efficiently whether refinement holds between two given finite-state processes. At the time of writing, they are FDR [Ros94a, Ros98a, FS97] and ARC [Yan96].

Example 2.1.5 The processes *BImpl* from Example 2.1.3 and *BUFF* from Example 2.1.4 can be entered into FDR or ARC, and they can tell us that *BImpl* refines *BUFF* in all three models and that *BUFF* refines *BImpl* only in the finite-traces model.

When performing such checks, the data type of values that *BImpl* and *BUFF* communicate has to be defined to be some concrete finite nonempty set, because the model checkers require the processes to be concrete (i.e. to have no parameters left undefined) and finite-state.

However, because *BImpl* and *BUFF* are data independent (see Section 2.7.1), it will follow that it suffices to consider certain small finite sets (see Example 5.4.1). ■

The model checkers make the contributions of the thesis immediately useful in practice (see in particular Sections 3.8 and 5.6). The other direction is also important: as we have discussed in Chapter 1, the work reported in the thesis was motivated by the problem of automatic verification of parameterised processes, and has continued to receive vital input from practice, in particular from case studies.

To keep the link with practice close, our language is based on CSP$_M$ [Sca92, Sca98, Ros98a], the machine-readable version of CSP used in tools; it is smaller and more theoretically presented than CSP$_M$ to make the developments in the thesis more focussed. The relationship between our language and CSP$_M$ will be discussed further in Section 3.8.

On the other hand, in some of our papers [Laz99, LR98], and in [Ros98a, Section 15.2], we have worked with CSP$_M$ directly.

2.2 Integration with typed λ-calculus

CSP has traditionally [Hoa85, Ros98a] been presented and studied without going into details of how data within processes are computed. Those details had to be filled in when the machine-readable version CSP$_M$, for use in tools, was being designed [Sca92, Sca98, Ros98a]: CSP was combined with a functional language that can be used to compute the data. (For background on functional programming, see e.g. [Bir98].)

Although CSP can be combined with an imperative language, as was done in the case of Occam [Inm88, JG88], a functional language is simpler and it corresponds to the way CSP scripts are usually written, as sequences of process definitions (see Examples 2.1.1–2.1.3).

The functional language and the whole resulting language are typed. This is very important for our work: the data types in which a process is data independent will be represented by type variables, and the fact that the language is typed will then enable us to see where exactly values of those types are used, and to control the range of operations and predicates that can be applied to them.

As we have already said, our language is for simplicity and clarity basically a smaller version of CSP$_M$. In place of the functional language, we use a typed λ-calculus [Mit90, GLT89], which in addition to function space types has:

Sum types. The sum of any finite number of types, thought of as their disjoint union, can be formed.

Types of all events under consideration, in CSP called *alphabets*, will be sum types: for any channel under consideration, one component of the sum will be the type of all values that can be communicated along the channel, and the tag of the component (which ensures disjointness from other components) will be the name of the channel.

Product types. For any finite number of types, their cartesian product can also be formed.

Sometimes, values communicated along a channel will in fact be tuples of values, so that the type associated with the channel will be a product type.

Inductive types. In our typed λ-calculus, these are types that are defined recursively and such that the type being defined can appear in the body of the recursive definition only within sum and product types.

For example, types of lists or trees can be defined as inductive types (see Section 2.4).

A respect in which our language is significantly simpler than CSP$_M$ is that it does not have a general recursion construct, but only allows recursions over elements of the inductive types (see under 'Inductive type elimination' in Section 2.5.1) and recursive definitions of processes (see under 'Recursion' in Section 2.5.2). Consequently, it will be impossible to express a nonterminating computation in our language. More precisely, we shall have the strong normalisation property (see e.g. [GLT89]), namely that there are no infinite sequences of consecutive reductions of λ-calculus constructs. This will in particular be a considerable simplifying factor for the symbolic operational semantics (see Chapter 3).

The λ-calculus part of our language, although smaller than the functional language in CSP$_M$, is still quite expressive. Consequently, there seemed to be little point in trying to present the work in the thesis as explicitly independent of it, which could have been done by formulating assumptions that a functional language part has to satisfy for the work to be applicable. On the other hand, extending the thesis to other kinds of languages, in particular integrations of CSP with imperative languages, is a topic for future research (see Section 7.2.3).

The integration of CSP and the λ-calculus is done so that the λ-calculus in a sense "swallows up" the CSP part: some types in the resulting language will be types of processes, and the CSP operators will be used to form terms of those types. In other words, our whole language will be an applied typed λ-calculus, which in addition to types found in functional languages has process types.

A practical advantage of such an integration, which is also the case with CSP$_M$, is that we can form and manipulate networks of processes by working with, for example, lists or trees of processes.

2.3 Distinguishing features

The following are some features that distinguish our language and refinement as the notion of correctness from other languages and frameworks for reasoning about concurrent systems. A part of the contribution of the thesis is to have studied data independence and obtained results about it in the presence of these features.

Refinement and normalisation. As we have discussed in Section 2.1.3, both specifications (i.e. correctness conditions) and implementations are processes, and a specification is satisfied by an implementation if it is refined by it. This is adequate in CSP, as well as in CSP$_M$ and in our language, and means that no extra work is required to accommodate methods such as compositional or step-wise verification.

The fact that specifications can be arbitrary processes means that, for a state of an implementation, there may be more than one corresponding state of a specification. Automatic refinement checking (see Section 2.1.4) consequently involves explicit or implicit *normalising* (or *determinising*) of specifications, which ensures that any trace can be executed in at most one way. Essentially the same was needed in proofs of some of our results, which caused us to develop normalisation of symbolic transition systems (see Sections 3.6, 3.7 and 3.8).

Wealth of operators. CSP has a wide range of operators, which to a large extent is why refinement works as the notion of correctness. Indeed some of the operators (such as nondeterministic choice) are more likely to be used in specifications, and some others (like hiding) are more often used in implementations.

The wealth of operators has in particular required aspects of the symbolic operational semantics to be more involved than for other languages (see Chapter 3).

Three denotational models. CSP has for a long time been distinguished by the fact that the most of its theory is based on denotational semantics with refinement being the primary ordering on denotational models (see Sections 2.1.2, 2.1.3 and 2.4).

In this thesis, we are equally concerned with each of its three main denotational models (finite-traces, stable-failures and failures/divergences), and we often have to pass to and fro between denotational and symbolic operational semantics.

Channels are not directed. As we have pointed out in Section 2.1.1, inputs in CSP are formally just abbreviations of choices of events (one for each value that can be input), and outputs are formally the same as ordinary events. In particular, a process can use the same channel sometimes for input and at other times for output. Also, a specification and an implementation can, even in corresponding places, use the same channel for two different purposes.

The latter feature is sometimes quite convenient. For example, by means of nondeterministic choices over types T_1 and T_2 of some channels c_1 and c_2, the following specification requires only that events along c_1 alternate with those along c_2:

$$\mathit{Spec} = \sqcap_{x:T_1} c_1.x \longrightarrow \sqcap_{y:T_2} c_2.y \longrightarrow \mathit{Spec}$$

It is refined by each of:

$$\mathit{Impl} = c_1?x \longrightarrow c_2!x \longrightarrow \mathit{Impl}$$
$$\mathit{Impl}' = c_1?x \longrightarrow c_2?y \longrightarrow \mathit{Impl}'$$

Nondirectedness of channels also means that synchronisation between two processes executing in parallel (see Section 2.1.1) is not restricted to one of the processes performing an output that matches an input of the other. Thus synchronisation between two inputs is allowed, turning them into a common input of the two processes, and also between two outputs, which is of course possible only when they are outputs of the same value. The latter has required special attention within our work, because it provides a way of implicitly introducing equality tests between values from data independent types (see the statement of the condition ($\mathbf{NoEqT}_X$) in Section 3.5.1).

Many-way synchronisation. A related feature is many-way synchronisation: in a parallel composition of a number of processes, there may be events whose performance requires participation of more than two processes. This is sometimes useful in practice, especially in specifications: a specification that is a conjunction of a number of requirements can be written as a parallel composition of one processes for each requirement.

Higher types. Our language is obtained by integrating CSP and a typed λ-calculus in such a way that the whole language is an applied λ-calculus having types of processes (see Section 2.2). This means that 'higher types' are present, such as those of functions from processes to processes, or even of functions from functions on processes to processes. Presence of types of lists or trees of processes means that networks of processes can be formed and manipulated within the language.

The next section will show how each type can be modelled by a partial order. In Chapter 4, we go further and show how types can be modelled by relations, giving us a way of reasoning about parametric polymorphism (in our case, free type variables) in the denotational semantics.

Higher types also affect symbolic operational semantics: it is not only the case that λ-calculus constructs such as functional abstraction or decomposition of an ordered pair can be present in data components of CSP constructs (such as output), but also CSP constructs can be present within λ-calculus constructs (for example in a function that takes a list of processes and forms their parallel composition). Thus, in order to give symbolic operational semantics to processes, rules for reducing the λ-calculus part of the language need to be carefully combined with rules for generating symbolic transitions of the CSP part (see Chapter 3).

2.4 The types

In this section, we shall present the types of our language, and formally define their meanings by a denotational semantics.

The denotational semantics will interpret the types by partial orders. We need partial orders rather than simply sets to be able to interpret recursions in the denotational semantics of the terms of our language (see under 'Recursion' in Section 3.2).

The process types will thus be interpreted by CSP models with orderings which facilitate straightforward interpretation of recursions, namely as least fixed points of continuous functions on complete partial orders. Although those orderings will not be the same as the refinement ordering (see Sections 2.1.3 and 2.6), they will be very closely related to it (see under 'Process types' below).

In any case, the fact that the types will be interpreted by partial orders rather than simply sets will be important only in those preparatory or technical parts of the thesis in which it is significant how exactly recursions are interpreted. In particular, the refinement ordering should still be thought of as the primary ordering on processes, and it is the refinement ordering that Chapter 5 will be concerned with.

Partial orders. A partial order is a set $\mathcal{V}$ together with a relation $\leq$ on $\mathcal{V}$ which is

- reflexive: $\forall v \in \mathcal{V}. v \leq v$,
- anti-symmetric: $\forall v, w \in \mathcal{V}. (v \leq w \wedge w \leq v) \Rightarrow v = w$, and

- transitive: $\forall u, v, w \in \mathcal{V}.(u \leq v \wedge v \leq w) \Rightarrow u \leq w$.

(For an introduction to partial orders, see e.g. [DP90]. For a survey article on domain theory, which is the study of partial orders for denotational semantics, see e.g. [AJ94].) ■

Notation for denotational semantics. If T is a type, we shall write

$$[\![T]\!]^M_\delta$$

for the partial order which is the interpretation of T with respect to M and δ, where:

- M is one of Tr, Fa, or FD. It specifies one of the three denotational models of CSP: finite-traces, stable-failures, or failures/divergences (see Section 2.1.2). M needs to be included because T may be a process type or involve a process type.
- δ assigns a partial order to each type variable, subject to the restrictions stated under 'Type variables' below.

■

Some notation for assignments to type variables. For any mutually distinct type variables $X_1, \ldots, X_k$, and for any natural numbers $n_1, \ldots, n_k$ such that $n_i > 0$ whenever $X_i \in \mathit{NEFTypeVars}$ (see under 'Type variables' below), we define the following assignment of partial orders to type variables:

$$[X_1 \mapsto n_1, \ldots, X_k \mapsto n_k][\![Y]\!] = \begin{cases} \{0, \ldots, n_i - 1\} \textit{ with the flat ordering,} \\ \quad \textit{if } Y = X_i \textit{ for some } i \in \{1, \ldots, k\} \\ \{0\} \textit{ with the flat ordering,} \\ \quad \textit{otherwise} \end{cases}$$

where the flat ordering is $v \leq w \Leftrightarrow v = w$.

More generally, for any mutually distinct type variables $X_1, \ldots, X_k$, and for any cardinals $\kappa_1, \ldots, \kappa_k$ such that $\kappa_i > 0$ whenever $X_i \in \mathit{NEFTypeVars}$ and such that κ_i is a natural number whenever $X_i \in \mathit{FinTypeVars}$ (see under 'Type variables' below), we define the following assignment of partial orders to type variables:

$$[X_1 \mapsto \kappa_1, \ldots, X_k \mapsto \kappa_k][\![Y]\!] = \begin{cases} \{\alpha \mid \alpha \textit{ is an ordinal } \wedge \alpha < \kappa_i\} \textit{ with the flat ordering,} \\ \quad \textit{if } Y = X_i \textit{ for some } i \in \{1, \ldots, k\} \\ \{0\} \textit{ with the flat ordering,} \\ \quad \textit{otherwise} \end{cases}$$

(For ordinal and cardinal numbers, see e.g. [End77].) ■

The types are formed by the following constructions:

Type variables. We assume there is a set

$$\mathit{AllTypeVars}$$

of all type variables, and that there are subsets of it

$$\mathit{NEFTypeVars}, \mathit{FinTypeVars}$$

such that each of

$$\begin{array}{l}\mathit{NEFTypeVars} \cap \mathit{FinTypeVars},\\ \mathit{NEFTypeVars} \setminus \mathit{FinTypeVars},\\ \mathit{FinTypeVars} \setminus \mathit{NEFTypeVars},\\ \mathit{AllTypeVars} \setminus (\mathit{NEFTypeVars} \cup \mathit{FinTypeVars})\end{array}$$

is countably infinite.

In theoretical developments, we shall try to use letters

$$X, Y, Z$$

and their variations for type variables. More precisely, X, Y, Z and their variations are to be thought of as being able to range over type variables rather than actually being type variables.

A type variable X can be assigned any partial order by any δ (see under 'Notation for denotational semantics' above), except that:

- if $X \in \mathit{NEFTypeVars}$, then any $\delta[\![X]\!]$ must be a nonempty set with the flat ordering, i.e. $v \leq w \Leftrightarrow v = w$;
- if $X \in \mathit{FinTypeVars}$, then any $\delta[\![X]\!]$ must be a finite partial order.

Also, a type variable X can be substituted by any type T, except that:

- if $X \in \mathit{NEFTypeVars}$, then any T to be substituted for X must satisfy T^{nef} (see under 'Conditions on types' below);
- if $X \in \mathit{FinTypeVars}$, then any T to be substituted for X must satisfy T^{fin} (see under 'Conditions on types' below).

Type variables will play a very important role in this thesis, because data independent types will be type variables (see Section 2.7).

The interpretation of a type variable is simply the partial order that δ assigns to it:

$$[\![X]\!]^M_\delta = \delta[\![X]\!]$$

Sum types. Given a number of types $T_1, \ldots, T_n$, we write

$$id_1.T_1 + \cdots + id_n.T_n$$

for their sum, where $id_1, \ldots, id_n$ are some mutually distinct identifiers.

Sum types are thought of as disjoint unions. When $n = 0$, the sum type is written as 0 and thought of as having no elements. The main reason why we use explicit identifiers to separate the component types is that we shall be using certain sum types as alphabets:

the identifiers will be names of channels, and for each i the type of all values that can be communicated along the channel id_i will be T_i (see under 'Process types' below).

Although the notation may be suggesting otherwise, the order of the components is significant. In other words, two sum types which differ only in the order of their components are to be considered different, at least in formal developments.

The partial order interpreting a sum type is thus the disjoint union of the partial orders interpreting the component types, with the identifiers used as labels that ensure disjointness:

$$[\![id_1.T_1 + \cdots + id_n.T_n]\!]^M_\delta = \{id_1\} \times [\![T_1]\!]^M_\delta \cup \ldots \cup \{id_n\} \times [\![T_n]\!]^M_\delta$$

$$(id_i, v) \leq (id_j, w) \;\Leftrightarrow\; i = j \wedge v \leq w$$

Product types. Given a number of types T_1, ..., T_n, we write

$$T_1 \times \cdots \times T_n$$

for their product.

Product types are thought of as cartesian products of the component types. We allow $n = 0$ here also, in which case the product type is written as 1 and thought of as consisting of only one element.

As with sum types, the order of the components is significant.

The partial order interpreting a product type is indeed the cartesian product of the partial orders interpreting the component types:

$$[\![T_1 \times \cdots \times T_n]\!]^M_\delta = [\![T_1]\!]^M_\delta \times \cdots \times [\![T_n]\!]^M_\delta$$

$$(v_1, \ldots, v_n) \leq (w_1, \ldots, w_n) \;\Leftrightarrow\; \forall i \in \{1, \ldots, n\}. v_i \leq w_i$$

Inductive types. If T is a type, if X is a type variable which is neither in *NEFTypeVars* nor in *FinTypeVars*, and if any free occurence of X in T is only within sums, products and μs (in the sense that those are the only type constructs that can appear between the free occurences of X and the top level of T), then we can form an inductive type

$$\mu\, X.T$$

in which the free occurences of X in T become bound.

The condition that any free occurence of X in T is only within sums, products and μs, is abbreviated by writing $T^{(+,\times,\mu)X}$ (see under 'Conditions on types' below).

The type $\mu\, X.T$ can be thought of as the minimal type T' which satisfies the equation $T' = T[T'/X]$, where $T[T'/X]$ is T with T' substituted for each free occurence of X. Inductive types, and more general recursively defined types, are common in functional programming, primarily as types of various kinds of lists or trees. (See Example 2.4.1.)

In the denotational semantics, the partial order interpreting $\mu X.T$ is the least fixed point of the operator on partial orders that corresponds to T, where the ordering on partial orders is

$$(\mathcal{V}, \leq) \trianglelefteq (\mathcal{V}', \leq') \;\Leftrightarrow\; \mathcal{V} \subseteq \mathcal{V}' \wedge \;\leq = \leq' \restriction ((\mathcal{V} \times \mathcal{V}') \cup (\mathcal{V}' \times \mathcal{V}))$$

If we are interpreting $\mu X.T$ with respect to M and δ, the operator is

$$\mathcal{F}(\mathcal{V}, \leq) = [\![T]\!]^{M}_{\delta[(\mathcal{V},\leq)/X]}$$

where $\delta[(\mathcal{V}, \leq)/X]$ is δ updated so that the partial order $(\mathcal{V}, \leq)$ is assigned to X.

It is straightforward to prove that $\trianglelefteq$ is a complete ordering, and using the assumption that T satisfies $T^{(+,\times,\mu)X}$, it is straightforward to prove that $\mathcal{F}$ is continuous with respect to $\trianglelefteq$. Hence the least fixed point of $\mathcal{F}$ can be obtained as the least upper bound of the chain of all partial orders which are produced by iterating $\mathcal{F}$ finitely many times from the least partial order:

$$\begin{array}{rcl}(\mathcal{V}_0, \leq_0) & = & (\{\}, \{\}) \\ (\mathcal{V}_{k+1}, \leq_{k+1}) & = & \mathcal{F}(\mathcal{V}_k, \leq_k)\end{array}$$

$$[\![\mu X.T]\!]^{M}_{\delta} = \bigcup_{k \in \mathbb{N}} \mathcal{V}_k$$

$$v \leq w \;\Leftrightarrow\; v \leq_{k(v,w)} w$$

where $k(v, w)$ is any natural number such that $v, w \in \mathcal{V}_{k(v,w)}$.

(For least fixed points, completeness and continuity, see e.g. [DP90].)

Thus any element of any interpretation of $\mu X.T$ is obtained by finitely many iterations. For example, the types of lists and trees in Example 2.4.1 consist of only finite lists and trees. This is in line with the fact that our language does not have a general recursion construct (see the remarks after 'Inductive types' in Section 2.2), which means that for example infinite lists or trees are not expressible in it.

Function types. For any two types T_1 and T_2, we can form the function type

$$T_1 \to T_2$$

A function type $T_1 \to T_2$ is thought of as consisting of all functions from T_1 into T_2.

The interpretation of a function type is the set of all continuous functions with the pointwise ordering:

$$[\![T_1 \to T_2]\!]^{M}_{\delta} = \{f : [\![T_1]\!]^{M}_{\delta} \to [\![T_2]\!]^{M}_{\delta} \mid f \;\textit{ is continuous}\}$$

$$f \leq g \;\Leftrightarrow\; \forall\, v \in [\![T_1]\!]^{M}_{\delta}.f(v) \leq g(v)$$

where:

Least upper bounds. An element v of a partial order $\mathcal{V}$ is said to be the least upper bound of a subset $\mathcal{W} \subseteq \mathcal{V}$ if

- $\forall\, w \in \mathcal{W}. w \leq v$, and
- $\forall\, v' \in \mathcal{V}. (\forall\, w \in \mathcal{W}. w \leq v') \Rightarrow v \leq v'$.

In that case we write $v = \bigsqcup \mathcal{W}$.

Directed sets. A subset $\mathcal{D}$ of a partial order $\mathcal{V}$ is said to be directed if

- $\mathcal{D} \neq \{\}$, and
- $\forall\, v, w \in \mathcal{D}. \exists\, u \in \mathcal{D}. v \leq u \wedge w \leq u$.

Continuous functions. A function f from a partial order $\mathcal{V}_1$ into a partial order $\mathcal{V}_2$ is said to be continuous if for any directed subset $\mathcal{D}_1$ of $\mathcal{V}_1$ which has a least upper bound, the subset

$$\mathcal{D}_2 = \{f(v) \mid v \in \mathcal{D}_1\}$$

of $\mathcal{V}_2$ is also directed and has a least upper bound, and

$$f(\bigsqcup \mathcal{D}_1) = \bigsqcup \mathcal{D}_2$$

Process types. If T is a type satisfying T^{alph} (see under 'Conditions on types' below), which means that it is of the form

$$\sum_{i=1}^{n} id_i. \prod_{j=1}^{m_i} T_{i,j}^{comm}$$

where $n \geq 1$, we say that ϕ is a *ways specifier* with respect to T if

$$\phi(id_i, j) \subseteq \{!, ?, \$\}$$

for any $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m_i\}$. We shall use letters

$$\phi, \psi, \chi$$

and their variations as ways specifiers.

For any such T and ϕ, we write

$$Proc_{T,\phi}$$

for the type of all processes with alphabet T which can use the components of the channels in T only in the ways specified by ϕ.

More precisely, $Proc_{T,\phi}$ is the type of all processes whose visible events are of type T and which can, for any i and j, use the jth component of the channel id_i only in the ways which are in the set $\phi(id_i, j)$, where ! stands for outputs, ? stands for inputs, and \$ stands for nondeterministic selections. (See Section 2.1.1 and under 'Channels are not directed' in Section 2.3. For nondeterministic selections, which we have not yet met, see under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2.)

Visible events are simply values from the alphabet type T: we call them 'visible events' only to aid the intuition. Thus, since T satisfies T^{alph}, any visible event a is of the form

(id_i, v), where v is of the form $(v_1, \ldots, v_{m_i})$. We refer to id_i as the channel, and to v as the value communicated along id_i.

At this stage, it would be too complex to formulate useful criteria for when an element of a CSP model cannot be the interpretation of a process which observes a particular ways specifier. Hence the interpretations of the types $Proc_{T,\phi}$ will not depend on ϕs, but will contain enough elements for all ϕs. Some consequences of ways specifiers on the denotational semantics will be established as a part of the developments in Chapter 4. For some consequences on the symbolic operational semantics, which are much more straightforward, see under 'Visible' in Section 3.2.2 and under 'Normalised symbolic transitions' in Section 3.6.1.

The partial order interpreting a type $Proc_{T,\phi}$ with respect to some M and δ is the denotational model of CSP of kind M (see Section 2.1.2) and alphabet $[\![T]\!]_\delta^M$, with an ordering which facilitates recursions to be interpreted by least fixed points of continuous functions on complete partial orders (see under 'Recursion' in Section 2.5.2).

Thus, if $M = \mathrm{FD}$,

$$[\![Proc_{T,\phi}]\!]_\delta^M$$

is the set of all pairs $(Fails_\perp, Divs)$ such that:

$$\begin{aligned} Fails_\perp &\subseteq ([\![T]\!]_\delta^M)^* \times \mathcal{P}([\![T]\!]_\delta^M) \\ Divs &\subseteq ([\![T]\!]_\delta^M)^* \end{aligned}$$

and such that the following conditions are satisfied:

F1. $Trs_\perp = \{t \mid (t, A) \in Fails_\perp\}$ is nonempty and prefix closed (i.e. $t\hat{\ }u \in Trs_\perp \Rightarrow t \in Trs_\perp$).

F2. $(t, A) \in Fails_\perp \wedge B \subseteq A \Rightarrow (t, B) \in Fails_\perp$.

F3. $(t, A) \in Fails_\perp \wedge \forall\, b \in B. t\hat{\ }\langle b\rangle \notin Trs_\perp \Rightarrow (t, A \cup B) \in Fails_\perp$.

D1. $Divs$ is closed under extensions, i.e. $t \in Divs \Rightarrow t\hat{\ }u \in Divs$.

D2. $t \in Divs \Rightarrow (t, A) \in Fails_\perp$.

The ordering is known as the alternative ordering on the failures/divergences model. It is stronger than the refinement ordering (see Section 2.1.3), and unlike the refinement ordering, it always makes $[\![Proc_{T,\phi}]\!]_\delta^M$ into a complete partial order. (For more on this ordering, in particular on its close relationship with the refinement ordering, see [Ros88], [Ros98a, Section 9.2].)

$$\begin{aligned} &(Fails_\perp, Divs) \leq (Fails'_\perp, Divs') \Leftrightarrow \\ &\quad Divs \supseteq Divs' \wedge \\ &\quad \forall\, t.((t, \{\}) \in Fails_\perp \wedge t \notin Divs) \Rightarrow \\ &\quad (\forall\, A.(t, A) \in Fails_\perp \Leftrightarrow (t, A) \in Fails'_\perp) \wedge \\ &\quad (\forall\, a.(t\hat{\ }\langle a\rangle, \{\}) \in Fails_\perp \Leftrightarrow (t\hat{\ }\langle a\rangle, \{\}) \in Fails'_\perp) \end{aligned}$$

For definitions in the two simpler cases $M = \mathrm{Tr}$ and $M = \mathrm{Fa}$, see [Ros98a, Chapter 8]. The orderings are the reverse refinement orderings, i.e. $\subseteq$ and pointwise $\subseteq$, and they are also always complete.

In [Ros98a, Chapter 8], it is also explained why the least fixed points (rather than any other fixed points) with respect to the three orderings are the right interpretations of recursions. That is confirmed formally by the congruence theorems: see [Ros98a, Section 9.4] and Section 3.4.2 of this thesis.

Conditions on types. The following are a number of useful conditions on types, which we write as superscripts:

$T^{+,\times}$, $T^{\mathit{AllTypeVars},+,\times}$, $T^{\mathit{AllTypeVars},+,\times,\mu}$, $T^{\mathit{AllTypeVars},+,\times,\rightarrow}$, $T^{\mathit{AllTypeVars},+,\times,\mu,\rightarrow}$ are satisfied if T is built only from the type constructs listed, where *AllTypeVars* means that arbitrary type variables can be used.

$T^{nef'}$ is defined by induction on the structure of types:

$$\begin{array}{rcl}
X^{nef'} & \Leftrightarrow & X \in \mathit{NEFTypeVars} \\
(id_1.U_1 + \cdots + id_n.U_n)^{nef'} & \Leftrightarrow & (\forall\, i.U_i^{\mathit{AllTypeVars},+,\times,\mu,\rightarrow}) \wedge (\exists\, i.U_i^{nef'}) \\
(U_1 \times \cdots \times U_n)^{nef'} & \Leftrightarrow & \forall\, i.U_i^{nef'} \\
(\mu\, Y.U)^{nef'} & \Leftrightarrow & U^{nef'} \\
(U_1 \rightarrow U_2)^{nef'} & \Leftrightarrow & U_2^{nef'} \\
(\mathit{Proc}_{U,\phi})^{nef'} & \Leftrightarrow & \mathit{False}
\end{array}$$

It is easy to see that for any T which satisfies $T^{nef'}$, for any M, and for any δ such that the ordering on $\delta[\![X]\!]$ is flat for each X which occurs free in T, we have that $[\![T]\!]_\delta^M$ is a nonempty set with the flat ordering.

T^{nef} is satisfied if $T^{nef'}$ and $X \in \mathit{NEFTypeVars}$ for each X which occurs free in T.

Thus for any T which satisfies T^{nef} and for any M and δ, we have that $[\![T]\!]_\delta^M$ is a nonempty set with the flat ordering. (Recall the restrictions stated under 'Type variables' above.)

T^{comm} is satisfied if T^{nef} and $T^{\mathit{AllTypeVars},+,\times,\mu}$, i.e. if T^{nef} and T does not involve function types or process types.

We use the types satisfying this condition as the types whose elements can be communicated or tested for equality: see the definition of T^{alph} below, under 'Equality testing' in Section 2.5.1, and under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2.

T^{alph} is satisfied if T is a nonempty sum of products in which each component type satisfies *comm*, i.e. if it is of the form

$$\sum_{i=1}^{n} id_i.\prod_{j=1}^{m_i} T_{i,j}^{comm}$$

where $n \geq 1$.

We use the types satisfying this condition as alphabets, where an alphabet is the type of all visible events a process is allowed to perform. Each identifier id_i is then thought of as a channel for communicating values of type $\prod_{j=1}^{m_i} T_{i,j}$. (See under 'Process types' above.)

T^{fin} is satisfied if $T^{AllTypeVars,+,\times,\to}$ and $X \in \mathit{FinTypeVars}$ for each X which occurs free in T, i.e. if T is built only from the type variables in $\mathit{FinTypeVars}$, sums, products and function types.

It is easy to see that for any T which satisfies T^{fin} and for any M and δ, we have that $[\![T]\!]^M_\delta$ is finite. (Recall the restrictions stated under 'Type variables' above.)

$T^{(+,\times,\mu)X}$ is satisfied if any free occurence of X in T is only within sums, products and μs.

■

The type constructs that we have can be used to define many commonly used types:

Example 2.4.1 The type of booleans can be defined as a sum of two singleton types:

$$\mathit{Bool} = \mathit{true}.1 + \mathit{false}.1$$

The type of all finite lists of elements of a type V can be defined as an inductive type:

$$\mathit{List}(V) = \mu X.(\mathit{nil}.1 + \mathit{cons}.(V \times X))$$

where X is a type variable which is neither in $\mathit{NEFTypeVars}$ nor in $\mathit{FinTypeVars}$, and which does not occur free in V. *nil* and *cons* are just identifiers which separate the two components of the sum type $\mathit{nil}.1 + \mathit{cons}.(V \times X)$, but they can also be thought of as constructors for the empty list and for making a new list by pairing an element of V and a list.

The type of all natural numbers can be defined by simplifying the same idea:

$$\mathit{Num} = \mu Y.(\mathit{zero}.1 + \mathit{succ}.Y)$$

Another example of an inductive type is the type of all finite binary trees whose leaves are labelled by elements of V:

$$\mathit{LBTree}(V) = \mu X.(\mathit{leaf}.V + \mathit{node}.(X \times X))$$

The type

$$T = c.(\mathit{Bool} \times \mathit{Num} \times \mathit{Num}) + d.\mathit{List}(\mathit{Num})$$

satisfies T^{alph}, and ϕ defined by

$$\begin{aligned} \phi(c,1) &= \{?\} \\ \phi(c,2) &= \{\$\} \\ \phi(c,3) &= \{\$\} \\ \phi(d,1) &= \{?,!\} \end{aligned}$$

is a ways specifier with respect to T, so we can form the type

$$\mathit{Proc}_{T,\phi}$$

of all processes that can use the first component of the channel c for inputting booleans and the second and third components for nondeterministically selecting natural numbers, and can use the only component of the channel d for inputting or outputting lists of natural numbers.

■

Let us record here the fact that ways specifiers were not taken into account in the denotational semantics of the types.

Definition 2.4.1 For two types T and T', let us write

$$T \sim T'$$

if and only if they differ at most in ways specifiers. ■

Proposition 2.4.1 *If $T \sim T'$, then $[\![T]\!]^M_\delta = [\![T']\!]^M_\delta$ for any M and δ.*

Proof. By induction on the structure of T and T'. ■

Junk elements. In general theoretical CSP, all elements of finite-traces models and stable-failures models, and all elements of failures/divergences models with finite alphabets, are expressible as interpretations of processes [Ros98a, Section 9.3].

However, our language will have free type variables, a concrete data part consisting of computable λ-calculus constructs (see Section 2.5.1), and a finitary and computable syntax for sets (see Section 2.5.2), and so there will be many elements of the interpretations of process types which are not interpretations of any terms. Another factor contributing to this presence of junk elements is that ways specifiers were not taken into account when interpreting the process types.

The interpretations of function types will also contain many junk elements. For example, for any T_1 which has no free type variables and which does not involve process types (such as $T_1 = \mathit{Num}$), any interpretation of T_1 has the flat ordering. Consequently, for any T_2, any interpretation of $T_1 \to T_2$ consists of all functions.

The presence of junk elements will not cause problems in this thesis, and so we leave it for future research. ■

Some conventions. When T does not involve process types, its interpretations do not depend on which CSP model M we are working with, so in such cases we may write $[\![T]\!]_\delta$ instead of $[\![T]\!]^M_\delta$.

For any type T, let $\mathit{Free}(T)$ be the set of all type variables that occur free in T. When $\mathit{Free}(T) = \{\}$, the interpretations of T do not depend on δ, so we may then write only $[\![T]\!]^M$, or $[\![T]\!]$ if M also is unnecessary. ■

2.5 The terms

Terms are pieces of syntax, and they can be of the various types we defined in the previous section. For example, processes will be terms of process types. This section is devoted to presenting the terms and their denotational semantics.

Type contexts, type judgements and typing rules. Whenever we consider a term, the types of the term variables that occur free in it will be provided by a *type context*. We shall write type contexts as

$$\{x_1 : T_1, \ldots, x_k : T_k\}$$

where the x_i are some mutually distinct term variables, and the T_i are any types. (An alternative to using type contexts is to use typed term variables.)

A *type judgement*

$$\Gamma \vdash r : T$$

will mean that r is a term of type T in the type context Γ.

Following a standard way of presenting typed languages (see e.g. [Mit90]), the terms of our language will be defined as those that appear in true type judgements. A type judgement will be true if it is derivable from a number of *typing rules*: each term construct has its typing rule, specifying type judgements that the subterms have to satisfy for the constructed term to be well-typed. In this way, the syntax and the typing of terms are given at the same time by the typing rules.

Some typing rules will have side conditions, which have to be satisfied for the rules to be applicable.

We shall often abbreviate the phrase 'a term r such that $\Gamma \vdash r : T$ is derivable' by 'a term $\Gamma \vdash r : T$'. ■

Notation for denotational semantics. For any term $\Gamma \vdash r : T$, we shall write

$$[\![\Gamma \vdash r : T]\!]^M_{\delta,\eta}$$

for its interpretation with respect to M, δ and η, where:

- M, as in the previous section, specifies one of the three models of CSP: Tr, Fa, or FD.
- δ, as in the previous section, assigns a partial order to each type variable, subject to the restrictions stated under 'Type variables' in the previous section.
- η assigns a value to each term variable in Γ with respect to M and δ. More precisely, $\eta[\![x]\!] \in [\![T]\!]^M_\delta$ for each $(x : T) \in \Gamma$.

The denotational semantics will be defined by induction on the typing rules. For a sketch of how well-definedness is proved, which involves proving that any $[\![\Gamma \vdash r : T]\!]^M_{\delta,\eta}$ is an element of the interpretation $[\![T]\!]^M_\delta$ of the type T, see Section 2.5.3. ■

Some conventions. When r has no free term variables, we may omit η, and when in addition there are no free type variables, we may also omit δ.

When none of the relevant types involve process types, we may omit M. ■

2.5.1 The λ-calculus constructs

The λ-calculus constructs we have are term variables, fundamental term constructs [Mit90, GLT89, Loa97] for each of the type constructs of our language (see Section 2.4) except process types, and equality tests:

Term variables. In theoretical developments, we shall try to use letters x, y, z and their variations for term variables. More precisely, as with type variables, x, y, z and their variations are to be thought of as being able to range over term variables, rather than actually being term variables.

The typing rule is

$$\frac{}{\{x_1 : T_1, \ldots, x_k : T_k\} \vdash x_i : T_i}$$

which says that x_i is of the type T_i given to it by the type context.

The interpretation of a term variable is the value assigned to it by η:

$$[\![\{x_1 : T_1, \ldots, x_k : T_k\} \vdash x_i : T_i]\!]^M_{\delta,\eta} = \eta[\![x_i]\!]$$

Sum type introduction. Given a term whose type is a component type of a sum, we can construct a term of the sum type:

$$\frac{\Gamma \vdash r : T_i}{\Gamma \vdash id_i^{id_1.T_1+\cdots+id_n.T_n}.r : id_1.T_1 + \cdots + id_n.T_n}$$

We have included the sum type in the superscript because we wish the type of any term to be determined as closely as possible by the term and the type context (see Section 2.5.3). When it is clear which sum type is to be used, we shall write just $id_i.r$.

$$\begin{aligned}&[\![\Gamma \vdash id_i^{id_1.T_1+\cdots+id_n.T_n}.r : id_1.T_1 + \cdots + id_n.T_n]\!]^M_{\delta,\eta} = \\ &\quad (id_i, [\![\Gamma \vdash r : T_i]\!]^M_{\delta,\eta})\end{aligned}$$

Sum type elimination. Given a term r of a sum type, the construct for 'taking r out of the sum type' involves a branching: for each component type T_i, there is a branch which gets selected when r is from T_i, and which consists of applying a given function s_i to r:

$$\frac{\begin{array}{c}\Gamma \vdash r : id_1.T_1 + \cdots + id_n.T_n \\ \Gamma \vdash s_1 : T_1 \to U \qquad \ldots \qquad \Gamma \vdash s_n : T_n \to U\end{array}}{\Gamma \vdash \mathit{case}\ r\ \mathit{of}\ s_1\ \mathit{or}\ \ldots\ \mathit{or}\ s_n : U}$$

For example, recalling that the type of booleans was defined in Example 2.4.1 as a sum type

$$\mathit{Bool} = \mathit{true}.1 + \mathit{false}.1$$

we can define *if-then-else* as

$$\mathit{if}\ r\ \mathit{then}\ r'\ \mathit{else}\ r'' = \mathit{case}\ r\ \mathit{of}\ (\lambda\, x' : 1.r')\ \mathit{or}\ (\lambda\, x'' : 1.r'')$$

where r is of type *Bool*, and r' and r'' are of any type U and do not contain free occurences of x' and x'' respectively. (For the meaning of λs, see under 'Functional abstraction' below.)

$$\begin{aligned}&[\![\Gamma \vdash \mathit{case}\ r\ \mathit{of}\ s_1\ \mathit{or}\ \dots\ \mathit{or}\ s_n : U]\!]^M_{\delta,\eta} = \\ &\quad [\![\Gamma \vdash s_i : T_i \to U]\!]^M_{\delta,\eta}(v), \\ &\quad \mathit{where}\ (id_i, v) = [\![\Gamma \vdash r : id_1.T_1 + \cdots + id_n.T_n]\!]^M_{\delta,\eta}\end{aligned}$$

Construction of tuples. Given a number of terms, they can be put together to form a term of the appropriate product type:

$$\frac{\Gamma \vdash r_1 : T_1 \qquad \dots \qquad \Gamma \vdash r_n : T_n}{\Gamma \vdash (r_1, \dots, r_n) : T_1 \times \cdots \times T_n}$$

$$\begin{aligned}&[\![\Gamma \vdash (r_1, \dots, r_n) : T_1 \times \cdots \times T_n]\!]^M_{\delta,\eta} = \\ &\quad ([\![\Gamma \vdash r_1 : T_1]\!]^M_{\delta,\eta}, \dots, [\![\Gamma \vdash r_n : T_n]\!]^M_{\delta,\eta})\end{aligned}$$

Selection from tuples. Given a term of a product type $T_1 \times \cdots \times T_n$, we can for any $i \in \{1, \dots, n\}$ select its ith component:

$$\frac{\Gamma \vdash r : T_1 \times \cdots \times T_n}{\Gamma \vdash \mathit{pr}_i\ r : T_i}$$

The denotational semantics is:

$$[\![\Gamma \vdash \mathit{pr}_i\ r : T_i]\!]^M_{\delta,\eta} = ([\![\Gamma \vdash r : T_1 \times \cdots \times T_n]\!]^M_{\delta,\eta})(i)$$

where $\cdot(i)$ selects the ith component of a tuple.

Inductive type introduction. If $\mu\, X.T$ is an inductive type and r a term of type $T[(\mu\, X.T)/X]$ (i.e. T with $\mu\, X.T$ substituted for every free occurence of X in T), we can formally make r into a term of type $\mu\, X.T$:

$$\frac{\Gamma \vdash r : T[(\mu\, X.T)/X]}{\Gamma \vdash \mathit{in}_{\mu\, X.T}\ r : \mu\, X.T}$$

For instance, recall from Example 2.4.1 that the type of all finite lists of elements of a type V can be defined as

$$\mathit{List}(V) = \mu\, X.(\mathit{nil}.1 + \mathit{cons}.(V \times X))$$

To obtain an empty list of this type we use the *nil* identifier followed by the $\mathit{in}_{\mathit{List}(V)}$ construct, and to obtain an operator for joining elements of V to lists we use *cons* followed by $\mathit{in}_{\mathit{List}(V)}$:

$$\begin{aligned}\mathit{Nil}_{\mathit{List}(V)} &= \mathit{in}_{\mathit{List}(V)}\ \mathit{nil}.() \\ \mathit{Cons}_{\mathit{List}(V)} &= \lambda\, x : V.\, \lambda\, y : \mathit{List}(V).\mathit{in}_{\mathit{List}(V)}\ \mathit{cons}.(x, y)\end{aligned}$$

(See under 'Functional abstraction' below for the meaning of λs.) Any finite list can then be formed by repeated application of the operator starting from the empty list:

$$[s_1, \ldots, s_k] = \mathit{Cons}_{\mathit{List}(V)}\, s_1\, (\mathit{Cons}_{\mathit{List}(V)}\, \ldots\, (\mathit{Cons}_{\mathit{List}(V)}\, s_k\, \mathit{Nil}_{\mathit{List}(V)}) \ldots)$$

The interpretation of $\Gamma \vdash \mathit{in}_{\mu X.T} r : \mu X.T$ is just the interpretation of $\Gamma \vdash r : T[(\mu X.T)/X]$, since the denotational semantics of types (see Section 2.4) ensures that the types $\mu X.T$ and $T[(\mu X.T)/X]$ have the same interpretations:

$$[\![\Gamma \vdash \mathit{in}_{\mu X.T}\, r : \mu X.T]\!]^M_{\delta,\eta} = [\![\Gamma \vdash r : T[(\mu X.T)/X]]\!]^M_{\delta,\eta}$$

Inductive type elimination. If $\mu X.T$ is an inductive type and U any type, a term s of type $T[U/X] \to U$ can be seen as providing a definition for computing elements of U by recursion on the structure of elements of $\mu X.T$. Given a term r of type $\mu X.T$, the following construct can be thought of as performing along r the recursion provided by s:

$$\frac{\Gamma \vdash r : \mu X.T \qquad \Gamma \vdash s : T[U/X] \to U}{\Gamma \vdash \mathit{fold}\; r\; s : U}$$

For example, consider again the inductive type $\mathit{List}(V)$. In its case

$$T = \mathit{nil}.1 + \mathit{cons}.(V \times X)$$

and so s is of type

$$(\mathit{nil}.1 + \mathit{cons}.(V \times U)) \to U$$

which can indeed be seen as providing a recursive definition on lists of elements of V: given the empty list, s maps it to some element of U, and given a pair consisting of the head of the list we are recursing on and the result in U corresponding to the rest of the list, s maps it to the result for the whole list.

More concretely, if

$$\begin{aligned} \mathit{Zero} &= \mathit{in}_{\mathit{Num}}\, \mathit{zero}.() \\ \mathit{Succ} &= \lambda x : \mathit{Num}.\mathit{in}_{\mathit{Num}}\, \mathit{succ}.x \end{aligned}$$

represent 0 and the successor function on the type Num of natural numbers (see Example 2.4.1), the following s provides a recursive definition of list length:

$$\begin{aligned} s \;=\; & \lambda x : \mathit{nil}.1 + \mathit{cons}.(V \times \mathit{Num}). \\ & \quad \mathit{case}\ x\ \mathit{of}\ (\lambda y : 1.\mathit{Zero})\ \mathit{or}\ (\lambda z : V \times \mathit{Num}.\mathit{Succ}(\mathit{pr}_2\, z)) \end{aligned}$$

(See below about λs and functional application.) Then, for any term r of type $\mathit{List}(V)$, $\mathit{fold}\; r\; s$ is the length of the finite list r represents.

Denotational semantics of the $\mathit{fold}\; r\; s$ construct is itself defined recursively, by informally speaking applying s to r in which any part r' that is in a place where X occurs free in T is replaced by $\mathit{fold}\; r'\; s$. We skip the details.

Intuitively, any element of $\mu\, X.T$ is finite, and so any recursion expressible using the *fold* construct always terminates. That is confirmed by the fact that any element any interpretation of $\mu\, X.T$ in the denotational semantics is obtained by finitely many iterations (see under 'Inductive types' in Section 2.4). It will also be confirmed by the fact that our language has the strong normalisation property (see e.g. [GLT89, Loa97]), namely that there are no infinite sequences of consecutive one-step reductions of λ-calculus constructs (see Table 3.1).

Functional abstraction. If r is a term and x a term variable, we can form

$$\frac{(\Gamma \setminus \{x\}) \cup \{x : T\} \vdash r : U}{\Gamma \vdash (\lambda\, x : T.r) : T \to U}$$

which is a term representing the function given by r. The result of the function is r with the argument substituted for all free occurences of x.

The type context $(\Gamma \vdash \{x\}) \cup \{x : T\}$ of r is Γ with any occurence of x (if any) substituted by $x : T$. The λ construct binds x, so that Γ can be used as the type context of $\lambda\, x : T.r$.

$$\begin{aligned}&[\![\Gamma \vdash (\lambda\, x : T.r) : T \to U]\!]^M_{\delta,\eta} = f,\\ &\textit{where } f(v) = [\![(\Gamma \setminus \{x\}) \cup \{x : T\} \vdash r : U]\!]^M_{\delta,\eta[v/x]}\end{aligned}$$

Functional application. If r is of a function type $T \to U$ and s is of type T, we can apply the function that r represents to s:

$$\frac{\Gamma \vdash r : T \to U \qquad \Gamma \vdash s : T}{\Gamma \vdash r\ s : U}$$

$$[\![\Gamma \vdash r\ s : U]\!]^M_{\delta,\eta} = [\![\Gamma \vdash r : T \to U]\!]^M_{\delta,\eta}([\![\Gamma \vdash s : T]\!]^M_{\delta,\eta})$$

Equality testing. Two terms r and s of any type T satisfying T^{comm} (see under 'Conditions on types' in Section 2.4) can be tested for equality

$$\frac{\Gamma \vdash r, s : T \qquad \Gamma \vdash u, v : U}{\Gamma \vdash ((r = s) \rightsquigarrow u, v) : U}\, T^{comm}$$

so that if their values are equal, the value of a term u is returned, and if they are not, the value of a term v is returned.

We have not presented equality testing as a predicate (i.e. a function that returns a boolean) because the form above is more convenient for formulating restrictions about equality tests on data independent types (see Section 3.5). The predicate version can be defined as

$$(r = s) \rightsquigarrow (\textit{true}.()), (\textit{false}.())$$

(see the definition of *Bool* in Example 2.4.1).

$$\begin{aligned}&[\![\Gamma \vdash ((r = s) \rightsquigarrow u, v) : U]\!]^M_{\delta,\eta} =\\ &\qquad\begin{cases}[\![\Gamma \vdash u : U]\!]^M_{\delta,\eta}, & \textit{if } [\![\Gamma \vdash r : T]\!]^M_{\delta,\eta} = [\![\Gamma \vdash s : T]\!]^M_{\delta,\eta}\\ [\![\Gamma \vdash v : U]\!]^M_{\delta,\eta}, & \textit{if } [\![\Gamma \vdash r : T]\!]^M_{\delta,\eta} \neq [\![\Gamma \vdash s : T]\!]^M_{\delta,\eta}\end{cases}\end{aligned}$$

More complex functions and syntactic sugar. As we have said, the constructs we have presented are the fundamental ones for the types we have. They can be used to define all the more complex functions found in common functional languages, except those that require general recursion (see the remarks after 'Inductive types' in Section 2.2, and at the end of 'Inductive types' in Section 2.4). We have already given some small examples, like the length function under 'Inductive type elimination' in this section.

It would be outside the scope of this thesis to give more extensive examples; interested readers should consult texts on functional programming, e.g. [Bir98, GLT89].

From now on, we shall feel free to use more complex functions or constants, as well as syntactic sugar, without having to define them, provided their meanings are clear from the context. ■

2.5.2 The CSP constructs

The previous section was devoted to the λ-calculus constructs in our language. In this section, we present the CSP constructs, used for building terms of process types.

The constructs and their denotational semantics are essentially those of standard CSP [Hoa85, Ros98a], although presented more formally to fit into the applied λ-calculus framework of our language (see Section 2.2), which is similar to the framework of CSP$_M$ [Sca92, Sca98, Ros98a].

It is good to keep in mind that the CSP constructs are not separated from the λ-calculus ones in the language: on one hand the λ-calculus constructs can be used to compute data within terms of process types (for example outputs), and on the other hand terms of process types can appear within λ-calculus constructs (for example as alternatives in an equality test).

We define denotational semantics of the constructs for the failures/divergences model of CSP (see Section 2.1.2 and under 'Process types' in Section 2.4); definitions for the two simpler models should be deducible.

Some conventions.

- We shall refer to 'terms of process types' simply as 'processes', and in theoretical developments try to use letters

  $$P, Q$$

  and their variations to denote them, rather than r, s.

  Similarly, we shall refer to 'term variables of process types' as 'process variables', and in theoretical developments try to use letters

  $$p, q$$

  and their variations for them, rather than x, y, z.

  We shall sometimes use the same notations for terms or term variables of types $U \to Proc_{T,\phi}$, which can be thought of as processes with arguments of type U (see under 'Recursion' below).

- As with any other term, the interpretations of a process $\Gamma \vdash P : Proc_{T,\phi}$ are written as $[\![\Gamma \vdash P : Proc_{T,\phi}]\!]^{M}_{\delta,\eta}$. Depending on the model M, we shall use the following conventions:

$$\begin{aligned}
traces(\Gamma \vdash P : Proc_{T,\phi})^{\mathrm{Tr}}_{\delta,\eta} &= [\![\Gamma \vdash P : Proc_{T,\phi}]\!]^{\mathrm{Tr}}_{\delta,\eta} \\
traces(\Gamma \vdash P : Proc_{T,\phi})^{\mathrm{Fa}}_{\delta,\eta} &= ([\![\Gamma \vdash P : Proc_{T,\phi}]\!]^{\mathrm{Fa}}_{\delta,\eta})(1) \\
failures(\Gamma \vdash P : Proc_{T,\phi})_{\delta,\eta} &= ([\![\Gamma \vdash P : Proc_{T,\phi}]\!]^{\mathrm{Fa}}_{\delta,\eta})(2) \\
failures_{\perp}(\Gamma \vdash P : Proc_{T,\phi})_{\delta,\eta} &= ([\![\Gamma \vdash P : Proc_{T,\phi}]\!]^{\mathrm{FD}}_{\delta,\eta})(1) \\
divergences(\Gamma \vdash P : Proc_{T,\phi})_{\delta,\eta} &= ([\![\Gamma \vdash P : Proc_{T,\phi}]\!]^{\mathrm{FD}}_{\delta,\eta})(2)
\end{aligned}$$

  where $\cdot(1)$ and $\cdot(2)$ select the first and second components of ordered pairs.

  We may omit M from $traces(\Gamma \vdash P : Proc_{T,\phi})^{M}_{\delta,\eta}$ when that causes no confusion.

  In the same way as for any term, we may omit η or both δ and η when they are unnecessary.

- When working with the failures/divergences model, by 'traces' we shall mean both ordinary traces and those introduced as extensions of divergences. (The difference between $failures(\Gamma \vdash P : Proc_{T,\phi})_{\delta,\eta}$ and $failures_{\perp}(\Gamma \vdash P : Proc_{T,\phi})_{\delta,\eta}$ was explained in Section 2.1.2.)

- We shall use pointwise liftings of set operations and set predicates with ways specifiers which are with respect to the same T (see under 'Process types' in Section 2.4). For example, we shall have:

$$\forall i,j.(\phi \cup \psi)(id_i,j) = \phi(id_i,j) \cup \psi(id_i,j)$$

$$\phi \subseteq \chi \Leftrightarrow \forall i,j.\phi(id_i,j) \subseteq \chi(id_i,j)$$

■

STOP. For any alphabet T, i.e. for any type T satisfying T^{alph} (see under 'Conditions on types' and Example 2.4.1 in Section 2.4), $STOP_T$ is a process that does nothing: it cannot perform any events, visible or invisible. It observes any ways specifier ϕ.

$$\frac{}{\Gamma \vdash STOP_T : Proc_{T,\phi}}$$

It has only the empty trace, after which it can refuse any set of visible events, and it has no divergences:

$$\begin{aligned}
failures_{\perp}(\Gamma \vdash STOP_T : Proc_{T,\phi})_{\delta,\eta} &= \{(\langle\rangle, A) \mid A \subseteq [\![T]\!]_{\delta}\} \\
divergences(\Gamma \vdash STOP_T : Proc_{T,\phi})_{\delta,\eta} &= \{\}
\end{aligned}$$

(M in $[\![T]\!]_{\delta}$ can be omitted by the conventions at the end of Section 2.4.)

Outputs, inputs and nondeterministic selections. A process can perform outputs, inputs and nondeterministic selections using a construct called 'prefixing'.

A prefix is of the form

$$id\ it_1\ \ldots\ it_m$$

where id is a channel and each it_j is an output, input or a nondeterministic selection. The type of id is a product of m components (see under 'Process types' in Section 2.4, and Example 2.4.1) to which the items $it_1, \ldots, it_m$ correspond. Given such a prefix and a process P, the prefixing construct produces a process

$$id\ it_1\ \ldots\ it_m \longrightarrow P$$

whose initial visible events are determined by the prefix and which subsequently behaves like P.

For example, let the alphabet be

$$T = c.(Bool \times Num \times Num) + d.List(Num)$$

(as in Example 2.4.1). Using $STOP_T$ and prefixing three times, we can form a process

$$c?x?y?y' \longrightarrow d?z \longrightarrow d!([y] +\!\!+\, z +\!\!+\, [y', y]) \longrightarrow STOP_T$$

which inputs a boolean x and two natural numbers y and y' along the channel c, inputs a list z of natural numbers along d, outputs the list $[y] +\!\!+\, z +\!\!+\, [y', y]$ (formed by concatenating lists $[y]$, z and $[y', y]$) along d, and then stops.

Another example with the same alphabet is

$$c?x?y?y' \longrightarrow c!(not\ x)!y'?y'' \longrightarrow d![y, y', y''] \longrightarrow STOP_T$$

in which the second prefix at the same time outputs $not\ x$ and y' and inputs y''. This may seem counter-intuitive, but recall that inputs and outputs in CSP in a sense have no direction: an output is a communication, and an input is a choice of communications (see Section 2.1.1).

A third example is

$$c?x?y?y' \longrightarrow d![y, y'] \longrightarrow c?x?y?y'' \longrightarrow d![y, y', y''] \longrightarrow STOP_T$$

Here x and y are used for two different inputs. The y in $[y, y']$ is from the first input, and the y in $[y, y', y'']$ is from the second.

These examples have not contained 'nondeterministic selections', which are abbreviations for nondeterministic choices performed before the prefix. For example,

$$c?x\$y?y' \longrightarrow d![y, y'] \longrightarrow STOP_T$$

is equivalent to

$$\sqcap_{y:Num} c?x!y?y' \longrightarrow d![y, y'] \longrightarrow STOP_T$$

where $\sqcap_{y:Num}$ is a construct that nondeterministically chooses a natural number y. It is important to notice something here, which is true of all nondeterministic selections: $\$y$ is performed before $?x$, although it is after $?x$ in the prefix. Therefore the process first nondeterministically selects a y and then offers any x and any y', rather than offering any x and with it a nondeterministically chosen y together with an offering of any y', which would have different semantics (see below).

Nondeterministic selections are not a part of ordinary CSP [Hoa85, Ros98a], and any nondeterministic selection can be replaced as above by a replicated nondeterministic choice. However, it will be useful to have a separate notation for nondeterministic selections in this thesis, primarily because of the condition ($\mathbf{RecCho}_X$) (see Section 3.5).

It is worth remembering that our whole language is an applied functional language and therefore does not have assignment. This means that once a variable acquires a value by an input or a nondeterministic selection, it has that same value for the rest of its lifetime. Although we sometimes speak of the same variable acquiring a different value by another input or nondeterministic selection, what really happens is that the old variable ceases to exist and a new variable with the same name is introduced (as with x and y in the third example above).

To make the typing rule for prefixing simpler, we introduce separate typing rules for outputs, inputs and nondeterministic selections. The types U of values which can be output, input or nondeterministically selected will be required to satisfy U^{comm} (see under 'Conditions on types' in Section 2.4), i.e. to be types that can be components of channel types. The outputs, inputs and nondeterministic selections themselves will then be of auxiliary types $Item_U$.

- An output of a term r is written

  $$\frac{\Gamma \vdash r : U}{\Gamma \vdash !r : Item_U} U^{comm}$$

  The term r can be built from the λ-calculus constructs (see the previous section).

- An input from a type U into a term variable x is written

  $$\frac{}{\Gamma \vdash (?x : U) : Item_U} U^{comm}$$

- A nondeterministic selection from a type U into a term variable x is written

  $$\frac{}{\Gamma \vdash (\$x : U) : Item_U} U^{comm}$$

The types $Item_U$ and the output, input and nondeterministic selection terms will not be treated as proper types and terms. In particular, we shall not specifically define their denotational semantics. They are to be thought of as a part of the typing rule for prefixing (see below).

As in the examples above, we shall omit types of inputs and nondeterministic selections when they are known from the context.

The typing rule for prefixing with an alphabet

$$T = \sum_{i=1}^{n} id_i . \prod_{j=1}^{m_i} T_{i,j}$$

is the following:

$$\frac{\begin{array}{c}\tilde{\Gamma} \vdash it_1 : Item_{T_{i,1}} \\ \vdots \\ \tilde{\Gamma} \vdash it_{m_i} : Item_{T_{i,m_i}} \\ \tilde{\Gamma} \cup Vars(it_1 \ldots it_{m_i}) \vdash P : Proc_{T,\phi}\end{array}}{\Gamma \vdash id_i\, it_1 \ldots it_{m_i} \longrightarrow P : Proc_{T,\chi}} \quad \left(\begin{array}{c} Dist(it_1 \ldots it_{m_i}) \wedge \\ Typ(it_1 \ldots it_{m_i}) \wedge \\ \phi \cup Ways(T, id_i, it_1 \ldots it_{m_i}) \subseteq \chi \end{array}\right)$$

where:

- id_i is any of the channels $id_1, \ldots, id_n$ in the alphabet T.
- $Vars(it_1 \ldots it_{m_i})$ is the type context consisting of variables of inputs and nondeterministic selections in the prefix and their types. We shall be referring to those variables as 'the variables of the prefix'.

  For example, if the alphabet is

  $$T = c.(Bool \times Num \times Num) + d.List(Num)$$

  (as in the examples above) and if $i = 1$ so that the prefix is with the channel c, we have:

  $$\begin{aligned} Vars(\$x : Bool\ !(r+3)\ ?y : Num) &= \{x : Bool, y : Num\} \\ Vars(!x\ !y\ !(r'+5)) &= \{\} \end{aligned}$$

- $\tilde{\Gamma} = \Gamma \setminus Domain(Vars(it_1 \ldots it_{m_i}))$, i.e. $\tilde{\Gamma}$ is Γ without variables which have the same names as the variables of the prefix.

  In particular, the variables of the prefix cannot be used within outputs in the prefix, so that it is for example not possible to write

  $$c\ ?x : Bool\ ?y : Num\ !(y+1) \longrightarrow \ldots$$

  (This is one of the points where our language is more restrictive than CSP$_M$.)

  $Vars(it_1 \ldots it_{m_i})$ is added to $\tilde{\Gamma}$ in the type judgement for P because the variables of the prefix can of course be used in P.

  When the new process is formed, the variables of the prefix become bound, so that Γ can be used as the type context of the final type judgement.

- $Dist(it_1 \ldots it_{m_i})$, the first part of the side condition, requires the variables of the prefix to be mutually distinct.

  For example:

  $$\begin{aligned} Dist(\$x : Bool\ !(r+3)\ ?y : Num) &= True \\ Dist(\$x : Bool\ !(r+3)\ ?x : Num) &= False \end{aligned}$$

- $Typ(it_1 \ldots it_{m_i})$ is more complex. It requires that either types of all inputs are products of type variables (i.e. of the form $X_1 \times \cdots \times X_k$) or types of all nondeterministic selections have no free type variables. In other words, if the prefix contains a nondeterministic selection which involves ranging over a type variable, then all inputs in the prefix must range only over type variables.

For example:

$$\begin{array}{rcl}
\mathit{Typ}(\$y : \mathit{List}(\mathit{Num})\ ?z : \mathit{Bool}) & = & \mathit{True} \\
\mathit{Typ}(\$y : \mathit{List}(Y)\ ?z : \mathit{Bool}) & = & \mathit{False} \\
\mathit{Typ}(\$y : \mathit{List}(Y)\ ?z : Z \times Z') & = & \mathit{True} \\
\mathit{Typ}(\$y : \mathit{List}(Y)\ ?z : \mathit{List}(Z)) & = & \mathit{False}
\end{array}$$

We have included this requirement for simplicity of the symbolic operational semantics of prefixing (see Section 3.2.3). If a prefixing does not satisfy it, we can for example replace its nondeterministic selections by replicated nondeterministic choices (see below), which would leave us a prefixing that does satisfy the requirement, although in that way we would fail the (**RecCho**$_X$) condition (see Section 3.5).

- $\mathit{Ways}(T, id_i, it_1\ \ldots\ it_{m_i})$ is the ways specifier with respect to T generated by the prefix. More precisely:

$$\mathit{Ways}(T, id_i, it_1\ \ldots\ it_{m_i})(id_{i'}, j') = \begin{cases} \{!\}, & \mathit{if}\ i' = i\ \mathit{and}\ it_{j'}\ \mathit{is\ an\ output} \\ \{?\}, & \mathit{if}\ i' = i\ \mathit{and}\ it_{j'}\ \mathit{is\ an\ input} \\ \{\$\}, & \mathit{if}\ i' = i\ \mathit{and}\ it_{j'}\ \mathit{is\ a\ nondeterministic\ selection} \\ \{\}, & \mathit{if}\ i' \neq i \end{cases}$$

The denotational semantics can be defined as follows.

Let

$$\begin{array}{rcl}
J_! & = & \langle j \mid j \in \langle 1, \ldots, m_i \rangle \wedge \exists\, r. it_j = !r \rangle \\
J_? & = & \langle j \mid j \in \langle 1, \ldots, m_i \rangle \wedge \exists\, x. it_j = ?x : T_{i,j} \rangle \\
J_\$ & = & \langle j \mid j \in \langle 1, \ldots, m_i \rangle \wedge \exists\, x. it_j = \$x : T_{i,j} \rangle
\end{array}$$

be the sequences of indices of outputs, inputs and nondeterministic selections (respectively) in $it_1\ \ldots\ it_{m_i}$.

For any tuple

$$v \in \prod_{j=1}^{m_i} [\![T_{i,j}]\!]_\delta$$

let

$$\begin{array}{rcl}
\pi_!(v) & = & (v(j) \mid j \in J_!) \\
\pi_?(v) & = & (v(j) \mid j \in J_?) \\
\pi_\$(v) & = & (v(j) \mid j \in J_\$)
\end{array}$$

be the tuples of the components of v whose indices are the indices of the outputs, inputs and nondeterministic selections (respectively).

Conversely, for any three tuples

$$w_! \in \prod_{j \in J_!} [\![T_{i,j}]\!]_\delta,\ w_? \in \prod_{j \in J_?} [\![T_{i,j}]\!]_\delta,\ w_\$ \in \prod_{j \in J_\$} [\![T_{i,j}]\!]_\delta$$

let

$$\iota(w_!, w_?, w_\$) \in \prod_{j=1}^{m_i} [\![T_{i,j}]\!]_\delta$$

be the tuple formed by interleaving $w_!$, $w_?$ and $w_\$$ appropriately.

At the empty trace the refusal sets are (informally speaking) complements of the prefix and the process cannot diverge, whereas at nonempty traces the process behaves like P:

$$\begin{aligned}
&\mathit{failures}_\bot(\Gamma \vdash \mathit{id}_i\ \mathit{it}_1\ \ldots\ \mathit{it}_{m_i} \longrightarrow P : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\
&\quad \{(\langle\rangle, A) \mid \exists\, w_\$ \in \prod_{j\in J_\$} [\![T_{i,j}]\!]_\delta . \forall\, w_? \in \prod_{j\in J_?} [\![T_{i,j}]\!]_\delta . (\mathit{id}_i, \iota(w_!, w_?, w_\$)) \notin A\} \\
&\quad \cup \{(\langle(\mathit{id}_i, v)\rangle \hat{}\, t, A) \mid\ \pi_!(v) = w_! \wedge \\
&\qquad\qquad (t, A) \in \mathit{failures}_\bot(\ \tilde{\Gamma} \cup \mathit{Vars}(\mathit{it}_1\ \ldots\ \mathit{it}_{m_i}) \vdash \\
&\qquad\qquad\qquad P : \mathit{Proc}_{T,\phi})_{\delta,\eta'_v}\} \\
&\mathit{divergences}(\Gamma \vdash \mathit{id}_i\ \mathit{it}_1\ \ldots\ \mathit{it}_{m_i} \longrightarrow P : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\
&\quad \{\langle(\mathit{id}_i, v)\rangle \hat{}\, t \mid\ \pi_!(v) = w_! \wedge \\
&\qquad\qquad t \in \mathit{divergences}(\tilde{\Gamma} \cup \mathit{Vars}(\mathit{it}_1\ \ldots\ \mathit{it}_{m_i}) \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta'_v}\}
\end{aligned}$$

where

$$w_! = ([\![\tilde{\Gamma} \vdash r : T_{i,j}]\!]^{\mathrm{FD}}_{\delta,\eta} \mid j \in J_! \wedge !r = \mathit{it}_j)$$

is the tuple of interpretations of outputs in the prefix, and

$$\eta'_v = \eta[\pi_?(v), \pi_\$(v) / \mathit{Vars}(\mathit{it}_1\ \ldots\ \mathit{it}_{m_i})]$$

is η updated by assigning the values given by v to the variables of inputs and nondeterministic selections in the prefix.

External choice. The external choice between two processes P and Q is a process $P \mathrel{\Box} Q$ which offers a choice of all initial visible events of P and Q, and depending on whose event is chosen behaves like P or Q subsequently:

$$\frac{\Gamma \vdash P : \mathit{Proc}_{T,\phi} \qquad \Gamma \vdash Q : \mathit{Proc}_{T,\psi}}{\Gamma \vdash P \mathrel{\Box} Q : \mathit{Proc}_{T,\chi}} \phi \cup \psi \subseteq \chi$$

For examples involving $\Box$, see for instance Examples 2.1.1–2.1.3 and Example 2.5.1.

At the empty trace the refusals of $P \mathrel{\Box} Q$ are those that both P and Q can refuse, whereas at nonempty traces $P \mathrel{\Box} Q$ behaves either like P or like Q:

$$\begin{aligned}
&\mathit{failures}_\bot(\Gamma \vdash P \mathrel{\Box} Q : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\
&\quad \{(\langle\rangle, A) \mid (\langle\rangle, A) \in\ \mathit{failures}_\bot(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \\
&\qquad\qquad \cap \mathit{failures}_\bot(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta}\} \\
&\quad \cup \{(\langle\rangle, A) \mid \langle\rangle \in\ \mathit{divergences}(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \\
&\qquad\qquad \cup\, \mathit{divergences}(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta}\} \\
&\quad \cup \{(t, A) \mid t \neq \langle\rangle\ \wedge\ (t, A) \in\ \mathit{failures}_\bot(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \\
&\qquad\qquad \cup \mathit{failures}_\bot(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta}\}
\end{aligned}$$

Any divergence of P or Q is a divergence of $P \,\square\, Q$:

$$\begin{aligned}&\mathit{divergences}(\Gamma \vdash P \,\square\, Q : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\ &\qquad \mathit{divergences}(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \,\cup\, \mathit{divergences}(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta}\end{aligned}$$

Nondeterministic choice. The nondeterministic choice between P and Q is a process which at the beginning nondeterministically (i.e. internally) decides to behave either like P or like Q:

$$\frac{\Gamma \vdash P : \mathit{Proc}_{T,\phi} \qquad \Gamma \vdash Q : \mathit{Proc}_{T,\psi}}{\Gamma \vdash P \sqcap Q : \mathit{Proc}_{T,\chi}} \phi \cup \psi \subseteq \chi$$

The process *NondReg* in Example 2.5.1 involves $\sqcap$.

The behaviours of $P \sqcap Q$ are all the behaviours of P and all the behaviours of Q:

$$\begin{aligned}&\mathit{failures}_\perp(\Gamma \vdash P \sqcap Q : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\ &\qquad \mathit{failures}_\perp(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \,\cup\, \mathit{failures}_\perp(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta} \\ &\mathit{divergences}(\Gamma \vdash P \sqcap Q : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\ &\qquad \mathit{divergences}(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \,\cup\, \mathit{divergences}(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta}\end{aligned}$$

Thus the interpretation of $P \sqcap Q$ is the greatest lower bound of the interpretations of P and Q with respect to refinement (see Sections 2.1.3 and 2.6). In other words, $P \sqcap Q$ is the most refined process that both P and Q refine, which confirms the intention that 'refines' should mean 'is less nondeterministic than'.

Replicated external choice. An alternative to external choice between two processes is replicated external choice over a type U:

$$\frac{(\Gamma \setminus \{x\}) \cup \{x : U\} \vdash P : \mathit{Proc}_{T,\phi}}{\Gamma \vdash \square_{x:U} P : \mathit{Proc}_{T,\chi}} U^{+,\times} \wedge \phi \subseteq \chi$$

U is required to satisfy the condition $U^{+,\times}$, i.e. to be built only from sums and products, which means that it has no free type variables and that it is finite. Free type variables are not allowed because our current theory, in particular symbolic operational semantics (see Chapter 3), is not capable of dealing with replicated external choices over type variables; in other words, we are not able to treat replicated external choice as a construct that preserves data independence (see Section 2.7).[2] On the other hand, U is required to be finite not because of data independence but because it would otherwise be impossible to define congruent denotational semantics of replicated external choice, since replicated external choice among infinitely many processes each of which begins with a τ event (see Section 2.1.1) is a divergent process.

Such a type U can be enumerated by a finite number of terms $r_1, \ldots, r_k$, so we can reduce replicated external choices to binary ones:

$$\square_{x:U} P = P[r_1/x] \,\square\, \cdots \,\square\, P[r_k/x]$$

[2]Extending the theory to allow such choices is a topic for future research, although it is not clear how useful that would be in practice.

Thus there is no need to define the denotational semantics directly: we can use the one for binary external choices given above.

The other direction is also true, namely that any binary external choice can be rewritten as a replicated one over a two-element type, for instance the type *Bool* (see Example 2.4.1). This is also true of nondeterministic choices (see below).

Replicated nondeterministic choice. There is also a replicated version of nondeterministic choice:

$$\frac{(\Gamma \setminus \{x\}) \cup \{x : U\} \vdash P : \mathit{Proc}_{T,\phi}}{\Gamma \vdash \bigsqcap_{x:U} P : \mathit{Proc}_{T,\chi}} U^{comm} \ \wedge \ \phi \subseteq \chi$$

In contrast to replicated external choice, U can contain free type variables and/or be infinite (but see under 'Restrictions with FD' below).

As with binary nondeterministic choice, the denotational semantics is defined by taking unions:

$$\begin{aligned}
&\mathit{failures}_{\bot}(\Gamma \vdash \textstyle\bigsqcap_{x:U} P : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\
&\qquad \bigcup_{v \in [\![U]\!]_\delta} \mathit{failures}_{\bot}((\Gamma \setminus \{x\}) \cup \{x : U\} \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta[v/x]} \\
&\mathit{divergences}(\Gamma \vdash \textstyle\bigsqcap_{x:U} P : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\
&\qquad \bigcup_{v \in [\![U]\!]_\delta} \mathit{divergences}((\Gamma \setminus \{x\}) \cup \{x : U\} \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta[v/x]}
\end{aligned}$$

Parallel composition. Given processes P and Q with an alphabet T, and a set S of elements of T, we can form the parallel composition of P and Q with the synchronisation set S:

$$\frac{\Gamma \vdash P : \mathit{Proc}_{T,\phi} \qquad \Gamma \vdash Q : \mathit{Proc}_{T,\psi} \qquad \Gamma \vdash S : \mathit{Set}_T}{\Gamma \vdash P \underset{S}{\parallel} Q : \mathit{Proc}_{T,\chi}} \phi \underset{S}{\parallel} \psi \ \subseteq \ \chi$$

The set S is a term of the auxiliary type Set_T. Similarly to outputs, inputs and nondeterministic selections (see above), set types and set terms, as well as the associated elem types and elem terms (see under 'Sets' and 'Elems' later in this section), will not be treated as proper types and terms. They are to be thought of as parts of the typing rules for parallel composition and hiding.

The ways specifier $\phi \underset{S}{\parallel} \psi$ is defined as follows, which is saying that $P \underset{S}{\parallel} Q$ might use a channel component in any way in which either P or Q might, except that it might use for inputs a component of a channel all of whose events are in the set S only if both P and Q might:

$$! \in (\phi \underset{S}{\parallel} \psi)(id_i, j) \ \Leftrightarrow \ ! \in \phi(id_i, j) \vee ! \in \psi(id_i, j)$$

$$? \in (\phi \underset{S}{\parallel} \psi)(id_i, j) \ \Leftrightarrow \ \begin{cases} ? \in \phi(id_i, j) \wedge ? \in \psi(id_i, j), \\ \quad \textit{if } *_T \textit{ is an elem of } S \\ \quad \textit{or } id_i.* \textit{ is an elem of } S \\ ? \in \phi(id_i, j) \vee ? \in \psi(id_i, j), \\ \quad \textit{otherwise} \end{cases}$$

$$\$ \in (\phi \underset{S}{\|} \psi)(id_i, j) \;\Leftrightarrow\; \$ \in \phi(id_i, j) \vee \$ \in \psi(id_i, j)$$

Some simple examples of parallel compositions were given in Examples 2.1.1–2.1.3.

In the definition of denotational semantics, we shall be using the operator $t_1 \underset{B}{\|} t_2$ which produces the set of all traces that can be obtained by putting a trace t_1 in parallel with a trace t_2 with respect to a synchronisation set B of visible events; it is defined in [Ros98a, Section 2.4].

If u is a trace obtained in this way from a trace t_1 of P and a trace t_2 of Q (where B is the interpretation of S), any refusal of $P \underset{S}{\|} Q$ at u consists of two parts: the part in B can be refused partially by P and partially by Q, whereas the part outside B has to be refused by both P and Q:

$$\begin{aligned}
&\mathit{failures}_\perp(\Gamma \vdash P \underset{S}{\|} Q : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\
&\quad \{(u, A_1 \cup A_2) \mid \; A_1 \setminus [\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta} = A_2 \setminus [\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta} \\
&\qquad\qquad \wedge \exists\, t_1, t_2 . (t_1, A_1) \in \mathit{failures}_\perp(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \\
&\qquad\qquad \wedge (t_2, A_2) \in \mathit{failures}_\perp(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta} \\
&\qquad\qquad \wedge u \in t_1 \underset{[\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta}}{\|} t_2\} \\
&\quad \cup \{(u, A) \mid u \in \mathit{divergences}(\Gamma \vdash P \underset{S}{\|} Q : \mathit{Proc}_{T,\chi})_{\delta,\eta}\}
\end{aligned}$$

A divergence of $P \underset{S}{\|} Q$ is a parallel composition of a trace of P and a trace of Q where at least one of them is a divergence, or an extension of such:

$$\begin{aligned}
&\mathit{divergences}(\Gamma \vdash P \underset{S}{\|} Q : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\
&\quad \{u\hat{\ }u' \mid \; \exists (t_1, \{\}) \in \mathit{failures}_\perp(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta}, \\
&\qquad\quad (t_2, \{\}) \in \mathit{failures}_\perp(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta} . \\
&\qquad\quad u \in t_1 \underset{[\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta}}{\|} t_2 \\
&\qquad\quad \wedge\; (t_1 \in \mathit{divergences}(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta} \\
&\qquad\qquad \vee\, t_2 \in \mathit{divergences}(\Gamma \vdash Q : \mathit{Proc}_{T,\psi})_{\delta,\eta})\}
\end{aligned}$$

Hiding. A set of visible events S can be hidden from a process P:

$$\frac{\begin{array}{c}\Gamma \vdash P : \mathit{Proc}_{T,\phi} \\ \Gamma \vdash S : \mathit{Set}_T\end{array}}{\Gamma \vdash P \setminus S : \mathit{Proc}_{T,\chi}} \phi \setminus S \subseteq \chi$$

The ways specifier $\phi \setminus S$ is defined as follows, which is saying that $P \setminus S$ might use any channel component in any of the ways in which P might, except that it does not use any component of a channel all of whose events are in the set S:

$$(\phi \setminus S)(id_i, j) = \begin{cases} \{\}, \; \textit{if } *_T \textit{ is an elem of } S \textit{ or } id_i . * \textit{ is an elem of } S \\ \phi(id_i, j), \; \textit{otherwise} \end{cases}$$

Some simple examples of hiding can be found in Examples 2.1.2 and 2.1.3.

Visible events are hidden by being turned into τ events, which are not recorded in the denotational semantics. Thus if B is the interpretation of S and if $t \setminus B$ is the trace obtained from a trace t by removing all events in B, then for any failure $(t, A \cup B)$ of P, $P \setminus S$ has the failure $(t \setminus B, A)$; the refusal set of P needs to contain B in order for the stable state in which it is refused to remain stable in $P \setminus S$. We also need to add failures generated by divergences:

$$\begin{array}{l} \mathit{failures}_\perp(\Gamma \vdash P \setminus S : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\ \quad \{(t \setminus [\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta}, A) \mid \ (t, A \cup [\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \in \mathit{failures}_\perp(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta}\} \\ \quad \cup \{(t, A) \mid t \in \mathit{divergences}(\Gamma \vdash P \setminus S : \mathit{Proc}_{T,\chi})_{\delta,\eta}\} \end{array}$$

We define the divergences to be those arising from divergences of P, and those arising from 'infinite traces' of P from which all but finitely many events have been hidden:

$$\begin{array}{l} \mathit{divergences}(\Gamma \vdash P \setminus S : \mathit{Proc}_{T,\chi})_{\delta,\eta} = \\ \quad \{(t \setminus [\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta})\hat{\ }t' \mid t \in \mathit{divergences}(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta}\} \\ \quad \cup \{(u \setminus [\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta})\hat{\ }t' \mid \ u \in ([\![T]\!]_\delta)^\omega \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, u \setminus [\![\Gamma \vdash S : \mathit{Set}_T]\!]^{\mathrm{FD}}_{\delta,\eta} \text{ is finite} \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \forall t < u. \\ \qquad\qquad\qquad\qquad\qquad\qquad (t, \{\}) \in \mathit{failures}_\perp(\Gamma \vdash P : \mathit{Proc}_{T,\phi})_{\delta,\eta}\} \end{array}$$

where $([\![T]\!]_\delta)^\omega$ is the set of all infinite sequences of events from the interpretation $[\![T]\!]_\delta$ of the alphabet, and $t < u$ is true when t is a strict prefix of u.

However, as explained in [Ros98a, Section 8.3.2], the identification of infinite traces with infinite sequences all of whose strict prefixes are traces is correct provided P contains no infinitary nondeterminism. Therefore, whenever we are working with the failures/divergences model, we restrict ourselves to finitely nondeterministic CSP constructs (see below, under 'Restrictions with FD' (i)).

Recursion. One often needs to define processes that can communicate unboundedly many times, and recursion is the construct to achieve that.

Recall the processes in Examples 2.1.1–2.1.3 which were defined by recursion. Most of them, such as COPY_1, do not have an argument:

$$\mathit{COPY}_1 = \mathit{in}?x \longrightarrow \mathit{mid}!x \longrightarrow \mathit{COPY}_1$$

whereas B' has an argument:

$$\begin{array}{l} B = \mathit{in}?x \longrightarrow B'(x) \\ B'(x) = (\mathit{in}?y \longrightarrow \mathit{out}!x \longrightarrow B'(y)) \\ \qquad\quad \Box\ (\mathit{out}!x \longrightarrow B) \end{array}$$

When there is no argument, the right-hand side of a recursive definition can be seen as a function from processes to processes, so that the process being defined is its fixed point.

The same is true for processes with an argument, except that they can be seen as functions from arguments to processes and the right-hand side of a recursive definition as a function from such functions to such functions. In fact, the former case is included in the latter by taking the type of the argument to be the singleton type 1 (see under 'Product types' in Section 2.4), and so it will suffice for our recursion construct to cover the latter case.

Another issue is defining a finite number of processes by simultaneous recursion, such as B and $B'(x)$ above. However, such definitions can for example be reduced by substitution to a number of nested recursive definitions of single processes, so we do not need to explicitly support them in our construct.

If U is a type which does not involve process types, $\mathit{Proc}_{T,\phi}$ is a process type, p is a variable of type $U \to \mathit{Proc}_{T,\phi}$, and P is a term of type $U \to \mathit{Proc}_{T,\phi}$ which can contain free occurences of p, then the construct produces a process which is the result of the recursive definition $p = P$. The type $U \to \mathit{Proc}_{T,\phi}$ can be thought of as a type of processes which have arguments of type U.

$$\frac{(\Gamma \setminus \{p\}) \cup \{p : U \to \mathit{Proc}_{T,\phi}\} \vdash P : U \to \mathit{Proc}_{T,\phi}}{\Gamma \vdash (\mu\, p : (U \to \mathit{Proc}_{T,\phi}).P) : U \to \mathit{Proc}_{T,\phi}} U^{\mathit{AllTypeVars},+,\times,\mu,\to}$$

(There should be no confusion with the μ construct for forming inductive types.)

In Example 2.5.1 we show the formalities of using the construct to write recursive definitions such as those given above.

The interpretation

$$[\![\Gamma \vdash (\mu\, p : (U \to \mathit{Proc}_{T,\phi}).P) : U \to \mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{\delta,\eta}$$

of the recursively defined process is the least fixed point in the partial order

$$[\![U \to \mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{\delta}$$

of the function $\mathcal{G}$ that informally speaking maps p to P. Formally, $\mathcal{G}$ is defined as

$$\mathcal{G}(g) = [\![(\Gamma \setminus \{p\}) \cup \{p : U \to \mathit{Proc}_{T,\phi}\} \vdash P : U \to \mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{\delta,\eta[g/p]}$$

Since $[\![\mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{\delta}$ is a complete partial order (see under 'Process types' in Section 2.4), so is $[\![U \to \mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{\delta}$ (see under 'Function types' in Section 2.4). By well-definedness, which is proved at the same time as defining the denotational semantics (see Section 2.5.3), $\mathcal{G}$ is continuous. Therefore, the least fixed point of $\mathcal{G}$ can be obtained as the least upper bound of the chain of all its finite iterations from the least element:

$$[\![\Gamma \vdash (\mu\, p : (U \to \mathit{Proc}_{T,\phi}).P) : U \to \mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{\delta,\eta} = \bigsqcup_{k \in \mathbb{N}} \mathcal{G}^k(\bot)$$

where $\bot(v)$ is the least element of $[\![\mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{\delta}$ for all v.

The definitions in the other two models, Tr and Fa, are the same.

The parallel composition and hiding constructs involved sets, which are in our language formed by the following constructs. Let us emphasise again that set types and set terms, as well as the associated elem types and elem terms, are auxiliary. They will not be treated as proper types and terms, but are to be thought of as parts of the typing rules for parallel composition and hiding.

Sets. We shall restrict ourselves to sets of visible events, although sets of any type satisfying $\mathit{AllTypeVars}, +, \times, \mu$ (see under 'Conditions on types' in Section 2.4) can be formed in the same way.

Given an alphabet T (see under 'Process types' in Section 2.4 and Example 2.4.1), a set of events from T consists of finitely many 'elems' which are terms of another auxiliary type Elem_T (see below):

$$\frac{\Gamma \vdash e_1 : \mathit{Elem}_T \qquad \Gamma \vdash e_k : \mathit{Elem}_T}{\Gamma \vdash \{e_1, \ldots, e_k\}_{\mathit{Set}_T} : \mathit{Set}_T} T^{\mathit{alph}}$$

Each elem will either represents one event, or a number of events if it contains $*_U$ constructs which mean 'any element of a type U'.

When $k = 0$, we get the empty set. The subscript Set_T exists for this case, to avoid ambiguity over the type of $\{\}$ (see Section 2.5.3). As with other type tags, including those on elem terms (see below), we shall omit it when it is clear which type is to be used.

Although the notation may be suggesting otherwise, the order of elems in a set is significant. In other words, two sets which differ only in the order of their elems are to be considered different, at least in formal developments.

An elem will be interpreted by a set, and the interpretation of a set term is the union of the interpretations of its elems:

$$\begin{aligned}&[\![\Gamma \vdash \{e_1, \ldots, e_k\}_{\mathit{Set}_T} : \mathit{Set}_T]\!]^M_{\delta,\eta} = \\ &\qquad [\![\Gamma \vdash e_1 : \mathit{Elem}_T]\!]^M_{\delta,\eta} \cup \cdots \cup [\![\Gamma \vdash e_k : \mathit{Elem}_T]\!]^M_{\delta,\eta}\end{aligned}$$

Elems. To formulate the typing rules for elems, it is convenient to allow the types U in the types Elem_U of elems to be any types satisfying $U^{\mathit{AllTypeVars},+,\times,\mu}$ (see under 'Conditions on types' in Section 2.4).

Any term r of such a type U which has no free type variables can be regarded as an elem:

$$\frac{\Gamma \vdash r : U}{\Gamma \vdash r_{\mathit{Elem}_U} : \mathit{Elem}_U} U^{\mathit{AllTypeVars},+,\times,\mu} \wedge \mathit{Free}(U) = \{\}$$

which represents the single element of U given by r:

$$[\![\Gamma \vdash r_{\mathit{Elem}_U} : \mathit{Elem}_U]\!]^M_{\delta,\eta} = \{[\![\Gamma \vdash r : U]\!]^M_{\delta,\eta}\}$$

On the other hand, the elem $*_U$ represents all elements of U:

$$\frac{}{\Gamma \vdash *_U : \mathit{Elem}_U} U^{\mathit{AllTypeVars},+,\times,\mu}$$

where U is allowed to have free type variables.

$$[\![\Gamma \vdash *_U : \mathit{Elem}_U]\!]^M_{\delta,\eta} = [\![U]\!]_\delta$$

On top of those two basic ways of forming elems, we can use the three familiar λ-calculus constructs to produce elems of sum types, product types, and inductive types:

$$\frac{\Gamma \vdash e : \mathit{Elem}_{U_i}}{\Gamma \vdash id_i^{id_1.U_1+\cdots+id_l.U_l}.e : \mathit{Elem}_{id_1.U_1+\cdots+id_l.U_l}} \;\forall\, j.U_j^{\mathit{AllTypeVars},+,\times,\mu}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Elem}_{U_1} \qquad \ldots \qquad \Gamma \vdash e_l : \mathit{Elem}_{U_l}}{\Gamma \vdash (e_1, \ldots, e_l) : \mathit{Elem}_{U_1\times\cdots\times U_l}}$$

$$\frac{\Gamma \vdash e : \mathit{Elem}_{U[(\mu\, X.U)/X]}}{\Gamma \vdash in_{\mu\, X.U}\; e : \mathit{Elem}_{\mu\, X.U}}$$

We omit their interpretations, which are liftings to sets of the interpretations given in Section 2.5.1.

We aimed for the simplest possible syntax of sets and elems that suffices for most practical parallel compositions and hidings. Thus we excluded sets which are selective about elements of variable types. For example, if c is a channel of type X (a type variable) in some alphabet, it is not possible to form a set such as

$$\{c.x, c.y, c.z\}$$

This is achieved by requiring elems to have $*_{U'}$ constructs in places where a subterm of a variable type is needed. If the following more precise explanation helps, given an elem e of type Elem_U for some U and given a free occurence of a type variable X in U, if we try to reach the corresponding place in e by descending from the top of its syntax tree, we shall pass through a number of constructs for forming elems of sum types, product types and inductive types until we arrive at an $*_{U'}$ where either $U' = X$ or X occurs free within U'. (See the example below, and see how in this way the firing rules for parallel composition and hiding in Section 3.2.3 are made simpler. See also under 'Parallel composition' and 'Hiding' in the proof of Theorem 4.2.2.)

On the other hand, we allowed more general sets than for instance sets of all events along a number of channels, because some components of events may be used as indices and sets in parallel compositions and hidings may need to be selective about those. That is why we allow elems to have as subterms any specific terms of types without free type variables. For example, if d and d' are channels of type $\mathit{Num} \times X$ in some alphabet, we can form a set

$$\{d.(r+1, *_X), d'.*_{\mathit{Num}\times X}\}$$

which is the set of all events along d whose first component is equal to $r+1$ (where r is some term) and of all events along d'.

Restrictions with FD. Whenever we are working with $M = \mathrm{FD}$, i.e. with the failures/divergences model of CSP, the following two restrictions have to be observed:

(i) Only finitely nondeterministic CSP constructs can be considered.

More precisely,

- in any nondeterministic selection $\$x : U$,
- in any replicated nondeterministic choice $\bigsqcap_{x:U} P$, and
- in any term $*_U$ which is anywhere within the set S of any hiding $P \setminus S$,

U has to satisfy U^{fin} (see under 'Conditions on types' in Section 2.4).

(ii) Any type variable X which is free in a type T which is used as an alphabet (i.e. to form a process type $Proc_{T,\phi}$) has to be from *FinTypeVars* (see under 'Type variables' in Section 2.4).

(Any such X is already required to be from *NEFTypeVars*: see the definition of T^{alph} under 'Conditions on types' in Section 2.4.)

We saw the need for (i) under 'Hiding' above. The need for (ii) arises in the proofs of Proposition 4.1.7 and Theorem 4.2.2. ■

More precise ways specifiers. The requirements on ways specifiers in the typing rules are simple and entirely syntactical. Most notably, the treatment of sets in the typing rules for parallel composition and hiding is very simple.

Consequently, the typing rules sometimes require the ways specifiers to be less precise (i.e. pointwise larger) than they could be.

More substantially, any ways specifier assigns a set of ways to a channel component irrespectively of the values of other components of the same channel. Thus our very notion of ways specifiers is not general enough to be useful with arrays of channels, i.e. with channels some of whose components are used for indexing.

For example, any ways specifier of any of the processes *Comp*, *Comps* and *MultMatr* in Example 2.7.4 is required to include both ! and ? for the third components of the channels *hor* and *vert*.

However, for better focus on the essentials of our work, we leave the subject of more precise ways specifiers for future research. Allowing ways specifiers which would be useful with arrays of channels seems to require using the denotational semantics and/or the symbolic operational semantics (see Chapter 3) in the typing rules. ■

About CSP constructs not in our language. CSP and CSP$_M$ have a number of constructs that are not present in our language. Firstly, there are constructs such as boolean guard, untimed time-out and various kinds of parallel composition, and special processes such as RUN, *Chaos* and $\bot$, which can be defined in the standard ways [Ros98a] in terms of the constructs we have.

Secondly, there are constructs which cannot in general be defined in our language, such as renaming, sequential composition and interrupt. Consequently, the theorems in Chapter 5 are not immediately applicable to practical processes which involve those constructs. However, it seemed appropriate to devote the thesis to the essentials of our work, and to state versions of the theorems which apply directly to full CSP$_M$ in other publications [Laz99, Ros98a, LR98].

Finally, there are various kinds of replicated parallel composition. As with replicated external choice (see above), our current theory would not be able to deal with replicated parallel compositions over type variables, i.e. to regard them as constructs preserving data independence. This is not surprising, becuase verifying systems which are parameterised by how many parallel components they have or more generally by their network structure is a different, although quite related, research problem (see Sections 1.1.3, 1.2, 5.5.5 and 7.2.2). Again as with replicated external choice, replicated parallel compositions would also be restricted to finite types, which would make them expressible by iterating the parallel composition construct we have, either manually or dynamically over lists.

The relationship between our language and CSP$_M$ will be discussed further in Section 3.8. ■

Example 2.5.1 Suppose X is a type variable from *NEFTypeVars*. Let T be an alphabet consisting of channels *in* and *out* of type X:

$$T = \mathit{in}.X + \mathit{out}.X$$

and let ϕ be a ways specifier with respect to T which says that *in* can be used only for inputs and *out* only for outputs:

$$\begin{aligned}\phi(\mathit{in},1) &= \{?\}\\ \phi(\mathit{out},1) &= \{!\}\end{aligned}$$

The following is a process with alphabet T and ways specifier ϕ:

$$\begin{aligned}&\mathit{NondReg} = \mathit{in}?x \longrightarrow \mathit{NondReg}'(x)\\ &\mathit{NondReg}'(x) = (\mathit{in}?y \longrightarrow \mathit{if}\ x = y\\ &\qquad\qquad\qquad\qquad\qquad \mathit{then}\ \mathit{NondReg}'(x)\\ &\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \mathit{NondReg}'(x) \sqcap \mathit{NondReg}'(y))\\ &\qquad\qquad\quad \Box\ (\mathit{out}!x \longrightarrow \mathit{NondReg}'(x))\end{aligned}$$

At the beginning, *NondReg* inputs a value x of type X and becomes the recursively defined process *NondReg*$'(x)$. At any point, *NondReg*$'(x)$ can either input another value y of type X or output the value x it has. If x and y are different, it can then nondeterministically become *NondReg*$'(x)$ or *NondReg*$'(y)$, i.e. nondeterministically decide to keep one of x and y and forget the other one. If x and y are equal, and also in case output was chosen instead of input, it goes back to being *NondReg*$'(x)$.

Thus *NondReg* can be thought of as a nondeterministic register: at any point, it can be keeping and it can output any one of the values that have been input so far, where which one it is depends on which branches have been chosen in nondeterministic choices. In particular, the *if-then-else* can be replaced by just *NondReg*$'(x) \sqcap$ *NondReg*$'(y)$, since *NondReg*$'(x)$ is equivalent to *NondReg*$'(x) \sqcap$ *NondReg*$'(y)$ when $x = y$.

The definitions of *NondReg* and *NondReg*$'(x)$ involve some 'syntactic sugar'. *NondReg* can be formally defined as a process in our language as follows:

$$\begin{aligned}\mathit{NondReg} = \mathit{in}\,?x : X \longrightarrow\ &((\mu\, p : (X \to \mathit{Proc}_{T,\phi}).\\ &\lambda\, x' : X.\\ &(\mathit{in}\,?y : X \longrightarrow (x' = y) \rightsquigarrow p\, x',\\ &\qquad\qquad\qquad\qquad\qquad (p\, x') \sqcap (p\, y))\\ &\Box\ (\mathit{out}\,!x' \longrightarrow p\, x'))\ x)\end{aligned}$$

where:

- The type X is explicitly displayed in the inputs.
- The recursive definition of $\mathit{NondReg}'(x)$ is written using the μ construct with the type $X \to \mathit{Proc}_{T,\phi}$, since x in $\mathit{NondReg}'(x)$ is an argument of type X (see under 'Recursion' above).
- The x which is input at the beginning of $\mathit{NondReg}$ is supplied as the argument of the recursively defined process.
- The *if-then-else* can in this case written by a single equality test construct (see under 'Equality testing' in Section 2.5.1).

$\mathit{NondReg}$ is then a process in our language, in the sense that the type judgement

$$\{\} \vdash \mathit{NondReg} : \mathit{Proc}_{T,\phi}$$

is derivable from the typing rules given in this section and in Section 2.5.1. The type context is $\{\}$ because $\mathit{NondReg}$ has no free term variables.

Instead of ϕ, we could have used any ways specifier ϕ' with respect to T which is less restrictive, i.e. such that $\phi \subseteq \phi'$.

We shall use the process $\mathit{NondReg}$ in a number of examples in the thesis. ■

Other CSP constructs and syntactic sugar. Before Example 2.5.1 we mentioned a number of CSP constructs which are not formally present in our language but which can be defined from the constructs we have, and in Example 2.5.1 we saw how various kinds of syntactic sugar can be translated into exact terms of our language.

From now on, we shall often use such CSP constructs and syntactic sugar without bothering to define them, provided there is no ambiguity about their meanings. ■

2.5.3 Well-definedness

In this section, we complete the presentation of our language and its denotational semantics by sketching how well-definedness of the denotational semantics of terms is proved.

The denotational semantics of terms was defined by induction on the typing rules. Therefore, for the time being, we have to allow for the possibility that the interpretations of terms depend on derivation trees.

Derivation trees. Derivation trees are the ways in which type judgements can be derived using the typing rules. More precisely, nodes of derivation trees are instances of the typing rules in which the side conditions are true, their edges are type judgements, the order of edges from any node is significant, and they have root nodes (which are at the bottom).

Thus a type judgement $\Gamma \vdash r : T$ is derivable if and only if it is the conclusion of the root node of a derivation tree $\mathcal{R}$. In that case we say that $\mathcal{R}$ derives $\Gamma \vdash r : T$. ■

Consequently, for any term $\Gamma \vdash r : T$, any derivation tree $\mathcal{R}$ which derives it, and any M, δ and η, we shall write

$$[\![\mathcal{R}]\!]^{M}_{\delta,\eta}$$

for the interpretation of $\Gamma \vdash r : T$ which is obtained by applying the definition of the denotational semantics over $\mathcal{R}$.

The first half of the proof of well-definedness is proving that the interpretations of terms are elements of the interpretations of their types:

Proposition 2.5.1 *For any term $\Gamma \vdash r : T$, any derivation tree $\mathcal{R}$ which derives it, and any M, δ and η, we have that*

$$[\![\mathcal{R}]\!]^M_{\delta,\eta} \in [\![T]\!]^M_\delta$$

Proof. It suffices to prove the following stronger statement: for any term $\Gamma \vdash r : T$, any derivation tree $\mathcal{R}$ which derives it, and any M and δ, we have that

$$\eta \mapsto [\![\mathcal{R}]\!]^M_{\delta,\eta}$$

is a continuous function from the product $\prod\{[\![U]\!]^M_\delta \mid (x : U) \in \Gamma\}$ of the partial orders which interpret the types in Γ into the partial order $[\![T]\!]^M_\delta$ which interprets T.

The stronger statement is proved at the same time as defining the denotational semantics of terms, i.e. by the same induction. Most of the work, namely that the CSP constructs preserve the axioms of the three CSP models and that they are continuous with respect to the orderings on those models (see under 'Process types' in Section 2.4), where in the case of the failures/divergences model we are restricted to finitely nondeterministic CSP constructs, can be borrowed from the literature on CSP (see [Ros98a, Chapter 8], [Ros88]).

The proofs of the continuity of equality tests, prefixings, parallel compositions and hidings use the assumptions that the types U of any terms which are tested for equality and of any outputs satisfy U^{comm}, and that the types U of any terms r_{Elem_U} anywhere within sets satisfy $U^{AllTypeVars,+,\times,\mu} \wedge \mathit{Free}(U) = \{\}$. The assumptions imply that all interpretations of those types have flat orderings, so that any continuous function into such an interpretation must be constant on any directed set which has a least upper bound. ■

The second half of the proof of well-definedness is the following proposition, whose part (b) says that the interpretations of terms depend neither on their derivation trees nor on their types.

Definition 2.5.1 For two type judgements $\Gamma \vdash r : T$ and $\Gamma' \vdash r' : T'$, let us write

$$(\Gamma \vdash r : T) \sim (\Gamma' \vdash r' : T')$$

if and only if $\Gamma = \Gamma'$, $r = r'$ and $T \sim T'$ (see Definition 2.4.1). ■

Proposition 2.5.2 *Suppose derivation trees $\mathcal{R}$ and $\mathcal{R}'$ derive type judgements $\Gamma \vdash r : T$ and $\Gamma \vdash r : T'$ (respectively).*

(a) $\mathcal{R}$ and $\mathcal{R}'$ are isomorphic, their corresponding edges are related by $\sim$, and $T \sim T'$.

(b) For any M, δ and η, we have that

$$[\![\mathcal{R}]\!]^M_{\delta,\eta} = [\![\mathcal{R}']\!]^M_{\delta,\eta}$$

Proof. (a) is proved by induction on the structure of r.

To prove (b), it suffices by (a) to observe that, for any typing rule

$$\frac{\Delta_1 \vdash s_1 : V_1 \qquad \ldots \qquad \Delta_k \vdash s_k : V_k}{\Delta' \vdash s' : V'}\,\mathit{Cond}$$

the definition of the interpretations of $\Delta' \vdash s' : V'$ depends neither on ways specifiers within the types $V_1, \ldots, V_k$, nor on which ways specifier is chosen for the type V', where the latter applies only for the typing rules of the CSP constructs other than recursion. ■

What part (a) of the proposition is saying is that, for any Γ and r, all derivation trees which derive type judgements of the form $\Gamma \vdash r : T$ are the same except that they may differ in ways specifiers within final types of their type judgements. In particular, all the types T in derivable type judgements of the form $\Gamma \vdash r : T$ are the same except that they may differ in ways specifiers. The reason why there can be more than one T and more than one derivation tree is the freedom of choosing different ways specifiers in the typing rules for the CSP constructs other than recursion (see Section 2.5.2).

Given any Γ and r, the following proposition tells us more about how the types T for which $\Gamma \vdash r : T$ is derivable are related.

Definition 2.5.2 Let $\preceq$ be the partial ordering on the types of our language which is such that

$$\begin{array}{rcl}
X \preceq X' & \Leftrightarrow & X = X' \\
\begin{array}{r} id_1.T_1 + \cdots + id_n.T_n \preceq \\ id'_1.T'_1 + \cdots + id'_{n'}.T'_{n'} \end{array} & \Leftrightarrow & \begin{array}{r} id_1.T_1 + \cdots + id_n.T_n = \\ id'_1.T'_1 + \cdots + id'_{n'}.T'_{n'} \end{array} \\
\begin{array}{r} T_1 \times \cdots \times T_n \preceq \\ T'_1 \times \cdots \times T'_{n'} \end{array} & \Leftrightarrow & n = n' \wedge \forall i.T_i \preceq T'_i \\
\mu X.T \preceq \mu X'.T' & \Leftrightarrow & \mu X.T = \mu X'.T' \\
T_1 \to T_2 \preceq T'_1 \to T'_2 & \Leftrightarrow & T_1 = T'_1 \wedge T_2 \preceq T'_2 \\
\mathit{Proc}_{T,\phi} \preceq \mathit{Proc}_{T',\phi'} & \Leftrightarrow & T = T' \wedge \phi \subseteq \phi'
\end{array}$$

and such that no two types whose principal type constructs are of different kinds are related.

For any two types T and T' which have a greatest lower bound with respect to $\preceq$, we shall denote it by

$$T \curlywedge T'$$

■

Proposition 2.5.3 *(a) If $\Gamma \vdash r : T$ and $\Gamma \vdash r : T'$ are derivable, then $T \curlywedge T'$ exists and $\Gamma \vdash r : T \curlywedge T'$ is derivable.*

(b) If Γ and r are such that the set

$$\{T \mid \Gamma \vdash r : T\}$$

is nonempty, then it has a least element with respect to $\preceq$.

Proof. (a) is proved by induction on the structure of r.

(b) follows from (a), once we observe that, by Proposition 2.5.2 (a), $\{T \mid \Gamma \vdash r : T\}$ is finite. ■

Given any derivable type judgement $\Gamma \vdash r : T$, the following proposition tells us which type contexts can be used instead of Γ.

Proposition 2.5.4 *Suppose* $\Gamma \vdash r : T$ *is a derivable type judgement, and let*

$$\Gamma' = \{(x : U) \in \Gamma \mid x \textit{ occurs free in } r\}$$

Then, for any type context Γ'', *we have that* $\Gamma'' \vdash r : T$ *is derivable if and only if* $\Gamma' \subseteq \Gamma''$.

Proof. By induction on the structure of r. ■

Some conventions. When it is either clear or insignificant which of the possible types T to use, we may write $\Gamma \vdash r$ instead of $\Gamma \vdash r : T$.

Instead of $[\![\mathcal{R}]\!]^M_{\delta,\eta}$, we shall write $[\![\Gamma \vdash r : T]\!]^M_{\delta,\eta}$ (as we did in Sections 2.5.1 and 2.5.2) or $[\![\Gamma \vdash r]\!]^M_{\delta,\eta}$. ■

2.6 Definition of refinement

We have informally introduced refinement in Section 2.1.3. As we have indicated there, in CSP refinement is the primary ordering on denotational models and the notion of correctness, and thus both theory and practice (see Section 2.1.4) of CSP are based on it.

Formally, if $\Gamma \vdash P : Proc_{T,\phi}$ and $\Gamma \vdash Q : Proc_{T,\psi}$ are two processes with a common alphabet T, we shall write

$$(\Gamma \vdash P) \sqsubseteq^M_{\delta,\eta} (\Gamma \vdash Q)$$

to mean that $\Gamma \vdash P$ is refined by $\Gamma \vdash Q$ with respect to M, δ and η, where as before:

- M specifies one of the three models Tr, Fa, FD of CSP,
- δ assigns a partial order to each type variable, subject to the restrictions stated under 'Type variables' in Section 2.4, and
- η is an assignment to Γ with respect to M and δ.

By Proposition 2.5.4, the requirement that P and Q have the same type context Γ can be met whenever they have some type contexts Γ_1 and Γ_2 which assign the same type to any variable which occurs free in both P and Q.

Refinement was defined as the reverse subset ordering, so that we have:

$$\begin{array}{lcl}
(\Gamma \vdash P) \sqsubseteq^{\mathrm{Tr}}_{\delta,\eta} (\Gamma \vdash Q) & \Leftrightarrow & \textit{traces}(\Gamma \vdash P)_{\delta,\eta} \supseteq \textit{traces}(\Gamma \vdash Q)_{\delta,\eta} \\
(\Gamma \vdash P) \sqsubseteq^{\mathrm{Fa}}_{\delta,\eta} (\Gamma \vdash Q) & \Leftrightarrow & \textit{traces}(\Gamma \vdash P)_{\delta,\eta} \supseteq \textit{traces}(\Gamma \vdash Q)_{\delta,\eta} \\
 & & \wedge\, \textit{failures}(\Gamma \vdash P)_{\delta,\eta} \supseteq \textit{failures}(\Gamma \vdash Q)_{\delta,\eta} \\
(\Gamma \vdash P) \sqsubseteq^{\mathrm{FD}}_{\delta,\eta} (\Gamma \vdash Q) & \Leftrightarrow & \textit{failures}_\perp(\Gamma \vdash P)_{\delta,\eta} \supseteq \textit{failures}_\perp(\Gamma \vdash Q)_{\delta,\eta} \\
 & & \wedge\, \textit{divergences}(\Gamma \vdash P)_{\delta,\eta} \supseteq \textit{divergences}(\Gamma \vdash Q)_{\delta,\eta}
\end{array}$$

Some conventions. When it is either clear or insignificant which Γ to use, we may omit it.

When $\Gamma = \{\}$, we may omit η, and when in addition there are no free type variables, we may also omit δ. ■

In Example 2.1.4, we discussed refinement between the processes *BImpl* and *BUFF*, which can be written in our language with the alphabet

$$in.X + mid.X + out.X$$

where X is a type variable from *NEFTypeVars*.

Example 2.6.1 Consider the following process *LastReg*, with the same alphabet as the process *NondReg* in Example 2.5.1:

$$\begin{aligned}
&\mathit{LastReg} = in?x \longrightarrow \mathit{LastReg}'(x)\\
&\mathit{LastReg}'(x) = (in?y \longrightarrow \mathit{LastReg}'(y))\\
&\qquad\qquad\quad \Box\ (out!x \longrightarrow \mathit{LastReg}'(x))
\end{aligned}$$

LastReg is in a sense a simpler version of *NondReg*: whereas *NondReg* is a nondeterministic register, *LastReg* is a deterministic register which always decides to keep the last value it has input.

Thus we can expect that *NondReg* is refined by *LastReg*, and indeed

$$\mathit{NondReg} \sqsubseteq^M_\delta \mathit{LastReg}$$

in any model M and with any nonempty set $\delta[\![X]\!]$.

The question of reverse refinement is more interesting. When X is assigned a one-element set, the two processes have the same interpretations, because there is only one possible value they can ever output. In particular,

$$\mathit{LastReg} \sqsubseteq^M_\delta \mathit{NondReg}$$

for any M when $|\delta[\![X]\!]| = 1$.

On the other hand, when the set assigned to X has at least two elements, *NondReg* has traces such as $\langle in.v, in.w, out.v\rangle$ with $v \neq w$, which are not traces of *LastReg*, so that

$$\mathit{LastReg} \not\sqsubseteq^M_\delta \mathit{NondReg}$$

for any M and any $|\delta[\![X]\!]| \geq 2$.

Thus in contrast to *BImpl* and *BUFF* in Example 2.1.4, refinement of *LastReg* by *NondReg* depends on the size of the set assigned to X. We shall see many more such examples in the rest of the thesis, especially in Chapter 5. ■

Although for purposes of illustration we considered the counter-intuitive refinements of *LastReg* by *NondReg* and of *BImpl* by *BUFF*, in the rest of the thesis we shall think of refinement primarily as the notion of correctness. Thus we shall typically think of the left-hand side of refinement as the specification and of the right-hand side as the implementation, and use names such as *Spec* and *Impl* instead of P and Q.

More precisely, as we have explained in Chapter 1, the whole thesis is concerned with how we can automatically verify that a data independent specification is refined by a (proposed) data independent implementation for all instances of (i.e. assignments to) the data independent type (see Section 2.7 and Chapter 5).

2.7 Definition of data independence

In this section we shall define when a process is data independent, and also weaker properties of weak data independence and of parameterisation by data types and process-free type contexts.

All three properties are restrictions of the basic phenomenon of a process having a data type that can be varied. This phenomenon is in our language formalised as a process having a free type variable, so that each of the three properties will consist of restrictions that have to be added for the property to be present.

Although a process can have each property in a number of free type variables, we shall for simplicity be concentrating on the case of only one free type variable. When there are more than one, they can be analysed manually one at a time by substituting concrete data types (i.e. data types which have no free type variables) for the rest; see Example 2.7.1 below and the case study in Chapter 6. Alternatively, theorems which cover any number of free type variables explicitly are straightforward to obtain from the single-variables versions that can be found in Chapter 5.

2.7.1 Data independence

Intuitively, a process P is data independent in a data type X when the only things it can do with values from X is to

- input them,
- keep them and copy them within its variables,
- output them, and
- perform equality tests between them.

In our language, this will be defined by P having X as a free type variable and the only operations that can involve values from X being the operations already provided by the language constructs: equality tests, construction of tuples, selection from tuples, etc.

We shall have such a simple definition because of the simplicity of our language. In a more complex language such as CSP$_M$ [Sca92, Sca98, Ros98a], the definition of data independence has to contain a number of additional restrictions which we have incorporated into our language from the beginning: for example, replicated external choices over free type variables or computing sizes of data types with free type variables would not be allowed. (See [Laz99], [Ros98a, Section 15.2.2].)

Since otherwise values of type X could not be communicated or tested for equality (see Section 2.5), it will make sense to for simplicity require X to be from *NEFTypeVars*. Since we are concentrating on single free type variables (see above), X will be the only free type variable of P.

As for free term variables, P will be allowed to have only those of type X, which we shall call 'constants'. We shall think of any constant as being able to initially assume any value from X and then (as any other variable in our language, since we do not have assignments) having to stay fixed for the rest of its lifetime. What we shall typically want to do with a process which has constants is to verify that a refinement involving it is true no matter what values the constants assume. A data independent process with a constant is given in Example 2.7.1 below.

P will not allowed to have free term variables of any other types, since they would either be operations that can involve values from X (such as those of type $X \to Bool$ or $X \to X$) or they would be outside the scope of this thesis (such as those of type Num or $Proc_{T,\phi}$, which could be used to parameterise P by the number of components in a parallel composition). Other possibilities, such as free term variables of type $Bool \to X$ which can be seen as pairs of constants, will be excluded for simplicity.

Definition 2.7.1 A process $\Gamma \vdash P : Proc_{T,\phi}$ is said to be data independent with respect to X and Γ if and only if

- X is from *NEFTypeVars*,
- X is the only free type variable of $\Gamma \vdash P : Proc_{T,\phi}$, and
- Γ is of the form $\{x_1 : X, \ldots, x_k : X\}$.

We shall abbreviate this by saying that P is X,Γ-DI. When there are no constants, we shall often write X-DI instead of X,$\{\}$-DI. We shall also use alternative phrases such as 'data independent in X with constants Γ' or 'independent of X with constants Γ'.

When X is clear from the context, we may write Γ as just $\{x_1, \ldots, x_k\}$. ■

For example, each of the processes

- *BImpl* from Example 2.1.3,
- B from Example 2.1.3,
- *BUFF* from Example 2.1.4,
- *NondReg* from Example 2.5.1, and
- *LastReg* from Example 2.6.1

are data independent in X and have no constants, i.e. they are X-DI.

Example 2.7.1 Consider a process $\{init : V\} \vdash Mem : Proc_{T,\phi}$ defined as follows, where A and V are type variables from *NEFTypeVars*:

$$\begin{aligned}
&Mem = Mem'(\lambda\, a' : A.init)\\
&Mem'(f) = (read\_addr\, ?a : A \longrightarrow read\_val!f(a) \longrightarrow Mem'(f))\\
&\qquad\qquad \Box\, (write\, ?a : A\, ?v : V \longrightarrow Mem'(\lambda\, a' : A.(a' = a) \rightsquigarrow v, f(a')))
\end{aligned}$$

Mem can be thought of as a memory in which the type of location addresses is A and the type of storable values is V. At any point *Mem* keeps a function f from addresses to values which is the current state of the memory. Initially all locations in the memory contain the value of the constant *init*, so that the function initially maps any address to this value.

When a user asks to read from an address a, the memory outputs $f(a)$ and carries on with the same f. On the other hand, when a user writes a value v to an address a, the memory substitutes f by a function that, given an address a' returns v if a' is equal to a and returns $f(a)$ otherwise.

Since we are formally restricting ourselves to one type variable at a time (see the beginning of Section 2.7), what we can state is that *Mem* is data independent in each of A and V separately, i.e. that $Mem[U/V]$ is A-DI and $Mem[U/A]$ is V,$\{init\}$-DI for any type U such that U^{nef} and $Free(U) = \{\}$ (see under 'Type variables' in Section 2.4).

It might be puzzling that *Mem* is data independent in A and V in spite of working with functions from A to V. However, working with such functions falls into the category mentioned above of operations which are already provided by constructs of the language, in this case functional abstraction, functional applications and equality tests between elements of A. This also conforms with the intuitive definition of data independence given above, since the only things *Mem* can do with elements of A and V is to input them, keep them in its variables (most notably f), output elements of V, and perform equality tests between elements of A. ■

Example 2.7.2 Another simple example is the following process, taken from [Ros98a, Section 15.2.2]. It is an X-DI process that removes adjacent duplicates from a stream of values of type X:

$$\begin{aligned}
&RemDups = left?x \longrightarrow right!x \longrightarrow RemDups'(x)\\
&RemDups'(x) = left?y \longrightarrow (x = y) \rightsquigarrow RemDups'(x),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad right!y \longrightarrow RemDups'(y)
\end{aligned}$$

■

Example 2.7.3 The following is not a data independent process. It applies a function $g : X \to X$ pointwise to lists of values from X, which is not allowed since g would have to be a free term variable and we only allow free term variables of type X. Thus, although the flow of control of the process does not depend on values from X, it fails to be data independent in our sense because it uses an operation on X.

$$Q = in\,?l : List(X) \longrightarrow out!(map\ g\ l) \longrightarrow Q$$

■

Moving on from toy examples, many practical processes are data independent, which is in a way surprising given how strong a property data independence seems to be. For example:

- Communication protocols are typically data independent in the type of values they communicate.
- Memories, databases and systems involving them are frequently data independent in the types of addresses and values (for a discussion of cases with more than one type variable, see the beginning of Section 2.7).
- Agents and the spy in security protocols (see [Ros98a, Section 15.3] and references given there) are typically data independent or nearly so in the types of agent identities, nonces and keys. As Roscoe and Broadfoot have recently been showing by using techniques presented in this thesis, those properties can be exploitied to prove such protocols with model checkers [Ros98b, Bro98, RB98].
- Nodes in a system consisting of many nodes are sometimes data independent in the type of their identities, although the whole system is typically not because we do not regard replicated parallel compositions over type variables as preserving data independence (and we consequently do not have them in our language). As Lowe, Roscoe and Creese have been showing by building on the work presented in this thesis, data independence enables such systems to be proved with model checkers by induction, through reducing the base case and inductive step proof obligations to finite instaces [Low96, Cre98, CR99b].

Something to point out is that the same system may or may not be data independent depending on the level of abstraction at which it is represented. For example, the process *Mem* in Example 2.7.1 is data independent in the type A of addresses, although at a lower level of abstraction we can expect various routing operations to be applied to the addresses.

2.7.2 Weak data independence

The intuitive definition of when a process P is weakly data independent in X is that the only possible way values from X can affect the control flow of P is by equality tests.

Formally, in addition to free term variables of type X that were allowed by data independence (see the previous section), weak data independence allows free term variables of types $U \to X$ for any type U that does not involve process types: these are thought of as arbitrary operations whose results are from X.

Operations whose result types involve concrete data types would either provide ways for values from X to influence the control flow (such as predicates of type $X \to \mathit{Bool}$) or they would be outside of the scope the thesis (such as a free term variable of type $1 \to \mathit{Num}$ which amounts to parameterisation by a natural number). On the other hand, operations whose results are for example of type $X \times X$ are excluded for simplicity, since they are the same as a number of operations with results from X. Finally, allowing the types U to involve process types seemed pointless and would complicate the symbolic operational semantics (see Chapter 3).

Definition 2.7.2 A process $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$ is said to be weakly data independent with respect to X and Γ if and only if

- X is from *NEFTypeVars*,
- X is the only free type variable of $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$, and
- Γ is of the form $\{x_1 : X, \ldots, x_k : X, y_1 : U_1 \to X, \ldots, y_l : U_l \to X\}$, where the types U_j do not involve process types.

We shall abbreviate this by saying that P is X,Γ-WDI, and also use alternative phrases such as 'weakly data independent in X and Γ' or 'independent of X with constants and operations in Γ'. We shall refer to the x_is as 'the constants in Γ' and to the y_js as 'the operations in Γ'. ■

Thus any data independent process is also weakly data independent, but because of allowing operations the class of weakly data independent processes is much larger.

We have already seen a weakly data independent process Q that uses an operation g on X in Example 2.7.3. More precisely, Q is X,$\{g : X \to X\}$-WDI.

Example 2.7.4 An important subclass of weakly data independent processes are systolic arrays, which are regular (e.g. linear or rectangular) arrays of identical component processes, and whose purpose is to perform a costly computation in parallel by having the components repeatedly perform simple computation steps on data which is travelling through the array. (See e.g. [JG88].)

For example, a systolic array for multiplying matrices can be built as a rectangular array of the same dimensions as resulting matrices, in which each component process computes the corresponding entry. The generic component process can be defined as *Comp* in:

$$
\begin{aligned}
&\mathit{Comp}(i,j) = \mathit{Comp}'(i,j,\mathit{zer})\\
&\mathit{Comp}'(i,j,x) = (\mathit{hor}!i!j?y \longrightarrow \mathit{vert}!i!j?y' \longrightarrow\\
&\qquad\qquad\qquad\qquad \mathit{hor}!i!(j+1)!y \longrightarrow \mathit{vert}!(i+1)!j!y' \longrightarrow\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Comp}'(i,j,\mathit{add}(x,\mathit{mult}(y,y'))))\\
&\qquad\qquad\quad \Box\ (\mathit{finished}!i!j \longrightarrow \mathit{out}!i!j!x \longrightarrow \mathit{Comp}(i,j))
\end{aligned}
$$

where:

- i and j are arguments of type *Num*, representing the row and column;
- *zer* is a constant of type X, and *add* and *mult* are binary operations on X (i.e. of type $(X \times X) \to X$);
- *hor* and *vert* are channels of type $\mathit{Num} \times \mathit{Num} \times X$, so they can be used as collections of channels of type X indexed by ordered pairs of natural numbers: they provide horizontal and vertical links between adjacent components, and between the environment and the components which are on the edges of the array.

The parallel composition

$\mathit{Comps}(g,h)$

of g rows and h columns of component processes can then be defined by iterating the parallel composition construct over lists, and then the whole systolic array

$\mathit{MultMatr}(g,h)$

can be defined by hiding the internal events from $\mathit{Comps}(g,h)$. The arguments g and h are of type *Num*, and if either of them is 0, both $\mathit{Comps}(g,h)$ and $\mathit{MultMatr}(g,h)$ are for example STOP. We omit the details.

The point of this example is that each of the processes *Comp*, *Comps* and *MultMatr* is weakly data independent in X with the constant *zer* and the operations *add* and *mult*, except that their types are of the form

$$(\mathit{Num} \times \mathit{Num}) \to \mathit{Proc}_{T,\phi}$$

rather than just $\mathit{Proc}_{T,\phi}$ (see Section 2.7.4).

For some remarks about ways specifiers of *Comp*, *Comps* and *MultMatr*, see under 'More precise ways specifiers' in Section 2.5.2.

If r and s are two positive integers, then $\mathit{MultMatr}(r,s)$ is a systolic array for multiplying matrices of dimensions $r \times q$ and $q \times s$ (for any positive integer q). The type of entries is the type variable X, with the formal constant and operations. By varying the interpretations of X, *zer*, *add* and *mult*, we can obtain for example an array for the natural numbers (with *Num*, 0, $+$ and $\times$) or an array for the booleans (with *Bool*, *false*, *or* and *and*). In Section 5.1, we shall see that, by exploiting weak data independence, some properties can be verified for all the possible instances of $\mathit{MultMatr}(r,s)$ by checking only the minimal instance, in which X is assigned a singleton set.

Thus in order to exploit weak data independence formally in a process that is 'weakly data independent' in a concrete data type such as *Num*, the concrete data type needs to be replaced by a type variable and all the operations on it by formal operations. The process will then conform to the precise definition of weak data independence given above, and theorems in Section 5.1 may apply and say that the process refines a specification no matter how the type variable, constants and operations are interpreted. ■

Example 2.7.5 A simpler example is the following process which inputs a value x of type X and then outputs successive iterations of a function f on x until, if ever, it reaches a fixed point:

$$\begin{aligned}&\mathit{FP} = \mathit{left}?x \longrightarrow \mathit{FP}'(x)\\&\mathit{FP}'(x) = \mathit{right}!x \longrightarrow (f(x) = x) \leadsto \mathit{STOP},\\&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{FP}'(f(x))\end{aligned}$$

FP is X,$\{f : X \to X\}$-WDI and, in contrast to the processes we have seen above, it uses equality tests on X. ■

Example 2.7.6 Another example is the following peculiar process which is weakly data independent in X with two operations h and h' on X. It nondeterministically chooses a value from X and then repeatedly applies h and h' to it and performs *blink* events until, if ever, the value produced by h is not equal to the value produced by h':

$$\begin{aligned}&\mathit{Blinking} = \textstyle\bigsqcap_{x:X} \mathit{Blinking}'(x, x)\\&\mathit{Blinking}'(y, z) = (y = z) \leadsto \mathit{blink} \longrightarrow \mathit{Blinking}'(h(y), h'(z)),\\&\qquad\qquad\qquad\qquad\qquad\quad \mathit{STOP}\end{aligned}$$

■

2.7.3 Parameterisation by data types and process-free type contexts

Definition 2.7.3 A process $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$ is said to be parameterised by X and process-free Γ if and only if

- X is from *NEFTypeVars*,
- X is the only free type variable of $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$, and
- for any $(x : U) \in \Gamma$, U involves no process types.

We shall abbreviate this by saying that P is X,Γ-PPF. ■

This class of processes is much larger than the class of weakly data independent ones: it consists of all processes which are parameterised by a data type X and any number of operations, predicates and constants whose types may or may not involve X. It is thus not surprising that refinements involving X,Γ-PPF processes cannot in general be verified by reduction to small finite instances of X, which is why we do not consider them in this thesis.

However, there are useful subclasses for which verification by reduction to finite instances is possible. In particular, as we have recently shown for verifying determinism and conjectured for verifying refinement [LR98], reduction to finite instances is possible if Γ consists of only constants and 'many-valued predicates', where many-valued predicates are functions from X into nonempty finite types which have no free type variables.

As well as considering subclasses, it is sometimes convenient to consider extensions. An example is provided by the same paper [LR98], whose main contribution was to show that reduction to finite instances can be done with processes that use arrays whose types of indices and values are two type variables X and Y: although arrays already exist in our language as functions, it was more convenient to formally extend the language by types $\mathit{Array}_{X,Y}$, and by constructs for creating arrays, reading from arrays, and updating arrays. (For further remarks, see Section 5.5.)

Parameterisation by data types and process-free type contexts is the most general case to which the symbolic operational semantics presented in Chapter 3 is applicable (see Section 3.1.3), and that is partly why we have given it a special status.

2.7.4 Terms of any type

It will be useful to extend the definitions of data independence, weak data independence, and parameterisation by data types and process-free type contexts to terms of any type. More precisely, for any term $\Gamma \vdash r : T$, we shall say that it is X,Γ-DI, X,Γ-WDI or X,Γ-PPF if it satisfies the corresponding definition above, except that T is not required to be a process type.

For example, if a process $c!r?x \longrightarrow P$ is X,Γ-DI, then so will be the term r.

2.8 Notes

The contribution of this chapter was to combine CSP [Hoa85, Ros98a] with typed λ-calculus [Mit90, GLT89, Loa97, Bir98] and obtain a simple, formal and expressive language in which data independent types are the same as free type variables (see Section 2.7). Having such a language will naturally lead us to, in Chapter 4, extend logical relations [Mit90, OHe96, Plo80, Rey83, Wad89] to process types and thus generalise reasoning about data independence [Wol86] and abstraction [CC77, CGL94, LG+95, Gra94, DF95, Dam96, Kur90] to higher types and to the stable-failures and failures/divergences models of CSP.

CSP$_M$ [Sca92, Sca98, Ros98a], the machine-readable version of CSP used in tools, is also a combination of CSP and a functional programming language. Our language can be seen as a smaller and more theoretically presented version of it, which makes the work in the thesis immediately useful in practice (see Sections 3.8 and 5.6).

Taking a wider view, our language is related in varying degrees to many languages in the literature which have both concurrent and sequential constructs. For example:

- most languages used in model checking (see e.g. [McM93, IP96, CAV]), which typically are concurrent languages with sublanguages for computations on data;
- various combinations of versions of CSP with languages that are either imperative or have some imperative features, such as Occam [Inm88, JG88], action systems [BK83, But96] and Idealized CSP [Bro97];
- various extensions of λ-calculi by concurrent constructs (see e.g. [PGM90, Ong93]), aiming to fully integrate concurrent and sequential computation, so that unlike in our language processes are treated in the same way as data and can be communicated along channels.

A number of features that distinguish CSP and our language from other languages and frameworks were discussed in Section 2.3.

Instead of combining a concurrent language with a specific sequential language, one can work with properties that a sequential part has to satisfy but otherwise leave it unspecified (as in e.g. [HL95a, HL95b]). For reasons why we did not take this approach, see the end of Section 2.2.

The denotational semantics we have given is basically the standard denotational semantics of CSP [Hoa85, Ros98a] and typed λ-calculus [Mit90, GLT89, Loa97], and it has much in common with the denotational semantics of CSP$_M$ [Sca98]. We have included it for completeness and because of the special constructs in our language such as nondeterministic selections and sets.

Chapter 3

Symbolic operational semantics

As we have discussed in Chapter 1, the aim of our semantic study of data independence are the theorems in Chapter 5 about how the truth of refinements between data independent processes changes with varying the size of the data independent type. Since refinement is defined in terms of denotational semantics (see Sections 2.1.3 and 2.6), the proofs of those theorems will require knowledge of how the denotational semantics of a data independent process changes with the size of the data independent type.

Symbolic operational semantics, that this chapter is devoted to, will provide a way of tackling this problem. In contrast to denotational semantics, symbolic operational semantics does not require data independent types to be assigned concrete sets, but allows them to be left as type variables. The single interpretation of a data independent process in symbolic operational semantics will then act as a bridge among all the various interpretations in the denotational semantics, by translations to and from it.

Operational semantics in general is at a lower level of abstraction than denotational semantics: it interprets a program by specifying how it executes. In the standard operational semantics of CSP [Ros98a, Chapter 7], this amounts to interpreting a process by a *labelled transition system (LTS)*, which is a directed graph whose nodes interpret the states of the process and whose arcs (which are called transitions) are labelled by events. In symbolic operational semantics, the idea is the same, except that instead of assigning concrete values, some or all variables are left as symbols. The variables we shall be leaving as symbols will be those of data independent types, which is how we shall be able to interpret processes whose data independent types are left as uninterpreted type variables. The resulting interpretations will be *symbolic labelled transition systems (SLTSs)*.

More precisely, we shall present and study symbolic operational semantics for weakly data independent processes in our language (of which the data independent ones are a subclass: see Sections 2.7.1 and 2.7.2), so that in addition to variables of weakly data independent types, operations involving such types will also be left as symbols. We shall also point towards how the same can be done for the more general class of processes parameterised by data types and process-free type contexts (see Section 2.7.3).

We shall also present and study *symbolic normalisation*, a procedure for transforming SLTSs of data independent processes into a form where any trace can be executed in at most one way. (This will be needed in Section 5.4.)

The symbolic operational semantics is presented in the first three sections, and the fourth section is devoted to its relationship with the denotational semantics. In the fifth section, a number of conditions that will subsequently be important and their consequences are presented. The two sections following are on symbolic normalisation. In the section after those, implications for tools are discussed.

3.1 The λ-calculus constructs

As we have said, the first goal of this chapter is to give symbolic operational semantics to weakly data independent processes in our language.

Given such a process P, it is general built from both the λ-calculus constructs (see Section 2.5.1) and the CSP constructs (see Section 2.5.2). In particular, the λ-calculus constructs may be present at the top level, so that it may be necessary to reduce them before we can determine the initial symbolic events of P. For example, if

$$\begin{aligned} P = pr_2\,(Q_1, ((f\ x) = y) \rightsquigarrow\ & c!x \longrightarrow Q_2, \\ & STOP) \end{aligned}$$

where f is an operation on X and where x and y are constants of type X, then P can be reduced to $c!x \longrightarrow Q_2$ (which will have a single initial symbolic event $c.x$) under the condition $(f\ x) = y$, or to $STOP$ (which will have no symbolic events) under the condition $\neg(f\ x) = y$.

In this section, we shall state formally that this is always possible, namely that given any weakly data independent process we can always by reducing only λ-calculus constructs arrive at one or more processes with determinable initial symbolic events. The proofs, which are outside the scope of the thesis, will be omitted: readers who are interested in reduction in typed λ-calculi can consult e.g. [Mit90, GLT89, Loa97].

In a number of places, we shall be relying on the strong normalisation property (see e.g. [GLT89, Loa97]) of the λ-calculus constructs of our language, i.e. that there are no infinite sequences of consecutive reductions. Since the property is not present in CSP$_M$, corresponding developments for it would be more complex.

3.1.1 Conditional symbolic transitions

The λ-calculus constructs are reduced by repeatedly performing the one-step reductions listed in Table 3.1.

The first four one-step reductions are standard and show how to reduce an elimination construct applied to an introduction construct, for each of the sum, product, inductive and function types (see Section 2.4). The one for inductive types can be thought of as a recursive definition of the *fold* construct (see Section 2.5.1): intuitively, the term

$$T[(\lambda\, x : (\mu\, Y.U).\mathit{fold}\ x\ s)/Y](r)$$

is obtained from r by applying the function

$$\lambda\, x : (\mu\, Y.U).\mathit{fold}\ x\ s$$

at the places in r which correspond to the places in U where Y occurs free. If Y is a type variable which is neither in *NEFTypeVars* nor in *FinTypeVars*, if U is a type satisfying $U^{(+,\times,\mu)_Y}$, if f is a term of a function type $V \to V'$, and if r is a term of type $U[V/Y]$, then the term $U[f/Y](r)$ can be defined by induction on the structure of U, which is known as the 'types as functors' construction (see e.g. [Loa97]).

The two remaining one-step reductions are the two branches arising from reducing an equality test: since we shall be using the reductions within a symbolic operational semantics, it will

$$
\begin{array}{rcl}
\begin{array}{r}\mathit{case}\ (id_i^{id_1.U_1+\cdots+id_n.U_n}.r)\\ \mathit{of}\ s_1\ \mathit{or}\ \ldots\ \mathit{or}\ s_n\end{array} & \longrightarrow & s_i\,r \\
pr_i\,(r_1,\ldots,r_n) & \longrightarrow & r_i \\
\mathit{fold}\ (in_{\mu\,Y.U}\ r)\ s & \longrightarrow & s\,U[(\lambda\,x:(\mu\,Y.U).\mathit{fold}\ x\ s)/Y](r) \\
(\lambda\,x:U.r)\ s & \longrightarrow & r[s/x] \\
((r=s)\rightsquigarrow u,v) & \xrightarrow{r=s} & u \\
((r=s)\rightsquigarrow u,v) & \xrightarrow{\neg r=s} & v
\end{array}
$$

Table 3.1: The one-step reductions

not in general be possible to determine the outcome of an equality test, so instead we shall allow for both possibilities and label the two reductions with the conditions under which they are valid.

We shall not insist that the conditions be exactly $r = s$ and $\neg r = s$, but shall allow any equivalent ones instead.

Conditional symbolic transitions. Given a weakly data independent process, it will be reduced by performing the one-step reductions repeatedly, either to the whole of it or to a part. A sequence of such one-step reductions will be called a *conditional symbolic transition* and it will be labelled by the conjunction of all the conditions by which the one-step reductions are labelled (where an empty conjunction is *True*).

More generally, suppose a term $\Gamma \vdash r_0 : T$ is X,Γ-WDI (see Section 2.7.4) and $r_1, \ldots, r_k$ are such that any r_{i+1} is obtained from r_i by performing a one-step reduction either to the whole of r_i or to a part, except that an equality test within r_i can be reduced only when all the free variables in the terms being compared are from Γ. Then each r_{i+1} is also a term for which the type judgement $\Gamma \vdash r_{i+1} : T$ is derivable and which is X,Γ-WDI.

If out of those k one-step reductions l are labelled and $cond_1, \ldots, cond_l$ are their labels, let $cond' = cond_1 \wedge \cdots \wedge cond_l$. (If $l = 0$, this means that $cond' = True$.)

We then write

$$r_0 \quad [cond'\rangle \quad r_k$$

and say that r_0 has a conditional symbolic transition $[cond'\rangle$ to r_k.

As with labels of one-step reductions, we shall allow any equivalent condition instead of the exact $cond_1 \wedge \cdots \wedge cond_l$. ■

For example, for P given above we have

$$P \quad [(f\ x) = y\rangle \quad (c!x \longrightarrow Q_2)$$

and

$$P \quad [\neg(f\ x) = y\rangle \quad STOP$$

where in each case the conditional symbolic transition consists of two one-step reductions, the first to reduce the selection of the second component of the pair, and the second to reduce the equality test.

x_i	$STOP_V$
y_j	$id_i\ it_1^\sharp\ \ldots\ it_m^\sharp \longrightarrow P$
$id_i^{id_1.V_1+\cdots+id_n.V_n}.r^\sharp$	$P^\sharp \;\Box\; Q^\sharp$
$(r_1^\sharp,\ldots,r_n^\sharp)$	$P \sqcap Q$
$in_{\mu\, Z.V}\ r^\sharp$	$\Box_{z:V}\, P^\sharp$
$\lambda\, z : V.r^\sharp$	$\sqcap_{z:V}\, P$
$y_j\ r^\sharp$	$P^\sharp \underset{S^\sharp}{\parallel} Q^\sharp$
$\mu\, p : (W \to Proc_{V,\phi}).P$	$P^\sharp \setminus S^\sharp$
	$(\mu\, p : (W \to Proc_{V,\phi}).P)\ r^\sharp$

The subterms which have the $^\sharp$ superscripts are also required to be in evaluated form. For a communication item it_j, this applies only when it_j is an output $!r$ and means that the term r should be in evaluated form. For a set S, it means that for any r_{Elem_W} which is a subterm of an elem e in S but which is not a subterm of a different subterm $r'_{Elem_{W'}}$ of e (see under 'Sets' and 'Elems' in Section 2.5.2), the term r is required to be in evaluated form.

Table 3.2: X,Γ-WDI terms in evaluated form

We have used the name 'conditional symbolic transitions' because they will be one of the three kinds of transitions in symbolic labelled transition systems by which processes will be interpreted. The other two will be called visible and invisible symbolic transitions, through which visible and invisible (τ) events are performed (see Section 3.2.2).

3.1.2 Evaluated form

In this section we shall state that any weakly data independent term (see Section 2.7.4) has one or more conditional symbolic transitions to terms which are in *evaluated form*. For processes, this will be the result we need, since when a process is in evaluated form its initial visible and invisible symbolic transitions will be determinable (see Section 3.2).

Definition 3.1.1 An X,Γ-WDI term, where

$$\Gamma = \{x_1 : X,\ldots,x_k : X, y_1 : U_1 \to X,\ldots,y_l : U_l \to X\}$$

is in evaluated form if and only if it is of one of the forms in Table 3.2. ■

We shall often be working with X,Γ-WDI evaluated forms (i.e. X,Γ-WDI terms in evaluated form) of process types, of type X, and of types satisfying $AllTypeVars,+,\times,\mu$, so it will be useful to know their structure more precisely:

Process types. An evaluated form which is a process must be of one of the forms in the right-hand column in Table 3.2, since no form in the left-hand column can be a process.

Type variable X. An evaluated form of type X is either

- a constant x_i from Γ, or
- of the form $y_j\ r^\sharp$, where y_j is an operation from Γ and r is an evaluated form.

Since by the definition of weak data independence (see Section 2.7.2) the type U_j of r does not involve process types, it can be shown that r does not involve process types or CSP constructs.

When the evaluated form is X,Γ-DI, there are no operations in Γ, and so the evaluated form can only be a constant x_i.

Types satisfying $\mathit{AllTypeVars}, +, \times, \mu$**.** This condition is weaker than *comm* (see under 'Conditions on types' in Section 2.4), and so it is satisfied by the types of terms which can be communicated along channels or tested for equality. It is also satisfied by the types of terms which can be used to form elems of sets (see under 'Sets' and 'Elems' in Section 2.5.2).

An evaluated form of such a type is built from

- evaluated forms of type X, and
- the constructs $id_i^{id_1.V_1+\cdots+id_n.V_n}.\cdot$, $(\cdot,\ldots,\cdot)$, and $in_{\mu\, Z.V}\ \cdot$ for introducing terms of sum, product and inductive types.

Thus the part of the evaluated form consisting of the three kinds of introduction constructs can be thought of as a 'shell' which contains the evaluated forms of type X. When using this fact in constructions and proofs (see for example the remarks after Proposition 3.1.1), we shall write the evaluated form as

$$H[r_1, \ldots, r_h]$$

where H stands for the shell and $r_1, \ldots, r_h$ are the evaluated forms of type X that the shell contains.

For example, if Γ contains constants x, y and z, and operations $f : X \to X$ and $g : ((X \to X) \times X) \to X$, then

$$(f\ x, 4, [y, g\ (f, z)])$$

is an X,Γ-WDI evaluated form of type $X \times \mathit{Num} \times \mathit{List}(X)$ in which the shell is

$$(\cdot, 4, [\cdot, \cdot])$$

and the evaluated forms of type X are $f\ x$, y and $g\ (f, z)$.

Proposition 3.1.1 *For any X,Γ-WDI term r, we can construct some $\mathit{cond}_1, \ldots, \mathit{cond}_n$ and $s_1, \ldots, s_n$ (where $n \geq 1$) such that:*

- *for each i,*

$$r \quad [\mathit{cond}_i\rangle \quad s_i$$

(see Section 3.1.1);

- *each $cond_i$ is either True, or False, or built from equality tests between X,Γ-WDI evaluated forms of type X and the boolean connectives $\neg$, $\wedge$ and $\vee$;*
- *each s_i is in evaluated form;*
- *the conditions $cond_1$, ..., $cond_n$ disjointly partition True, in the sense that for any concrete interpretation of the constants and operations in Γ, a unique $cond_i$ is true.*

■

More informally, the proposition is saying that the equality tests in r which need to be resolved in order to reduce r to evaluated form can be resolved by a finite branching over the conditions $cond_i$, and that each of those branches leads to an evaluated form s_i.

It is proved by showing that there is an appropriate reduction strategy which yields the $cond_i$ and s_i of the stated forms. More precisely, the reduction strategy produces a finite tree of one-step reductions (see Section 3.1.1) whose root is r, whose leaves are the s_i, and which branches whenever an equality test is reduced. Each $cond_i$ is then equivalent to the conjunction of all the labels along the path from r to s_i in the tree.

The particularly simple form of each $cond_i$ is achieved by ensuring that before reducing any equality test $u = v$, the terms u and v are reduced to evaluated form, so that we can then use the fact that two evaluated forms of a type satisfying $AllTypeVars, +, \times, \mu$ (see above) are equal if and only if their shells are identical and the corresponding evaluated forms of type X contained in the shells are equal. For example, the shells of the evaluated forms

$$\begin{aligned} u' &= (f\,x, 4, [y, g\,(f, z)]) \\ v' &= (x, 4, [x, g\,(f, y)]) \end{aligned}$$

are identical and hence the label $u' = v'$ is equivalent to

$$f\,x = x \;\wedge\; y = x \;\wedge\; g\,(f, z) = g\,(f, y)$$

and the label $\neg u' = v'$ to

$$\neg f\,x = x \;\vee\; \neg y = x \;\vee\; \neg g\,(f, z) = g\,(f, y)$$

We fix a reduction strategy. There is certainly more than one reduction strategy that can be used to obtain the $cond_i$ and the s_i, and different reduction strategies in general yield different results. For example, we may or may not choose to also reduce to evaluated form the X,Γ-WDI subterms of the s_i that are not required to be in evaluated form.

However, in order to be able to use Proposition 3.1.1 as a deterministic procedure, we shall from now on fix a reduction strategy, so that any application of Proposition 3.1.1 to the same X,Γ-WDI term r yields the same $cond_i$ and s_i. ■

3.1.3 Processes parameterised by data types and process-free type contexts

Terms which are parameterised by a data type X and a process-free type context Γ (see Sections 2.7.3 and 2.7.4), which is a much larger class than weakly data independent terms, can also be reduced to appropriate evaluated forms. In this section, we shall sketch how this can be achieved. However, subsequently, we shall be assuming weak data independence.

X,Γ-PPF terms can be more difficult to reduce than the X,Γ-WDI because in addition to branching that is required to resolve equality tests, they may require branching for resolving which component of a sum type a term belongs to. For example, if x is a constant in Γ of type *Bool* (which was defined in Example 2.4.1 as a two-component sum type), then reducing a process

$$\mathit{if}\ x\ \mathit{then}\ P_1\ \mathit{else}\ P_2$$

which is formally written as

$$\mathit{case}\ x\ \mathit{of}\ (\lambda\, w_1 : 1.P_1)\ \mathit{or}\ (\lambda\, w_2 : 1.P_2)$$

to evaluated form (so that we can determine its initial symbolic events) requires branching to resolve whether $x = \mathit{true}$ or $x = \mathit{false}$. As a more complex example, suppose y is a constant in Γ of type *Num* (see Example 2.4.1) and Q is an X,Γ-PPF process of a type $\mathit{Proc}_{V,\phi}$, and consider a process

$$\overbrace{Q \,\Box \cdots \Box\, Q}^{y}$$

which is formally written as

$$\begin{aligned}\mathit{fold}\ y\ \ &(\lambda\, v : (\mathit{zero}.1 + \mathit{succ}.\mathit{Proc}_{V,\phi}).\mathit{case}\ v\ \mathit{of}\\ &(\lambda\, w : 1.\mathit{STOP}_V)\ \mathit{or}\ (\lambda\, Q' : \mathit{Proc}_{V,\phi}.Q \,\Box\, Q'))\end{aligned}$$

To reduce it to evaluated form, we need to resolve which natural number y is, which requires an infinitary branching.

To be able to formalise such branchings as one-step reductions, we need to extend our language by two new constructs:

- In a branching that resolves which component of a sum type a term r belongs to, after following the ith branch we need a construct $\mathit{eject}_i\, r$ to explicitly take r out of the sum type and make it a term of the ith component type.
- A branching on a term of an inductive type $\mu\, Z.V$ (as on the constant y above) is in general infinitary because the type $\mu\, Z.V$ can be unfolded infinitely many times, revealing infinitely many options for what the term can be. For each unfolding, we need a construct $\mathit{unpack}\ u$ which explicitly makes a term u of type $\mu\, Z.V$ a term of the unfolded type $V[(\mu\, Z.V)/Z]$.

The typing rules for these two constructs, their denotational semantics, and the associated one-step reductions are given in Table 3.3.

A version of Proposition 3.1.1 is then provable, the main difference being that for some terms a countably infinite number of conditional symbolic transitions is needed because of branchings on terms of inductive types. The form of the conditions cond_i is more complex in that the equality tests they contain are not necessarily between evaluated forms of type X but between evaluated forms types satisfying $\mathit{AllTypeVars}, +, \times, \mu$, and that they can also contain the conditions $\mathit{From}_{i'}(r')$ which evaluate to True when r' belongs to the i'th component of its sum type. The evaluated forms are also more complex than for weakly data independent terms, although the initial symbolic events of processes in evaluated form can still be determined.

$$\frac{\Gamma \vdash r : id_1.V_1 + \cdots + id_n.V_n}{\Gamma \vdash eject_i\, r : V_i}$$

$$[\![\Gamma \vdash eject_i\, r : V_i]\!]^M_{\delta,\eta} = v, \;\; \mathit{if}\; [\![\Gamma \vdash r : id_1.V_1 + \cdots + id_n.V_n]\!]^M_{\delta,\eta} = (id_i, v)$$

$$\begin{array}{rcl} case\; r\; of\; s_1\; or\; \ldots\; or\; s_n & \xrightarrow{From_i(r)} & s_i\,(eject_i\, r) \\ eject_i\,(id_i^{id_1.V_1+\cdots+id_n.V_n}.r) & \longrightarrow & r \end{array}$$

$$\frac{\Gamma \vdash u : \mu\, Z.V}{\Gamma \vdash unpack\; u : V[(\mu\, Z.V)/Z]}$$

$$[\![\Gamma \vdash unpack\; u : V[(\mu\, Z.V)/Z]]\!]^M_{\delta,\eta} = [\![\Gamma \vdash u : \mu\, Z.V]\!]^M_{\delta,\eta}$$

$$\begin{array}{rcl} fold\; u\; s & \longrightarrow & s\, V[(\lambda\, y : (\mu\, Z.V).fold\; y\; s)/Z](unpack\; u) \\ unpack\;(in_{\mu\, Z.V}\; u) & \longrightarrow & u \end{array}$$

Table 3.3: The additional two constructs

The possibility of infinitely many conditional symbolic transitions can be eliminated by ensuring that branching on terms of inductive types is not required (in which case the additional construct *unpacku* is not needed). That can be done by for example restricting the type contexts Γ to be of the form

$$\{x_1 : U_1, \ldots, x_k : U_k, y_1 : V_1 \to W_1, \ldots, y_l : V_l \to W_l\}$$

where $U_j^{\mathit{AllTypeVars},+,\times}$, $W_j^{\mathit{AllTypeVars},+,\times}$ (see under 'Conditions on types' in Section 2.4), and the types V_j do not involve any process types, which still allows a much larger class of terms than the weakly data independent ones.

3.2 The CSP constructs

The previous section showed how the λ-calculus constructs in our language can be interpreted symbolically by conditional symbolic transitions, which is the first half of what is needed for interpreting weakly data independent processes by symbolic labelled transition systems (SLTSs). This section is devoted to the second half: showing how the CSP constructs can be interpreted by visible and invisible symbolic transitions which are the transitions through which visible and invisible events are performed.

The main part is Section 3.2.3 where we give the 'firing rules' by which the visible and invisible symbolic transitions are deduced, and preparatory to it is Section 3.2.2 where we define symbolic transitions in general. But first in Section 3.2.1 we outline how the two halves of the work are to be put together.

3.2.1 Constructing SLTSs

Given an X,Γ-WDI process P, an SLTS interpreting it can be constructed by the following procedure:

(i) The initial node is $\Gamma \vdash P$.

(ii) Proposition 3.1.1 is applied to $\Gamma \vdash P$, yielding a finite number of conditional symbolic transitions

$$\begin{array}{ccc} (\Gamma \vdash P) & [\mathit{cond}_1\rangle & (\Gamma \vdash Q_1) \\ & \vdots & \\ (\Gamma \vdash P) & [\mathit{cond}_n\rangle & (\Gamma \vdash Q_n) \end{array}$$

to some X,Γ-WDI processes Q_i which are in evaluated form.

(iii) For each Q_i, its initial visible and invisible symbolic transitions are then determined by the firing rules given in Section 3.2.3. The fact that Q_i is in evaluated form will mean either that those transitions can be determined by a single firing rule and straight from the top syntactic level of Q_i (as when Q_i is a prefixing), or that they are determined by a firing rule which constructs them from the initial transitions of the subprocesses of Q_i (as when Q_i is a parallel composition) which are themselves determined in the same way by the firing rules.

What we have been meaning by saying that the initial visible and invisible symbolic transitions are determinable for processes in evaluated form is that the procedure outlined above never requires further conditional symbolic transitions to be performed. In other words, a process in evaluated form is such that any subprocesses upon which its initial visible and invisible symbolic transitions depend are also in evaluated form, as can be seen in Table 3.2.

In contrast to the conditional symbolic transitions, Q_i can have a countably infinite number of initial visible and invisible symbolic transitions

$$\begin{array}{lll} (\Gamma \vdash Q_i) & \alpha_1 & (\Gamma'_{i,1} \vdash Q'_{i,1}) \\ (\Gamma \vdash Q_i) & \alpha_2 & (\Gamma'_{i,2} \vdash Q'_{i,2}) \\ & \vdots & \end{array}$$

The new type contexts $\Gamma'_{i,j}$ are needed instead of Γ because, as we shall see in Section 3.2.2, the visible and invisible symbolic transitions can introduce finite numbers of new variables of type X.

The step (ii) is then repeated with each $\Gamma'_{i,j} \vdash Q'_{i,j}$, which as we shall see in Proposition 3.2.1 is still X,$\Gamma'_{i,j}$-WDI.

Ways specifiers. By Proposition 2.5.2 (a), any node $\Gamma' \vdash P'$ has a unique alphabet T, but not necessarily a unique ways specifier ϕ. However, which exactly ways specifiers are used will be insignificant, and so the types $Proc_{T,\phi}$ can be omitted. ■

Constants, operations and new variables. The type contexts Γ' of the nodes in general consist of three kinds of entities: constants from the type context Γ of the initial node, operations from Γ, and variables of type X that have been introduced by the transitions. Sometimes it will be important to distinguish among them, but at other times we shall refer to all as just 'variables'. ■

Minimal type contexts. From now on, we shall assume that the type context Γ' of every node $\Gamma' \vdash P'$ is contracted to be minimal, i.e. that any variables which do not occur free in P' are omitted from Γ'. ■

Reusing nodes. The assumption that the type contexts are contracted to be minimal enables us to reuse nodes, which in turn makes it possible for SLTSs constructed by the procedure to be finite.

From now on, we shall assume that for each $\Gamma'_{i,j} \vdash Q'_{i,j}$ in the step (iii) for which a node with the same type context and the same term has been constructed previously (which includes the nodes $\Gamma'_{i',j'} \vdash Q'_{i',j'}$ such that either $i' < i$ or $i' = i$ and $j' < j$), that node is reused instead of creating a new node and repeating the step (ii) with it.

On the other hand, in order to avoid the possibility of having consecutive conditional symbolic transitions, reusing of nodes is not to be attempted in the step (ii). Consequently, it may happen that the procedure creates different nodes which have the same type contexts and the same terms, and so the nodes in the procedure should be thought of as objects labelled by type contexts and terms rather than being simply ordered pairs of type contexts and terms.

For example, the reusing of nodes makes the SLTS of the process *NondReg* from Example 2.5.1 finite, as we shall see in Example 3.3.1. See also Section 3.3.3. ■

3.2.2 Symbolic transitions

This section is devoted to defining the three kinds of symbolic transitions: conditional (which were introduced in Section 3.1.1), visible and invisible.

In the previous section we outlined a procedure for constructing SLTSs, which produces SLTSs whose nodes are labelled by weakly data independent processes and whose transitions are the symbolic transitions obtained from Proposition 3.1.1 and the firing rules in Section 3.2.3. In addition to the procedure, we shall in Section 3.3.1 define more abstractly what an SLTS is, which will also serve as a description of the structure of SLTSs that are produced by the procedure.

In that more abstract view, both nodes and transitions will be objects which are appropriately labelled. The nodes will be labelled only by type contexts, and so given two such type contexts Γ and Γ', it will be important to distinguish which symbolic transitions can consistently be used as transitions from a node labelled by Γ to a node labelled by Γ'. We shall call such symbolic transitions Γ,Γ' *symbolic transitions.*

In the same way that an SLTS can have many different nodes labelled by the same type context, it can also have many different transitions with the same labels, even between the same two nodes. That is why we shall define symbolic transitions as labelled objects, avoiding to identify them with tuples of their labels. This peculiarity, which is not present in ordinary LTSs where any number of transitions with the same label between the same two nodes are equivalent to one such transition, is due to nondeterministic selections in our visible symbolic transitions (see below).

Suppose $X \in \mathit{NEFTypeVars}$, T^{alph}, $\mathit{Free}(T) \subseteq \{X\}$, ϕ is a ways specifier with respect to T (see under 'Process types' in Section 2.4), and Γ and Γ' are type contexts satisfying the restrictions of weak data independence in X (see Section 2.7.2).

Conditional. α is a Γ,Γ' conditional symbolic transition with respect to X, T and ϕ if it is labelled by:

(i) a condition $\mathit{cond}(\alpha)$ which is either *True*, or *False*, or built from equality tests between X,Γ-WDI evaluated forms of type X (see Section 3.1.2) and the boolean connectives $\neg$, $\wedge$ and $\vee$ (see Proposition 3.1.1);

and if:

(I) $\Gamma \supseteq \Gamma'$.

As in Section 3.1.1, α is written as

$$[\mathit{cond}(\alpha)\rangle$$

Informally, α can be performed when the values assigned to the variables in Γ satisfy $\mathit{cond}(\alpha)$. (The formal meanings of the symbolic transitions will be defined in Section 3.4.)

Visible. α is a Γ,Γ' visible symbolic transition with respect to X, T and ϕ if it is labelled by:

(i) type contexts $ys(\alpha)$ and $zs(\alpha)$ in which all variables are of type X and which are disjoint from each other and from Γ;

(ii) a condition $cond(\alpha)$ which is a finite conjunction of equality tests such that the terms in the equality tests can be either X,Γ-WDI evaluated forms of type X or variables from $ys(\alpha)$ (where an empty conjunction is *True*);

(iii) a term $r(\alpha)$ which is an X,$\Gamma \cup ys(\alpha) \cup zs(\alpha)$-WDI evaluated form $H[r_1, \ldots, r_h]$ of type T (see Section 3.1.2) such that:

(iii.a) any r_i is either an X,Γ-WDI evaluated form of type X or a variable from $ys(\alpha)$ or a variable from $zs(\alpha)$;

(iii.b) each variable from $ys(\alpha)$ which is not an element of any equivalence class of $\approx$ (see below) is exactly one r_i;

(iii.c) for each equivalence class C of $\approx$ which does not contain an X,Γ-WDI evaluated form of type X, there is exactly one variable from C which is some r_i, and it is exactly one r_i;

(iii.d) for each equivalence class C of $\approx$ which contains an X,Γ-WDI evaluated form of type X, there is at least one element of C which is some r_i;

(iii.e) each variable from $zs(\alpha)$ is exactly one r_i;

where $\approx$ is the partial equivalence relation induced by $cond(\alpha)$, i.e. it is the symmetric and transitive closure of the relation

$$\{(s, s') \mid (s = s') \textit{ is a conjunct in } cond(\alpha)\}$$

and if:

(I) the variable names in $ys(\alpha)$ and $zs(\alpha)$ are the minimal ones which are not in Γ, in the sense that they are those variable names not in Γ which occur earliest in a fixed enumeration x_1, x_2, ... of all possible variable names;

(II) $\Gamma \cup ys(\alpha) \cup zs(\alpha) \supseteq \Gamma'$;

(III) the ways specifier ϕ is observed, namely if $H[r_1, \ldots, r_h]$ (see above) is of the form

$$id_i.(G_1[s_{1,1}, \ldots, s_{1,g_1}], \ldots, G_{m_i}[s_{m_i,1}, \ldots, s_{m_i,g_{m_i}}])$$

then:

(III.a) for any j and k, if $s_{j,k}$ is an X,Γ-WDI evaluated form of type X, or $s_{j,k}$ is a variable from $ys(\alpha)$ which is related by $\approx$ (see above) to an X,Γ-WDI evaluated form of type X, then $! \in \phi(id_i, j)$;

(III.b) for any j and k, if $s_{j,k}$ is a variable from $zs(\alpha)$, then $? \in \phi(id_i, j)$;

(III.c) for any j and k, if $s_{j,k}$ is a variable from $ys(\alpha)$, then $\$ \in \phi(id_i, j)$;

(III.d) for any equivalence class C of $\approx$ which contains a variable from $ys(\alpha)$, there exist j and k such that $s_{j,k}$ is an element of C and $\$ \in \phi(id_i, j)$.

The transition α is written as

$$[\$ys(\alpha)?zs(\alpha)|cond(\alpha) \bullet r(\alpha)\rangle$$

Informally, α is performed by

- a nondeterministic choice over all possible assignments of values to the variables $ys(\alpha)$, then
- an external choice over all assignments of values to the variables $zs(\alpha)$ for which the condition $cond(alpha)$ is satisfied, and then
- communicating the visible event $r(\alpha)$,

where we should stress that the nondeterministic choice is always performed before the external one, regardless of how the variables from $ys(\alpha)$ and $zs(\alpha)$ are interleaved in the term $r(\alpha)$.

The form of α and the recipe for performing it are more general than necessary for the language in this thesis, in order to be usable for more complex languages such as CSP$_M$ [Sca92, Sca98, Ros98a]. In particular:

- Since each variable from $zs(\alpha)$ occurs exactly once in $r(\alpha)$, it was not necessary to have $zs(\alpha)$ as a separate label.
- Since the condition $cond(\alpha)$ does not contain any variables from $zs(\alpha)$, its outcome when performing α is known before the external choice. If it is *True*, the external choice is over all possible assignments, and if it is *False*, the external choice reduces to $STOP_T$.

Together with the minimality of type contexts of nodes (see Section 3.2.1), the minimality of variable names in $ys(\alpha)$ and $zs(\alpha)$ enables SLTSs to be finite by reusing variable names: if a node N has been reached from the initial node and if a variable x which was either in the type context of the initial node or was introduced by one of the transitions on the path to N no longer appears in the type context $\Gamma(N)$ of N, then x can be reused in $ys(\alpha)$ or $zs(\alpha)$ of a transition α from N. (See Example 3.3.1 and Section 3.3.3.)

When a node has more than one visible symbolic transition, they are to be thought of as joined by binary external choices. In other words, the nondeterministic choices of all its visible symbolic transitions are to be performed before any of their external choices, so that the transitions form the refusal sets (see Section 2.1.2) of the node together. (For a formalisation of this, see Section 3.4.)

That is why we are avoiding the identification of symbolic transitions with tuples of their labels. More precisely, two visible symbolic transitions which are from the same node and all of whose labels are the same are not equivalent to one because their nondeterministic choices are to be performed independently and thus they can produce different refusal sets than one transition.

We shall call the variables in $ys(\alpha)$ 'nondeterministic selections' and the variables in $zs(\alpha)$ 'inputs'. When using the notation $H[r_1, \ldots, r_h]$ for $r(\alpha)$, we shall call the r_i which are X,Γ-WDI evaluated forms of type X 'outputs'.

Invisible. α is a Γ,Γ' invisible symbolic transition with respect to X, T and ϕ if it is labelled by:

(i) a type context $ys(\alpha)$ in which all variables are of type X and which is disjoint from Γ;

(ii) a condition $cond(\alpha)$ which, as in the visible symbolic transitions, is a finite conjunction of equality tests such that the terms in the equality tests can be either X,Γ-WDI evaluated forms of type X or variables from $ys(\alpha)$ (where an empty conjunction is *True*);

and if:

(I) as in the visible symbolic transitions, the variable names in $ys(\alpha)$ are the minimal ones which are not in Γ;

(II) $\Gamma \cup ys(\alpha) \supseteq \Gamma'$.

The transition α is written as

$$[\sqcap\, ys(\alpha) | cond(\alpha) \;\bullet\; \tau\rangle$$

Informally, α is performed as a nondeterministic choice over all assignments of values to the variables $ys(\alpha)$ for which the condition $cond(\alpha)$ is satisfied. If there are no assignments which satisfy $cond(\alpha)$, that has the same effect as the transition not being present.

For any symbolic transition α, let

$$Vars(\alpha) = \begin{cases} \{\}, \text{ if } \alpha \text{ is conditional} \\ ys(\alpha) \cup zs(\alpha), \text{ if } \alpha \text{ is visible} \\ ys(\alpha), \text{ if } \alpha \text{ is invisible} \end{cases}$$

3.2.3 Firing rules

In this section we shall complete our definition of the symbolic operational semantics of weakly data independent processes by giving the 'firing rules'. They are the deduction rules which can be used to determine the initial visible and invisible symbolic transitions of any X,Γ-WDI process in evaluated form.

Only processes in evaluated form. A firing rule shows how an initial visible or invisible transition of a process is deduced from appropriate initial visible or invisible transitions of its subprocesses. As we have remarked in Section 3.2.1, the evaluated forms of process types (see Section 3.1.2) were defined in such a way that if the process in the conclusion of the firing rule is in evaluated form, then all the subprocesses in the assumptions also are. Consequently, on the left-hand side of any transition in any firing rule we shall need to consider only processes in evaluated form. ■

Assumptions. Recalling the assumption that type contexts of all nodes are contracted to be minimal (see Section 3.2.1), we shall for brevity adopt the following two conventions:

- The type contexts of the processes on the right-hand sides of the transitions will be omitted, and assumed to be defined as minimal. More precisely, in any transition

$$(\Gamma \vdash P) \quad \alpha \quad P'$$

the type context of P' will be assumed to be defined as

$$\Gamma' = (\Gamma \cup Vars(\alpha)) \restriction Free(P')$$

- If Q' is a subprocess of a process $\Gamma \vdash Q$, we shall write $\Gamma \vdash Q'$ instead of the more cumbersome $(\Gamma \restriction \mathit{Free}(Q')) \vdash Q'$.

Similarly, when writing transitions

$$(\Gamma \vdash P) \quad [\$ys?zs|cond \bullet r\rangle \quad P'$$

$$(\Gamma \vdash P) \quad [\sqcap\, ys|cond \bullet \tau\rangle \quad P'$$

we shall assume that the names of the variables in the sets ys and zs are if necessary renamed not to be in Γ and to be the minimal such (see Section 3.2.2).

The final assumption is more subtle. Recalling that we defined the symbolic transitions to be objects α which are appropriately labelled and that we avoided to identify them with tuples of their labels (see Section 3.2.2), we shall assume that the firing rules are injective on transitions as objects. More precisely, if by a firing rule transitions α and α' are deduced from transitions $\alpha_1, \ldots, \alpha_n$ and $\alpha'_1, \ldots, \alpha'_n$ (respectively) and if $\alpha_i \neq \alpha'_i$ for some i then it must be that $\alpha \neq \alpha'$, where two transitions are considered unequal if they are not the same object although they might have the same labels. ■

The firing rules are the following:

STOP. The processes $\{\} \vdash STOP_T$ have no transitions, and so there are no firing rules associated with them.

Outputs, inputs and nondeterministic selections. A prefixing $\Gamma \vdash e \longrightarrow P$ where $e = id\ it_1 \ldots it_m$ will be interpreted by a two-stage branching. Informally, the first stage will consist of choosing 'concrete parts' for the nondeterministic selections, and the second stage will consist of choosing 'concrete parts' for the inputs and 'parts of type X' for the nondeterministic selections and for the inputs.

Consider a nondeterministic selection $\$x : U$ in the prefix e. To interpret it by visible or invisible symbolic transitions, which are capable of performing nondeterministic choices only of values of type X, we have to decompose it into

(i) a nondeterministic choice of an evaluated form $H[y_1, \ldots, y_h]$ of type U (the 'concrete part') where the y_k are fresh variables (see Section 3.1.2), followed by

(ii) a nondeterministic choice of values for the variables y_k (the 'parts of type X').

We shall then be able to perform the choice (i) by a branching consisting of invisible symbolic transitions with no variables, and after each of those transitions the corresponding choice (ii) will be performable within a visible symbolic transition.

More precisely, the first stage of the interpretation of $\Gamma \vdash e \longrightarrow P$ will be a branching consisting of invisible symbolic transitions each of which chooses an evaluated form as in (i) for every nondeterministic selection in e.

An alternative would be to perform both choices (i) and (ii) in the first stage, by invisible symbolic transitions with variables. However, that would amount to rewriting the nondeterministic selections in e by replicated nondeterministic choices, and would thus make having nondeterministic selections as a separate construct pointless (see under 'Outputs, inputs and nondeterministic selections' in Section 2.5 and about the condition $(\mathbf{RecCho}_X)$ in Section 3.5).

To formalise the first stage, let

$$\mathit{EvForms}_{\Gamma'}(U)$$

for a type U satisfying $U^{\mathit{AllTypeVars},+,\times,\mu}$ (which is weaker than U^{comm}: see under 'Conditions on types' in Section 2.4) and having at most the free type variable X be the set of all evaluated forms $H[y_1, \ldots, y_h]$ of type U in which the y_k are variables that are not in Γ'. More precisely, the names $y_1, \ldots, y_h$ are picked to be minimal with respect to not being in Γ', so that $\mathit{EvForms}_{\Gamma'}(U)$ contains at most one evaluated form with any particular shell H. Then let

$$\mathit{Expand}_{\$}(\Gamma \vdash e \longrightarrow P)$$

be the set of all $e' \longrightarrow P'$ that are obtained from $e \longrightarrow P$ as follows:

- for each it_j which is a nondeterministic selection $\$x_j : U_j$, an evaluated form in $\mathit{EvForms}_{\Gamma'_j}(U_j)$ is substituted for x_j in both e and P, where Γ'_j is defined as Γ enlarged by the variables in the evaluated forms that were substituted for the $x_{j'}$ which are earlier in e, and
- the variables of the inputs in e are renamed if necessary to avoid clashes with variables in the substituted evaluated forms.

For any such $e' \longrightarrow P'$, let

$$\mathit{NewVars}_{\$}(e')$$

be the type context consisting of all the variables in the substituted evaluated forms: they are the variables of type X whose values remain to be nondeterministically chosen to complete the nondeterministic selections in e.

Since they contain 'constructs' of the form $\$H[y_1, \ldots, y_h] : U$, the prefixings $e' \longrightarrow P'$ in general do not belong to our language. However, when working with the firing rules, we shall extend our language by such prefixings. They have the same denotational semantics as the prefixings $e \longrightarrow P$ that they were obtained from, except that each nondeterministic selection $\$H[y_1, \ldots, y_h] : U$ ranges not over the whole type U but only over values that the evaluated form $H[y_1, \ldots, y_h]$ can assume. Moreover, we shall regard the $e' \longrightarrow P'$ as being in evaluated form. It is straightforward to check that with these two extensions the developments in Sections 3.1 and 3.2 remain valid.

The first stage is then generated by the firing rule

$$\frac{}{(\Gamma \vdash e \longrightarrow P) \quad [\sqcap \{\} | \mathit{True} \bullet \tau\rangle \quad (e' \longrightarrow P')} \left(\begin{array}{c} (e' \longrightarrow P') \in \\ \mathit{Expand}_{\$}(\Gamma \vdash e \longrightarrow P) \end{array} \right)$$

Since the processes $\Gamma \vdash e' \longrightarrow P'$ are regarded as being in evaluated form, we can can immediately proceed with the second stage of the interpretation of $\Gamma \vdash e \longrightarrow P$, that is without first applying Proposition 3.1.1 (see Section 3.2.1).

The second stage will consist of visible symbolic transitions. For any $e' \longrightarrow P'$ produced by the first stage, there will be one transition for every possible choice of the evaluated forms for the inputs in e' (which are the inputs in e), and in each transition the interpretation of the prefixing will be completed by performing the choices of values for the variables introduced in the evaluated forms in both stages.

We can formalise this in the same way as we formalised the first stage, by letting

$$\mathit{Expand}_?(\Gamma \vdash e' \longrightarrow P')$$

be the set of all $e'' \longrightarrow P''$ that can be obtained from $e' \longrightarrow P'$ by substituting the evaluated forms for the inputs. The only difference is that the variables

$$\mathit{NewVars}_?(e'')$$

introduced in the evaluated forms now need to be disjoint both from Γ and from $\mathit{NewVars}_\$(e')$.

The second stage is then generated by the firing rule

$$\cfrac{}{\begin{array}{c}(\Gamma \vdash e' \longrightarrow P') \\ \left[\begin{array}{c}\$\mathit{NewVars}_\$(e')?\mathit{NewVars}_?(e'')|\,\mathit{True} \\ \bullet\ \mathit{ToTerm}(e'')\end{array}\right\rangle \\ P''\end{array}} \left(\begin{array}{c}(e'' \longrightarrow P'') \in \\ \mathit{Expand}_?(\Gamma \vdash e' \longrightarrow P')\end{array}\right)$$

where $\mathit{ToTerm}(e'')$ is the term corresponding to e'': if $e'' = id\, it''_1 \ldots it''_m$, then $\mathit{ToTerm}(e'')$ is $id.(it''_1, \ldots, it''_m)$ in which any $!r$, $?r : U$ or $\$r : U$ has been replaced by just r.

However, the meaning of prefixing in the denotational semantics (see Section 2.5) was defined to correspond to all nondeterministic selections being performed before any inputs, and so the symbolic operational semantics of prefixing we have just given is correct only when it does not perform any nondeterministic choices after external ones. That is the case when either $\mathit{NewVars}_\$(e') = \{\}$ for all $e' \longrightarrow P'$ or when $\mathit{Expand}_?(\Gamma \vdash e' \longrightarrow P')$ has at most one element for all $e' \longrightarrow P'$, which is what the side condition $\mathit{Typ}(it_1\ \ldots\ it_{m_i})$ in the typing rule for prefixing (see Section 2.5) ensures.

As an example of how the two firing rules are used, consider a prefixing

$$\{f : (X \times \mathit{Num}) \rightarrow X, x : X\} \vdash c\,?v : (X \times X)\ !f(x,5)\ \$w : \mathit{List}(X) \longrightarrow P$$

The purpose of the first rule is then to choose an evaluated form for the nondeterministic selection $\$w : \mathit{List}(X)$, which amounts to choosing a length for a list. Indeed, for each length $h \in \mathbb{N}$, the rule produces a transition

$$\begin{array}{c}\left(\begin{array}{c}\{f : (X \times \mathit{Num}) \rightarrow X, x : X\} \vdash \\ c\,?v : (X \times X)\ !f(x,5)\ \$w : \mathit{List}(X) \longrightarrow P\end{array}\right) \\ [\sqcap\,\{\}|\,\mathit{True}\ \bullet\ \tau\rangle \\ (c\,?v : (X \times X)\ !f(x,5)\ \$[y_1, \ldots, y_h] : \mathit{List}(X) \longrightarrow P[[y_1, \ldots, y_h]/w])\end{array}$$

where the variable names y_1, y_2, ... are picked to be the minimal ones with respect to being distinct from f and x.

The type $(X \times X)$ of the input is a product of type variables (in accordance with the *Typ* condition), and so there is only one evaluated form that can be chosen for it in the second rule. Thus for any process produced by the first rule, the second rule produces a transition

$$\begin{array}{c} \left(\begin{array}{c} \{f : (X \times \mathit{Num}) \rightarrow X, x : X\} \vdash \\ c\,?v : (X \times X)\,!f(x,5)\ \$[y_1, \ldots, y_h] : \mathit{List}(X) \longrightarrow P[[y_1, \ldots, y_h]/w] \end{array} \right) \\ \left[\begin{array}{c} \$\{y_1 : X, \ldots, y_h : X\}?\{z_1 : X, z_2 : X\} | \mathit{True} \\ \bullet\ c.((z_1, z_2), f(x,5), [y_1, \ldots, y_h]) \end{array} \right\rangle \\ P[[y_1, \ldots, y_h]/w][(z_1, z_2)/v] \end{array}$$

where the variable names z_1 and z_2 are now picked to be the minimal ones with respect to being distint from f, x and the y_k.

External choice. Each component of an external choice is free to perform any of its initial transitions, where a visible transition resolves the choice in favour of that component but an invisible transition does not:

$$\frac{(\Gamma \vdash P)\quad [\$ys?zs | \mathit{cond}\ \bullet\ r\rangle\quad P'}{(\Gamma \vdash P \mathbin{\Box} Q)\quad [\$ys?zs | \mathit{cond}\ \bullet\ r\rangle\quad P'}$$

$$\frac{(\Gamma \vdash P)\quad [\sqcap\ ys | \mathit{cond}\ \bullet\ \tau\rangle\quad P'}{(\Gamma \vdash P \mathbin{\Box} Q)\quad [\sqcap\ ys | \mathit{cond}\ \bullet\ \tau\rangle\quad P' \mathbin{\Box} Q}$$

The rules for the right-hand component are symmetrical.

Nondeterministic choice. A nondeterministic choice is interpreted by two invisible transitions, to each of its components:

$$\frac{}{(\Gamma \vdash P \sqcap Q)\quad [\sqcap\ \{\} | \mathit{True}\ \bullet\ \tau\rangle\quad P}$$

The rule for the right-hand component is again symmetrical.

Replicated external choice. A replicated external choice in our language does not have to be interpreted directly, since as we have shown in Section 2.5 it can be rewritten in terms of binary external choices.

Replicated nondeterministic choice. A replicated nondeterministic choice is interpreted by first choosing a 'concrete part' by branching and then choosing the 'parts of type X' within the invisible symbolic transitions that the branching consists of. This is the same way that inputs were interpreted in the second firing rule for prefixing, except that invisible transitions are now used instead of the visible ones.

If the replicated nondeterministic choice to be interpreted is $\Gamma \vdash \bigsqcap_{x:U} P$ then the appropriate 'concrete parts' are the evaluated forms in the set $\mathit{EvForms}_\Gamma(U)$ (see under 'Outputs, inputs and nondeterministic selections' above), which gives us the firing rule

$$\frac{}{\begin{array}{c} (\Gamma \vdash \bigsqcap_{x:U} P) \\ [\sqcap\ \{y_1 : X, \ldots, y_h : X\} | \mathit{True}\ \bullet\ \tau\rangle \\ P[H[y_1, \ldots, y_h]/x] \end{array}} \left(\begin{array}{c} H[y_1, \ldots, y_h] \in \\ \mathit{EvForms}_\Gamma(U) \end{array} \right)$$

Parallel composition. To interpret a parallel composition, we need firing rules for three kinds of transitions:

- visible transitions which are synchronisations between visible transitions of the components whose events agree and belong to the synchronisation set,
- visible transitions of one of the components whose events do not belong to the synchronisation set, and
- invisible transitions of one of the components.

The firing rule for synchronisations is:

$$\frac{\begin{array}{c}(\Gamma \vdash P)\quad [\$ys?zs|\mathit{cond}\ \bullet\ H[r_1,\ldots,r_h]\rangle\quad P' \\ (\Gamma \vdash Q)\quad [\$ys'?zs'|\mathit{cond}'\ \bullet\ H[r'_1,\ldots,r'_h]\rangle\quad Q'\end{array}}{\begin{array}{c}(\Gamma \vdash P \underset{S}{\parallel} Q) \\ \left[\begin{array}{c}\$(ys \cup ys')?(zs \setminus \mathit{Domain}(\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]})) \\ |(\mathit{Match}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]} \wedge \mathit{cond} \wedge \mathit{cond}') \\ \bullet\ \mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}(H[r_1,\ldots,r_h])\end{array}\right\rangle \\ (\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}(P') \underset{S}{\parallel} \mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}(Q'))\end{array}}\ H[\cdot,\ldots,\cdot] \in S$$

where:

- The events $H[r_1,\ldots,r_h]$ and $H[r'_1,\ldots,r'_h]$ of the transitions of P and Q (see under 'Visible' in Section 3.2.2) are required to have the same shell H and thus the same number h of components because otherwise a synchronisation would not be possible.
- The side condition $H[\cdot,\ldots,\cdot] \in S$ is the other requirement that needs to be satisfied for the firing rule to be applicable: it requires that the two events belong to the synchronisation set S. This does not generate any symbolic conditions but is determinable from only the shell $H[\cdot,\ldots,\cdot]$ and the set S because $P \underset{S}{\parallel} Q$ is in evaluated form and so S is (see Section 3.1.2) and because sets which are defined in terms of specific values of type X are not expressible in our language (see Section 2.5).
- The variables ys' are renamed if necessary to make the union $ys \cup ys'$ disjoint.
- $\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}$ is a substitution which substitutes all inputs in the transition of Q by whatever is in their place in the transition of P and does the opposite for all inputs in the transition of P for which what occurs in their place in the transition of Q is not an input. More formally, $\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}$ substitutes each r'_j such that $r'_j \in zs'$ by r_j and each r_i such that $r_i \in zs$ and $r'_i \notin zs'$ by r'_i.
  $\mathit{Domain}(\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]})$ is the set of all inputs for which $\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}$ is substituting.
- $\mathit{Match}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}$ is a symbolic condition on outputs and nondeterministic selections that has to be satisfied for a synchronisation to be possible. It is the conjunction of all equality tests $r_i = r'_i$ in which r_i and r'_i are outputs or nondeterministic selections, i.e. such that $r_i \notin zs$ and $r'_i \notin zs'$.

(In the condition $\mathit{Match}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]} \wedge \mathit{cond} \wedge \mathit{cond}'$, the substitution $\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}$ does not need to be applied to *cond* and *cond'* because they do not contain inputs: see under 'Visible' in Section 3.2.2.)

For example, if $\Gamma = \{f : (X \times \mathit{Num}) \to X, x : X\}$, if the transitions of P and Q are

$$\begin{array}{r}
(\Gamma \vdash P) \left[\begin{array}{c} \$\{y : X\}?\{z_1 : X, z_2 : X\}|(f(x,1) = f(x,5)) \\ \bullet\ c.(z_1, [f(x,1), y, z_2], 7) \end{array} \right\rangle P' \\
(\{x : X\} \vdash Q) \left[\begin{array}{c} \$\{y' : X\}?\{z' : X\}|(y' = x) \\ \bullet\ c.(y', [x, x, z'], 7) \end{array} \right\rangle Q'
\end{array}$$

and if the shell $c.(\cdot, [\cdot, \cdot, \cdot], 7)$ is in the synchronisation set S, then

$$\begin{array}{rcl}
\mathit{Match}^{?\{z_1:X,z_2:X\}\,.\,(c.(z_1,[f(x,1),y,z_2],7))}_{?\{z':X\}\,.\,(c.(y',[x,x,z'],7))} & = & (f(x,1) = x \ \wedge\ y = x) \\
\mathit{Subst}^{?\{z_1:X,z_2:X\}\,.\,(c.(z_1,[f(x,1),y,z_2],7))}_{?\{z':X\}\,.\,(c.(y',[x,x,z'],7))} & = & [z_2/z', y'/z_1]
\end{array}$$

and so the rule produces a transition

$$\begin{array}{c}
(\Gamma \vdash P \mathbin{\underset{S}{\|}} Q) \\
\left[\begin{array}{c} \$\{y : X, y' : X\}?\{z_2 : X\} \\ |(f(x,1) = x \ \wedge\ y = x \ \wedge\ f(x,1) = f(x,5) \ \wedge\ y' = x) \\ \bullet\ c.(y', [f(x,1), y, z_2], 7) \end{array} \right\rangle \\
(P'[y'/z_1] \mathbin{\underset{S}{\|}} Q'[z_2/z'])
\end{array}$$

(which is then of course subject to the assumptions about the minimality of type contexts and variable names, as stated at the beginning of this section).

The firing rules for transitions which are transitions of one of the components are much simpler: the parallel composition then acts in the same way as external choice does for invisible transitions. We give only the rules for P, the rules for Q being symmetric:

$$\frac{(\Gamma \vdash P) \quad [\$ys?zs|\mathit{cond} \ \bullet\ H[r_1, \ldots, r_h]\rangle \quad P'}{(\Gamma \vdash P \mathbin{\underset{S}{\|}} Q) \quad [\$ys?zs|\mathit{cond} \ \bullet\ H[r_1, \ldots, r_h]\rangle \quad P' \mathbin{\underset{S}{\|}} Q} H[\cdot, \ldots, \cdot] \notin S$$

$$\frac{(\Gamma \vdash P) \quad [\sqcap\ ys|\mathit{cond} \ \bullet\ \tau\rangle \quad P'}{P \mathbin{\underset{S}{\|}} Q \quad [\sqcap\ ys|\mathit{cond} \ \bullet\ \tau\rangle \quad P' \mathbin{\underset{S}{\|}} Q}$$

Hiding. The hiding construct turns visible transitions whose events are to be hidden into invisible transitions, but does not affect other visible transitions or any invisible transitions:

$$\frac{(\Gamma \vdash P) \quad [\$ys?zs|\mathit{cond} \ \bullet\ H[r_1, \ldots, r_h]\rangle \quad P'}{(\Gamma \vdash P \setminus S \quad [\sqcap\ (ys \cup zs)|\mathit{cond} \ \bullet\ \tau\rangle \quad P' \setminus S} H[\cdot, \ldots, \cdot] \in S$$

$$\frac{(\Gamma \vdash P) \quad [\$ys?zs|\mathit{cond} \ \bullet\ H[r_1, \ldots, r_h]\rangle \quad P'}{(\Gamma \vdash P \setminus S \quad [\$ys?zs|\mathit{cond} \ \bullet\ H[r_1, \ldots, r_h]\rangle \quad P' \setminus S} H[\cdot, \ldots, \cdot] \notin S$$

$$\frac{(\Gamma \vdash P) \quad [\sqcap\ ys|\mathit{cond} \ \bullet\ \tau\rangle \quad P'}{(\Gamma \vdash P \setminus S \quad [\sqcap\ ys|\mathit{cond} \ \bullet\ \tau\rangle \quad P' \setminus S}$$

Recursion. Whenever a recursion is unfolded, an invisible symbolic transition with no variables is performed:

$$\frac{}{\begin{array}{c}(\Gamma \vdash (\mu\, p : (U \rightarrow \mathit{Proc}_{T,\phi}).P)\; r) \\ [\sqcap\, \{\} \,|\, \mathit{True}\; \bullet\; \tau\rangle \\ (P[(\mu\, p : (U \rightarrow \mathit{Proc}_{T,\phi}).P)/p]\; r)\end{array}}$$

The following proposition states a number of properties of the transitions and processes produced by the firing rules:

Proposition 3.2.1 *If a process $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$ is X,Γ-WDI and in evaluated form, then for any*

$$(\Gamma \vdash P) \quad \alpha \quad (\Gamma' \vdash P')$$

deducible by the firing rules, we have that

- *α is a Γ,Γ' symbolic transition with respect to X, T and ϕ,*
- *a type judgement $\Gamma' \vdash P' : \mathit{Proc}_{T,\phi'}$ with $\phi \supseteq \phi'$ (see under 'Some conventions' at the beginning of Section 2.5.2) is derivable, and*
- *P' is X,Γ'-WDI.*

Proof. By induction on the firing rules. ■

3.3 SLTSs and ASLTSs

In the previous two sections, we have given symbolic operational semantics to weakly data independent processes, or more precisely how to interpret any X,Γ-WDI process by a symbolic labelled transition system (SLTS).

In this section, we shall define SLTSs in general, then introduce an alternative form of SLTSs which are for short called ASLTSs, and finally state a few results about finiteness of SLTSs and ASLTSs that are produced by the symbolic operational semantics.

3.3.1 SLTSs

Definition 3.3.1 Suppose $X \in \mathit{NEFTypeVars}$, T^{alph}, $\mathit{Free}(T) \subseteq \{X\}$, and ϕ is a ways specifier with respect to T (see under 'Process types' in Section 2.4). $\mathcal{S}$ is said to be an SLTS with respect to X, T and ϕ if and only if it is a tuple

$$(\mathcal{N}, N_0, \mathcal{T}, \mathit{source}, \mathit{target})$$

such that:

(i) $\mathcal{N}$ is a set of symbolic nodes, where a symbolic node is an object N labelled by a type context

$$\mathit{Context}(N)$$

which satisfies the restrictions of weak data independence in X (see Section 2.7.2).

(ii) $N_0 \in \mathcal{N}$ and it is called 'the initial node'.

(iii) $\mathcal{T}$ is a set of symbolic transitions with respect to X, T and ϕ (see Section 3.2.2).

(iv) For any $\alpha \in \mathcal{T}$, $\mathit{source}(\alpha)$ and $\mathit{target}(\alpha)$ specify the source and target symbolic nodes of α, which must be such that α is a $\mathit{Context}(\mathit{source}(\alpha))$,$\mathit{Context}(\mathit{target}(\alpha))$ symbolic transition (see Section 3.2.2).

(v) Any $N \in \mathcal{N}$ can be reached from N_0, and has countably many symbolic transitions.

(vi) Any $N \in \mathcal{N}$ either has no conditional symbolic transitions, or it has only conditional symbolic transitions, in which case there are finitely many of them and they disjointly partition *True*, in the sense that any assignment of values to the variables in $\mathit{Context}(N)$ satisfies exactly one of them.

(vii) There are no two consecutive conditional symbolic transitions.

■

Notation. As we did in Sections 3.1 and 3.2, we shall write $N\alpha$ to mean that $\mathit{source}(\alpha) = N$, and $\alpha N'$ to mean that $\mathit{target}(\alpha) = N'$. ■

The SLTS of a given process. For any X,Γ-WDI process $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$, the procedure outlined in Section 3.2.1 indeed produces an SLTS with respect to X, T and ϕ, which we shall call

$$\mathcal{S}_{\Gamma \vdash P}$$

It is the interpretation of $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$ in the symbolic operational semantics. ■

Example 3.3.1 For example, recall the process *NondReg* from Example 2.5.1, which is data independent in X with no constants. Its SLTS $\mathcal{S}_{\{\} \vdash \mathit{NondReg}}$ is shown in Figure 3.1, where the sets displayed within the nodes N_i are their type contexts $\mathit{Context}(N_i)$.

In fact, the SLTS shown is not exactly the $\mathcal{S}_{\{\} \vdash \mathit{NondReg}}$ obtained by following the procedure in Section 3.2.1, but a simplified version of it. Namely, for any transition of the form $[\mathit{True}\rangle$ or $[\sqcap\ \{\}|\mathit{True}\ \bullet\ \tau\rangle$ which is the only transition of its source node, we have omitted it and identified its source and target nodes. These transformations are valid because they preserve the denotational semantic values corresponding to the SLTS (see Section 3.4.3).

The other thing to observe is that the SLTS is finite. The node N_3 corresponds to the process $\mathit{NondReg}'(x) \sqcap \mathit{NondReg}'(y)$ (see Example 2.5.1) and thus its type context contains both x and y. Its right-hand transition represents choosing the right-hand component of the choice, and so the node N_4 reached by it corresponds to $\mathit{NondReg}'(y)$, meaning that its type context is by the assumption of minimality contracted to contain only y. This in turn means that by the assumption of minimality of names of fresh variables, we can reuse x in the left-hand transition of N_4. In this way, instead of introducing a new variable name in each input, we end up with x and y being sufficient. More precisely, the part of the SLTS consisting of N_4, N_5 and N_6 ends up being the same as the part consisting of N_1, N_2 and N_3 except that x and y are swapped, and the key is that instead of creating a new node like we did by the transition from N_3 to N_4, the corresponding transition from N_6 can go back to N_1. ■

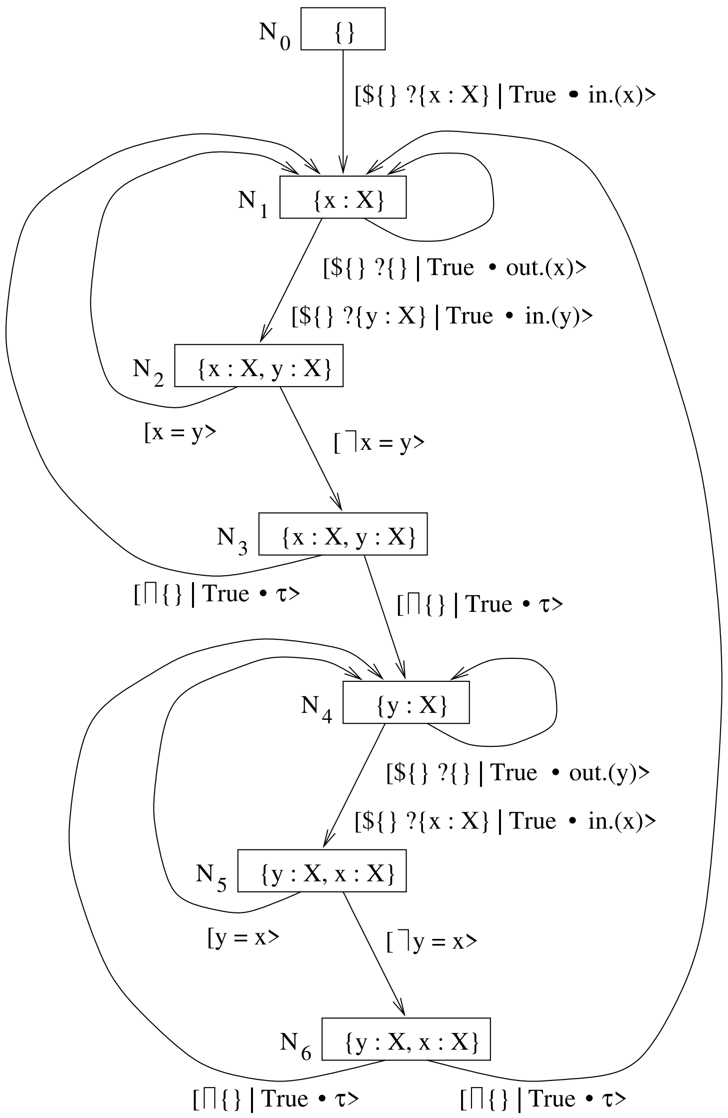


Figure 3.1: $\mathcal{S}_{\{\}\vdash NondReg}$

3.3.2 ASLTSs

It will sometimes be more convenient to work with SLTSs of an alternative form, which for short we call ASLTSs.

Alternative symbolic transitions. The difference between ASLTSs and SLTSs is that, instead of having separate conditional transitions as in SLTSs, any visible or invisible transition in an ASLTS is prefixed by a condition which has to be satisfied for the transition to be performed.

We shall call the transitions in ASLTSs 'alternative symbolic transitions'. If $X \in \mathit{NEFTypeVars}$, T^{alph}, $\mathit{Free}(T) \subseteq \{X\}$, ϕ is a ways specifier with respect to T (see under 'Process types' in Section 2.4), and Γ and Γ' are type contexts satisfying the restrictions of weak data independence in X (see Section 2.7.2), then a Γ,Γ' alternative symbolic transition α with respect to X, T and ϕ is written as:

Visible. $[\mathit{guard}(\alpha) \ \Rightarrow\ \$\mathit{ys}(\alpha)?\mathit{zs}(\alpha)|\mathit{cond}(\alpha)\ \bullet\ r(\alpha)\rangle$

Invisible. $[\mathit{guard}(\alpha)\ \Rightarrow \sqcap\ \mathit{ys}(\alpha)|\mathit{cond}(\alpha)\ \bullet\ \tau\rangle$

where:

- $\mathit{guard}(\alpha)$ is either *True*, or *False*, or built from equality tests between X,Γ-WDI evaluated forms of type X (see Section 3.1.2) and the boolean connectives $\neg$, $\wedge$ and $\vee$;
- the remaining labels are as in an ordinary Γ,Γ' visible or invisible symbolic transition with respect to X, T and ϕ (see Section 3.2.2).

Informally, provided the condition $\mathit{guard}(\alpha)$ is satisfied, α is performed in the same way as an ordinary symbolic transition which is labelled by the rest of the labels.

For any alternative symbolic transition α, let

$$\mathit{Vars}(\alpha) = \begin{cases} \mathit{ys}(\alpha) \cup \mathit{zs}(\alpha), & \text{if } \alpha \text{ is visible} \\ \mathit{ys}(\alpha), & \text{if } \alpha \text{ is invisible} \end{cases}$$

■

Definition 3.3.2 Suppose $X \in \mathit{NEFTypeVars}$, T^{alph}, $\mathit{Free}(T) \subseteq \{X\}$, and ϕ is a ways specifier with respect to T (see under 'Process types' in Section 2.4). $\mathcal{AS}$ is said to be an ASLTS with respect to X, T and ϕ if and only if it is a tuple

$$(\mathcal{N}, N_0, \mathcal{T}, \mathit{source}, \mathit{target})$$

such that:

(i) $\mathcal{N}$ is a set of symbolic nodes, where a symbolic node is an object N labelled by a type context

$$\mathit{Context}(N)$$

which satisfies the restrictions of weak data independence in X.

(ii) $N_0 \in \mathcal{N}$ and it is called 'the initial node'.

(iii) $\mathcal{T}$ is a set of alternative symbolic transitions with respect to X, T and ϕ.

(iv) For any $\alpha \in \mathcal{T}$, *source*(α) and *target*(α) specify the source and target symbolic nodes of α, which must be such that α is a *Context*(*source*(α)),*Context*(*target*(α)) alternative symbolic transition.

(v) Any $N \in \mathcal{N}$ can be reached from N_0, and has countably many alternative symbolic transitions.

(vi) For any $N \in \mathcal{N}$, there are finitely many different *guard*(α) for $\alpha \in \mathcal{T}$ and *source*$(\alpha) = N$. (However, they do not need to be disjoint nor to partition *True*.)

■

Notation. As with SLTSs, we shall write $N\alpha$ to mean that *source*$(\alpha) = N$, and $\alpha N'$ to mean that *target*$(\alpha) = N'$. ■

Transforming SLTSs into ASLTSs and vice-versa. Any SLTS (see Definition 3.3.1) $\mathcal{S}$ with respect to X, T and ϕ can be transformed into an ASLTS $\overline{\mathcal{S}}^{AS}$ with respect to X, T and ϕ by forming alternative symbolic transitions from conditional symbolic transitions followed by visible or invisible symbolic transitions.

Conversely, any ASLTS $\mathcal{AS}$ with respect to X, T and ϕ can be transformed into an SLTS $\overline{\mathcal{AS}}^{S}$ with respect to X, T and ϕ by decomposing alternative symbolic transitions into conditional and visible or invisible symbolic transitions.

The transformations produce an equivalent ASLTS or SLTS in the sense that they preserve the denotational semantic values (see Section 3.4). Thus ASLTSs and SLTSs have the same expressive power. ■

The ASLTS of a given process. The obvious way to interpret an X,Γ-WDI process $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$ by an ASLTS is to transform its SLTS $\mathcal{S}_{\Gamma \vdash P}$ as outlined above. However, the resulting ASLTS would be essentially the same as $\mathcal{S}_{\Gamma \vdash P}$, and there would be little point in working with it rather than with $\mathcal{S}_{\Gamma \vdash P}$.

To construct an ASLTS that is potentially genuinely simpler than $\mathcal{S}_{\Gamma \vdash P}$, we use a procedure that differs from the one for constructing $\mathcal{S}_{\Gamma \vdash P}$ (see Section 3.2.1) in the following ways:

(a) We assume that in any equality test

$$(r = s) \rightsquigarrow u, v$$

in $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$, the type of u and v is a process type.

(b) We consider any

$$(r^\sharp = s^\sharp) \rightsquigarrow Q^\sharp, R^\sharp$$

to be an evaluated form.

This means that equality tests are no longer reduced in step (ii) of the procedure, but are postponed until step (iii) in which they will require their own firing rules. Step (ii) therefore always produces a single evaluated form, and can thus be performed without generating any transitions.

(c) Equality tests are interpreted by four new firing rules, to cover the cases of equality and inequality, and of visible and invisible transitions:

$$\frac{(\Gamma \vdash Q)\quad [\mathit{guard} \;\Rightarrow\; \$ys?zs|\mathit{cond} \;\bullet\; r\rangle \quad Q'}{\begin{array}{c}(\Gamma \vdash ((r = s) \rightsquigarrow Q, R))\\ [(\mathit{guard} \;\wedge\; r = s) \;\Rightarrow\; \$ys?zs|\mathit{cond} \;\bullet\; r\rangle\\ Q'\end{array}}$$

$$\frac{(\Gamma \vdash Q)\quad [\mathit{guard} \;\Rightarrow\sqcap\; ys|\mathit{cond} \;\bullet\; \tau\rangle \quad Q'}{\begin{array}{c}(\Gamma \vdash ((r = s) \rightsquigarrow Q, R))\\ [(\mathit{guard} \;\wedge\; r = s) \;\Rightarrow\sqcap\; ys|\mathit{cond} \;\bullet\; \tau\rangle\\ Q'\end{array}}$$

$$\frac{(\Gamma \vdash R)\quad [\mathit{guard} \;\Rightarrow\; \$ys?zs|\mathit{cond} \;\bullet\; r\rangle \quad R'}{\begin{array}{c}(\Gamma \vdash ((r = s) \rightsquigarrow Q, R))\\ [(\mathit{guard} \;\wedge\; \neg r = s) \;\Rightarrow\; \$ys?zs|\mathit{cond} \;\bullet\; r\rangle\\ R'\end{array}}$$

$$\frac{(\Gamma \vdash R)\quad [\mathit{guard} \;\Rightarrow\sqcap\; ys|\mathit{cond} \;\bullet\; \tau\rangle \quad R'}{\begin{array}{c}(\Gamma \vdash ((r = s) \rightsquigarrow Q, R))\\ [(\mathit{guard} \;\wedge\; \neg r = s) \;\Rightarrow\sqcap\; ys|\mathit{cond} \;\bullet\; \tau\rangle\\ R'\end{array}}$$

The rest of the firing rules are essentially the same as the ones given in Section 3.2.3.

Provided the equality tests in $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$ do satisfy the assumption in (a), this alternative procedure indeed produces an ASLTS with respect to X, T and ϕ, which we shall call

$\mathcal{AS}_{\Gamma \vdash P}$

■

Example 3.3.2 Consider the following X-DI process, in which the event *dup* is enabled whenever the last two inputs are equal:

$$\begin{array}{l}
\mathit{TestDups} = \mathit{in}?x : X \longrightarrow \mathit{in}?y : X \longrightarrow \mathit{TestDups}'(x, y)\\
\mathit{TestDups}'(x, y) = (\mathit{if}\ x = y\ \mathit{then}\ \mathit{dup} \longrightarrow \mathit{TestDups}'(x, y)\\
\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \mathit{STOP})\\
\qquad\qquad\qquad\quad \Box\ (\mathit{in}?z : X \longrightarrow \mathit{TestDups}'(y, z))
\end{array}$$

The ASLTS constructed by transforming the SLTS $\mathcal{S}_{\{\}\vdash \mathit{TestDups}}$ is shown in Figure 3.2. (Or rather what is shown is its simplified version, as we did for the SLTS $\mathcal{S}_{\{\}\vdash \mathit{NondReg}}$ in Example 3.3.1.) In the procedure for constructing $\mathcal{S}_{\{\}\vdash \mathit{TestDups}}$, the equality test in $\mathit{TestDups}'(x,y)$ is reduced before interpreting the external choice, and that is why each of the nodes N_2, N_3 and N_4 in the ASLTS has two transitions generated by the prefix $\mathit{in}?z : X$, one for the positive outcome of the equality test and the other for the negative.

On the other hand, *TestDups* satisfies the assumption (a) above, and thus the alternative procedure can be applied to construct the ASLTS $\mathcal{AS}_{\{\}\vdash \mathit{NondReg}}$. This ASTLS (or rather its simplified version again) is shown in Figure 3.3. It is simpler than $\overline{\mathcal{S}_{\{\}\vdash \mathit{TestDups}}}^{AS}$ in that the nodes N_2, N_3 and N_4 now each have only one transition generated by the prefix $\mathit{in}?z : X$, and its *guard* condition is *True*. This is the kind of simplification that the alternative procedure achieves: equality tests are performed more lazily than in the procedure for constructing SLTSs and are thus more closely reflected in the *guard* conditions of the alternative symbolic transitions.

More generally, we shall see in Section 3.5 that for any X,Γ-WDI process P which satisfies a condition (**PosConjEqT**$_X$) restricting its equality tests, and *RemDups* is an example, its ASLTS $\mathcal{AS}_{\Gamma\vdash P}$ has the property that the *guard* conditions of its transitions are conjunctions of equality tests (which includes empty conjunctions *True*). That is not the case with conditional transitions of the SLTSs $\mathcal{S}_{\Gamma\vdash P}$ and thus neither with the *guard* conditions in the ASLTSs obtained from them, and this is what justifies introducing ASLTSs and developing the alternative procedure for constructing them. ■

3.3.3 Finiteness

As we have already remarked, the SLTSs and ASLTSs produced by the procedures in Sections 3.2.1 and 3.3.2 are not always finite.

Knowing when various degrees of finiteness are present will be significant both in establishing a relationship with the denotational semantics (see Section 3.4) and when it comes to automation (see Sections 3.8 and 5.6), and that is what the results in this section are devoted to.

Let us first define a few quantities which measure various aspects of SLTSs and ASLTSs. They will be particularly important in the rest of the thesis, most notably Sections 3.7.2 and 5.4.

Definition 3.3.3 For any SLTS $\mathcal{S} = (\mathcal{N}, N_0, \mathcal{T}, \mathit{source}, \mathit{target})$, let

$$\begin{aligned}
W(\mathcal{S}) &= \sup\{|\mathit{Context}(N)| \mid N \in \mathcal{N} \wedge N \textit{ has no cond. symb. trans.}\}\\
L_{\$}(\mathcal{S}) &= \sup\{|\mathit{ys}(\alpha)| \mid \alpha \in \mathcal{T} \wedge \alpha \textit{ is a vis. symb. trans.}\}\\
L_{?}(\mathcal{S}) &= \sup\{|\mathit{zs}(\alpha)| \mid \alpha \in \mathcal{T} \wedge \alpha \textit{ is a vis. symb. trans.}\}\\
L_{\sqcap}(\mathcal{S}) &= \sup\{|\mathit{ys}(\alpha)| \mid \alpha \in \mathcal{T} \wedge \alpha \textit{ is an inv. symb. trans.}\}
\end{aligned}$$

where the supremum of an empty set is 0, and the supremum of an unbounded set is ω (the smallest infinite cardinal).

For any ASLTS $\mathcal{AS}$, the definitions are the same. ■

Proposition 3.3.1 *Any node N in any SLTS $\mathcal{S}_{\Gamma\vdash P}$ (respectively, in any ASLTS $\mathcal{AS}_{\Gamma\vdash P}$) has*

(i) at most finitely many visible (alternative) symbolic transitions α whose events $r(\alpha)$ have any fixed shell H, and

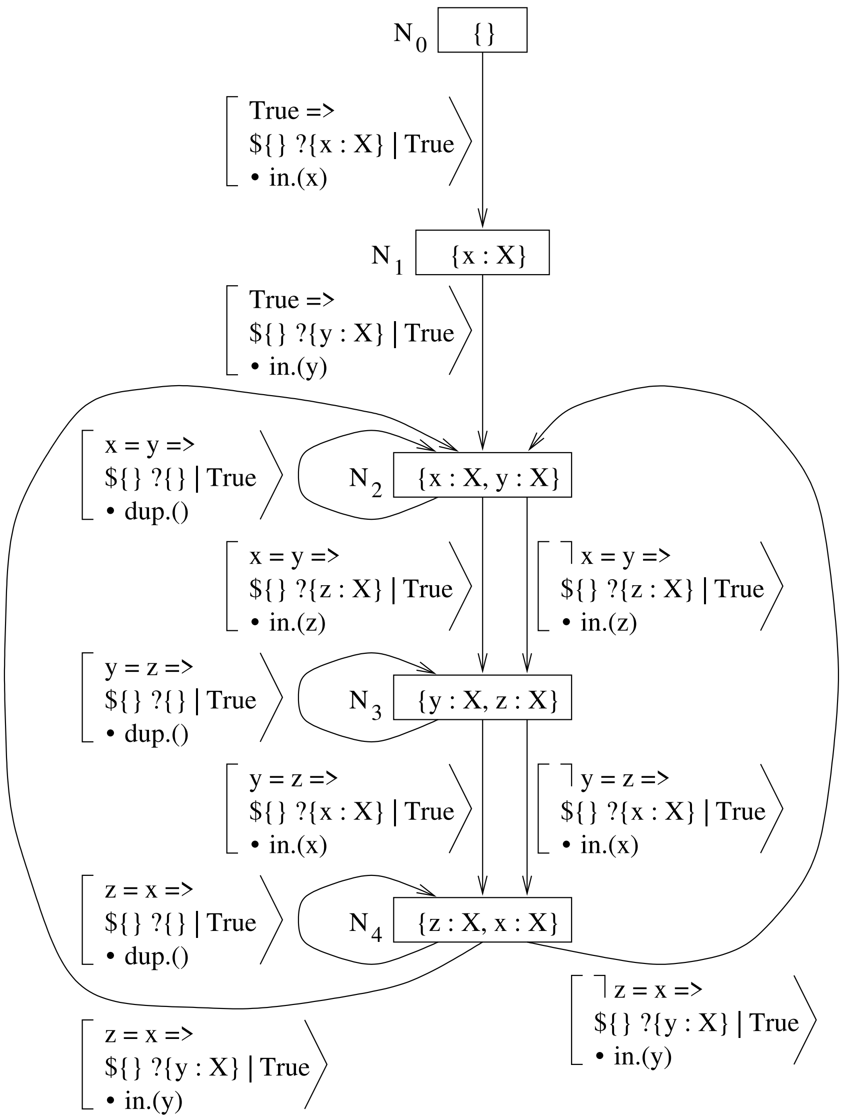
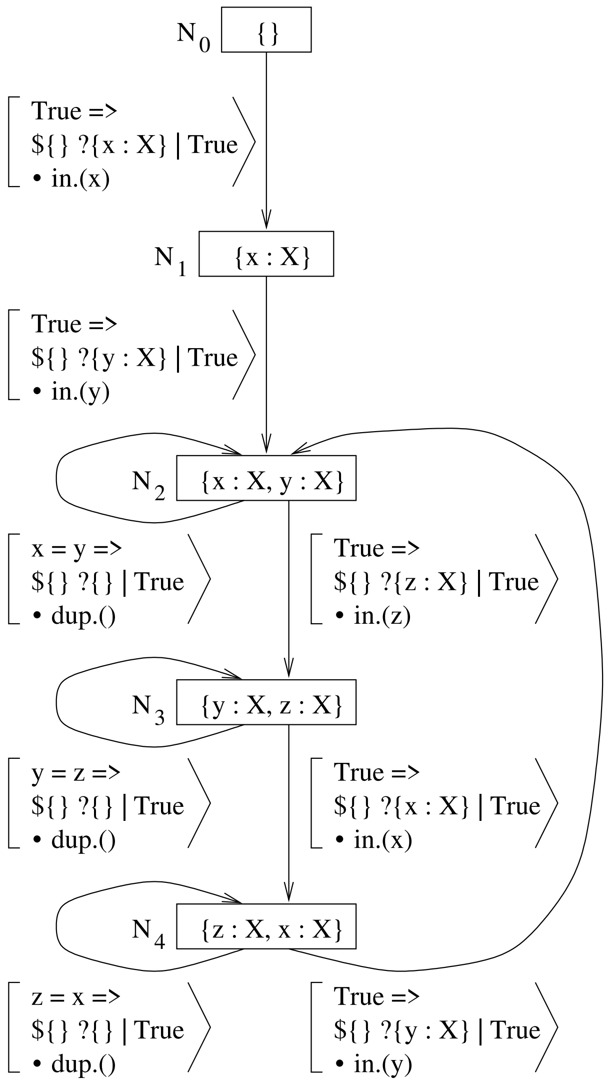


Figure 3.2: $\overline{\mathcal{S}_{\{\}\vdash TestDups}}^{AS}$



N0 {}
True => ${} ?{x : X} | True • in.(x)
N1 {x : X}
True => ${} ?{y : X} | True • in.(y)
N2 {x : X, y : X}
x = y => ${} ?{} | True • dup.()
True => ${} ?{z : X} | True • in.(z)
N3 {y : X, z : X}
y = z => ${} ?{} | True • dup.()
True => ${} ?{x : X} | True • in.(x)
N4 {z : X, x : X}
z = x => ${} ?{} | True • dup.()
True => ${} ?{y : X} | True • in.(y)

Figure 3.3: $\mathcal{AS}_{\{\}\vdash TestDups}$

(ii) at most finitely many visible (alternative) symbolic transitions α whose sets $ys(\alpha)$ of nondeterministic selections are not empty.

Proof. The proof is a straightforward induction on the firing rules. ■

Proposition 3.3.2 *An SLTS $\mathcal{S}_{\Gamma \vdash P}$ (respectively, an ASLTS $\mathcal{AS}_{\Gamma \vdash P}$) is finitely branching, in the sense that any node has finitely many transitions, if the type of any input, nondeterministic selection or replicated nondeterministic choice in P satisfies the condition $AllTypeVars, +, \times$ (see under 'Conditions on types' in Section 2.4).*

Proof. The proof is again a straightforward induction on the firing rules. ■

Proposition 3.3.3 *(i) An SLTS $\mathcal{S}_{\Gamma \vdash P}$ (respectively, an ASLTS $\mathcal{AS}_{\Gamma \vdash P}$) is finite, in the sense that it has finitely many nodes and finitely many transitions, if in addition to the assumption in Proposition 3.3.2, there are finitely many different node processes up to renaming their free variables, i.e. there are finitely many equivalence classes of the equivalence relation in which node processes $\Gamma' \vdash P'$ and $\Gamma'' \vdash P''$ are related if and only if P'' can be obtained from P' by a bijection between Γ' and Γ'' that preserves types.*

(ii) Under the same assumptions, it is also the case that the algorithmic construction of $\mathcal{S}_{\Gamma \vdash P}$ by the procedure in Section 3.2.1 (respectively, of $\mathcal{AS}_{\Gamma \vdash P}$ by the procedure in Section 3.3.2) terminates.

Proof. The proof of (i) proceeds as follows. By the assumption in Proposition 3.3.2, $L = L_{\$}(\mathcal{S}_{\Gamma \vdash P}) + L_{?}(\mathcal{S}_{\Gamma \vdash P})$ (respectively, $L = L_{\$}(\mathcal{AS}_{\Gamma \vdash P}) + L_{?}(\mathcal{AS}_{\Gamma \vdash P})$) is finite. On the other hand, by the additional assumption given, $W = W(\mathcal{S}_{\Gamma \vdash P})$ (respectively, $W = W(\mathcal{AS}_{\Gamma \vdash P})$) is finite. Then, by the assumption of minimality of variable names (see Section 3.2.2), it follows that the set of all variable names appearing in the type contexts of the node processes has at most $|\Gamma| + W + L$ members. Hence, there can in fact be only finitely many node processes, and so $\mathcal{S}_{\Gamma \vdash P}$ (respectively, $\mathcal{AS}_{\Gamma \vdash P}$) has finitely many nodes by the assumption of reusing nodes (see Section 3.2.1).

The proof of (ii) is straightforward. (See the remarks after Proposition 3.1.1, and at the beginning of Section 3.2.3.) ■

What Proposition 3.3.3 (i) is in other words saying is that an SLTS $\mathcal{S}_{\Gamma \vdash P}$ (an ASLTS $\mathcal{AS}_{\Gamma \vdash P}$) cannot be infinite because of using unboundedly many variable names unnecessarily.

The assumption that there are finitely many node processes up to renaming their free variables is satisfied provided the following are true as $\Gamma' \vdash P'$ ranges over the node processes:

- $|\Gamma'|$ is bounded,
- the structure of P' is bounded, meaning that the P' do not for example contain parallel compositions with unboundedly many components, and
- the data of concrete types in P' is bounded, meaning that the P' do not for example contain unboundedly long lists.

Although syntactic restrictions which are sufficient for the assumptions in Proposition 3.3.3 to be satisfied can certainly be formulated (see [Ros94a] or [Ros98a, Section C.1.1] for ideas), the general problem of determining whether $\mathcal{S}_{\Gamma \vdash P}$ (respectively, $\mathcal{AS}_{\Gamma \vdash P}$) is finite is undecidable, as can be shown by reducing the Halting Problem to it. (For an introduction to computability, see e.g. [Cut80].)

3.4 Relationship with denotational semantics

In this section we define denotational semantic values that correspond to SLTSs and ASLTSs, which in particular gives formal meanings to symbolic transitions and alternative symbolic transitions. We then present the so-called Congruence Theorem, which states that those values are the same as the values produced by the denotational semantics given in Sections 2.4 and 2.5. The Congruence Theorem will in particular be vital in Chapters 4 and 5, where we shall often work with SLTSs and ASLTSs to establish results about the denotational semantics and refinement.

The jump from SLTSs and ASLTSs to denotational semantic values is not small, and perhaps it would be more natural to define it via ordinary LTSs. However, for brevity we do not do that.

3.4.1 Corresponding denotational semantic values

Suppose $\mathcal{S}$ is an SLTS (respectively, $\mathcal{AS}$ is an ASLTS) with respect to some X, T and ϕ. For any δ (which assigns a nonempty set to X), an *execution* in $\mathcal{S}$ (in $\mathcal{AS}$) with respect to δ is a sequence

$$N_1^{\eta_1} \alpha_1^{\eta'_1} N_2^{\eta_2} \alpha_2^{\eta'_2} \ldots$$

such that

- it either ends with some $N_k^{\eta_k}$ or it is infinite,
- $N_1 \alpha_1 N_2 \alpha_2 \ldots$ is a path,
- each η_i is an assignment to $Context(N_i)$ with respect to δ (which in $\mathcal{AS}$ satisfies $guard(\alpha_{i+1})$),
- each η'_i is an assignment to $Vars(\alpha_i)$ with respect to δ, for which $\eta_i \cup \eta'_i$ satisfies $cond(\alpha_i)$, and
- each η_{i+1} equals $(\eta_i \cup \eta'_i) \upharpoonright Context(N_{i+1})$.

We shall also use fragments of this notation and their negations, so that for example $\neg N_1^{\eta_1} \alpha_1^{\eta'_1}$ will mean that there do not exist N_2 and η_2 such that $N_1^{\eta_1} \alpha_1^{\eta'_1} N_2^{\eta_2}$ is an execution.

If an execution is finite, so that it ends with some $N_k^{\eta_k}$, it generates a trace

$$t = \langle [\![r(\alpha_i)]\!]_{\delta, \eta_i \cup \eta'_i} \mid \alpha_i \textit{ is a visible transition} \rangle$$

and we write

$$N_1^{\eta_1} \stackrel{t}{\Longrightarrow} N_k^{\eta_k} \textit{ w.r.t. } \delta$$

We can then define the set of all traces corresponding to a node as the set of all traces generated by finite executions from that node:

$$traces(N)_{\delta,\eta} = \{t \mid \exists\, N', \eta'. N^{\eta} \stackrel{t}{\Longrightarrow} N'^{\eta'} \ \ w.r.t.\ \delta\}$$

To define the set of all failures corresponding to a node, let us first define when a set A is a refusal of a node $N'^{\eta'}$. According to what we have outlined under 'Visible' in Section 3.2.2, A has to be generated by an assignment η^* to all nondeterministic selections in the visible transitions of N'. More precisely, we write

$$N'^{\eta'} \ \ ref^{\eta^*} \ A \ w.r.t.\ \delta$$

provided

- N' has no conditional transitions,
- for any invisible transition α of N' and any assignment η'' to $ys(\alpha)$ with respect to δ, we have that $\neg N'^{\eta'} \alpha^{\eta''}$,
- $\eta^*(\alpha)$ is an assignment to $ys(\alpha)$ with respect to δ, for every visible transition α of N', where Proposition 3.3.1 (ii) tells us that there are at most finitely many such α with nonempty $ys(\alpha)$, and
- A is disjoint from the set of all

  $$[\![r(\alpha)]\!]_{\delta,\eta' \cup \eta^*(\alpha) \cup \eta''}$$

  as α ranges over the visible transitions of N' (for which, in the case of $\mathcal{AS}$, η' satisfies $guard(\alpha)$) and η'' ranges over the assignments to $zs(\alpha)$ with respect to δ for which $\eta' \cup \eta^*(\alpha) \cup \eta''$ satisfies $cond(\alpha)$.

The set of all failures of a node can now be defined as

$$\begin{aligned} failures(N)_{\delta,\eta} = \{(t, A) \mid \exists\, N', \eta', \eta^*.\ & N^{\eta} \stackrel{t}{\Longrightarrow} N'^{\eta'} \ \ w.r.t.\ \delta \\ & \wedge N'^{\eta'} \ \ ref^{\eta^*} \ A \ w.r.t.\ \delta\} \end{aligned}$$

To define the set of all divergences of a node, we write

$$N'^{\eta'} \ \ div \ w.r.t.\ \delta$$

provided there is an infinite execution with respect to δ which begins at $N'^{\eta'}$ and which consists of only invisible and conditional transitions. We can then define

$$\begin{aligned} divergences(N)_{\delta,\eta} = \{t\hat{\ }u \mid \exists\, N', \eta'.\ & N^{\eta} \stackrel{t}{\Longrightarrow} N'^{\eta'} \ \ w.r.t.\ \delta \\ & \wedge N'^{\eta'} \ \ div \ w.r.t.\ \delta\} \end{aligned}$$

and as usual

$$failures_{\bot}(N)_{\delta,\eta} = failures(N)_{\delta,\eta} \ \cup \ \{(t, A) \mid t \in divergences(N)_{\delta,\eta}\}$$

We now have all the ingredients for defining the denotational semantic values corresponding to $\mathcal{S}$ in each of the three denotational models. They are those that correspond to its initial node:

$$\begin{aligned} [\![\mathcal{S}]\!]^{\mathrm{Tr}}_{\delta,\eta} &= traces(N_0)_{\delta,\eta} \\ [\![\mathcal{S}]\!]^{\mathrm{Fa}}_{\delta,\eta} &= (traces(N_0)_{\delta,\eta}, fails(N_0)_{\delta,\eta}) \\ [\![\mathcal{S}]\!]^{\mathrm{FD}}_{\delta,\eta} &= (failures_{\bot}(N_0)_{\delta,\eta}, divs(N_0)_{\delta,\eta}) \end{aligned}$$

The definitions for $\mathcal{AS}$ are the same.

3.4.2 The Congruence Theorem

The following is the so-called Congruence Theorem. It states that, for any of the three denotational models, the denotational semantic value corresponding to the SLTS or ASLTS of any process (see the previous section) is the same as the interpretation of the process in the denotational semantics (see Sections 2.4 and 2.5).

For the failures/divergences model, the theorem relies on the restriction to finitely nondeterministic CSP constructs (see under 'Restrictions with FD' (i) in Section 2.5.2).

Theorem 3.4.1 *(a) For any $\mathcal{S}_{\Gamma\vdash P}$ (see Section 3.3.1) and any M, δ and η,*

$$[\![\mathcal{S}_{\Gamma\vdash P}]\!]^M_{\delta,\eta} = [\![\Gamma \vdash P]\!]^M_{\delta,\eta}$$

(b) For any $\mathcal{AS}_{\Gamma\vdash P}$ (see Section 3.3.2) and any M, δ and η,

$$[\![\mathcal{AS}_{\Gamma\vdash P}]\!]^M_{\delta,\eta} = [\![\Gamma \vdash P]\!]^M_{\delta,\eta}$$

Proof. The proof follows the same pattern as for ordinary LTSs (see [Ros98a, Section 9.4]), so we only sketch it concentrating on SLTSs and the failures/divergences model.

It is by induction on the complexity of P. In particular, we shall need to consider 'right-hand sides' P' of recursively defined processes $\mu\, p' : (U \to \mathit{Proc}_{T,\psi}).P'$, so that in addition to constants and operations allowed by weak data independence we shall need to allow variables of types $U \to \mathit{Proc}_{T,\psi}$.

More precisely, let us fix a type variable X from *NEFTypeVars* and a type T such that T^{alph} and $\mathit{Free}(T) \subseteq \{X\}$. The proof is then by induction on the complexity of a process

$$\Gamma \cup \{q_1 : V_1 \to \mathit{Proc}_{T,\psi_1}, \ldots, q_n : V_n \to \mathit{Proc}_{T,\psi_n}\} \vdash P : \mathit{Proc}_{T,\phi}$$

such that the type judgement has at most the free type variable X, Γ satisfies the restrictions of weak data independence in X (see Section 2.7.2), and the V_i do not involve any process types. What it shows is that for any

$$\Delta_1 \vdash Q_1 : V_1 \to \mathit{Proc}_{T,\psi_1}, \ldots, \Delta_n \vdash Q_n : V_n \to \mathit{Proc}_{T,\psi_n}$$

which are weakly data independent in X and where the Δ_i need not be pairwise disjoint nor disjoint from Γ, and for any δ and η, we have

$$\begin{array}{l} [\![\mathcal{S}_{\Gamma\cup\Delta_1\cup\cdots\cup\Delta_n\vdash P[Q_1/q_1,\ldots,Q_n/q_n]}]\!]^{\mathrm{FD}}_{\delta,\eta} = \\ [\![\Gamma \cup \{q_1 : V_1 \to \mathit{Proc}_{T,\psi_1}, \ldots, q_n : V_n \to \mathit{Proc}_{T,\psi_n}\} \vdash P]\!]^{\mathrm{FD}}_{\delta,(\eta\restriction\Gamma)\cup\eta'} \end{array} \tag{3.1}$$

where for each i, $\eta'[\![q_i]\!]$ is the denotational semantic value corresponding to the SLTS of Q_i. More formally,

$$\eta'[\![q_i]\!] = \Phi(\Delta_i \vdash Q_i)_{\delta,\eta\restriction\Delta_i}$$

where Φ is defined as follows. If W is a type involving no process types and such that $\mathit{Free}(W) \subseteq \{X\}$, and if δ assigns a nonempty set to X, let us call a value $w \in [\![W]\!]_\delta$ 'X-WDI expressible' if there exists an X,$\Xi(w)$-WDI term $s(w)$ of type W and an assignment $\xi(w)$ to $\Xi(w)$ with respect to δ such that $[\![s(w)]\!]^{\mathrm{FD}}_{\delta,\xi(w)} = w$. Then for any X,Θ-WDI process R of type $W \to \mathit{Proc}_{T,\chi}$, any assignment θ to Θ with respect to δ, and any value $w \in [\![W]\!]_\delta$, we define

$$\Phi(\Theta \vdash R)_{\delta,\theta}(w) = \begin{cases} [\![\mathcal{S}_{\Theta\cup\Xi(w)\vdash(R\,s(w))}]\!]^{\mathrm{FD}}_{\delta,\theta\cup\xi(w)}, & \textit{if } w \textit{ is } X\textit{-WDI expressible} \\ \bot, & \textit{otherwise} \end{cases}$$

R_i:	P'':
$q_i\, r^\sharp$	none
$STOP_T$	none
$id_i\ it_1^\sharp\ \ldots\ it_m^\sharp \longrightarrow P'$	the processes P'' in the second firing rule for prefixing (see Section 3.2.3)
$P'^\sharp \,\Box\, Q'^\sharp$	P', Q'
$P' \sqcap Q'$	P', Q'
$\Box_{x:U} P'^\sharp$	the processes $P'[r_j/x]$ (see under 'Replicated external choice' in Section 2.5.2)
$\sqcap_{x:U} P'$	the processes $P'[H[y_1,\ldots,y_h]/x]$ (see under 'Replicated nondeterministic choice' in Section 3.2.3)
$P'^\sharp \underset{S^\sharp}{\parallel} Q'^\sharp$	P', Q'
$P'^\sharp \setminus S^\sharp$	P'
$(\mu\, p' : (U \to Proc_{T,\phi}).P')\, r^\sharp$	the processes $P'\ s(u)$ (see below)

Table 3.4: Defining the relation $\mathcal{I}$

where $\Xi(w)$ can always be picked to be disjoint from Θ.

Let us now specify what we mean by 'induction on the complexity of P'. We mean that the induction is on a relation $\mathcal{I}$ defined as follows. If P is a process as above, it is straightforward to show that Proposition 3.1.1 can be applied to P to yield a finite number of conditional transitions

$$\begin{array}{lll}(\Gamma \cup \{q_1 : V_1 \to Proc_{T,\psi_1}, \ldots, q_n : V_n \to Proc_{T,\psi_n}\} \vdash P) & [cond_1\rangle & R_1 \\ & \vdots & \\ (\Gamma \cup \{q_1 : V_1 \to Proc_{T,\psi_1}, \ldots, q_n : V_n \to Proc_{T,\psi_n}\} \vdash P) & [cond_k\rangle & R_k\end{array}$$

with each R_i being in evaluated form (see Section 3.1.2), where the definition of evaluated forms is extended to include processes of the form

$$q_i\, r^\sharp$$

Then for any R_i we define that $P''\,\mathcal{I}\,P$ for the processes P'' given in Table 3.4. To show that the relation $\mathcal{I}$ thus defined is suitable for induction, it suffices to show that there are no infinite sequences $P_0''', P_1''', \ldots$ with $P_{j+1}'''\,\mathcal{I}\,P_j'''$ for all $j \in \mathbb{N}$ (see e.g. Definition 3.1 and Exercise 7 in [Kun80, Chapter III]). This is not difficult given the strong normalisation property (see e.g. [GLT89, Loa97]) of the λ-calculus constructs of our language, i.e. that there are no infinite sequences of consecutive one-step reductions (see Table 3.1).

To prove Equation 3.1 by induction on $\mathcal{I}$, suppose P is a process as above, and consider any $Q_1, \ldots, Q_n$, δ and η as above. We can then apply Proposition 3.1.1 to P and obtain some $cond_1$, $\ldots$, $cond_k$ and $R_1, \ldots, R_k$ as in the definition of $\mathcal{I}$. Since $cond_1, \ldots, cond_k$ disjointly partition *True*, exactly one $cond_i$ is satisfied by η. By showing that conditional symbolic transitions (see

Section 3.1.1) preserve the denotational semantics, it suffices to consider R_i instead of P, or in other words we can assume that P is of one of the forms in the left-hand column of Table 3.4.

If P is of the form $q_i\, r^\sharp$, no q_j occurs in r (see under 'Type variable X' in Section 3.1.2), and so Equation 3.1 reduces to

$$[\![\mathcal{S}_{\Gamma\cup\Delta_i\vdash(Q_i\ r)}]\!]^{\mathrm{FD}}_{\delta,\eta} = [\![\mathcal{S}_{\Delta_i\cup\Xi(v)\vdash(Q_i\ s([\![r]\!]^{\mathrm{FD}}_{\delta,\eta}))}]\!]^{\mathrm{FD}}_{\delta,(\eta\restriction\Delta_i)\cup\xi(v)}$$

which is an instance of the following result. If

- δ assigns a nonempty set to X,
- for each $j = 1,\ldots,m$, s'_j and s''_j are X,Ξ'_j-WDI and X,Ξ''_j terms of a type W_j involving no process types and such that $\mathit{Free}(W_j) \subseteq \{X\}$ (where no disjointness among the Ξ'_j and the Ξ''_j is required),
- ξ' and ξ'' are assignments to $\Xi'_1\cup\cdots\cup\Xi'_m$ and $\Xi''_1\cup\cdots\cup\Xi''_m$ with respect to δ such that for each j,

  $$[\![s'_j]\!]^{\mathrm{FD}}_{\delta,\xi'\restriction\Xi'_j} = [\![s''_j]\!]^{\mathrm{FD}}_{\delta,\xi''\restriction\Xi''_j}$$

  and
- $R[\cdot,\ldots,\cdot]$ is an X,Θ-WDI process which is a W_1, …, W_m term context (where no disjointness of Θ from the Ξ'_j or from the Ξ''_j is required),

then for any assignment θ to Θ with respect to δ which is compatible with ξ' and ξ'', we have that

$$[\![\mathcal{S}_{\Theta\cup\Xi'_1\cup\cdots\cup\Xi'_m\vdash R[s'_1,\ldots,s'_m]}]\!]^{\mathrm{FD}}_{\delta,\theta\cup\xi'} = [\![\mathcal{S}_{\Theta\cup\Xi''_1\cup\cdots\cup\Xi''_m\vdash R[s''_1,\ldots,s''_m]}]\!]^{\mathrm{FD}}_{\delta,\theta\cup\xi''}$$

This can be shown by establishing a correspondence between executing the two SLTSs from their initial nodes with respect to δ and $\theta\cup\xi'$, $\theta\cup\xi''$.

If P is of any other form in the left-hand column of Table 3.4 except the recursion, Equation 3.1 follows by the inductive hypothesis for each of the predecessor processes P'' on the right, and by showing that the firing rules for the principal CSP construct of P preserve the denotational semantics in the sense that the relationship of

$$[\![\mathcal{S}_{P[Q_1/q_1,\ldots,Q_n/q_n]}]\!]^{\mathrm{FD}}_{\delta,\eta}$$

with the

$$[\![\mathcal{S}_{P''[Q_1/q_1,\ldots,Q_n/q_n]}]\!]^{\mathrm{FD}}_{\delta,\eta\cup\eta''}$$

is the same as the way in which

$$[\![P]\!]^{\mathrm{FD}}_{\delta,(\eta\restriction\Gamma)\cup\eta'}$$

is defined in terms of the

$$[\![P'']\!]^{\mathrm{FD}}_{\delta,(\eta\restriction\Gamma)\cup\eta'\cup\eta''}$$

Finally, if P is of the form $(\mu\, p' : (U \to \mathit{Proc}_{T,\phi}).P')\ r^\sharp$, Equation 3.1 follows once we show that

$$\Phi(\Gamma\cup\Delta_1\cup\cdots\cup\Delta_n \vdash (\mu\, p' : (U \to \mathit{Proc}_{T,\phi}).P')[Q_1/q_1,\ldots,Q_n/q_n])_{\delta,\eta}$$

is on all X-WDI expressible values $u \in [\![U]\!]_\delta$ equal to the least fixed point (with respect to the refinement ordering lifted pointwise to the function space) of

$$\mathcal{G}(g) = \left[\!\!\left[\Gamma \cup \left\{ \begin{array}{c} q_1 : V_1 \to Proc_{T,\psi_1}, \\ \vdots \\ q_n : V_n \to Proc_{T,\psi_n}, \\ p' : U \to Proc_{T,\phi} \end{array} \right\} \vdash P' \right]\!\!\right]^{\mathrm{FD}}_{\delta,(\eta\restriction\Gamma)\cup\eta'\cup\{p'\mapsto g\}}$$

and this can be established in the same way that recursion was dealt with in [Ros98a, Section 9.4]. It involves applying the inductive hypothesis to processes of the form

$$\Gamma \cup \left\{ \begin{array}{c} q_1 : V_1 \to Proc_{T,\psi_1}, \\ \vdots \\ q_n : V_n \to Proc_{T,\psi_n}, \\ p' : U \to Proc_{T,\phi} \end{array} \right\} \vdash (P'\ s(u))$$

and it relies on the following auxiliary result. For any

- process

  $$\Theta \cup \{r_1 : W_1 \to Proc_{T,\chi_1}, \ldots, r_l : W_l \to Proc_{T,\chi_l}\} \vdash R : Proc_{T,\chi'}$$

  such that the type judgement has at most the free type variable X, Θ satisfies the restrictions of weak data independence in X, and the W_j do not involve any process types,

- δ (which assigns a nonempty set to X), and

- assignments θ and θ' with respect to FD and δ such that

  $$\theta[\![x]\!] = \theta'[\![x]\!]$$

  for each $(x : W') \in \Theta$ and

  $$\theta[\![r_j]\!](w) = \theta'[\![r_j]\!](w)$$

  for each r_j and all X-WDI expressible values $w \in [\![W_j]\!]_\delta$,

we have

$$[\![R]\!]^{\mathrm{FD}}_{\delta,\theta} = [\![R]\!]^{\mathrm{FD}}_{\delta,\theta'}$$

This can be proved for example by induction on the relation $\mathcal{I}$. ■

3.4.3 Equivalence and transformations

Since we consider denotational semantic values to be the ultimate meanings of processes, and SLTSs and ASLTSs to be stepping stones to those meanings, we can regard two SLTSs or ASLTSs to be equivalent if their corresponding denotational semantic values are the same. In particular, this allows us to study and perform transformations of SLTSs and ASLTSs which preserve the equivalence, aiming for simplification.

Definition 3.4.1 More precisely, given an SLTS or ASLTS $\mathcal{S}$ and an SLTS or ASLTS $\mathcal{S}'$ which are both with respect to some X, T and ϕ and such that $\mathit{Context}(N_0(\mathcal{S})) = \mathit{Context}(N_0(\mathcal{S}'))$, and given a denotational model M, we say that $\mathcal{S}$ and $\mathcal{S}'$ are *equivalent with respect to M* if and only if

$$[\![\mathcal{S}]\!]^M_{\delta,\eta} = [\![\mathcal{S}']\!]^M_{\delta,\eta}$$

for all δ and η. ■

Although equivalence-preserving transformations are a very important subject, we leave them for future research and only informally point towards three possible kinds:

Simple. We have already met simple equivalence-preserving transformations in Examples 3.3.1 and 3.3.2, such as omitting a transition $[\mathit{True}\rangle$ or $[\sqcap\ \{\}|\ \mathit{True}\ \bullet\ \tau\rangle$ which is the only transition of its source node and identifying its source and target nodes. Other possibilities include removing a conditional transition which can never be performed and all nodes and transitions reachable only by it, or identifying two invisible transitions which have the same source and target nodes and the same labels.

Model-specific. Another kind are transformations which preserve the equivalence with respect to one or two specific denotational models. For example, with respect to the finite-traces model we can turn all nondeterministic selections into inputs, and omit any $[\sqcap\ \{\}|\ \mathit{True} \bullet \tau\rangle$ transition and add the transitions of its target node to those of its source node; or with respect to the failures/divergences model we can replace any node which always diverges and all transitions and nodes reachable only through it by a node which only has a $[\sqcap\ \{\}|\ \mathit{True}\ \bullet\ \tau\rangle$ transition to itself.

Complex. There are a number of more complex equivalence-preserving transformations of ordinary LTSs, such as normalisation, factoring by strong bisimulation, τ-loop elimination and diamond elimination (see [RG+95] or [Ros98a, Section C.2]). Extending them to SLTSs, ASLTSs and other notions of symbolic transition systems is a developing research topic (see e.g. [HL95a, Sch94, Lin96, Pac96, Rat97, ST97, Kok97, Kok98] and Sections 3.6 and 3.7 in this thesis).

3.5 Conditions

This section is devoted to stating a few conditions on terms (in particular, on processes), which will be important in the rest of the thesis. Their purpose is to restrict the degrees to which a term can contain equality tests between values of variable types or nondeterminism whose effects are not immediately apparent.

For each condition, we shall also state a proposition about the consequences it has on SLTSs and ASLTSs. Their proofs proceed mainly by inductions on the one-step reductions (see Table 3.1) and on the firing rules (see Sections 3.2.3 and 3.3.2), and are straightforward, so we omit them.

3.5.1 Equality tests

When no condition is assumed, a term can contain arbitrary equality tests between values of variable types. At the opposite extreme, the following condition does not allow any equality tests between values of type X, whether they are explicit or present implicitly by virtue of synchronisations in parallel compositions:

Condition $(\mathbf{NoEqT}_X)$. A term $\Gamma \vdash r$ satisfies this condition if

(i) for any equality test

$$\Gamma' \vdash ((s = s') \rightsquigarrow u, v)$$

within r, X does not occur free in the type of s and s', and

(ii) for any parallel composition

$$\Gamma' \vdash P \underset{S}{\|} Q$$

within r, there exist derivable type judgements $\Gamma' \vdash P : \mathit{Proc}_{T,\phi}$ and $\Gamma' \vdash Q : \mathit{Proc}_{T,\psi}$ such that if $T = \sum_{i=1}^{n} id_i. \prod_{j=1}^{m_i} T_{i,j}$ then for any i and j for which X occurs free in $T_{i,j}$ and S has an elem which is either $*_T$ or of the form $id_i^T.e$, we have that

$$\phi(id_i, j) \subseteq \{?\} \vee \psi(id_i, j) \subseteq \{?\}$$

■

For example, each of the processes *BImpl*, *B*, *BUFF* and *LastReg* satisfy $(\mathbf{NoEqT}_X)$, but the process *NondReg* does not (see Section 2.7.1). Another interesting example is the process *Mem* in Example 2.7.1, which satisfies $(\mathbf{NoEqT}_V)$ but not $(\mathbf{NoEqT}_A)$.

Among the more general examples of data independent processes that we have mentioned in Section 2.7.1, communication protocols usually satisfy $(\mathbf{NoEqT}_X)$ if X is the type of data they communicate, and memories, databases and systems involving them frequently satisfy $(\mathbf{NoEqT}_V)$ if V is the type of values or records.

Proposition 3.5.1 *In any SLTS $\mathcal{S}_{\Gamma \vdash P}$ or any ASLTS $\mathcal{AS}_{\Gamma \vdash P}$ where P is X,Γ-WDI and satisfies $(\mathbf{NoEqT}_X)$, we have that any condition in any transition is True.*

In particular, for any visible transition α in $\mathcal{S}_{\Gamma \vdash P}$ or $\mathcal{AS}_{\Gamma \vdash P}$, each variable from $ys(\alpha)$ occurs in $r(\alpha)$ (see under 'Visible' in Section 3.2.2). ■

Assumption of transformation. Therefore we can from now on assume that any $\mathcal{S}_{\Gamma \vdash P}$ where P is X,Γ-WDI and satisfies $(\mathbf{NoEqT}_X)$ is transformed by omitting all its conditional transitions (see Section 3.4.3).

It is easy to see that this does not affect any of Propositions 3.3.1–3.3.3 (where the additional assumption in Proposition 3.3.3 is to be understood as still applying to $\mathcal{S}_{\Gamma \vdash P}$ as defined in Section 3.3.1), Theorem 3.4.1, and Proposition 3.5.1. ■

Between allowing any equality tests between values of variable types and allowing none, we have the following condition which allows only those equality tests involving values of type X for which the success branch is a prefixing and the failure branch is $STOP$. Since any equality test present implicitly by virtue of a synchronisation in a parallel composition is of that form, they do not have to be mentioned.

Condition (**PosConjEqT**$_X$). A term $\Gamma \vdash r$ satisfies this condition if for any equality test

$$\Gamma' \vdash ((s = s') \rightsquigarrow u, v)$$

within r in which X occurs free in the type of s and s', we have that u is a prefixing (see under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2) and v is a $STOP_T$. ■

When (**PosConjEqT**$_X$) is assumed, it is useful to remember that a finite conjunction of equality tests can be expressed by a single equality test between tuples. Also, provided a condition such as (**Norm**) (see below) is not assumed, finite disjunctions of equality tests can be expressed by external choices.

In addition to the examples which satisfy the stronger condition (**NoEqT**$_X$), the condition (**PosConjEqT**$_X$) is satisfied by for instance the processes *Blinking* in Example 2.7.6 and *TestDups* in Example 3.3.2. On the negative side, it is for instance the case that *NondReg* from Example 2.5.1 fails (**PosConjEqT**$_X$) and that *Mem* from Example 2.7.1 fails (**PosConjEqT**$_A$).

More substantially, the weaker version of (**PosConjEqT**$_X$) which allows any process as the success branch rather than only prefixings, has played a very important role in the recent work by Roscoe and Broadfoot on proving security protocols with model checkers [Ros98b, Bro98, RB98].

The condition (**PosConjEqT**$_X$) does not have particularly useful consequences on SLTSs, but as the following proposition states, it does on ASLTSs. In fact, that is the main reason why we have introduced ASLTSs.

Proposition 3.5.2 *In any ASLTS $\mathcal{AS}_{\Gamma \vdash P}$ where P is X,Γ-WDI and satisfies the condition* (**PosConjEqT**$_X$)*, we have that for any transition α, the condition guard(α) is not False and it does not contain any connectives $\neg$ and $\vee$ (see Section 3.3.2).* ■

3.5.2 Nondeterminism

The following condition allows only those replicated nondeterministic choices of values of type X which are 'recorded', i.e. which are immediately followed by a visible event in which the chosen values appear. More precisely, the condition bans replicated nondeterministic choices of values of type X altogether while allowing arbitrary nondeterministic selections, and it also bans hidings of inputs or nondeterministic selections of values of type X.

Condition (**RecCho**$_X$). A term $\Gamma \vdash r$ satisfies this condition if

(i) for any replicated nondeterministic choice

$$\Gamma' \vdash \textstyle\bigsqcap_{x:U} P$$

within r, X does not occur free in U, and

(ii) for any hiding

$$\Gamma' \vdash P \setminus S$$

within r, there exists a derivable type judgement $\Gamma' \vdash P : \mathit{Proc}_{T,\phi}$ such that if $T = \sum_{i=1}^{n} \mathit{id}_i. \prod_{j=1}^{m_i} T_{i,j}$ then for any i and j for which X occurs free in $T_{i,j}$ and S has an elem which is either $*_T$ or of the form $\mathit{id}_i^T.e$, we have that

$$\phi(\mathit{id}_i, j) \subseteq \{!\}$$

■

It is mainly because of the (**RecCho**$_X$) condition that we have introduced nondeterministic selections as a separate construct in our language (see under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2).

Among the concrete examples we have seen so far, (**RecCho**$_X$) is not satisfied only by *MultMatr* in Example 2.7.4 (for the reason why, see under 'More precise ways specifiers' in Section 2.5.2), and *Blinking* in Example 2.7.6. For another interesting example where (**RecCho**$_X$) is not satisfied, see *Spec* in Example 5.0.4.

The situation is the same in practice, where (**RecCho**$_X$) is satisfied by most processes which do not use channel components as indices of arrays of channels. That is especially true for specification processes, which are the ones we shall be requiring to satisfy (**RecCho**$_X$) (see Chapter 5).

Proposition 3.5.3 *In any $\mathcal{S}_{\Gamma \vdash P}$ or any $\mathcal{AS}_{\Gamma \vdash P}$ where P is X,Γ-WDI and satisfies* (**RecCho**$_X$), *we have that $\mathit{ys}(\alpha) = \{\}$ for any invisible transition α.* ■

Assumption of transformation. Therefore we can from now on assume that any $\mathcal{S}_{\Gamma \vdash P}$ where P is X,Γ-WDI and satisfies (**RecCho**$_X$) is transformed (in a straightforward manner) to ensure that both $\mathit{ys}(\alpha) = \{\}$ and $\mathit{cond}(\alpha) = \mathit{True}$ for any invisible transition α.

It is easy to see that this does not affect any of Propositions 3.3.1–3.3.3 (where the additional assumption in Proposition 3.3.3 is to be understood as still applying to $\mathcal{S}_{\Gamma \vdash P}$ as defined in Section 3.3.1), Theorem 3.4.1, and Propositions 3.5.1–3.5.3. ■

The following stronger condition is designed to ban any nondeterminism whose effects are not immediately apparent.

Condition (**Norm**). A term $\Gamma \vdash r$ satisfies this condition if:

(i) in any prefixing within r, the types of all nondeterministic selections are products of type variables (i.e. of the form $X_1 \times \cdots \times X_k$);

(ii) for any external choice

$$\Gamma' \vdash P \,\Box\, Q$$

within r:

- each of P and Q is either a prefixing (which obeys (i)) or an external choice (which obeys (ii));
- $\mathit{Chans}(P)$ and $\mathit{Chans}(Q)$ are disjoint, where

$$\begin{aligned} \mathit{Chans}(id_i\ it_1\ \ldots\ it_{m_i} \longrightarrow R) &= \{id_i\} \\ \mathit{Chans}(R \,\Box\, R') &= \mathit{Chans}(R) \cup \mathit{Chans}(R') \end{aligned}$$

(iii) for any nondeterministic choice

$$\Gamma' \vdash P \sqcap Q$$

within r:

- each of P and Q is either a STOP_T or a prefixing (which obeys (i)) or an external choice (which obeys (ii)) or a nondeterministic choice (which obeys (iii));
- $\mathit{Chans}(P)$ and $\mathit{Chans}(Q)$ are disjoint, where the definition of the map Chans is extended by

$$\begin{aligned} \mathit{Chans}(\mathit{STOP}_T) &= \{\} \\ \mathit{Chans}(R \sqcap R') &= \mathit{Chans}(R) \cup \mathit{Chans}(R') \end{aligned}$$

(iv) r does not contain replicated external choices, replicated nondeterministic choices, parallel compositions or hidings.

■

Out of the concrete examples we have seen so far, (**Norm**) is not satisfied by *BImpl* in Example 2.1.3, *BUFF* in Example 2.1.4, *NondReg* in Example 2.5.1, *Comps* and *MultMatr* in Example 2.7.4, *Blinking* in Example 2.7.6, and *TestDups* in Example 3.3.2.

In practice, (**Norm**) is too restrictive to be satisfied by many implementation processes. However, it is satisfied by a considerable proportion of specification processes,[1] and it can even be thought of as a criterion of when a specification process is clearly written. It is specification processes that we shall sometimes require to satisfy (**Norm**) (see Chapter 5).

Assumption of transformation. From now on, we assume that when constructing any $\mathcal{S}_{\Gamma \vdash P}$ or any $\mathcal{AS}_{\Gamma \vdash P}$, any prefixing in which the types of all nondeterministic selections are products of type variables does not generate any invisible transitions. (Normally it would generate a single invisible transition with no variables: see under 'Outputs, inputs and nondeterministic selections' in Section 3.2.3.)

It is easy to see that this does not affect any of Propositions 3.3.1–3.3.3 (where the additional assumption in Proposition 3.3.3 is to be understood as still applying to $\mathcal{S}_{\Gamma \vdash P}$ as defined in Section 3.3.1), Theorem 3.4.1, and Propositions 3.5.1–3.5.3. ■

[1]Perhaps the main classes of specification processes which do not satisfy (**Norm**) are those in the area of fault-tolerance since they are constructed from implementation processes (see [Ros98a, Section 12.3]) and, for the version of (**Norm**) in this thesis (see the remarks after Proposition 3.5.4), those in which adding further constraints by parallel compositions is used (see under 'Many-way synchronisation' in Section 2.3).

Proposition 3.5.4 *In any SLTS $\mathcal{S}_{\Gamma \vdash P}$ or any ASLTS $\mathcal{AS}_{\Gamma \vdash P}$ where $\Gamma \vdash P$ satisfies* (**Norm**), *we have that:*

(a) For any visible transition α, $cond(\alpha) = True$, and so in particular each variable from $ys(\alpha)$ occurs in $r(\alpha)$ (see under 'Visible' in Section 3.2.2).

(b) For any invisible transition α, $ys(\alpha) = \{\}$ and $cond(\alpha) = True$.

(c) Whenever $N_0^{\eta_0} \overset{t}{\Longrightarrow} N^{\eta}$ w.r.t. δ, $N^{\eta} \overset{\langle\rangle}{\Longrightarrow} N'^{\eta'}$ w.r.t. δ and $N^{\eta} \overset{\langle\rangle}{\Longrightarrow} N''^{\eta''}$ w.r.t. δ (see Section 3.4.1), then for any two visible transitions $N'\alpha'$ and $N''\alpha''$ (such that, in the case of $\mathcal{AS}_{\Gamma \vdash P}$, η' satisfies $guard(\alpha')$ and η'' satisfies $guard(\alpha'')$), either $\alpha' = \alpha''$ (i.e. they are the same transition) or $r(\alpha')$ and $r(\alpha'')$ have different channels.

(d) Whenever two executions $N_0^{\eta_0} \alpha_0^{\eta_0'} \ldots N_k^{\eta_k}$ w.r.t. δ and $N_0^{\theta_0} \beta_0^{\theta_0'} \ldots M_l^{\theta_l}$ w.r.t. δ give rise to the same trace (see Section 3.4.1), then

$$\langle \alpha_i \mid \alpha_i \textit{ is a vis. trans.} \rangle = \langle \beta_j \mid \beta_j \textit{ is a vis. trans.} \rangle$$

i.e. the two executions contain the same sequences of visible transitions.[2]

■

(**Norm**) can be weakened considerably, especially if one is prepared to use denotational semantics and/or symbolic operational semantics in its statement, and one such weaker version is used in [Ros98a, Section 15.2.2]. However, the corresponding versions of Proposition 3.5.4 can be quite complex, and so in this thesis we for simplicity stick to the statement of (**Norm**) given above.

3.5.3 Satisfaction by transformation

We have stated the conditions entirely syntactically, and so it is often possible to transform a term which does not satisfy one of them into a term that does and that has the same denotational semantics. Moreover, it might be feasible to search for such a term automatically (see Section 3.8).

For example, the process *TestDups* in Example 3.3.2 does not satisfy (**Norm**), but by distributing the external choice over the equality test, it can be transformed to satisfy (**Norm**).

However, we leave this topic for future research.

3.6 Symbolic normalisation

When considering refinements between data independent processes that can contain arbitrary equality tests (see Section 5.4), we shall need to be able to *normalise symbolically* the SLTSs of specification processes. The resulting structures will be called *normalised SLTSs* (or *NSLTSs* for short), and their key property will be that they can execute any trace in at most one way.

Any such specification process will be required to satisfy the condition (**RecCho**$_X$) (see Section 3.5.2), and so the symbolic normalisation of its SLTS will be made easier by the fact that

[2]This is a nontrivial statement because in general many visible transitions which are different or even have different labels can give rise to the same visible event (see under 'Visible' in Section 3.2.2).

each of its invisible symbolic transitions is of the form $[\sqcap\,\{\}\,|\,\mathit{True}\;\bullet\;\tau\rangle$ (see under 'Assumption of transformation' after Proposition 3.5.3).

In this section, we define normalised symbolic nodes and transitions from which NSLTSs are built, and present a procedure for symbolic normalisation. In the next section, we shall define NSLTSs in general, state a few results about their finiteness, and examine their relationships with the denotational semantics and with the SLTSs they can be constructed from.

3.6.1 Normalised symbolic nodes and transitions

To achieve the key property that the resulting NSLTS can execute any trace in at most one way, the procedure for symbolic normalisation will form normalised symbolic nodes by merging symbolic nodes that can be reached one from another by invisible symbolic transitions, and it will form normalised symbolic transitions by merging visible symbolic transitions that can overlap, i.e. that have the same source symbolic node and can give rise to the same visible event.

Consequently, a normalised symbolic node will be a set of symbolic nodes, and it will be considered to mean the nondeterministic choice of all the symbolic nodes it consists of. However, this precise structure will be important only during symbolic normalisation, whereas when defining NSLTSs in general (see Section 3.7.1) and when working with them (see Section 5.4), the following more abstract definition will be appropriate.

Normalised symbolic nodes. Suppose $X \in \mathit{NEFTypeVars}$, T^{alph}, $\mathit{Free}(T) \subseteq \{X\}$, and ϕ is a ways specifier with respect to T (see under 'Process types' in Section 2.4). A normalised symbolic node with respect to X, T and ϕ is an object N labelled by:

(i) a type context $\mathit{Context}(N)$ satisfying the restriction of data independence in X (see Section 2.7.1), i.e. such that each variable in it has type X;

(ii) a set $\mathit{AccSets}(N)$ of sets of visible symbolic transitions, each of which is a $\mathit{Context}(N)$,Γ' visible symbolic transition with respect to X, T and ϕ for some Γ' (see under 'Visible' in Section 3.2.2);

(iii) a boolean $\mathit{DivCond}(N)$, which is either True or False.[3]

Informally, as with ordinary symbolic nodes, $\mathit{Context}(N)$ consists of those variables introduced by preceding transitions which are needed for subsequent execution. $\mathit{AccSets}(N)$ records symbolically the acceptance sets and thus the refusal sets associated with N, and $\mathit{DivCond}(N)$ is True if N is to be considered as divergent. (For more formal definitions, see Section 3.7.3.)

As we shall see in Section 3.6.2, when N is constructed as a set of symbolic nodes, $\mathit{Context}(N)$ is the union of $\mathit{Context}(N')$ for $N' \in N$, the elements of $\mathit{AccSets}(N)$ are the sets of visible symbolic transitions of those $N' \in N$ which have no conditional or invisible symbolic transitions, and $\mathit{DivCond}(N)$ is True if some $N' \in N$ can diverge. ■

Since symbolic normalisation factors out invisible symbolic transitions and constructs normalised symbolic transitions by merging visible symbolic transitions, normalised symbolic transitions will always be visible. As for conditional symbolic transitions in the SLTS being symbolically normalised, their effect will be produced by the second parts of normalised symbolic

[3]More complex conditions will not be necessary, at least with the restrictions on SLTSs that we are assuming. See the paragraph before 'Normalised symbolic transitions' below, and under 'Eager branching', 'Less restricted $\mathcal{S}$' and 'Lazy vs eager branching' in Section 3.6.2.

transitions, which will be conditional branchings. Thus a normalised symbolic transition will have an arbitrary nonzero finite number of target nodes. The following is an appropriate abstract definition.

Normalised symbolic transitions. Suppose $X \in \mathit{NEFTypeVars}$, T^{alph}, $\mathit{Free}(T) \subseteq \{X\}$, ϕ is a ways specifier with respect to T (see under 'Process types' in Section 2.4), and Γ and $\Gamma'_1, \ldots, \Gamma'_l$ are type contexts satisfying the restriction of data independence in X (see Section 2.7.1). Then a Γ,$(\Gamma'_1, \ldots, \Gamma'_l)$ normalised symbolic transition with respect to X, T and ϕ is an object α labelled by:

(i) a type context $\mathit{ys}(\alpha)$ in which all variables are of type X and which is disjoint from Γ;

(ii) a condition $\mathit{cond}(\alpha)$ which is either *True*, or built from equality tests between variables in $\Gamma \cup \mathit{ys}(\alpha)$ and the boolean connectives $\wedge$ and $\vee$;

(iii) a term $r(\alpha)$ which is an X,$\Gamma \cup \mathit{ys}(\alpha)$-DI evaluated form of type T (see Section 3.1.2) such that any variable from $\mathit{ys}(\alpha)$ occurs in $r(\alpha)$ exactly once;

(iv) conditions $\mathit{cond}'_1(\alpha), \ldots, \mathit{cond}'_l(\alpha)$, each of which is either *True*, or *False*, or built from equality tests between variables in $\Gamma \cup \mathit{ys}(\alpha)$ and the boolean connectives $\neg$, $\wedge$ and $\vee$;

such that:

(I) the variable names in $\mathit{ys}(\alpha)$ are the minimal ones which are not in Γ (see under 'Visible' in Section 3.2.2);

(II) $\Gamma \cup \mathit{ys}(\alpha) \supseteq \Gamma'_i$ for all i;

(III) if $r(\alpha)$ is of the form $\mathit{id}_i.(s_1, \ldots, s_{m_i})$, then for any $j \in \{1, \ldots, m_i\}$:

(III.a) if s_j contains a variable from Γ, then $! \in \phi(\mathit{id}_i, j)$;

(III.b) if s_j contains a variable from $\mathit{ys}(\alpha)$, then $\phi(\mathit{id}_i, j) \neq \{\}$;

(IV) $\mathit{cond}'_1(\alpha), \ldots, \mathit{cond}'_l(\alpha)$ disjointly partition

$$\mathit{cond}^\dagger(\alpha) \;=\; \bigwedge\{\neg x_{i'} = x_{j'} \mid 1 \leq i' < j' \leq k\} \;\wedge\; \mathit{cond}(\alpha)$$

where $\Gamma = \{x_1, \ldots, x_k\}$, i.e. for any assignment of values to the variables in $\Gamma \cup \mathit{ys}(\alpha)$ which satisfies $\mathit{cond}^\dagger(\alpha)$, exactly one $\mathit{cond}'_i(\alpha)$ is satisfied;

(V) for any i, $\mathit{cond}^\dagger(\alpha) \wedge \mathit{cond}'_i(\alpha)$ implies

$$\bigwedge\{\neg x'_{i,i'} = x'_{i,j'} \mid 1 \leq i' < j' \leq k'_i\}$$

where $\Gamma'_i = \{x'_{i,1}, \ldots, x'_{i,k'_i}\}$, i.e. for any assignment of values to the variables in $\Gamma \cup \mathit{ys}(\alpha)$ which satisfies $\mathit{cond}^\dagger(\alpha)$ and $\mathit{cond}'_i(\alpha)$, it assigns mutually distinct values to the variables in Γ'_i.

(The requirements (IV) and (V) are as given because the procedure for symbolic normalisation we shall present in Section 3.6.2 uses 'eager branching': see under 'Eager branching' in that section, and see steps (2.8) and (2.9) in that section.)

The transition α is written as

$$[[ys(\alpha)|cond(\alpha) \bullet r(\alpha) \begin{cases} cond'_1(\alpha)\rangle\rangle \\ \cdots \\ cond'_l(\alpha)\rangle\rangle \end{cases}$$

Informally, α is performed by

- choosing an assignment of values to the variables $ys(\alpha)$ for which the condition $cond(\alpha)$ is satisfied, then
- communicating the visible event $r(\alpha)$, and then
- if $cond'_i(\alpha)$ is satisfied, moving to the ith target node,

where it is irrelevant if the choice of the assignment is external or nondeterministic, since all information about refusal sets of a source node N of α is recorded in $AccSets(N)$ and $DivCond(N)$ (see above). (For a more formal definition, see Section 3.7.3.)

When considering the term $r(\alpha)$, we shall call the variables in Γ 'outputs', and the variables in $ys(\alpha)$ 'inputs' (for lack of a more neutral word).

For any normalised symbolic transition α, let

$$Vars(\alpha) = ys(\alpha)$$

■

3.6.2 A procedure for symbolic normalisation

In this section, we present a procedure for symbolic normalisation.

More precisely, we show how to symbolically normalise any SLTS $\mathcal{S}$ which is with respect to some X, T and ϕ (see Section 3.3.1), for which the type context of its initial node $Context(N_0)$ is empty, and whose every invisible transition is of the form $[\sqcap \{\}|True \bullet \tau\rangle$. According to 'Assumption of transformation' after Proposition 3.5.3, these restrictions are met by $\mathcal{S}_{\{\}\vdash P}$ for any process P which is data independent in X with no constants (see Section 2.7.1) and which satisfies the condition ($\mathbf{RecCho}_X$) (see Section 3.5.2).

The result of the procedure will be called

$$\mathcal{NS}(\mathcal{S})$$

and it will be an NSLTS with respect to X, T and ϕ in accordance with the general definition in Section 3.7.1.

The procedure is based on the procedure for normalising ordinary LTSs (see [Ros94a] or [Ros98a, Section C.1.2]), but the details are considerably more complex.

Omitting the type X. Since all variables in $\mathcal{S}$ and $\mathcal{NS}(\mathcal{S})$ will be of type X, we shall for simplicity be omitting X, and for example writing $y \in ys(\alpha)$ instead of $(y : X) \in ys(\alpha)$. ■

Overlining. To distinguish the components of $\mathcal{NS}(\mathcal{S})$ from those of $\mathcal{S}$, we shall be overlining some of the former. For example: $\overline{N}$, $\overline{ys}$, $\overline{cond}$. ■

Nodes of $\mathcal{NS}(\mathcal{S})$ and their type contexts. As we have mentioned at the beginning of Section 3.6.1, the nodes of $\mathcal{NS}(\mathcal{S})$ will be sets of nodes of $\mathcal{S}$. More precisely, since $\mathcal{S}$ is an SLTS, the nodes of $\mathcal{NS}(\mathcal{S})$ will have to be sets of nodes of $\mathcal{S}$ with renamed variables.

Thus a node $\overline{N}$ of $\mathcal{NS}(\mathcal{S})$ will be a set of ordered pairs of the form (N, f), where N is a node of $\mathcal{S}$ and f is a function from $\mathit{Context}(N)$ into variable names.

To make such an $\overline{N}$ a normalised symbolic node according to the definition in Section 3.6.1, we shall have to define its labels. Its type context, in which all variables are implicitly of type X, will be defined as

$$\mathit{Context}(\overline{N}) = \bigcup_{(N,f)\in\overline{N}} \mathit{Range}(f)$$

and the other two labels will be defined in the procedure. ■

Eager branching. During the procedure, we shall often need to resolve conditions built from equality tests between variables, partly due to conditional symbolic transitions in $\mathcal{S}$.

To do this, we shall use 'eager branching' as follows. Whenever new variables are introduced by a normalised symbolic transition, enough branching will be performed to completely determine their equality or inequality relationships with existing variables. Moreover, one variable will be substituted for any set of variables thus determined as equal.

Thus, for any node $\overline{N}$ of $\mathcal{NS}(\mathcal{S})$, we shall be able to treat all the variables in $\mathit{Context}(\overline{N})$ as mutually distinct. ■

Equality vs identity of variables. For equality tests within symbolic conditions, we shall as usual be using the equality sign, so that for example $x = y \wedge y = y'$ is a symbolic condition.

On the other hand, for identity of variables, to distinguish it from symbolic equality tests, we shall in this section be using the identity sign, so that for example $x \equiv x'$ means that x and x' are the same variable name. ■

Invisible paths and reachability. Supposing that N is a node of $\mathcal{S}$ and f is a function from $\mathit{Context}(N)$ into variable names, the following two definitions will be useful in the procedure:

(a) We say that a path $N_1\alpha_1N_2\alpha_2\ldots$ through $\mathcal{S}$ is an 'invisible path from (N, f)' if

- $N = N_1$,
- each α_i is a conditional or invisible transition, and
- for any α_j which is a conditional transition, $\mathit{cond}(\alpha_j)$ is satisfied by postulating that $x = x'$ if and only if $f(x) \equiv f(x')$ (see under 'Eager branching' and 'Equality vs identity of variables' above).

(b) We say that (N', f') is 'invisibly reachable from (N, f)' if there exists some invisible path $N_1\alpha_1N_2\alpha_2\ldots N_k$ from (N, f) with $N_k = N'$ and $f' = f \restriction \mathit{Context}(N')$.

■

We can now present the procedure for symbolic normalisation, i.e. for constructing $\mathcal{NS}(\mathcal{S})$ from $\mathcal{S}$:

(1) The procedure begins by forming the initial node $\overline{N}_0$ of $\mathcal{NS}(\mathcal{S})$. If N_0 is the initial node of $\mathcal{S}$ and $\{\}$ denotes the empty function, $\overline{N}_0$ is the set of all $(N', \{\})$ which are invisibly reachable from $(N_0, \{\})$ in $\mathcal{S}$ and which are such that N' has no conditional symbolic transitions.

$$\mathit{AccSets}(\overline{N}_0) = \left\{ \{\alpha \mid N'\alpha\} \;\middle|\; \begin{array}{c} (N', \{\}) \in \overline{N}_0 \\ \wedge\ N' \ \mathit{has\ no\ inv.\ symb.\ trans.} \end{array} \right\}$$

$\mathit{DivCond}(\overline{N}_0) = \mathit{True}$ if there is an infinite invisible path from $(N_0, \{\})$ in $\mathcal{S}$, and otherwise $\mathit{DivCond}(\overline{N}_0) = \mathit{False}$.

$\overline{N}_0$ is then marked as unexpanded.

(2) Suppose $\overline{N}$ is a node that has been formed but not expanded. If there are no such nodes, the procedure finishes.

Consider an $(N, f) \in \overline{N}$ and a visible transition α of N. Since $\mathit{cond}(\alpha)$ is a finite conjunction of equality tests between variables in $\mathit{Context}(N) \cup \mathit{ys}(\alpha)$ (see under 'Visible' in Section 3.2.2), we can let $\approx$ be the partial equivalence relation on $\mathit{Context}(N) \cup \mathit{ys}(\alpha)$ (i.e. an equivalence relation on a subset of it) induced by $\mathit{cond}(\alpha)$.

(2.1) If there exist $x, x' \in \mathit{Context}(N)$ such that $x \approx x'$ but $f(x) \not\equiv f(x')$, then α cannot be executed from (N, f). Hence we drop it from consideration.

(2.2) For any $y \in \mathit{ys}(\alpha)$ for which there is some $x \in \mathit{Context}(N)$ with $y \approx x$, let $g(y) \equiv f(x)$. (By the previous step, $f(x)$ must be the same for all such x.)

(2.3) For any equivalence class C which does not intersect $\mathit{Context}(N)$, pick a fresh variable name w_C not occuring in $\mathit{Context}(\overline{N})$, and define $g(y) \equiv w_C$ for all $y \in C$. (It will be important later that for any such C, some $y \in C$ must occur in $r(\alpha)$: see under 'Visible' in Section 3.2.2.)

(2.4) For each $y \in \mathit{ys}(\alpha)$ which is outside $\approx$, and for each $y \in \mathit{zs}(\alpha)$, define $g(y)$ to be a fresh variable name not occuring in $\mathit{Context}(\overline{N})$. In other words, they are simply renamed bijectively.

The result of performing (2.1)–(2.4) is the following: either α was dropped in (2.1), or we have constructed a renaming g of $\mathit{Vars}(\alpha)$ which completely incorporates $\mathit{cond}(\alpha)$ so that we may subsequently not pay attention to it, i.e. treat α as if it had $\mathit{cond}(\alpha) = \mathit{True}$. (It is important to note that this transformation is semantically valid because we are ignoring the difference between $\mathit{ys}(\alpha)$ and $\mathit{zs}(\alpha)$: since all information about nondeterminism at $\overline{N}$ will be recorded in $\mathit{AccSets}(\overline{N})$ and $\mathit{DivCond}(\overline{N})$, there is no need to record it in the transitions of $\overline{N}$, and consequently we can treat any $y \in \mathit{Vars}(\alpha)$ as just a variable ranging over the type X. The transformation would not in general be valid otherwise: once we recall the meaning of visible symbolic transitions (see under 'Visible' in Section 3.2.2, and Section 3.4.1), we see that choosing values for $\mathit{ys}(\alpha)$ which do not satisfy $\mathit{cond}(\alpha)$ results in an acceptance set not containing any event generated by α, whereas in the transformed α such a choice cannot be made.)

As we have said at the beginning of Section 3.6.1, transitions of $\overline{N}$ will be formed by merging visible transitions from $\mathcal{S}$ that can overlap. More precisely, $\overline{N}$ is going to have one normalised symbolic transition for each equivalence class of the following equivalence relation on the set of all renamed transitions (α, f, g) as above: (α, f, g) is related to (α', f', g') if the shells of $r(\alpha)$ and $r(\alpha')$ (see Section 3.1.2) are the same.

Any equivalence class $\mathcal{D}$ of that equivalence relation gives rise to a normalised symbolic transition

$$\overline{N} \quad [[\overline{ys}|\overline{cond} \;\bullet\; \overline{r} \;\left\{\begin{array}{l} \mathit{cond}'_1\rangle\rangle \quad \overline{N}'_1 \\ \cdots \\ \mathit{cond}'_l\rangle\rangle \quad \overline{N}'_l \end{array}\right.$$

(see under 'Normalised symbolic transitions' in Section 3.6.1) whose labels and target nodes are constructed as follows:

(2.5) The shell of $\overline{r}$ is defined to be the same as the shell $K[\cdot_1, \ldots, \cdot_k]$ of all the $r(\alpha)$ for $(\alpha, f, g) \in \mathcal{D}$.

Consider a place $\cdot_i$ in $\overline{r}$, which is to be occupied by a variable (see Section 3.1.2). For any $(\alpha, f, g) \in \mathcal{D}$, that place in $r(\alpha)$ is occupied by some x, which is renamed to $f(x)$ if $x \in \mathit{Context}(\mathit{source}(\alpha))$, or to $g(x)$ if $x \in \mathit{Vars}(\alpha)$. Let W be the set of all $f(x)$ or $g(x)$ thus obtained, i.e. as (α, f, g) ranges over $\mathcal{D}$.

If $W = \{x'\}$ for some $x' \in \mathit{Context}(\overline{N})$, we fill the place in $\overline{r}$ by x'. Otherwise, we fill it by a fresh variable. In other words, a place of type X in the normalised symbolic transition gets occupied by an output if it is occupied by the same output in each of the transitions that are being merged. Otherwise, it gets occupied by an input (see under 'Normalised symbolic transitions' in Section 3.6.1).

Let us record, for later use, some of the maps on variable names that are involved:

(2.5.1) Suppose $(\alpha, f, g) \in \mathcal{D}$. For any $y \in \mathit{Vars}(\alpha)$ for which $g(y) \notin \mathit{Context}(\overline{N})$ (i.e. which is not required by $\mathit{cond}(\alpha)$ to be equal to an output), we know that there is some $y' \in \mathit{Vars}(\alpha)$ such that $y \approx y'$ and hence $g(y) \equiv g(y')$ and such that y' occurs in $r(\alpha)$ (see under 'Visible' in Section 3.2.2), as we have already partially observed in (2.3). Let

$$\widehat{g}$$

denote the modification of g obtained by mapping each such y to the fresh variable we have put in $\overline{r}$ in the place which is occupied by y' in $r(\alpha)$. (If there is more than one possible y' and consequently more than one possible fresh variable in $\overline{r}$, choose one.)

More informally, for any nondeterministic selection or input in α that is not required to be equal to an output, $\widehat{g}$ maps it to an input in the normalised symbolic transiton that occupies its place.

(2.5.2) For any $(\alpha, f, g) \in \mathcal{D}$ and any fresh variable y we have put in $\overline{r}$, if the corresponding place in $r(\alpha)$ is occupied by some x, let

$$h_{(\alpha,f,g)}(y) \equiv \left\{\begin{array}{ll} f(x), & \mathit{if}\ x \in \mathit{Context}(\mathit{source}(\alpha)) \\ g(x), & \mathit{if}\ x \in \mathit{Vars}(\alpha) \end{array}\right.$$

In other words, $h_{(\alpha,f,g)}$ maps y to the variable that is in its place in the renamed transition (α, f, g).

(2.6) $\overline{ys}$ is the set of all the fresh variables introduced in the previous step, whose names should be picked to be the minimal ones that do not occur in $\mathit{Context}(\overline{N})$ (see under 'Visible' in Section 3.2.2). In particular, $\mathit{Context}(\overline{N})$ and $\overline{(}ys)$ are disjoint.

(2.7) $\overline{\mathit{cond}}$ constrains the variables in $\overline{ys}$ so that the normalised symbolic transition really corresponds to a merging (or 'union') of all the transitions in $\mathcal{D}$. Formally,

$$\overline{\mathit{cond}} = \bigvee\{\zeta_{(\alpha,f,g)} \mid (\alpha, f, g) \in \mathcal{D}\}$$

where

$$\begin{aligned}\zeta_{(\alpha,f,g)} \;=\; & \bigwedge\{y = h_{(\alpha,f,g)}(y) \mid y \in \overline{ys} \,\wedge\, h_{(\alpha,f,g)}(y) \in \mathit{Context}(\overline{N})\} \\ & \wedge \bigwedge\left\{ y = y' \;\middle|\; \begin{array}{c} y, y' \in \overline{ys} \,\wedge\, y \not\equiv y' \\ \wedge\, h_{(\alpha,f,g)}(y), h_{(\alpha,f,g)}(y') \notin \mathit{Context}(\overline{N}) \\ \wedge\, h_{(\alpha,f,g)}(y) \equiv h_{(\alpha,f,g)}(y') \end{array} \right\}\end{aligned}$$

(It is clear that the two indexed $\bigwedge$ range over finite sets. Although $\mathcal{D}$, the set over which the indexed $\bigvee$ ranges, may be infinite (see under 'Infinite $\mathcal{S}$' below), there are only finitely many possibilities for the $\zeta_{(\alpha,f,g)}$, so that the whole formula $\overline{\mathit{cond}}$ is always finite.)

Thus $\overline{\mathit{cond}}$ demands the normalised symbolic transition to be compatible with at least one transition $(\alpha, f, g) \in \mathcal{D}$, which is to say that at least one $\zeta_{(\alpha,f,g)}$ be satisfied. In the definition of each $\zeta_{(\alpha,f,g)}$, the compatibility with (α, f, g) consists of two parts:

* any new variable of the normalised symbolic transition which occurs in the place of an output in (α, f, g) is required to have the same value as that output, by a subformula $y = h_{(\alpha,f,g)}(y)$, and
* any two new variables which occur in places occupied by a single nondeterministic selection or input in (α, f, g) are required to have equal values, by a subformula $y = y'$.

(2.8) Any cond'_i is a condition completely determining the equality or inequality relationships of the new variables (i.e. $\overline{ys}$), both among themselves and with the old variables (i.e. $\mathit{Context}(\overline{N})$). The cond'_i cover all the possibilities that are consistent with $\overline{\mathit{cond}}$. (Since all the equality or inequality relationships among $\mathit{Context}(\overline{N}) \cup \overline{ys}$ are completely determined by any cond'_i, saying that cond'_i is consistent with $\overline{\mathit{cond}}$ is the same as saying that $\overline{\mathit{cond}}$ satisfies cond'_i.)

More precisely, each condition cond'_i corresponds to a partial equivalence relation $\approx_i$ on $\mathit{Context}(\overline{N}) \cup \overline{ys}$ which covers $\overline{ys}$ and is such that any equivalence class contains at most one variable from $\mathit{Context}(\overline{N})$. The correspondence is the following: cond'_i specifies that $x = x'$ whenever $x \approx_i x'$, and that $\neg x = x'$ whenever $x \not\approx_i x'$. (We omit exact definitions of the cond'_i, which are now straightforward.)

(2.9) Any target normalised symbolic node $\overline{N}'_i$ can be formed as follows.

Observe first that $\overline{N}'_i$ is the node we reach by executing the normalised symbolic transition and following the cond'_i branch, which represents the execution of any $(\alpha, f, g) \in \mathcal{D}$ such that cond'_i is consistent with $\zeta_{(\alpha,f,g)}$ (see (2.7)), i.e. such that $\zeta_{(\alpha,f,g)}$ satisfies cond'_i.

Hence $\overline{N}'_i$ should be the nondeterministic choice of all the nodes to which such (α, f, g) lead. More precisely, let Z be the set of all ordered pairs

$$(\mathit{target}(\alpha), (f \cup \widehat{g}) \restriction \mathit{Context}(\mathit{target}(\alpha)))$$

(see (2.5.1)) for any (α, f, g) as above.

It remains to fulfil our commitment of being able to treat the free variables at any node of $\mathcal{NS}(\mathcal{S})$ as mutually distinct (see under 'Eager branching' above). To achieve that, choose one variable from each equivalence class of $\approx_i$,[4] and for any $x \in \mathit{Context}(\overline{N}) \cup \overline{ys}$ let $q(x)$ be the variable chosen from the equivalence class of x if x is covered by $\approx_i$, and let $q(x) \equiv x$ otherwise. (Observe that, for any (α, f, g) as above and for any $y \in \mathit{Vars}(\alpha)$ for which there was more than one possible variable for $\widehat{g}(y)$ in (2.5.1), $\zeta_{(\alpha,f,g)}$ requires all those variables to have equal values, and so since $\zeta_{(\alpha,f,g)}$ satisfies cond'_i, the variables are all mapped to the same one by q. This means that, once we perform q after $\widehat{g}$, the choice will disappear.)

We can now let $\overline{N}'_i$ be the set of all (N'', f'') which are invisibly reachable from some element of the set

$$Z' = \{(N', q \circ f') \mid (N', f') \in Z\}$$

and which are such that N'' has no conditional symbolic transitions.

Suppose $(N'', f'') \in \overline{N}'_i$ is such that N'' has no invisible symbolic transitions, and α is a transition (necessarily visible) of N''. Let α' be a fresh visible symbolic transition obtained from α as follows:

* We rename the variables from $\mathit{Context}(N'')$ by f''. (Recalling that we can treat the variables from $\mathit{Context}(\overline{N}'_i)$, and in particular from the range of f'' as being mutually distinct, this renaming might enable us to simplify $\mathit{cond}(\alpha')$ compared with $\mathit{cond}(\alpha)$.)
* We rename the variables from $\mathit{Vars}(\alpha)$ bijectively to ensure that the variable names in $\mathit{Vars}(\alpha')$ are the minimal ones not occuring in $\mathit{Context}(\overline{N}'_i)$.

Then let $\mathcal{A}(N'', f'')$ be the set of all α' as α ranges over the transitions of N'', and let

$$\mathit{AccSets}(\overline{N}'_i) = \left\{ \mathcal{A}(N'', f'') \;\middle|\; \begin{array}{c} (N'', f'') \in \overline{N}'_i \\ \wedge\ N''\ \mathit{has\ no\ inv.\ symb.\ trans.} \end{array} \right\}$$

Let $\mathit{DivCond}(\overline{N}'_i) = \mathit{True}$ if there is an infinite invisible path from some $(N'', f'') \in \overline{N}'_i$, or equivalently from some element of Z'. Otherwise, let $\mathit{DivCond}(\overline{N}'_i) = \mathit{False}$.

(2.10) The node $\overline{N}$ is marked as expanded.

(3) Whenever a node is formed in (2), we check whether it had been formed before, and if it had the old copy is reused. In other words, we avoid having separate nodes that are equal as sets.

The nodes that are genuinely new are marked as unexpanded, and we go back to step (2).

That completes the procedure.

Infinite $\mathcal{S}$. The first thing to remark is that the procedure is applicable not only to finite $\mathcal{S}$, but also to infinite. Namely, although we have presented it as an executable algorithm, it is easy to see that the procedure is a general recipe for constructing $\mathcal{NS}(\mathcal{S})$ which does not depend on $\mathcal{S}$ being finite.

In fact, even in practice (see Sections 3.8 and Section 5.6), one may very well want to apply the procedure lazily to an infinite $\mathcal{S}$ on the basis that only a finite portion of $\mathcal{NS}(\mathcal{S})$ may be

[4]Always choosing the one with the minimal name may result in $\mathcal{NS}(\mathcal{S})$ having fewer nodes.

needed in a refinement check, in which case whether the procedure terminates as an algorithm is irrelevant.

For finite $\mathcal{S}$, sufficient conditions for termination of the procedure when executed as an algorithm will be given in Section 3.7.2. ■

Fixing a model beforehand. As we shall formally see in Section 3.7.3, the resulting NSLTS $\mathcal{NS}(\mathcal{S})$ has the same denotational semantic value as the SLTS $\mathcal{S}$ in any of the three denotational models (see Section 2.1.2).

Alternatively, especially in practice, one may choose to fix a model beforehand and have the procedure pay attention only to those aspects of $\mathcal{NS}(\mathcal{S})$ that are relevant for that model. For example, the procedure may not record the $\mathit{AccSets}(\overline{N})$ and $\mathit{DivCond}(\overline{N})$ in the case of the finite-traces model, or it may not expand the $\overline{N}$ for which $\mathit{DivCond}(\overline{N}) = \mathit{True}$ in the case of the failures/divergences model.

The main differences of the alternative approach is that some quantities which measure $\mathcal{NS}(\mathcal{S})$ may be smaller than before, and that the procedure might terminate when executed as an algorithm in some cases in which it would not have done so before. ■

The $\mathcal{NS}(\mathcal{S})$ are not 'normal forms'. We should stress that the resulting NSLTSs $\mathcal{NS}(\mathcal{S})$ are 'normalised' only in the sense that they can execute any trace in at most one way.

In other words, in contrast to the normalisation of ordinary LTSs (see [Ros94a] or [Ros98a, Section C.1.2]), the NSLTSs constructed by the procedure we have presented do not in general have the property of being in some sense unique representations of denotational semantic values. Indeed, in the terminology of [Ros94a] and [Ros98a, Section C.1.2], what we have presented is a procedure for symbolic *pre-normalisation*. We have not attempted to achieve a uniqueness property because it is not necessary for the applications in this thesis (see Section 5.4), and because it seems to require further transformations that are much more complex than in the nonsymbolic case.

In particular, elements of the labels $\mathit{AccSets}(\overline{N})$ are not required to be in any sense minimal. However, it might be advantageous in practice to reduce the $\mathit{AccSets}(\overline{N})$ as much as possible, which can be done in the following two ways:

- If any refusal set generated by some $\mathcal{A} \in \mathit{AccSets}(\overline{N})$ is a subset of some refusal set generated by some other element of $\mathit{AccSets}(\overline{N})$, then $\mathcal{A}$ can be removed from $\mathit{AccSets}(\overline{N})$ without affecting the denotational semantic values corresponding to $\mathcal{NS}(\mathcal{S})$ (see Section 3.7.3).
- Similarly, for any $\mathcal{A} \in \mathit{AccSets}(\overline{N})$, redundant $\alpha \in \mathcal{A}$ can be removed.

■

Less restricted $\mathcal{S}$. Although some more technical parts of the procedure have relied heavily on the precise structure of $\mathcal{S}$ which was specified at the beginning of this section, the procedure can be modified to apply to structurally less well behaved $\mathcal{S}$ and hence to less restricted processes. It may indeed be necessary to do so if we want to cover more complex languages, in particular CSP$_M$ [Sca92, Sca98, Ros98a].

Most notably, for visible symbolic transitions α in $\mathcal{S}$, it suffices that whenever α is reached during execution and values satisfying $\mathit{cond}(\alpha)$ are chosen for $\mathit{Vars}(\alpha)$, they must all occur in $r(\alpha)$. To accommodate such $\mathcal{S}$, the procedure can be modified in the following two main ways (say):

- Before merging visible symbolic transitions of $\mathcal{S}$ to form normalised symbolic transitions, i.e. in place of the steps (2.1)–(2.4), each visible symbolic transition α involved can be replaced by a finite number of transitions $\alpha'_1, \ldots, \alpha'_m$ as follows. Each α'_j is obtained from α by substituting for any variable in $\mathit{Vars}(\alpha)$ that does not explicitly occur in $r(\alpha)$ some variable from $\mathit{Context}(\mathit{source}(\alpha)) \cup \mathit{Vars}(\alpha)$ that does, and $\alpha'_1, \ldots, \alpha'_m$ cover all such combinations or at least those that result in satisfiable $\mathit{cond}(\alpha'_j)$.
- Each resulting transition α'_j has the property that all variables in $\mathit{Vars}(\alpha'_j)$ occur explicitly in $r(\alpha'_j)$, and so they can be merged in essentially the same way as in the procedure we have presented above. The main exception is that the $\mathit{cond}(\alpha'_j)$ have to be taken into account, by including them as conjuncts in the subformulae ζ that make up the condition $\overline{\mathit{cond}}$ of the normalised symbolic transition.

It also seems possible to extend the procedure to cover $\mathcal{S}$ in which, for any invisible symbolic transition α, it is not necessary that $\mathit{ys}(\alpha) = \{\}$ and $\mathit{cond}(\alpha) = \mathit{True}$, but only that for every visible symbolic transition α' reachable from α without performing any intermediate visible symbolic transitions, all the variables from $\mathit{ys}(\alpha)$ occur in $r(\alpha')$ (or perhaps only that in any execution all their values occur in $r(\alpha')$). However, specification processes which do not satisfy ($\mathbf{RecCho}_X$) but whose SLTSs are of this form seem rare, and also this relaxed condition on $\mathcal{S}$ is not in general preserved by constructs like external choices and parallel composition. ■

Lazy vs eager branching. In the procedure as presented, the branchings at the ends of normalised symbolic transitions are 'eager' (see under 'Eager branching' at the beginning of this section, and see steps (2.8) and (2.9) of the procedure).

The alternative approach of 'lazy branching', which may be preferable in practice especially when $\mathcal{S}$ contains few conditional symbolic transitions, is as follows. At the end of any normalised symbolic transition, we perform just enough branching to determine which of the visible symbolic transitions that were merged any particular execution of the normalised symbolic transition can correspond to, and to be able to resolve all the conditional symbolic transitions that are reachable from the determined transitions without performing any further visible symbolic transitions.

Although lazy branching can usually be used as successfully as eager branching, it has a drawback of being in a sense less precise. In particular, some quantites that measure the resulting $\mathcal{NS}(\mathcal{S})$ can be larger than with eager branching, and consequently some results which are true with eager branching may be false or at least require revised statements. Most notably, this is the case with the quantity $W(\mathcal{NS}(\mathcal{S}))$ and with Proposition 3.7.5 (see Section 3.7.2). ■

3.7 NSLTSs

In this section, we define normalised symbolic labelled transition systems (NSLTSs) in general and give an example, then state a few results about finiteness of NSLTSs constructed by the procedure for symbolic normalisation (see Section 3.6.2), and finally examine their relationships with the denotational semantics and with the SLTSs they can be constructed from.

3.7.1 Definition of NSLTSs in general

Definition 3.7.1 Suppose $X \in \mathit{NEFTypeVars}$, T^{alph}, $\mathit{Free}(T) \subseteq \{X\}$, and ϕ is a ways specifier with respect to T (see under 'Process types' in Section 2.4). $\mathcal{NS}$ is said to be an NSLTS with respect to X, T and ϕ if and only if it is a tuple

$$(\mathcal{N}, N_0, \mathcal{T}, \mathit{source}, \mathit{target})$$

such that:

(i) $\mathcal{N}$ is a set of normalised symbolic nodes with respect to X, T and ϕ (see Section 3.6.1).

(ii) $N_0 \in \mathcal{N}$, it is called 'the initial node', and $\mathit{Context}(N_0) = \{\}$.

(iii) $\mathcal{T}$ is a set of normalised symbolic transitions with respect to X, T and ϕ (see Section 3.6.1).

(iv) For any $\alpha \in \mathcal{T}$, if α is a Γ,$(\Gamma'_1, \ldots, \Gamma'_l)$ normalised symbolic transition, then $\mathit{source}(\alpha)$ specifies the source node of α which must be such that $\mathit{Context}(\mathit{source}(\alpha)) = \Gamma$, and $\mathit{target}(\alpha, 1)$, $\ldots$, $\mathit{target}(\alpha, l)$ specify the target nodes of α which must be such that $\mathit{Context}(\mathit{target}(\alpha, i)) = \Gamma'_i$ for all i.

(v) Any $N \in \mathcal{N}$ can be reached from N_0, and has countably many normalised symbolic transitions.

(vi) For any two different $\alpha, \alpha' \in \mathcal{T}$ with $\mathit{source}(\alpha) = \mathit{source}(\alpha')$, the shells of $r(\alpha)$ and $r(\alpha')$ (see Section 3.1.2) must be different.

■

As we shall see in Section 3.7.3, Definition 3.7.1 (vi) and the fact that for any normalised symbolic transition α all the variables $\mathit{ys}(\alpha)$ occur in $r(\alpha)$ ensure the key property of NSLTSs that they can execute any trace in at most one way.

Notation. We shall write $N\alpha$ or $N(\alpha, i)$ to mean that $\mathit{source}(\alpha) = N$, and $(\alpha, i)N'$ to mean that $\mathit{target}(\alpha, i) = N'$. ■

NSLTSs constructed by symbolic normalisation. Given an SLTS $\mathcal{S}$ which is with respect to some X, T and ϕ (see Section 3.3.1), for which the type context of its initial node $\mathit{Context}(N_0)$ is empty, and whose every invisible transition is of the form $[\sqcap \{\} | \mathit{True} \bullet \tau\rangle$, the structure $\mathcal{NS}(\mathcal{S})$ constructed by the procedure for symbolic normalisation in Section 3.6.2 is indeed an NSLTS with respect to X, T and ϕ.

Thus, given an X-DI process P which satisfies ($\mathbf{RecCho}_X$) (see Sections 2.7.1 and 3.5.2), the procedure can be applied to $\mathcal{S}_{\{\}\vdash P}$ (see Section 3.3.1 and under 'Assumption of transformation' after Proposition 3.5.3) to construct an NSLTS

$$\mathcal{NS}(\mathcal{S}_{\{\}\vdash P})$$

which we shall call 'the NSLTS of P'. ■

Example 3.7.1 For example, recall the process *NondReg* from Example 2.5.1, which is X-DI and satisfies ($\mathbf{RecCho}_X$), and recall its SLTS $\mathcal{S}_{\{\}\vdash NondReg}$ from Example 3.3.1.

An initial portion of its NSLTS $\mathcal{NS}(\mathcal{S}_{\{\}\vdash NondReg})$ is shown in Figure 3.4. As in the procedure for symbolic normalisation, the normalised symbolic nodes of $\mathcal{NS}(\mathcal{S}_{\{\}\vdash NondReg})$ are denoted by $\overline{N}_i$ to distinguish them from the symbolic nodes N_j of $\mathcal{S}_{\{\}\vdash NondReg}$. Each $\overline{N}_i$ is represented as a rectangle consisting of three parts which display its three labels $\mathit{Context}(\overline{N}_i)$, $\mathit{AccSets}(\overline{N}_i)$ and $\mathit{DivCond}(\overline{N}_i)$.

As indicated in Figure 3.4, $\mathcal{NS}(\mathcal{S}_{\{\}\vdash NondReg})$ is infinite in spite of the fact that $\mathcal{S}_{\{\}\vdash NondReg}$ is finite. (See Proposition 3.7.2.) This is because, after consecutively performing any number k of inputs $in?x_1, \ldots, in?x_k$ in which all the x_i have distinct values, *NondReg* can nondeterministically be any of the processes $\mathit{NondReg}'(x_1), \ldots, \mathit{NondReg}(x_k)$. Consequently, the normalised symbolic node of $\mathcal{NS}(\mathcal{S}_{\{\}\vdash NondReg})$ reached by the corresponding k normalised symbolic transitions must be the nondeterministic choice of the symbolic nodes of $\mathcal{S}_{\{\}\vdash NondReg}$ that correspond to the processes $\mathit{NondReg}'(x_1), \ldots, \mathit{NondReg}(x_k)$, and so it is different from any node that preceds it on that path.

More specifically, in the notations of Figures 3.1 and 3.4, and of the procedure for symbolic normalisation (see Section 3.6.2), the normalised symbolic nodes reachable by normalised symbolic transitions which correspond to consecutively performing mutually distinct inputs are

$$\begin{array}{rcl}
\overline{N}_0 & = & \{(N_0, \{\})\} \\
\overline{N}_1 & = & \{(N_1, \{x \mapsto x\})\} \\
\overline{N}_2 & = & \left\{ \begin{array}{c} (N_3, \{x \mapsto x, y \mapsto y\}), \\ (N_1, \{x \mapsto x\}), (N_4, \{y \mapsto y\}) \end{array} \right\} \\
\overline{N}_4 & = & \left\{ \begin{array}{c} (N_3, \{x \mapsto x, y \mapsto z\}), \\ (N_1, \{x \mapsto x\}), (N_4, \{y \mapsto z\}), \\ (N_6, \{y \mapsto y, x \mapsto z\}) \\ (N_4, \{y \mapsto y\}), (N_1, \{x \mapsto z\}) \end{array} \right\} \\
\vdots & & \vdots
\end{array}$$

where for any u the renamed symbolic nodes $(N_1, \{x \mapsto u\})$ and $(N_4, \{y \mapsto u\})$ correspond to the process $\mathit{NondReg}'(u)$, and for any u and v the renamed symbolic nodes $(N_3, \{x \mapsto u, y \mapsto v\})$ and $(N_6, \{y \mapsto u, x \mapsto v\})$ correspond to the process $\mathit{NondReg}'(u) \sqcap \mathit{NondReg}'(v)$. ■

3.7.2 Finiteness

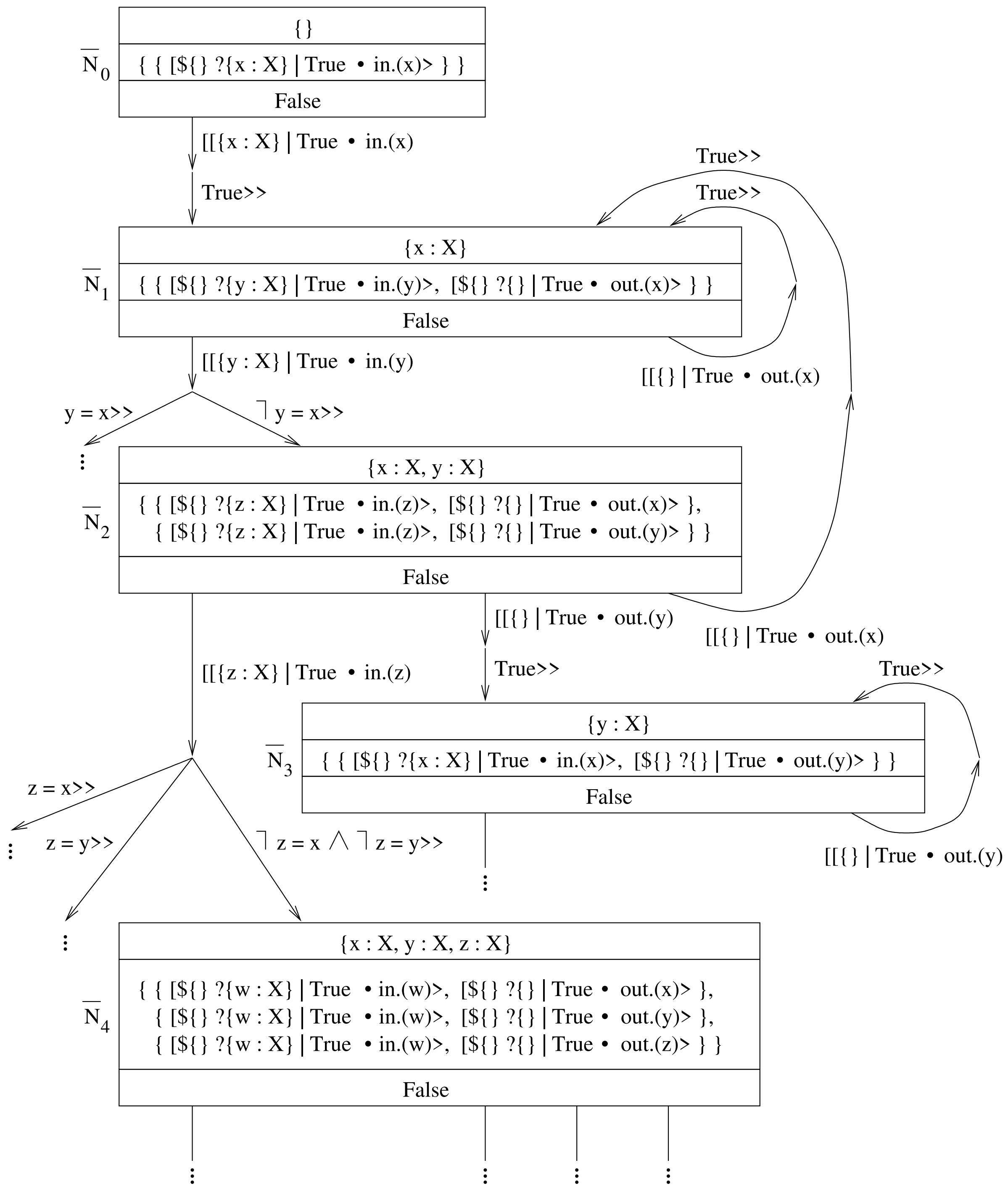
Let us first define two quantities which measure NSLTSs, and which will be important both in this section and in Section 5.4. (Corresponding quantities for SLTSs and ASLTSs were defined in Section 3.3.3.)

Definition 3.7.2 For any NSLTS $\mathcal{NS} = (\mathcal{N}, N_0, \mathcal{T}, \mathit{source}, \mathit{target})$, let

$$\begin{array}{rcl}
W(\mathcal{NS}) & = & \sup\{|\mathit{Context}(N)| \mid N \in \mathcal{N}\} \\
L(\mathcal{NS}) & = & \sup\{|ys(\alpha)| \mid \alpha \in \mathcal{T}\}
\end{array}$$

where the supremum of an empty set is 0, and the supremum of an unbounded set is ω (the smallest infinite cardinal). ■

Proposition 3.7.1 *In any NSLTS $\mathcal{NS}(\mathcal{S}_{\{\}\vdash P})$, for any node N and any $\mathcal{A} \in \mathit{AccSets}(N)$ we have that*



{}
$\overline{N}_0$
{ { [${} ?{x : X} | True • in.(x)> } }
False
[[{x : X} | True • in.(x)
True>>
{x : X}
$\overline{N}_1$
{ { [${} ?{y : X} | True • in.(y)>, [${} ?{} | True • out.(x)> } }
False
True>>
True>>
[[{} | True • out.(x)
[[{y : X} | True • in.(y)
y = x>>
⌉ y = x>>
{x : X, y : X}
$\overline{N}_2$
{ { [${} ?{z : X} | True • in.(z)>, [${} ?{} | True • out.(x)> },
{ [${} ?{z : X} | True • in.(z)>, [${} ?{} | True • out.(y)> } }
False
[[{} | True • out.(y)
True>>
[[{} | True • out.(x)
[[{z : X} | True • in.(z)
True>>
{y : X}
$\overline{N}_3$
{ { [${} ?{x : X} | True • in.(x)>, [${} ?{} | True • out.(y)> } }
False
[[{} | True • out.(y)
z = x>>
z = y>>
⌉ z = x ∧ ⌉ z = y>>
{x : X, y : X, z : X}
$\overline{N}_4$
{ { [${} ?{w : X} | True • in.(w)>, [${} ?{} | True • out.(x)> },
{ [${} ?{w : X} | True • in.(w)>, [${} ?{} | True • out.(y)> },
{ [${} ?{w : X} | True • in.(w)>, [${} ?{} | True • out.(z)> } }
False

Figure 3.4: An initial portion of $\mathcal{NS}(\mathcal{S}_{\{\}\vdash NondReg})$

(a) there are at most finitely many $\alpha \in \mathcal{A}$ whose events $r(\alpha)$ have any fixed shell H, and

(b) there are at most finitely many $\alpha \in \mathcal{A}$ whose sets $ys(\alpha)$ of nondeterministic selections are not empty.

Proof. By Proposition 3.3.1. ■

The result of normalising a finite ordinary LTS is always finite (see [Ros94a] or [Ros98a, Section C.1.2]). However, as we saw in Example 3.7.1, symbolic normalisation of a finite SLTS can yield an infinite NSLTS. The following proposition examines this interesting phenomenon:

Proposition 3.7.2 *Suppose $\mathcal{S}$ is an SLTS which satisfies the assumptions of the procedure for symbolic normalisation (see Section 3.6.2), and which is finite (i.e. has finitely many nodes and transitions). Then:*

(a) For any node $\overline{N}$ of $\mathcal{NS}(\mathcal{S})$, $AccSets(\overline{N})$ is a finite set of finite sets.

(b) $\mathcal{NS}(\mathcal{S})$ is finitely branching (i.e. any node of $\mathcal{NS}(\mathcal{S})$ has finitely many transitions).

(c) If $W(\mathcal{NS}(\mathcal{S}))$ is finite, then $\mathcal{NS}(\mathcal{S})$ is finite.

(d) If $\mathcal{NS}(\mathcal{S})$ is finite, then the procedure for symbolic normalisation terminates for $\mathcal{S}$ when executed as an algorithm.

Proof. (a) and (b) follow from the fact that any node $\overline{N}$ of $\mathcal{NS}(\mathcal{S})$ is a set of finitely many renamed nodes of $\mathcal{S}$, which can easily be shown by induction on the minimal length of a path from the initial node $\overline{N}_0$ to $\overline{N}$.

(c) is proved similarly to Proposition 3.3.3 (i). Namely, $L(\mathcal{NS}(\mathcal{S}))$ is finite by the finiteness of $\mathcal{S}$, and so any node of $\mathcal{NS}(\mathcal{S})$ is a subset of the finite set

$$\left\{ (N, f) \;\middle|\; \begin{array}{c} N \textit{ is a node of } \mathcal{S} \land \\ f : \mathit{Context}(N) \to \{(x_1 : X), \ldots, (x_{W(\mathcal{NS}(\mathcal{S}))+L(\mathcal{NS}(\mathcal{S}))} : X)\} \end{array} \right\}$$

where $x_1, \ldots, x_{W(\mathcal{NS}(\mathcal{S}))+L(\mathcal{NS}(\mathcal{S}))}$ are the first $W(\mathcal{NS}(\mathcal{S})) + L(\mathcal{NS}(\mathcal{S}))$ in the enumeration of all possible variable names.

(d) is straightforward. ■

It would therefore be very useful to be able to determine $W(\mathcal{NS}(\mathcal{S}))$ without first constructing $\mathcal{NS}(\mathcal{S})$: two possibilities for this will be offered in Proposition 3.7.5.

3.7.3 Relationships with denotational semantics and SLTSs

Suppose $\mathcal{NS}$ is an NSLTS with respect to some X, T and ϕ. To define the denotational semantic values corresponding to $\mathcal{NS}$, we follow the same pattern as we did for SLTSs and ASLTSs in Section 3.4.1.

For any δ (which assigns a nonempty set to X), an *execution* in $\mathcal{NS}$ with respect to δ is a sequence

$$N_1^{\eta_1}(\alpha_1, i_1)^{\eta'_1} N_2^{\eta_2}(\alpha_2, i_2)^{\eta'_2} \ldots$$

such that

- it either ends with some $N_k^{\eta_k}$ or it is infinite,
- $N_1(\alpha_1, i_1)N_2(\alpha_2, i_2)\ldots$ is a path (see under 'Notation' after Definition 3.7.1),
- each η_j is an assignment to $\mathit{Context}(N_j)$ with respect to δ,
- the values assigned by η_1 to the variables in $\mathit{Context}(N_1)$ are mutually distinct,
- each η'_j is an assignment to $\mathit{ys}(\alpha_j)$ with respect to δ, for which $\eta_j \cup \eta'_j$ satisfies $\mathit{cond}(\alpha_j)$ and $\mathit{cond}'_{i_j}(\alpha_j)$, and
- each η_{j+1} equals $(\eta_j \cup \eta'_j) \restriction \mathit{Context}(N_{j+1})$.

By the requirement (V) under 'Normalised symbolic transitions' in Section 3.6.1, it follows that the values assigned by any η_{j+1} to the variables in $\mathit{Context}(N_{j+1})$ are also mutually distinct.

As with SLTSs and ASLTSs, we shall sometimes use fragments of the notation for executions, and also their negations, so that for example $\neg N_1^{\eta_1}(\alpha_1, i_1)^{\eta'_1}$ will mean that there do not exist N_2 and η_2 such that $N_1^{\eta_1}(\alpha_1, i_1)^{\eta'_1}N_2^{\eta_2}$ is an execution.

If an execution is finite, so that it ends with some $N_k^{\eta_k}$, it generates a trace

$$t = \langle [\![r(\alpha_j)]\!]_{\delta,\eta_j \cup \eta'_j} \mid j \in \langle 1, \ldots, k-1 \rangle \rangle$$

and we write

$$N_1^{\eta_1} \overset{t}{\Longrightarrow} N_k^{\eta_k} \;\; \mathit{w.r.t.}\ \delta$$

We can then define the set of all traces corresponding to a node as before:

$$\mathit{traces}(N)_{\delta,\eta} = \{t \mid \exists\, N', \eta'. N^{\eta} \overset{t}{\Longrightarrow} N'^{\eta'} \;\; \mathit{w.r.t.}\ \delta\}$$

As we have remarked when defining normalised symbolic nodes in Section 3.6.1, the refusals of a node N are determined by its second label $\mathit{AccSets}(N)$. More precisely, we write

$$N'^{\eta'} \;\; \mathit{ref}^{\mathcal{A},\eta^*} \;\; A \;\; \mathit{w.r.t.}\ \delta$$

provided

- η' is an assignment of mutually distinct values to $\mathit{Context}(N')$ with respect to δ,
- $\mathcal{A} \in \mathit{AccSets}(N)$,
- $\eta^*(\alpha)$ is an assignment to $\mathit{ys}(\alpha)$ with respect to δ, for every $\alpha \in \mathcal{A}$, where Proposition 3.7.1 (b) tells us that there are at most finitely many such α with nonempty $\mathit{ys}(\alpha)$, and
- A is disjoint from the set of all

  $$[\![r(\alpha)]\!]_{\delta,\eta' \cup \eta^*(\alpha) \cup \eta''}$$

  as α ranges over $\mathcal{A}$, and η'' ranges over the assignments to $\mathit{zs}(\alpha)$ with respect to δ for which $\eta' \cup \eta^*(\alpha) \cup \eta''$ satisfies $\mathit{cond}(\alpha)$.

The set of all failures of a node can now be defined as before:

$$\begin{aligned} \mathit{failures}(N)_{\delta,\eta} = \{(t,A) \mid \exists\, N',\eta',\mathcal{A},\eta^*.\ & N^{\eta} \overset{t}{\Longrightarrow} N'^{\eta'}\ \mathit{w.r.t.}\ \delta \\ & \wedge N'^{\eta'}\ \mathit{ref}^{\mathcal{A},\eta^*}\ A\ \mathit{w.r.t.}\ \delta\} \end{aligned}$$

Using the third labels $\mathit{DivCond}(N')$, the set of all divergences of a node can be defined simply as

$$\begin{aligned} \mathit{divergences}(N)_{\delta,\eta} = \{t\hat{\ }u \mid \exists\, N',\eta'.\ & N^{\eta} \overset{t}{\Longrightarrow} N'^{\eta'}\ \mathit{w.r.t.}\ \delta \\ & \wedge \mathit{DivCond}(N')\} \end{aligned}$$

and as before

$$\mathit{failures}_{\perp}(N)_{\delta,\eta} = \mathit{failures}(N)_{\delta,\eta}\ \cup\ \{(t,A) \mid t \in \mathit{divergences}(N)_{\delta,\eta}\}$$

The denotational semantic values corresponding to $\mathcal{NS}$ are then those that correspond to its initial node:

$$\begin{aligned} [\![\mathcal{NS}]\!]^{\mathrm{Tr}}_{\delta,\{\}} &= \mathit{traces}(N_0)_{\delta,\{\}} \\ [\![\mathcal{NS}]\!]^{\mathrm{Fa}}_{\delta,\{\}} &= (\mathit{traces}(N_0)_{\delta,\{\}}, \mathit{fails}(N_0)_{\delta,\{\}}) \\ [\![\mathcal{NS}]\!]^{\mathrm{FD}}_{\delta,\{\}} &= (\mathit{failures}_{\perp}(N_0)_{\delta,\{\}}, \mathit{divs}(N_0)_{\delta,\{\}}) \end{aligned}$$

We can now formalise the key property of NSLTSs, namely that they can execute any trace in at most one way:

Proposition 3.7.3 *For any NSLTS $\mathcal{NS}$, any δ and any t, there is at most one execution from the initial node in $\mathcal{NS}$ with respect to δ which gives rise to t.*

Proof. Straightforward, by induction on the length of t. See under 'Normalised symbolic transitions' in Section 3.6.1, under 'Eager branching' in Section 3.6.2, and Definition 3.7.1 (vi). ■

The following theorem states that the procedure for symbolic normalisation given in Section 3.6.2 is semantically valid, i.e. that the denotational semantic values corresponding to any $\mathcal{NS}(\mathcal{S})$ are the same as those corresponding to $\mathcal{S}$. Together with the Congruence Theorem for symbolic operational semantics (see Section 3.4.2), this also means that the denotational semantic values corresponding to any $\mathcal{NS}(\mathcal{S}_{\{\}\vdash P})$ are the same as those of the process $\{\} \vdash P$.

Theorem 3.7.4 *(a) For any $\mathcal{NS}(\mathcal{S})$ and any M and δ,*

$$[\![\mathcal{NS}(\mathcal{S})]\!]^{M}_{\delta,\{\}} = [\![\mathcal{S}]\!]^{M}_{\delta,\{\}}$$

(b) For any $\mathcal{NS}(\mathcal{S}_{\{\}\vdash P})$ and any M and δ,

$$[\![\mathcal{NS}(\mathcal{S}_{\{\}\vdash P})]\!]^{M}_{\delta,\{\}} = [\![P]\!]^{M}_{\delta,\{\}}$$

Proof. (a) is shown by the following two facts, each of which can be proved by induction on the length of t:

(I) Whenever

$$\overline{N}_0^{\{\}} \overset{t}{\Longrightarrow} \overline{N}^{\eta} \ \textit{w.r.t.}\ \delta$$

in $\mathcal{NS}(\mathcal{S})$ and $(N, f) \in \overline{N}$, then

$$N_0^{\{\}} \overset{t}{\Longrightarrow} N^{\eta \circ f} \ \textit{w.r.t.}\ \delta$$

in $\mathcal{S}$.

(II) Whenever

$$N_0^{\{\}} \overset{t}{\Longrightarrow} N^{\theta} \ \textit{w.r.t.}\ \delta$$

in $\mathcal{S}$ and N has no conditional symbolic transitions, then there are some $\overline{N}$, η and f such that

$$\overline{N}_0^{\{\}} \overset{t}{\Longrightarrow} \overline{N}^{\eta} \ \textit{w.r.t.}\ \delta$$

in $\mathcal{NS}$, $(N, f) \in \overline{N}$ and $\eta \circ f = \theta$.

(b) follows from (a) and Theorem 3.4.1 (a). ■

Equivalence and transformations. In the same way as with SLTSs and ASLTSs, we can now define two NSLTSs which are both with respect to some X, T and ϕ to be equivalent if they have the same corresponding denotational semantic values, and we can study equivalence-preserving transformations. (See Section 3.4.3.) ■

As we have mentioned after Proposition 3.7.2, the following proposition offers two possibilities for determining $W(\mathcal{NS}(\mathcal{S}))$ (see Definition 3.7.2) without first constructing $\mathcal{NS}(\mathcal{S})$:

Proposition 3.7.5 *(a) For any $\mathcal{NS}(\mathcal{S})$ which is with respect to some X, T and ϕ, we have that*

$$W(\mathcal{NS}(\mathcal{S})) = \sup \left\{ V_{[X \mapsto n],t}(\mathcal{S}) \;\middle|\; \begin{array}{c} n \in \mathbb{Z}^+ \wedge \\ t \in [\![\mathcal{S}]\!]^{\mathrm{Tr}}_{[X \mapsto n],\{\}} \end{array} \right\}$$

where

$$V_{\delta,t}(\mathcal{S}) = \left| \bigcup \left\{ \mathit{Range}(\eta) \;\middle|\; \begin{array}{c} N_0^{\{\}} \overset{t}{\Longrightarrow} N^{\eta} \ \textit{w.r.t.}\ \delta \wedge \\ N \textit{ has no cond. symb. trans.} \end{array} \right\} \right|$$

where N_0 is the initial node of $\mathcal{S}$.

(For the definition of $[X \mapsto n]$, see under 'Some notation for assignments to type variables' in Section 2.4.)

(b) For any X-DI process P which satisfies (**Norm**) *(and thus also* (**RecCho**$_X$)*: see Section 3.5.2), we have that*

$$W(\mathcal{NS}(\mathcal{S}_{\{\}\vdash P})) \leq W(\mathcal{S}_{\{\}\vdash P})$$

(where $W(\mathcal{S}_{\{\}\vdash P})$ was defined in Section 3.3.3).

Proof. (a) is proved by (I) and (II) in the proof of Theorem 3.7.4 (a).

(b) can be proved by (a) and Proposition 3.5.4 (d).

Alternatively, (b) can be proved by showing that for any node $\overline{N}$ of $\mathcal{NS}(\mathcal{S}_{\{\}\vdash P})$ there is some $(N, f) \in \overline{N}$ such that any $(N', f') \in \overline{N}$ is invisibly reachable from (N, f) (in the terminology of Section 3.6.2), so that $|Context(\overline{N})| = |Range(f)| \leq |Context(N)|$. This can in turn be shown either by (I) and (II) in the proof of Theorem 3.7.4 (a) and by Proposition 3.5.4 (d), or from the procedure in Section 3.6.2 and Proposition 3.5.4 (c). ■

3.8 Tools

In this section, we point informally towards various possibilities for automation arising from the research presented in this chapter. They can be seen either in their own right, or as means to automating the results we shall present in Chapter 5. (See Section 5.6.)

3.8.1 CSP$_M$

As we have remarked (see Section 2.1.4, Section 2.2, and before Example 2.5.1 in Section 2.5.2), the language in this thesis is based on CSP$_M$ [Sca92, Sca98, Ros98a], the machine-readable version of CSP used in tools, but it is smaller and more theoretically presented.

Thus, when it comes to automation, there are two basic approaches:

- The first is to isolate a sublanguage of CSP$_M$ that corresponds to our language, so that what we have presented applies to that sublanguage. Moreover, we can attempt to transform arbitrary processes in CSP$_M$ to those in the sublanguage, which may itself be automated.
- The second, which is more involved but more suitable in the long term, is to adapt the research in this thesis to CSP$_M$. Although that has mostly been done (see [Laz99], [Ros98a, Section 15.2], [LR98]), due to the complexity of CSP$_M$ it would have been too lengthy to present.

With either approach, a type checker for CSP$_M$ is needed (in connection with this, see Section 2.5.3). In particular, a type checker would enable conditions of data independence, weak data independence, and parameterisation by data types and process-free type contexts (see Section 2.7) to be checked automatically.[5]

At any rate, in the following sections we shall examine the possibilities for automation in terms of the language in this thesis. The important subjects of how those conclusions can

[5]At the time of writing (April 1999), type checkers for CSP$_M$ are under development at Formal Systems (Europe) Ltd, who are the developers of the model checker FDR [Ros94a, Ros98a, FS97], and by Ping Gao at Adelaide University (Australia), where the model checker ARC [Yan96] was developed.

be applied to CSP$_M$ and of automation issues that are specific to CSP$_M$ are left for future research.[6]

3.8.2 SLTSs and ASLTSs

The procedure for constructing SLTSs (see Section 3.2.1) can be completely automated, and Proposition 3.3.3 provides sufficient conditions for termination.

The procedure for constructing ASLTSs (see Section 3.3.2) can also be completely automated, including the checking of the prerequisite condition (a) in Section 3.3.2. Sufficient conditions for its termination are also provided by Proposition 3.3.3.

However, when it comes to automation, there are a number of issues that we did not examine when presenting the two procedures. The following are some of those:

Reusing nodes. In the procedures as presented, each visible or invisible transition of a node is separately determined from the visible or invisible transitions of its subprocesses, so that the reusing of nodes (see under 'Reusing nodes' in Section 3.2.1) is done only 'at the highest level'.

Alternatively, when the process labelling a node has an appropriate principal construct, the subsequent portion of the SLTS or ASLTS can be constructed by combining the SLTSs or ASLTSs corresponding to the subprocesses. The reusing of nodes is then done at the level of the subprocesses.

Another thing to point out is that, in either case, it may be inefficient to store all the processes which label the nodes generated so far. Deciding which ones to store then becomes an important heuristic.

For ordinary LTSs, topics of this kind are discussed in greater detail in [Sca98] and [Ros98a, Section C.1.1].

Permutations of variable names. The reusing of nodes can be improved further by considering a node with a label $\Gamma \vdash P$ to be reusable instead of a node with a label $\Gamma' \vdash P'$ if P can be obtained from P' by a bijection between Γ' and Γ which preserves types. For example, in the SLTS in Figure 3.1, the nodes N_1, N_2 and N_3 would be reusable instead of N_4, N_5 and N_6 (respectively), and in the ASLTSs in Figures 3.2 and 3.3, each of the nodes N_2, N_3 and N_4 would be reusable instead of any of the other two.

One way of formalising and implementing this elimination of symmetry in variable names is to allow permutations of variable names at the ends of transitions. This can reduce the number of nodes of an SLTS or ASLTS exponentially in the number of variable names, although it cannot reduce an infinite SLTS or ASLTS to a finite one.

Nodes with conditions. Another characteristic of the procedures as presented is that the conditional transitions of a node in an SLTS, or the *guard* conditions in the visible and invisible transitions of a node in an ASLTS, are constructed separately and have no effect on the subsequent portion of the SLTS or ASLTS.

Alternatively, node labels can be of the form $(\Gamma \vdash P, \mathit{cond})$, where *cond* is a condition on the variables in Γ which is satisfied whenever the node is reached during executions. The

[6] A substantial research project whose aim is automation based on the work presented in this thesis will begin in 1999.

condition *cond* can then be used to simplify the process $\Gamma \vdash P$ and the transitions of the node.

Moreover, the conditions in the node labels can either be purely boolean (i.e. *True*, or *False*, or built from equality tests and the boolean connectives $\neg$, $\wedge$ and $\vee$), or also contain $\exists$ and possibly $\forall$ quantifiers. Using quantifiers is potentially more accurate, by taking into account variables which were present previously but are no longer in the type context. In either case, a node with a label $(\Gamma \vdash P, \mathit{cond})$ can be reused instead of a node with a label $(\Gamma \vdash P, \mathit{cond}')$ provided the implication $\mathit{cond}' \Rightarrow \mathit{cond}$ holds.

Those were three progressively more accurate kinds of 'lazy branching'. In contrast to them, for SLTSs of data independent processes at least, one can use 'eager branching' as in the procedure for symbolic normalisation (see Section 3.6.2). Namely, any visible or invisible transition is immediately followed by conditional transitions which completely determine the equality or inequality relationships of the newly introduced variables with the old, and those are the only conditional transitions.

However, one should keep in mind that altering the two procedures may make some of the results in this thesis (such as Propositions 3.3.1–3.3.3) inapplicable or false.

As we have pointed out in Section 3.4.3, the subject of equivalence-preserving transformations of SLTSs and ASLTSs and of their automation is a developing research topic. As with ordinary LTSs, integration of such transformations with the procedures for constructing SLTSs and ASLTSs is very important.

Once the construction of SLTSs and ASLTSs is automatic, the quantities W, $L_{\$}$, $L_{?}$ and $L_{\sqcap}$ (see Section 3.3.3) can be automatically computed. Alternatively, if one does not wish to depend on the termination of the procedures for construction, it may be possible to automatically compute useful estimates from the process syntax (see Section 5.6.2).

3.8.3 Conditions

Each of the conditions ($\mathbf{NoEqT}_X$), ($\mathbf{PosConjEqT}_X$), ($\mathbf{RecCho}_X$) and (**Norm**) (see Section 3.5) can be checked completely automatically, with guaranteed termination. For their weaker versions which use denotational semantics or symbolic operational semantics, complete automation is also likely to be possible, but without guaranteed termination.

As we have remarked in Section 3.5.3, it may also be possible and useful to automate semantics-preserving transformations which aim to bring a process into a form which satisfies one of the conditions. However, we leave this for future research.

3.8.4 NSLTSs

The procedure for symbolic normalisation (see Section 3.6.2) can be entirely automated, and it can be applied either to finite SLTSs or lazily to infinite SLTSs. Sufficient conditions for its termination are provided by Proposition 3.7.2.

As with SLTSs and ASLTSs, a number of issues that we did not examine when presenting the procedure become significant. All the remarks under 'Reusing nodes', 'Permutations of variable names' and 'Nodes with conditions' in Section 3.8.2 apply in appropriate ways. Again, one should keep in mind that altering the procedure may make some of the results in this thesis (such as Propositions 3.7.1 and 3.7.2) inapplicable or false.

Equivalence-preserving transformations of NSLTSs (see Section 3.7.3) and their automation belong to the same developing research topic as equivalence-preserving transformations of SLTSs and ASLTSs. With a notion of symbolic labelled transition systems which is general enough to include both SLTSs and NSLTSs in this thesis, symbolic normalisation can itself be seen as an equivalence-preserving transformation (see Theorem 3.7.4 (a)).

Once symbolic normalisation is automatic, the quantities W and L (see Section 3.7.2) can be automatically computed. Alternatively, if one does not wish to depend on the termination of symbolic normalisation, it may be possible to automatically compute useful estimates from the process and/or from the SLTS that is being symbolically normalised (see Section 5.6.2). Two such possibilites are provided by Proposition 3.7.5.

3.9 Notes

The developments in Section 3.1 relied heavily on known results about reduction in typed λ-calculi: see e.g. [Mit90, GLT89, Loa97].

In Sections 3.2 and 3.3, we borrowed many ideas from the large literature on symbolic execution, most notably [JP93, HL95a]. Some of the other references are [BJ78, Sch94, Lin96, Pac96, Rat97, ST97, Kok97, Kok98, SB95, HB95, HI+96, Hoj96, CZ+97]. The subject of symbolic execution is neither restricted to concurrent systems nor to data independence. It is also related to many different techniques for nonexplicit verification, in particular in the area of verifying timed and hybrid systems (see e.g. [Hen96]).

The proof of the Congruence Theorem in Section 3.4.2 followed the same pattern as for ordinary LTSs in [Ros98a, Section 9.4], but the details were considerably more complex. For further references on that subject, see [Ros98a, Section 9.5].

For a few references on the developing research topic of equivalence-preserving transformations of symbolic transition systems, see Section 3.4.3.

Our main contributions in Sections 3.1–3.3 are:

- to have addressed a number of distinguishing features of our language (see Section 2.3), such as the wealth of operators (see Section 3.2.3), the nondirectedness of channels and many-way synchronisation (see under 'Visible' in Section 3.2.2, under 'Parallel composition' in Section 3.2.3, Section 3.5.1, and under 'Normalised symbolic transitions' in Section 3.6.1) and the presence of higher types (see Section 2.5.1),

- that the SLTSs and ASLTSs obtained are particularly useful for reasoning about data independence, since their transitions can only introduce variables whose types are type variables (see Section 3.2.2), and

- the study of when the various aspects of SLTSs and ASLTSs are finite (see Section 3.3.3).

Determinisation (or, in our terminology, normalisation) of automata (see e.g. [Per90, Tho90]) and of ordinary LTSs (see e.g. [Ros94a] or [Ros98a, Section C.1.2]) is a developed subject. However, we are not aware of any previous work for symbolic transition systems, although there are some relations with [HB95, HI+96, Hoj96, CZ+97] which combine symbolic execution and BDDs (see e.g. [Bry86, BC+92, McM93]).

The process *NondReg* in Example 2.5.1 was constructed by A.W. Roscoe, to show that the result of symbolically normalising a finite SLTS can be an infinite NSLTS (see Example 3.7.1).

Chapter 4

Logical relations

In a typed language, a ‘logical relation’ is an assignment of relations to types which is such that, for each type construct (such as forming function types), the relation assigned to any constructed type is obtained in a specific way from the relations assigned to its component types. Thus any logical relation can be obtained from the relations it assigns to type variables and ground types, by induction on the structure of types.

More precisely, any logical relation can be obtained from some ρ, δ and δ' such that, for type variables and ground types T_0, $\rho[\![T_0]\!]$ is a relation between $\delta[\![T_0]\!]$ and $\delta'[\![T_0]\!]$. For any type T, the relation $[\![T]\!]_{\rho,\delta,\delta'}$ which the logical relation assigns to T is then a relation between the denotational interpretations $[\![T]\!]_\delta$ and $[\![T]\!]_{\delta'}$.

The so-called Basic Lemma of Logical Relations (see e.g. [Mit90, OHe96, Plo80, Rey83, Wad89]) states that, for any logical relation such that the relations it assigns to ground types are appropriate (in a precise sense), we have that, for any term, related assignments to its free term variables yield related denotational interpretations. More precisely, it states that, for any logical relation as above such that the relations $\rho[\![T_0]\!]$ for ground types T_0 are appropriate, we have that, for any term $\Gamma \vdash r : T$, if η and η' are assignments to Γ with respect to δ and δ' (respectively) such that

$$\forall (x : U) \in \Gamma . (\eta[\![x]\!])([\![U]\!]_{\rho,\delta,\delta'})(\eta'[\![x]\!])$$

then

$$([\![\Gamma \vdash r : T]\!]_{\delta,\eta})([\![T]\!]_{\rho,\delta,\delta'})([\![\Gamma \vdash r : T]\!]_{\delta',\eta'})$$

The Basic Lemma has many applications in both theory and practice of typed languages. On the theoretical side, it can for example be used for cutting out junk elements from denotational interpretations of types with universally quantified type variables (see e.g. the references given above).

On the practical side, the Basic Lemma can for example be used for reasoning about abstraction and data refinement (see e.g. [KO+97]). For instance, if δ and η denote abstract data types with abstract operations, if δ' and η' denote concrete data types with concrete operations, and if $\rho[\![T_0]\!] : \delta[\![T_0]\!] \leftrightarrow \delta'[\![T_0]\!]$ is the identity relation for each ground type T_0, then for any term of a ground type, the Basic Lemma states that its abstract and concrete denotational interpretations are the same.

This chapter is devoted to defining logical relations for the language in this thesis (see Chapter 2) and to presenting the Basic Lemma for them. That will confirm formally the intuition that the language is ‘parametrically polymorphic’, or in other words that any term ‘behaves uniformly’ for different instances of (i.e. assignments to) its free type variables.

In the next chapter, in cases when the Basic Lemma can be applied to data independent or weakly data independent processes, it will provide more direct ways of relating their various denotational interpretations than by translating to and from their symbolic operational interpretations (see the remarks at the beginning of Chapter 3).

4.1 Definition

We have said above that any logical relation can be obtained from the relations it assigns to type variables and ground types. Since our language has no ground types (such as an atomic type of natural numbers), defining logical relations for it will amount to defining how an assignment of relations to type variables is 'lifted' to all types.

Some terminology for relations. If $\mathcal{R}$ is a relation between a set $\mathcal{V}$ and a set $\mathcal{V}'$, we shall say that (with respect to $\mathcal{V}$ and $\mathcal{V}'$):

- $\mathcal{R}$ is total if and only if $\forall\, v \in \mathcal{V}.\,\exists\, v'.v\mathcal{R}v'$;
- $\mathcal{R}$ is surjective if and only if $\forall\, v' \in \mathcal{V}'.\,\exists\, v.v\mathcal{R}v'$;
- $\mathcal{R}$ is a partial function if and only if $\forall\, v, v_1', v_2'.(v\mathcal{R}v_1' \wedge v\mathcal{R}v_2') \Rightarrow v_1' = v_2'$;
- $\mathcal{R}$ is injective if and only if $\forall\, v_1, v_2, v'.(v_1\mathcal{R}v' \wedge v_2\mathcal{R}v') \Rightarrow v_1 = v_2$;
- $\mathcal{R}$ is the identity relation if and only if $\mathcal{V} = \mathcal{V}'$ and $\forall\, v_1, v_2 \in \mathcal{V}.v_1\mathcal{R}v_2 \Leftrightarrow v_1 = v_2$;
- $\mathcal{R}$ is reflexive if and only if $\mathcal{V} = \mathcal{V}'$ and $\forall\, v \in \mathcal{V}.v\mathcal{R}v$;
- $\mathcal{R}$ is symmetric if and only if $\mathcal{V} = \mathcal{V}'$ and $\forall\, v_1, v_2.v_1\mathcal{R}v_2 \Rightarrow v_2\mathcal{R}v_1$;
- $\mathcal{R}$ is transitive if and only if $\mathcal{V} = \mathcal{V}'$ and $\forall\, v_1, v_2, v_3.(v_1\mathcal{R}v_2 \wedge v_2\mathcal{R}v_3) \Rightarrow v_1\mathcal{R}v_3$;
- $\mathcal{R}$ is a partial equivalence relation if and only if it is symmetric and transitive;
- $\mathcal{R}$ is an equivalence relation if and only if it is reflexive, symmetric and transitive.

■

Some notation for relations. If δ and δ' assign partial orders to type variables, subject to the restrictions stated under 'Type variables' in Section 2.4, we shall write

$$\rho : \delta \leftrightarrow \delta'$$

to mean that, for each type variable X, $\rho[\![X]\!]$ is a relation between $\delta[\![X]\!]$ and $\delta'[\![X]\!]$.

Similarly, we shall use pointwise liftings of operations on relations. For example, for each type variable X, we shall have:

$$\begin{aligned} \rho^{-1}[\![X]\!] &= (\rho[\![X]\!])^{-1} \\ (\rho \cup \sigma)[\![X]\!] &= \rho[\![X]\!] \cup \sigma[\![X]\!] \end{aligned}$$

Also, for each type variable X, we shall have:

$$(\delta \times \delta')[\![X]\!] = \delta[\![X]\!] \times \delta'[\![X]\!] \quad ■$$

Notation for logical relations. Supposing that

- M specifies one of the models Tr, Fa, FD of CSP,
- $\rho : \delta \leftrightarrow \delta'$, and
- T is a type,

we shall write

$$[\![T]\!]^M_{\rho,\delta,\delta'}$$

for the relation assigned to T by the logical relation which is obtained from ρ, δ and δ' with respect to M.

Any $[\![T]\!]^M_{\rho,\delta,\delta'}$ will be defined as a relation between $[\![T]\!]^M_\delta$ and $[\![T]\!]^M_{\delta'}$. ■

Some notation for sequences. The following is some notation for sequences, which will in particular be useful for traces under 'Process types' below.

For any sequence t, $|t|$ will denote its length. For any $k \in \{1, \ldots, |t|\}$, $t(k)$ will denote the kth element of t, so that $t = \langle t(1), \ldots, t(|t|)\rangle$. For any $D \subseteq \{1, \ldots, |t|\}$, $t \restriction D$ will denote the sequence consisting of the elements of t whose indices are in D, in the same order as in t.

For any relation $\mathcal{R}$, $\mathcal{R}^*$ will denote its pointwise lifting to sequences, so that

$$t\mathcal{R}^*t' \Leftrightarrow |t| = |t'| \wedge \forall k \in \{1, \ldots, |t|\}.(t(k))\mathcal{R}(t'(k))$$

For any set $\mathcal{Q}$ of sequences, $\min \mathcal{Q}$ will denote its minimal elements with respect to the prefix ordering, so that

$$\min \mathcal{Q} = \{t \in \mathcal{Q} \mid \forall t' \in \mathcal{Q}.t' \leq t \Rightarrow t' = t\}$$

■

The relations $[\![T]\!]^M_{\rho,\delta,\delta'} : [\![T]\!]^M_\delta \leftrightarrow [\![T]\!]^M_{\delta'}$ are defined as follows, by induction on the structure of types:

Type variables. The relation assigned to a type variable is the relation that ρ assigns to it:

$$[\![X]\!]^M_{\rho,\delta,\delta'} = \rho[\![X]\!]$$

Sum types. Two elements of a sum type are related if and only if they belong to the same component and are related by the relation assigned to that component:

$$(id_i, v)([\![id_1.T_1 + \cdots + id_n.T_n]\!]^M_{\rho,\delta,\delta'})(id_{i'}, v') \Leftrightarrow i = i' \wedge v([\![T_i]\!]^M_{\rho,\delta,\delta'})v'$$

Product types. Two tuples are related if and only if they are related componentwise:

$$(v_1, \ldots, v_n)([\![T_1 \times \cdots \times T_n]\!]^M_{\rho,\delta,\delta'})(v'_1, \ldots, v'_n) \Leftrightarrow \forall i \in \{1, \ldots, n\}.v_i([\![T_i]\!]^M_{\rho,\delta,\delta'})v'_i$$

Inductive types. For an inductive type $\mu X.T$, the relation $[\![\mu X.T]\!]^M_{\rho,\delta,\delta'}$ is defined as the least fixed point of the operator on relations which corresponds to T.

More precisely, the operator is on triples which consist of two partial orders and a relation between them:

$$\begin{array}{l}\mathcal{H}((\mathcal{V},\leq),\mathcal{R},(\mathcal{V}',\leq')) = \\ \quad ([\![T]\!]^M_{\delta[(\mathcal{V},\leq)/X]}, [\![T]\!]^M_{\rho[\mathcal{R}/X],\delta[(\mathcal{V},\leq)/X],\delta'[(\mathcal{V}',\leq')/X]}, [\![T]\!]^M_{\delta'[(\mathcal{V}',\leq')/X]})\end{array}$$

The ordering on the triples is:

$$\begin{array}{l}((\mathcal{V},\leq),\mathcal{R},(\mathcal{V}',\leq')) \preceq ((\mathcal{V}'',\leq''),\mathcal{R}',(\mathcal{V}''',\leq''')) \Leftrightarrow \\ \quad (\mathcal{V},\leq) \trianglelefteq (\mathcal{V}'',\leq'') \ \wedge\ (\mathcal{V}',\leq') \trianglelefteq (\mathcal{V}''',\leq''') \\ \quad \wedge\, \mathcal{R} = \mathcal{R}' \restriction ((\mathcal{V}\times\mathcal{V}''') \cup (\mathcal{V}''\times\mathcal{V}'))\end{array}$$

where $\trianglelefteq$ is the ordering on partial orders defined under 'Inductive types' in Section 2.4.

Similarly to the situation under 'Inductive types' in Section 2.4, it is straightforward to prove that $\preceq$ is a complete ordering, and using the assumption that T satisfies $T^{(+,\times,\mu)_X}$, it is straightforward to prove that $\mathcal{H}$ is continuous with respect to $\preceq$. Hence the least fixed point of $\mathcal{H}$ can be obtained as the least upper bound of the chain of all triples which are produced by iterating $\mathcal{H}$ finitely many times from the least triple:

$$\begin{array}{rcl}((\mathcal{V}_0,\leq_0),\mathcal{R}_0,(\mathcal{V}'_0,\leq'_0)) & = & ((\{\},\{\}),\{\},(\{\},\{\})) \\ ((\mathcal{V}_{k+1},\leq_{k+1}),\mathcal{R}_{k+1},(\mathcal{V}'_{k+1},\leq'_{k+1})) & = & \mathcal{H}((\mathcal{V}_k,\leq_k),\mathcal{R}_k,(\mathcal{V}'_k,\leq'_k))\end{array}$$

The relation $[\![\mu X.T]\!]^M_{\rho,\delta,\delta'}$ is then defined to be the relation in the least fixed point, which is the union of the relations in the iterations:

$$[\![\mu X.T]\!]^M_{\rho,\delta,\delta'} = \bigcup_{k\in\mathbb{N}} \mathcal{R}_k$$

As required, $[\![\mu X.T]\!]^M_{\rho,\delta,\delta'}$ is a relation between $[\![\mu X.T]\!]^M_{\delta}$ and $[\![\mu X.T]\!]^M_{\delta'}$, since they are the two partial orders in the least fixed point (see under 'Inductive types' in Section 2.4).

Function types. Two functions are related if and only if they map related elements to related elements:

$$f([\![T_1 \to T_2]\!]^M_{\rho,\delta,\delta'})f' \Leftrightarrow \forall v,v'.v([\![T_1]\!]^M_{\rho,\delta,\delta'})v' \Rightarrow (f(v))([\![T_2]\!]^M_{\rho,\delta,\delta'})(f'(v'))$$

Process types. Here we need to define the relation $[\![\mathit{Proc}_{T,\phi}]\!]^M_{\rho,\delta,\delta'}$ for a process type $\mathit{Proc}_{T,\phi}$.

Since T satisfies T^{alph}, it is of the form

$$\sum_{i=1}^{n} id_i.\prod_{j=1}^{m_i} T_{i,j}$$

It also does not involve any process types, so we can omit M in $[\![T]\!]_{\rho,\delta,\delta'}$ and $[\![T_{i,j}]\!]_{\rho,\delta,\delta'}$ (see under 'Some conventions' below).

For $M = \mathrm{Tr}$, the definition is given in Table 4.1. It says that two sets of traces Trs and Trs' are related if and only if

(Tr.i) for any strict prefix of any trace from *Trs*, if all inputs and nondeterministic selections in it are in the domain of the logical relation, then all outputs in the prefix and the next event are also in the domain, and

(Tr.ii) for any trace t from *Trs* which is in the domain of the logical relation, there exists a nonempty subset I of Trs' such that t is related to any trace in I and such that I is closed under those alterations of inputs and nondeterministic selections which preserve the logical relation.

(Tr.i) reflects the fact that, for $M = \mathrm{Tr}$, in the Basic Lemma we shall be assuming that, for any type variable X, either $\rho[\![X]\!]$ is total or the condition ($\mathbf{RecCho}_X$) is satisfied (see Theorem 4.2.2).

For $M = \mathrm{Fa}$, the definition is given in Table 4.2. It says that two ordered pairs of sets of traces and failures $(\mathit{Trs}, \mathit{Fails})$ and $(\mathit{Trs}', \mathit{Fails}')$ are related if and only if

(Fa.i) *Trs* and Trs' are related as in (Tr.ii) above, except that t is not constrained to be in the domain of the logical relation,

(Fa.ii) for any failure (t, A) from *Fails*, there exists a superset B of A such that (t, B) is also a failure from *Fails* and such that the complement of B is closed under altering inputs, and

(Fa.iii) for any failure (t, A) from *Fails*, there exists a nonempty subset J of Fails' such that (t, A) is related to any failure in J and such that J is closed under those alterations of inputs and nondeterministic selections which preserve the logical relation.

To state (Fa.iii) more precisely, a failure (t, A) is related to a failure (t', A') if and only if t is related to t' and, for any $\sigma : \delta \leftrightarrow \delta'$ such that the inverse of $\sigma[\![X]\!]$ is a function on the complement of the range of $\rho[\![X]\!]$ for each type variable X, any element of the complement of A' is related to some element of the complement of A by the logical relation obtained from $\rho \cup \sigma$.

An analogue of (Tr.i) is not needed, and traces are not constrained to be in the domain of the logical relation, because for $M = \mathrm{Fa}$, in the Basic Lemma we shall be assuming that $\rho[\![X]\!]$ is total for all type variables X (see Theorem 4.2.2).

For $M = \mathrm{FD}$, the definition is given in Table 4.3. It says that two ordered pairs of sets of failures and divergences $(\mathit{Fails}_\perp, \mathit{Divs})$ and $(\mathit{Fails}'_\perp, \mathit{Divs}')$ are related if and only if

(FD.i) $\mathit{Fails}_\perp$ has the same property as *Fails* in (Fa.ii),

(FD.ii) $\mathit{Fails}_\perp$ and $\mathit{Fails}'_\perp$ are related as *Fails* and Fails' in (Fa.iii), and

(FD.iii) *Divs* and Divs' are related as *Trs* and Trs' in (Fa.i).

(As for $M = \mathrm{Fa}$, in the Basic Lemma, for $M = \mathrm{FD}$, we shall be assuming that $\rho[\![X]\!]$ is total for all type variables X (see Theorem 4.2.2).)

In contrast to the definitions for the other type constructs, the definition of $[\![\mathit{Proc}_{T,\phi}]\!]^M_{\rho,\delta,\delta'}$ is not symmetric, so that it is not true in general that $[\![\mathit{Proc}_{T,\phi}]\!]^M_{\rho^{-1},\delta',\delta}$ is the inverse of $[\![\mathit{Proc}_{T,\phi}]\!]^M_{\rho,\delta,\delta'}$. However, the lack of symmetry will not cause difficulties in this thesis, and the definition given and the Basic Lemma that we shall present in the next section (see Theorem 4.2.2) will be suitable for the applications in Chapter 5 (see the remarks before the beginning of this section).

$$
\begin{array}{l}
\mathit{Trs}\,(\llbracket \mathit{Proc}_{T,\phi}\rrbracket^{\mathrm{Tr}}_{\rho,\delta,\delta'})\,\mathit{Trs}' \;\Leftrightarrow \\
\quad (\forall\, t \in \mathit{Trs}.\mathit{Recorded}(T,\phi,\rho,\delta,\delta',t)) \;\wedge \\
\quad (\forall\, t \in \mathit{Trs}.t \in \mathit{Domain}((\llbracket T\rrbracket_{\rho,\delta,\delta'})^{*}) \Rightarrow \\
\quad \exists\, I \subseteq \mathit{Trs}'.I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T,\rho,\delta,\delta',t,I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T,\phi,\rho,\delta,\delta',t,I))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Recorded}(T,\phi,\rho,\delta,\delta',t) = \\
\quad \forall\, k \in \{1,\ldots,|t|\}. \\
\quad (\forall\, k^{\dagger}, i, v, j. \\
\quad (k^{\dagger} < k \wedge t(k^{\dagger}) = (id_i, v) \wedge \phi(id_i, j) \not\subseteq \{!\}) \Rightarrow \\
\quad v(j) \in \mathit{Domain}(\llbracket T_{i,j}\rrbracket_{\rho,\delta,\delta'})) \\
\quad \Rightarrow \\
\quad (\forall\, k^{\dagger}, i, v, j. \\
\quad (k^{\dagger} \leq k \wedge t(k^{\dagger}) = (id_i, v) \wedge \phi(id_i, j) \subseteq \{!\}) \Rightarrow \\
\quad v(j) \in \mathit{Domain}(\llbracket T_{i,j}\rrbracket_{\rho,\delta,\delta'}))
\end{array}
$$

$$
\mathit{Related}^{\mathrm{Tr}}(T,\rho,\delta,\delta',t,I) \quad = \quad \forall\, t' \in I.t(\llbracket T\rrbracket_{\rho,\delta,\delta'})^{*}t'
$$

$$
\begin{array}{l}
\mathit{Closed}^{\mathrm{Tr}}(T,\phi,\rho,\delta,\delta',t,I) = \\
\quad \forall\, t' \in I.\forall\, k, i, v, v', j, w'. \\
\quad (t(k) = (id_i, v) \wedge t'(k) = (id_i, v') \wedge \phi(id_i, j) \subseteq \{?, \$\} \wedge (v(j))(\llbracket T_{i,j}\rrbracket_{\rho,\delta,\delta'})w') \Rightarrow \\
\quad (\exists\, t'' \in I.t'' \upharpoonright \{1,\ldots,k-1\} = t' \upharpoonright \{1,\ldots,k-1\} \wedge t''(k) = (id_i, v'[w'/j]))
\end{array}
$$

Table 4.1: Definition of $\llbracket \mathit{Proc}_{T,\phi}\rrbracket^{\mathrm{Tr}}_{\rho,\delta,\delta'}$

$$
\begin{array}{l}
(\mathit{Trs}, \mathit{Fails})(\llbracket \mathit{Proc}_{T,\phi} \rrbracket^{\mathrm{Fa}}_{\rho,\delta,\delta'})(\mathit{Trs}', \mathit{Fails}') \;\Leftrightarrow \\
\quad (\forall\, t \in \mathit{Trs}.\, \exists\, I \subseteq \mathit{Trs}'. \\
\quad I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T, \rho, \delta, \delta', t, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T, \phi, \rho, \delta, \delta', t, I)) \;\wedge \\
\quad (\forall (t, A) \in \mathit{Fails}.\, \exists\, B \supseteq A.(t, B) \in \mathit{Fails} \wedge \mathit{Full}(T, \phi, \delta, B)) \;\wedge \\
\quad (\forall (t, A) \in \mathit{Fails}.\, \exists\, J \subseteq \mathit{Fails}'. \\
\quad J \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T, \rho, \delta, \delta', (t, A), J) \wedge \mathit{Closed}^{\mathrm{Fa}}(T, \phi, \rho, \delta, \delta', (t, A), J))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Full}(T, \phi, \delta, B) = \\
\quad \forall\, i, v, j, w. \\
\quad ((id_i, v) \in (\llbracket T \rrbracket_\delta \setminus B) \wedge \phi(id_i, j) \subseteq \{?\} \wedge (v(j))(\llbracket T_{i,j} \rrbracket_{\delta \times \delta,\delta,\delta})w) \Rightarrow \\
\quad (id_i, v[w/j]) \in (\llbracket T \rrbracket_\delta \setminus B)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Related}^{\mathrm{Fa}}(T, \rho, \delta, \delta', (t, A), J) = \\
\quad \forall (t', A') \in J.(t(\llbracket T \rrbracket_{\rho,\delta,\delta'})^* t' \;\wedge \\
\quad \forall\, \sigma : \delta \leftrightarrow \delta'.(\forall\, X \in \mathit{AllTypeVars}.(\sigma\llbracket X \rrbracket)^{-1} : (\delta'\llbracket X \rrbracket \setminus \mathit{Range}(\rho\llbracket X \rrbracket)) \to \delta\llbracket X \rrbracket) \Rightarrow \\
\quad (\forall\, a' \in (\llbracket T \rrbracket_{\delta'} \setminus A').\, \exists\, a \in (\llbracket T \rrbracket_\delta \setminus A).a(\llbracket T \rrbracket_{(\rho \cup \sigma),\delta,\delta'})a'))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Closed}^{\mathrm{Fa}}(T, \phi, \rho, \delta, \delta', (t, A), J) = \\
\quad \forall (t', A') \in J.\, \forall\, k, i, v, v', j, w'. \\
\quad (t(k) = (id_i, v) \wedge t'(k) = (id_i, v') \wedge \phi(id_i, j) \subseteq \{?, \$\} \wedge (v(j))(\llbracket T_{i,j} \rrbracket_{\rho,\delta,\delta'})w') \Rightarrow \\
\quad (\exists (t'', A'') \in J.t'' \restriction \{1, \ldots, k-1\} = t' \restriction \{1, \ldots, k-1\} \wedge t''(k) = (id_i, v'[w'/j]))
\end{array}
$$

Table 4.2: Definition of $\llbracket \mathit{Proc}_{T,\phi} \rrbracket^{\mathrm{Fa}}_{\rho,\delta,\delta'}$

$$
\begin{array}{l}
(\mathit{Fails}_\perp, \mathit{Divs})(\llbracket \mathit{Proc}_{T,\phi} \rrbracket^{\mathrm{FD}}_{\rho,\delta,\delta'})(\mathit{Fails}'_\perp, \mathit{Divs}') \;\Leftrightarrow \\
\quad (\forall (t, A) \in \mathit{Fails}_\perp.\, \exists\, B \supseteq A.(t, B) \in \mathit{Fails}_\perp \wedge \mathit{Full}(T, \phi, \delta, B)) \;\wedge \\
\quad (\forall (t, A) \in \mathit{Fails}_\perp.\, \exists\, J \subseteq \mathit{Fails}'_\perp. \\
\quad J \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T, \rho, \delta, \delta', (t, A), J) \wedge \mathit{Closed}^{\mathrm{Fa}}(T, \phi, \rho, \delta, \delta', (t, A), J)) \;\wedge \\
\quad (\forall\, t \in \mathit{Divs}.\, \exists\, I \subseteq \mathit{Divs}'. \\
\quad I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T, \rho, \delta, \delta', t, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T, \phi, \rho, \delta, \delta', t, I))
\end{array}
$$

Table 4.3: Definition of $\llbracket \mathit{Proc}_{T,\phi} \rrbracket^{\mathrm{FD}}_{\rho,\delta,\delta'}$

The parts (Tr.i), (Fa.ii) and (FD.i) are restrictions on only *Trs*, *Fails* and $Fails_\perp$ (respectively). They could have been incorporated earlier in the thesis, as means of cutting out 'junk elements' depending on which type variables X the condition ($\mathbf{RecCho}_X$) (see Section 3.5.2) is assumed for and depending on which ways specifiers can be used. We have included them as parts of the definition of $[\![Proc_{T,\phi}]\!]^M_{\rho,\delta,\delta'}$ for brevity.

Observe also that $[\![Proc_{T,\phi}]\!]^M_{\rho,\delta,\delta'}$ depend on the ways specifier ϕ, which is not the case with $[\![Proc_{T,\phi}]\!]^M_{\delta}$ and $[\![Proc_{T,\phi}]\!]^M_{\delta'}$ (see under 'Process types' and Proposition 2.4.1 in Section 2.4).

Some conventions. For types T which do not involve process types, the relations $[\![T]\!]^M_{\rho,\delta,\delta'}$ do not depend on M, and so we may write just $[\![T]\!]_{\rho,\delta,\delta'}$.

When it does not cause ambiguity, we may write only $[\![T]\!]^M_{\rho}$. ■

The following two propositions are about the relations $[\![T]\!]_{\rho,\delta,\delta'}$ for types T which do not involve process types.

The first one says that the definitions given above for the type constructs other than process types preserve the property of being an identity relation, whereas the second one says that they preserve inverses (which confirms formally that they are symmetric).

Both can be proved by induction on the structure of T.

Proposition 4.1.1 *For any type T which does not involve process types, and for any δ, δ' and $\rho : \delta \leftrightarrow \delta'$ such that $\rho[\![X]\!]$ is the identity relation for each $X \in Free(T)$, we have that $[\![T]\!]_{\rho,\delta,\delta'}$ also is the identity relation.* ■

Proposition 4.1.2 *For any type T which does not involve process types, and for any δ, δ', $\rho : \delta \leftrightarrow \delta'$ and $\sigma : \delta' \leftrightarrow \delta$ such that*

$$\forall X \in Free(T).\sigma[\![X]\!] = (\rho[\![X]\!])^{-1}$$

we have that

$$[\![T]\!]_{\sigma,\delta',\delta} = ([\![T]\!]_{\rho,\delta,\delta'})^{-1}$$

■

The following three proposition are about the relations $[\![T]\!]_{\rho,\delta,\delta'}$ for types T which satisfy $T^{AllTypeVars,+,\times,\mu}$ (i.e. which involve neither function types nor process types).

The first one says that the definitions given above for the type constructs other than function types and process types preserve any of the properties defined under 'Some terminology for relations' above, the second one says that they are monotonic with respect to $\subseteq$, and the third one says that they preserve relational composition.

Again, all three can be proved by induction on the structure of T.

Proposition 4.1.3 *Suppose π is any of the properties defined under 'Some terminology for relations' above.*

Then, for any type T which satisfies $T^{AllTypeVars,+,\times,\mu}$, and for any δ, δ' and $\rho : \delta \leftrightarrow \delta'$ such that $\rho[\![X]\!]$ has the property π for each $X \in Free(T)$, we have that $[\![T]\!]_{\rho,\delta,\delta'}$ also has the property π. ■

Proposition 4.1.4 *For any type T which satisfies $T^{AllTypeVars,+,\times,\mu}$, and for any δ, δ', $\rho_1 : \delta \leftrightarrow \delta'$ and $\rho_2 : \delta \leftrightarrow \delta'$ such that*

$$\forall X \in \mathit{Free}(T).\rho_1[\![X]\!] \subseteq \rho_2[\![X]\!]$$

we have that

$$[\![T]\!]_{\rho_1,\delta,\delta'} \subseteq [\![T]\!]_{\rho_2,\delta,\delta'}$$

■

Proposition 4.1.5 *For any type T which satisfies $T^{AllTypeVars,+,\times,\mu}$, and for any δ, δ', δ'', $\rho_1 : \delta \leftrightarrow \delta'$, $\rho_2 : \delta' \leftrightarrow \delta''$ and $\sigma : \delta \leftrightarrow \delta''$ such that*

$$\forall X \in \mathit{Free}(T).\sigma[\![X]\!] = \rho_1[\![X]\!] \circ \rho_2[\![X]\!]$$

we have that

$$[\![T]\!]_{\sigma,\delta,\delta''} = [\![T]\!]_{\rho_1,\delta,\delta'} \circ [\![T]\!]_{\rho_2,\delta',\delta''}$$

■

The following corollary is straightforward to deduce from Propositions 4.1.3, 4.1.4 and 4.1.5. It says that, for any relation $[\![T]\!]_{\rho,\delta,\delta'}$ where T satisfies $T^{AllTypeVars,+,\times,\mu}$, if v and v' as well as w and w' are related by it, then v and w differ at most in elements of variable types if and only if v' and w' do.

Corollary 4.1.6 *For any type T which satisfies $T^{AllTypeVars,+,\times,\mu}$, and for any δ, δ' and $\rho : \delta \leftrightarrow \delta'$, we have that*

$$\forall v, w, v', w'.(v([\![T]\!]_{\rho,\delta,\delta'})v' \wedge w([\![T]\!]_{\rho,\delta,\delta'})w') \;\Rightarrow\; (v([\![T]\!]_{\delta\times\delta,\delta,\delta})w \Leftrightarrow v'([\![T]\!]_{\delta'\times\delta',\delta',\delta'})w')$$

■

The following proposition will be used in the part of the proof of the Basic Lemma (see Theorem 4.2.2) which deals with the CSP construct of recursion.

Definition 4.1.1 Suppose $\mathcal{R}$ is a relation between two complete partial orders $(\mathcal{V}, \leq)$ and $(\mathcal{V}', \leq')$, and let $\preceq$ be the pointwise ordering on $\mathcal{R}$:

$$(v, v') \preceq (w, w') \;\Leftrightarrow\; v \leq w \wedge v' \leq' w'$$

We shall say that $\mathcal{R}$ is *complete* if and only if $(\mathcal{R}, \preceq)$ is a complete partial order such that:

- the least element of $(\mathcal{R}, \preceq)$ is $(\bot, \bot')$, where $\bot$ and $\bot'$ are the least elements of $(\mathcal{V}, \leq)$ and $(\mathcal{V}', \leq')$ (respectively), and
- for any directed set $\mathcal{D} \subseteq \mathcal{R}$, its least upper bound in $(\mathcal{R}, \preceq)$ consists of pointwise least upper bounds in $(\mathcal{V}, \leq)$ and $(\mathcal{V}', \leq')$:

$$\bigsqcup \mathcal{D} = (\bigsqcup\{v \mid (v, v') \in \mathcal{D}\}, \bigsqcup\{v' \mid (v, v') \in \mathcal{D}\})$$

■

Proposition 4.1.7 *For any process type $Proc_{T,\phi}$, and for any M, δ, δ' and $\rho : \delta \leftrightarrow \delta'$ such that either $M = \mathrm{Tr}$ or $\rho[\![X]\!]$ is total for each $X \in Free(T)$, we have that $[\![Proc_{T,\phi}]\!]^M_{\rho,\delta,\delta'}$ is complete (with respect to the orderings on $[\![Proc_{T,\phi}]\!]^M_{\delta}$ and $[\![Proc_{T,\phi}]\!]^M_{\delta'}$ defined under 'Process types' in Section 2.4).*

Proof. For $M = \mathrm{Tr}$ or $M = \mathrm{Fa}$, the proof is straightforward. The least elements of $[\![Proc_{T,\phi}]\!]^{\mathrm{Tr}}_{\delta}$ and $[\![Proc_{T,\phi}]\!]^{\mathrm{Tr}}_{\delta'}$ are both $\{\langle\rangle\}$, and the least upper bounds are unions. The least elements of $[\![Proc_{T,\phi}]\!]^{\mathrm{Fa}}_{\delta}$ and $[\![Proc_{T,\phi}]\!]^{\mathrm{Fa}}_{\delta'}$ are both $(\{\langle\rangle\}, \{\})$, and the least upper bounds are pointwise unions.

For $M = \mathrm{FD}$, the proof is more involved. The least elements of $[\![Proc_{T,\phi}]\!]^{\mathrm{FD}}_{\delta}$ and $[\![Proc_{T,\phi}]\!]^{\mathrm{FD}}_{\delta'}$ are $(([\![T]\!]_{\delta})^* \times \mathcal{P}([\![T]\!]_{\delta}), ([\![T]\!]_{\delta})^*)$ and $(([\![T]\!]_{\delta'})^* \times \mathcal{P}([\![T]\!]_{\delta'}), ([\![T]\!]_{\delta'})^*)$, and the least upper bounds are pointwise intersections.

Let us present only the most complex part, namely that the pointwise least upper bounds of any directed subset of $[\![Proc_{T,\phi}]\!]^{\mathrm{FD}}_{\rho,\delta,\delta'}$ satisfy (FD.ii).

So suppose $\mathcal{D} \subseteq [\![Proc_{T,\phi}]\!]^{\mathrm{FD}}_{\rho,\delta,\delta'}$ is directed (see Definition 4.1.1), let

$$\begin{aligned} F^{\dagger} &= \bigcap\{F \mid ((F,D),(F',D')) \in \mathcal{D}\} \\ F^{\dagger'} &= \bigcap\{F' \mid ((F,D),(F',D')) \in \mathcal{D}\} \end{aligned}$$

and suppose $(t, A) \in F^{\dagger}$.

Then, for each $((F,D),(F',D')) \in \mathcal{D}$, we have that $(t,A) \in F$, and so there exists some $J_{((F,D),(F',D'))}$ such that

$$\begin{aligned} & J_{((F,D),(F',D'))} \subseteq F' \wedge J_{((F,D),(F',D'))} \neq \{\} \\ & \wedge\ Related^{\mathrm{Fa}}(T, \rho, \delta, \delta', (t,A), J_{((F,D),(F',D'))}) \\ & \wedge\ Closed^{\mathrm{Fa}}(T, \phi, \rho, \delta, \delta', (t,A), J_{((F,D),(F',D'))}) \end{aligned}$$

Now, for each $((F,D),(F',D')) \in \mathcal{D}$, let

$$\begin{aligned} K_{((F,D),(F',D'))} &= \{(t',A') \mid \exists((F'',D''),(F''',D''')) \in \mathcal{D}. \\ &\qquad ((F,D),(F',D')) \preceq ((F'',D''),(F''',D''')) \\ &\qquad \wedge (t',A') \in J_{((F'',D''),(F''',D'''))}\} \\ L_{((F,D),(F',D'))} &= \{(t',A') \mid (\exists A''.(t',A'') \in K_{((F,D),(F',D'))}) \\ &\qquad \wedge (\exists((F'',D''),(F''',D''')) \in \mathcal{D}.(t',A') \in K_{((F'',D''),(F''',D'''))}) \\ &\qquad \wedge (t',A') \in F'\} \end{aligned}$$

(For the definition of $\preceq$, see Definition 4.1.1.)

Using the fact that

$$((F,D),(F',D')) \preceq ((F'',D''),(F''',D''')) \Rightarrow F' \supseteq F'''$$

(see under 'Process types' in Section 2.4), it is straightforward to show that

$$\begin{aligned} & \forall((F,D),(F',D')) \in \mathcal{D}. \\ & L_{((F,D),(F',D'))} \subseteq F' \wedge L_{((F,D),(F',D'))} \neq \{\} \\ & \wedge\ Related^{\mathrm{Fa}}(T, \rho, \delta, \delta', (t,A), L_{((F,D),(F',D'))}) \\ & \wedge\ Closed^{\mathrm{Fa}}(T, \phi, \rho, \delta, \delta', (t,A), L_{((F,D),(F',D'))}) \end{aligned} \tag{4.1}$$

and that

$$\begin{array}{l}\forall((F,D),(F',D')),((F'',D''),(F''',D''')) \in \mathcal{D}.\\ ((F,D),(F',D')) \preceq ((F'',D''),(F''',D''')) \Rightarrow\\ (L_{((F,D),(F',D'))} \supseteq L_{((F'',D''),(F''',D'''))} \wedge\\ \forall t'.(\exists A''.(t',A'') \in L_{((F'',D''),(F''',D'''))}) \Rightarrow\\ \{(t'',A'') \in L_{((F'',D''),(F''',D'''))} \mid t''=t'\} =\\ \{(t'',A'') \in L_{((F,D),(F',D'))} \mid t''=t'\} \cap F''')\end{array} \tag{4.2}$$

Now let

$$L^\dagger = \bigcap\{L_{((F,D),(F',D'))} \mid ((F,D),(F',D')) \in \mathcal{D}\}$$

That is a subset of $F^{\dagger\prime}$, and so it suffices to show that

$$L^\dagger \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T,\rho,\delta,\delta',(t,A),L^\dagger) \wedge \mathit{Closed}^{\mathrm{Fa}}(T,\phi,\rho,\delta,\delta',(t,A),L^\dagger)$$

To show that $L^\dagger \neq \{\}$, consider

$$\mathcal{M} = \{\{t' \mid \exists A'.(t',A') \in L_{((F,D),(F',D'))}\} \mid ((F,D),(F',D')) \in \mathcal{D}\}$$

Since $t([\![T]\!]_{\rho,\delta,\delta'})^* t'$ whenever $t' \in M \in \mathcal{M}$, and since $\delta'[\![X]\!]$ must be finite for each $X \in \mathit{Free}(T)$ (see under 'Restrictions with FD' (ii) in Section 2.5.2), we have that $\mathcal{M}$ is a directed set of nonempty finite sets with respect to $\supseteq$, and so $\bigcap\mathcal{M} \neq \{\}$.

Let $t' \in \bigcap\mathcal{M}$, and for each $((F,D),(F',D')) \in \mathcal{D}$, let

$$N_{((F,D),(F',D'))} = \{(t'',A'') \in L_{((F,D),(F',D'))} \mid t''=t'\}$$

If $t' \in D'$ for all $((F,D),(F',D')) \in \mathcal{D}$, then, using (4.2), it follows that $N_{((F,D),(F',D'))}$ is the same for all $((F,D),(F',D')) \in \mathcal{D}$. Thus it is a subset of $L^\dagger$, and so $L^\dagger$ is nonempty.

Otherwise, let $((F,D),(F',D')) \in \mathcal{D}$ be such that $t' \notin D'$. Then, using (4.2) again, it follows that $N_{((F,D),(F',D'))} \subseteq N_{((F'',D''),(F''',D'''))}$ for all $((F'',D''),(F''',D''')) \in \mathcal{D}$. Thus $N_{((F,D),(F',D'))}$ is a subset of $L^\dagger$, and so $L^\dagger$ is again nonempty.

That $\mathit{Related}^{\mathrm{Fa}}(T,\rho,\delta,\delta',(t,A),L^\dagger)$ holds is immediate from (4.2) and the definition of $L^\dagger$.

To show that $\mathit{Closed}^{\mathrm{Fa}}(T,\phi,\rho,\delta,\delta',(t,A),L^\dagger)$ holds, suppose $(t',A') \in L^\dagger$ and

$$t(k) = (id_i,v) \wedge t'(k) = (id_i,v') \wedge \phi(id_i,j) \subseteq \{?,\$\} \wedge (v(j))([\![T_{i,j}]\!]_{\rho,\delta,\delta'})w'$$

where T is of the form $\sum_{i=1}^n id_i. \prod_{j=1}^{m_i} T_{i,j}$.

Then, for each $((F,D),(F',D')) \in \mathcal{D}$, the set

$$O_{((F,D),(F',D'))} = \{(t'',A'') \in L_{((F,D),(F',D'))} \mid \begin{array}{l} t'' \restriction \{1,\ldots,k-1\} = t' \restriction \{1,\ldots,k-1\}\\ \wedge\, t''(k) = (id_i, v'[w'/j])\}\end{array}$$

is nonempty by (4.1).

Since the condition

$$t'' \restriction \{1,\ldots,k-1\} = t' \restriction \{1,\ldots,k-1\} \wedge t''(k) = (id_i, v'[w'/j])$$

depends neither on $((F,D),(F',D'))$ nor on A'', it is straightforward to show that the sets $O_{((F,D),(F',D'))}$ also satisfy (4.2). Consequently, that their intersection is nonempty can be shown in the same way as that $L^\dagger$ is nonempty. Thus there exists $(t'',A'') \in L^\dagger$ such that

$$t'' \restriction \{1,\ldots,k-1\} = t' \restriction \{1,\ldots,k-1\} \wedge t''(k) = (id_i, v'[w'/j])$$

which establishes that $\mathit{Closed}^{\mathrm{Fa}}(T,\phi,\rho,\delta,\delta',(t,A),L^\dagger)$ holds. ■

4.2 The Basic Lemma

In this section, we shall present the Basic Lemma of Logical Relations for the typed λ-calculus fragment of our language, then for the whole language, and then a corollary of the latter. Finally, we shall give two examples. (For a few remarks about the Basic Lemma and its applications in general, and about its applications in Chapter 5, see the beginning of the present chapter, and Section 4.3.)

Theorem 4.2.1 *Suppose that:*

- $\Gamma \vdash r : T$ *is a term in the fragment of our language obtained by omitting process types, equality testing and CSP constructs;*
- $\rho : \delta \leftrightarrow \delta'$*;*
- η *and* η' *are assignments to* Γ *with respect to* δ *and* δ' *(respectively);*
- $\forall (x : U) \in \Gamma . (\eta[\![x]\!])([\![U]\!]_{\rho,\delta,\delta'})(\eta'[\![x]\!])$.

Then

$$([\![\Gamma \vdash r]\!]_{\delta,\eta})([\![T]\!]_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]_{\delta',\eta'})$$

Proof. This result is well-known (see e.g. [Mit90, OHe96, Plo80, Rey83, Wad89]), and can be proved by induction on the structure of r. ■

Theorem 4.2.2 *Suppose that:*

- $\Gamma \vdash r : T$ *is a term;*
- M *specifies one of the models Tr, Fa, FD of CSP;*
- $\rho : \delta \leftrightarrow \delta'$*;*
- *if* $M = \mathrm{Tr}$*, then for each* $X \in \mathit{AllTypeVars}$*:*
  - *either* $\Gamma \vdash r$ *satisfies* ($\mathbf{RecCho}_X$) *or* $\rho[\![X]\!]$ *is total;*
  - *either* $\Gamma \vdash r$ *satisfies* ($\mathbf{NoEqT}_X$) *or* $\rho[\![X]\!]$ *is a partial function;*
  - *either* $\Gamma \vdash r$ *satisfies* ($\mathbf{PosConjEqT}_X$) *or* $\rho[\![X]\!]$ *is injective;*
- *if* $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$*, then for each* $X \in \mathit{AllTypeVars}$*:*
  - $\rho[\![X]\!]$ *is total;*
  - *either* $\Gamma \vdash r$ *satisfies* ($\mathbf{NoEqT}_X$) *or* $\rho[\![X]\!]$ *is a partial function and injective;*
- η *and* η' *are assignments to* Γ *with respect to* M *and* δ*, and* M *and* δ' *(respectively);*
- $\forall (x : U) \in \Gamma . (\eta[\![x]\!])([\![U]\!]^M_{\rho,\delta,\delta'})(\eta'[\![x]\!])$.

Then

$$([\![\Gamma \vdash r]\!]^M_{\delta,\eta})([\![T]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]^M_{\delta',\eta'})$$

Proof. The proof is by induction on the structure of r.

The cases for the typed λ-calculus constructs other than equality testing proceed as in the proof of Theorem 4.2.1. The following are the more interesting of the remaining cases.

Equality testing. Here

$$r = ((s = u) \rightsquigarrow v, w)$$

for some $\Gamma \vdash s, u : U$ and $\Gamma \vdash v, w : T$ such that U^{comm}.

Thus, for each $X \in \mathit{Free}(U)$, $\Gamma \vdash r$ does not satisfy ($\mathbf{NoEqT}_X$), and so $\rho[\![X]\!]$ is a partial function.

If $M = \mathrm{Fa}$ or $M = \mathrm{FD}$, we have by the same token that $\rho[\![X]\!]$ is injective for each $X \in \mathit{Free}(U)$.

If $M = \mathrm{Tr}$, we have that either $\Gamma \vdash r$ satisfies ($\mathbf{PosConjEqT}_X$) for some $X \in \mathit{Free}(U)$, or $\rho[\![X]\!]$ is injective for each $X \in \mathit{Free}(U)$.

By the inductive hypothesis, we have that:

$$([\![\Gamma \vdash s]\!]^M_{\delta,\eta})([\![U]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash s]\!]^M_{\delta',\eta'}) \tag{4.3}$$

$$([\![\Gamma \vdash u]\!]^M_{\delta,\eta})([\![U]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash u]\!]^M_{\delta',\eta'}) \tag{4.4}$$

$$([\![\Gamma \vdash v]\!]^M_{\delta,\eta})([\![T]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash v]\!]^M_{\delta',\eta'}) \tag{4.5}$$

$$([\![\Gamma \vdash w]\!]^M_{\delta,\eta})([\![T]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash w]\!]^M_{\delta',\eta'}) \tag{4.6}$$

If $\rho[\![X]\!]$ is a partial function and injective for each $X \in \mathit{Free}(U)$, then by (4.3) and (4.4) it follows that either

$$[\![\Gamma \vdash r]\!]^M_{\delta,\eta} = [\![\Gamma \vdash v]\!]^M_{\delta,\eta} \ \mathit{and} \ [\![\Gamma \vdash r]\!]^M_{\delta',\eta'} = [\![\Gamma \vdash v]\!]^M_{\delta',\eta'} \tag{4.7}$$

or

$$[\![\Gamma \vdash r]\!]^M_{\delta,\eta} = [\![\Gamma \vdash w]\!]^M_{\delta,\eta} \ \mathit{and} \ [\![\Gamma \vdash r]\!]^M_{\delta',\eta'} = [\![\Gamma \vdash w]\!]^M_{\delta',\eta'} \tag{4.8}$$

and so either (4.5) or (4.6) gives us that $([\![\Gamma \vdash r]\!]^M_{\delta,\eta})([\![T]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]^M_{\delta',\eta'})$ as required.

Otherwise, we have that $\rho[\![X]\!]$ is a partial function for each $X \in \mathit{Free}(U)$, that $M = \mathrm{Tr}$, and that $\Gamma \vdash r$ satisfies ($\mathbf{PosConjEqT}_X$) for some $X \in \mathit{Free}(U)$. In particular, T is a process type $\mathit{Proc}_{T^\dagger,\phi}$ and $w = \mathit{STOP}_{T^\dagger}$.

If either (4.7) or (4.8) holds, we have that $([\![\Gamma \vdash r]\!]^{\mathrm{Tr}}_{\delta,\eta})([\![\mathit{Proc}_{T^\dagger,\phi}]\!]^{\mathrm{Tr}}_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]^{\mathrm{Tr}}_{\delta',\eta'})$ as above.

By (4.3) and (4.4), the remaining case is that

$$[\![\Gamma \vdash r]\!]^{\mathrm{Tr}}_{\delta,\eta} = [\![\Gamma \vdash w]\!]^{\mathrm{Tr}}_{\delta,\eta} = \{\langle\rangle\} \ \mathit{and} \ [\![\Gamma \vdash r]\!]^{\mathrm{Tr}}_{\delta',\eta'} = [\![\Gamma \vdash v]\!]^{\mathrm{Tr}}_{\delta',\eta'}$$

and there we have that $([\![\Gamma \vdash r]\!]^{\mathrm{Tr}}_{\delta,\eta})([\![\mathit{Proc}_{T^\dagger,\phi}]\!]^{\mathrm{Tr}}_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]^{\mathrm{Tr}}_{\delta',\eta'})$ is almost immediate (see the definition of $[\![\mathit{Proc}_{T^\dagger,\phi}]\!]^{\mathrm{Tr}}_{\rho,\delta,\delta'}$ under 'Process types' in Section 4.1).

Outputs, inputs and nondeterministic selections. Here

$$\begin{array}{rcl} r & = & id_i\ it_1\ \ldots\ it_{m_i} \longrightarrow P \\ T & = & Proc_{T^\dagger,\chi} \end{array}$$

for some $\tilde{\Gamma} \vdash it_1 : Item_{T^\dagger_{i,1}}, \ldots, \tilde{\Gamma} \vdash it_{m_i} : Item_{T^\dagger_{i,m_i}}$ and $\tilde{\Gamma} \cup Vars(it_1\ \ldots\ it_{m_i}) \vdash P : Proc_{T^\dagger,\phi}$ such that $T^\dagger = \sum_{i^\dagger=1}^{n} id_{i^\dagger}.\prod_{j^\dagger=1}^{m_{i^\dagger}} T^\dagger_{i^\dagger,j^\dagger}$, $Dist(it_1\ \ldots\ it_{m_i})$, $Typ(it_1\ \ldots\ it_{m_i})$ and $\phi \cup Ways(T^\dagger, id_i, it_1\ \ldots\ it_{m_i}) \subseteq \chi$.

Let us first consider $M = \mathrm{Tr}$.

To show that

$$\forall\, t \in traces(\Gamma \vdash r)_{\delta,\eta}.Recorded(T^\dagger, \chi, \rho, \delta, \delta', t)$$

suppose $t \in traces(\Gamma \vdash r)_{\delta,\eta}$.

If $t = \langle\rangle$, then $Recorded(T^\dagger, \chi, \rho, \delta, \delta', t)$ is immediate.

Otherwise, t is of the form $\langle (id_i, v)\rangle\hat{\ }t^\dagger$, where $\pi_!(v) = w_!$ and

$$t^\dagger \in traces(\tilde{\Gamma} \cup Vars(it_1\ \ldots\ it_{m_i}) \vdash P)_{\delta,\eta^\dagger_v}$$

(for the notation, see under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2, except that we have written $t^\dagger$ and $\eta^\dagger_v$ instead of t and η'_v). Suppose $k \in \{1, \ldots, |t|\}$ and

$$\begin{array}{l} \forall\, k^\dagger, i^\dagger, v^\dagger, j^\dagger. \\ (k^\dagger < k \wedge t(k^\dagger) = (id_{i^\dagger}, v^\dagger) \wedge \chi(id_{i^\dagger}, j^\dagger) \not\subseteq \{!\}) \Rightarrow \\ v^\dagger(j^\dagger) \in Domain([\![T^\dagger_{i^\dagger,j^\dagger}]\!]_{\rho,\delta,\delta'}) \end{array} \qquad (4.9)$$

If $k = 1$, then

$$\begin{array}{l} \forall\, k^\dagger, i^\dagger, v^\dagger, j^\dagger. \\ (k^\dagger \leq k \wedge t(k^\dagger) = (id_{i^\dagger}, v^\dagger) \wedge \chi(id_{i^\dagger}, j^\dagger) \subseteq \{!\}) \Rightarrow \\ v^\dagger(j^\dagger) \in Domain([\![T^\dagger_{i^\dagger,j^\dagger}]\!]_{\rho,\delta,\delta'}) \end{array} \qquad (4.10)$$

follows by observing that, whenever $j \in J_!$ and $!r^\dagger = it_j$, we have that $[\![\tilde{\Gamma} \vdash r^\dagger]\!]^{\mathrm{Tr}}_{\delta,\eta} \in Domain([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'})$. Otherwise, by (4.9) with $k^\dagger = 1$, we can apply the inductive hypothesis to $\tilde{\Gamma} \cup Vars(it_1\ \ldots\ it_{m_i}) \vdash P : Proc_{T^\dagger,\phi}$, $\eta^\dagger_v$ and some $\eta'^\dagger$ such that $\eta'^\dagger \restriction \tilde{\Gamma} = \eta' \restriction \tilde{\Gamma}$. By (4.9) with $k^\dagger \geq 2$, the inductive hypothesis gives us that (4.10) holds for ϕ, $t^\dagger$ and $k - 1$ instead of χ, t and k. That fact and what we have observed for $k = 1$ imply that (4.10) holds.

To show that

$$\begin{array}{l} \forall\, t \in traces(\Gamma \vdash r)_{\delta,\eta}.t \in Domain(([\![T^\dagger]\!]_{\rho,\delta,\delta'})^*) \Rightarrow \\ \exists\, I \subseteq traces(\Gamma \vdash r)_{\delta',\eta'}. \\ I \neq \{\} \wedge Related^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t, I) \wedge Closed^{\mathrm{Tr}}(T^\dagger, \chi, \rho, \delta, \delta', t, I) \end{array}$$

suppose $t \in \mathit{traces}(\Gamma \vdash r)_{\delta,\eta}$ and $t \in \mathit{Domain}((\llbracket T^\dagger \rrbracket_{\rho,\delta,\delta'})^*)$.

If $t = \langle\rangle$, choosing $I = \{\langle\rangle\}$ suffices.

Otherwise, as above, t is of the form $\langle(id_i, v)\rangle\hat{}\,t^\dagger$, where $\pi_!(v) = w_!$ and

$$t^\dagger \in \mathit{traces}(\tilde{\Gamma} \cup \mathit{Vars}(it_1\ \ldots\ it_{m_i}) \vdash P)_{\delta,\eta_v^\dagger}$$

For any v' such that $v(\llbracket T^\dagger_{i,1} \times \cdots \times T^\dagger_{i,m_i} \rrbracket_{\rho,\delta,\delta'})v'$ and $\pi_!(v') = w'_!$ (where $w'_!$ is the tuple of interpretations of outputs in the prefix with respect to δ' and η'), we can apply the inductive hypothesis to

$$\tilde{\Gamma} \cup \mathit{Vars}(it_1\ \ldots\ it_{m_i}) \vdash P : \mathit{Proc}_{T^\dagger,\phi}$$

with $\eta^\dagger$ and ${\eta'}^\dagger_{v'}$ (where ${\eta'}^\dagger_{v'}$ is η' updated by assigning the values given by v' to the variables of inputs and nondeterministic selections in the prefix), and obtain that there exists

$$I^\dagger(v') \subseteq \mathit{traces}(\tilde{\Gamma} \cup \mathit{Vars}(it_1\ \ldots\ it_{m_i}) \vdash P)_{\delta',{\eta'}^\dagger_{v'}}$$

such that

$$I^\dagger(v') \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t^\dagger, I^\dagger(v')) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \phi, \rho, \delta, \delta', t^\dagger, I^\dagger(v'))$$

Let

$$\begin{aligned} I = \{\langle(id_i, v')\rangle\hat{}\,{t'}^\dagger \mid\ & v(\llbracket T^\dagger_{i,1} \times \cdots \times T^\dagger_{i,m_i} \rrbracket_{\rho,\delta,\delta'})v' \\ & \wedge \pi_!(v') = w'_! \\ & \wedge {t'}^\dagger \in I^\dagger(v')\} \end{aligned}$$

It is then straightforward to check that $I \subseteq \mathit{traces}(\Gamma \vdash r)_{\delta',\eta'}$ and

$$I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \chi, \rho, \delta, \delta', t, I)$$

as required. That completes the case $M = \mathrm{Tr}$.

Let us now consider $M = \mathrm{FD}$.

To show that

$$\forall (t, A) \in \mathit{failures}_\perp(\Gamma \vdash r)_{\delta,\eta}.\ \exists\, B \supseteq A.(t, B) \in \mathit{failures}_\perp(\Gamma \vdash r)_{\delta,\eta} \wedge \mathit{Full}(T^\dagger, \chi, \delta, B)$$

suppose $(t, A) \in \mathit{failures}_\perp(\Gamma \vdash r)_{\delta,\eta}$.

If $t = \langle\rangle$, then there exists $w_\$ \in \prod_{j \in J_\$} \llbracket T^\dagger_{i,j} \rrbracket_\delta$ such that

$$\forall\, w_? \in \prod_{j \in J_?} \llbracket T^\dagger_{i,j} \rrbracket_\delta.(id_i, \iota(w_!, w_?, w_\$)) \notin A$$

(for the notation, see under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2). Let

$$\begin{aligned} B = \{(id_{i^\dagger}, v) \mid\ & i^\dagger \in \{1, \ldots, n\} \wedge v \in \textstyle\prod_{j=1}^{m_{i^\dagger}} [\![T^\dagger_{i^\dagger,j}]\!]_\delta \wedge \\ & (i^\dagger \neq i \vee \forall\, w_? \in \textstyle\prod_{j \in J_?} [\![T^\dagger_{i,j}]\!]_\delta . v \neq \iota(w_!, w_?, w_\$))\} \end{aligned}$$

It is then straightforward to check $B \supseteq A$, $(t, B) \in \mathit{failures}_\perp(\Gamma \vdash r)_{\delta,\eta}$ and $\mathit{Full}(T^\dagger, \chi, \delta, B)$, as required.

If $t \neq \langle\rangle$, then we have that t is of the form $\langle(id_i, v)\rangle^\frown t^\dagger$, where $\pi_!(v) = w_!$ and

$$(t^\dagger, A) \in \mathit{failures}_\perp(\tilde{\Gamma} \cup \mathit{Vars}(it_1\ \ldots\ it_{m_i}) \vdash P)_{\delta,\eta^\dagger_v}$$

(for the notation, see under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2, except that we have written $t^\dagger$ and $\eta^\dagger_v$ instead of t and η'_v). It then suffices to observe that, by the inductive hypothesis,

$$\exists\, B \supseteq A.(t^\dagger, B) \in \mathit{failures}_\perp(\tilde{\Gamma} \cup \mathit{Vars}(it_1\ \ldots\ it_{m_i}) \vdash P)_{\delta,\eta^\dagger_v} \wedge \mathit{Full}(T^\dagger, \phi, \delta, B)$$

The remaining parts for $M = \mathrm{Fa}$ and $M = \mathrm{FD}$ are similar to the parts we have presented.

Replicated nondeterministic choice. Here

$$\begin{aligned} r &= \textstyle\bigsqcap_{x:U} P \\ T &= \mathit{Proc}_{T^\dagger,\chi} \end{aligned}$$

for some $(\Gamma \setminus \{x\}) \cup \{x : U\} \vdash P : \mathit{Proc}_{T^\dagger,\phi}$ such that U^{comm} and $\phi \subseteq \chi$.

For any $X \in \mathit{Free}(U)$, $\Gamma \vdash r$ does not satisfy $(\mathbf{RecCho}_X)$ (see the definition of $(\mathbf{RecCho}_X)$ in Section 3.5.2), and so $\rho[\![X]\!]$ is total (by the assumptions in the theorem we are proving).

Therefore, for any $u \in [\![U]\!]_\delta$, there exists $u' \in [\![U]\!]_{\delta'}$ such that $u([\![U]\!]_{\rho,\delta,\delta'})u'$. Using this fact and the inductive hypothesis,

$$([\![\Gamma \vdash r]\!]^M_{\delta,\eta})([\![T]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]^M_{\delta',\eta'})$$

follows straightforwardly.

Parallel composition. Here

$$\begin{aligned} r &= P \underset{S}{\parallel} Q \\ T &= \mathit{Proc}_{T^\dagger,\chi} \end{aligned}$$

for some $\Gamma \vdash P : \mathit{Proc}_{T^\dagger,\phi}$, $\Gamma \vdash Q : \mathit{Proc}_{T^\dagger,\psi}$ and $\Gamma \vdash S : \mathit{Set}_{T^\dagger}$ such that $T^\dagger = \sum_{i^\dagger=1}^{n} id_{i^\dagger} . \prod_{j^\dagger=1}^{m_{i^\dagger}} T^\dagger_{i^\dagger,j^\dagger}$ and $\phi \underset{S}{\parallel} \psi \subseteq \chi$.

By Proposition 2.5.3 (b), we can assume that ϕ and ψ are least with respect to $\subseteq$.

The following are some observations about $[\![\Gamma \vdash S]\!]_{\delta,\eta}$ and $[\![\Gamma \vdash S]\!]_{\delta',\eta'}$ that we shall be using below, either with or without explicitly referencing them.

By the syntax and denotational semantics of sets and elems (see under 'Sets' and 'Elems' in Section 2.5.2), it follows that

$$\forall\, a, b. a([\![T^\dagger]\!]_{\delta\times\delta,\delta,\delta})b \;\Rightarrow\; (a \in [\![\Gamma \vdash S]\!]_{\delta,\eta} \Leftrightarrow b \in [\![\Gamma \vdash S]\!]_{\delta,\eta}) \tag{4.11}$$

and

$$\forall\, a', b'. a'([\![T^\dagger]\!]_{\delta'\times\delta',\delta',\delta'})b' \;\Rightarrow\; (a' \in [\![\Gamma \vdash S]\!]_{\delta',\eta'} \Leftrightarrow b' \in [\![\Gamma \vdash S]\!]_{\delta',\eta'}) \tag{4.12}$$

which say that $[\![\Gamma \vdash S]\!]_{\delta,\eta}$ and $[\![\Gamma \vdash S]\!]_{\delta',\eta'}$ are not selective about elements of variable types.

Furthermore, for any r_{Elem_U} which is a subterm of an elem e in S but not a subterm of a different subterm $r^\dagger_{\mathit{Elem}_{U^\dagger}}$ of e, the inductive hypothesis gives us that

$$([\![\Gamma \vdash r]\!]_{\delta,\eta})([\![U]\!]_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]_{\delta',\eta'})$$

But $U^{\mathit{AllTypeVars},+,\times,\mu}$ and $\mathit{Free}(U) = \{\}$, which by Proposition 4.1.1 means that $[\![U]\!]_{\rho,\delta,\delta'}$ is the identity relation, and so $[\![\Gamma \vdash r]\!]_{\delta,\eta} = [\![\Gamma \vdash r]\!]_{\delta',\eta'}$. It then follows that

$$\forall\, a, a'. a([\![T^\dagger]\!]_{\delta\times\delta',\delta,\delta'})a' \;\Rightarrow\; (a \in [\![\Gamma \vdash S]\!]_{\delta,\eta} \Leftrightarrow a' \in [\![\Gamma \vdash S]\!]_{\delta',\eta'}) \tag{4.13}$$

which says that $[\![\Gamma \vdash S]\!]_{\delta,\eta}$ and $[\![\Gamma \vdash S]\!]_{\delta',\eta'}$ are the same up to elements of variable types. By Proposition 4.1.4, $a([\![T^\dagger]\!]_{\delta\times\delta',\delta,\delta'})a'$ is implied by $a([\![T^\dagger]\!]_{\rho,\delta,\delta'})a'$, so we also have that

$$\forall\, a, a'. a([\![T^\dagger]\!]_{\rho,\delta,\delta'})a' \;\Rightarrow\; (a \in [\![\Gamma \vdash S]\!]_{\delta,\eta} \Leftrightarrow a' \in [\![\Gamma \vdash S]\!]_{\delta',\eta'}) \tag{4.14}$$

Let us now begin with showing that

$$([\![\Gamma \vdash r]\!]_{\delta,\eta})([\![T]\!]_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]_{\delta',\eta'})$$

and let us first consider $M = \mathrm{Tr}$.

To show that

$$\forall\, u \in \mathit{traces}(\Gamma \vdash r)_{\delta,\eta}.\mathit{Recorded}(T^\dagger, \chi, \rho, \delta, \delta', u)$$

suppose $u \in \mathit{traces}(\Gamma \vdash r)_{\delta,\eta}$, $k \in \{1, \ldots, |u|\}$ and

$$\begin{array}{l} \forall\, k^\dagger, i, v, j. \\ (k^\dagger < k \wedge u(k^\dagger) = (id_i, v) \wedge \chi(id_i, j) \not\subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'}) \end{array}$$

Let $t_1 \in \mathit{traces}(\Gamma \vdash P)_{\delta,\eta}$ and $t_2 \in \mathit{traces}(\Gamma \vdash Q)_{\delta,\eta}$ be such that $u \in t_1 \underset{[\![\Gamma\vdash S]\!]_{\delta,\eta}}{\|} t_2$. Then there exist integers

$$0 = l_1^0 \leq l_1^1 \leq \cdots \leq l_1^{|u|} = |t_1|$$

and

$$0 = l_2^0 \leq l_2^1 \leq \cdots \leq l_2^{|u|} = |t_2|$$

such that

$$\begin{array}{l} \forall\, k^\ddagger \in \{1, \ldots, |u|\}. \\ (u \restriction \{k^\ddagger\}) \in (t_1 \restriction \{l_1^{k^\ddagger-1}+1, \ldots, l_1^{k^\ddagger}\}) \underset{[\![\Gamma \vdash S]\!]_{\delta,\eta}}{\|} (t_2 \restriction \{l_2^{k^\ddagger-1}+1, \ldots, l_2^{k^\ddagger}\}) \end{array}$$

Using the inductive hypothesis for $\Gamma \vdash P : \mathit{Proc}_{T^\dagger,\phi}$ and $\Gamma \vdash Q : \mathit{Proc}_{T^\dagger,\psi}$, and the fact that $\phi \underset{S}{\|} \psi \subseteq \chi$, it is straightforward to prove by induction that, for all $k^\ddagger \in \{1, \ldots, k-1\}$, we have that

$$\begin{array}{l} \forall\, l^\dagger, i, v, j. \\ (l^\dagger \leq l_1^{k^\ddagger} \wedge t_1(l^\dagger) = (id_i, v) \wedge \phi(id_i, j) \not\subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'}) \end{array}$$

and

$$\begin{array}{l} \forall\, l^\dagger, i, v, j. \\ (l^\dagger \leq l_2^{k^\ddagger} \wedge t_2(l^\dagger) = (id_i, v) \wedge \psi(id_i, j) \not\subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'}) \end{array}$$

In particular, the inductive hypothesis then gives us that

$$\begin{array}{l} \forall\, l^\dagger, i, v, j. \\ (l^\dagger \leq l_1^{k} \wedge t_1(l^\dagger) = (id_i, v) \wedge \phi(id_i, j) \subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'})) \end{array}$$

and

$$\begin{array}{l} \forall\, l^\dagger, i, v, j. \\ (l^\dagger \leq l_2^{k} \wedge t_2(l^\dagger) = (id_i, v) \wedge \psi(id_i, j) \subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'})) \end{array}$$

which imply that

$$\begin{array}{l} \forall\, k^\dagger, i, v, j. \\ (k^\dagger \leq k \wedge u(k^\dagger) = (id_i, v) \wedge \chi(id_i, j) \subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'})) \end{array}$$

as required.

The rest of the case $M = \mathrm{Tr}$ and the whole case $M = \mathrm{Fa}$ are not more complex than what is involved for $M = \mathrm{FD}$, so let us consider $M = \mathrm{FD}$.

To show that

$$\begin{array}{l} \forall (u, A) \in \mathit{failures}_\bot(\Gamma \vdash r)_{\delta,\eta}.\, \exists\, B \supseteq A. \\ (u, B) \in \mathit{failures}_\bot(\Gamma \vdash r)_{\delta,\eta} \wedge \mathit{Full}(T^\dagger, \chi, \delta, B) \end{array}$$

suppose $(u, A) \in \mathit{failures}_{\perp}(\Gamma \vdash r)_{\delta,\eta}$.

If $u \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}$, taking $B = [\![T^{\dagger}]\!]_{\delta}$ suffices.

Otherwise, there exist $(t_1, A_1) \in \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta,\eta}$ and $(t_2, A_2) \in \mathit{failures}_{\perp}(\Gamma \vdash Q)_{\delta,\eta}$ such that $u \in t_1 \underset{[\![\Gamma \vdash S]\!]_{\delta,\eta}}{\|} t_2$, $A_1 \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta} = A_2 \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta}$ and $A = A_1 \cup A_2$. By the inductive hypothesis, there exists $B_1 \supseteq A_1$ such that

$$(t_1, B_1) \in \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta,\eta} \wedge \mathit{Full}(T^{\dagger}, \phi, \delta, B_1)$$

and there exists $B_2 \supseteq A_2$ such that

$$(t_2, B_2) \in \mathit{failures}_{\perp}(\Gamma \vdash Q)_{\delta,\eta} \wedge \mathit{Full}(T^{\dagger}, \psi, \delta, B_2)$$

Let $B_1^{\dagger} = B_1 \cap ([\![\Gamma \vdash S]\!]_{\delta,\eta} \cup B_2)$, $B_2^{\dagger} = B_2 \cap ([\![\Gamma \vdash S]\!]_{\delta,\eta} \cup B_1)$ and $B = B_1^{\dagger} \cup B_2^{\dagger}$. It is then straightforward to check that $B \supseteq A$ and

$$(u, B) \in \mathit{failures}_{\perp}(\Gamma \vdash r)_{\delta,\eta} \wedge \mathit{Full}(T^{\dagger}, \chi, \delta, B)$$

as required.

To show that

$$\begin{array}{l} \forall\, u^{\ddagger} \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}.\, \exists\, I \subseteq \mathit{divergences}(\Gamma \vdash r)_{\delta',\eta'}. \\ I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^{\dagger}, \rho, \delta, \delta', u^{\ddagger}, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^{\dagger}, \chi, \rho, \delta, \delta', u^{\ddagger}, I) \end{array}$$

suppose $u^{\ddagger} \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}$. Then there exist u, $u^{\dagger}$, t_1 and t_2 such that $u^{\ddagger} = u\hat{\ }u^{\dagger}$, $(t_1, \{\}) \in \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta,\eta}$, $(t_2, \{\}) \in \mathit{failures}_{\perp}(\Gamma \vdash Q)_{\delta,\eta}$, $u \in t_1 \underset{[\![\Gamma \vdash S]\!]_{\delta,\eta}}{\|} t_2$, and either $t_1 \in \mathit{divergences}(\Gamma \vdash P)_{\delta,\eta}$ or $t_2 \in \mathit{divergences}(\Gamma \vdash Q)_{\delta,\eta}$. Let $D_1, D_2 \subseteq \{1, \ldots, |u|\}$ be such that

$$\begin{array}{rcl} u \upharpoonright D_1 & = & t_1 \\ u \upharpoonright D_2 & = & t_2 \\ D_1 \cup D_2 & = & \{1, \ldots, |u|\} \\ D_1 \cap D_2 & = & \{k \in \{1, \ldots, |u|\} \mid u(k) \in [\![\Gamma \vdash S]\!]_{\delta,\eta}\} \end{array}$$

Without loss of generality, we can assume that $t_1 \in \mathit{divergences}(\Gamma \vdash P)_{\delta,\eta}$. By the inductive hypothesis, there exists $I_1 \subseteq \mathit{divergences}(\Gamma \vdash P)_{\delta,\eta}$ such that

$$I_1 \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^{\dagger}, \rho, \delta, \delta', t_1, I_1) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^{\dagger}, \phi, \rho, \delta, \delta', t_1, I_1) \qquad (4.15)$$

and there exists $J_2 \subseteq \mathit{failures}_{\perp}(\Gamma \vdash Q)_{\delta,\eta}$ such that

$$J_2 \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T^{\dagger}, \rho, \delta, \delta', (t_2, \{\}), J_2) \wedge \mathit{Closed}^{\mathrm{Fa}}(T^{\dagger}, \psi, \rho, \delta, \delta', (t_2, \{\}), J_2)$$

Letting

$$I_2 = \{t_2' \mid \exists\, A_2'.(t_2', A_2') \in J_2$$

we have that

$$I_2 \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t_2, I_2) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \psi, \rho, \delta, \delta', t_2, I_2) \tag{4.16}$$

Now let

$$\begin{aligned} I = \{u'^\frown u'^\dagger \mid\ & u' \in (\llbracket T^\dagger \rrbracket_{\delta'})^{|u|} \wedge u' \restriction D_1 \in I_1 \wedge u' \restriction D_2 \in I_2 \\ & \wedge D_1 \cap D_2 = \{k \in \{1, \ldots, |u|\} \mid u'(k) \in \llbracket \Gamma \vdash S \rrbracket_{\delta', \eta'}\} \\ & \wedge u^\dagger (\llbracket T^\dagger \rrbracket_{\rho,\delta,\delta'})^* u'^\dagger\} \end{aligned}$$

It is straightforward to check that $I \subseteq \mathit{divergences}(\Gamma \vdash r)_{\delta', \eta'}$.

To show that $I \neq \{\}$, let $u'^0 = \langle\rangle$, and pick any $t'^0_1 \in I_1$ and $t'^0_2 \in I_2$. Now recall the assumption that, for each $X \in \mathit{AllTypeVars}$, either $\Gamma \vdash r$ satisfies $(\mathbf{NoEqT}_X)$, or $\rho\llbracket X \rrbracket$ is a partial function. Whenever $\Gamma \vdash r$ satisfies $(\mathbf{NoEqT}_X)$, the leastness of ϕ and ψ with respect to $\subseteq$ gives us that, for any i and j for which X occurs free in $T^\dagger_{i,j}$ and S has an elem which is either $*_{T^\dagger}$ or of the form $id_i^{T^\dagger}.e$, we have that

$$\phi(id_i, j) \subseteq \{?\} \ \vee\ \psi(id_i, j) \subseteq \{?\}$$

Using those observations, and using (4.15) and (4.16), it follows by induction on $k \in \{1, \ldots, |u|\}$ that there exist $u'^k \in (\llbracket T^\dagger \rrbracket_{\delta'})^k$, $t'^k_1 \in I_1$ and $t'^k_2 \in I_2$ such that

$$\begin{aligned} u'^k \restriction (D_1 \cap \{1, \ldots, k\}) &= t'^k_1 \restriction \{1, \ldots, |D_1 \cap \{1, \ldots, k\}|\} \\ u'^k \restriction (D_2 \cap \{1, \ldots, k\}) &= t'^k_2 \restriction \{1, \ldots, |D_2 \cap \{1, \ldots, k\}|\} \\ D_1 \cap D_2 \cap \{1, \ldots, k\} &= \{k^\dagger \in \{1, \ldots, k\} \mid u'^k(k^\dagger) \in \llbracket \Gamma \vdash S \rrbracket_{\delta', \eta'}\} \\ t'^k_1 \restriction \{1, \ldots, |D_1 \cap \{1, \ldots, k-1\}|\} &= t'^{k-1}_1 \\ t'^k_2 \restriction \{1, \ldots, |D_2 \cap \{1, \ldots, k-1\}|\} &= t'^{k-1}_2 \end{aligned}$$

In particular, we have that

$$\begin{aligned} & u'^{|u|} \in (\llbracket T^\dagger \rrbracket_{\delta'})^{|u|} \wedge u'^{|u|} \restriction D_1 \in I_1 \wedge u'^{|u|} \restriction D_2 \in I_2 \\ & \wedge D_1 \cap D_2 = \{k^\dagger \in \{1, \ldots, |u|\} \mid u'^{|u|}(k^\dagger) \in \llbracket \Gamma \vdash S \rrbracket_{\delta', \eta'}\} \end{aligned}$$

It remains to observe that, by the assumption that $\rho\llbracket X \rrbracket$ is total for each $X \in \mathit{AllTypeVars}$, there exists $u'^\dagger$ such that $u^\dagger (\llbracket T^\dagger \rrbracket_{\rho,\delta,\delta'})^* u'^\dagger$.

Showing that $\mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', u^\ddagger, I)$ is straightforward.

To show that $\mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \chi, \rho, \delta, \delta', u^\ddagger, I)$, suppose $u'^\ddagger \in I$ and suppose k, i, v, v', j and w' are such that

$$u^\ddagger(k) = (id_i, v) \wedge u'^\ddagger(k) = (id_i, v') \wedge \chi(id_i, j) \subseteq \{?, \$\} \wedge (v(j))(\llbracket T^\ddagger_{i,j} \rrbracket_{\rho,\delta,\delta'}) w'$$

If $k \leq |u|$, then by $\phi(id_i, j) \subseteq \{?, \$\}$, $\psi(id_i, j) \subseteq \{?, \$\}$, (4.15), (4.16), and an argument like the argument above which showed that $I \neq \{\}$, it follows that

$$\exists\, u''^\ddagger \in I. u''^\ddagger \restriction \{1, \ldots, k-1\} = u'^\ddagger \restriction \{1, \ldots, k-1\} \wedge u''^\ddagger(k) = (id_i, v'[w'/j])$$

Otherwise, we can simply define

$$\begin{array}{rcl} u''^{\ddagger} \upharpoonright \{1,\ldots,k-1,k+1,\ldots,|u''^{\ddagger}|\} & = & u'^{\ddagger} \upharpoonright \{1,\ldots,k-1,k+1,\ldots,|u'^{\ddagger}|\} \\ u''^{\ddagger}(k) & = & (id_i, v'[w'/j]) \end{array}$$

and $u''^{\ddagger} \in I$ is immediate by the definition of I.

To show that

$$\begin{array}{l} \forall (u,A) \in \mathit{failures}_{\perp}(\Gamma \vdash r)_{\delta,\eta}.\, \exists\, J \subseteq \mathit{failures}_{\perp}(\Gamma \vdash r)_{\delta',\eta'}. \\ J \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T^{\dagger}, \rho, \delta, \delta', (u,A), J) \wedge \mathit{Closed}^{\mathrm{Fa}}(T^{\dagger}, \chi, \rho, \delta, \delta', (u,A), J) \end{array}$$

suppose $(u,A) \in \mathit{failures}_{\perp}(\Gamma \vdash r)_{\delta,\eta}$.

If $u \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}$, then by what we have shown above, there exists $I \subseteq \mathit{divergences}(\Gamma \vdash r)_{\delta',\eta'}$ such that

$$I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^{\dagger}, \rho, \delta, \delta', u, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^{\dagger}, \chi, \rho, \delta, \delta', u, I)$$

It is then straightforward to check that taking

$$J = \{(u', [\![T^{\dagger}]\!]_{\delta'}) \mid u' \in I\}$$

suffices.

Otherwise, there exist $(t_1, A_1) \in \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta,\eta}$ and $(t_2, A_2) \in \mathit{failures}_{\perp}(\Gamma \vdash Q)_{\delta,\eta}$ such that $u \in t_1 \underset{[\![\Gamma \vdash S]\!]_{\delta,\eta}}{\|} t_2$, $A_1 \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta} = A_2 \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta}$ and $A = A_1 \cup A_2$. Let $D_1, D_2 \subseteq \{1,\ldots,|u|\}$ be such that

$$\begin{array}{rcl} u \upharpoonright D_1 & = & t_1 \\ u \upharpoonright D_2 & = & t_2 \\ D_1 \cup D_2 & = & \{1,\ldots,|u|\} \\ D_1 \cap D_2 & = & \{k \in \{1,\ldots,|u|\} \mid u(k) \in [\![\Gamma \vdash S]\!]_{\delta,\eta}\} \end{array}$$

By the inductive hypothesis, there exists $B_1 \supseteq A_1$ such that

$$(t_1, B_1) \in \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta,\eta} \wedge \mathit{Full}(T^{\dagger}, \phi, \delta, B_1)$$

and there exists $B_2 \supseteq A_2$ such that

$$(t_2, B_2) \in \mathit{failures}_{\perp}(\Gamma \vdash Q)_{\delta,\eta} \wedge \mathit{Full}(T^{\dagger}, \psi, \delta, B_2)$$

Then, also by the inductive hypothesis, there exists $J_1 \subseteq \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta',\eta'}$ such that

$$\begin{array}{l} J_1 \neq \{\} \\ \wedge\, \mathit{Related}^{\mathrm{Fa}}(T^{\dagger}, \rho, \delta, \delta', (t_1, B_1), J_1) \\ \wedge\, \mathit{Closed}^{\mathrm{Fa}}(T^{\dagger}, \phi, \rho, \delta, \delta', (t_1, B_1), J_1) \end{array} \qquad (4.17)$$

and there exists $J_2 \subseteq \mathit{failures}_\perp(\Gamma \vdash Q)_{\delta',\eta'}$ such that

$$\begin{array}{l} J_2 \neq \{\} \\ \wedge\ \mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (t_2, B_2), J_2) \\ \wedge\ \mathit{Closed}^{\mathrm{Fa}}(T^\dagger, \psi, \rho, \delta, \delta', (t_2, B_2), J_2) \end{array} \tag{4.18}$$

Now let

$$\begin{array}{rl} J = \{(u', B') \mid & \exists (t'_1, B'_1) \in J_1, (t'_2, B'_2) \in J_2. \\ & u' \in (\llbracket T^\dagger \rrbracket_{\delta'})^{|u|} \wedge u' \restriction D_1 = t'_1 \wedge u' \restriction D_2 = t'_2 \\ & \wedge\ D_1 \cap D_2 = \{k \in \{1, \ldots, |u|\} \mid u'(k) \in \llbracket \Gamma \vdash S \rrbracket_{\delta',\eta'}\} \\ & \wedge\ B' = ((B'_1 \cup B'_2) \cap \llbracket \Gamma \vdash S \rrbracket_{\delta',\eta'}) \cup (B'_1 \cap B'_2)\} \end{array}$$

It is straightforward to show that $J \subseteq \mathit{failures}_\perp(\Gamma \vdash r)_{\delta',\eta'}$.

Showing that $J \neq \{\}$ is very similar to showing that $I \neq \{\}$ in the proof of

$$\begin{array}{l} \forall u^\ddagger \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}. \exists I \subseteq \mathit{divergences}(\Gamma \vdash r)_{\delta',\eta'}. \\ I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', u^\ddagger, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \chi, \rho, \delta, \delta', u^\ddagger, I) \end{array}$$

above.

To show that $\mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (u, A), J)$, suppose $(u', B') \in J$, and let $(t'_1, B'_1) \in J_1$ and $(t'_2, B'_2) \in J_2$ be as in the definition of J.

Showing that $u(\llbracket T^\dagger \rrbracket_{\rho,\delta,\delta'})^* u'$ is straightforward.

Suppose $\sigma : \delta \leftrightarrow \delta'$ is such that

$$\forall X \in \mathit{AllTypeVars}. (\sigma\llbracket X \rrbracket)^{-1} : (\delta'\llbracket X \rrbracket \setminus \mathit{Range}(\rho\llbracket X \rrbracket)) \to \delta\llbracket X \rrbracket$$

and suppose $b' \in (\llbracket T^\dagger \rrbracket_{\delta'} \setminus B')$.

If $b' \in \llbracket \Gamma \vdash S \rrbracket_{\delta',\eta'}$, then $b' \in (\llbracket T^\dagger \rrbracket_{\delta'} \setminus B'_1)$ and $b' \in (\llbracket T^\dagger \rrbracket_{\delta'} \setminus B'_2)$. Hence, by (4.17) and (4.18), there exists $b_1 \in (\llbracket T^\dagger \rrbracket_\delta \setminus B_1)$ such that $b_1(\llbracket T^\dagger \rrbracket_{(\rho\cup\sigma),\delta,\delta'})b'$, and there exists $b_2 \in (\llbracket T^\dagger \rrbracket_\delta \setminus B_2)$ such that $b_2(\llbracket T^\dagger \rrbracket_{(\rho\cup\sigma),\delta,\delta'})b'$. We then have that b_1 and b_2 are related by the equivalence relation

$$\llbracket T^\dagger \rrbracket_{\delta\times\delta,\delta,\delta} : \llbracket T^\dagger \rrbracket_\delta \leftrightarrow \llbracket T^\dagger \rrbracket_\delta$$

(see Proposition 4.1.3 and Corollary 4.1.6), and that their equivalence class is a subset of $\llbracket \Gamma \vdash S \rrbracket_{\delta,\eta}$ (see (4.14) and (4.11)). In particular, b_1 and b_2 are of the forms (id_i, w_1) and (id_i, w_2). Now let $v \in \llbracket \prod_{j=1}^{m_i} T^\dagger_{i,j} \rrbracket_\delta$ be defined by

$$v(j) = \begin{cases} w_1(j), & \mathit{if}\ \phi(id_i, j) \not\subseteq \{?\} \\ w_2(j), & \mathit{if}\ \phi(id_i, j) \subseteq \{?\} \end{cases}$$

and let $a = (id_i, v)$. Then $a(\llbracket T^\dagger \rrbracket_{(\rho\cup\sigma),\delta,\delta'})b'$, and a belongs to the equivalence class of b_1 and b_2. Furthermore, using the assumption that, for each $X \in \mathit{AllTypeVars}$, either $\Gamma \vdash r$ satisfies ($\mathbf{NoEqT}_X$) or $\rho\llbracket X \rrbracket$ is injective, and using the facts that $\mathit{Full}(T^\dagger, \phi, \delta, B_1)$ and $\mathit{Full}(T^\dagger, \psi, \delta, B_2)$, it follows that $a \in (\llbracket T^\dagger \rrbracket_\delta \setminus B_1)$ and $a \in (\llbracket T^\dagger \rrbracket_\delta \setminus B_2)$, and so $a \in (\llbracket T^\dagger \rrbracket_\delta \setminus A)$ as required.

If $b' \notin [\![\Gamma \vdash S]\!]_{\delta',\eta'}$, then without loss of generality we can assume that $b' \in ([\![T^\dagger]\!]_{\delta'} \setminus B'_1)$. Hence, as above, there exists $b_1 \in ([\![T^\dagger]\!]_\delta \setminus B_1)$ such that $b_1([\![T^\dagger]\!]_{(\rho\cup\sigma),\delta,\delta'})b'$, and we have that $b_1 \notin [\![\Gamma \vdash S]\!]_{\delta,\eta}$. But then $b_1 \in ([\![T^\dagger]\!]_\delta \setminus A)$, so we can take $a = b_1$.

Showing that $\mathit{Closed}^{\mathrm{Fa}}(T^\dagger, \chi, \rho, \delta, \delta', (u, A), J)$ is again very similar to showing that $\mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \chi, \rho, \delta, \delta', u^\ddagger, I)$ above.

Hiding. Here

$$\begin{array}{rcl} r & = & P \setminus S \\ T & = & \mathit{Proc}_{T^\dagger,\chi} \end{array}$$

for some $\Gamma \vdash P : \mathit{Proc}_{T^\dagger,\phi}$ and $\Gamma \vdash S : \mathit{Set}_{T^\dagger}$ such that $T^\dagger = \sum_{i^\dagger=1}^{n} \mathit{id}_{i^\dagger}.\prod_{j^\dagger=1}^{m_{i^\dagger}} T^\dagger_{i^\dagger,j^\dagger}$ and $\phi \setminus S \subseteq \chi$.

By Proposition 2.5.3 (b), we can assume that ϕ is least with respect to $\subseteq$.

As under 'Parallel composition' above, we have that (4.11)–(4.14) hold.

Let us first consider $M = \mathrm{Tr}$.

To show that

$$\forall t \in \mathit{traces}(\Gamma \vdash r)_{\delta,\eta}.\mathit{Recorded}(T^\dagger, \chi, \rho, \delta, \delta', t)$$

suppose $t \in \mathit{traces}(\Gamma \vdash r)_{\delta,\eta}$, $k \in \{1, \ldots, |t|\}$ and

$$\begin{array}{l} \forall k^\ddagger, i, v, j. \\ (k^\ddagger < k \wedge t(k^\ddagger) = (\mathit{id}_i, v) \wedge \chi(\mathit{id}_i, j) \not\subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'}) \end{array}$$

Let $t^\dagger \in \mathit{traces}(\Gamma \vdash P)_{\delta,\eta}$ be such that $t = t^\dagger \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta}$, and let $k^\dagger \in \{1, \ldots, |t^\dagger|\}$ be such that $t^\dagger(k^\dagger) \notin [\![\Gamma \vdash S]\!]_{\delta,\eta}$ and $t \restriction \{1, \ldots, k\} = (t^\dagger \restriction \{1, \ldots, k^\dagger\}) \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta}$. By the assumption that, for each $X \in \mathit{AllTypeVars}$, either $\Gamma \vdash r$ satisfies $(\mathbf{RecCho}_X)$ or $\rho[\![X]\!]$ is total, and by the leastness of ϕ with respect to $\subseteq$, it follows that

$$\begin{array}{l} \forall k^\ddagger, i, v, j. \\ (k^\ddagger < k^\dagger \wedge t^\dagger(k^\ddagger) = (\mathit{id}_i, v) \wedge \phi(\mathit{id}_i, j) \not\subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'}) \end{array}$$

Hence, by the inductive hypothesis, we have that

$$\begin{array}{l} \forall k^\ddagger, i, v, j. \\ (k^\ddagger \leq k^\dagger \wedge t^\dagger(k^\ddagger) = (\mathit{id}_i, v) \wedge \phi(\mathit{id}_i, j) \subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'}) \end{array}$$

and so

$$\begin{array}{l} \forall k^\ddagger, i, v, j. \\ (k^\ddagger \leq k \wedge t(k^\ddagger) = (\mathit{id}_i, v) \wedge \chi(\mathit{id}_i, j) \subseteq \{!\}) \Rightarrow \\ v(j) \in \mathit{Domain}([\![T^\dagger_{i,j}]\!]_{\rho,\delta,\delta'}) \end{array}$$

as required.

The rest of the case $M = \mathrm{Tr}$ and the whole case $M = \mathrm{Fa}$ are not more complex than what is involved for $M = \mathrm{FD}$, so let us consider $M = \mathrm{FD}$.

Showing that

$$\begin{array}{l}\forall (t, A) \in \mathit{failures}_\bot(\Gamma \vdash r)_{\delta,\eta}.\, \exists\, B \supseteq A. \\ (t, B) \in \mathit{failures}_\bot(\Gamma \vdash r)_{\delta,\eta} \wedge \mathit{Full}(T^\dagger, \chi, \delta, B)\end{array}$$

is straightforward.

To show that

$$\begin{array}{l}\forall\, t \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}.\, \exists\, I \subseteq \mathit{divergences}(\Gamma \vdash r)_{\delta',\eta'}. \\ I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \chi, \rho, \delta, \delta', t, I)\end{array}$$

suppose $t \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}$.

The case when there exist $t^\dagger \in \mathit{divergences}(\Gamma \vdash P)_{\delta,\eta}$ and $t^\ddagger$ with $t = (t^\dagger \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta})\hat{}\,t^\ddagger$ is straightforward.

Otherwise, there exist $u \in ([\![T^\dagger]\!]_\delta)^\omega$ and $t^\ddagger$ such that $t = (u \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta})\hat{}\,t^\ddagger$ and

$$\forall\, k \in \mathbb{N}.(u \upharpoonright \{1, \ldots, k\}, \{\}) \in \mathit{failures}_\bot(\Gamma \vdash P)_{\delta,\eta}$$

By the inductive hypothesis, for each $k \in \mathbb{N}$, there exists $J_k \subseteq \mathit{failures}_\bot(\Gamma \vdash P)_{\delta',\eta'}$ such that

$$\begin{array}{l}J_k \neq \{\} \\ \wedge\, \mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (u \upharpoonright \{1, \ldots, k\}, \{\}), J_k) \\ \wedge\, \mathit{Closed}^{\mathrm{Fa}}(T, \phi, \rho, \delta, \delta', (u \upharpoonright \{1, \ldots, k\}, \{\}), J_k)\end{array}$$

Let

$$K_k = \{{t'}^\dagger \mid \exists\, {A'}^\dagger.({t'}^\dagger, {A'}^\dagger) \in J_k\}$$

and let

$$L_k = \{{t'}^\dagger \upharpoonright \{1, \ldots, k\} \mid \exists\, k^\dagger \geq k.{t'}^\dagger \in K_{k^\dagger}\}$$

Then, for each $k \in \mathbb{N}$, we have that

$$\begin{array}{l}\{({t'}^\dagger, \{\}) \mid {t'}^\dagger \in L_k\} \subseteq \mathit{failures}_\bot(\Gamma \vdash P)_{\delta',\eta'} \\ \wedge\, L_k \neq \{\} \\ \wedge\, \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', u \upharpoonright \{1, \ldots, k\}, L_k) \\ \wedge\, \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \phi, \rho, \delta, \delta', u \upharpoonright \{1, \ldots, k\}, L_k) \\ \wedge\, \forall {t'}^\dagger \in L_{k+1}.{t'}^\dagger \upharpoonright \{1, \ldots, k\} \in L_k\end{array} \tag{4.19}$$

Now let

$$\begin{array}{ll}I = \{(u' \setminus [\![\Gamma \vdash S]\!]_{\delta',\eta'})\hat{}\,{t'}^\ddagger \mid & u' \in ([\![T^\dagger]\!]_{\delta'})^\omega \wedge \forall\, k \in \mathbb{N}.u' \upharpoonright \{1, \ldots, k\} \in L_k \\ & \wedge\, t^\ddagger([\![T^\dagger]\!]_{\rho,\delta,\delta'})^* {t'}^\ddagger\}\end{array}$$

It is straightforward to show that $I \subseteq \mathit{divergences}(\Gamma \vdash r)_{\delta',\eta'}$.

To show that $I \neq \{\}$, observe that, by 'Restrictions with FD' (ii) in Section 2.5.2, each L_k is a finite set. Hence, by König's Lemma (see e.g. [Kun80, Chapter II, Lemma 5.7] or [Ros98a, Theorem 7.4.1]), there exists $u' \in (\llbracket T^\dagger \rrbracket_{\delta'})^\omega$ such that $\forall\, k \in \mathbb{N}.u' \restriction \{1,\ldots,k\} \in L_k$. It then remains to recall that, for each $X \in \mathit{AllTypeVars}$, $\rho\llbracket X \rrbracket$ is total, so that there exists $t'^\ddagger$ such that $t^\ddagger(\llbracket T^\dagger \rrbracket_{\rho,\delta,\delta'})^* t'^\ddagger$.

Showing that $\mathit{Related}^{\mathrm{Tr}}(T^\dagger,\rho,\delta,\delta',t,I)$ is straightforward.

To show that $\mathit{Closed}^{\mathrm{Tr}}(T^\dagger,\chi,\rho,\delta,\delta',t,I)$, suppose $t' \in I$ and suppose l, i, v, v', j and w' are such that

$$t(l) = (id_i, v) \wedge t'(l) = (id_i, v') \wedge \chi(id_i, j) \subseteq \{?,\$\} \wedge (v(j))(\llbracket T^\dagger_{i,j} \rrbracket_{\rho,\delta,\delta'})w'$$

By the definition of I, there exist u' and $t'^\ddagger$ such that

$$\begin{aligned}&t' = (u' \setminus \llbracket \Gamma \vdash S \rrbracket_{\delta',\eta'})\hat{}\,t'^\ddagger\\ &\wedge u' \in (\llbracket T^\dagger \rrbracket_{\delta'})^\omega \wedge \forall\, k \in \mathbb{N}.u' \restriction \{1,\ldots,k\} \in L_k\\ &\wedge t^\ddagger(\llbracket T^\dagger \rrbracket_{\rho,\delta,\delta'})^* t'^\ddagger\end{aligned}$$

The case $l > |u \setminus \llbracket \Gamma \vdash S \rrbracket_{\delta,\eta}|$ is immediate.

Otherwise, let $l^\dagger \in \mathbb{N}$ be such that $|(u \restriction \{1,\ldots,l^\dagger\}) \setminus \llbracket \Gamma \vdash S \rrbracket_{\delta,\eta}| = l$ and $u(l^\dagger) \notin \llbracket \Gamma \vdash S \rrbracket_{\delta,\eta}$. We then have that

$$\begin{aligned}&u(l^\dagger) = (id_i, v) \wedge u'(l^\dagger) = (id_i, v')\\ &\wedge \phi(id_i, j) \subseteq \{?,\$\} \wedge (v(j))(\llbracket T^\dagger_{i,j} \rrbracket_{\rho,\delta,\delta'})w'\end{aligned} \tag{4.20}$$

Now, for each $k \geq l^\dagger$, let

$$\begin{aligned}M_k = \{t''^\dagger \in L_k \mid\ & t''^\dagger \restriction \{1,\ldots,l^\dagger - 1\} = u' \restriction \{1,\ldots,l^\dagger - 1\}\\ & \wedge t''^\dagger(l^\dagger) = (id_i, v'[w'/j])\}\end{aligned}$$

Then, for each $k \geq l^\dagger$, it follows that

- $M_k \neq \{\}$ (by (4.19) and (4.20)),
- $\forall\, t''^\dagger \in M_{k+1}.t''^\dagger \restriction \{1,\ldots,k\} \in M_k$ (by (4.19)), and
- M_k is a finite set (by the fact that L_k is a finite set, which was observed above).

Therefore, by König's Lemma again, there exists u'' such that

$$u'' \in (\llbracket T^\dagger \rrbracket_{\delta'})^\omega \wedge \forall\, k \geq l^\dagger.u'' \restriction \{1,\ldots,k\} \in M_k$$

In particular, we have that

$$u'' \restriction \{1,\ldots,l^\dagger - 1\} = u' \restriction \{1,\ldots,l^\dagger - 1\} \wedge u''(l^\dagger) = (id_i, v'[w'/j])$$

Let

$$t'' = (u'' \setminus \llbracket \Gamma \vdash S \rrbracket_{\delta',\eta'})\hat{}\,t'^\ddagger$$

It is then straightforward to check that $t'' \in I$ and

$$t'' \restriction \{1, \ldots, l-1\} = t' \restriction \{1, \ldots, l-1\} \wedge t''(l) = (id_i, v'[w'/j])$$

as required.

To show that

$$\forall (t, A) \in \mathit{failures}_\perp(\Gamma \vdash r)_{\delta,\eta} . \exists J \subseteq \mathit{failures}_\perp(\Gamma \vdash r) \mathit{dpep}.$$
$$J \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (t, A), J) \wedge \mathit{Closed}^{\mathrm{Fa}}(T^\dagger, \chi, \rho, \delta, \delta', (t, A), J)$$

suppose $(t, A) \in \mathit{failures}_\perp(\Gamma \vdash r)_{\delta,\eta}$.

If $t \in \mathit{divergences}(\Gamma \vdash r)_{\delta,\eta}$, then by what we have shown above, there exists

$$I \subseteq \mathit{divergences}(\Gamma \vdash r)_{\delta',\eta'}$$

such that

$$I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \chi, \rho, \delta, \delta', t, I)$$

It is then straightforward to check that taking

$$J = \{(t', [\![T^\dagger]\!]_{\delta'}) \mid t' \in I\}$$

suffices.

Otherwise, there exists $(t^\dagger, A^\dagger) \in \mathit{failures}_\perp(\Gamma \vdash P)_{\delta,\eta}$ such that $t = t^\dagger \setminus [\![\Gamma \vdash S]\!]_{\delta,\eta}$ and $A \cup [\![\Gamma \vdash S]\!]_{\delta,\eta} = A^\dagger$. Then, by the inductive hypothesis, there exists

$$J^\dagger \subseteq \mathit{failures}_\perp(\Gamma \vdash P)_{\delta',\eta'}$$

such that

$$J^\dagger \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (t^\dagger, A^\dagger), J^\dagger) \wedge \mathit{Closed}^{\mathrm{Fa}}(T^\dagger, \phi, \rho, \delta, \delta', (t^\dagger, A^\dagger), J^\dagger)$$

Let

$$J = \{({t'}^\dagger \setminus [\![\Gamma \vdash S]\!]_{\delta',\eta'}, {A'}^\dagger) \mid ({t'}^\dagger, {A'}^\dagger) \in J^\dagger\}$$

To show that $J \subseteq \mathit{failures}_\perp(\Gamma \vdash r)_{\delta',\eta'}$, suppose $({t'}^\dagger, {A'}^\dagger) \in J^\dagger$. By

$$\mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (t^\dagger, A^\dagger), J^\dagger)$$

we have that

$$\forall \sigma : \delta \leftrightarrow \delta' . (\forall X \in \mathit{AllTypeVars} . (\sigma[\![X]\!])^{-1} : (\delta'[\![X]\!] \setminus \mathit{Range}(\rho[\![X]\!])) \rightarrow \delta[\![X]\!]) \Rightarrow$$
$$(\forall a' \in ([\![T^\dagger]\!]_{\delta'} \setminus {A'}^\dagger) . \exists a \in ([\![T^\dagger]\!]_\delta \setminus A^\dagger) . a([\![T^\dagger]\!]_{(\rho\cup\sigma),\delta,\delta'})a')$$

By the surjectivity of $[\![T^\dagger]\!]_{(\rho\cup\sigma),\delta,\delta'}$ (see Proposition 4.1.3), by Proposition 4.1.4, by (4.13) and by $[\![\Gamma \vdash S]\!]_{\delta,\eta} \subseteq A^\dagger$, it then follows that $[\![\Gamma \vdash S]\!]_{\delta',\eta'} \subseteq A'^\dagger$. Therefore, since $(t'^\dagger, A'^\dagger) \in$ *failures*$_\perp(\Gamma \vdash P)_{\delta',\eta'}$, we have that

$$(t'^\dagger \setminus [\![\Gamma \vdash S]\!]_{\delta',\eta'}, A'^\dagger) \in \mathit{failures}_\perp(\Gamma \vdash r)_{\delta',\eta'}$$

as required.

Showing that $J \neq \{\}$, $\mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (t, A), J)$ and $\mathit{Closed}^{\mathrm{Fa}}(T^\dagger, \chi, \rho, \delta, \delta', (t, A), J)$ is straightforward.

Recursion. Here

$$\begin{array}{rcl} T & = & U \to \mathit{Proc}_{T^\dagger,\phi} \\ r & = & \mu\, p : T.P \end{array}$$

where $U^{\mathit{AllTypeVars},+,\times,\mu,\to}$ and $(\Gamma \setminus \{p\}) \cup \{p : T\} \vdash P : T$.

As under 'Recursion' in Section 2.5.2, let $\mathcal{G} : [\![T \to T]\!]^M_\delta$ and $\mathcal{G}' : [\![T \to T]\!]^M_{\delta'}$ be defined by

$$\begin{array}{rcl} \mathcal{G}(g) & = & [\![(\Gamma \setminus \{p\}) \cup \{p : T\} \vdash P]\!]^M_{\delta,\eta[g/p]} \\ \mathcal{G}'(g') & = & [\![(\Gamma \setminus \{p\}) \cup \{p : T\} \vdash P]\!]^M_{\delta',\eta'[g'/p]} \end{array}$$

and let $\perp$ and $\perp'$ be the least elements of $[\![T]\!]^M_\delta$ and $[\![T]\!]^M_{\delta'}$, so that

$$\begin{array}{rcl} [\![\Gamma \vdash r]\!]^M_{\delta,\eta} & = & \bigsqcup_{k\in\mathbb{N}} \mathcal{G}^k(\perp) \\ [\![\Gamma \vdash r]\!]^M_{\delta',\eta'} & = & \bigsqcup_{k\in\mathbb{N}} \mathcal{G}'^k(\perp') \end{array}$$

By Proposition 4.1.7, we have that $\perp([\![T]\!]^M_{\rho,\delta,\delta'})\perp'$. By the inductive hypothesis, we have that $\mathcal{G}([\![T \to T]\!]^M_{\rho,\delta,\delta'})\mathcal{G}'$. Hence, it follows that

$$\forall\, k \in \mathbb{N}.(\mathcal{G}^k(\perp))([\![T]\!]^M_{\rho,\delta,\delta'})(\mathcal{G}'^k(\perp'))$$

But then, by Proposition 4.1.7 again, we have that

$$([\![\Gamma \vdash r]\!]^M_{\delta,\eta})([\![T]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash r]\!]^M_{\delta',\eta'})$$

as required.

■

The following corollary, which is straightforward to obtain from Theorem 4.2.2, states that whether a refinement between two processes holds or fails is preserved by bijections.

Corollary 4.2.3 *Suppose that:*

- *$\Gamma \vdash P : Proc_{T,\phi}$ and $\Gamma \vdash Q : Proc_{T,\psi}$ are processes;*
- *M specifies one of the models Tr, Fa, FD of CSP;*
- *$\rho : \delta \leftrightarrow \delta'$ is such that, for each $X \in AllTypeVars$, $\rho[\![X]\!]$ is a bijection (i.e. it is a total function, injective and surjective);*
- *η and η' are assignments to Γ with respect to M and δ, and M and δ' (respectively);*
- $\forall (x : U) \in \Gamma . (\eta[\![x]\!])([\![U]\!]^M_{\rho,\delta,\delta'})(\eta'[\![x]\!])$

Then $(\Gamma \vdash P) \sqsubseteq^M_{\delta,\eta} (\Gamma \vdash Q)$ *if and only if* $(\Gamma \vdash P) \sqsubseteq^M_{\delta',\eta'} (\Gamma \vdash Q)$. ■

Example 4.2.1 Consider the process *BImpl* from Example 2.1.3. In our language, it can be written as a term

$$\{\} \vdash BImpl : Proc_{T^\dagger,\phi}$$

where

$$T^\dagger = in.X + mid.X + out.X$$

for some $X \in NEFTypeVars$, and where

$$\phi(in, 1) = \{?\} \qquad \phi(mid, 1) = \{\} \qquad \phi(out, 1) = \{!\}$$

Moreover, for any $Y \in AllTypeVars$, $\{\} \vdash BImpl$ satisfies ($\mathbf{NoEqT}_Y$) and ($\mathbf{RecCho}_Y$).

Therefore, for any M, δ, δ' and $\rho : \delta \leftrightarrow \delta'$ such that either $M =$ Tr or $\rho[\![Y]\!]$ is total for each $Y \in AllTypeVars$, Theorem 4.2.2 applies and gives us that

$$([\![\{\} \vdash BImpl]\!]^M_\delta)([\![Proc_{T^\dagger,\phi}]\!]^M_{\rho,\delta,\delta'})([\![\{\} \vdash BImpl]\!]^M_{\delta'}) \tag{4.21}$$

For $M =$ Tr, (4.21) is

$$\begin{array}{l} (\forall\, t_1 \in traces(\{\} \vdash BImpl)_\delta . Recorded(T^\dagger, \phi, \rho, \delta, \delta', t_1)) \wedge \\ (\forall\, t_2 \in traces(\{\} \vdash BImpl)_\delta . t_2 \in Domain(([\![T^\dagger]\!]_{\rho,\delta,\delta'})^*) \Rightarrow \\ \exists\, I \subseteq traces(\{\} \vdash BImpl)_{\delta'} . \\ I \neq \{\} \wedge Related^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t_2, I) \wedge Closed^{\mathrm{Tr}}(T^\dagger, \phi, \rho, \delta, \delta', t_2, I)) \end{array}$$

which, for example, holds with

$$\begin{array}{rcl} \delta[\![X]\!] &=& \{0\} \\ \delta'[\![X]\!] &=& \{0, 1, 2\} \\ \rho[\![X]\!] &=& \{(0,0), (0,1), (0,2)\} \\ t_1 &=& \langle (in, 0), (in, 0), (out, 0), (out, 0) \rangle \\ t_2 &=& t_1 \\ I &=& \{\langle (in, v_1), (in, v_2), (out, v_1), (out, v_2) \rangle \mid v_1, v_2 \in \delta'[\![X]\!]\} \end{array}$$

and with

$$\begin{array}{rcl} \delta[\![X]\!] &=& \{0, 1, 2, 3, 4\} \\ \delta'[\![X]\!] &=& \{0, 1\} \\ \rho[\![X]\!] &=& \{(0,0), (1,1)\} \\ t_1 &=& \langle (in, 1), (out, 1), (in, 0), (in, 3), (out, 0), (out, 3) \rangle \\ t_2 &=& \langle (in, 1), (out, 1), (in, 0) \rangle \\ I &=& \{t_2\} \end{array}$$

In particular, the former case illustrates an application of Theorem 4.2.2 to $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ in the proof of Theorem 5.1.3, and the latter case illustrates an application of Theorem 4.2.2 to $\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$ in the proof of Theorem 5.2.2).

For $M = \mathrm{Fa}$, (4.21) is

$$\begin{array}{l}
(\forall t_1 \in \mathit{traces}(\{\} \vdash \mathit{BImpl})_\delta .\, \exists I \subseteq \mathit{traces}(\{\} \vdash \mathit{BImpl})_{\delta'} . \\
I \neq \{\} \wedge \mathit{Related}^{\mathrm{Tr}}(T^\dagger, \rho, \delta, \delta', t_1, I) \wedge \mathit{Closed}^{\mathrm{Tr}}(T^\dagger, \phi, \rho, \delta, \delta', t_1, I)) \wedge \\
(\forall (t_2, A_2) \in \mathit{failures}(\{\} \vdash \mathit{BImpl})_\delta .\, \exists B \supseteq A_2 . \\
(t_2, B) \in \mathit{failures}(\{\} \vdash \mathit{BImpl})_\delta \wedge \mathit{Full}(T^\dagger, \phi, \delta, B)) \wedge \\
(\forall (t_3, A_3) \in \mathit{failures}(\{\} \vdash \mathit{BImpl})_\delta .\, \exists J \subseteq \mathit{failures}(\{\} \vdash \mathit{BImpl})_{\delta'} . \\
J \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T^\dagger, \rho, \delta, \delta', (t_3, A_3), J) \wedge \mathit{Closed}^{\mathrm{Fa}}(T^\dagger, \phi, \rho, \delta, \delta', (t_3, A_3), J))
\end{array}$$

which, for example, holds with

$$\begin{array}{rcl}
\delta[\![X]\!] & = & \{0,1,2,3,4\} \\
\delta'[\![X]\!] & = & \{0,1\} \\
\rho[\![X]\!] & = & \{(0,0),(1,0),(2,0),(3,1),(4,0)\} \\
t_1 & = & \langle (\mathit{in},1),(\mathit{out},1),(\mathit{in},0),(\mathit{in},3),(\mathit{out},0),(\mathit{out},3)\rangle \\
I & = & \{\langle (\mathit{in},0),(\mathit{out},0),(\mathit{in},0),(\mathit{in},1),(\mathit{out},0),(\mathit{out},1)\rangle \\
t_2 & = & \langle (\mathit{in},1),(\mathit{out},1),(\mathit{in},0),(\mathit{in},3)\rangle \\
A_2 & = & \{(\mathit{in},1),(\mathit{mid},3),(\mathit{out},3)\} \\
B & = & \{(\mathit{in},v) \mid v \in \delta[\![X]\!]\} \cup \{(\mathit{mid},3),(\mathit{out},3)\} \\
t_3 & = & t_2 \\
A_3 & = & A_2 \\
J & = & \{(\langle (\mathit{in},0),(\mathit{out},0),(\mathit{in},0),(\mathit{in},1)\rangle, \{(\mathit{in},0),(\mathit{mid},1),(\mathit{out},1)\})\}
\end{array}$$

In particular, this illustrates an application of Theorem 4.2.2 to $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ in the proof of Theorem 5.3.6 (which is, however, with $M = \mathrm{Tr}$ rather than $M = \mathrm{Fa}$). See also Example 5.3.1.

Since $\mathit{divergences}(\{\} \vdash \mathit{BImpl})_\delta = \{\}$ for any δ, the case $M = \mathrm{FD}$ is no more interesting than the case $M = \mathrm{Fa}$. ■

Example 4.2.2 Consider the process *MultMatr* from Example 2.7.4. In our language, it can be written as a term

$$\Gamma \vdash \mathit{MultMatr} : (\mathit{Num} \times \mathit{Num}) \rightarrow \mathit{Proc}_{T^\dagger,\psi}$$

where

$$\begin{array}{rcl}
\Gamma & = & \{\mathit{zer} : X, \mathit{add} : (X \times X) \rightarrow X, \mathit{mult} : (X \times X) \rightarrow X\} \\
T^\dagger & = & \mathit{hor}.(\mathit{Num} \times \mathit{Num} \times X) + \mathit{vert}.(\mathit{Num} \times \mathit{Num} \times X) + \\
 & & \mathit{finished}.(\mathit{Num} \times \mathit{Num}) + \mathit{out}.(\mathit{Num} \times \mathit{Num} \times X)
\end{array}$$

for some $X \in \mathit{NEFTypeVars}$, and where

$$\begin{array}{lll}
\psi(\mathit{hor},1) = \{!\} & \psi(\mathit{hor},2) = \{!\} & \psi(\mathit{hor},3) = \{!,?\} \\
\psi(\mathit{vert},1) = \{!\} & \psi(\mathit{vert},2) = \{!\} & \psi(\mathit{vert},3) = \{!,?\} \\
\psi(\mathit{finished},1) = \{!\} & \psi(\mathit{finished},2) = \{!\} & \\
\psi(\mathit{out},1) = \{!\} & \psi(\mathit{out},2) = \{!\} & \psi(\mathit{out},3) = \{!\}
\end{array}$$

Moreover, $\Gamma \vdash \mathit{MultMatr}$ satisfies:

- (**NoEqT**$_Y$) if and only if $Y \neq X$;
- (**PosConjEqT**$_Y$) for all Y;
- (**RecCho**$_Y$) if and only if $Y \neq X$.

Therefore, for any M, δ, δ' and $\rho : \delta \leftrightarrow \delta'$ such that

- $\rho[\![X]\!]$ is total and a partial function;
- either $M = \mathrm{Tr}$ or $\rho[\![X]\!]$ is injective;
- either $M = \mathrm{Tr}$ or $\rho[\![Y]\!]$ is total for all $Y \neq X$;

and for any η and η' such that

$$\begin{array}{l}(\eta[\![zer]\!])([\![X]\!]^M_{\rho,\delta,\delta'})(\eta'[\![zer]\!])\\ (\eta[\![add]\!])([\![(X \times X) \to X]\!]^M_{\rho,\delta,\delta'})(\eta'[\![add]\!])\\ (\eta[\![mult]\!])([\![(X \times X) \to X]\!]^M_{\rho,\delta,\delta'})(\eta'[\![mult]\!])\end{array}$$

Theorem 4.2.2 applies and gives us that

$$([\![\Gamma \vdash \mathit{MultMatr}]\!]^M_{\delta,\eta})([\![(\mathit{Num} \times \mathit{Num}) \to \mathit{Proc}_{T^\dagger,\psi}]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash \mathit{MultMatr}]\!]^M_{\delta',\eta'}) \tag{4.22}$$

For example, (4.22) holds with

$$\begin{array}{l}M = \mathrm{Tr}\\ \textit{any } \delta\\ \textit{any } \delta' \textit{ such that } \delta'[\![X]\!] = \{0\}\\ \textit{any } \rho : \delta \leftrightarrow \delta' \textit{ such that } \rho[\![X]\!] = \{(v,0) \mid v \in \delta[\![X]\!]\}\\ \textit{any assignment } \eta \textit{ to } \Gamma \textit{ with respect to } \delta\\ \eta'[\![zer]\!] = 0\\ \eta'[\![add]\!] = \{((0,0),0)\}\\ \eta'[\![mult]\!] = \{((0,0),0)\}\end{array}$$

We can also observe that, since $[\![\mathit{Num}]\!]_{\rho,\delta,\delta'}$ is always an identity relation (see Proposition 4.1.1), (4.22) is always equivalent to the statement that, for all natural numbers r and s,

$$([\![\Gamma \vdash \mathit{MultMatr}(r,s)]\!]^M_{\delta,\eta})([\![\mathit{Proc}_{T^\dagger,\psi}]\!]^M_{\rho,\delta,\delta'})([\![\Gamma \vdash \mathit{MultMatr}(r,s)]\!]^M_{\delta',\eta'}) \tag{4.23}$$

In particular, this illustrates an application of Theorem 4.2.2 to $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ in the proof of Theorem 5.1.2. See also Example 5.1.1. ■

4.3 Notes

The literature on logical relations and their applications to theory and practice of programming languages is large: see e.g. [Mit90, OHe96, Plo80, Rey83, Wad89].

The contribution of this chapter was to define logical relations at process types of our language (see under 'Process types' in Section 4.1), and to show that a Basic Lemma holds for that definition (see Theorem 4.2.2).

The starting point for our work was the definition of data independence in [Wol86] (which seems to be the first formal study of data independence, having been preceded by case studies in e.g. [SWL85, Sab88]) and a suggestion by Oege de Moor and Bill Roscoe to investigate whether logical relations could be useful for reasoning about data independence in CSP. There also are connections with logical relations between possible worlds (see [Bro96]) and with abstraction (see e.g. [CC77, CGL94, LG+95, Gra94, DF95, Dam96, Kur90]).

The Basic Lemma we have obtained (i.e. Theorem 4.2.2) is of interest in itself. As we have remarked at the beginning of the present chapter, it will in addition be useful in Chapter 5. However, the whole of Theorem 4.2.2 will not be necessary for those applications: a considerably simpler result, which is provable by translating to and from symbolic operational semantics, would suffice. In other words, the work in the present chapter is on one hand of interest in itself, and on the other hand useful but not strictly necessary for the rest of this thesis.

A topic for future research is studying possible definitions of logical relations at process types which are interpreted by labelled transition systems rather than denotational models of CSP, and connections between such definitions and the definition we have presented in this chapter.

Another topic for future research is investigating to what extent the work in this chapter can be generalised, and to what extent it can be obtained from existing category-theoretic treatments of logical relations (for such treatments, see e.g. [OHe96, KO+97] and the references given there).

Chapter 5

Threshold collections

In this chapter, we address the problem which has been the main motivation for the work presented in the thesis. Namely, given two weakly data independent processes *Spec* and *Impl* with the same alphabet, how does whether *Spec* is refined by *Impl* change with varying the interpretation of the weakly data independent type and the associated constants and operations?

More precisely, our principal aim has been to obtain theorems which provide sufficient finite collections of interpretations of the weakly data independent type and the associated constants and operations, in the sense that if the refinement holds for those interpretations, then it holds for all interpretations, or for all interpretations which assign larger sets to the weakly data independent type. We call such sufficient finite collections of interpretations 'threshold collections'.

Additionally, we have been aiming to obtain theorems which state that, if *Spec* is not refined by *Impl* for some interpretation of the weakly data independent type and the associated constants and operations, then the refinement fails for all interpretations, or for all interpretations with some property. Thus, if the refinement fails for some interpretation from a threshold collection which is provided by a theorem of the former kind, a theorem of this latter kind may apply and give more information.

Both the general problem of how does whether *Spec* is refined by *Impl* change with varying the interpretation, and the aiming for the two particular kinds of theorems, have in turn been motivated by the needs of practical model checking: see Sections 2.1.4, 3.8 and 5.6.

Since weak data independence is quite a strong property (see Section 2.7), one may expect the general problem we are addressing to be straightforward. However, the first and the fourth of the following examples show that threshold collections which consist of interpretations which assign finite sets to the weakly data independent type may not exist, and the second and the third show that the required sizes of the sets may depend on the structures of *Spec* and *Impl*.

Each of the four examples will also be used later in this chapter. Some of their properties, which are not stated at this stage, will be important then.

Example 5.0.1 Consider

$$\begin{aligned}
Spec = {} & in\$x : X\$y : X \longrightarrow out!q(x)!q(y) \longrightarrow \\
& in'\$z : X \longrightarrow out'!z \longrightarrow STOP \\
Impl = {} & in?x : X?y : X \longrightarrow out!q(g(f(q(x), q(y))))!q(h(f(q(x), q(y)))) \longrightarrow \\
& in'?z : X \longrightarrow out'!f(q(g(z)), q(h(z))) \longrightarrow STOP
\end{aligned}$$

where q, g and h are variables of type $X \to X$, and f is a variable of type $(X \times X) \to X$.

We have that *Spec* is weakly data independent with respect to X with no constants and with the operation q, and that *Impl* is weakly data independent with respect to X with no constants and with the operations q, g, h and f.

A refinement

$$\mathit{Spec} \sqsubseteq^{M}_{\delta,\eta} \mathit{Impl}$$

(see Section 2.6) holds if and only if

$$\eta[\![f]\!] \upharpoonright (\mathit{Range}(\eta[\![q]\!]) \times \mathit{Range}(\eta[\![q]\!]))$$

and

$$(\eta[\![q]\!] \circ \eta[\![g]\!], \eta[\![q]\!] \circ \eta[\![h]\!])$$

are mutual inverses, which is possible if and only if $|\delta[\![X]\!]|$ is either a square of a positive integer or infinite. ■

Example 5.0.2 Let us fix an integer $K \geq 2$.

Then consider

$$\begin{aligned}
\mathit{Spec} = {} & \mathit{in}?x_1 : X \longrightarrow \cdots \longrightarrow \mathit{in}?x_K : X \longrightarrow \\
& \quad \textstyle\bigsqcap_{i,j \in \{1,\ldots,K\} \wedge i<j} \mathit{out}!x_1 \longrightarrow \cdots \longrightarrow \mathit{out}!x_{j-1} \longrightarrow \mathit{out}!x_i \longrightarrow \\
& \qquad\qquad \mathit{out}!x_{j+1} \longrightarrow \cdots \longrightarrow \mathit{out}!x_K \longrightarrow \mathit{STOP} \\
\mathit{Impl} = {} & \mathit{in}?x_1 : X \longrightarrow \cdots \longrightarrow \mathit{in}?x_K : X \longrightarrow \\
& \quad \mathit{out}!x_1 \longrightarrow \cdots \longrightarrow \mathit{out}!x_K \longrightarrow \mathit{STOP}
\end{aligned}$$

where $\bigsqcap_{i,j \in \{1,\ldots,K\} \wedge i<j}$ is an abbreviation for $K(K-1)/2-1$ binary nondeterministic choices.

We have that *Spec* and *Impl* are data independent with respect to X with no constants.

A refinement

$$\mathit{Spec} \sqsubseteq^{M}_{\delta} \mathit{Impl}$$

holds if and only if $|\delta[\![X]\!]| < K$. ■

Example 5.0.3 Let us fix $K_1, K_2, K_3 \in \mathbb{Z}^+$, and M which is one of Tr, Fa or FD.

Then consider

```
Spec = $c?x_1 : X \longrightarrow \cdots \longrightarrow c?x_{K_1} : X \longrightarrow$
          if $|\{x_1, \ldots, x_{K_1}\}| < K_1$
            then $\bot_M$
            else $c?y_1 : X \longrightarrow$
              if $y_1 \in \{x_1, \ldots, x_{K_1}\}$
                then $\bot_M$
                else
                  $\cdots$ $c?y_{K_2} \longrightarrow$
                        if $y_{K_2} \in \{x_1, \ldots, x_{K_1}\}$
                          then $\bot_M$
                          else $d?z_1 : X \ldots ?z_{K_3} : X \longrightarrow$
                            if $\{z_1, \ldots, z_{K_3}\} \cap \{x_1, \ldots, x_{K_1}\} \neq \{\}$
                              then $\bot_M$
                              else STOP
```

```
$Impl = c?x_1 : X \longrightarrow \cdots \longrightarrow c?x_{K_1} : X \longrightarrow$
          $c?y_1 : X \longrightarrow \cdots \longrightarrow c?y_{K_2} : X \longrightarrow$
            $d?z_1 : X \ldots ?z_{K_3} : X \longrightarrow$
              $if\ \ |\{y_1, \ldots, y_{K_2}, z_1, \ldots, z_{K_3}\}| < K_2 + K_3$
                $then\ STOP$
                $else\ e \longrightarrow STOP$
```

where, for expressing *Spec* and *Impl* in the language in this thesis, the conditions involving sets stand for collections of equality tests, and $\bot_M$ stands for a process whose interpretation is the least element with respect to the refinement ordering. We omit the details.

We have that *Spec* and *Impl* are data independent with respect to X with no constants.

A refinement

$$Spec \sqsubseteq_\delta^M Impl$$

holds if and only if $|\delta[\![X]\!]| < K_1 + K_2 + K_3$.

(This example was constructed by A.W. Roscoe.) ■

Example 5.0.4 Consider

```
$Spec = \sqcap_{x:X} Spec'(x)$
$Spec'(x) = c?u : X?v : X \longrightarrow$
            $if\ \ u = v$
              $then\ Impl$
              $else$
                $if\ \ (u = x)\ or\ (v = x)$
                  $then\ STOP$
                  $else\ \ (Spec''(x, u)$
                         $\sqcap\ Spec''(x, v))$
$Spec''(x, y) = c?u : X?v : X \longrightarrow$
                $if\ \ (u = v)\ or\ (u = y)\ or\ (v = y)$
                  $then\ Impl$
                  $else$
                    $if\ \ (u = x)\ or\ (v = x)$
                      $then\ STOP$
                      $else\ \ (Spec''(x, y)$
                             $\sqcap\ Spec''(x, u)$
                             $\sqcap\ Spec''(x, v))$
$Impl = c?u : X?v : X \longrightarrow Impl$
```

(Observe the similarities between *Spec* and the process *NondReg* in Example 2.5.1.)

We have that *Spec* and *Impl* are data independent with respect to X with no constants.

A refinement

$$Spec \sqsubseteq_\delta^M Impl$$

holds if and only if, for any trace

$$\langle (c, (w_1, w_2)), \ldots, (c, (w_{2K-1}, w_{2K})) \rangle$$

from $traces(\{\} \vdash Impl)_\delta$ such that $w_i \neq w_j$ whenever $i \neq j$, there exists $w' \in \delta[\![X]\!]$ such that $w_i \neq w'$ for all i, which is the case if and only if $|\delta[\![X]\!]|$ is either finite and odd, or infinite.

By extending the same idea, it is for example possible to construct *Spec* and *Impl* which are X-DI and such that

$$Spec \sqsubseteq_{\delta}^{M} Impl$$

holds if and only if $|\delta[\![X]\!]|$ is either finite and prime, or infinite.

(This example, as well as the extension we have just mentioned, was constructed by A.W. Roscoe.) ■

For some better-behaved examples involving data independent processes, see Examples 2.1.4 and 2.6.1.

An assumption about implementation processes. As in the statement above of the general problem we are addressing in this chapter, we shall be calling the processes we are working with *Spec* and *Impl*, and they shall be weakly data independent with respect to some X and Γ, and have the same alphabet T.

For simplicity, throughout the chapter, we shall be assuming that there exists a ways specifier χ such that

$$\Gamma \vdash Impl : Proc_{T,\chi}$$

and

$$\forall\, id_i, j.X \in Free(T_{i,j}) \Rightarrow \chi(id_i, j) \subseteq \{!, ?\}$$

where T is of the form $\sum_{i=1}^{n} id_i.\prod_{j=1}^{m_i} T_{i,j}$.

More informally, we shall be assuming that $\Gamma \vdash Impl$ cannot perform any nondeterministic selections of values of type X.

(See under 'Outputs, inputs and nondeterministic selections' in Section 2.5.2. As we have remarked there, nondeterministic selections are not a part of ordinary CSP [Hoa85, Ros98a], but were introduced into the language in this thesis primarily because of the condition ($\mathbf{RecCho}_X$) (see Section 3.5.2), which $\Gamma \vdash Impl$ will not be required to satisfy anywhere in the present chapter.) ■

The following proposition says that the assumption has a simple consequence on SLTSs and ASLTSs, namely that they do not involve any nondeterministic selections of values of type X either.

Proposition 5.0.1 *In any SLTS $\mathcal{S}_{\Gamma \vdash Impl}$ or any ASLTS $\mathcal{AS}_{\Gamma \vdash Impl}$ where $\Gamma \vdash Impl$ satisfies the assumption above, we have that, for any visible transition α, $ys(\alpha) = \{\}$.*

Proof. The proof is immediate (see under 'The SLTS of a given process' in Section 3.3.1, under 'The ASLTS of a given process' in Section 3.3.2, and under 'Visible' (III.c) and (III.d) in Section 3.2.2). ■

The first section of this chapter applies when *Spec* does not place any restrictions on values from the weakly data independent type, and *Impl* is weakly data independent with restricted equality tests. It provides a threshold collection consisting of a single interpretation which assigns a set of size 1 to the weakly data independent type.

In the second, third and fourth sections, *Spec* and *Impl* are required to be data independent.

- The second section applies when *Spec* either has restricted equality tests or satisfies (**RecCho**$_X$) (see Section 3.5.2). It enables one to deduce that, if *Spec* is not refined by *Impl* for some interpretation, then the refinement fails for all interpretations with an appropriate property.
- The third section applies when *Spec* has restricted equality tests and restricted nondeterminism, and *Impl* has restricted equality tests. It provides threshold collections which consist of interpretations with a set of size 2.
- The fourth section applies when *Spec* satisfies (**RecCho**$_X$). It provides threshold collections which consist of interpretations with a set whose size depends on the structures of *Spec* and *Impl*.

The fifth section mentions some further developments which we could not present in detail in this thesis, and the sixth section discusses implications for tools.

5.1 Set of size 1

In this section, according to the general problem we are considering (see the beginning of the present chapter), we shall be requiring *Spec* and *Impl* to be weakly data independent with respect to some X and Γ, and to have the same alphabet T.

The section contains two theorems. The first theorem requires that *Spec* does not place any restrictions on values of type X and that *Impl* has restricted equality tests between values of type X, and it provides a threshold collection consisting of a single interpretation which assigns a set of size 1 to X.

The requirement that *Spec* does not place any restrictions on values of type X more precisely amounts to requiring that:

- *Spec* does not involve any equality tests between values of type X;
- for the finite-traces model, *Spec* does not involve any outputs of values of type X;
- for the stable-failures and failures/divergences models, *Spec* does not involve any outputs or inputs of values of type X (but it may involve nondeterministic selections of values of type X).

The second theorem requires that *Spec* and *Impl* have restricted equality tests between values of type X, and it states that, if *Spec* is not refined by *Impl* for the interpretation which is provided by the first theorem, then the refinement fails for all interpretations.

Both theorems are proved by the Basic Lemma of Logical Relations (i.e. Theorem 4.2.2), except that an additional lemma is used in the proof of the first theorem.

We shall first define the threshold interpretation, then present the lemma, and then present the two theorems. Finally, we shall give three examples.

5.1.1 The threshold interpretation

Definition 5.1.1 Suppose that X is a type variable from *NEFTypeVars* and that Γ is a type context which satisfies the restrictions of weak data independence with respect to X (see Section 2.7.2).

- Let δ_0 be any assignment of partial orders to type variables (subject to the restrictions stated under 'Type variables' in Section 2.4) such that

$$\delta_0[\![X]\!] = \{0\}$$

- Let η_0 be the assignment to Γ with respect to δ_0 defined by

$$\begin{array}{l} \forall (x : X) \in \Gamma . \eta_0[\![x]\!] = 0 \\ \forall (y : U \to X) \in \Gamma . \forall u \in [\![U]\!]_{\delta_0} . (\eta_0[\![y]\!])(u) = 0 \end{array}$$

(Observe that η_0 is the unique assignment to Γ with respect to δ_0.) ■

5.1.2 The lemma

The lemma says that, if (t, A) is any failure of a process P which is weakly data independent with respect to some X and Γ, and if B is a set of visible events which contains A and which is such that, for any b which is in B but not in A, the set

$$\{b^\dagger \in B \mid b^\dagger \textit{ differs from } b \textit{ at most in values of type } X\}$$

is not exhaustive in the channel components which P can use only for nondeterministic selections, then (t, B) is also a failure of P.

Lemma 5.1.1 *Suppose that:*

(i) $\Gamma \vdash P : \mathit{Proc}_{T,\phi}$ *is* X,Γ-*WDI, where* T *is of the form* $\sum_{i=1}^{n} id_i . \prod_{j=1}^{m_i} T_{i,j}$*;*

(ii) $(t, A) \in \mathit{failures}(\Gamma \vdash P)_{\delta,\eta}$ *for some* δ *and* η*;*

(iii) $B \subseteq [\![T]\!]_\delta$ *is such that* $A \subseteq B$ *and*

$$\begin{array}{l} \forall i, v . (id_i, v) \in (B \setminus A) \Rightarrow \exists w \in [\![\prod_{j=1}^{m_i} T_{i,j}]\!]_\delta . v [\![\prod_{j=1}^{m_i} T_{i,j}]\!]_{\delta \times \delta,\delta,\delta} w \wedge \\ \forall v^\dagger . ((id_i, v^\dagger) \in B \wedge v [\![\prod_{j=1}^{m_i} T_{i,j}]\!]_{\delta \times \delta,\delta,\delta} v^\dagger) \Rightarrow \exists j . \phi(id_i, j) \subseteq \{\$\} \wedge w(j) \neq v^\dagger(j) \end{array}$$

Then $(t, B) \in \mathit{failures}(\Gamma \vdash P)_{\delta,\eta}$.

Proof. By translating to and from the SLTS $\mathcal{S}_{\Gamma \vdash P}$. (See Sections 3.3.1, 3.2.2 and 3.4.) ■

5.1.3 The theorems

Theorem 5.1.2 *Suppose that:*

(i) $\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$ *and* $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ *are* X,Γ-*WDI, where* T *is of the form* $\sum_{i=1}^{n} id_i . \prod_{j=1}^{m_i} T_{i,j}$*;*

(ii) M *specifies one of the models Tr, Fa, FD of CSP;*

(iii) $\Gamma \vdash \mathit{Spec}$ *satisfies* $(\mathbf{NoEqT}_X)$*;*

(iv) if $M = \mathrm{Tr}$*, then*

$$\forall id_i, j . X \in \mathit{Free}(T_{i,j}) \Rightarrow \phi(id_i, j) \subseteq \{?, \$\}$$

(v) if $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$*, then*

$$\forall\, id_i, j.X \in \mathit{Free}(T_{i,j}) \Rightarrow \phi(id_i, j) \subseteq \{\$\}$$

(vi) if $M = \mathrm{Tr}$*, then* $\Gamma \vdash \mathit{Impl}$ *satisfies* $(\mathbf{PosConjEqT}_X)$*;*

(vii) if $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$*, then* $\Gamma \vdash \mathit{Impl}$ *satisfies* $(\mathbf{NoEqT}_X)$*.*

If $\mathit{Spec} \sqsubseteq^{M}_{\delta_0,\eta_0} \mathit{Impl}$*, then* $\mathit{Spec} \sqsubseteq^{M}_{\delta,\eta} \mathit{Impl}$ *for all* δ *and* η *such that, for all* $Y \neq X$*,* $\delta[\![Y]\!]$ *is nonempty if and only if* $\delta_0[\![Y]\!]$ *is.*[1]

Proof. Let us consider the case $M = \mathrm{FD}$.

Suppose $\mathit{Spec} \sqsubseteq^{\mathrm{FD}}_{\delta_0,\eta_0} \mathit{Impl}$, and suppose δ and η are such that, for all $Y \neq X$, $\delta[\![Y]\!]$ is nonempty if and only if $\delta_0[\![Y]\!]$ is.

Let $\rho_0 : \delta \leftrightarrow \delta_0$ be such that

- $\rho_0[\![X]\!]$ is the unique total function from $\delta[\![X]\!]$ to $\delta_0[\![X]\!]$, and
- $\rho_0[\![Y]\!]$ is total and surjective for all $Y \neq X$.

Then we have that:

$$\begin{aligned}&\forall (x : X) \in \Gamma.(\eta[\![x]\!])([\![X]\!]_{\rho_0,\delta,\delta_0})(\eta_0[\![x]\!])\\ &\forall (y : U \to X) \in \Gamma.(\eta[\![y]\!])([\![U \to X]\!]_{\rho_0,\delta,\delta_0})(\eta_0[\![y]\!])\end{aligned}$$

and so Proposition 4.1.2 and Theorem 4.2.2 give us that:

$$([\![\Gamma \vdash \mathit{Spec}]\!]^{\mathrm{FD}}_{\delta_0,\eta_0}) \quad ([\![\mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{(\rho_0)^{-1},\delta_0,\delta}) \quad ([\![\Gamma \vdash \mathit{Spec}]\!]^{\mathrm{FD}}_{\delta,\eta}) \tag{5.1}$$

$$([\![\Gamma \vdash \mathit{Impl}]\!]^{\mathrm{FD}}_{\delta,\eta}) \quad ([\![\mathit{Proc}_{T,\psi}]\!]^{\mathrm{FD}}_{\rho_0,\delta,\delta_0}) \quad ([\![\mathit{Impl}]\!]^{\mathrm{FD}}_{\delta_0,\eta_0}) \tag{5.2}$$

Suppose $(t, A) \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Impl})_{\delta,\eta}$. By (5.2), there exists

$$(t', A') \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Impl})_{\delta_0,\eta_0}$$

such that:

$$t([\![T]\!]_{\rho_0,\delta,\delta_0})^{*} t' \tag{5.3}$$

$$\forall\, a' \in ([\![T]\!]_{\delta_0} \setminus A').\, \exists\, a \in ([\![T]\!]_{\delta} \setminus A).a([\![T]\!]_{\rho_0,\delta,\delta_0})a' \tag{5.4}$$

Since $\mathit{Spec} \sqsubseteq^{\mathrm{FD}}_{\delta_0,\eta_0} \mathit{Impl}$, we have that $(t', A') \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Spec})_{\delta_0,\eta_0}$.

Now, by (5.1), we have that

$$\begin{aligned}&\exists\, J \subseteq \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}.\\ &J \neq \{\} \wedge \mathit{Related}^{\mathrm{Fa}}(T, (\rho_0)^{-1}, \delta_0, \delta, (t', A'), J) \wedge \mathit{Closed}^{\mathrm{Fa}}(T, \phi, (\rho_0)^{-1}, \delta_0, \delta, (t', A'), J)\end{aligned}$$

and, by Proposition 4.1.2, we have that $t'([\![T]\!]_{(\rho_0)^{-1},\delta_0,\delta})^{*} t$. By assumption (v), it then follows that there exists $(t^{\dagger}, A^{\dagger}) \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$ such that $t^{\dagger} = t$ and

$$\forall\, a^{\dagger} \in ([\![T]\!]_{\delta} \setminus A^{\dagger}).\, \exists\, a' \in ([\![T]\!]_{\delta_0} \setminus A').a'([\![T]\!]_{(\rho_0)^{-1},\delta_0,\delta})a^{\dagger} \tag{5.5}$$

If $t \in \mathit{divergences}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$, then $(t, A) \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$ is immediate.

[1]The requirement for $Y \neq X$ is present only because the statement of Theorem 4.2.2 involves requirements for all type variables.

Otherwise, we have that $(t, A^\dagger) \in \mathit{failures}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$. Let

$$A^\ddagger = \{a \in A \mid \forall\, b \in [\![T]\!]_\delta . a([\![T]\!]_{\delta\times\delta,\delta,\delta})b \Rightarrow b \in A\}$$

By (5.5), (5.4) and Proposition 4.1.5, it follows that $A^\ddagger \subseteq A^\dagger$, and so

$$(t, A^\ddagger) \in \mathit{failures}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$$

But then, by the definition of $A^\ddagger$ and by assumption (v), we can apply Lemma 5.1.1 to $(t, A^\ddagger)$ and A, which gives us that $(t, A) \in \mathit{failures}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$.

Therefore, $\mathit{failures}_\perp(\Gamma \vdash \mathit{Impl})_{\delta,\eta} \subseteq \mathit{failures}_\perp(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$.

Showing the other half of

$$\mathit{Spec} \sqsubseteq^{\mathrm{FD}}_{\delta,\eta} \mathit{Impl}$$

namely that $\mathit{divergences}(\Gamma \vdash \mathit{Impl})_{\delta,\eta} \subseteq \mathit{divergences}(\Gamma \vdash \mathit{Spec})_{\delta,\eta}$, is simpler.

That completes the case $M = \mathrm{FD}$. The cases $M = \mathrm{Tr}$ and $M = \mathrm{Fa}$ are simpler. ■

Theorem 5.1.3 *Suppose that:*

(i) $\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$ *and* $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ *are X,Γ-WDI;*

(ii) M specifies one of the models Tr, Fa, FD of CSP;

(iii) if $M = \mathrm{Tr}$*, then* $\Gamma \vdash \mathit{Spec}$ *satisfies* (**PosConjEqT**$_X$)*;*

(iv) if $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$*, then* $\Gamma \vdash \mathit{Spec}$ *satisfies* (**NoEqT**$_X$)*;*

(v) $\Gamma \vdash \mathit{Impl}$ *satisfies* (**NoEqT**$_X$)*.*

If $\mathit{Spec} \not\sqsubseteq^M_{\delta_0,\eta_0} \mathit{Impl}$*, then* $\mathit{Spec} \not\sqsubseteq^M_{\delta,\eta} \mathit{Impl}$ *for all* δ *and* η *such that, for all* $Y \neq X$*,* $\delta[\![Y]\!]$ *is nonempty if and only if* $\delta_0[\![Y]\!]$ *is.*[2]

Proof. Let us consider the case $M = \mathrm{FD}$.

Suppose $\mathit{Spec} \sqsubseteq^{\mathrm{FD}}_{\delta,\eta} \mathit{Impl}$ for some δ and η such that, for all $Y \neq X$, $\delta[\![Y]\!]$ is nonempty if and only if $\delta_0[\![Y]\!]$ is.

Let $\rho_0 : \delta \leftrightarrow \delta_0$ be as in the proof of Theorem 5.1.2. Then, in the same way as (5.1) and (5.2) were obtained in the proof of Theorem 5.1.2, we have that:

$$([\![\Gamma \vdash \mathit{Spec}]\!]^{\mathrm{FD}}_{\delta,\eta}) \quad ([\![\mathit{Proc}_{T,\psi}]\!]^{\mathrm{FD}}_{\rho_0,\delta,\delta_0}) \quad ([\![\mathit{Spec}]\!]^{\mathrm{FD}}_{\delta_0,\eta_0}) \tag{5.6}$$

$$([\![\Gamma \vdash \mathit{Impl}]\!]^{\mathrm{FD}}_{\delta_0,\eta_0}) \quad ([\![\mathit{Proc}_{T,\phi}]\!]^{\mathrm{FD}}_{(\rho_0)^{-1},\delta_0,\delta}) \quad ([\![\Gamma \vdash \mathit{Impl}]\!]^{\mathrm{FD}}_{\delta,\eta}) \tag{5.7}$$

Suppose $(t, A) \in \mathit{failures}_\perp(\Gamma \vdash \mathit{Impl})_{\delta_0,\eta_0}$. By (5.7), there exists

$$(t', A') \in \mathit{failures}_\perp(\Gamma \vdash \mathit{Impl})_{\delta,\eta}$$

such that:

$$t([\![T]\!]_{(\rho_0)^{-1},\delta_0,\delta})^* t' \tag{5.8}$$

[2]As in Theorem 5.1.2, the requirement for $Y \neq X$ is present only because the statement of Theorem 4.2.2 involves requirements for all type variables.

$$\forall\, a' \in ([\![T]\!]_\delta \setminus A').\, \exists\, a \in ([\![T]\!]_{\delta_0} \setminus A).a([\![T]\!]_{(\rho_0)^{-1},\delta_0,\delta})a' \tag{5.9}$$

Since $Spec \sqsubseteq^{\mathrm{FD}}_{\delta,\eta} Impl$, we have that $(t', A') \in failures_\perp(\Gamma \vdash Spec)_{\delta,\eta}$.

Now, by (5.6), there exists $(t^\dagger, A^\dagger) \in failures_\perp(\Gamma \vdash Spec)_{\delta_0,\eta_0}$ such that:

$$t'([\![T]\!]_{\rho_0,\delta,\delta_0})^* t^\dagger \tag{5.10}$$

$$\forall\, a^\dagger \in ([\![T]\!]_{\delta_0} \setminus A^\dagger).\, \exists\, a' \in ([\![T]\!]_\delta \setminus A').a'([\![T]\!]_{\rho_0,\delta,\delta_0})a^\dagger \tag{5.11}$$

By (5.8), (5.10), Proposition 4.1.5, the fact that $(\delta_0 \times \delta_0)[\![X]\!]$ is the identity relation, and Proposition 4.1.1, it follows that $t = t^\dagger$. Secondly, by (5.11), (5.9), Proposition 4.1.5, the fact that $(\delta_0 \times \delta_0)[\![X]\!]$ is the identity relation, and Proposition 4.1.1, it follows that $A \subseteq A^\dagger$. Hence $(t, A) \in failures_\perp(\Gamma \vdash Spec)_{\delta_0,\eta_0}$.

Therefore, $failures_\perp(\Gamma \vdash Impl)_{\delta_0,\eta_0} \subseteq failures_\perp(\Gamma \vdash Spec)_{\delta_0,\eta_0}$.

Showing the other half of

$$Spec \sqsubseteq^{\mathrm{FD}}_{\delta_0,\eta_0} Impl$$

namely that $divergences(\Gamma \vdash Impl)_{\delta_0,\eta_0} \subseteq divergences(\Gamma \vdash Spec)_{\delta_0,\eta_0}$, is simpler.

That completes the case $M = \mathrm{FD}$. The cases $M = \mathrm{Tr}$ and $M = \mathrm{Fa}$ are simpler. ■

The requirements on $\Gamma \vdash Spec : Proc_{T,\phi}$ in Theorem 5.1.2 are quite restrictive. However, they are concerned only with values of type X, and there are many important specification processes which satisfy them.

For example, the requirements are satisfied by the specification of deadlock freedom (for $M = \mathrm{Fa}$ or $M = \mathrm{FD}$) and the specification of divergence freedom (for $M = \mathrm{FD}$), when they are written in the following ways:

$$\begin{aligned} DF_T = {} & (id_1 \$ x_{1,1} : T_{1,1} \ldots \$ x_{1,m_1} : T_{1,m_1} \longrightarrow DF_T) \\ & \sqcap \cdots \sqcap \\ & (id_n \$ x_{n,1} : T_{n,1} \ldots \$ x_{n,m_n} : T_{n,m_n} \longrightarrow DF_T) \end{aligned}$$

$$\begin{aligned} Chaos_T = {} & (id_1 \$ x_{1,1} : T_{1,1} \ldots \$ x_{1,m_1} : T_{1,m_1} \longrightarrow Chaos_T) \\ & \sqcap \cdots \sqcap \\ & (id_n \$ x_{n,1} : T_{n,1} \ldots \$ x_{n,m_n} : T_{n,m_n} \longrightarrow Chaos_T) \\ & \sqcap\, STOP \end{aligned}$$

5.1.4 Some examples

Example 5.1.1 Consider the process

$$\Gamma \vdash MultMatr : (Num \times Num) \to Proc_{T^\dagger,\psi}$$

from Examples 2.7.4 and 4.2.2, and consider the following process, which in our language can be written as a term

$$\Gamma \vdash Cnt : (Num \times Num) \to Proc_{T^\dagger,\phi}$$

where

$\phi(hor, 1) = \{!\}$	$\phi(hor, 2) = \{!\}$	$\phi(hor, 3) = \{?\}$
$\phi(vert, 1) = \{!\}$	$\phi(vert, 2) = \{!\}$	$\phi(vert, 3) = \{?\}$
$\phi(finished, 1) = \{!\}$	$\phi(finished, 2) = \{!\}$	
$\phi(out, 1) = \{!\}$	$\phi(out, 2) = \{!\}$	$\phi(out, 3) = \{?\}$

$$
\begin{array}{l}
\mathit{Cnt}(g,h) = \mathit{Cnt}'(g,h,\lambda\, i : \mathit{Num}.0, \lambda\, j : \mathit{Num}.0, \lambda\, i : \mathit{Num}.0, \lambda\, j : \mathit{Num}.0) \\
\mathit{Cnt}'(g,h,\mathit{hi},\mathit{vi},\mathit{ho},\mathit{vo}) = \\
\quad (\Box_{i:\{0,\ldots,g-1\}}\ \mathit{hor}!i!0?y \longrightarrow \mathit{Cnt}'(g,h,\mathit{hi}[(\mathit{hi}(i)+1)/i],\mathit{vi},\mathit{ho},\mathit{vo})) \\
\quad \Box\ (\Box_{j:\{0,\ldots,h-1\}}\ \mathit{vert}!0!j?y' \longrightarrow \mathit{Cnt}'(g,h,\mathit{hi},\mathit{vi}[(\mathit{vi}(j)+1)/j],\mathit{ho},\mathit{vo})) \\
\quad \Box\ (\Box_{i:\{0,\ldots,g-1\}}\ \mathit{if}\ \ \mathit{ho}(i) < \mathit{hi}(i) \wedge \mathit{ho}(i) < \mathit{vi}(0) \wedge \cdots \wedge \mathit{ho}(i) < \mathit{vi}(h-1) \\
\qquad\qquad\qquad\quad \mathit{then}\ \mathit{hor}!i!h?y \longrightarrow \mathit{Cnt}'(g,h,\mathit{hi},\mathit{vi},\mathit{ho}[(\mathit{ho}(i)+1)/i],\mathit{vo}) \\
\qquad\qquad\qquad\quad \mathit{else}\ \mathit{STOP}) \\
\quad \Box\ (\Box_{j:\{0,\ldots,h-1\}}\ \mathit{if}\ \ \mathit{vo}(j) < \mathit{vi}(j) \wedge \mathit{vo}(j) < \mathit{hi}(0) \wedge \cdots \wedge \mathit{vo}(j) < \mathit{hi}(g-1) \\
\qquad\qquad\qquad\quad \mathit{then}\ \mathit{hor}!g!j?y' \longrightarrow \mathit{Cnt}'(g,h,\mathit{hi},\mathit{vi},\mathit{ho},\mathit{vo}[(\mathit{vo}(j)+1)/j]) \\
\qquad\qquad\qquad\quad \mathit{else}\ \mathit{STOP}) \\
\quad \Box\ (\Box_{i:\{0,\ldots,g-1\}}\ \Box_{j:\{0,\ldots,h-1\}}\ \mathit{finished}!i!j \longrightarrow \mathit{Cnt}'(g,h,\mathit{hi},\mathit{vi},\mathit{ho},\mathit{vo})) \\
\quad \Box\ (\Box_{i:\{0,\ldots,g-1\}}\ \Box_{j:\{0,\ldots,h-1\}}\ \mathit{out}!i!j?z \longrightarrow \mathit{Cnt}'(g,h,\mathit{hi},\mathit{vi},\mathit{ho},\mathit{vo}))
\end{array}
$$

where $f[u/k]$ is an abbreviation for $\lambda\, k' : \mathit{Num}.(k' = k) \rightsquigarrow u, f(k')$.

Suppose r and s are two positive integers. Then the assumptions of Theorem 5.1.2 are satisfied with

$$
\begin{array}{rcl}
\mathit{Spec} & = & \mathit{Cnt}(r,s) \\
\mathit{Impl} & = & \mathit{MultMatr}(r,s) \\
T & = & T^{\dagger} \\
M & = & \mathrm{Tr}
\end{array}
$$

so we have that if

$$\mathit{Cnt}(r,s) \sqsubseteq^{\mathrm{Tr}}_{\delta_0,\eta_0} \mathit{MultMatr}(r,s)$$

then

$$\mathit{Cnt}(r,s) \sqsubseteq^{\mathrm{Tr}}_{\delta,\eta} \mathit{MultMatr}(r,s)$$

for all δ and η such that, for all $Y \neq X$, $\delta[\![Y]\!]$ is nonempty if and only if $\delta_0[\![Y]\!]$ is.

(In connection with this application of Theorem 5.1.2, see Example 4.2.2.)

More informally, as a specification for the systolic array $\mathit{MultMatr}(r,s)$ in the finite-traces model, $\mathit{Cnt}(r,s)$ specifies that:

- any event along the channels *hor* or *vert* has indices which correspond to the edges of the $r \times s$ array;
- for any row, the number of past events in that row at the right-hand edge is always no more than the number of past events at the left-hand edge in the same row, and it is no more than the number of past events at the top edge, in each column separately;
- for any column, the number of past events in that column at the bottom edge is always no more than the number of past events at the top edge in the same column, and it is no more than the number of past events at the left-hand edge, in each row separately;
- any event along the channels *finished* or *out* has indices which correspond to one of the components of the $r \times s$ array;

and Theorem 5.1.2 gives us that, if the minimal instance of $\mathit{MultMatr}(r, s)$ (i.e. when the type variable X is assigned a singleton set) satisfies that specification, then any instance of $\mathit{MultMatr}(r, s)$ also does.

(For the reason why $\Gamma \vdash \mathit{MultMatr}(r, s)$ does not satisfy $(\mathbf{NoEqT}_X)$, which causes Theorem 5.1.2 not to be applicable with $M = \text{Fa}$ and $M = \text{FD}$, see under 'More precise ways specifiers' in Section 2.5.2.) ■

Example 5.1.2 Recall the processes *Spec* and *Impl* from Example 5.0.1.

In our language, they can be written as terms

$$\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$$
$$\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$$

where

$$\Gamma = \{q : X \to X, f : (X \times X) \to X, g : X \to X, h : X \to X\}$$

$$T = \mathit{in}.(X \times X) + \mathit{out}.(X \times X) + \mathit{in}'.X + \mathit{out}'.X$$

$$\phi(\mathit{in}, 1) = \phi(\mathit{in}, 2) = \phi(\mathit{in}', 1) = \{\$\}$$
$$\phi(\mathit{out}, 1) = \phi(\mathit{out}, 2) = \phi(\mathit{out}', 1) = \{!\}$$

$$\psi(\mathit{in}, 1) = \psi(\mathit{in}, 2) = \psi(\mathit{in}', 1) = \{?\}$$
$$\psi(\mathit{out}, 1) = \psi(\mathit{out}, 2) = \psi(\mathit{out}', 1) = \{!\}$$

As we have observed in Example 5.0.1, $\Gamma \vdash \mathit{Spec}$ and $\Gamma \vdash \mathit{Impl}$ are X,Γ-WDI, and we have that

$$\mathit{Spec} \sqsubseteq^M_{\delta,\eta} \mathit{Impl}$$

if and only if

$$\eta[\![f]\!] \restriction (\mathit{Range}(\eta[\![q]\!]) \times \mathit{Range}(\eta[\![q]\!]))$$

and

$$(\eta[\![q]\!] \circ \eta[\![g]\!], \eta[\![q]\!] \circ \eta[\![h]\!])$$

are mutual inverses, which is possible if and only if $|\delta[\![X]\!]|$ is either a square of a positive integer or infinite. In particular,

$$\mathit{Spec} \not\sqsubseteq^M_{\delta_0,\eta_0} \mathit{Impl}$$

for any M.

Since $\Gamma \vdash \mathit{Spec}$ and $\Gamma \vdash \mathit{Impl}$ both satisfy $(\mathbf{NoEqT}_X)$, and since the conclusion of Theorem 5.1.2 is not true for $\Gamma \vdash \mathit{Spec}$ and $\Gamma \vdash \mathit{Impl}$, this example shows that assumption (iv) of Theorem 5.1.2 cannot be omitted, and that the set $\{\$\}$ in assumption (v) cannot be replaced by the set $\{!, \$\}$. In other words, the example shows that the requirement that *Spec* does not involve outputs of values of type X cannot be omitted from Theorem 5.1.2.

On the other hand, the assumptions of Theorem 5.1.3 are satisfied, but since

$$\mathit{Spec} \not\sqsubseteq^M_{\delta_0,\eta_0} \mathit{Impl}$$

for any M, its conclusion is vacuous. ■

Example 5.1.3 Consider

$$\begin{aligned}&\mathit{Spec} = c\$x : X \longrightarrow c\$x : X \longrightarrow d \longrightarrow \mathit{STOP}\\ &\mathit{Impl} = c?x : X \longrightarrow c?y : X \longrightarrow (x = y \leadsto d \longrightarrow \mathit{STOP},\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{STOP})\end{aligned}$$

In our language, *Spec* and *Impl* can be written as terms

$$\begin{aligned}&\{\} \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}\\ &\{\} \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}\end{aligned}$$

where

$$T = c.X + d.1$$

$$\phi(c, 1) = \{\$\}$$

$$\psi(c, 1) = \{?\}$$

Since

- $\{\} \vdash \mathit{Spec}$ and $\{\} \vdash \mathit{Impl}$ are X,$\{\}$-WDI;
- $\{\} \vdash \mathit{Spec}$ satisfies ($\mathbf{NoEqT}_X$);
- $\{\} \vdash \mathit{Impl}$ satisfies ($\mathbf{PosConjEqT}_X$), but does not satisfy ($\mathbf{NoEqT}_X$);
- if $M = \mathrm{Fa}$ or $M = \mathrm{FD}$, then $\mathit{Spec} \sqsubseteq_\delta^M \mathit{Impl}$ if and only if $|\delta[\![X]\!]| = 1$;

this example shows that ($\mathbf{NoEqT}_X$) in assumption (vii) of Theorem 5.1.2 cannot be replaced by the weaker condition ($\mathbf{PosConjEqT}_X$). ∎

5.2 Deducing failure

In this and the following two sections, we shall be requiring *Spec* and *Impl* to be not only weakly data independent, but data independent (see Section 2.7.1). More precisely, *Spec* and *Impl* will be required to be data independent with respect to some X and Γ, and to have the same alphabet T.

In the same sections, we shall for simplicity be focussing on refinements of the form

$$\mathit{Spec} \ \sqsubseteq_{[X\mapsto\kappa],\eta}^M ? \ \mathit{Impl}$$

where κ is a nonzero cardinal (see under 'Some notation for assignments to type variables' in Section 2.4). For justification, see Corollary 4.2.3.

In this section, we shall present a theorem which states that, if *Spec* is not refined by *Impl* for some κ and η, then the refinement fails for all κ' which are greater than or equal to κ and all η' which induce the same equality or inequality relationships among the constants in Γ as η does. For the finite-traces model, the theorem requires that *Spec* satisfies either ($\mathbf{PosConjEqT}_X$) or ($\mathbf{RecCho}_X$). For the stable-failures and failures/divergences models, the theorem requires either that *Spec* satisfies ($\mathbf{NoEqT}_X$), or that *Spec* satisfies ($\mathbf{RecCho}_X$) and that $\kappa \geq 2$.

Before the theorem, we shall present a lemma which, together with Theorem 4.2.2, facilitates its proof. After the theorem, we shall give two examples.

5.2.1 The lemma

Lemma 5.2.1 *Suppose that:*

- $\Gamma \vdash P : Proc_{T,\phi}$ *is* X,Γ*-WDI and satisfies* $(\mathbf{RecCho}_X)$*;*
- $\rho : \delta \leftrightarrow \delta'$*;*
- η *and* η' *are assignments to* Γ *with respect to* δ *and* δ' *(respectively);*
- $\forall (x : U) \in \Gamma . (\eta[\![x]\!])([\![U]\!]_{\rho,\delta,\delta'})(\eta'[\![x]\!])$.

Then:

(i) if $\rho[\![X]\!]$ *is a total function and surjective, and if* $|\delta'[\![X]\!]| \geq 2$*, we have that*

$$\begin{array}{l}\forall (t, A) \in \mathit{failures}(\Gamma \vdash P)_{\delta,\eta} . \\ (\exists\, D \subseteq \delta[\![X]\!] . ((\rho[\![X]\!] \restriction D) \text{ is injective}) \wedge (t \in \mathit{Domain}(([\![T]\!]_{\rho[(\rho[\![X]\!] \restriction D)/X],\delta,\delta'})^*))) \Rightarrow \\ \exists (t', A') \in \mathit{failures}(\Gamma \vdash P)_{\delta',\eta'} . \\ t([\![T]\!]_{\rho,\delta,\delta'})^* t' \wedge \forall\, a' \in ([\![T]\!]_{\delta'} \setminus A') . \exists\, a \in ([\![T]\!]_{\delta} \setminus A) . a([\![T]\!]_{\rho,\delta,\delta'}) a'\end{array}$$

(ii) if $\rho[\![X]\!]$ *is a total function and surjective, and if* $|\delta'[\![X]\!]| \geq 2$*, we have that*

$$\begin{array}{l}\forall (t, A) \in \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta,\eta} . \\ (\exists\, D \subseteq \delta[\![X]\!] . ((\rho[\![X]\!] \restriction D) \text{ is injective}) \wedge (t \in \mathit{Domain}(([\![T]\!]_{\rho[(\rho[\![X]\!] \restriction D)/X],\delta,\delta'})^*))) \Rightarrow \\ \exists (t', A') \in \mathit{failures}_{\perp}(\Gamma \vdash P)_{\delta',\eta'} . \\ t([\![T]\!]_{\rho,\delta,\delta'})^* t' \wedge \forall\, a' \in ([\![T]\!]_{\delta'} \setminus A') . \exists\, a \in ([\![T]\!]_{\delta} \setminus A) . a([\![T]\!]_{\rho,\delta,\delta'}) a'\end{array}$$

(iii) if $\rho[\![X]\!]$ *is a partial function and injective, we have that*

$$\begin{array}{l}\forall\, t \in \mathit{divergences}(\Gamma \vdash P)_{\delta,\eta} . t \in \mathit{Domain}(([\![T]\!]_{\rho,\delta,\delta'})^*) \Rightarrow \\ \exists\, t' \in \mathit{divergences}(\Gamma \vdash P)_{\delta',\eta'} . t([\![T]\!]_{\rho,\delta,\delta'})^* t'\end{array}$$

Proof. (i) and (iii) are proved by translating to and from the SLTS $\mathcal{S}_{\Gamma \vdash P}$ with the help of Theorem 4.2.1. (See Sections 3.3.1, 3.2.2 and 3.4, and see under 'Assumption of transformation' after Proposition 3.5.3.)

(ii) follows straightforwardly from (i) and (iii). ■

5.2.2 The theorem

Theorem 5.2.2 *Suppose that:*

(i) $\Gamma \vdash \mathit{Spec} : Proc_{T,\phi}$ *and* $\Gamma \vdash \mathit{Impl} : Proc_{T,\psi}$ *are* X,Γ*-DI;*

(ii) M *specifies one of the models Tr, Fa, FD of CSP;*

(iii) κ *is a nonzero cardinal;*

(iv) if $M = \mathrm{Tr}$*, then* $\Gamma \vdash \mathit{Spec}$ *satisfies either* $(\mathbf{PosConjEqT}_X)$ *or* $(\mathbf{RecCho}_X)$*;*

(v) if $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$*, then either* $\Gamma \vdash \mathit{Spec}$ *satisfies* $(\mathbf{NoEqT}_X)$*, or* $\Gamma \vdash \mathit{Spec}$ *satisfies* $(\mathbf{RecCho}_X)$ *and* $\kappa \geq 2$*;*

(vi) η is an assignment to Γ with respect to κ.

If Spec $\not\sqsubseteq^{M}_{[X\mapsto\kappa],\eta}$ Impl, then Spec $\not\sqsubseteq^{M}_{[X\mapsto\kappa'],\eta'}$ Impl for all cardinals κ' such that $\kappa' \geq \kappa$ and all η' such that

$$\forall (x : X), (y : X) \in \Gamma.(\eta[\![x]\!] = \eta[\![y]\!]) \Leftrightarrow (\eta'[\![x]\!] = \eta'[\![y]\!])$$

(If $X \in$ FinTypeVars, which may be required when $M = \mathrm{FD}$ (see under 'Restrictions with FD' in Section 2.5.2), then it is implicitly assumed that κ and κ' are natural numbers (see Section 2.4).)

Proof. Let us consider the case when $M = \mathrm{FD}$, $\Gamma \vdash$ *Spec* satisfies ($\mathbf{RecCho}_X$) and $\kappa \geq 2$.

Suppose *Spec* $\sqsubseteq^{\mathrm{FD}}_{[X\mapsto\kappa'],\eta'}$ *Impl* for some cardinal κ' such that $\kappa' \geq \kappa$ and some η' such that

$$\forall (x : X), (y : X) \in \Gamma.(\eta[\![x]\!] = \eta[\![y]\!]) \Leftrightarrow (\eta'[\![x]\!] = \eta'[\![y]\!]) \tag{5.12}$$

By (5.12), we can let $\rho : [X \mapsto \kappa] \leftrightarrow [X \mapsto \kappa']$ be such that:

- $\rho[\![X]\!]$ is a total function, injective and

$$\forall (x : X) \in \Gamma.(\eta[\![x]\!])(\rho[\![X]\!])(\eta'[\![x]\!])$$

- for each $Y \in$ *AllTypeVars* with $Y \neq X$, $\rho[\![Y]\!] = \{(0, 0)\}$ (see under 'Some notation for assignments to type variables' in Section 2.4).

Theorem 4.2.2 then gives us that

$$([\![\Gamma \vdash \mathit{Impl}]\!]^{\mathrm{FD}}_{[X\mapsto\kappa],\eta})([\![\mathit{Proc}_{T,\psi}]\!]^{\mathrm{FD}}_{\rho,[X\mapsto\kappa],[X\mapsto\kappa']})([\![\Gamma \vdash \mathit{Impl}]\!]^{\mathrm{FD}}_{[X\mapsto\kappa'],\eta'}) \tag{5.13}$$

Suppose $(t, A) \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Impl})_{[X\mapsto\kappa],\eta}$. By (5.13), there exists

$$(t', A') \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Impl})_{[X\mapsto\kappa'],\eta'}$$

such that

$$t([\![T]\!]_{\rho,[X\mapsto\kappa],[X\mapsto\kappa']})^{*}t' \tag{5.14}$$

and

$$\begin{aligned}&\forall \sigma : [X \mapsto \kappa] \leftrightarrow [X \mapsto \kappa'].\\&(\forall Y \in \mathit{AllTypeVars}.(\sigma[\![Y]\!])^{-1} : ([X \mapsto \kappa'][\![Y]\!] \setminus \mathit{Range}(\rho[\![Y]\!])) \to [X \mapsto \kappa][\![Y]\!]) \Rightarrow\\&(\forall a' \in ([\![T]\!]_{[X\mapsto\kappa']} \setminus A').\exists a \in ([\![T]\!]_{[X\mapsto\kappa]} \setminus A).a([\![T]\!]_{(\rho\cup\sigma),[X\mapsto\kappa],[X\mapsto\kappa']})a')\end{aligned} \tag{5.15}$$

We have supposed that *Spec* $\sqsubseteq^{\mathrm{FD}}_{[X\mapsto\kappa'],\eta'}$ *Impl*, and so $(t', A') \in \mathit{failures}_{\perp}(\Gamma \vdash \mathit{Spec})_{[X\mapsto\kappa'],\eta'}$.

Let $\pi : [X \mapsto \kappa'] \leftrightarrow [X \mapsto \kappa]$ be such that:

- $\pi[\![X]\!] : ([X \mapsto \kappa'][\![X]\!] \setminus \mathit{Range}(\rho[\![X]\!])) \to ([X \mapsto \kappa][\![X]\!])$;
- for each $Y \in$ *AllTypeVars* with $Y \neq X$, $\pi[\![Y]\!] = \{\}$.

Then Lemma 5.2.1 (ii) gives us that there exists

$$(t^\dagger, A^\dagger) \in \mathit{failures}_\perp(\Gamma \vdash \mathit{Spec})_{[X \mapsto \kappa],\eta}$$

such that

$$t'(\llbracket T \rrbracket_{(\rho^{-1} \cup \pi),[X \mapsto \kappa'],[X \mapsto \kappa]})^* t^\dagger \tag{5.16}$$

and

$$\forall\, a^\dagger \in (\llbracket T \rrbracket_{[X \mapsto \kappa]} \setminus A^\dagger).\, \exists\, a' \in (\llbracket T \rrbracket_{[X \mapsto \kappa']} \setminus A').a'(\llbracket T \rrbracket_{(\rho^{-1} \cup \pi),[X \mapsto \kappa'],[X \mapsto \kappa]})a^\dagger \tag{5.17}$$

Now, $t^\dagger = t$ follows from (5.14) and (5.16), and $A^\dagger \supseteq A$ follows from (5.15) and (5.17). Thus we have that $(t, A) \in \mathit{failures}_\perp(\Gamma \vdash \mathit{Spec})_{[X \mapsto \kappa],\eta}$.

Therefore, $\mathit{failures}_\perp(\Gamma \vdash \mathit{Impl})_{[X \mapsto \kappa],\eta} \subseteq \mathit{failures}_\perp(\Gamma \vdash \mathit{Spec})_{[X \mapsto \kappa],\eta}$.

Showing the other half of

$$\mathit{Spec} \sqsubseteq^{\mathrm{FD}}_{[X \mapsto \kappa],\eta} \mathit{Impl}$$

namely that $\mathit{divergences}(\Gamma \vdash \mathit{Impl})_{[X \mapsto \kappa],\eta} \subseteq \mathit{divergences}(\Gamma \vdash \mathit{Spec})_{[X \mapsto \kappa],\eta}$, is simpler.

That completes the present case. The remaining cases are simpler. When either $M = \mathrm{Tr}$ or $\Gamma \vdash \mathit{Spec}$ satisfies $(\mathbf{NoEqT}_X)$, Theorem 4.2.2 is used instead of Lemma 5.2.1. ■

5.2.3 Some examples

Example 5.2.1 Recall the processes *NondReg* from Example 2.5.1 and *LastReg* from Example 2.6.1.

Since $\{\} \vdash \mathit{LastReg}$ satisfies $(\mathbf{NoEqT}_X)$ (and so also the weaker condition $(\mathbf{PosConjEqT}_X)$), the assumptions of Theorem 5.2.2 are satisfied with $\Gamma = \{\}$, $\mathit{Spec} = \mathit{LastReg}$, $\mathit{Impl} = \mathit{NondReg}$, any M, and $\kappa = 2$.

Thus Theorem 5.2.2 gives us that, if $\mathit{LastReg} \not\sqsubseteq^M_{[X \mapsto 2]} \mathit{NondReg}$ (which is the case, as we have observed in Example 2.6.1), then $\mathit{LastReg} \not\sqsubseteq^M_{[X \mapsto \kappa']} \mathit{NondReg}$ for all cardinals $\kappa' \geq 2$. ■

Example 5.2.2 Recall the processes *Spec* and *Impl* from Example 5.0.4, where we have observed that *Spec* and *Impl* are X-DI and that $\mathit{Spec} \sqsubseteq^M_\delta \mathit{Impl}$ holds if and only if $|\delta\llbracket X \rrbracket|$ is either finite and odd, or infinite.

Therefore, neither assumption (iv) nor assumption (v) can be omitted from Theorem 5.2.2. ■

5.3 Set of size 2

In this section, as in the previous section, we shall be requiring *Spec* and *Impl* to be data independent with respect to some X and Γ and to have the same alphabet T, and we shall be focussing on interpretations of X by nonzero cardinals κ (see the beginning of the previous section).

We shall present two theorems. The first theorem requires that:

- any interpretation of T is finite;

- for any channel component whose type $T_{i,j}$ contains a free occurence of X, *Impl* can use it either only for outputs or only for inputs (see under 'An assumption about implementation processes' at the beginning of the present chapter);
- for any channel component whose type $T_{i,j}$ contains a free occurence of X, if *Impl* can use it for inputs, then *Spec* can use it only for inputs;
- *Spec* satisfies (**PosConjEqT**$_X$) and (**Norm**);
- *Impl* satisfies (**NoEqT**$_X$);

and it provides a collection C of refinements, in which X is interpreted by the set $\{0, 1\}$, in which at most one constant from Γ is interpreted by 1, in which *Spec* and *Impl* are composed in parallel with one of the special processes *OneOne* or *Zeros* (see the remarks below), and which is such that:

- if all refinements in C hold, then *Spec* is refined by *Impl* for any interpretation of X and Γ;
- if a refinement in C fails, then *Spec* is not refined by *Impl* for all interpretations with an appropriate property;
- in the finite-traces model, for any refinement in C, the refinement obtained by omitting the special process from the specification side is equivalent to it (which may be significant in practical applications, where the latter refinement may be more efficient to check using a model checker than the former).

Moreover, the first theorem implies that the collection of all interpretations of X and Γ which interpret X by the set $\{0, 1\}$ and which interpret at most one constant from Γ by 1 is a threshold collection (in the terminology of the beginning of the present chapter).

However, the reason for using the parallel compositions with *OneOne* or *Zeros* is that they typically have exponentially fewer states than just *Spec* and *Impl*, which is likely to make the refinements exponentially more efficient to check on a model checker. More concretely, in the parallel composition of *Impl* and *OneOne* (respectively, of *Impl* and *Zeros*), *Impl* is restricted to, in any execution, input at most one value 1 (respectively, no values 1). (See Definition 5.3.1, Lemma 5.3.2 and Lemma 5.3.3.)

The second theorem, roughly speaking, has weaker assumptions and weaker conclusions. It requires that:

- *Spec* satisfies (**PosConjEqT**$_X$) and (**Norm**);
- for the finite-traces model, *Impl* satisfies (**PosConjEqT**$_X$);
- for the stable-failures and failures/divergences models, *Impl* satisfies (**NoEqT**$_X$);

and it states that:

- the collection of all interpretations of X and Γ which interpret X by the set $\{0, 1\}$ and which interpret a fixed constant from Γ by 0 is a threshold collection;
- if *Spec* is not refined by *Impl* for some interpretation from that collection, then *Spec* is not refined by *Impl* for all interpretations with an appropriate property.

We shall first present a few definitions and lemmas, then the two theorems, and finally three examples.

5.3.1 The definitions and lemmas

As we have remarked above, both of the theorems which we shall present require that *Spec* satisfies ($\mathbf{PosConjEqT}_X$) and (**Norm**). The following lemma states a consequence of those two conditions which is not a consequence of either of them separately.

Lemma 5.3.1 *In any ASLTS $\mathcal{AS}_{\Gamma \vdash P}$ where P is X,Γ-WDI and satisfies* ($\mathbf{PosConjEqT}_X$) *and* (**Norm**), *we have that for any invisible transition α, $guard(\alpha) = True$.*

Proof. The proof proceeds by inductions on the one-step reductions (see Table 3.1) and on the firing rules (see Sections 3.3.2 and 3.2.3), and it uses the 'Assumption of transformation' which was stated after the condition (**Norm**) in Section 3.5.2. ■

We are now going to define the special processes *OneOne* and *Zeros* (see the remarks at the beginning of Section 5.3). They will depend on X, T and ψ: in Theorem 5.3.5 (which is the first of the two theorems that Section 5.3 is devoted to), *Impl* will be data independent with respect to X and some Γ, and T and ψ will be its alphabet and ways specifier.

Instead of interpreting X by the set $\{0, 1\}$, we shall in fact be substituting X by a type $Two = zz.1 + oo.1$. In other words, X will be instantiated in the syntax rather than in the denotational semantics. That enables us to, in the definitions of $OneOne_{X,T,\psi}$ and $Zeros_{X,T,\psi}$, work directly with terms $(zz.())$ and $(oo.())$.

After defining the processes $OneOne_{X,T,\psi}$ and $Zeros_{X,T,\psi}$, we are also going to define three maps $Occs_{X,T_{i,j}}(\cdot)$, $Occs'_{X,T,\psi}(\cdot)$ and $Occs''_{X,T,\psi}(\cdot)$. They will be used for counting occurences of the value $(oo, ())$ in denotational semantic values, in places which correspond to inputs of values of type X.

Definition 5.3.1 Suppose that:

- $X \in NEFTypeVars \cap FinTypeVars$ (see under 'Type variables' in Section 2.4);
- T is a type which satisfies T^{alph} and T^{fin} (see under 'Conditions on types' in Section 2.4) and which is of the form

$$\sum_{i=1}^{n} id_i.\prod_{j=1}^{m_i} T_{i,j}$$

- $Free(T) \subseteq \{X\}$;
- ψ is a ways specifier with respect to T such that

$$\forall\, id_i, j.X \in Free(T_{i,j}) \Rightarrow (\psi(id_i, j) \subseteq \{!\} \vee \psi(id_i, j) \subseteq \{?\})$$

Then let the processes

$$\{\} \vdash OneOne_{X,T,\psi} : Proc_{\widetilde{T}^X,\psi'}$$
$$\{\} \vdash Zeros_{X,T,\psi} : Proc_{\widetilde{T}^X,\psi''}$$

be defined by

$$
\begin{array}{l}
\mathit{OneOne}_{X,T,\psi} = \mathit{OneOne}'_{id_1,X,T,\psi} \;\Box \cdots \Box\; \mathit{OneOne}'_{id_n,X,T,\psi} \\
\mathit{OneOne}'_{id_i,X,T,\psi} = \Box_{x_1:(T_{i,1}[\mathit{Two}/X])} \cdots \Box_{x_{m_i}:(T_{i,m_i}[\mathit{Two}/X])} \\
\qquad \mathit{if}\ \ (\sum\{\mathit{occs}_{X,T_{i,j}}\ x_j \mid j \in \{1,\ldots,m_i\} \wedge \\
\qquad\quad X \in \mathit{Free}(T_{i,j}) \wedge \psi(id_i,j) \subseteq \{?\}\}) \leq 1 \\
\qquad\quad \mathit{then}\ id_i!x_1\ldots!x_{m_i} \longrightarrow \mathit{if}\ \ (\sum\{\mathit{occs}_{X,T_{i,j}}\ x_j \mid j \in \{1,\ldots,m_i\} \wedge \\
\qquad\qquad\qquad X \in \mathit{Free}(T_{i,j}) \wedge \psi(id_i,j) \subseteq \{?\}\}) \geq 1 \\
\qquad\qquad\qquad \mathit{then}\ \mathit{Zeros}_{X,T,\psi} \\
\qquad\qquad\qquad \mathit{else}\ \mathit{OneOne}_{X,T,\psi} \\
\qquad\quad \mathit{else}\ \mathit{STOP} \\
\mathit{Zeros}_{X,T,\psi} = \mathit{Zeros}'_{id_1,X,T,\psi} \;\Box \cdots \Box\; \mathit{Zeros}'_{id_n,X,T,\psi} \\
\mathit{Zeros}'_{id_i,X,T,\psi} = \Box_{x_1:(T_{i,1}[\mathit{Two}/X])} \cdots \Box_{x_{m_i}:(T_{i,m_i}[\mathit{Two}/X])} \\
\qquad \mathit{if}\ \ (\sum\{\mathit{occs}_{X,T_{i,j}}\ x_j \mid j \in \{1,\ldots,m_i\} \wedge \\
\qquad\quad X \in \mathit{Free}(T_{i,j}) \wedge \psi(id_i,j) \subseteq \{?\}\}) = 0 \\
\qquad\quad \mathit{then}\ id_i!x_1\ldots!x_{m_i} \longrightarrow \mathit{Zeros}_{X,T,\psi} \\
\qquad\quad \mathit{else}\ \mathit{STOP}
\end{array}
$$

where:

- the indices X, T, ψ, id_i and $T_{i,j}$ are parameters at the meta level, rather than arguments within the language;
- $\mathit{Two} = zz.1 + oo.1$;
- $\widetilde{\cdot}^X$ is an abbreviation for $\cdot[\mathit{Two}/X]$;
- ψ' and ψ'' are ways specifiers with respect to $\widetilde{T}^X$;
- $\mathit{occs}_{X,T_{i,j}}\ x_j$ returns the number of occurences of $(oo,())$ in those places in the value of x_j which correspond to the places in $T_{i,j}$ in which X occurs free (we omit the details).

Also, for any id_i, $j \in \{1,\ldots,m_i\}$ and $w \in [\![\widetilde{T_{i,j}}^X]\!]$ such that $X \in \mathit{Free}(T_{i,j})$ and $\psi(id_i,j) \subseteq \{?\}$, let

$$\mathit{Occs}_{X,T_{i,j}}(w)$$

be the number of occurences of $(oo,())$ in those places in w which correspond to the places in $T_{i,j}$ in which X occurs free.

For any $a \in [\![\widetilde{T}^X]\!]$, let

$$
\begin{array}{rl}
\mathit{Occs}'_{X,T,\psi}(a) = & \sum\{\mathit{Occs}_{X,T_{i,j}}(v(j)) \mid a = (id_i,v) \wedge j \in \{1,\ldots,m_i\} \wedge \\
& X \in \mathit{Free}(T_{i,j}) \wedge \psi(id_i,j) \subseteq \{?\}\}
\end{array}
$$

For any $t \in ([\![\widetilde{T}^X]\!])^*$, let

$$\mathit{Occs}''_{X,T,\psi}(t) = \sum_{k=1}^{|t|} \mathit{Occs}'_{X,T,\psi}(t(k))$$

■

For any process P whose alphabet is $\widetilde{T}^X$ (i.e. T with the type Two substituted for X), the following lemma states how to compute the denotational semantics of P composed in parallel with $\mathit{OneOne}_{X,T,\psi}$ and of P composed in parallel with $\mathit{Zeros}_{X,T,\psi}$ from the denotational semantics of P.

Lemma 5.3.2 *Suppose that X, T and ψ are as in Definition 5.3.1. Then, for any process $\Gamma \vdash P : Proc_{\widetilde{T}^X, X}$, we have that:*

$$\begin{aligned}
& t \in traces(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} OneOne_{X,T,\psi})^{\mathrm{Tr}}_{\delta,\eta} \Leftrightarrow \\
& \quad t \in traces(\Gamma \vdash P)^{\mathrm{Tr}}_{\delta,\eta} \ \wedge \ Occs''_{X,T,\psi}(t) \leq 1
\end{aligned}$$

$$\begin{aligned}
& t \in traces(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} OneOne_{X,T,\psi})^{\mathrm{Fa}}_{\delta,\eta} \Leftrightarrow \\
& \quad t \in traces(\Gamma \vdash P)^{\mathrm{Fa}}_{\delta,\eta} \ \wedge \ Occs''_{X,T,\psi}(t) \leq 1
\end{aligned}$$

$$\begin{aligned}
& (t,A) \in failures(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} OneOne_{X,T,\psi})_{\delta,\eta} \Leftrightarrow \\
& \quad \exists\, B.(t,B) \in failures(\Gamma \vdash P)_{\delta,\eta} \ \wedge \\
& \quad Occs''_{X,T,\psi}(t) \leq 1 \ \wedge \ A \subseteq (B \cup \{a \in [\![\widetilde{T}^X]\!] \mid Occs''_{X,T,\psi}(t\hat{\ }\langle a\rangle) > 1\})
\end{aligned}$$

$$\begin{aligned}
& (t,A) \in failures_{\perp}(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} OneOne_{X,T,\psi})_{\delta,\eta} \wedge \\
& t \notin divergences(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} OneOne_{X,T,\psi})_{\delta,\eta} \Leftrightarrow \\
& \quad \exists\, B.(t,B) \in failures_{\perp}(\Gamma \vdash P)_{\delta,\eta} \ \wedge \\
& \quad t \notin divergences(\Gamma \vdash P)_{\delta,\eta} \ \wedge \\
& \quad Occs''_{X,T,\psi}(t) \leq 1 \ \wedge \ A \subseteq (B \cup \{a \in [\![\widetilde{T}^X]\!] \mid Occs''_{X,T,\psi}(t\hat{\ }\langle a\rangle) > 1\})
\end{aligned}$$

$$\begin{aligned}
& t \in \min(divergences(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} OneOne_{X,T,\psi})_{\delta,\eta}) \\
& \quad t \in \min(divergences(\Gamma \vdash P)_{\delta,\eta}) \ \wedge \ Occs''_{X,T,\psi}(t) \leq 1
\end{aligned}$$

and that:

$$\begin{aligned}
& t \in traces(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} Zeros_{X,T,\psi})^{\mathrm{Tr}}_{\delta,\eta} \Leftrightarrow \\
& \quad t \in traces(\Gamma \vdash P)^{\mathrm{Tr}}_{\delta,\eta} \ \wedge \ Occs''_{X,T,\psi}(t) = 0
\end{aligned}$$

$$\begin{aligned}
& t \in traces(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} Zeros_{X,T,\psi})^{\mathrm{Fa}}_{\delta,\eta} \Leftrightarrow \\
& \quad t \in traces(\Gamma \vdash P)^{\mathrm{Fa}}_{\delta,\eta} \ \wedge \ Occs''_{X,T,\psi}(t) = 0
\end{aligned}$$

$$\begin{aligned}
& (t,A) \in failures(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} Zeros_{X,T,\psi})_{\delta,\eta} \Leftrightarrow \\
& \quad \exists\, B.(t,B) \in failures(\Gamma \vdash P)_{\delta,\eta} \ \wedge \\
& \quad Occs''_{X,T,\psi}(t) = 0 \ \wedge \ A \subseteq (B \cup \{a \in [\![\widetilde{T}^X]\!] \mid Occs''_{X,T,\psi}(t\hat{\ }\langle a\rangle) > 0\})
\end{aligned}$$

$$\begin{aligned}
& (t,A) \in failures_{\perp}(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} Zeros_{X,T,\psi})_{\delta,\eta} \wedge \\
& t \notin divergences(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\parallel} Zeros_{X,T,\psi})_{\delta,\eta} \Leftrightarrow \\
& \quad \exists\, B.(t,B) \in failures_{\perp}(\Gamma \vdash P)_{\delta,\eta} \ \wedge \\
& \quad t \notin divergences(\Gamma \vdash P)_{\delta,\eta} \ \wedge
\end{aligned}$$

$$Occs''_{X,T,\psi}(t) = 0 \ \wedge \ A \subseteq (B \cup \{a \in [\![\widetilde{T}^X]\!] \mid Occs''_{X,T,\psi}(t\hat{}\langle a\rangle) > 0\})$$

$$t \in \min(divergences(\Gamma \vdash P \underset{\{*_{\widetilde{T}^X}\}}{\|} Zeros_{X,T,\psi})_{\delta,\eta})$$

$$t \in \min(divergences(\Gamma \vdash P)_{\delta,\eta}) \ \wedge \ Occs''_{X,T,\psi}(t) = 0$$

(For the definition of min*, see under 'Some notation for sequences' in Section 4.1.)*

Proof. Straightforward, by Definition 5.3.1. ■

For any type variable X from *NEFTypeVars*, any nonempty set D, and any subset E of D, we are now going to define an assignment

$$\pi_{X,D,E} : [X \mapsto D] \leftrightarrow [X \mapsto [\![Two]\!]]$$

of relations to type variables. For X, $\pi_{X,D,E}[\![X]\!]$ will be the relation between D and $[\![Two]\!]$ which relates any element of E both to $(zz,())$ and to $(oo,())$, and relates any element of $D \setminus E$ only to $(zz,())$. For any $Y \neq X$, $\pi_{X,D,E}[\![Y]\!]$ will be the relation between $\{0\}$ and $\{0\}$ which relates 0 to 0.

In the proof of Theorem 5.3.5, the assignments $\pi_{X,D,E}$ will be used for relating finite traces of *Spec* or *Impl* with X interpreted by D to finite traces of $\widetilde{Spec}^X$ or $\widetilde{Impl}^X$ composed in parallel with $OneOne_{X,T,\psi}$ (where $\widetilde{Spec}^X$ and $\widetilde{Impl}^X$ are *Spec* and *Impl* with X substituted by the type *Two*).

Definition 5.3.2 Suppose that $X \in NEFTypeVars$, D is a nonempty set, and $E \subseteq D$.
Then let

$$\pi_{X,D,E} : [X \mapsto D] \leftrightarrow [X \mapsto [\![Two]\!]]$$

be defined as follows:

- for X,

$$\begin{array}{rcl} [X \mapsto D][\![X]\!] & = & D \\ [X \mapsto [\![Two]\!]][\![X]\!] & = & [\![Two]\!] \\ v(\pi_{X,D,E}[\![X]\!])v' & \Leftrightarrow & (v \in E \wedge v' \in [\![Two]\!]) \vee (v \in (D \setminus E) \wedge v' = (zz,())) \end{array}$$

- for each $Y \in AllTypeVars$ with $Y \neq X$,

$$\begin{array}{rcl} [X \mapsto D][\![Y]\!] & = & \{0\} \\ [X \mapsto [\![Two]\!]][\![Y]\!] & = & \{0\} \\ \pi_{X,D,E}[\![Y]\!] & = & \{(0,0)\} \end{array}$$

(If additionally $X \in FinTypeVars$, then it is implicitly assumed that D is a finite set: see under 'Type variables' in Section 2.4.) ■

The following lemma states that, for any process P which is data independent with respect to X with no constants, which has T and ψ as its alphabet and ways specifier, and which satisfies $(\mathbf{NoEqT}_X)$, and for any finite trace t of P with X interpreted by D, and any place of type X (informally speaking) in t which contains a value v from D, there exists a finite trace t' of $\widetilde{P}^X$ composed in parallel with $\mathit{OneOne}_{X,T,\psi}$ (where $\widetilde{P}^X$ is P with X substituted by the type Two) such that t is related to t' by the logical relation determined by $\pi_{X,D,\{v\}}$ (see Definition 5.3.2) and such that t' contains the value $(oo, ())$ in the place which corresponds to the place in t.

Thus t' differs from t at most in places of type X, where any value $w \neq v$ is replaced by $(zz, ())$ and the value v can be replaced by either $(zz, ())$ or $(oo, ())$, except that in the chosen place the value v is replaced by $(oo, ())$. The possibility that v is replaced by $(zz, ())$ in some of the places of type X is allowed because when $\widetilde{P}^X$ is composed in parallel with $\mathit{OneOne}_{X,T,\psi}$, its inputs of type X become restricted so that, in any execution, it can input at most one value $(oo, ())$.

Lemma 5.3.3 *Suppose that:*

- X, T *and* ψ *are as in Definition 5.3.1;*
- $\{\} \vdash P : \mathit{Proc}_{T,\psi}$ *is* X*-DI and satisfies* $(\mathbf{NoEqT}_X)$*;*
- D *is a nonempty set, and* $v \in D$.

Then, for any $t \in \mathit{traces}(\{\} \vdash P)_{[X \mapsto D]}$ *and for any place in some* $t(l)$ *which contains* v *and which corresponds to a place in* T *in which* X *occurs free, there exists*

$$t' \in \mathit{traces}(\{\} \vdash \widetilde{P}^X \underset{\{*_{\widetilde{T}^X}\}}{\parallel} \mathit{OneOne}_{X,T,\psi})$$

such that

$$t (\llbracket T \rrbracket_{\pi_{X,D,\{v\}},[X \mapsto D],[X \mapsto \llbracket \mathit{Two} \rrbracket]})^* t'$$

and $(oo, ())$ *is contained in the place in* $t'(l)$ *which corresponds to the place in* $t(l)$*. (We omit the notational details.)*

Proof. By translating to and from the ASLTS $\mathcal{AS}_{\{\}\vdash P}$ with the help of Theorem 4.2.1 (see Sections 3.3.2 and 3.4, and see Proposition 3.5.1), and by Lemma 5.3.2. ■

For any data independent process, the following lemma states that any new channel can be added to its alphabet while essentially preserving its ways specifier and its denotational semantics.

Lemma 5.3.4 *Suppose that:*

- $\Gamma \vdash P : \mathit{Proc}_{T,\chi}$ *is* X,Γ*-DI, where* T *is of the form* $\sum_{i=1}^{n} id_i. \prod_{j=1}^{m_i} T_{i,j}$*;*
- id_{n+1} *is an identifier which is distinct from each of* $id_1, \ldots, id_n$*;*
- m_{n+1} *is a natural number, and* $T_{n+1,1}, \ldots, T_{n+1,m_{n+1}}$ *are types such that* $T_{n+1,j}^{comm}$ *and* $\mathit{Free}(T_{n+1,j}) \subseteq \{X\}$ *for all* j.

Then there exists $\Gamma \vdash P^\dagger : Proc_{T^\dagger,\chi^\dagger}$ *which is* X,Γ-*DI and such that:*

(i) $T^\dagger = \sum_{i=1}^{n+1} id_i . \prod_{j=1}^{m_i} T_{i,j}$*;*

(ii) for any id_i *and* j*,*

$$\chi^\dagger(id_i, j) = \begin{cases} \chi(id_i, j), & \text{if } id_i \neq id_{n+1} \\ \{\}, & \text{if } id_i = id_{n+1} \end{cases}$$

(iii)

$$t \in traces(\Gamma \vdash P^\dagger)_{\delta,\eta} \Leftrightarrow$$
$$\quad t \in traces(\Gamma \vdash P)_{\delta,\eta}$$

$$(t, A) \in failures(\Gamma \vdash P^\dagger)_{\delta,\eta} \Leftrightarrow$$
$$\quad A \subseteq [\![T^\dagger]\!]_\delta \ \wedge \ (t, A \cap [\![T]\!]_\delta) \in failures(\Gamma \vdash P)_{\delta,\eta}$$

$$(t, A) \in failures_\perp(\Gamma \vdash P^\dagger)_{\delta,\eta} \ \wedge \ t \notin divergences(\Gamma \vdash P^\dagger)_{\delta,\eta} \Leftrightarrow$$
$$\quad A \subseteq [\![T^\dagger]\!]_\delta \ \wedge \ (t, A \cap [\![T]\!]_\delta) \in failures_\perp(\Gamma \vdash P)_{\delta,\eta} \ \wedge \ t \notin divergences(\Gamma \vdash P)_{\delta,\eta}$$

$$t \in \min(divergences(\Gamma \vdash P^\dagger)_{\delta,\eta}) \Leftrightarrow$$
$$\quad t \in \min(divergences(\Gamma \vdash P)_{\delta,\eta})$$

Proof. Not difficult, using induction on the structure of the term P. ■

For any process P which is data independent with respect to X and Γ, and for any channel name id_{n+1} which is distinct from each channel name in the alphabet of P, we are now going to define a process $NoC(X, \Gamma, P, id_{n+1})$. It has no constants, it begins with inputs along id_{n+1} which correspond to the constants from Γ, and it then becomes a process obtained from P by adding the channel id_{n+1} of type X to its alphabet (using Lemma 5.3.4).

The processes $NoC(X, \Gamma, P, id_{n+1})$ will be used in the proofs of Theorems 5.3.5, 5.3.6 and 5.4.7 to reduce considering arbitrary data independent processes to considering only those with no constants.

Definition 5.3.3 Suppose that:

- $\Gamma \vdash P : Proc_{T,\chi}$ is X,Γ-DI, where T is of the form $\sum_{i=1}^{n} id_i . \prod_{j=1}^{m_i} T_{i,j}$;
- id_{n+1} is an identifier which is distinct from each of id_1, ..., id_n.

Then let

$$NoC(X, \Gamma, P, id_{n+1}) = id_{n+1}?x_1 : X \longrightarrow \cdots \longrightarrow id_{n+1}?x_k : X \longrightarrow P^\dagger$$

where $\Gamma = \{x_1 : X, \ldots, x_k : X\}$, and where $P^\dagger$ is a process obtained by applying Lemma 5.3.4 with $m_{n+1} = 1$ and $T_{n+1,1} = X$. ■

5.3.2 The theorems

Theorem 5.3.5 *Suppose that:*

(i) $\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$ *and* $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ *are* X,Γ*-DI, where* Γ *is of the form* $\{x_1 : X, \ldots, x_k : X\}$*, and* T *is of the form* $\sum_{i=1}^{n} id_i . \prod_{j=1}^{m_i} T_{i,j}$*;*

(ii) M *specifies one of the models Tr, Fa, FD of CSP;*

(iii) T *satisfies* T^{fin}*;*

(iv) $\forall\, id_i, j. X \in \mathit{Free}(T_{i,j}) \Rightarrow (\psi(id_i, j) \subseteq \{!\} \vee \psi(id_i, j) \subseteq \{?\})$*;*

(v) $\forall\, id_i, j. (X \in \mathit{Free}(T_{i,j}) \wedge \psi(id_i, j) = \{?\}) \Rightarrow \phi(id_i, j) \subseteq \{?\}$*;*

(vi) $\Gamma \vdash \mathit{Spec}$ *satisfies* $(\mathbf{PosConjEqT}_X)$ *and* $(\mathbf{Norm})$*;*

(vii) $\Gamma \vdash \mathit{Impl}$ *satisfies* $(\mathbf{NoEqT}_X)$*.*

Let C *be the following collection of* $k+1$ *refinements:*

$$\begin{array}{l}
\{\widetilde{\Gamma}^X \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi} \sqsubseteq^M_{\delta,\eta} \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi},\\
\quad \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{Zeros}_{X,T,\psi} \sqsubseteq^M_{\delta,\eta'_1} \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{Zeros}_{X,T,\psi},\\
\qquad\qquad\qquad\vdots\\
\quad \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{Zeros}_{X,T,\psi} \sqsubseteq^M_{\delta,\eta'_k} \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{Zeros}_{X,T,\psi}\}
\end{array}$$

where δ *is arbitrary, and for any* $l, l^\dagger \in \{1, \ldots, k\}$*:*

$$\begin{array}{rcl}
\eta[\![x_{l^\dagger}]\!] & = & (zz, ())\\
\eta'_l[\![x_{l^\dagger}]\!] & = & \left\{\begin{array}{ll} (oo, ()), & \text{if } l^\dagger = l\\ (zz, ()), & \text{if } l^\dagger \neq l \end{array}\right.
\end{array}$$

Then:

(I) if all the refinements in C *hold, then* $\mathit{Spec} \sqsubseteq^M_{[X \mapsto \kappa],\eta''} \mathit{Impl}$ *for all nonzero natural numbers* κ *and all* η''*;*

(II) if

$$\widetilde{\Gamma}^X \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi} \not\sqsubseteq^M_{\delta,\eta} \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

then $\mathit{Spec} \not\sqsubseteq^M_{[X \mapsto \kappa],\eta''} \mathit{Impl}$ *for all natural numbers* $\kappa \geq 2$ *and all* η'' *such that*

$$\eta''[\![x_1]\!] = \cdots = \eta''[\![x_k]\!]$$

(III) if, for some $l \in \{1, \dots, k\}$,

$$\widetilde{\Gamma}^X \vdash \widetilde{Spec}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} Zeros_{X,T,\psi} \not\sqsubseteq^M_{\delta,\eta'_l} \widetilde{\Gamma}^X \vdash \widetilde{Impl}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} Zeros_{X,T,\psi}$$

then $Spec \not\sqsubseteq^M_{[X\mapsto\kappa],\eta''} Impl$ *for all natural numbers* $\kappa \geq 2$ *and all* η'' *such that*

$$\eta''[\![x_1]\!] = \dots = \eta''[\![x_{l-1}]\!] = \eta''[\![x_{l+1}]\!] = \dots = \eta''[\![x_k]\!] \neq \eta''[\![x_l]\!]$$

(IV)
$$\widetilde{\Gamma}^X \vdash \widetilde{Spec}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} OneOne_{X,T,\psi} \sqsubseteq^{\mathrm{Tr}}_{\delta,\eta} \widetilde{\Gamma}^X \vdash \widetilde{Impl}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} OneOne_{X,T,\psi}$$

if and only if

$$\widetilde{\Gamma}^X \vdash \widetilde{Spec}^X \sqsubseteq^{\mathrm{Tr}}_{\delta,\eta} \widetilde{\Gamma}^X \vdash \widetilde{Impl}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} OneOne_{X,T,\psi}$$

(V) for any $l \in \{1, \dots, k\}$,

$$\widetilde{\Gamma}^X \vdash \widetilde{Spec}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} Zeros_{X,T,\psi} \sqsubseteq^{\mathrm{Tr}}_{\delta,\eta'_l} \widetilde{\Gamma}^X \vdash \widetilde{Impl}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} Zeros_{X,T,\psi}$$

if and only if

$$\widetilde{\Gamma}^X \vdash \widetilde{Spec}^X \sqsubseteq^{\mathrm{Tr}}_{\delta,\eta'_l} \widetilde{\Gamma}^X \vdash \widetilde{Impl}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} Zeros_{X,T,\psi}$$

Proof.

(I) By considering processes $NoC(X, \Gamma, Spec, id_{n+1})$ and $NoC(X, \Gamma, Impl, id_{n+1})$ where id_{n+1} is an identifier which is distinct from each of $id_1, \dots, id_n$ (see Definition 5.3.3), and by Lemmas 5.3.4 and 5.3.2, it follows that it suffices to consider the case $\Gamma = \{\}$.

Therefore, we assume that

$$Spec \not\sqsubseteq^M_{[X\mapsto\kappa]} Impl$$

for some nonzero natural number κ, and show that

$$\widetilde{Spec}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} OneOne_{X,T,\psi} \not\sqsubseteq^M \widetilde{Impl}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} OneOne_{X,T,\psi}$$

Case $M = \mathrm{Tr}$. Since $Spec \not\sqsubseteq^{\mathrm{Tr}}_{[X\mapsto\kappa]} Impl$, we can pick some

$$t\hat{\ }\langle a\rangle \in \min(traces(\{\} \vdash Impl)_{[X\mapsto\kappa]} \setminus traces(\{\} \vdash Spec)_{[X\mapsto\kappa]})$$

Then $t \in traces(\{\} \vdash Spec)_{[X\mapsto\kappa]}$ by minimality, so let

$$N_0^{\{\}} \alpha_0^{\theta'_0} \dots N_l^{\theta_l}$$

be an execution in $\mathcal{AS}_{\{\}\vdash Spec}$ with respect to $[X \mapsto \kappa]$ which generates t, which is such that either $l = 0$ or α_{l-1} is a visible transition, and where N_0 is the initial node. (See Section 3.4.)

If a visible transition β such that the shell of $r(\beta)$ matches a (see under 'Alternative symbolic transitions' in Section 3.3.2, see under 'Visible' in Section 3.2.2, and see Section 3.1.2) is not reachable from N_l by zero or more invisible transitions, then it follows by Propositions 3.5.2 and 3.5.4 and by Lemma 5.3.1 that

$$t'' \notin \mathit{traces}(\{\} \vdash \widetilde{\mathit{Spec}}^{X})$$

where t'' is such that $(t\hat{\ }\langle a\rangle)(\llbracket T\rrbracket_{\pi_{X,\kappa,\{\}},[X\mapsto\kappa],[X\mapsto\llbracket \mathit{Two}\rrbracket]})^{*}t''$. Hence

$$t'' \notin \mathit{traces}(\{\} \vdash \widetilde{\mathit{Spec}}^{X} \underset{\{*_{\tilde{T}}X\}}{\parallel} \mathit{OneOne}_{X,T,\psi})$$

by Lemma 5.3.2. On the other hand, it follows by Theorem 4.2.2 that

$$t'' \in \mathit{traces}(\{\} \vdash \widetilde{\mathit{Impl}}^{X})$$

and so

$$t'' \in \mathit{traces}(\{\} \vdash \widetilde{\mathit{Impl}}^{X} \underset{\{*_{\tilde{T}}X\}}{\parallel} \mathit{OneOne}_{X,T,\psi})$$

by Lemma 5.3.2. Therefore, as required,

$$\widetilde{\mathit{Spec}}^{X} \underset{\{*_{\tilde{T}}X\}}{\parallel} \mathit{OneOne}_{X,T,\psi} \not\sqsubseteq^{\mathrm{Tr}} \widetilde{\mathit{Impl}}^{X} \underset{\{*_{\tilde{T}}X\}}{\parallel} \mathit{OneOne}_{X,T,\psi}$$

Otherwise, let β be a visible transition such that the shell of $r(\beta)$ matches a and such that β is reachable from N_l by zero or more invisible transitions. By Proposition 3.5.4, we have that:

- $\mathit{cond}(\beta) = \mathit{True}$, and so in particular each variable from $\mathit{ys}(\beta)$ occurs in $r(\beta)$;
- β is unique.

Let

$$N_l^{\theta_l}\alpha_l^{\theta'_l}\ldots N_{l^\dagger}^{\theta_{l^\dagger}}$$

be an execution in $\mathcal{AS}_{\{\}\vdash \mathit{Spec}}$ with respect to $[X\mapsto\kappa]$ which consists of only invisible transitions and such that $N_{l^\dagger} = \mathit{source}(\beta)$.

If $\theta_{l^\dagger}$ does not satisfy $\mathit{guard}(\beta)$, then by Proposition 3.5.2, there exist

$$(x_1 : X), (x_2 : X) \in \mathit{Context}(N_{l^\dagger})$$

such that $\theta_{l^\dagger}\llbracket x_1\rrbracket \neq \theta_{l^\dagger}\llbracket x_2\rrbracket$. In that case, let i be such that $x_2 \in \mathit{Vars}(\alpha_i)$ and $x_2 \notin \mathit{Vars}(\alpha_j)$ for all $j \in \{i+1,\ldots,l^\dagger-1\}$. By applying Lemma 5.3.3, we get some

$$t'\hat{\ }\langle a'\rangle \in \mathit{traces}(\{\} \vdash \widetilde{\mathit{Impl}}^{X} \underset{\{*_{\tilde{T}}X\}}{\parallel} \mathit{OneOne}_{X,T,\psi})$$

such that $(t\hat{\ }\langle a\rangle)(\llbracket T\rrbracket_{\pi_{X,\kappa,\{\theta_{l^\dagger}\llbracket x_2\rrbracket\}},[X\mapsto\kappa],[X\mapsto\llbracket \mathit{Two}\rrbracket]})^{*}(t'\hat{\ }\langle a'\rangle)$ and such that $t'(i^\dagger)$ contains $(\mathit{oo}, ())$ in the place which corresponds to the unique place in which x_2 occurs in $r(\alpha_i)$ (see Proposition 3.5.4 (a) and under 'Visible' in Section 3.2.2), where

$$i^\dagger = |\{j \mid j \in \{0,\ldots,i\} \wedge \alpha_j \textit{ is a visible transition}\}|$$

Now, if we let

$$N_0^{\{\}}{\alpha'}_0^{\theta'''_0}\ldots N_l^{\theta''_l}{\alpha'}_l^{\theta'''_l}\ldots N_{l^\dagger}^{\theta''_{l^\dagger}}$$

be an execution in $\mathcal{AS}_{\{\}\vdash Spec}$ with respect to $[X \mapsto [\![Two]\!]]$ which generates t', where by Proposition 3.5.4 any such execution contains N_l, then it follows that $\theta''_{l^\dagger}[\![x_1]\!] = (zz, ())$ and $\theta''_{l^\dagger}[\![x_2]\!] = (oo, ())$, and so $\theta''_{l^\dagger}$ does not satisfy $guard(\beta)$. Consequently, it follows that $t'^\frown\langle a'\rangle \notin traces(\{\} \vdash \widetilde{Spec}^X)$, and so

$$t'^\frown\langle a'\rangle \notin traces(\{\} \vdash \widetilde{Spec}^X \underset{\{*_{\tilde{T}X}\}}{\|} OneOne_{X,T,\psi})$$

by Lemma 5.3.2. Therefore, as required,

$$\widetilde{Spec}^X \underset{\{*_{\tilde{T}X}\}}{\|} OneOne_{X,T,\psi} \not\sqsubseteq^{\mathrm{Tr}} \widetilde{Impl}^X \underset{\{*_{\tilde{T}X}\}}{\|} OneOne_{X,T,\psi}$$

The only other possible reason for $t^\frown\langle a\rangle \notin traces(\{\} \vdash Spec)_{[X\mapsto\kappa]}$ is that there exists $x \in Context(N_{l^\dagger})$ which occurs in $r(\beta)$ and for which $\theta_{l^\dagger}[\![x]\!]$ does not agree with a. If that is the case, we can apply Lemma 5.3.3 to x as we did above to x_2, which again gives us some

$$\begin{aligned} t'^\frown\langle a'\rangle \in\ & (traces(\{\} \vdash \widetilde{Impl}^X \underset{\{*_{\tilde{T}X}\}}{\|} OneOne_{X,T,\psi}) \setminus \\ & traces(\{\} \vdash \widetilde{Spec}^X \underset{\{*_{\tilde{T}X}\}}{\|} OneOne_{X,T,\psi})) \end{aligned}$$

Case $M = \mathrm{Fa}$. If $Spec \not\sqsubseteq^{\mathrm{Fa}}_{[X\mapsto\kappa]} Impl$ because

$$traces(\{\} \vdash Impl)_{[X\mapsto\kappa]} \not\subseteq traces(\{\} \vdash Spec)_{[X\mapsto\kappa]}$$

then it follows from the case $M = \mathrm{Tr}$ that

$$\widetilde{Spec}^X \underset{\{*_{\tilde{T}X}\}}{\|} OneOne_{X,T,\psi} \not\sqsubseteq^{\mathrm{Fa}} \widetilde{Impl}^X \underset{\{*_{\tilde{T}X}\}}{\|} OneOne_{X,T,\psi}$$

Otherwise, we can assume that $traces(\{\} \vdash Impl)_{[X\mapsto\kappa]} \subseteq traces(\{\} \vdash Spec)_{[X\mapsto\kappa]}$ and $failures(\{\} \vdash Impl)_{[X\mapsto\kappa]} \not\subseteq failures(\{\} \vdash Spec)_{[X\mapsto\kappa]}$, so that there exists (t, B) such that

$$\begin{aligned} & (t, B) \in (failures(\{\} \vdash Impl)_{[X\mapsto\kappa]} \setminus failures(\{\} \vdash Spec)_{[X\mapsto\kappa]}) \\ & \wedge t \in traces(\{\} \vdash Spec)_{[X\mapsto\kappa]} \end{aligned}$$

Let O_0 be the initial node of $\mathcal{AS}_{\{\}\vdash Impl}$, and let $O^\dagger$, $\xi^\dagger$ and ξ^* be such that

$$O_0^{\{\}} \overset{t}{\Longrightarrow} O^{\dagger\xi^\dagger}\ w.r.t.\ [X \mapsto \kappa] \ \wedge\ O^{\dagger\xi^\dagger}\ ref^{\xi^*}\ B\ w.r.t.\ [X \mapsto \kappa]$$

(see Section 3.4). Then we have that:

- $ys(\beta) = \{\}$ for any visible transition β of $O^\dagger$ (by Proposition 5.0.1);
- $O^\dagger$ has no invisible transitions (by Proposition 3.5.1).

Let $B^\dagger$ be the maximum subset of $[\![T]\!]_{[X\mapsto\kappa]}$ (with respect to $\subseteq$) such that

$$O^{\dagger\xi^\dagger}\ ref^{(\beta\mapsto\{\})}\ B^\dagger\ w.r.t.\ [X \mapsto \kappa]$$

In particular, we have that $B \subseteq B^\dagger$.

On the specification side, let

$$N_0^{\{\}} \alpha_0^{\theta'_0} \ldots N_l^{\theta_l}$$

be an execution in $\mathcal{AS}_{\{\}\vdash Spec}$ with respect to $[X \mapsto \kappa]$ which generates t, which is such that either $l = 0$ or α_{l-1} is a visible transition, and where N_0 is the initial node. Since $(t, B^\dagger) \notin \mathit{failures}(\{\} \vdash \mathit{Spec})_{[X\mapsto\kappa]}$, we have that

$$\begin{aligned}&\neg\,\exists\, N^\dagger, \theta^\dagger, \theta^*.\\ &N_l^{\theta_l} \overset{\langle\rangle}{\Longrightarrow} N^{\dagger\theta^\dagger} \;\; w.r.t.\ [X \mapsto \kappa]\\ &\wedge N^{\dagger\theta^\dagger} \;\; \mathit{ref}^{\theta^*} \;\; B^\dagger \;\; w.r.t.\ [X \mapsto \kappa]\end{aligned} \tag{5.18}$$

Let t' be such that $t(\llbracket T\rrbracket_{\pi_{X,\kappa,\{\}},[X\mapsto\kappa],[X\mapsto\llbracket Two\rrbracket]})^* t'$, and let $\xi''^\dagger\llbracket x\rrbracket = (zz, ())$ for all $x \in \mathit{Context}(O^\dagger)$. Then, by Proposition 3.5.1 and Theorem 4.2.1, we have that

$$O_0^{\{\}} \overset{t'}{\Longrightarrow} O^{\dagger\xi''^\dagger} \;\; w.r.t.\ [X \mapsto \llbracket Two\rrbracket]$$

Let $B'^\dagger$ be the maximum subset of $\llbracket T\rrbracket_{[X\mapsto\llbracket Two\rrbracket]}$ (with respect to $\subseteq$) such that

$$O^{\dagger\xi''^\dagger} \;\; \mathit{ref}^{(\beta\mapsto\{\})} \;\; B'^\dagger \;\; w.r.t.\ [X \mapsto \llbracket Two\rrbracket]$$

and let

$$D'^\dagger = B'^\dagger \cup \{b' \in \llbracket \widetilde{T}^X\rrbracket \mid \mathit{Occs}'_{X,T,\psi}(b') > 1\}$$

We claim that

$$\begin{aligned}&(t', D'^\dagger) \in (\mathit{failures}(\{\} \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi})\setminus\\ &\mathit{failures}(\{\} \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi}))\end{aligned} \tag{5.19}$$

Since $(t', B'^\dagger) \in \mathit{failures}(\{\} \vdash \widetilde{\mathit{Impl}}^X)$, Lemma 5.3.2 gives us that

$$(t', D'^\dagger) \in \mathit{failures}(\{\} \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi})$$

On the specification side, by Propositions 3.5.2 and 3.5.4, by Lemma 5.3.1, and by Theorem 4.2.1, it follows that there exist θ_1'', …, θ_l'' and θ_0''', …, θ_{l-1}''' such that

$$N_0^{\{\}} \alpha_0^{\theta_0'''} \ldots N_l^{\theta_l''}$$

is an execution in $\mathcal{AS}_{\{\}\vdash Spec}$ with respect to $[X \mapsto \llbracket Two\rrbracket]$ which generates t'.
By Lemma 5.3.2,

$$t' \in \mathit{traces}(\{\} \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi})$$

Consider any

$$(t', C') \in \mathit{failures}(\{\} \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi})$$

By Lemma 5.3.2 again, there exist $N^\dagger$, $\theta''^\dagger$, θ''^* and $A'^\dagger$ such that:

- $N_l^{\theta_l''} \overset{\langle\rangle}{\Longrightarrow} N^{\dagger\theta''^\dagger}$ $w.r.t.\ [X \mapsto \llbracket Two\rrbracket]$;
- $A'^\dagger$ is the maximum subset of $\llbracket T\rrbracket_{[X\mapsto\llbracket Two\rrbracket]}$ (with respect to $\subseteq$) such that
$$N^{\dagger\theta''^\dagger} \;\; \mathit{ref}^{\theta''^*} \;\; A'^\dagger \;\; w.r.t.\ [X \mapsto \llbracket Two\rrbracket]$$
- $C' \subseteq (A'^\dagger \cup \{a' \in \llbracket \widetilde{T}^X\rrbracket \mid \mathit{Occs}'_{X,T,\psi}(a') > 1\})$.

Now, let $\theta^\dagger = \theta_l \restriction \mathit{Context}(N^\dagger)$. By (5.18), $N^\dagger$ has a visible transition $\alpha^\dagger$ such that $\theta^\dagger$ satisfies $\mathit{guard}(\alpha^\dagger)$ and such that, for any assignment $\theta^\natural$ to $\mathit{ys}(\alpha^\dagger)$ with respect to $[X \mapsto \kappa]$, we have that

$$\{[\![r(\alpha^\dagger)]\!]_{[X\mapsto\kappa],\theta^\dagger\cup\theta^\natural\cup\theta^\sharp} \mid \theta^\sharp \textit{ is an ass. to } \mathit{zs}(\alpha^\dagger) \textit{ w.r.t. } [X \mapsto \kappa]\} \cap B^\dagger \neq \{\}$$

On the other hand, recalling that $\mathit{traces}(\{\} \vdash \mathit{Impl})_{[X\mapsto\kappa]} \subseteq \mathit{traces}(\{\} \vdash \mathit{Spec})_{[X\mapsto\kappa]}$, it follows by Propositions 3.5.2 and 3.5.4 and by Lemma 5.3.1 that, for any visible transition β of $O^\dagger$ such that the shell of $r(\beta)$ is the same as the shell of $r(\alpha^\dagger)$,

$$\begin{array}{ll}\{[\![r(\beta)]\!]_{[X\mapsto\kappa],\xi^\dagger\cup\xi^\sharp} \mid \xi^\sharp \textit{ is an assignment to } \mathit{zs}(\beta) \textit{ w.r.t. } [X \mapsto \kappa]\} \subseteq & \\ \{[\![r(\alpha^\dagger)]\!]_{[X\mapsto\kappa],\theta^\dagger\cup\theta^\natural\cup\theta^\sharp} \mid & \theta^\natural \textit{ is an assignment to } \mathit{ys}(\alpha^\dagger) \textit{ w.r.t. } [X \mapsto \kappa] \wedge \\ & \theta^\sharp \textit{ is an assignment to } \mathit{zs}(\alpha^\dagger) \textit{ w.r.t. } [X \mapsto \kappa]\}\end{array}$$

Therefore, for any such β, there must exist $g_\beta \in \{1,\ldots,h\}$ such that

$$s_{\beta,g_\beta} \in \mathit{Context}(O^\dagger) \ \wedge\ r_{g_\beta} \in \mathit{zs}(\alpha^\dagger)$$

where $H[r_1,\ldots,r_h] = r(\alpha^\dagger)$ and $H[s_{\beta,1},\ldots,s_{\beta,h}] = r(\beta)$.

Consequently, by assumptions (iv) and (v), $\alpha^\dagger$ gives rise to some $a' \in ([\![\widetilde{T}^X]\!] \setminus A'^\dagger)$ such that $a' \in ([\![\widetilde{T}^X]\!] \setminus C')$ and such that a' contains $(\mathit{oo},())$ in each of the places which correspond to the places in which the r_{g_β} occur in $r(\alpha^\dagger)$. Since any $b' \in ([\![\widetilde{T}^X]\!] \setminus B'^\dagger)$ which matches the shell $H[\cdot,\ldots,\cdot]$ must come from some β as above, and thus contain $(\mathit{zz},())$ in the place which corresponds to the place in which s_{β,g_β} occurs in $r(\beta)$, we have that $a' \in B'^\dagger$. Since $B'^\dagger \subseteq D'^\dagger$, we have that $C' \neq D'^\dagger$.

That establishes (5.19), which means that, as required,

$$\widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi} \not\sqsubseteq^{\mathrm{Fa}} \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

Case $M = \mathrm{FD}$. If $\mathit{Spec} \not\sqsubseteq^{\mathrm{FD}}_{[X\mapsto\kappa]} \mathit{Impl}$ because

$$\begin{array}{l}\exists\, t \in \mathit{traces}(\{\} \vdash \mathit{Impl})_{[X\mapsto\kappa]}.\\ t \notin (\mathit{divergences}(\{\} \vdash \mathit{Impl})_{[X\mapsto\kappa]} \setminus \min(\mathit{divergences}(\{\} \vdash \mathit{Impl})_{[X\mapsto\kappa]}))\\ \wedge\, (t,\{\}) \notin \mathit{failures}_\perp(\{\} \vdash \mathit{Spec})_{[X\mapsto\kappa]}\end{array}$$

then it follows as in the case $M = \mathrm{Tr}$ that

$$\widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi} \not\sqsubseteq^{\mathrm{FD}} \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

If $\mathit{Spec} \not\sqsubseteq^{\mathrm{FD}}_{[X\mapsto\kappa]} \mathit{Impl}$ because

$$\begin{array}{l}\exists (t,A) \in (\mathit{failures}_\perp(\{\} \vdash \mathit{Impl})_{[X\mapsto\kappa]} \setminus \mathit{failures}_\perp(\{\} \vdash \mathit{Spec})_{[X\mapsto\kappa]}).\\ t \notin \mathit{divergences}(\{\} \vdash \mathit{Impl}) \ \wedge\ (t,\{\}) \in \mathit{failures}_\perp(\{\} \vdash \mathit{Spec})_{[X\mapsto\kappa]}\end{array}$$

then it follows as in the case $M = \mathrm{Fa}$ that

$$\widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi} \not\sqsubseteq^{\mathrm{FD}} \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

Otherwise, $\mathit{Spec} \not\sqsubseteq^{\mathrm{FD}}_{[X\mapsto\kappa]} \mathit{Impl}$ because

$$\begin{array}{l}\exists\, t \in (\min(\mathit{divergences}(\{\} \vdash \mathit{Impl})_{[X\mapsto\kappa]}) \setminus \mathit{divergences}(\{\} \vdash \mathit{Spec})_{[X\mapsto\kappa]}).\\ t \in \mathit{traces}(\{\} \vdash \mathit{Spec})_{[X\mapsto\kappa]}\end{array}$$

Let t' be such that $t([\![T]\!]_{\pi_{X,\kappa,\{\}},[X\mapsto\kappa],[X\mapsto[\![Two]\!]]})^* t'$. By Theorem 4.2.2, we have that $t' \in \mathit{divergences}(\{\} \vdash \widetilde{\mathit{Impl}}^X)$, and so Lemma 5.3.2 gives us that

$$t' \in \mathit{divergences}(\{\} \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}}X\}}{\|} \mathit{OneOne}_{X,T,\psi})$$

On the other hand, by Propositions 3.5.2 and 3.5.4 and by Lemma 5.3.1, it follows that $t' \notin \mathit{divergences}(\{\} \vdash \widetilde{\mathit{Spec}}^X)$, and so Lemma 5.3.2 gives us that

$$t' \notin \mathit{divergences}(\{\} \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}}X\}}{\|} \mathit{OneOne}_{X,T,\psi})$$

Therefore, as required,

$$\widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}}X\}}{\|} \mathit{OneOne}_{X,T,\psi} \not\sqsubseteq^{\mathrm{FD}} \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}}X\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

(II) If

$$\widetilde{\Gamma}^X \vdash \widetilde{\mathit{Spec}}^X \underset{\{*_{\widetilde{T}}X\}}{\|} \mathit{OneOne}_{X,T,\psi} \not\sqsubseteq^M_{\delta,\eta} \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Impl}}^X \underset{\{*_{\widetilde{T}}X\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

then it is straightforward to see that $\widetilde{\Gamma}^X \vdash \widetilde{\mathit{Spec}}^X \not\sqsubseteq^M_{\delta,\eta} \widetilde{\Gamma}^X \vdash \widetilde{\mathit{Impl}}^X$. Corollary 4.2.3 then gives us that $\mathit{Spec} \not\sqsubseteq^M_{[X\mapsto 2],\eta^\dagger} \mathit{Impl}$, where $\eta^\dagger[\![x_{l^\dagger}]\!] = 0$ for all $l^\dagger \in \{1,\ldots,k\}$, and so Theorem 5.2.2 gives us that $\mathit{Spec} \not\sqsubseteq^M_{[X\mapsto\kappa],\eta''} \mathit{Impl}$ for all natural numbers $\kappa \geq 2$ and all η'' such that

$$\eta''[\![x_1]\!] = \cdots = \eta''[\![x_k]\!]$$

(III) This part is shown in the same way as part (II).

(IV) Straightforward, for example by Lemma 5.3.2.

(V) The same as part (IV).

■

Theorem 5.3.6 *Suppose that:*

(i) $\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$ *and* $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ *are* X,Γ*-DI, where* Γ *is of the form* $\{x_1 : X, \ldots, x_k : X\}$*;*

(ii) M *specifies one of the models* Tr, Fa, FD *of CSP;*

(iii) $\Gamma \vdash \mathit{Spec}$ *satisfies* (**PosConjEqT**$_X$) *and* (**Norm**)*;*

(iv) if $M = \mathrm{Tr}$*, then* $\Gamma \vdash \mathit{Impl}$ *satisfies* (**PosConjEqT**$_X$)*;*

(v) if $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$*, then* $\Gamma \vdash \mathit{Impl}$ *satisfies* (**NoEqT**$_X$)*.*

For any $D \subseteq \{2, \ldots, k\}$*, and any* $l \in \{1, \ldots, k\}$*, let*

$$\eta_D[\![x_l]\!] = \begin{cases} 1, & \text{if } l \in D \\ 0, & \text{if } l \notin D \end{cases}$$

Then:

(I) if Spec $\sqsubseteq^M_{[X \mapsto 2], \eta_D}$ *Impl for all* $D \subseteq \{2, \ldots, k\}$*, then Spec* $\sqsubseteq^M_{[X \mapsto \kappa], \eta'}$ *Impl for all nonzero cardinals* κ *and all* η'*;*

(II) if Spec $\not\sqsubseteq^M_{[X \mapsto 2], \eta_D}$ *Impl for some* $D \subseteq \{2, \ldots, k\}$*, then Spec* $\not\sqsubseteq^M_{[X \mapsto \kappa], \eta'}$ *Impl for all cardinals* $\kappa \geq 2$ *and all* η' *such that*

$$\forall l, l^\dagger \in \{1, \ldots, k\}.(\eta'[\![x_l]\!] = \eta'[\![l^\dagger]\!]) \Leftrightarrow ((l \in D \wedge l^\dagger \in D) \vee (l \notin D \wedge l^\dagger \notin D))$$

(If $X \in \mathit{FinTypeVars}$*, which may be required when* $M = \mathrm{FD}$ *(see under 'Restrictions with FD' in Section 2.5.2), then it is implicitly assumed that* κ *is a natural number (see Section 2.4).)*

Proof.

(I) The proof of this part is very similar to the proof of Theorem 5.3.5 (I). (A difference is that Theorem 4.2.2 is used instead of Lemma 5.3.3.)

(II) This part follows from Theorem 5.2.2.

■

5.3.3 Some examples

Example 5.3.1 Recall the processes B and *BImpl* from Example 2.1.3, which in our language can be written as terms

$$\{\} \vdash B : \mathit{Proc}_{T,\phi}$$
$$\{\} \vdash \mathit{BImpl} : \mathit{Proc}_{T,\psi}$$

where

$$T = \mathit{in}.X + \mathit{mid}.X + \mathit{out}.X$$

for some $X \in \mathit{NEFTypeVars}$, and where

$$\phi(\mathit{in}, 1) = \{?\} \qquad \phi(\mathit{mid}, 1) = \{\} \qquad \phi(\mathit{out}, 1) = \{!\}$$
$$\psi(\mathit{in}, 1) = \{?\} \qquad \psi(\mathit{mid}, 1) = \{\} \qquad \psi(\mathit{out}, 1) = \{!\}$$

The assumptions of Theorem 5.3.6 are satisfied with $\Gamma = \{\}$, $\mathit{Spec} = B$, $\mathit{Impl} = \mathit{BImpl}$ and any M, so in particular we have that if $B \sqsubseteq^M_{[X \mapsto 2]} \mathit{BImpl}$ (which is the case since, as we have observed in Example 2.1.3, $\{\} \vdash B$ and $\{\} \vdash \mathit{BImpl}$ have the same denotational semantics), then $B \sqsubseteq^M_{[X \mapsto \kappa]} \mathit{BImpl}$ for all nonzero cardinals κ.

(In connection with this application of Theorem 5.3.6, see Example 4.2.1.)

If additionally $X \in \mathit{FinTypeVars}$, then the assumptions of Theorem 5.3.5 are satisfied with $\Gamma = \{\}$, $\mathit{Spec} = B$, $\mathit{Impl} = \mathit{BImpl}$ and any M, so in particular we have that $B \sqsubseteq^M_{[X \mapsto 2]} \mathit{BImpl}$ holds if and only if

$$\widetilde{B}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi} \sqsubseteq^M \widetilde{\mathit{BImpl}}^X \underset{\{*_{\widetilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

does. ■

Example 5.3.2 Recall the processes *Spec* and *Impl* from Example 5.0.2, where we have observed that *Spec* and *Impl* are X-DI and that $\mathit{Spec} \sqsubseteq_{\delta}^{M} \mathit{Impl}$ holds if and only if $|\delta[\![X]\!]| < K$.

By considering any $K \geq 3$, it follows that the requirement that $\Gamma \vdash \mathit{Spec}$ satisfies (**Norm**) can be omitted neither from Theorem 5.3.5 nor from Theorem 5.3.6. ■

Example 5.3.3 Consider the processes

$$\{\} \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$$
$$\{\} \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$$

where:

- $T = c.(X \times X)$ for some $X \in \mathit{NEFTypeVars} \cap \mathit{FinTypeVars}$;
- $\mathit{Spec} = c?x_1?x_2 \longrightarrow c\$y_1\$y_2 \longrightarrow \mathit{STOP}$
  $\mathit{Impl} = c?x_1'?x_2' \longrightarrow \mathit{STOP}$
- $$\begin{array}{ll} \phi(c,1) = \{?,\$\} & \phi(c,2) = \{?,\$\} \\ \psi(c,1) = \{?\} & \psi(c,2) = \{?\} \end{array}$$

If $M = \mathrm{Fa}$ or $M = \mathrm{FD}$, we have that

$$\widetilde{\mathit{Spec}}^{X} \underset{\{*_{\tilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi} \sqsubseteq^{M} \widetilde{\mathit{Impl}}^{X} \underset{\{*_{\tilde{T}^X}\}}{\|} \mathit{OneOne}_{X,T,\psi}$$

but that $\mathit{Spec} \not\sqsubseteq^{M}_{[X \mapsto \kappa]} \mathit{Impl}$ for all nonzero natural numbers κ.

Therefore, assumption (v) cannot be omitted from Theorem 5.3.5. ■

5.4 Sets of dependent sizes

In this section, as in the previous two sections, we shall be requiring *Spec* and *Impl* to be data independent with respect to some X and Γ and to have the same alphabet T, and we shall be focussing on interpretations of X by nonzero cardinals κ (see the beginning of Section 5.2).

The theorem we shall present requires that *Spec* satisfies ($\mathbf{RecCho}_X$), and it provides a cardinal λ such that, for any nonzero cardinal $\kappa \geq \lambda$, the collection of all interpretations of X and Γ which interpret X by κ is a threshold collection. More precisely, the collection contains only one interpretation of Γ for each equivalence relation on Γ, where the equivalence relation specifies the equality or inequality relationships among the constants from Γ. The cardinal λ depends on the structures of *Spec* and *Impl*, and on which model of CSP is used.

Another part of the theorem states that, if *Spec* is not refined by *Impl* for some interpretation of X and Γ from a threshold collection provided by the theorem, then the refinement fails for all interpretation with an appropriate property.

Furthermore, the theorem offers a possibility of considering only those interpretations of Γ which satisfy a given condition, where the condition is built from equality tests and boolean connectives. This feature is important in some applications, for example when the constants from Γ are indices of nodes in a ring, in which case it may be necessary to require them to be mutually distinct.

The assumptions in the theorem are therefore far weaker than the assumptions in either of the two theorems in the previous section (i.e. Theorems 5.3.5 and 5.3.6). In particular, there are no restrictions on equality tests in either *Spec* or *Impl*. However, the interpretations of X in the threshold collections provided by the theorem are of size at least λ (see above), whereas the two theorems in the previous section provide threshold collections in which the interpretations of X are of the constant size 2.

Since the only assumption in the theorem, in addition to data independence and having the same alphabet, is that *Spec* satisfies ($\mathbf{RecCho}_X$), it turns out that the Basic Lemma of Logical Relations (i.e. Theorem 4.2.2) is not an adequate tool for its proof. Instead, the theorem is proved by *eliminating symmetry* (see e.g. [Sta91, CE+96, ES96, ES95, ID96a, Jen96]).

More precisely, for any interpretation of X and Γ, the state space of the check of whether *Spec* is refined by *Impl* can be seen as a directed graph, so that the refinement holds if and only if all nodes which are reachable from the initial node of the graph have an appropriate property. Since *Spec* and *Impl* are data independent, any permutation of the interpretation of X induces an automorphism of the graph. The graph can then be quotiented by the group of all such automorphisms, which is what we mean by 'eliminating symmetry'.

The principal result in the proof of the theorem is that, for any two interpretations of X and Γ which interpret X by nonzero cardinals greater than or equal to λ and which induce the same equality or inequality relationships among the constants from Γ, the two corresponding graphs have the same quotients.

In fact, the proof contains direct definitions of those quotient graphs and the associated maps. That enables the statement of the theorem to offer a possibility of checking whether all nodes which are reachable from the initial node of an appropriate quotient graph have an appropriate property, instead of checking whether *Spec* is refined by *Impl* for all interpretations from a threshold collection. This feature is very significant for practical applications: see Section 5.6.3.

We shall first define the graphs which represent the state spaces of the checks of whether *Spec* is refined by *Impl*. We shall then show how symmetry can be eliminated from them, by defining the quotient graphs and the associated maps, and by presenting a number of lemmas about those. That will provide most of the proof of the theorem, which will then be presented. Finally, we shall give four examples.

5.4.1 Exploration graphs

The graphs which represent the state spaces of the checks of whether *Spec* is refined by *Impl* will be directed graphs with a unary predicate on their nodes. We shall call such structures 'exploration graphs':

Definition 5.4.1 $\mathcal{G}$ is an *exploration graph* if and only if it is a tuple

$$(\mathcal{N}, E_0, \mathit{Good}, \mathcal{T}, \mathit{source}, \mathit{target})$$

such that:

(i) $\mathcal{N}$ is a set. Its elements will be called 'nodes'.

(ii) $E_0 \in \mathcal{N}$. E_0 will be called 'the initial node'.

(iii) *Good* is a unary predicate on $\mathcal{N}$.

(iv) $\mathcal{T}$ is a set. Its elements will be called 'transitions'.

(v) For any $\epsilon \in \mathcal{T}$, $\mathit{source}(\epsilon)$ specifies its source node, and $\mathit{target}(\epsilon)$ specifies its target node.

■

Notation. We shall write $E\epsilon$ to mean that $\mathit{source}(\epsilon) = E$, and $\epsilon E'$ to mean that $\mathit{target}(\epsilon) = E'$. ■

In fact, in the present section and in the next one (i.e. Sections 5.4.1 and 5.4.2), we shall not be working with processes *Spec* and *Impl* directly, but with NSLTSs $\mathcal{NS}$ and SLTSs $\mathcal{S}$ such that the type context of the initial node of $\mathcal{S}$ is empty.

In particular, we are now going to define exploration graphs which represent state spaces of checks of whether a denotational semantic value corresponding to $\mathcal{NS}$ is refined by a denotational semantic value corresponding to $\mathcal{S}$ (see Sections 3.7.3 and 3.4).

For that purpose, the conditions $\mathit{NotStCond}(O)$ and $\mathit{DivCond}(O)$ which are provided by the following proposition will be useful. For any node O of $\mathcal{S}$ such that O has no conditional transitions, $\mathit{NotStCond}(O)$ is satisfied if and only if O is not stable, and $\mathit{DivCond}(O)$ is satisfied if and only if O can diverge.

Proposition 5.4.1 *Suppose that:*

(i) $\mathcal{S}$ is an SLTS with respect to some X, T and ψ such that $\mathit{Context}(O_0) = \{\}$, where O_0 is the initial node of $\mathcal{S}$;

(ii) O is a node of $\mathcal{S}$ such that it has no conditional transitions.

Then:

(I) there exists a condition $\mathit{NotStCond}(O)$ such that:

(I.a) $\mathit{NotStCond}(O)$ is either True, or False, or built from formulae of the form $x = x'$ with $(x : X), (x', X) \in \mathit{Context}(O)$ and the boolean connectives $\wedge$ and $\vee$;

(I.b) for any nonzero cardinal κ and any assignment ξ to $\mathit{Context}(O)$ with respect to $[X \mapsto \kappa]$, we have that ξ satisfies $\mathit{NotStCond}(O)$ if and only if there exist an invisible transition β of O and an assignment ξ' to $\mathit{ys}(\beta)$ with respect to $[X \mapsto \kappa]$ such that $\xi \cup \xi'$ satisfies $\mathit{cond}(\beta)$.

(II) there exists a condition $\mathit{DivCond}(O)$ such that:

(II.a) $\mathit{DivCond}(O)$ is either False, or of the form

$$(\mathit{conj}_1 \;\wedge\; |X| \geq \mu_1) \;\vee\; \cdots \;\vee\; (\mathit{conj}_n \;\wedge\; |X| \geq \mu_n)$$

(II.b) for any i, conj_i is either True, or built from formulae of the forms $x = x'$ and $\neg x = x'$ with $(x : X), (x' : X) \in \mathit{Context}(O)$ and the boolean connective $\wedge$;

(II.c) $\mathit{conj}_1, \ldots, \mathit{conj}_n$ are mutually disjoint, i.e. any assignment to $\mathit{Context}(O)$ satisfies at most one of them;

(II.d) for any i, μ_i is a nonzero cardinal such that $\mu_i \leq W(\mathcal{S}) + L_{\sqcap}(\mathcal{S})$ (see Section 3.3.3);

(II.e) for any nonzero cardinal κ and any assignment ξ to $\mathit{Context}(O)$ with respect to $[X \mapsto \kappa]$, we have that $[X \mapsto \kappa]$ and ξ satisfy $\mathit{DivCond}(O)$ (i.e. that there exists i such that ξ satisfies conj_i and $\kappa \geq \mu_i$) if and only if O^{ξ} div w.r.t. $[X \mapsto \kappa]$ (see Section 3.4.1).

Proof. Not difficult, by Definition 3.3.1. ■

The following definition of exploration graphs which represent state spaces of checks of whether a denotational semantic value corresponding to $\mathcal{NS}$ is refined by a denotational semantic value corresponding to $\mathcal{S}$ can be seen as the definition in the nonsymbolic setting (see Section 21.2.2 of [Ros94a], or Section C.1.3 of [Ros98a]) together with translations from the symbolic to the nonsymbolic setting. It is thus based on executing $\mathcal{NS}$ and $\mathcal{S}$ in parallel, where all visible events are synchronised on.

The definition could be presented by presenting the definition in the nonsymbolic case and the translations separately and then combining them. However, the presentation we give is more direct, for brevity and for a clearer connection with the definition of the quotient exploration graphs (see Section 5.4.2).

Definition 5.4.2 Suppose that:

(i) $\mathcal{NS}$ is an NSLTS with respect to some X, T and ϕ;

(ii) $\mathcal{S}$ is an SLTS with respect to X, T and some ψ such that $\mathit{Context}(O_0) = \{\}$, where O_0 is the initial node of $\mathcal{S}$;

(iii) M specifies one of the models Tr, Fa, FD of CSP;

(iv) if $X \in \mathit{FinTypeVars}$, then κ is a nonzero natural number, and otherwise κ is a nonzero cardinal.

Then an exploration graph

$$\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa} = (\mathcal{N}, E_0, \mathit{Good}, \mathcal{T}, \mathit{source}, \mathit{target})$$

is defined as follows:

(I) $\mathcal{N}$ is the set of all tuples (N, θ, O, ξ) such that:

(I.a) N is a node of $\mathcal{NS}$, and θ is an assignment of mutually distinct values to $\mathit{Context}(N)$ with respect to $[X \mapsto \kappa]$;

(I.b) O is a node of $\mathcal{S}$ which has no conditional transitions, and ξ is an assignment to $\mathit{Context}(O)$ with respect to $[X \mapsto \kappa]$;

(II) $E_0 = (N_0, \{\}, O_0', \{\})$, where N_0 is the initial node of $\mathcal{NS}$, and where either the initial node O_0 of $\mathcal{S}$ has no conditional transitions and $O_0' = O_0$, or O_0' is the unique node of $\mathcal{S}$ to which O_0 has a conditional transition $[\mathit{True}\rangle$ (see Definition 3.3.1 (vi) and (vii));

(III) for any $(N, \theta, O, \xi) \in \mathcal{N}$, $\mathcal{G}(N, \theta, O, \xi)$ holds if and only if:

case $M = \mathrm{Tr}$: for any visible transition β of O such that ξ satisfies $\mathit{cond}(\beta)$ (see under 'Visible' in Section 3.2.2 and see Proposition 5.0.1), and any assignment $\xi^\sharp$ to $\mathit{zs}(\beta)$ with respect to $[X \mapsto \kappa]$, there exist a transition α of N and an assignment $\theta^\natural$ to $\mathit{ys}(\alpha)$ with respect to $[X \mapsto \kappa]$ such that $\theta \cup \theta^\natural$ satisfies $\mathit{cond}(\alpha)$ and

$$[\![r(\alpha)]\!]_{[X\mapsto\kappa],\theta\cup\theta^\natural} = [\![r(\beta)]\!]_{[X\mapsto\kappa],\xi\cup\xi^\sharp}$$

(such α and $\theta^\natural$ are then unique, and $\mathit{Range}(\theta^\natural) \subseteq \mathit{Range}(\xi \cup \xi^\sharp)$: see under 'Normalised symbolic transitions' (iii) in Section 3.6.1 and see Definition 3.7.1 (vi));

case $M = \mathrm{Fa}$**:** the condition for the case $M = \mathrm{Tr}$ is satisfied, and if ξ does not satisfy $\mathit{NotStCond}(O)$, then there must exist some $\mathcal{A} \in \mathit{AccSets}(N)$ such that, for any $\alpha \in \mathcal{A}$,

(III.a) either there exists an assignment $\theta^\natural$ to $\mathit{ys}(\alpha)$ with respect to $[X \mapsto \kappa]$ such that $\theta \cup \theta^\natural$ does not satisfy $\mathit{cond}(\alpha)$,

(III.b) or there exists an assignment $\theta^\natural$ to $\mathit{ys}(\alpha)$ with respect to $[X \mapsto \kappa]$ such that $\theta \cup \theta^\natural$ satisfies $\mathit{cond}(\alpha)$ and

$$\{[\![r(\alpha)]\!]_{[X\mapsto\kappa],\theta\cup\theta^\natural\cup\theta^\sharp} \mid \theta^\sharp \textit{ is an assignment to } \mathit{zs}(\alpha) \textit{ w.r.t. } [X \mapsto \kappa]\} \subseteq$$
$$\left\{ [\![r(\beta)]\!]_{[X\mapsto\kappa],\xi\cup\xi^\sharp} \;\middle|\; \begin{array}{l} \beta \textit{ is a vis. trans. of } O \\ \wedge\, \xi \textit{ satisfies } \mathit{cond}(\beta) \\ \wedge\, \xi^\sharp \textit{ is an assignment to } \mathit{zs}(\beta) \textit{ w.r.t. } [X \mapsto \kappa] \end{array} \right\}$$

case $M = \mathrm{FD}$**:** either $\mathit{DivCond}(N) = \mathit{True}$, or $[X \mapsto \kappa]$ and ξ do not satisfy $\mathit{DivCond}(O)$ and the condition for the case $M = \mathrm{Fa}$ is satisfied;

(IV) $\mathcal{T}$ is the set of all tuples $(\alpha, \theta, \beta, \xi, \xi^\sharp)$ such that:

(IV.a) α is a transition in $\mathcal{NS}$, and θ is an assignment of mutually distinct values to $\mathit{Context}(\mathit{source}(\alpha))$ with respect to $[X \mapsto \kappa]$;

(IV.b) β is a visible transition in $\mathcal{S}$, ξ is an assignment to $\mathit{Context}(\mathit{source}(\beta))$ with respect to $[X \mapsto \kappa]$ such that ξ satisfies $\mathit{cond}(\beta)$, and $\xi^\sharp$ is an assignment to $\mathit{zs}(\beta)$ with respect to $[X \mapsto \kappa]$;

(IV.c) there exists an assignment $\theta^\natural$ to $\mathit{ys}(\alpha)$ with respect to $[X \mapsto \kappa]$ such that $\theta \cup \theta^\natural$ satisfies $\mathit{cond}(\alpha)$ and

$$[\![r(\alpha)]\!]_{[X\mapsto\kappa],\theta\cup\theta^\natural} = [\![r(\beta)]\!]_{[X\mapsto\kappa],\xi\cup\xi^\sharp}$$

(as in (III) above, such $\theta^\natural$ is then unique, and $\mathit{Range}(\theta^\natural) \subseteq \mathit{Range}(\xi \cup \xi^\sharp)$);

and of all tuples $(N, \theta, \beta, \xi, \xi^\natural)$ such that:

(IV.d) N is a node of $\mathcal{NS}$, and θ is an assignment of mutually distinct values to $\mathit{Context}(N)$ with respect to $[X \mapsto \kappa]$;

(IV.e) β is an invisible transition in $\mathcal{S}$, ξ is an assignment to $\mathit{Context}(\mathit{source}(\beta))$ with respect to $[X \mapsto \kappa]$, and $\xi^\natural$ is an assignment to $\mathit{ys}(\beta)$ with respect to $[X \mapsto \kappa]$ such that $\xi \cup \xi^\natural$ satisfies $\mathit{cond}(\beta)$;

(V) for any $(\alpha, \theta, \beta, \xi, \xi^\sharp) \in \mathcal{T}$,

$$\begin{array}{rcl} \mathit{source}(\alpha,\theta,\beta,\xi,\xi^\sharp) & = & (\mathit{source}(\alpha), \theta, \mathit{source}(\beta), \xi) \\ \mathit{target}(\alpha,\theta,\beta,\xi,\xi^\sharp) & = & (\mathit{target}(\alpha,i), (\theta \cup \theta^\natural) \restriction \mathit{Context}(\mathit{target}(\alpha,i)), \\ & & \mathit{target}(\beta'), (\xi \cup \xi^\sharp) \restriction \mathit{Context}(\mathit{target}(\beta'))) \end{array}$$

where:

(V.a) $\theta^\natural$ is as in (IV.c) above;

(V.b) i is such that $\theta \cup \theta^\natural$ satisfies $\mathit{cond}'_i(\alpha)$ (see under 'Normalised symbolic transitions' (IV) in Section 3.6.1);

(V.c) either $\mathit{target}(\beta)$ has no conditional transitions and $\beta' = \beta$, or β' is a conditional transition of $\mathit{target}(\beta)$ such that $\xi \cup \xi^\sharp$ satisfies $\mathit{cond}(\beta')$ (see Definition 3.3.1 (vi));

and, for any $(N, \theta, \beta, \xi, \xi^{\natural}) \in \mathcal{T}$,

$$\begin{array}{rcl} \mathit{source}(N, \theta, \beta, \xi, \xi^{\natural}) & = & (N, \theta, \mathit{source}(\beta), \xi) \\ \mathit{target}(N, \theta, \beta, \xi, \xi^{\natural}) & = & (N, \theta, \mathit{target}(\beta'), (\xi \cup \xi^{\natural}) \restriction \mathit{Context}(\mathit{target}(\beta'))) \end{array}$$

where:

(V.d) either $\mathit{target}(\beta)$ has no conditional transitions and $\beta' = \beta$, or β' is a conditional transition of $\mathit{target}(\beta)$ such that $\xi \cup \xi^{\natural}$ satisfies $\mathit{cond}(\beta')$ (see Definition 3.3.1 (vi)).

■

The following proposition confirms that the exploration graphs defined in Definition 5.4.2 do represent state spaces of checks of whether a denotational semantic value corresponding to $\mathcal{NS}$ is refined by a denotational semantic value corresponding to $\mathcal{S}$. More precisely, the proposition states that, for any node (N, θ, O, ξ) of such an exploration graph, the predicate *Good* holds at all nodes reachable from (N, θ, O, ξ) if and only if the denotational semantic value corresponding to N and θ is refined by the denotational semantic value corresponding to O and ξ.

Proposition 5.4.2 *Suppose that:*

(i) $\mathcal{NS}$ *is an NSLTS with respect to some* X, T *and* ϕ*;*

(ii) $\mathcal{S}$ *is an SLTS with respect to* X, T *and some* ψ *such that* $\mathit{Context}(O_0) = \{\}$, *where* O_0 *is the initial node of* $\mathcal{S}$*;*

(iii) M *specifies one of the models Tr, Fa, FD of CSP;*

(iv) if $X \in \mathit{FinTypeVars}$, *then* κ *is a nonzero natural number, and otherwise* κ *is a nonzero cardinal;*

(v) (N, θ, O, ξ) *is a node of* $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$.

Then $\mathit{Good}(E)$ *holds for all nodes* E *which are reachable from* (N, θ, O, ξ) *(by zero or more transitions) if and only if:*

case $M = \mathrm{Tr}$: $\mathit{traces}(N)_{[X \mapsto \kappa],\theta} \supseteq \mathit{traces}(O)_{[X \mapsto \kappa],\theta}$

case $M = \mathrm{Fa}$: $\mathit{traces}(N)_{[X \mapsto \kappa],\theta} \supseteq \mathit{traces}(O)_{[X \mapsto \kappa],\theta}$
$\wedge\ \mathit{failures}(N)_{[X \mapsto \kappa],\theta} \supseteq \mathit{failures}(O)_{[X \mapsto \kappa],\theta}$

case $M = \mathrm{FD}$: $\mathit{failures}_{\perp}(N)_{[X \mapsto \kappa],\theta} \supseteq \mathit{failures}_{\perp}(O)_{[X \mapsto \kappa],\theta}$
$\wedge\ \mathit{divergences}(N)_{[X \mapsto \kappa],\theta} \supseteq \mathit{divergences}(O)_{[X \mapsto \kappa],\theta}$

Proof. Straightforward, by Definition 5.4.2. (For a sketch of a proof in the nonsymbolic case, see Section 21.2.2 of [Ros94a] or Section C.1.3 of [Ros98a].) ■

5.4.2 Eliminating symmetry

In Definition 5.4.2, we defined exploration graphs $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$ which represent state spaces of checks of whether the denotational semantic value $[\![\mathcal{NS}]\!]^M_{[X\mapsto\kappa],\{\}}$ is refined by the denotational semantic value $[\![\mathcal{S}]\!]^M_{[X\mapsto\kappa],\{\}}$.

We are now going to define an exploration graph $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$, and maps $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ which are from $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$ into $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$. Two nodes (respectively, transitions) of any $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$ will be mapped by $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ to the same node (transition) of $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$ if and only if they can be obtained one from another by a permutation of the set

$$\{\alpha \mid \alpha \textit{ is an ordinal } \wedge \alpha < \kappa\}$$

Thus, in the sense of the remarks at the beginning of Section 5.4, the maps $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ will eliminate the symmetry from the exploration graphs $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$.

The definition of $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$ will be essentially the same as the definition of any $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$, except that equivalence relations on variables of type X will be used instead of assignments with respect to $[X \mapsto \kappa]$. The definition of any $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ will amount to replacing assignments with respect to $[X \mapsto \kappa]$ by equivalence relations which they induce.

We shall then present four lemmas. They provide sufficient conditions for a map $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ to be into, onto, and to preserve initial nodes, *Good*, *source*, and *target*. In particular, the lemmas imply that, for any nonzero cardinal κ which is no less than a cardinal which corresponds to the cardinal λ in the remarks at the beginning of Section 5.4, $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ is a homomorphism of exploration graphs, and $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$ is a quotient of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$ with respect to it.

Some notation and terminology.

(i) If A_1, …, A_k are sets, let

$$A_1 + \cdots + A_k = \{(i,a) \mid i \in \{1,\ldots,k\} \ \wedge \ a \in A_i\}$$

(ii) If π is a binary relation on a set A, and if $A \subseteq B$, we shall say that σ *is an extension of* π *to* B if and only if σ is a binary relation on B such that

$$\forall\, a, a' \in A.a\sigma a' \Leftrightarrow a\pi a'$$

(iii) If Γ is a type context which satisfies the restriction of data independence in some X (see Section 2.7.1), if π is an equivalence relation on Γ, and if *cond* is either *True*, or *False*, or built from equality tests between variables in Γ and the boolean connectives $\neg$, $\wedge$ and $\vee$, then we shall say that π *satisfies cond* if and only if *cond* holds when the equality symbol is interpreted by π.

(iv) If Γ is a type context which satisfies the restriction of data independence in some X (see Section 2.7.1), if π is an equivalence relation on Γ, and if $H[x_1,\ldots,x_h]$ and $H'[x'_1,\ldots,x'_{h'}]$ are two X,Γ-DI evaluated forms of some type T which satisfies $T^{\mathit{AllTypeVars},+,\times,\mu}$ (see Section 3.1.2), then we shall say that π *satisfies*

$$H[x_1,\ldots,x_h] = H'[x'_1,\ldots,x'_{h'}]$$

if and only if $H = H'$ (so that in particular $h = h'$) and π satisfies $x_i = x'_i$ (see (iii) above) for all $i \in \{1,\ldots,h\}$. ■

Definition 5.4.3 Suppose that:

(i) $\mathcal{NS}$ is an NSLTS with respect to some X, T and ϕ;

(ii) $\mathcal{S}$ is an SLTS with respect to X, T and some ψ such that $\mathit{Context}(O_0) = \{\}$, where O_0 is the initial node of $\mathcal{S}$;

(iii) M specifies one of the models Tr, Fa, FD of CSP.

Then an exploration graph

$$\mathcal{H}_{\mathcal{NS},\mathcal{S},M} = (\mathcal{N}, E_0, \mathit{Good}, \mathcal{T}, \mathit{source}, \mathit{target})$$

is defined as follows:

(I) $\mathcal{N}$ is the set of all tuples (N, O, π) such that:

(I.a) N is a node of $\mathcal{NS}$;

(I.b) O is a node of $\mathcal{S}$ which has no conditional transitions;

(I.c) π is an equivalence relation on

$$\mathit{Context}(N) + \mathit{Context}(O)$$

such that no two variables from $\mathit{Context}(N)$ belong to the same equivalence class;

(II) $E_0 = (N_0, O'_0, \{\})$, where N_0 is the initial node of $\mathcal{NS}$, and where either the initial node O_0 of $\mathcal{S}$ has no conditional transitions and $O'_0 = O_0$, or O'_0 is the unique node of $\mathcal{S}$ to which O_0 has a conditional transition $[\mathit{True}\rangle$ (see Definition 3.3.1 (vi) and (vii));

(III) for any $(N, O, \pi) \in \mathcal{N}$, $\mathcal{G}(N, O, \pi)$ holds if and only if:

case $M = \mathrm{Tr}$**:** for any visible transition β of O such that π satisfies $\mathit{cond}(\beta)$ (i.e. such that σ satisfies $\mathit{cond}(\beta)$, where σ is the equivalence relation on $\mathit{Context}(O)$ defined by $(x : X)\sigma(x' : X) \Leftrightarrow (2, (x : X))\pi(2, (x' : X))$: see under 'Visible' in Section 3.2.2 and see Proposition 5.0.1), and any equivalence relation $\pi^{\sharp}$ on

$$\mathit{Context}(N) + (\mathit{Context}(O) \cup \mathit{zs}(\beta))$$

which is an extension of π, there exist a transition α of N and an equivalence relation $\pi^{\sharp\natural}$ on

$$(\mathit{Context}(N) \cup \mathit{ys}(\alpha)) + (\mathit{Context}(O) \cup \mathit{zs}(\beta))$$

which is an extension of $\pi^{\sharp}$ such that $\pi^{\sharp\natural}$ satisfies $\mathit{cond}(\alpha)$ and $\pi^{\sharp\natural}$ satisfies $r(\alpha) = r(\beta)$ (such α and $\pi^{\sharp\natural}$ are then unique, and any variable from $\mathit{ys}(\alpha)$ is related by $\pi^{\sharp\natural}$ to some variable from $\mathit{Context}(O) \cup \mathit{zs}(\beta)$: see under 'Normalised symbolic transitions' (iii) in Section 3.6.1 and see Definition 3.7.1 (vi));

case $M = \mathrm{Fa}$**:** the condition for the case $M = \mathrm{Tr}$ is satisfied, and if π does not satisfy $\mathit{NotStCond}(O)$, then there must exist some $\mathcal{A} \in \mathit{AccSets}(N)$ such that, for any $\alpha \in \mathcal{A}$,

(III.a) either $\mathit{cond}(\alpha)$ contains some variable from $\mathit{ys}(\alpha)$,

(III.b) or $\mathit{cond}(\alpha)$ does not contain any variable from $\mathit{ys}(\alpha)$ and π does not satisfy $\mathit{cond}(\alpha)$, (III.c) or $cond(\alpha)$ does not contain any variable from $ys(\alpha)$, π satisfies $cond(\alpha)$, and there exists a visible transition β of O such that the following hold, where $r(\alpha) = H[x_1, \ldots, x_h]$ and $r(\beta) = H'[x'_1, \ldots, x'_{h'}]$ (see Section 3.1.2):

(III.c.1) π satisfies $cond(\beta)$;

(III.c.2) $H = H'$ (so that in particular $h = h'$);

(III.c.3) for any $i \in \{1, \ldots, h\}$, if $(x_i : X) \in Context(N)$ and $(x'_i : X) \in Context(O)$, then π satisfies $x_i = x'_i$;

(III.c.4) for any $i \in \{1, \ldots, h\}$, if $(x_i : X) \in zs(\alpha)$, then $(x'_i : X) \in zs(\beta)$;

case $M =$ FD**:** either $DivCond(N) = True$, or π does not satisfy any of the subformulae $conj_i$ of $DivCond(O)$ (see Proposition 5.4.1) and the condition for the case $M =$ Fa is satisfied;

(IV) $\mathcal{T}$ is the set of all tuples $(\alpha, \beta, \pi^\sharp)$ such that:

(IV.a) α is a transition in $\mathcal{NS}$;

(IV.b) β is a visible transition in $\mathcal{S}$;

(IV.c) $\pi^\sharp$ is an equivalence relation on

$$Context(source(\alpha)) + (Context(source(\beta)) \cup zs(\beta))$$

such that no two variables from $Context(source(\alpha))$ belong to the same equivalence class, and $\pi^\sharp$ satisfies $cond(\beta)$;

(IV.d) there exists an equivalence relation $\pi^{\sharp\natural}$ on

$$(Context(source(\alpha)) \cup ys(\alpha)) + (Context(source(\beta)) \cup zs(\beta))$$

which is an extension of $\pi^\sharp$ such that $\pi^{\sharp\natural}$ satisfies $cond(\alpha)$ and $\pi^{\sharp\natural}$ satisfies $r(\alpha) = r(\beta)$ (as in (III) above, such $\pi^{\sharp\natural}$ is then unique, and any variable from $ys(\alpha)$ is related by $\pi^{\sharp\natural}$ to some variable from $Context(source(\beta)) \cup zs(\beta)$);

and of all tuples $(N, \beta, \pi^\natural)$ such that:

(IV.e) N is a node of $\mathcal{NS}$;

(IV.f) β is an invisible transition in $\mathcal{S}$;

(IV.g) $\pi^\natural$ is an equivalence relation on

$$Context(N) + (Context(source(\beta)) \cup ys(\beta))$$

such that no two variables from $Context(N)$ belong to the same equivalence class, and $\pi^\natural$ satisfies $cond(\beta)$;

(V) for any $(\alpha, \beta, \pi^\sharp) \in \mathcal{T}$,

$$\begin{aligned} source(\alpha, \beta, \pi^\sharp) &= (source(\alpha), source(\beta), \pi^\sharp \cap ((source(\alpha) + source(\beta))^2)) \\ target(\alpha, \beta, \pi^\sharp) &= (target(\alpha, i), target(\beta'), \pi^{\sharp\natural} \cap ((target(\alpha, i) + target(\beta'))^2)) \end{aligned}$$

where:

(V.a) $\pi^{\sharp\natural}$ is as in (IV.d) above;

(V.b) i is such that $\pi^{\sharp\natural}$ satisfies $cond'_i(\alpha)$ (see under 'Normalised symbolic transitions' (IV) in Section 3.6.1);

(V.c) either $target(\beta)$ has no conditional transitions and $\beta' = \beta$, or β' is a conditional transition of $target(\beta)$ such that $\pi^\sharp$ satisfies $cond(\beta')$ (see Definition 3.3.1 (vi));

and, for any $(N, \beta, \pi^\natural) \in \mathcal{T}$,

$$\begin{aligned} source(N, \beta, \pi^\natural) &= (N, source(\beta), \pi^\natural \cap ((N + source(\beta))^2)) \\ target(N, \beta, \pi^\natural) &= (N, target(\beta'), \pi^\natural \cap ((N + target(\beta'))^2)) \end{aligned}$$

where:

(V.d) either $target(\beta)$ has no conditional transitions and $\beta' = \beta$, or β' is a conditional transition of $target(\beta)$ such that $\pi^\natural$ satisfies $cond(\beta')$ (see Definition 3.3.1 (vi)).

■

Definition 5.4.4 Suppose that:

(i) $\mathcal{NS}$ is an NSLTS with respect to some X, T and ϕ;

(ii) $\mathcal{S}$ is an SLTS with respect to X, T and some ψ such that $Context(O_0) = \{\}$, where O_0 is the initial node of $\mathcal{S}$;

(iii) M specifies one of the models Tr, Fa, FD of CSP;

(iv) if $X \in FinTypeVars$, then κ is a nonzero natural number, and otherwise κ is a nonzero cardinal.

Then a map $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ from nodes and transitions of the exploration graph $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$ (see Definition 5.4.2) is defined as follows:

(I) for any node (N, θ, O, ξ) of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$, let

$$\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(N, \theta, O, \xi) = (N, O, \pi)$$

where π is the equivalence relation on

$$Context(N) + Context(O)$$

induced by θ and ξ, i.e.

$$\begin{aligned} (1, (x : X))\pi(1, (x' : X)) &\Leftrightarrow \theta[\![x]\!] = \theta[\![x']\!] \\ (1, (x : X))\pi(2, (x' : X)) &\Leftrightarrow \theta[\![x]\!] = \xi[\![x']\!] \\ (2, (x : X))\pi(2, (x' : X)) &\Leftrightarrow \xi[\![x]\!] = \xi[\![x']\!] \end{aligned}$$

(II) for any transition $(\alpha, \theta, \beta, \xi, \xi^\sharp)$ of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$, let

$$\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(\alpha, \theta, \beta, \xi, \xi^\sharp) = (\alpha, \beta, \pi^\sharp)$$

where $\pi^\sharp$ is the equivalence relation on

$$Context(source(\alpha)) + (Context(source(\beta)) \cup zs(\beta))$$

induced by θ, ξ and $\xi^\sharp$;

(III) for any transition $(N, \theta, \beta, \xi, \xi^\natural)$ of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$, let

$$\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(N, \theta, \beta, \xi, \xi^\natural) = (N, \beta, \pi^\natural)$$

where $\pi^\natural$ is the equivalence relation on

$$\mathit{Context}(N) + (\mathit{Context}(\mathit{source}(\beta)) \cup \mathit{ys}(\beta))$$

induced by θ, ξ and $\xi^\natural$.

■

Lemma 5.4.3 *Suppose that $\mathcal{NS}$, $\mathcal{S}$, M and κ are as in Definition 5.4.4.*
Then we have that:

(I) for any node E of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$, $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E)$ is a node of $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$;

(II) for any transition ϵ of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$, $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(\epsilon)$ is a transition of $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$;

(III) $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}$ preserves initial nodes, source and target.

Proof. Straightforward, by Definitions 5.4.2, 5.4.3 and 5.4.4. ■

Lemma 5.4.4 *Suppose that $\mathcal{NS}$, $\mathcal{S}$, M and κ are as in Definition 5.4.4.*
Then we have that:

(I) if $\kappa \geq W(\mathcal{NS}) + W(\mathcal{S})$, then for any node E' of $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$, there exists a node E of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$ such that $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E) = E'$;

(II) if $\kappa \geq W(\mathcal{NS}) + W(\mathcal{S}) + \max\{L_?(\mathcal{S}), L_\sqcap(\mathcal{S})\}$, then for any node E of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$ and any transition ϵ' of $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E)$, there exists a transition ϵ of E such that $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(\epsilon) = \epsilon'$.

Proof. (I) follows by observing that, for any node (N, O, π) of $\mathcal{H}_{\mathcal{NS},\mathcal{S},M}$, we have that $|\mathit{Context}(N)| \leq W(\mathcal{NS})$ and $|\mathit{Context}(O)| \leq W(\mathcal{S})$, so that π has at most $W(\mathcal{NS}) + W(\mathcal{S})$ equivalence classes.

(II) follows similarly, although it is more complex. In particular, it involves recalling the remarks about $\pi^{\sharp\natural}$ in Definition 5.4.3 (IV.d). ■

Lemma 5.4.5 *Suppose that $\mathcal{NS}$, $\mathcal{S}$, M and κ are as in Definition 5.4.4, and that E is a node of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$.*
Then we have that:

case $M = \mathrm{Tr}$: *if $\mathit{Good}(\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E))$, then $\mathit{Good}(E)$;*

cases $M = \mathrm{Fa}$ **and** $M = \mathrm{FD}$: *if $\kappa \geq 2$ and $\mathit{Good}(\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E))$, then $\mathit{Good}(E)$.*

Proof. We shall consider only the case $M = \mathrm{FD}$, which is the most complex one.

So suppose that $\kappa \geq 2$ and $\mathit{Good}(\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E))$. Let N, θ, O, ξ and π be such that $E = (N, \theta, O, \xi)$ and $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E) = (N, O, \pi)$.

If $\mathit{DivCond}(N) = \mathit{True}$, then $\mathit{Good}(E)$ is immediate by Definition 5.4.2 (III).

Otherwise, since $\mathit{Good}(\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E))$, we have that π does not satisfy any of the subformulae conj_i of $\mathit{DivCond}(O)$ (see Proposition 5.4.1) and the condition for the case $M = \mathrm{Fa}$ in Definition 5.4.3 (III) is satisfied. Hence ξ does not satisfy any of the subformulae conj_i of $\mathit{DivCond}(O)$, so in order to show that $\mathit{Good}(E)$, it remains to show that the condition for the case $M = \mathrm{Fa}$ in Definition 5.4.2 (III) is satisfied.

Since we know that the condition for the case $M = \mathrm{Tr}$ in Definition 5.4.3 (III) is satisfied, it easily follows that the condition for the case $M = \mathrm{Tr}$ in Definition 5.4.2 (III) is also satisfied. So suppose that ξ does not satisfy $\mathit{NotStCond}(O)$. Then π does not satisfy $\mathit{NotStCond}(O)$, which means that there must exist some $\mathcal{A} \in \mathit{AccSets}(N)$ such that, for any $\alpha \in \mathcal{A}$, one of Definition 5.4.3 (III.a)–(III.c) is satisfied.

If Definition 5.4.3 (III.a) is satisfied, let $(y : X) \in \mathit{ys}(\alpha)$ be such that $\mathit{cond}(\alpha)$ contains y. Since $\kappa \geq 2$, we can use y to obtain an assignment $\theta^\natural$ to $\mathit{ys}(\alpha)$ with respect to $[X \mapsto \kappa]$ such that $\theta \cup \theta^\natural$ does not satisfy $\mathit{cond}(\alpha)$.

If Definition 5.4.3 (III.b) is satisfied, then $\theta \cup \theta^\natural$ does not satisfy $\mathit{cond}(\alpha)$ for any assignment $\theta^\natural$ to $\mathit{ys}(\alpha)$ with respect to $[X \mapsto \kappa]$.

If Definition 5.4.3 (III.c) is satisfied, let β, $H[x_1, \ldots, x_h]$ and $H'[x'_1, \ldots, x'_{h'}]$ be as in Definition 5.4.3 (III.c). Since $\mathit{cond}(\alpha)$ does not contain any variable from $\mathit{ys}(\alpha)$, we have that each variable from $\mathit{ys}(\alpha)$ is exactly one x_i (see under 'Visible' in Section 3.2.2). For any $i \in \{1, \ldots, h\}$ such that $(x_i : X) \in \mathit{ys}(\alpha)$, if $(x'_i : X) \in \mathit{Context}(O)$, then let $\theta^\natural[\![x_i]\!] = \xi[\![x'_i]\!]$, and otherwise let $\theta^\natural[\![x_i]\!]$ be an arbitrary element of the set κ. Then $\theta \cup \theta^\natural$ satisfies $\mathit{cond}(\alpha)$ (since $\mathit{cond}(\alpha)$ does not contain any variable from $\mathit{ys}(\alpha)$ and π satisfies $\mathit{cond}(\alpha)$), ξ satisfies $\mathit{cond}(\beta)$ (since π satisfies $\mathit{cond}(\beta)$), and it is straightforward to check that

$$\begin{aligned}&\{[\![r(\alpha)]\!]_{[X\mapsto\kappa],\theta\cup\theta^\natural\cup\theta^\sharp} \mid \theta^\sharp \textit{ is an assignment to } zs(\alpha) \textit{ w.r.t. } [X \mapsto \kappa]\} \subseteq \\ &\quad \{[\![r(\beta)]\!]_{[X\mapsto\kappa],\xi\cup\xi^\sharp} \mid \xi^\sharp \textit{ is an assignment to } zs(\beta) \textit{ w.r.t. } [X \mapsto \kappa]\}\end{aligned}$$

Therefore, for any of the three possibilities, we have shown that one of Definition 5.4.2 (III.a) or (III.b) holds, completing the proof that $\mathit{Good}(E)$. ■

Lemma 5.4.6 *Suppose that $\mathcal{NS}$, $\mathcal{S}$, M and κ are as in Definition 5.4.4, and that E is a node of $\mathcal{G}_{\mathcal{NS},\mathcal{S},M,\kappa}$.*

If $\kappa \geq W(\mathcal{NS}) + W(\mathcal{S}) + \max\{L_?(\mathcal{S}), L_\sqcap(\mathcal{S})\}$ and $\mathit{Good}(E)$, then $\mathit{Good}(\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E))$.

Proof. We shall consider only the case $M = \mathrm{FD}$, which is the most complex one.

So suppose that $\kappa \geq W(\mathcal{NS}) + W(\mathcal{S}) + \max\{L_?(\mathcal{S}), L_\sqcap(\mathcal{S})\}$ and $\mathit{Good}(E)$. Let N, θ, O, ξ and π be such that $E = (N, \theta, O, \xi)$ and $\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E) = (N, O, \pi)$.

If $\mathit{DivCond}(N) = \mathit{True}$, then $\mathit{Good}(\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E))$ is immediate by Definition 5.4.3 (III).

Otherwise, we have that $[X \mapsto \kappa]$ and ξ do not satisfy $\mathit{DivCond}(O)$ and that the condition for the case $M = \mathrm{Fa}$ in Definition 5.4.2 (III) is satisfied. Since

$$\kappa \geq W(\mathcal{NS}) + W(\mathcal{S}) + \max\{L_?(\mathcal{S}), L_\sqcap(\mathcal{S})\} \geq W(\mathcal{S}) + L_\sqcap(\mathcal{S}) \geq \mu_i$$

for any of the nonzero cardinals μ_i in the condition $\mathit{DivCond}(O)$ (see Proposition 5.4.1), we have that ξ does not satisfy any of the subformulae conj_i of $\mathit{DivCond}(O)$. Hence π does not satisfy any of the subformulae conj_i of $\mathit{DivCond}(O)$, so that it remains to show that the condition for the case $M = \mathrm{Fa}$ in Definition 5.4.3 (III) is satisfied.

Since the condition for the case $M = \mathrm{Tr}$ in Definition 5.4.2 (III) is satisfied and $\kappa \geq W(\mathcal{NS}) + W(\mathcal{S}) + \max\{L_?(\mathcal{S}), L_\sqcap(\mathcal{S})\}$, it is straightforward to see that the condition for the case $M = \mathrm{Tr}$ in Definition 5.4.3 (III) is also satisfied (which involves recalling the remarks about α and $\theta^\natural$ made at the end of the case $M = \mathrm{Tr}$ in Definition 5.4.2 (III)). So suppose that π does not satisfy $\mathit{NotStCond}(O)$. Then also ξ does not satisfy $\mathit{NotStCond}(O)$, and hence there exists some $\mathcal{A} \in \mathit{AccSets}(N)$ such that, for any $\alpha \in \mathcal{A}$, one of Definition 5.4.2 (III.a) or (III.b) is satisfied.

If Definition 5.4.2 (III.a) is satisfied, then it is clear that one of Definition 5.4.3 (III.a) or (III.b) is satisfied.

If Definition 5.4.2 (III.b) is satisfied, let $\theta^\natural$ be as in Definition 5.4.2 (III.b). We can assume that $\mathit{cond}(\alpha)$ does not contain any variable from $\mathit{ys}(\alpha)$ and that π satisfies $\mathit{cond}(\alpha)$, since otherwise one of Definition 5.4.3 (III.a) or (III.b) would be satisfied. For a contradiction, suppose that there does not exist a visible transition β of O such that Definition 5.4.3 (III.c.1)–(III.c.4) hold.

If $L_?(\mathcal{S}) = 0$ and $L_\sqcap(\mathcal{S}) = 0$, then also $W(\mathcal{S}) = 0$, and so the set inclusion in Definition 5.4.2 (III.b) simplifies to

$$[\![r(\alpha)]\!] \in \{[\![r(\beta)]\!] \mid \beta \textit{ is a vis. trans. of } O\}$$

which gives us a visible transition β of O such that Definition 5.4.3 (III.c.1)–(III.c.4) hold, producing a contradiction as required.

Otherwise, we have that $\kappa \geq W(\mathcal{NS}) + W(\mathcal{S}) + \max\{L_?(\mathcal{S}), L_\sqcap(\mathcal{S})\} > W(\mathcal{S})$. Therefore, for any $(z : X) \in \mathit{zs}(\alpha)$, we can define $\theta^\sharp[\![z]\!]$ to be an element of the set κ which is distinct from any $\xi[\![x]\!]$, where $(x : X) \in \mathit{Context}(O)$ and there exists a visible transition β of O such that Definition 5.4.3 (III.c.1)—(III.c.3) are satisfied and x occurs in $r(\beta)$ in the place in which z occurs in $r(\alpha)$. It is then straightforward to see that

$$\begin{aligned}&[\![r(\alpha)]\!]_{[X\mapsto\kappa],\theta\cup\theta^\natural\cup\theta^\sharp} \not\in \\ &\quad\left\{ [\![r(\beta)]\!]_{[X\mapsto\kappa],\xi\cup\xi^\sharp} \;\middle|\; \begin{array}{l} \beta \textit{ is a vis. trans. of } O \\ \wedge\, \xi \textit{ satisfies } \mathit{cond}(\beta) \\ \wedge\, \xi^\sharp \textit{ is an assignment to } \mathit{zs}(\beta) \textit{ w.r.t. } [X \mapsto \kappa] \end{array} \right\}\end{aligned}$$

which contradicts the set inclusion in Definition 5.4.2 (III.b).

Hence, there does exist a visible transition β of O such that Definition 5.4.3 (III.c.1)–(III.c.4) hold, completing the proof that $\mathit{Good}(\Phi_{\mathcal{NS},\mathcal{S},M,\kappa}(E))$. ■

5.4.3 The theorem

We are now going to present the theorem which Section 5.4 is devoted to. It will require that *Spec* and *Impl* are data independent with respect to some X and Γ, that they have the same alphabet, and that *Spec* satisfies ($\mathbf{RecCho}_X$), and it will provide threshold collections for the problem of whether *Spec* is refined by *Impl*. (For more remarks about the theorem, see the beginning of Section 5.4.)

Boolean conditions on constants. If Γ is of the form $\{x_1 : X, \ldots, x_k : X\}$, we shall say that γ is a *boolean condition on* Γ if γ is either *True*, or *False*, or built from equality tests of the form $x_i = x_j$ and the boolean connectives $\neg$, $\wedge$ and $\vee$. ■

Theorem 5.4.7 *Suppose that:*

(i) $\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$ *and* $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ *are* X,Γ*-DI, where* Γ *is of the form* $\{x_1 : X, \ldots, x_k : X\}$, *and* T *is of the form* $\sum_{i=1}^{n} id_i . \prod_{j=1}^{m_i} T_{i,j}$;

(ii) M *specifies one of the models Tr, Fa, FD of CSP;*

(iii) $\Gamma \vdash \mathit{Spec}$ *satisfies* $(\mathbf{RecCho}_X)$;

(iv) γ *is a boolean condition on* Γ.

Let id_{n+1} *be an identifier which is distinct from each of* $id_1, \ldots, id_n$. *For any equivalence relation* σ *on* Γ, *let* E_σ *be the unique node of*

$$\mathcal{H}_{\mathcal{NS}(\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Spec,id_{n+1})}),\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Impl,id_{n+1})},M}$$

(see Definition 5.3.3) which is reachable from the initial node by some k *transitions*

$$(\alpha_1, \beta_1, \pi_1^\sharp), \ldots, (\alpha_k, \beta_k, \pi_k^\sharp)$$

(see under 'Assumption of transformation' before Proposition 3.5.4) which observe σ, *i.e. such that*

$$\forall i, j \in \{1, \ldots, k\}. (i < j \wedge (z_i : X) \in \mathit{Context}(\mathit{source}(\beta_j))) \Rightarrow \\ ((2, (z_i : X))\pi_j^\sharp(2, (z_j : X)) \Leftrightarrow (x_i : X)\sigma(x_j : X))$$

where, for each $i \in \{1, \ldots, k\}$, $zs(\beta_i) = \{z_i : X\}$.

Let

$$\begin{aligned}\nu \;=\; & W(\mathcal{NS}(\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Spec,id_{n+1})})) + W(\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Impl,id_{n+1})}) + \\ & \max\{L_?(\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Impl,id_{n+1})}), L_\sqcap(\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Impl,id_{n+1})})\}\end{aligned}$$

Then:

(I) if, for each equivalence relation σ *on* Γ *which satisfies* γ, $\mathit{Good}(E)$ *holds for all nodes* E *which are reachable from* E_σ *(by zero or more transitions), then*

case $M = \mathrm{Tr}$: *Spec* $\sqsubseteq^M_{[X \mapsto \kappa'],\eta'}$ *Impl for all nonzero cardinals* κ' *and all* η' *which satisfy* γ;

cases $M = \mathrm{Fa}$ **and** $M = \mathrm{FD}$: *Spec* $\sqsubseteq^M_{[X \mapsto \kappa'],\eta'}$ *Impl for all cardinals* $\kappa' \geq 2$ *and all* η' *which satisfy* γ;

(II) if, for some equivalence relation σ *on* Γ, $\mathit{Good}(E)$ *fails for some node* E *which is reachable from* E_σ *(by zero or more transitions), then Spec* $\not\sqsubseteq^M_{[X \mapsto \kappa'],\eta'}$ *Impl for all nonzero cardinals* $\kappa' \geq \nu$ *and all* η' *such that*

$$\forall i, j \in \{1, \ldots, k\}. (\eta'[\![x_i]\!] = \eta'[\![x_j]\!]) \Leftrightarrow ((x_i : X)\sigma(x_j : X))$$

(III) *if* $M = \mathrm{Tr}$, *if* κ *is a nonzero cardinal such that* $\kappa \geq \nu$, *and if* *Spec* $\sqsubseteq^M_{[X\mapsto\kappa],\eta}$ *Impl for all* η *which satisfy* γ, *then* *Spec* $\sqsubseteq^M_{[X\mapsto\kappa'],\eta'}$ *Impl for all nonzero cardinals* κ' *and all* η' *which satisfy* γ*;*

(IV) *if* $M = \mathrm{Tr}$, *if* κ *is a nonzero cardinal such that* $\kappa \geq \nu$, *and if* *Spec* $\not\sqsubseteq^M_{[X\mapsto\kappa],\eta}$ *Impl for some* η, *then* *Spec* $\not\sqsubseteq^M_{[X\mapsto\kappa'],\eta'}$ *Impl for all nonzero cardinals* $\kappa' \geq \nu$ *and all* η' *such that*

$$\forall i,j \in \{1,\ldots,k\}.(\eta'[\![x_i]\!] = \eta'[\![x_j]\!]) \Leftrightarrow (\eta[\![x_i]\!] = \eta[\![x_j]\!])$$

(V) *if* $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$, *if* κ *is a cardinal such that* $\kappa \geq \max\{\nu, 2\}$, *and if* *Spec* $\sqsubseteq^M_{[X\mapsto\kappa],\eta}$ *Impl for all* η *which satisfy* γ, *then* *Spec* $\sqsubseteq^M_{[X\mapsto\kappa'],\eta'}$ *Impl for all cardinals* $\kappa' \geq 2$ *and all* η' *which satisfy* γ*;*

(VI) *if* $M = \mathrm{Fa}$ *or* $M = \mathrm{FD}$, *if* κ *is a cardinal such that* $\kappa \geq \max\{\nu, 2\}$, *and if* *Spec* $\not\sqsubseteq^M_{[X\mapsto\kappa],\eta}$ *Impl for some* η, *then* *Spec* $\not\sqsubseteq^M_{[X\mapsto\kappa'],\eta'}$ *Impl for all nonzero cardinals* $\kappa' \geq \nu$ *and all* η' *such that*

$$\forall i,j \in \{1,\ldots,k\}.(\eta'[\![x_i]\!] = \eta'[\![x_j]\!]) \Leftrightarrow (\eta[\![x_i]\!] = \eta[\![x_j]\!])$$

(If $X \in \mathit{FinTypeVars}$, *which may be required when* $M = \mathrm{FD}$ *(see under 'Restrictions with FD' in Section 2.5.2), then it is implicitly assumed that* κ *and* κ' *are natural numbers (see Section 2.4).)*

Proof. (I)–(II) follow by Lemmas 5.4.3–5.4.4, Proposition 5.4.2, Theorem 3.4.1 and Theorem 3.7.4.

(III)–(VI) follow by (I)–(II). ■

The following are some of the features of the statement of Theorem 5.4.7 which are present because of practical applications (for further remarks, see Section 5.6):

- both (I)–(II) (which refer to the quotient exploration graph) and (III)–(VI) (which do not) are stated;
- lower bounds, rather than fixed values, are used for κ;
- the lower bounds for κ are the same in (III) and (IV), and in (V) and (VI).

By making Lemmas 5.4.3–5.4.6 more precise, the expression for ν in Theorem 5.4.7 can be replaced by a more precise one, most notably to avoid counting the constants from Γ in both of the first two summands. The present expression was used for simplicity.

5.4.4 Some examples

Example 5.4.1 Recall the processes *BImpl* from Example 2.1.3 and *BUFF* from Example 2.1.4. (See also Examples 2.1.5 and 5.3.1.)

We can apply Theorem 5.4.7 with $\Gamma = \{\}$, $T = \mathit{in}.X + \mathit{mid}.X + \mathit{out}.X$, $\mathit{Spec} = \mathit{BUFF}$, $\mathit{Impl} = \mathit{BImpl}$, any M, and $\gamma = \mathit{True}$. We have that

$$W(\mathcal{NS}(\mathcal{S}_{\{\}\vdash \mathit{BUFF}})) = 2 \qquad W(\mathcal{S}_{\{\}\vdash \mathit{BImpl}}) = 2$$
$$L_?(\mathcal{S}_{\{\}\vdash \mathit{BImpl}}) = 1 \qquad L_\sqcap(\mathcal{S}_{\{\}\vdash \mathit{BImpl}}) = 0$$

so that, in particular, (III) and (V) give us that if $BUFF \sqsubseteq^M_{[X \mapsto 5]} BImpl$ (which is the case, as we have observed in Example 2.1.4), then

case $M = \mathrm{Tr}$**:** $BUFF \sqsubseteq^M_{[X \mapsto \kappa']} BImpl$ for all nonzero cardinals κ';

cases $M = \mathrm{Fa}$ **and** $M = \mathrm{FD}$**:** $BUFF \sqsubseteq^M_{[X \mapsto \kappa']} BImpl$ for all cardinals $\kappa' \geq 2$.

We can also apply Theorem 5.4.7 with $\Gamma = \{\}$, $T = in.X + mid.X + out.X$, $Spec = BImpl$, $Impl = BUFF$, any M, and $\gamma = True$. We have that

$$\begin{array}{ll} W(\mathcal{NS}(\mathcal{S}_{\{\}\vdash BImpl})) = 2 & W(\mathcal{S}_{\{\}\vdash BUFF}) = 2 \\ L_?(\mathcal{S}_{\{\}\vdash BUFF}) = 1 & L_\sqcap(\mathcal{S}_{\{\}\vdash BUFF}) = 0 \end{array}$$

so that, in particular,

case $M = \mathrm{Tr}$**:** (III) gives us that if $BImpl \sqsubseteq^M_{[X \mapsto 5]} BUFF$ (which is the case, as we have observed in Example 2.1.4), then $BImpl \sqsubseteq^M_{[X \mapsto \kappa']} BUFF$ for all nonzero cardinals κ';

cases $M = \mathrm{Fa}$ **and** $M = \mathrm{FD}$**:** (VI) gives us that if $BImpl \not\sqsubseteq^M_{[X \mapsto 5]} BUFF$ (which is the case, as we have observed in Example 2.1.4), then $BImpl \not\sqsubseteq^M_{[X \mapsto \kappa']} BUFF$ for all cardinals $\kappa' \geq 5$.

■

Example 5.4.2 Recall the processes *Spec* and *Impl* from Example 5.0.2. (See also Example 5.3.2.)

We can apply Theorem 5.4.7 with any M and $\gamma = True$. We have that

$$\begin{array}{ll} W(\mathcal{NS}(\mathcal{S}_{\{\}\vdash Spec})) = K & W(\mathcal{S}_{\{\}\vdash Impl}) = K \\ L_?(\mathcal{S}_{\{\}\vdash Impl}) = 1 & L_\sqcap(\mathcal{S}_{\{\}\vdash Impl}) = 0 \end{array}$$

so that, in particular, (IV) and (VI) give us that if $Spec \not\sqsubseteq^M_{[X \mapsto (2K+1)]} Impl$ (which is the case, as we have observed in Example 5.0.2), then $Spec \not\sqsubseteq^M_{[X \mapsto \kappa']} Impl$ for all cardinals $\kappa' \geq (2K+1)$. ■

Example 5.4.3 Recall the processes *Spec* and *Impl* from Example 5.0.3, where we have observed that *Spec* and *Impl* are X-DI and that $Spec \sqsubseteq^M_\delta Impl$ holds if and only if $|\delta[\![X]\!]| < K_1 + K_2 + K_3$.

Now

$$\begin{array}{ll} W(\mathcal{NS}(\mathcal{S}_{\{\}\vdash Spec})) = K_1 & W(\mathcal{S}_{\{\}\vdash Impl}) = K_2 \\ L_?(\mathcal{S}_{\{\}\vdash Impl}) = K_3 & L_\sqcap(\mathcal{S}_{\{\}\vdash Impl}) = 0 \end{array}$$

and so Example 5.0.3 provides a three-dimensional infinite collection of examples for which the expression for ν in Theorem 5.4.7 yields the minimal possible value.

(Example 5.0.3 was constructed by A.W. Roscoe for this purpose.) ■

Example 5.4.4 Consider the processes

$$\begin{array}{l} Spec = (c\$x : X \longrightarrow STOP) \underset{\{c.*_X\}}{\|} (c\$y : X \longrightarrow STOP) \\ Impl = STOP \end{array}$$

which are X-DI and such that, if $M = \mathrm{Fa}$ or $M = \mathrm{FD}$, $\mathit{Spec} \sqsubseteq_{\delta}^{M} \mathit{Impl}$ holds if and only if $|\delta[\![X]\!]| \geq 2$.

Since

$$\begin{array}{ll} W(\mathcal{NS}(\mathcal{S}_{\{\}\vdash \mathit{Spec}})) = 0 & W(\mathcal{S}_{\{\}\vdash \mathit{Impl}}) = 0 \\ L_{?}(\mathcal{S}_{\{\}\vdash \mathit{Impl}}) = 0 & L_{\sqcap}(\mathcal{S}_{\{\}\vdash \mathit{Impl}}) = 0 \end{array}$$

it follows that, in Theorem 5.4.7, the lower bound of 2 for κ' in (I), for κ' in (V), and for κ in (VI), cannot be omitted. ■

5.5 Further developments

This section is devoted to mentioning a few further developments which either could not be presented in detail in this thesis because of space limitations, or which are by other authors.

5.5.1 Another theorem with sets of dependent sizes

We have obtained a theorem which has stronger assumptions than Theorem 5.4.7, but which provides threshold collections with potentially smaller sets.

The theorem applies to *Spec* and *Impl* which are data independent in X with no constants and which have the same alphabet, and to the finite-traces model of CSP. It requires that *Spec* and *Impl* satisfy $(\mathbf{NoEqT}_X)$ and $(\mathbf{RecCho}_X)$ and a condition on their ways specifiers, and it provides singleton threshold collections with sets whose sizes are bounded below by an expression which depends on *Spec*. Since *Spec* and *Impl* are required to satisfy $(\mathbf{NoEqT}_X)$, the theorem can be thought of as being closer to Theorems 5.3.5 and 5.3.6 than to Theorem 5.4.7.

We conjecture that it is straightforward to extend this theorem to data independent processes with constants, and that it can be extended to the stable-failures and failures/divergences models.

5.5.2 Threshold collections for determinism

The present chapter has been devoted to threshold collections for refinement (see Sections 2.1.3 and 2.6). Since it is the notion of correctness in CSP, refinement is of central importance in practice.

Another practically important property is *determinism* [Ros98a, Sections 3.3 and 9.1], where a process $\Gamma \vdash P$ is deterministic with respect to δ and η if and only if

(i) $\mathit{divergences}(\Gamma \vdash P)_{\delta,\eta} = \{\}$ and

(ii) $t\hat{\ }\langle a\rangle \in \mathit{traces}(\Gamma \vdash P)_{\delta,\eta} \;\Rightarrow\; (t, \{a\}) \notin \mathit{failures}(\Gamma \vdash P)_{\delta,\eta}$.

Most notably perhaps, determinism has important applications in the area of security as non-interference (see e.g. [Ros98a, Section 12.4]). Being of such practical relevance, the property can be checked on the model checker FDR [Ros94a, Ros98a, FS97].

We have obtained three theorems which provide threshold collections for determinism. They apply to a process *Impl* which is data independent with respect to X with no constants and which cannot perform any nondeterministic selections of values of type X (in the sense of 'An assumption about implementation processes' at the beginning of the present chapter). The first theorem requires that *Impl* can contain values of type X only in inputs, and it provides a singleton threshold collection with a set of size 1. The second one requires that *Impl* satisfies

($\mathbf{NoEqT}_X$), and it provides singleton threshold collections with sets whose sizes are bounded below by an expression which depends on *Impl*, although it evaluates to 2 in most practical examples. The third one does not have any additional assumptions, and it provides singleton threshold collections with sets whose sizes are bounded below by an expression similar to the expression for ν in Theorem 5.4.7. For CSP$_M$ versions of the latter two theorems, see under 'Determinism checks' in Section 15.2.2 of [Ros98a]. We conjecture that it is straightforward to extend the theorems to data independent processes with constants.

5.5.3 Arrays and many-valued predicates

We have also obtained a theorem which provides threshold collections for determinism and allows: *arrays* whose indices come from a type variable $X \in \mathit{NEFTypeVars}$ and whose values come from a distinct type variable $Y \in \mathit{NEFTypeVars}$, *many-valued predicates* on X, and constants of type Y. An array whose indices come from X and whose values come from Y is here a member of a special type $\mathit{Array}_{X,Y}$, and there are special constructs for creating arrays, reading from arrays, and updating arrays. A many-valued predicate on X is here a free term variable of type $X \rightarrow U$, where U is a nonempty finite type with no free type variables, or more precisely U^{nef}, U^{fin} (see under 'Conditions on types' in Section 2.4) and $\mathit{Free}(U) = \{\}$.

The theorem applies to a process *Impl* whose free type variables are at most X and Y, which can use arrays of type $\mathit{Array}_{X,Y}$, whose free term variables are at most a many-valued predicate of a type $X \rightarrow U$ and a constant of type Y, and which cannot perform any nondeterministic selections of values of type X or of type Y (in the sense of 'An assumption about implementation processes' at the beginning of the present chapter). It requires that *Impl* satisfies ($\mathbf{NoEqT}_Y$) and that it satisfies a few technical conditions which are satisfied by most processes in practice, and it provides singleton threshold collections in which X is interpreted by sets whose sizes are bounded below by an expression depending on *Impl* and on the size of U, and Y is interpreted by a set of size 2. The theorem also provides a possibility of reducing the number of states of the threshold instance of *Impl* by composing it in parallel with a special process *OneOne* (as in Theorem 5.3.5) with respect to the type variable Y.

The lower bound for the sizes of sets which interpret X can be finite even when *Impl* can during its execution access an unbounded number of different array components. This is because, when determining the supremum of the number of elements of X that *Impl* ever has to store for future use, elements which are stored only because they are indices of array components that have previously been read or updated are not counted. This is in turn made possible by using the special type $\mathit{Array}_{X,Y}$ and the special constructs, rather than simply regarding the arrays as members of the function type $X \rightarrow Y$.

Since it allows arrays of type $\mathit{Array}_{X,Y}$ and many-valued predicates of types $X \rightarrow U$, the theorem can be seen as applying to processes which possess a substantially weaker property than data independence in X and Y. In Section 7.2.1, refinements between processes which possess the same property will be mentioned as an item for future work.

A CSP$_M$ version of the theorem can be found in [LR98]. The same paper also presents an application of the theorem in verifying that a database system which allows users to lock, read and write records at multiple security levels is secure.

5.5.4 Proving security protocols with model checkers

As we have mentioned in Section 2.7.1, Roscoe and Broadfoot have recently been showing, by using techniques presented in this thesis, how security protocols can be proved with model checkers [Ros98b, Bro98, RB98]. For two directions for related future work, see Section 7.2.1.

5.5.5 Parameterisation by network size/structure

As we have also mentioned in Section 2.7.1, Lowe, Roscoe and Creese have been showing, by building on the work presented in this thesis, how data independence enables systems which contain arbitrarily many nodes to be proved with model checkers by induction [Low96, Cre98, CR99b]. Some remarks about related future work can be found in Section 7.2.2.

5.6 Tools

In this section, we remark about possibilities for automation which arise from the results presented in this chapter. It is a continuation of Section 3.8, where we discussed automation based on the research presented in Chapter 3.

As we observed in Section 3.8.1, the first issue when it comes to automation is the fact that the language in this thesis is a smaller and more theoretically presented version of CSP$_M$ [Sca92, Sca98, Ros98a], the machine-readable version of CSP which is used in tools. We then pointed at the two basic approaches for dealing with this difference: isolating a sublanguage of CSP$_M$ that corresponds to the language in this thesis, and adapting the research in the thesis to CSP$_M$. Since we are leaving this issue for future research (see Section 7.2.4), we shall for the remainder of the present section make a hypothetical assumption that the language in this thesis can be used in tools directly.[3]

The theorems we presented in Sections 5.1–5.4 and the further developments we mentioned in Section 5.5 were all obtained with the aim of being immediately applicable to practical model checking. Indeed, a number of nontrivial case studies have already been performed using them: see e.g. Section 15.2.1 of [Ros98a] (a distributed database), Section 15.2.3 of [Ros98a] and Chapter 6 of this thesis (a cache), [LR98] (a database system with multiple security levels), [RB98] (a number of security protocols), [Cre98] (the Time-Triggered Group Membership Protocol).

However, all applications so far have been characterised by manual checking of assumptions, manual deciding of which theorem to apply, manual calculating of lower bounds for sizes of sets in threshold collections, and manual interpreting of conclusions. In other words, the results have been used in manual reasoning that surrounds applications of tools, which have remained as before.

In the following sections, we shall remark about possibilities for making applications of the theorems in Sections 5.1–5.4 and of the further developments in Section 5.5 more automatic and more efficient.

5.6.1 Checking of assumptions

As we remarked in Section 3.8.3, each of (**NoEqT**$_X$), (**PosConjEqT**$_X$), (**RecCho**$_X$) and (**Norm**) can be checked completely automatically, with guaranteed termination.

Consequently, the assumptions of each of the theorems in Sections 5.1–5.4 and of each of the theorems mentioned in Sections 5.5.1 and 5.5.2 can also be checked completely automatically, with guaranteed termination. We conjecture that the same is true of the theorem mentioned in Section 5.5.3.

[3]As we remarked in Section 3.8.1, a substantial research project whose aim is automation based on the work presented in this thesis will begin in 1999.

(For CSP$_M$, a type checker would enable conditions of data independence and weak data independence to be checked automatically, as we remarked in Section 3.8.1. Type checkers could also have capabilities which facilitate automatic checking of conditions involving ways specifiers.)

5.6.2 Calculating of lower bounds

There are two basic approaches for automatically calculating the lower bounds for sizes of sets in threshold collections in Theorem 5.4.7, in the theorem mentioned in Section 5.5.1, in the latter two theorems mentioned in Section 5.5.2, and in the theorem mentioned in Section 5.5.3.

The first is to automatically construct the relevant SLTSs, ASLTSs and NSLTSs (see Sections 3.8.2 and 3.8.4) and to calculate the lower bounds from those. Thus, at least in the simplest version of this approach, its termination depends on termination of the construction of the relevant SLTSs, ASLTSs and NSLTSs.

The second is to automatically estimate the lower bounds from the syntax of the relevant processes, which can be performed with different degrees of sophistication. The estimates would in general be greater than or equal to the exact lower bounds, and that is why the theorems we have just mentioned were stated in terms of lower bounds, rather than in terms of threshold collections with exact sizes of sets. An advantage of this approach is that it may yield finite estimates in cases where the first approach does not terminate. (In connection with the approach, see Proposition 3.7.5. We have worked on the approach in much more detail, but we do not have space to present any more of the results in this thesis.)

5.6.3 Greater efficiency

Applying a theorem involves checking its assumptions, possibly calculating a lower bound (or an estimate for it) for sizes of sets in threshold collections, and checking refinement (or determinism) for interpretations from a threshold collection. In the previous two sections, we remarked about possibilities for automating the first two stages.

The third stage can be performed automatically using existing model checkers (see Section 2.1.4, and see the remarks about CSP$_M$ at the beginning of Section 5.6). In this section, we briefly discuss possibilities for making this more efficient.

In Theorem 5.3.5, the parallel compositions with the special processes *OneOne* and *Zeros* can be seen as providing greater efficiency for checking refinements (see the remarks near the beginning of Section 5.3). We conjecture that efficiency of checking determinism in the second of the three theorems which were mentioned in Section 5.5.2 can be improved in a very similar way.

In Theorem 5.4.7, parts (I)–(II) offer a possibility of greater efficiency compared with parts (III)–(VI), because constructing and exploring the quotient exploration graph

$$\mathcal{H}_{\mathcal{NS}(\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Spec,id_{n+1})}),\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Impl,id_{n+1})},M}$$

is in general expected to be considerably more efficient than checking the refinements in parts (III)–(VI).

The constructing and exploring of the quotient exploration graph can be completely automated. As we remarked in Sections 3.8.2 and 3.8.4, the SLTS

$$\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Impl,id_{n+1})}$$

and the NSLTS

$$\mathcal{NS}(\mathcal{S}_{\{\}\vdash NoC(X,\Gamma,Spec,id_{n+1})})$$

can be constructed automatically. Provided those constructions terminate (see Sections 3.8.2 and 3.8.4), the quotient exploration graph can be constructed automatically with guaranteed termination. It is then possible to automatically check whether, for each equivalence relation σ on Γ which satisfies γ, $Good(E)$ holds for all nodes E which are reachable from E_σ. Alternatively, the constructing and the exploring can be performed lazily together, which avoids the dependence on termination of the constructions of the SLTS and the NSLTS.

It is also possible to prove a version of part (II) which does not refer to ν, so that calculating ν becomes unnecessary for parts (I)–(II).

Greater efficiency can be achieved very similarly in the third theorem mentioned in Section 5.5.2. The same approach can also be used for the theorem mentioned in Section 5.5.3, although the details are more complex.

The parallel compositions with the special processes *OneOne* and *Zeros* can be used on existing model checkers, but the constructing and the exploring of quotient exploration graphs requires the existing model checkers for CSP to be considerably extended. As an alternative to quotient exploration graphs, symmetry elimination [Sta91, CE+96, ES96, ES95, ID96a, Jen96] can be employed in checks of refinement (or determinism). This seems to have an advantage of requiring less substantial extensions of the existing model checkers for CSP while resulting in the same degree of efficiency (or being better by a constant factor), but seems to have a disadvantage of not being as open to more general classes of processes than data independent ones.

We conjecture that greater efficiency can also be achieved in the theorem mentioned in Section 5.5.1, in a way which is related to those we have been discussing above.

5.6.4 Proof environments

Taking a wider view, automation based on the research presented in this thesis could result in a proof environment which would enable assertions made up at least from refinements and universal quantification of data independent type variables to be either proved or refuted.

Deciding of which theorems to apply and of which approaches for greater efficiency to employ could either be left to the user, or performed by the proof environment, where the latter could be done with different degrees of sophistication.

The proper place of such a development is as a part of a much more comprehensive proof environment which, as suggested in [Ros94a], would in particular enable deduction of refinements by transitivity and monotonicity.

5.7 Notes

In the first four sections of this chapter, which is the main chapter of this thesis, we presented theorems which provide threshold collections for the problem of, given two weakly data independent processes *Spec* and *Impl* with the same alphabet, how does whether *Spec* is refined by *Impl* change with varying the interpretation of the weakly data independent type and the associated constants and operations. More precisely, weak data independence was allowed only in the first section, whereas the remaining three sections required data independence.

In the fifth section, we mentioned a few further developments which we could not present in detail in this thesis. In the sixth section, we remarked about possibilities for automation.

The starting point for the research presented in this chapter (and in the whole of this thesis) was [RM94], where a few theorems providing threshold collections were conjectured, focussing on the finite-traces model of CSP and without considering equality tests between values from weakly data independent or data independent types. In particular, all the theorems in Sections 5.1, 5.3 and 5.5.1 of this thesis developed from conjectures in [RM94].

Our previous writings on the topic of this thesis are [LR96b] and [Laz97], which also are in terms of languages that are smaller and more theoretically presented versions of CSP$_M$ [Sca92, Sca98, Ros98a]. Another previous presentation of our results is [Ros98a, Section 15.2.2], which is in terms of CSP$_M$, and which probably is the most accessible summary of this work. The basis for [Ros98a, Section 15.2.2] was [Laz99], which provides much more detail concerning CSP$_M$. The further development discussed in Section 5.5.3 of this thesis has been presented in [LR98]. Some references for two further developments which are by other authors were given in Sections 5.5.4 and 5.5.5.

We have certainly not been the first to prove theorems which facilitate model checking of data independent systems. Some references for others' research on this topic are [Wol86, JP93, ID93a, ID93b, ID96a, Ip96, HB95, HM+95, HI+96, Hoj96, HDB97]. See also the references on symbolic execution given in Section 3.9, on abstraction given in Section 4.3, and on eliminating symmetry given near the beginning of Section 5.4, which are three related topics. For some references related to the further development mentioned in Section 5.5.3, see [LR98]. Further references on data independence and on the Parameterised Verification Problem more generally can be found in [Laz98].

Firstly, a contribution of this chapter is that the language used is a combination of CSP and a typed λ-calculus and that the notion of correctness used is refinement (and determinism, in Sections 5.5.2 and 5.5.3), which are distinguished from other languages and frameworks for reasoning about concurrent systems by a number of features (see Section 2.3). In particular, the theorems in this chapter allow specifications to be processes which are expressed using a wealth of language constructs, and they allow refinements to be relative to denotational models of CSP which enable reasoning about nondeterminism and divergence. Secondly, we are not aware of results in the literature which are comparable to Theorem 5.2.2, to the second of the three theorems mentioned in Section 5.5.2, or to the theorem mentioned in Section 5.5.3 (which was based on a conjecture of A.W. Roscoe). Thirdly, a theoretically minor but practically useful contribution is that the notion of data independence used allows constants of the data independent type (see Section 2.7.1).

The following are some more precise remarks about relations between results in this chapter and results by other authors:

- The basic idea in Theorems 5.1.2 and 5.1.3 is the same as the basic idea of some results in [HGD95], and it is related to Proposition 5.1 in [Wol86].

- Theorems 5.3.5 and 5.3.6 are related to the 0-1 Theorem for Data-insensitive Controllers in [HB95] (which is Theorem 8.1 in [Hoj96]), and to techniques used in a number of case studies, e.g. [SWL85, Sab88, Kai96].

  In addition to the first and the third general contribution mentioned above, Theorems 5.3.5 and 5.3.6 are distinguished by allowing restricted equality tests between values of the data independent type, and Theorem 5.3.5 is distinguished by the parallel compositions with the special processes *OneOne* and *Zeros*.

- Theorem 5.4.7 is related to Theorems 5 and 6 in [ID96a] (which are Theorems 4.1 and 4.2 in [Ip96]), and to Lemma 4.1 in [HB95] (which is Lemma 8.4.1 in [Hoj96]).

In Theorem 5.4.7, the only assumptions on specifications are data independence with respect to X and Γ, and the condition ($\mathbf{RecCho}_X$). Consequently, in contrast to the three related results, Theorem 5.4.7 involves symbolic normalisation of specifications. Another distinguishing characteristic of Theorem 5.4.7 is the complex definition of ν, which is due to the expressiveness of the language and of the notion of correctness.

- The theorem mentioned in Section 5.5.1 is related to the results in [JP93], and it has some similarities with Theorem 5.4 in [Wol86].

Chapter 6

A case study: verifying a cache

This chapter is devoted to a more substantial example than those we have seen so far. Namely, we shall see how the theorems presented in the previous chapter can be applied in verifying a cache.

6.1 Defining the cache

Suppose we have a memory which is relatively slow to access compared to the clock-speed of a processor. The standard solution to this is to use a *cache*: a relatively small piece of fast memory through which the processor makes all accesses and which always keeps part of the main memory in it, in the hope that the addresses which are required will be already there. Modern computers may have several levels of cacheing, for example: virtual memory versus RAM, DRAM versus SRAM, and off-processor memory versus on-processor memory.

At the level of abstraction that we shall be working with, the cache will communicate with the processor along a channel ra of type A (for read requests), a channel rv of type V (for read values), and a channel w of type $A \times V$ (for writes). Similarly, the cache will communicate with the main memory along channels m_ra, m_rv and m_w of the same types. The types A and V of addresses and values will be type variables which can be assigned only finite nonempty sets with flat orderings (see under 'Type variables' in Section 2.4, under 'T^{alph}' in 'Conditions on types' in Section 2.4, and under 'Restrictions with FD' in Section 2.5.2). Therefore, we shall be using the following type T as the alphabet of all processes in this chapter.

$A \in \mathit{NEFTypeVars} \cap \mathit{FinTypeVars}$
$V \in \mathit{NEFTypeVars} \cap \mathit{FinTypeVars}$
$T = ra.A + m\_ra.A + rv.V + m\_rv.V + w.(A \times V) + m\_w.(A \times V)$

At any time, the cache keeps a list of elements of the type $(A \times (V \times \mathit{Bool}))$. The first component of such a triple is an address presently resident in the cache, the second component is the value stored in that address, and the third component is a boolean which records whether or not the address has been written to since it was brought into the cache. The boolean is necessary because it lets us tell, when the address is eventually removed from the cache, whether or not the stored value has to be written-back to the main memory.

The capacity of the cache, i.e. the maximum length of the list that it keeps, will be a free variable N of the type Num of natural numbers. We shall assume that N cannot be assigned the value 0.

The cache starts off empty and fills itself up as the processor asks to read and write to addresses. Such reads evidently require a fetch from the main memory. Any read or write to an address already in the cache is done without any interaction with the memory. When the cache is full and a miss occurs (i.e., an attempt to access an address not presently held), it is necessary to flush one triple out of the cache in order to make room for the new one. There are a number of *cache replacement policies* which can be used to do this, such as 'first in, first out', random, and 'least recently used'.

The following definition captures all the behaviour of such a cache except what happens when a flush is required. Note how the booleans that control write-backs are set to *true* whenever a write to the given address occurs.

```
$Cache(ps) =$
$ra?a : A \longrightarrow$
  $if\ \ elem(a, [a' \mid (a', x) \leftarrow ps])$
      $then\ rv!val(a, ps) \longrightarrow Cache(ps)$
      $else$
        $if\ \ len(ps) < N$
            $then\ \ m\_ra!a \longrightarrow m\_rv?v : V \longrightarrow$
                    $rv!v \longrightarrow Cache([(a, (v, false))] +\!\!+ ps)$
            $else\ \ FlushAndRead(ps, a)$
$\Box\ w?a : A?v : V \longrightarrow$
  $if\ \ elem(a, [a' \mid (a', x) \leftarrow ps])$
      $then\ Cache(update(a, v, ps))$
      $else$
        $if\ \ len(ps) < N$
            $then\ Cache([(a, (v, true))] +\!\!+ ps)$
            $else\ FlushAndWrite(ps, a, v)$
```

The first of the following two functions was used above to read from an address which is in the cache, and the second to write to an address which is in the cache. The other functions and functional programming constructs used above are standard (see under 'More complex functions and syntactic sugar' in Section 2.5.1).

$$val(a, ps) = head([v \mid (a', (v, b)) \leftarrow ps, a' = a])$$
$$update(a, v, ps) = [(a, (v, true))] +\!\!+ [(a', x) \mid (a', x) \leftarrow ps, a' \neq a]$$

The processes $FlushAndRead(ps, a)$ and $FlushAndWrite(ps, a, v)$ flush a nondeterministically chosen member of the list ps out, performing a write-back to the memory if necessary. They then fill up the vacant space with a new triple, whose address has just been either read-from or written-to by the processor.

The cache defined in this way is refined (see Section 2.6) by caches with any particular cache replacement policy, such as the three policies mentioned above. Consequently, correctness of this cache (see the next section) will imply correctness for any policy.

```
$FlushAndRead(ps, a) =$
$if\ \ len(ps) = 0$
    $then\ err \longrightarrow STOP$
    $else\ foldr1(\sqcap, [FlushAndRead'(ps, a, i) \mid i \leftarrow [1..len(ps)]])$
```

```
$FlushAndRead'(ps, a, i) =$
$if\ \ wr\_bk$
    $then\ \ m\_w!a'!v' \longrightarrow m\_ra!a \longrightarrow m\_rv?v : V \longrightarrow rv!v \longrightarrow$
          $Cache([(a, (v, false))] +\!\!+ take(i-1, ps) +\!\!+ drop(i, ps))$
    $else\ \ m\_ra!a \longrightarrow m\_rv?v : V \longrightarrow rv!v \longrightarrow$
          $Cache([(a, (v, false))] +\!\!+ take(i-1, ps) +\!\!+ drop(i, ps))$
$where\ (a', (v', wr\_bk)) = ith(i, ps)$
```

```
$FlushAndWrite(ps, a, v) =$
$if\ \ len(ps) = 0$
    $then\ err \longrightarrow STOP$
    $else\ foldr1(\sqcap, [FlushAndWrite'(ps, a, v, i) \mid i \leftarrow [1..len(ps)]])$
```

```
$FlushAndWrite'(ps, a, v, i) =$
$if\ \ wr\_bk$
    $then\ \ m\_w!a'!v' \longrightarrow$
          $Cache([(a, (v, true))] +\!\!+ take(i-1, ps) +\!\!+ drop(i, ps))$
    $else\ \ Cache([(a, (v, true))] +\!\!+ take(i-1, ps) +\!\!+ drop(i, ps))$
$where\ (a', (v', wr\_bk)) = ith(i, ps)$
```

6.2 Verifying the cache

There are various ways in which one can provide a model of the memory that the cache is intended to interact with. Perhaps the obvious one is to model it as a simple shell around a function from the type A of addresses to the type V of values.

```
$Memory(f) =$
$m\_ra?a : A \longrightarrow m\_rv!f(a) \longrightarrow Memory(f)$
$\Box\ m\_w?a : A?v : V \longrightarrow Memory(\lambda\, a' : A\,.\ if\ a = a'\ then\ v\ else\ f(a'))$
```

Suppose K is a nonzero natural number. The correctness condition for the cache of capacity K is that the parallel composition of it and the memory with the internal channels hidden should refine the memory itself with appropriately renamed channels. More precisely, let

$$S = \{m\_ra.*_A, m\_rv.*_V, m\_w.*_{A \times V}\}$$

and let $MemoryRen(f)$ have the same definition as $Memory(f)$ except that m_ra, m_rv and m_w are replaced by ra, rv and w. We then consider that verifying the cache of capacity K amounts to verifying the refinement

$$MemoryRen(\lambda\, a : A.v)\ \sqsubseteq^{\mathrm{FD}}_{\delta,\eta}\ (Cache([])[K/N] \underset{S}{\parallel} Memory(\lambda\, a : A.v)) \setminus S \tag{6.1}$$

for all δ (which assign finite nonempty sets with flat orderings to A and V) and all η (which are assignments to $\{v : V\}$ with respect to δ).

By its syntactic definition, the process $MemoryRen(\lambda\, a : A.v)$ is *deterministic* [Ros98a, Sections 3.3 and 9.1] for all δ and all η. Since any deterministic element of the failures/divergences model is maximal with respect to refinement [Ros98a, Lemma 9.1.1], we have that the refinement above in fact implies equality.

Suppose U is a type such that U^{nef}, U^{fin} (see under 'Conditions on types' in Section 2.4) and $\mathit{Free}(U) = \{\}$. It is then straightforward to check that Theorem 5.3.5 applies to the refinement (6.1) in which U is substituted for A. It gives us that if

$$\begin{array}{c} \mathit{MemoryRen}(\lambda\, a : A.v)[U/A, \mathit{Two}/V] \underset{\{*_{T[U/A,\mathit{Two}/V]}\}}{\parallel} \mathit{OneOne}_{V,T[U/A],\psi} \\ \sqsubseteq^{\mathrm{FD}}_{\delta',\{v\mapsto(zz,())\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\parallel} \mathit{Memory}(\lambda\, a : A.v)) \setminus S)[U/A, \mathit{Two}/V] \\ \underset{\{*_{T[U/A,\mathit{Two}/V]}\}}{\parallel} \mathit{OneOne}_{V,T[U/A],\psi} \end{array} \tag{6.2}$$

and

$$\begin{array}{c} \mathit{MemoryRen}(\lambda\, a : A.v)[U/A, \mathit{Two}/V] \underset{\{*_{T[U/A,\mathit{Two}/V]}\}}{\parallel} \mathit{Zeros}_{V,T[U/A],\psi} \\ \sqsubseteq^{\mathrm{FD}}_{\delta',\{v\mapsto(oo,())\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\parallel} \mathit{Memory}(\lambda\, a : A.v)) \setminus S)[U/A, \mathit{Two}/V] \\ \underset{\{*_{T[U/A,\mathit{Two}/V]}\}}{\parallel} \mathit{Zeros}_{V,T[U/A],\psi} \end{array} \tag{6.3}$$

where

$$\begin{array}{ll} \psi(ra, 1) = \{?\} & \psi(m\_ra, 1) = \{\} \\ \psi(rv, 1) = \{!\} & \psi(m\_rv, 1) = \{\} \\ \psi(w, 1) = \{?\} & \psi(w, 2) = \{?\} \\ \psi(m\_w, 1) = \{\} & \psi(m\_w, 2) = \{\} \end{array}$$

and where δ' is arbitrary, then

$$\begin{array}{c} \mathit{MemoryRen}(\lambda\, a : A.v)[U/A] \sqsubseteq^{\mathrm{FD}}_{[V\mapsto\kappa],\eta'} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\parallel} \mathit{Memory}(\lambda\, a : A.v)) \setminus S)[U/A] \end{array} \tag{6.4}$$

for all nonzero natural numbers κ and all η' (which are assignments to $\{v : V\}$ with respect to $[V \mapsto \kappa]$).

Therefore, in order to show (6.1) for all δ and all η, it suffices by Corollary 4.2.3 to show (6.2) and (6.3) for all types U such that U^{nef}, U^{fin} and $\mathit{Free}(U) = \{\}$. By Corollary 4.2.3 again, this follows if we show

$$\begin{array}{c} \mathit{MemoryRen}(\lambda\, a : A.v)[\mathit{Two}/V, (zz.())/v] \underset{\{*_{T[\mathit{Two}/V]}\}}{\parallel} \mathit{OneOneExt}_{V,T,\psi} \\ \sqsubseteq^{\mathrm{FD}}_{[A\mapsto\lambda],\{\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\parallel} \mathit{Memory}(\lambda\, a : A.v)) \setminus S)[\mathit{Two}/V, (zz.())/v] \\ \underset{\{*_{T[\mathit{Two}/V]}\}}{\parallel} \mathit{OneOneExt}_{V,T,\psi} \end{array} \tag{6.5}$$

and

$$\begin{array}{c} \mathit{MemoryRen}(\lambda\, a : A.v)[\mathit{Two}/V, (oo.())/v] \underset{\{*_{T[\mathit{Two}/V]}\}}{\parallel} \mathit{ZerosExt}_{V,T,\psi} \\ \sqsubseteq^{\mathrm{FD}}_{[A\mapsto\lambda],\{\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\parallel} \mathit{Memory}(\lambda\, a : A.v)) \setminus S)[\mathit{Two}/V, (oo.())/v] \\ \underset{\{*_{T[\mathit{Two}/V]}\}}{\parallel} \mathit{ZerosExt}_{V,T,\psi} \end{array} \tag{6.6}$$

for all nonzero natural numbers λ, where $\mathit{OneOneExt}_{V,T,\psi}$ and $\mathit{ZerosExt}_{V,T,\psi}$ are defined in the same way as in Definition 5.3.1, except that inputs over the type variable A need to be used instead of replicated external choices (see under 'Replicated external choice' in Section 2.5.2).

The condition ($\mathbf{PosConjEqT}_A$) (and thus also ($\mathbf{NoEqT}_A$)) is not satisfied by any of the specifications and implementations in (6.5) and (6.6). Consequently, among the theorems in Sections 5.1, 5.3 and 5.4, we can only attempt to apply Theorem 5.4.7. It is straightforward to check that its assumptions are satisfied. Unfortunately, the expression for ν evaluates to ω (the smallest infinite cardinal), which is because the number of values of type A that ever have to be stored by $\mathit{Memory}(\lambda\, a : A.v)$ for future use is unbounded.

This can be remedied in an interesting way: we can concentrate on a single but arbitrary address. Instead of showing that all interactions with the cache are as expected for a well-behaved memory, we show that for any particular address, each read from the cache gives the right answer purely on the assumption that the reads which the cache itself makes for the chosen address are well-behaved. The following process behaves like a reliable memory for the address a, for which it has the initial value v. On other addresses, it selects the value of a read nondeterministically rather than by reference to previous writes.

$$
\begin{array}{l}
\mathit{OneLoc}(a,v) = \\
m\_w?a' : A?v' : V \longrightarrow \mathit{if}\ \ a = a' \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathit{OneLoc}(a,v') \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \mathit{OneLoc}(a,v) \\
\Box\ m\_ra?a' : A \longrightarrow \mathit{if}\ \ a = a' \\
\qquad\qquad\qquad\qquad\quad \mathit{then}\ m\_rv!v \longrightarrow \mathit{OneLoc}(a,v) \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ \bigsqcap_{v':V} m\_rv!v' \longrightarrow \mathit{OneLoc}(a,v)
\end{array}
$$

The revised correctness condition for the cache of capacity K is then that

$$\mathit{OneLocRen}(a,v)\ \sqsubseteq^{\mathrm{FD}}_{\delta,\theta}\ (\mathit{Cache}([])[K/N] \underset{S}{\parallel} \mathit{OneLoc}(a,v)) \setminus S \tag{6.7}$$

for all δ (which assign finite nonempty sets with flat orderings to A and V) and all θ (which are assignments to $\{a : A, v : V\}$ with respect to δ), where $\mathit{OneLocRen}(a,v)$ has the same definition as $\mathit{OneLoc}(a,v)$, except that the channels are renamed and an equivalent nondeterministic selection is used instead of the replicated nondeterministic choice. (The latter is to make $\mathit{OneLocRen}(a,v)$ satisfy the condition (**Norm**), which is required in Theorem 5.3.5. On the other hand, a nondeterministic selection is not used in $\mathit{OneLoc}(a,v)$ because of 'An assumption about implementation processes' at the beginning of Chapter 5.)

$$
\begin{array}{l}
\mathit{OneLocRen}(a,v) = \\
w?a' : A?v' : V \longrightarrow \mathit{if}\ \ a = a' \\
\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathit{OneLocRen}(a,v') \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ \mathit{OneLocRen}(a,v) \\
\Box\ ra?a' : A \longrightarrow \mathit{if}\ \ a = a' \\
\qquad\qquad\qquad\quad \mathit{then}\ rv!v \longrightarrow \mathit{OneLocRen}(a,v) \\
\qquad\qquad\qquad\quad \mathit{else}\ rv\$v' : V \longrightarrow \mathit{OneLocRen}(a,v)
\end{array}
$$

It can be seen without difficulty that this revised correctness condition implies the correctness condition (6.1) above. (The details of the argument are outside the scope of this thesis.)

By applying Theorem 5.3.5, Theorem 5.4.7 and Corollary 4.2.3 in the same way as above, where we now have that

$$\nu = 1 + (K+2) + \max\{1,0\} = K + 4$$

it follows that if the following four refinements hold

$$\begin{array}{c} \mathit{OneLocRen}(a,v)[\mathit{Two}/V,(zz.())/v] \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{OneOneExt}_{V,T,\psi} \\ \sqsubseteq^{\mathrm{FD}}_{[A\mapsto(K+4)],\{a\mapsto 0\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\|} \mathit{OneLoc}(a,v)) \setminus S)[\mathit{Two}/V,(zz.())/v] \\ \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{OneOneExt}_{V,T,\psi} \end{array} \tag{6.8}$$

$$\begin{array}{c} \mathit{OneLocRen}(a,v)[\mathit{Two}/V,(zz.())/v] \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{OneOneExt}_{V,T,\psi} \\ \sqsubseteq^{\mathrm{FD}}_{[A\mapsto 1],\{a\mapsto 0\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\|} \mathit{OneLoc}(a,v)) \setminus S)[\mathit{Two}/V,(zz.())/v] \\ \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{OneOneExt}_{V,T,\psi} \end{array} \tag{6.9}$$

$$\begin{array}{c} \mathit{OneLocRen}(a,v)[\mathit{Two}/V,(oo.())/v] \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{ZerosExt}_{V,T,\psi} \\ \sqsubseteq^{\mathrm{FD}}_{[A\mapsto(K+4)],\{a\mapsto 0\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\|} \mathit{OneLoc}(a,v)) \setminus S)[\mathit{Two}/V,(oo.())/v] \\ \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{ZerosExt}_{V,T,\psi} \end{array} \tag{6.10}$$

$$\begin{array}{c} \mathit{OneLocRen}(a,v)[\mathit{Two}/V,(oo.())/v] \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{ZerosExt}_{V,T,\psi} \\ \sqsubseteq^{\mathrm{FD}}_{[A\mapsto 1],\{a\mapsto 0\}} \\ ((\mathit{Cache}([])[K/N] \underset{S}{\|} \mathit{OneLoc}(a,v)) \setminus S)[\mathit{Two}/V,(oo.())/v] \\ \underset{\{*T[\mathit{Two}/V]\}}{\|} \mathit{ZerosExt}_{V,T,\psi} \end{array} \tag{6.11}$$

then (6.7) holds for all δ and all θ.

It should not come as a surprise that the obtained value of ν increases with K, since the control-flow of the cache depends crucially on whether it is full or not, and obviously it could not get full if it had more slots than there were addresses! Now, as we dispensed with the type parameters A and V, we would like to dispense with the capacity parameter N. It is not, however, one that can be dealt with via data independence. At the time of writing, we are not aware of techniques of comparable generality to handle the parameterisation by N. We have to resort to *ad hoc* (i.e., application specific) arguments. For either of (6.8) and (6.10), it can be shown without too much difficulty that it is independent of which nonzero natural number K is substituted for N, by translating behaviours for smaller K to behaviours for larger K in which there may be more accesses to addresses other than a. For either of (6.9) and (6.11), it is straightforward to see that it also is independent of K. (The details of the arguments are again outside the scope of this thesis.)

Therefore, showing (6.8)–(6.11) for $K = 1$ implies that the cache is correct for any finite nonempty sets with flat orderings assigned to the types A of addresses and V of values, and any nonzero natural number assigned to the capacity N. Subject to the remarks about CSP$_M$ at the beginning of Section 5.6, those four refinements can be checked by an existing model checker.

6.3 Notes

This chapter is based on [Ros98a, Section 15.2.3], which is in turn based on [LR96b, Section 12].

We do not claim any considerable originality: the purpose of this case study was to demonstrate how the theorems from Chapter 5 can be applied to an interesting and nontrivial example. In particular, a similar case study is presented in [HM+95] (which is Chapter 9 of [Hoj96]), although there are substantial differences in frameworks.

More generally, verifying cache coherence has been an active theoretical and practical subject: see e.g. [ID93a, ID93b, ID96a, Ip96, Gra94, PS96]. Further references can in particular be found in [Ip96].

In Section 5.5.3, we remarked that refinements between processes which can use arrays whose indices and values come from two type variables will be mentioned in Section 7.2.1 as an item for future work. In particular, we hope to obtain a theorem which is applicable to (6.5) and (6.6) in Section 6.2 and which provides threshold collections with finite sets, since the functions in the processes $\mathit{Memory}(f)$ and $\mathit{MemoryRen}(f)$ can be seen as arrays with indices from A and values from V. In other words, we hope to obtain a theorem which in this case study makes it unnecessary to revise the correctness condition using the processes $\mathit{OneLoc}(a, v)$ and $\mathit{OneLocRen}(a, v)$. From another point of view, (6.5) and (6.6) in Section 6.2 provide examples in which suprema of how many values of type A ever have to be stored for future use is infinite due to unboundedness of how many different array components can be accessed during execution, and where it is nevertheless possible to obtain thresholds with finite sets.

Chapter 7

Conclusions and future work

We shall now reflect on the contributions and applicability of the work which has been presented, and point out a number of directions for future work.

7.1 Conclusions

We can conclude that the decision problem stated in Section 1.2 has been successfully addressed.

More precisely, a study of data independence in terms of operational semantics and denotational semantics has been performed, and applied in proving a number of theorems which provide threshold collections for the problem of whether refinement between two given weakly data independent processes holds (see Chapter 5). We have also proved theorems which can be used to give more information if such a refinement fails for some interpretation from a threshold collection.

Subject to the remarks about CSP$_M$ in Section 3.8.1 and at the beginning of Section 5.6, the obtained theorems can be applied in practical model checking, using the model checkers for CSP (see Section 2.1.4). Based on the research presented in this thesis, we have also obtained a number of theorems which are stated directly in terms of CSP$_M$ (see [Ros98a, Section 15.2.2], [LR98], [Laz99]). Thus, the thesis indeed facilitates practical verification of data independent (and some weakly data independent) concurrent systems.

A number of nontrivial case studies have already been carried out. We have presented one of those, namely a verification of a cache (see Chapter 6), which is particularly interesting because it involves two different data independent types. Another case study, a verification of a distributed database, which in fact was the starting point for our work (see the remarks about the technical report [RM94] in Section 5.7), can be found in [Ros98a, Section 15.2.1]. A third case study, verifying that a database system which allows users to lock, read and write records at multiple security levels is secure, was mentioned in Section 5.5.3.

Some substantial further developments, including theorems which provide threshold collections for verifying determinism, an extension of data independence allowing arrays whose indices and values come from two different type variables, proving security protocols with model checkers, and tackling parameterisation by network size/structure, were mentioned in Section 5.5. The last of these is particularly interesting because it shows that the work presented in this thesis, although directly concerned only with data type parameters, can also be applied to a different kind of parameterisation. Together with the directions pointed out in Section 7.2, these further developments indicate that the thesis is a solid basis for much further research.

From a more technical point of view, the main contributions of the thesis are the following:

- combining the process algebra CSP [Hoa85, Ros98a] with a typed λ-calculus which has inductive types [Mit90, GLT89, Loa97, Bir98] to obtain a simple, formal and expressive language in which data independent types are the same as free type variables (see Chapter 2, in particular Section 2.7);
- presenting and studying a symbolic operational semantics which addresses a number of distinguishing features of the language (such as the wealth of operators, the nondirectedness of channels, the many-way synchronisation, and the presence of higher types: see Section 2.3), and which is particularly useful for reasoning about data independence and weak data independence (see Sections 3.1–3.3);
- presenting and studying symbolic normalisation (see Sections 3.6 and 3.7);
- defining logical relations at process types of our language and showing that a Basic Lemma holds for that definition (see Chapter 4);
- proving a number of theorems that facilitate verification of (possibly weakly) data independent processes by model checking (see Chapter 5), where the theorems are either distinguished mainly by addressing features of our framework which are not present in other frameworks (see Section 2.3), or we are not aware of comparable results in the literature.

Much more detailed and specific discussions of the relations to work by other authors can be found in the 'Notes' sections, at the ends of each of the Chapters 2–7.

(One year of my D.Phil. studies was spent working on *non-well-founded sets* [LR96a]. Their significance lies in the Special Final Coalgebra Theorem [Acz88], which states that final coalgebras of appropriate functors on sets can be obtained as greatest fixed points. In particular, non-well-founded sets enable direct set-theoretic constructions of operational models of concurrency (which is especially significant in the presence of *unbounded nondeterminism*), and they have interesting applications to situation theory and situation semantics [Acz88]. Due to limitations on space and to little relevance to the topic of the thesis, this work was not presented here.)

7.2 Future work

We suggest four main directions for future work.

7.2.1 Theory of data independence

We will continue the existing programme of work devoted to extending the range of specifications and implementations that admit proof by data independence techniques. At present we imagine that the main vehicles for this will still be symbolic operational semantics, logical relations and symmetry elimination. The outputs of this research will be additional conditions under which decision problems such as the one stated in Section 1.2 either become decidable, or there exist algorithms that can usefully be applied to them (but that do not always terminate).

Some of the topics that we currently have are given below. We also anticipate a continuing flow of ideas from users of the model checkers for CSP (see Section 2.1.4).

1. We intend to fully incorporate many-valued predicates (see Section 5.5.3) into the theory, so that we can prove theorems which provide threshold collections for refinements involving them. This should be relatively straightforward, and we expect to produce a variety of theorems depending, for example, on whether arbitrary equality tests are allowed or not.

2. As mentioned in Section 5.5.3, we have obtained a theorem which provides threshold collections for determinism checking of processes that can include arrays of one type variable Y indexed by another type variable X, provided the control flow is never affected by values from Y. We expect to be able to allow such arrays in refinement checks, but anticipate that this will be significantly harder than for many-valued predicates. In connection with this, see Section 6.3.

3. For any type context Δ all of whose types are X, let $(\mathbf{NoEqT}_{X,\Delta})$ and $(\mathbf{PosConjEqT}_{X,\Delta})$ denote weakenings of the conditions $(\mathbf{NoEqT}_X)$ and $(\mathbf{PosConjEqT}_X)$ (see Section 3.5.1) by allowing arbitrary equality tests in which at least one of the terms being tested is a variable from Δ. Thus $(\mathbf{NoEqT}_{X,\{\}})$ and $(\mathbf{PosConjEqT}_{X,\{\}})$ are equivalent to $(\mathbf{NoEqT}_X)$ and $(\mathbf{PosConjEqT}_X)$ respectively.

   We conjecture that it is possible to generalise Theorems 5.2.2, 5.3.5 and 5.3.6 to use $(\mathbf{NoEqT}_{X,\Delta})$ and $(\mathbf{PosConjEqT}_{X,\Delta})$ instead of $(\mathbf{NoEqT}_X)$ and $(\mathbf{PosConjEqT}_X)$, where $\Delta \subseteq \Gamma$. In Theorem 5.2.2, this seems to require having a lower bound of $|\eta[\![\Delta]\!]| + 1$ for κ. In Theorem 5.3.5, it seems convenient to restrict to interpretations of Δ which satisfy a given boolean condition γ on Δ (see under 'Boolean conditions on constants' in Section 5.4.3 and the corresponding remarks near the beginning of Section 5.4), to instead of the type *Two* use types which have exactly two more elements than necessary for interpreting the constants from Δ, and to generalise the special processes *OneOne* and *Zeros* so that they restrict only one of such two additional elements. In Theorem 5.3.6, it seems convenient to also restrict to interpretations of Δ which satisfy a given boolean condition γ on Δ, and to interpret X by sets which have exactly two more elements than necessary for interpreting the constants from Δ.

   In order to prove such generalised theorems, Theorem 4.2.2 would also need to be generalised to use $(\mathbf{NoEqT}_{X,\Delta})$ and $(\mathbf{PosConjEqT}_{X,\Delta})$ respectively instead of $(\mathbf{NoEqT}_X)$ and $(\mathbf{PosConjEqT}_X)$, which seems possible.

   The weakened conditions $(\mathbf{NoEqT}_{X,\Delta})$ and $(\mathbf{PosConjEqT}_{X,\Delta})$ are important because processes which satisfy them seem to occur relatively frequently in practice. For example, a condition which is closely related to $(\mathbf{PosConjEqT}_{X,\Delta})$ has played a key role in the work on proving security protocols with model checkers that was mentioned in Section 5.5.4.

4. We have conjectures that the lower bounds for sizes of sets in the third theorem mentioned in Section 5.5.2 and in the theorem mentioned in Section 5.5.3 can be nearly halved. This would often result in exponential increases of efficiency of the corresponding checks, and it would thus substantially increase the range of problems in the area of security as non-interference (see e.g. [Ros98a, Section 12.4]) that could be handled practically.

5. Given the assumptions about *Spec* in Theorem 5.1.2, we can at present allow limited reasoning about weakly data independent processes which are not data independent (see Sections 2.7.2 and 2.7.1).

We would like to extend the range of specifications that can be proved of such processes, through the use of symbolic operational semantics, potentially automated (see Section 3.8). We anticipate that it may be hard to tell *a priori* between those checks for which termination will occur and ones for which it will not. Depending on how far this idea were taken, automation would probably require term rewriting (see e.g. [RTA]) on data values and might require integration with theorem proving (see Section 1.1.2).

A number of very interesting results in this area were presented in [HB95, HI+96] (on which Chapter 8 of [Hoj96] is based). In [HI+96], uninterpreted function symbols on variable types were considered, where 'uninterpreted' means that the specification has to be satisfied for all interpretations of the function symbols. These have applications in the verification of superscalar microprocessors, i.e. microprocessors that attempt to execute more than one instruction at a time. It was established that the problem was undecidable if arbitrary equality tests are also allowed, but for a restricted version, a theorem that provides thresholds was proved. For less restricted versions, finite sizes of the type and interpretations of the function symbols that are guaranteed to catch any bug that is not of a particular 'rare' form were suggested.

Combinations of BDDs (see e.g. [Bry86, BC+92, McM93]) and symbolic execution in the presence of uninterpreted function symbols whose types do not necessarily obey the restrictions of weak data independence were developed in [HI+96, CZ+97].

For systems which are not data independent, verification using symbolic execution naturally requires the help of theorem proving, and many authors have taken steps in this direction. For example, [HL95a, Sch94, Lin96, Pac96] are concerned with constructing a logic formula (or a more complex object) which is valid if and only if the two given symbolic labelled transition systems are bisimilar; one can then attempt to determine the validity of the formula using a theorem prover.

6. It would be extremely useful for many applications to allow orderings on variable types, or perhaps more general binary or higher order relations. We cannot expect theorems that provide threshold collections in every case here. For example, a process that continues running provided that a series of values of type X increase strictly can do so forever only when X is interpreted by an infinite set. However, we can hope to get partially effective techniques from symbolic operational semantics, or perhaps for some syntactically restricted class of processes.

7. In the work on proving security protocols with model checkers, which was mentioned in Section 5.5.4, logical relations were applied in an interesting and very direct way, in manipulating a process which requires an infinite lower bound for threshold collections to one which has a small finite lower bound. We hope these ideas will generalise to applications beyond security.

8. We intend to continue to seek new theoretical directions which can aid the general aims of the research (see Sections 1.1 and 1.2). These might well involve further analysis of semantic models of our language and related languages (see Sections 2.8 and 4.3). We anticipate there may be fruitful connections with decidability of fragments of 1^{st}-order logic, especially those which have the 'small model property' (see e.g. [BGG96, Fag93]).

7.2.2 Links with other approaches to the PVP

One of the most common versions of the Parameterised Verification Problem (see Section 1.1.3) one encounters is proving properties of arbitrary-sized parallel networks. Data independence, by itself, does not handle such problems because it only applies to fixed topology systems (see under 'About CSP constructs not in our language' near the end of Section 2.5.2). The most common technique for addressing variable-sized networks is *induction*: one formulates an appropriate property that is true of an increasing family of subnetworks (see e.g. [WL89, KM89, BCG89, LHR97, RJ+98, Cre97, Cre98, CR99a]).

In Section 5.5.5, we mentioned the research on combining the work presented in this thesis with induction, which has produced techniques for proving with model checkers systems that are parameterised by network size/structure. We intend to develop this approach further and seek more applications for it.

Some other common versions of the PVP arise where a system is parameterised by capacities of buffers, or by limits on sizes of dynamic data structures such as queues, stacks or trees. Methods for addressing such versions of the PVP have been presented in the literature. For example, QDDs which can symbolically (in the sense of BDDs: see e.g. [Bry86, BC+92, McM93]) represent sets of queue contents were proposed in [BG96]; see also [GL96] which is a related paper. In addition, the subject of *buffer tolerance* (see e.g. [Ros98a, Section 5.2]) is relevant here.

We intend to investigate this area, which is practically important partly thanks to the success of SDL in becoming widely used in telecommunications applications. In particular, it remains to be seen to what extent the work presented in this thesis will be applicable to it.

7.2.3 Broadening beyond our language

As we have remarked in Section 5.7 in particular, data independence is certainly not unique to our language (and to CSP_M). We hope to define a framework under which theorems such as those we have presented or mentioned in Chapter 5 and related techniques can be moved as simply as possible from language to language.

One route to this is via symbolic operational semantics (see Chapter 3). Since the theorems in Chapter 5 could be derived from corresponding theorems about SLTSs, ASLTSs and NSLTSs, there is every likelihood that similar results can be obtained for other languages, hopefully systematically. The languages we hope to consider may be of immediate practical importance, such as SDL, or be further theoretical tools such as the π-calculus.

A distinguishing feature of our framework is that specifications are cast in the same notation as implementations (see Section 2.3). In transferring results to other notations it will therefore be necessary to consider specification languages such as temporal logics (see e.g. [Eme90]). We foresee few particular difficulties with this, at least in so far as they can be modelled via state machines / labelled transition systems.

Theorems for verifying that data independent systems satisfy specifications in linear temporal logic were presented in [Wol86, HDB97].

7.2.4 Tools

As we have indicated in Sections 1.1 and 1.2, the research presented in this thesis has been motivated by immediate needs of practical model checking, in particular using the model checkers for CSP (see Section 2.1.4). In Sections 3.8 and 5.6, we have specifically discussed how the results obtained can be applied, ways of increasing the degree to which such applications are automatic, and other possibilities for automation based on the presented work.

We intend to continue to keep a close relationship with tools. In particular, we shall work on specifying and implementing tools based on theoretical work by us and by other authors. As can be seen from Sections 3.8 and 5.6, a considerable proportion of such work is likely to concentrate on extending existing tools.[1]

7.3 Notes

A three-year project, which is based on the work presented in this thesis, and on the further developments by other authors which were mentioned in Sections 5.5.4 and 5.5.5, will begin by October 1999, funded by the EPSRC. Section 7.2 is based on its proposal, which was produced by A.W. Roscoe and myself.

As we remarked in Section 3.8.1, a substantial project whose aim is automation based on this work, will also begin in 1999.

At the time of writing (April 1999), a project on computer security involving Oxford University and Tulane University is being set up. We expect close contact between this project and our continued work on data independence, since as indicated in Sections 5.5.2, 5.5.3 and 5.5.4, these two areas have close links.

An interest in this work was also expressed by Motorola, in the context of SDL and telecommunications applications.

[1]Most implementation work will probably be performed at Formal Systems (Europe) Ltd, who are the developers of the model checker FDR [Ros94a, Ros98a, FS97], and at Adelaide University (Australia), where the model checker ARC [Yan96] was developed.

Bibliography

[Acz88] Aczel, P., *Non-well-founded Sets*, CSLI Lecture Notes 14, 1988.

[AGM94] Abramsky, S., D.M. Gabbay and T.S.F. Maibaum (editors), *Handbook of Logic in Computer Science*, Clarendon Press, 1994.

[AJ94] Abramsky, S. and A. Jung, *Domain Theory*, in [AGM94], Volume 3, 1–168.

[AK86] Apt, K.R. and D.C. Kozen, *Limits for Automatic Verification of Finite-state Concurrent Systems*, Information Processing Letters 22 (6), 307–309, 1986.

[BC+92] Burch, J.R., E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang, *Symbolic Model Checking:* 10^{20} *States and Beyond*, Information and Computation 98 (2), 142–170, 1992.

[BCG89] Browne, M., E.M. Clarke and O. Grumberg, *Reasoning About Networks with Many Identical Finite State Processes*, Information and Computation 81 (1), 13–31, 1989.

[BG96] Boigelot, B. and P. Godefroid, *Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs*, Proceedings of the 8th CAV, Springer LNCS 1102, 1996.

[BGG96] Börger, Grädel and Gurevich, *The Classical Decision Problem*, Springer, 1996.

[Bir98] Bird, R., *Introduction to Functional Programming using Haskell*, second edition, Prentice Hall, 1998.

[BJ78] Brand, D. and W.H. Joyner, *Verification of Protocols Using Symbolic Execution*, Comput. Networks 2 (4–5), 351–360, 1978.

[BK83] Back, R.J.R. and R. Kurki-Suonio, *Decentralisation of Process Nets With Centralised Control*, Proceedings of the 2nd ACM PODC, 131–142, 1983.

[Bro94] Brookes, S.D., *Fair Communicating Processes*, in [Ros94b], 59–74.

[Bro96] Brookes, S.D., *The Essence of Parallel Algol*, Proceedings of the 11th IEEE LICS, 164–173, 1996.

The full version to appear in Information and Computation.

[Bro97] Brookes, S.D., *Idealized CSP: Combining Procedures With Communicating Processes*, Proceedings of the 13th MFPS, Electronic Notes in Theoretical Computer Science 6, 1997.

`http://www.elsevier.nl/locate/entcs/volume6.html`

[Bro98] Broadfoot, P.J., *Application of Data Independence Techniques to Security Protocols*, M.Sc. dissertation, Oxford University Computing Laboratory, 1998.

[Bry86] Bryant, R.E., *Graph-based Algorithms for Boolean Function Manipulation*, IEEE Trans. Comput. C-35 (8), 1986.

[But96] Butler, M.J., *Stepwise Refinement of Communicating Systems*, Science of Computer Programming 27 (2), 139–173, 1996.

[CADE] *Annual Proceedings of the Computer-aided Deduction (CADE) Conference*, Springer LNAI.

[CAV] *Annual Proceedings of the Computer-aided Verification (CAV) Conference*, Springer LNCS.

[CBK90] Clarke, E.M., I.A. Browne and R.P. Kurshan, *A Unified Approach for Showing Language Containment and Equivalence Between Various Types of ω-automata*, Proceedings of the 15th CAAP, Springer LNCS 431, 103–116, 1990.

[CC77] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, Proceedings of the 4th ACM POPL, 1977.

[CE+96] Clarke, E.M., R. Enders, T. Filkorn and S. Jha, *Exploiting Symmetry in Temporal Logic Model Checking*, in [Eme96], 77–104.

[CGL94] Clarke, E.M., O. Grumberg and D.E. Long, *Model Checking and Abstraction*, ACM Transactions on Programming Languages and Systems 16 (5), 1512–1542, 1994.

[Cre97] Creese, S.J., *An Inductive Technique for FDR*, M.Sc. dissertation, Oxford University Computing Laboratory, 1997.

[Cre98] Creese, S.J., *Techniques for the Application of CSP Methods to Practical, Scalable Systems*, transfer thesis, Oxford University Computing Laboratory, 1998.

[CR99a] Creese, S.J. and J.N. Reed, *Verifying End-to-End Protocols Using Induction with CSP/FDR*, Proceedings of FMPPTA, 1999.

[CR99b] Creese, S.J. and A.W. Roscoe, *Verifying an Infinite Family of Inductions Simultaneously Using Data Independence and FDR*, submitted to FORTE/PSTV, 1999.

Available from:
`http://www.comlab.ox.ac.uk/oucl/publications/`
`books/concurrency/textmat/`

[Cut80] Cutland, N., *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, 1980.

[CZ+97] Corella, F., Z. Zhou, X. Song, M. Langevin and E. Cerny, *Multiway Decision Graphs for Automated Hardware Verification*, Formal Methods in System Design 10 (1), 7–46, Kluwer, 1997.

[Dam96] Dams, D., *Abstract Interpretation and Partial Refinement for Model Checking*, Ph.D. thesis, Faculteit der Wiskunde en Informatica, Technische Universiteit Eindhoven, 1996.

[DF95] Dingel, J. and T. Filkorn, *Model Checking for Infinite-State Systems Using Data Abstraction, Assumption-Commitment Style Reasoning and Theorem Proving*, Proceedings of the 7th CAV, Springer LNCS 939, 54–69, 1995.

[DP90] Davey, B.A. and H.A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[DV90] De Nicola, R. and F. Vaandrager, *Action versus State based Logics for Transition Systems*, Semantics of Systems of Concurrent Processes, Springer LNCS 469, 407–419, 1990.

[Eme90] Emerson, E.A., *Temporal and Modal Logic*, in [van90], 995–1073.

[Eme96] Emerson, E.A. (editor), Formal Methods in System Design 9 (1–2) (Special Issue on Symmetry in Automatic Verification), Kluwer, 1996.

[End77] Enderton, H.B., *Elements of Set Theory*, Academic Press, 1977.

[ES95] Emerson, E.A. and A.P. Sistla, *Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach*, Proceedings of the 7th CAV, Springer LNCS 939, 309–324, 1995.

[ES96] Emerson, E.A. and A.P. Sistla, *Symmetry and Model Checking*, in [Eme96], 105–131.

[Fag93] Fagin, R., *Finite-model Theory — A Personal Perspective*, Theoretical Computer Science 166 (1), 3–31, 1993.

[FJ+96] Fiore, M.P., A. Jung, E. Moggi, P. O'Hearn, J. Reicke, G. Rosolini and I. Stark, *Domains and Denotational Semantics: History, Accomplishments and Open Problems*, Bulletin of the EATCS 59, 227–256, 1996.

[Fra86] Francez, N., *Fairness*, Springer, 1986.

[FS97] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, 1997.

[GL96] Godefroid, P. and D.E. Long, *Symbolic Protocol Verification with Queue BDDs*, Proceedings of the 11th IEEE LICS, 198–206, 1996.

[GLT89] Girard, J.-Y., Y. Lafont and P. Taylor, *Proofs and Types*, Cambridge University Press, 1989.

[Gra94] Graf, S., *Verification of a Distributed Cache Memory by Using Abstractions*, Proceedings of the 6th CAV, Springer LNCS 818, 207–219, 1994.

[HB95] Hojati, R. and R.K. Brayton, *Automatic Datapath Abstraction in Hardware Systems*, Proceedings of the 7th CAV, Springer LNCS 939, 98–113, 1995.

[HDB97] Hojati, R., D.L. Dill and R.K. Brayton, *Verifying Linear Temporal Properties of Data Insensitive Controllers Using Finite Instantiations*, Proceedings of the 13th CHDL, 1997.

[Hen96] Henzinger, T.A., *The Theory of Hybrid Automata*, Proceedings of the 11th IEEE LICS, 278–292, 1996.

[HGD95] Hungar, H., O. Grumberg and W. Damm, *What if Model Checking Must be Truly Symbolic?*, Proceedings of the 1st TACAS, BRICS Notes Series NS-95-2, 230–244, Department of Computer Science, University of Aarhus, 1995.

Also in Springer LNCS 1019.

[HI+96] Hojati, R., A. Isles, D. Kirkpatrick and R.K. Brayton, *Verification Using Uninterpreted Functions and Finite Instantiations*, Proceedings of FMCAD, Springer LNCS 1166, 1996.

[HL95a] Hennessy, M. and H. Lin, *Symbolic Bisimulations*, Theoretical Computer Science 138 (2), 353–389, 1995.

[HL95b] Hennessy, M. and X. Liu, *A Modal Logic for Message Passing Processes*, Acta Informatica 32 (4), 375–393, 1995.

[HM+95] Hojati, R., R. Mueller-Thuns, P. Lowenstein and R.K. Brayton, *Automatic Verification of Memory Systems which Execute Their Instructions Out of Order*, Proceedings of the 12th CHDL, 1995.

[Hoa85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.

[Hoj96] Hojati, R., Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1996(?).

[ID93a] Ip, C.N. and D.L. Dill, *Better Verification Through Symmetry*, Proceedings of the 11th CHDL, 1993.

[ID93b] Ip, C.N. and D.L. Dill, *Efficient Verification of Symmetric Concurrent Systems*, Proceedings of the International Conference of Computer Design: VLSI in Computers and Processors, 1993.

[ID96a] Ip, C.N. and D.L. Dill, *Better Verification Through Symmetry*, in [Eme96], 41–75.

[ID96b] Ip, C.N. and D.L. Dill, *Verifying Systems with Replicated Components in Murφ*, Proceedings of the 8th CAV, Springer LNCS 1102, 147–158, 1996.

[INF96] Proceedings of the 1st INFINITY, Electronic Notes in Theoretical Computer Science 5, 1996.

`http://www.elsevier.nl/locate/entcs/volume5.html`

[INF97] Proceedings of the 2nd INFINITY, Electronic Notes in Theoretical Computer Science 9, 1997.

`http://www.elsevier.nl/locate/entcs/volume9.html`

[INF98] Esparza, J. (editor), Informal proceedings of the 3rd INFINITY (1998), to appear as a Research Report of the Technical University of Munich.

[Inm88] Inmos Ltd, *Occam 2 Reference Manual*, Prentice Hall, 1988.

[Ip96] Ip, C.N., *State Reduction Methods for Automatic Formal Verification*, Ph.D. thesis, Department of Computer Science, Stanford University, 1996.

[IP96] Inverardi, P. and C. Priami, *Automatic Verification of Distributed Systems: The Process Algebra Approach*, Formal Methods in System Design 8 (1), 7–38, Kluwer, 1996.

[Jen96] Jensen, K., *Condensed State Spaces for Symmetrical Coloured Petri Nets*, in [Eme96], 7–40.

[JG88] Jones, G. and M. Goldsmith, *Programming in Occam 2*, Prentice Hall, 1988.

[JP93] Jonsson, B. and J. Parrow, *Deciding Bisimulation Equivalences for a Class of Non-finite-state Programs*, Information and Computation 107 (2), 272–302, 1993.

[Kai96] Kaivola, R., *Equivalences, Preorders and Compositional Verification for Linear Time Temporal Logic and Concurrent Systems*, Ph.D. thesis, Report A-1996-1, Department of Computer Science, University of Helsinki, 1996.

[KM89] Kurshan, R.P. and K.L. McMillan, *A Structural Induction Theorem for Processes*, Proceedings of the ACM Symposium on Principles of Distributed Computing, 1989.

[KO+97] Kinoshita, Y., P.W. O'Hearn, A.J. Power, M. Takeyama and R.D. Tennent, *An Axiomatic Approach to Binary Logical Relations with Applications to Data Refinement*, Proceedings of TACS, Springer LNCS 1281, 1997.

[Kok97] Kokkarinen, I., *Ω-algebraic Data in Labelled Transition Systems*, Proceedings of the 5th Symposium on Programming Languages and Software Tools, Publication C-1997-37, Department of Computer Science, University of Helsinki, 1997.

[Kok98] Kokkarinen, I., Ph.D. thesis, Department of Computer Science, Tampere University of Technology, 1998.

[Kun80] Kunen, K., *Set Theory: An Introduction to Independence Proofs*, North-Holland, 1980.

[Kur90] Kurshan, R.P., *Analysis of Discrete Event Coordination*, Stepwise Refinement of Distributed Systems, Springer LNCS 430, 414–453, 1990.

[Laz97] Lazić, R.S., *A Semantic Study of Data Independence with Applications to Model Checking*, a dissertation which won the first place in the 1997 Senior Mathematical Prize competition (an annual competition for Oxford University graduate students).

[Laz98] Lazić, R.S., *Some Literature on Data Independence and Parameterised Verification*, an informal literature survey, 1998.

Available from:
`http://www.formal.demon.co.uk/workshop/199803`

[Laz99] Lazić, R.S., *Theorems for Model Checking Data Independent CSP*, Technical Report, Oxford University Computing Laboratory, to appear in 1999.

`http://www.comlab.ox.ac.uk/oucl/publications/tr.html`

[LG+95] Loiseaux, C., S. Graf, J. Sifakis, A. Bouajjani and S. Bensalem, *Property Preserving Abstractions for the Verification of Concurrent Systems*, Formal Methods in System Design 6, 11–44, Kluwer, 1995.

[LHR97] Lesens, D., N. Halbwachs and P. Raymond, *Automatic Verification of Parameterised Linear Networks of Processes*, Proceedings of the 24th ACM POPL, 1997.

[Lin96] Lin, H., *Symbolic Transition Graphs with Assignment*, Proceedings of the 7th CONCUR, Springer LNCS 1119, 1996.

[Loa97] Loader, R., *Equational Theories for Inductive Types*, Annals of Pure and Applied Logic 84, 175–217, 1997.

[Low96] Lowe, G., *Model Checking Systems of Arbitrary Size*, a draft paper, November 1996.

[LR96a] Lazić, R.S. and A.W. Roscoe, *On Transition Systems and Non-well-founded Sets*, in 'Papers on General Topology and Applications: Eleventh Summer Conference at the University of Southern Maine (1995)', Annals of the New York Academy of Sciences 806, 238–264, 1996.

[LR96b] Lazić, R.S. and A.W. Roscoe, *Using Logical Relations for Automated Verification of Data Independent CSP*, Proceedings of the Workshop on Automated Formal Methods (Oxford, UK, June 1996), to appear in Electronic Notes in Theoretical Computer Science.

[LR98] Lazić, R.S. and A.W. Roscoe, *Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays*, in [INF98].

Also available from:
`http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/examples/security/`

A fuller version is the Technical Report PRG-TR-2-98, Oxford University Computing Laboratory.

`http://www.comlab.ox.ac.uk/oucl/publications/tr.html`

[McM93] McMillan, K.L., *Symbolic Model Checking*, Kluwer, 1993.

[Mit90] Mitchell, J.C., *Type Systems for Programming Languages*, in [van90], Volume B: Formal Models and Semantics, 365–458.

[OHe96] O'Hearn, P., *Parametric Polymorphism*, in [FJ+96].

[Ong93] Ong, C.-H.L., *Non-determinism in a Functional Setting*, Proceedings of the 8th IEEE LICS, 275–286, 1993.

[Pac96] Paczkowski, P., *Characterising Bisimilarity of Value-passing Parameterised Processes*, in [INF96].

[Per90] Perrin, D., *Finite Automata*, in [van90], Volume B: Formal Models and Semantics, 1–57.

[PGM90] Prasad, S., A. Giacalone and P. Mishra, *Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming*, Proceedings of ICALP '90, Springer LNCS 443, 765–780, 1990.

[PL90] Probst, D.K. and H.F. Li, *Using Partial Order Semantics to Avoid the State Explosion Problem in Asynchronous Systems*, Proceedings of the 2nd CAV, Springer LNCS 531, 1990.

[Plo80] Plotkin, G.D., *Lambda-definability in the Full Type Hierarchy*, in 'To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism' (J.P. Seldin and J.R. Hindley, editors), 363–373, Academic Press, 1980.

[PS96] Pnueli, A. and E. Shahar, *A Platform for Combining Deductive with Algorithmic Verification*, Proceedings of the 8th CAV, Springer LNCS 1102, 184–195, 1996.

[Rat97] Rathke, J., *Symbolic Techniques for Value-passing Calculi*, D.Phil. thesis, University of Sussex, 1997.

[RB98] Roscoe, A.W. and P.J. Broadfoot, *Proving Security Protocols with Model Checkers by Data Independence Techniques*, submitted for publication, 1998.

Available from:
`http://www.comlab.ox.ac.uk/oucl/publications/`
`books/concurrency/textmat/`

[Rey83] Reynolds, J.C., *Types, Abstraction and Parametric Polymorphism*, Information Processing '83, 513–523, North-Holland, 1983.

[RG+95] Roscoe, A.W., P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson and J.B. Scattergood, *Hierarchical Compression for Model Checking CSP* or *How to Check* 10^{20} *Dining Philosophers for Deadlock*, Proceedings of the 1st TACAS, BRICS Notes Series NS-95-2, 187–200, Department of Computer Science, University of Aarhus, 1995.

Also in Springer LNCS 1019.

[RJ+98] Reed, J.N., D.M. Jackson, B. Deianov and G.M. Reed, *Automated Formal Analysis of Networks: FDR Models of Arbitrary Topologies and Flow-Control Mechanisms*, Proceedings of the European Joint Conference on Theory and Practice of Software; Fundamental Approaches to Software Engineering (ETAPS-FASE), Lisbon, Portugal, March 1998.

[RM94] Roscoe, A.W. and H. MacCarthy, *Verifying a Replicated Database: A Case Study in Model Checking CSP*, Technical Report, Formal Systems (Europe) Ltd, 1994.

[Ros88] Roscoe, A.W., *An Alternative Order for the Failures Model*, in 'Two Papers on CSP', Technical Monograph PRG-67, Oxford University Computing Laboratory, July 1988.

Also in the Journal of Logic and Computation 2 (5), 557–577, 1992.

[Ros94a] Roscoe, A.W., *Model Checking CSP*, in [Ros94b], 353–378.

[Ros94b] Roscoe, A.W. (editor), *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice Hall, 1994.

[Ros98a] Roscoe, A.W., *The Theory and Practice of Concurrency*, Prentice Hall, 1998.

The associated web site is:
`http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/`

[Ros98b] Roscoe, A.W., *Proving Security Protocols with Model Checkers by Data Independence Techniques*, Proceedings of the 11th IEEE CSFW, 1998.

Also available from:
`http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/examples/security/`

[RR88] Reed, G.M. and A.W. Roscoe, *A Timed Model for Communicating Sequential Processes*, Theoretical Computer Science 58, 249–261, 1988.

[RTA] *Annual Proceedings of the Rewriting Techniques and Applications (RTA) Conference*, Springer LNCS.

[Sab88] Sabnani, K., *An Algorithmic Technique for Protocol Verification*, 924–931, IEEE Transactions on Communications 36 (8), 1988.

[SB95] Seger, C.-J.H. and R.E. Bryant, *Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories*, Formal Methods in System Design 6 (2), 147–190, Kluwer, 1995.

[Sca92] Scattergood, J.B., *Parsing and Type-checking CSP*, transfer thesis, Oxford University Computing Laboratory, 1992.

[Sca98] Scattergood, J.B., *Tools for CSP and Timed CSP*, D.Phil. thesis, Oxford University Computing Laboratory, 1998.

[Sch94] Schreiber, M.Z., *Value-passing Process Calculi as a Formal Method*, Ph.D. thesis, Imperial College, London, 1994.

[Sch98] Schneider, S.A., *Concurrency and Time*, a draft book, 1998.

[ST97] Shankland, C. and M. Thomas, *Symbolic Bisimulation for Full LOTOS*, Proceedings of AMAST, Springer LNCS 1349, 479–493, 1997.

[Sta91] Starke, P.H., *Reachability Analysis of Petri Nets Using Symmetries*, Systems Analysis - Modelling - Simulation 8 (4–5), 293–303, 1991.

[SWL85] Sabnani, K., P. Wolper and A. Lapone, *An Algorithmic Procedure for Protocol Verification*, Proceedings of GLOBECOM, 1985.

[Tho90] Thomas, W., *Automata on Infinite Objects*, in [van90], Volume B: Formal Models and Semantics, 133–191.

[Val89] Valmari, A., *Stubborn Sets for Reduced State Space Generation*, Proceedings of the 10th International Conference on Theory and Applications of Petri Nets, 1989.

[van90] van Leeuwen, J. (editor), *Handbook of Theoretical Computer Science*, Elsevier, 1990.

[Wad89] Wadler, P., *Theorems for Free!*, Proceedings of the 4th ACM FPLCA, 347–359, 1989.

[WL89] Wolper, P. and V. Lovinfosse, *Verifying Properties of Large Sets of Processes with Network Invariants*, Proceedings of the International Workshop on Automatic Verification Methods for Finite-state Systems, Springer LNCS 407, 68–80, 1989.

[Wol86] Wolper, P., *Expressing Interesting Properties of Programs in Propositional Temporal Logic*, Proceedings of the 13th ACM POPL, 184–193, 1986.

[Yan96] Yantchev, J.T., *ARC — A Tool for Efficient Refinement and Equivalence Checking for CSP*, Proceedings of the 2nd IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing, 1996.

List of notation

Some notation in connection with types

$[\![T]\!]_{\delta}^{M}$	20
Tr, Fa, FD	20
$[X_1 \mapsto n_1, \ldots, X_k \mapsto n_k]$	20
$[X_1 \mapsto \kappa_1, \ldots, X_k \mapsto \kappa_k]$	20
$\mathit{AllTypeVars}$	20
$\mathit{NEFTypeVars}$	21
$\mathit{FinTypeVars}$	21
$\phi(id_i, j)$	24
$T \sim T'$	28
$\mathit{Free}(T)$	28

Type constructs

$id_1.T_1 + \cdots + id_n.T_n$	21
$T_1 \times \cdots \times T_n$	22
$\mu\, X.T$	22
$T_1 \to T_2$	23
$\mathit{Proc}_{T,\phi}$	24

Conditions on types

$T^{+,\times}$	26
$T^{\mathit{AllTypeVars},+,\times}$	26
$T^{\mathit{AllTypeVars},+,\times,\mu}$	26
$T^{\mathit{AllTypeVars},+,\times,\to}$	26
$T^{\mathit{AllTypeVars},+,\times,\mu,\to}$	26
$T^{\mathit{nef}'}$	26
T^{nef}	26
T^{comm}	26
T^{alph}	26
T^{fin}	27
$T^{(+,\times,\mu)_X}$	27

Some example types

Some notation in connection with terms

λ-calculus term constructs

Denotational semantics of processes

CSP term constructs

$\mathit{Subst}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}$ 80
$\mathit{Match}^{?zs\,.\,H[r_i]}_{?zs'\,.\,H[r'_j]}$ 80

SLTSs, ASLTSs, NSLTSs, and their components

SLTS 82
ASLTS 85
NSLTS 114
$(\mathcal{N}, N_0, \mathcal{T}, \mathit{source}, \mathit{target})$ 82, 85, 114
$\mathit{Context}(N)$ 82, 85, 104
$\mathit{AccSets}(N)$ 104
$\mathit{DivCond}(N)$ 104
$N\alpha$ 83, 86, 114
$N(\alpha, i)$ 114
$\alpha N'$ 83, 86
$(\alpha, i)N'$ 114
$\mathcal{S}_{\Gamma \vdash P}$ 83
$\mathcal{AS}_{\Gamma \vdash P}$ 87
$\mathcal{NS}(\mathcal{S})$ 106
$\mathcal{NS}(\mathcal{S}_{\{\} \vdash P})$ 114
$\overline{\mathcal{S}}^{AS}$ 86
$\overline{\mathcal{AS}}^{S}$ 86

Quantities which measure SLTSs, ASLTSs and NSLTSs

$W(\mathcal{S})$, $W(\mathcal{AS})$ 88
$W(\mathcal{NS})$ 115
$L_\$(\mathcal{S})$, $L_\$(\mathcal{AS})$ 88
$L_?(\mathcal{S})$, $L_?(\mathcal{AS})$ 88
$L_\sqcap(\mathcal{S})$, $L_\sqcap(\mathcal{AS})$ 88
$L(\mathcal{NS})$ 115

Denotational semantic values corresponding to SLTSs, ASLTSs and NSLTSs

$N_1^{\eta_1} \alpha_1^{\eta'_1} N_2^{\eta_2} \alpha_2^{\eta'_2} \ldots$ 92
$N_1^{\eta_1} (\alpha_1, i_1)^{\eta'_1} N_2^{\eta_2} (\alpha_2, i_2)^{\eta'_2} \ldots$ 117
$N_1^{\eta_1} \stackrel{t}{\Longrightarrow} N_k^{\eta_k}$ *w.r.t.* δ 92, 118
$N'^{\eta'}$ ref^{η^*} A *w.r.t.* δ 93
$N'^{\eta'}$ $\mathit{ref}^{\mathcal{A}, \eta^*}$ A *w.r.t.* δ 118
$N'^{\eta'}$ *div* *w.r.t.* δ 93
$\mathit{traces}(N)_{\delta,\eta}$ 93, 118
$\mathit{failures}(N)_{\delta,\eta}$ 93, 119
$\mathit{failures}_\perp(N)_{\delta,\eta}$ 93, 119
$\mathit{divergences}(N)_{\delta,\eta}$ 93, 119